PascalABC.NET system

PascalABC.NET is the **next-generation Pascal programming system and language** for Microsoft's .NET platform. **PascalABC.**NET has all the main elements of modern programming languages: modules, classes, overloading, interfaces, exceptions, generalized classes, garbage collection, lambda expressions, and some tools for concurrency, including OpenMPdirectives. **PascalABC.NET** also includes a simple integrated environment focused on effective learning of modern programming.

- **PascalABC.NET**language description.
- The benefits**of PascalABC.NET** for program development and training.
- Differencesbetween**PascalABC.NET**andDelphi(Object Pascal).
- Examples to illustrate the main features **of PascalABC.NET** can be found in the Help/Summary menu.

The Pascal language was developed by the Swiss scientist Niklaus Wirth in 1970 as a language with strict typing and intuitive syntax. In the 80's, the most famous implementation was the Turbo Pascal compiler from Borland, in the 90'sit was replaced by the Delphi - programming environment, which became one of the best environments for the rapid creation of applications for Windows. Delphi has introduced a number of successful object-oriented extensions to the language of Pascal, the updated language was named Object Pascal. Since Delphi 7, the language Delphi Object Pascal became known simply as Delphi. Of alternative implementations of Object Pascal should be noted multi-platform open source compiler Free Pascal.

PascalABC.NET was created for two main reasons: the obsolescence of the standard Pascal language and systems based on it (Free Pascal); and the need for a modern, simple, free and powerful integrated programming environment.

PascalABC.NET relies on the advanced Microsoft.NET programming platform, which provides **PascalABC.NET** with a huge number of standard libraries and makes it easy to combine with other .NET languages: C#, Visual Basic.NET, Managed C++, Oxygene and others. The .NET platform also provides such linguistic tools as

a single mechanism for handling exceptions, a single mechanism for managing memory in the form of *garbage collection*, as well as the ability to freely use classes, inheritance, polymorphism and interfaces between modules written in different .NET-languages. You can read about what the Microsoft.NET platform is, its benefits for programming and for learning here.

The **PascalABC.NET** language is close to a Delphi (Object Pascal) implementation. It lacks some specific Delphi language constructsandchanges some constructs. It also has a number of new features: autodefinition of type when defining, variables can be defined inside a block, `+=`, `-=`, `*=`, `/=` operations are available, methods can be defined inside the class or record body, methods and properties built into standard types can be used, object memory is managed byGarbageCollector and doesn't require explicit freeing, **`set`** sets can be created based on arbitrary types, **`foreach`** statements are included, **`for`** and **`foreach`** loop variables can be defined directly in the loop header, generalized clauses are included, and there are many new features.

Close in ideology to **PascalABC.NET is** the language RemObjects Oxygene (21st century Object Pascal). However, it is strongly modified towards .NET: there are no global descriptions, all descriptions are placed in a class that contains a static method Main, there are no standard Pascal subroutines. In addition, RemObjects Oxygene is a paid system and does not contain its own shell (embedded in Visual Studio and other IDEs), which makes it almost impossible to use in education.

The integrated **PascalABC.NET** environment provides syntax highlighting, code hinting (point hinting, subroutine parameter hinting, code pop-up hinting), formatting of program text on demand, transition to name definition and implementation, refactoring elements.

A brief summary of the main points

This text provides a brief **overview of the features** of PascalABC.NET.

- PascalABC.NET is a lightweight and powerful programming environment with a detailed help system, code prompting, auto-formatting, built-in debugger and built-in form designer. The integrated PascalABC.NET development environment is geared towards creating projects of low to medium complexity, and to teach you about modern programming.
- PascalABC.NET is a powerful and modern programming language. It surpasses Delphi and contains almost all the features of C#.
- PascalABC.NET relies on the Microsoft .NET platform - its language capabilities and libraries - to make it flexible, efficient, and constantly evolving. You can also easily combine libraries developed in PascalABC.NET and other .NET languages.
- The PascalABC.NET compiler generates code that runs as fast as C# code and slightly slower than C++ and Delphi.
- PascalABC.NET is representative of the line of modern Pascal languages along with Delphi XE and Oxygene.
- The opinion that Pascal language is outdated and losing its positions is based on the idea of old Pascal and old programming environments (for example, Free Pascal with its outdated console shell and Delphi language of 2002 sample). Unfortunately, the mass of domestic educational literature with persistence worthy of a better use, is focused on outdated Turbo Pascal with the ancient console shell, poor graphics library and obsolete means of object-orientedprogramming, developing in students a persistent aversion to the Pascal language in general.
- PascalABC.NET is enhanced with modern language features for easy, compact and understandable programming.
- PascalABC.NET is a fairly mature environment. Its prototype, the Pascal ABC learning system, was introduced in 2002. PascalABC.NET is a growing environment. We're working on new language features and new libraries.

The following is **a series of programs with short commentary** that

explain the features and capabilities of PascalABC.NET.

The program texts are located in the working folder (by default C:\PABCWork.NET) in the subfolder Samples\!MainFeatures.

To start the programs, the window must be opened through the menu item "Help / Short Talk" so that it does not completely cover the window of the **PascalABC.NET** shell.

Main

1. **AssignExtpas.** This example illustrates the use of extended assignment operators += - = *= /= for integers and real. The /= operator for integers is, of course, forbidden.
2. **BlockVar. pas.** Variables can be described inside the begin-end block and initialized when described. This is very convenient for intermediate variables, and in PascalABC.NET due to specific implementation speeds up access to variables by 30%.
3. **AutoVars.pas.** If a variable is initialized during description, you may not specify its type: it is determined by the type of the right part (autodefinition of type). A variable - parameter of a for loop can be described directly in the title of the loop, combining this with auto-definition of type.
4. **Simp.leNewFeatures.pas.** An example that combines the features from the previous three examples.
5. **WriteAll.pas.** The write procedure outputs any type. In particular, it outputs all elements of a set. If it does not know the type, it prints the type name.
6. **WriteFormatpas.** The standard WriteFormat procedure allows for formatted output. The format string view is borrowed from .NET.
7. **StandardTypes.pas.** This program lists all standard integer and real types. The program outputs their sizes.
8. **RandomDiap.pas.** Random(a,b), which returns a random integer in the range [a,b], was added to the random number generation functions. It is not necessary to call Randomize procedure at the beginning of the program.
9. **Rea.lExtNums.pas.** Actions with real values cannot cause overflow in .NET. In case of incorrect operations (division by 0, overflow or taking the logarithm of a negative number) we will get either "infinity" value or "NaN" value (not a number).

10. **Foreach.pas.** The foreach statement is designed to loop through containers, such as arrays, sets, and standard library containers (e.g., List<T>). The container elements are read-only.
11. **Amp.pas.** Keywords can be used as names, in which case they should be preceded by a sign & remove keyword attribute. In addition, keywords can be used as fields. For example, &Type or System.Type.

Types

12. **CharFunctpas.** Char characters store Unicode and therefore occupy 2 bytes. The Ord and Chr functions work exactly with Unicode. For compatibility, the OrdAnsi and ChrAnsi functions work in single-byte Windows encoding.
13. **StringTypes.pas.** String strings occupy variable length memory and are projected onto .NET-type System.String. However, unlike NET strings, they are modifiable and indexed with 1. To work with fixed-length strings, you should use the type string[n] or shortstring=string[255]. In particular, typed files are only allowed for short strings.
14. **StringMethods.pas.** String strings have a number of methods as .NET classes. These methods assume that strings are indexed from scratch.
15. **Stringinteger.pas.** All types are classes. Simple types, too. Therefore, it is easier to convert string to integer and real by using static Parse methods of corresponding class (for example, integer.Parse(s)). It is more convenient to convert integer or real to string using instance method ToString (for example, r.ToString).
16. **Enum.pas.** An enumerated type allows you to refer to its constants not only directly, but also using an entry like TypeName.ConstantName. It is worth noting that all enumerated types are derived from System.Enum.
17. **Sets.pas.** Sets can be of any basic type. Internally, a set is stored as a hash table, but when the set is output in the write procedure, its elements are ordered.
18. **DynArray.pas.** Dynamic arrays of array of T are references. The memory for them must be allocated either by calling the standard procedure SetLength, or by using an initializer like new

T[n]. The SetLength procedure keeps the old contents of the array. Dynamic arrays are a class, derived from System.Array class, which has a rather rich interface. First of all we should mention the static methods &Array.Sort and &Array.Resize.

19. **InitRecords.pas.** Field initializers are allowed in records. Record fields are initialized when a record variable is created.
20. **UntypedFile.pas. The** file typeless files are changed compared to Delphi. There are no BlockRead and BlockWrite procedures, but you can directly write data of different types to the untyped file. As long as it is read in the same order.
21. **PointerToRef.pas.** There are some limitations for pointers to managed memory. For example, a pointer cannot directly or indirectly point to an object of a class whose memory is allocated by a constructor call.
22. **Pointers.pas and References.pas.** Pointers lose their position. We recommend actively using references instead.
23. **StructTypeEquiv.pas.** Unlike Delphi, some types have structural equivalence, not name equivalence. For example, structural equivalence is used for dynamic arrays, pointers, sets, and procedural types.

Subprograms

24. **FuncParams.pas.** Subroutines with variable number of parameters are made easy by adding the keyword params before the parameter - dynamic array. Such parameter must be the last in the list.
25. **Overload.pas.** Subroutine names are overloaded without the overload keyword.
26. **ProcVars.pas.** Procedural variables can "accumulate" actions with the += operator. These actions can be disabled using the -= operator. Procedural variables can be initialized not only by regular subroutines, but also by static and instance methods of class.
27. **SwapT. pas.** Generalized subroutines have a simple syntax and are used immediately along with the usual ones: procedure Swap<T>(var x,y: T);

Modules

28. **SystemUnitTestpas.** The system module is named

PABCSystem, not System as in Delphi, and is included implicitly first in the list of uses. The reason for this naming is that the most important namespace in .NET is System. The System module combines many subprograms of the System, Math, and Utils modules of Delphi. This program illustrates the overlap between the PABCSystem module and the System namespace.

29. **MainProgram.pas and MyUnitpas.** The module may have a simplified syntax (without dividing it into an interface section and an implementation section), which is convenient for initial training. In this case, all the names described fall into the interface section of the module.
30. **SystemUnitTestpas.** To use the .NET namespaces, the same syntax as for plugins is used: the .NET namespaces are specified in the uses list. The order of namespaces is the same as in Delphi - from right to left in the list of uses, the PABCSystem module is viewed last.
31. **Main.pas and MyDll.pas.** In PascalABC.NET it's easy to create and use a dll. A dll library is essentially a module that uses the word library instead of the unit keyword. To connect a dll to another program you use the compiler reference directive.
32. **CallCS.pas.** PascalABC.NET is a complete .NET language that can be easily combined with other .NET languages. This example shows how to call a function in a PascalABC.NET program from a dll made in C#.
33. **CallNative.pas.** PascalABC.NET makes it easy to call functions from ordinary dlls.

Standard graphics libraries

34. **GraphABCTestpas.** The GraphABC graphics library is designed for easy teaching of graphics programming. It hides most of the complexities of graphics programming: it redraws the graphics window itself at the right moment and takes care of synchronization of drawing in several handlers. In addition, graphics primitives are procedural, which means that you do not need to create numerous classes, as in NET. And you can also write graphical commands right after the main program starts, that is, you can use graphics in non-event applications.
35. **MouseEvents.pas.** For graphical applications you can use simple mouse and keyboard events implemented as global

procedural variables.

36. **ABC.pas.** We use the library of vector graphic objects ABCObjects to teach students the basics of object-oriented programming early on. However, it can be used to write simple graphical learning and game applications.

Classes

37. **AllFromObectpas.** All classes are descendants of Object, all types are classes. You can find out the type of each variable by calling the GetType method. The typeof operation for a type returns System.Type.
38. **WriteRecord.pas.** By overriding the ToString method in the class or record, we get to output their values in the writeln procedure
39. **ClassConstructor. pas.** The class keyword is used for static methods and fields. Static constructors are used for non-trivial initialization of static fields.
40. **PersonInternal.pas.** The new constructor syntax uses the keyword new and is preferred. For this reason, all constructors defined in the old style must be named Create. Methods can be described directly inside classes and records (as in C++, C#, and Java)
41. **Records.pas.** Methods and constructors in records can be used the same way as in classes. Records cannot be inherited and records cannot be inherited.
42. **Boxing.pas.** When you assign a dimensional type to an object of type Object, packing takes place. To unpack, use an explicit type conversion.
43. **GarbageCollection.pas.** There are no destructors. Automatic garbage collection to return memory allocated to an object variable requires that no one else refers to that memory, directly or indirectly. Therefore, it is usually sufficient to assign nil to the object variable to free the memory.
44. **OperatorOverloading.pas.** Like C++ and C#, in PascalABC.NET you can overload operation signs for records and classes.
45. **Interf.pas.** The interfaces are semantically the same as those in C# and Java. Delphi's sophisticated implementation of COM-

based interfaces is rejected.

46. **Stack.pas.** Generalized classes (generics) allow you to create classes parameterized by one or more types.
47. **Where.pas.** You can set restrictions on parameter types of generalized classes. Restrictions come in three varieties: having a default constructor for a parameter type, inheriting it from a particular class, or implementing an interface.

Standard .NET library

48. **DateTime.pas.** This example illustrates the use of the DateTime class from the standard .NET library.
49. **LinkedListpas.** This example illustrates the use of container classes from the standard .NET library.
50. **WinFormWithButton.pas.** This example illustrates the creation of a windowed application.

What is .NET

Microsoft .NET platform is a set of programs installed on top of the operating system and provides the execution of programs written specifically for .NET. .NET programs are compact, using a single set of data types and libraries. Microsoft is actively developing the .NET platform, releasing new versions with enhanced capabilities. As of early 2019, the latest version is .NET 4.7.1.

The compilation of a .NET program does not generate machine code, but rather so-called bytecode, which contains *virtual machine* commands (in .NET it is called **IL-code** from the Intermediate Language). Bytecode commands are independent of the processor and the operating system used. On start-up the program containing the IL-code is fed to the input of the virtual machine which executes the program. The part of the virtual machine called **JIT-compiler** (Just In Time) immediately translates the intermediate code into machine code (while optimizing it) after launching the .NET-program, and then launches the program for execution. To be precise, the intermediate code is translated into machine code in parts as the program runs.

This method of double compilation is more complex than the usual method, but has several advantages. First, the JIT compiler can detect the type of processor installed on a given computer, so it generates the most efficient machine code. Tests show that this makes some programs run even faster than usual. Second, IL code is much higher level than machine code and contains a number of object-oriented commands. These include the newobj command to call the object constructor, the callvirt command to call the virtual method of the object, and the throw exception generation command.

A program or library for .NET is called *an assembly* and has the traditional extension, exe or dll. Because assemblies contain IL code, they are much more compact than conventional programs and libraries. For example, an application with a main window, menus, and controls takes up only a few dozen kilobytes on disk.

The most "pure" .NET-language is **C#**: it was created specifically for the .NET platform and includes almost all of its features. .NET-languages easily communicate with each other, not only because of

the high-level intermediate code, but also because of the Common Type System (**CTS** - Common Type System). All standard types (string, character, numeric and logical) have the same representation in memory in all .NET-languages. This allows you, for example, to create a dll library in C#, put a class description in it, and then use that library from a PascalABC.NET program to construct an object of that class. You can also develop the library in PascalABC.NET, and then link it with your Visual Basic.NET project. Note that traditional dll libraries don't allow you to store externally accessible classes and have a number of other limitations.

The most important tools provided by the .NET platform are a unified way of handling errors - exception generation and interception - and automatic dynamic memory freeing management, called *garbage collection*. The latter, in particular, means that there is no need for class destructors.

There are programs that can restore the text of a program by IL-code (for example, the program ILSpy).

In addition to the JIT compiler, an important part of the .NET platform is a set of standard libraries (FCL - Foundation Class Library). These include libraries for graphics, networking, databases, XML, containers, and threads, containing thousands of classes. Every .NET language can use all the features of these libraries.

There is an open cross-platform implementation of Microsoft.NET environment - среденvironmentMono, allowing in particular to develop and run .NET-programs under Linux.

Let us briefly note the advantages and disadvantages of the .NET platform.

The advantages of the .NET platform

1. The .NET platform supports a number of .NET languages. These include C#, Visual Basic.NET, F#, managed C++, Delphi Prism, Oberon, Zonnon, Iron Python, Iron Ruby, PascalABC.NET.
2. Any .NET language contains the most modern language features: classes, properties, polymorphism, exceptions, overloading operations, easy creation of libraries.
3. .NET languages are easily combined with each other, similar to each other in syntactic constructions and type system.
4. There is an extensive library of standard FCL classes.
5. .NET applications are compact.
6. The .NET platform is actively being developed by Microsoft, adding both new language features and new libraries.
7. It is much easier to create a .NET compiler than a compiler for a conventional language.

Disadvantages of the .NET platform

1. Running a .NET application is several times slower than running an ordinary application, because it requires loading virtual machine components and external libraries into RAM.
2. .NET-code in some situations is slower than usual (however, in most tasks, this lag is insignificant, and in some - the .NET applications can be ahead of conventional programs).
3. The garbage collector starts when the dynamic memory runs out and takes a few milliseconds to complete. For real-time applications this is unacceptable.
4. Running a .NET application requires that the .NET platform be installed on the computer. Without it, the application will not work (Note that Windows Vista and Windows 7 have the .NET platform built in).

Note that the advantages of the .NET platform many times overlap its disadvantages.

Benefits of PascalABC.NET

Modern Object Pascal programming language

The **PascalABC.NET** language includes almost all of the standard Pascal language, as well as most of the language extensions to Delphi. However, these tools are not enough for modern programming. That's why **PascalABC.NET** has been extended with a number of constructs, and its standard module with a number of subroutines, types and classes, to make it easy to create applications of medium complexity.

In addition, **PascalABC.NET** makes use of most of the facilities provided by the .NET platform: single type system, classes, interfaces, exceptions, delegates, overloading operations, generics, extension methods, lambda expressions.

The standard PABCSystem module, which automatically plugs into any program, contains a huge number of standard types and subroutines that allow you to write clear and compact programs.

PascalABC.NET has all the facilities of the .NET class libraries, constantly expanding with the latest features. This makes it easy to write **PascalABC.NET** applications for the web, the Web, XML documents, regular expressions, and more.

PascalABC.NET allows you to program in a classical *procedural style*, an *object-orientedstyle*, and many elements of *functional style* programming. The choice of style or a combination of these styles is a matter of programmer taste, and when used in teaching - a methodical approach of the teacher.

The combination of rich and modern language tools, the ability to choose different learning curves allows you to recommend **PascalABC.NET on the one hand** as a language for learning programming (from school children to junior and high school students), on the other hand - as a language for creating projects and libraries of medium complexity.

Simple and powerful development environment

The integrated **PascalABC.NET** development environment is designed to create projects of low to medium complexity. It's fairly lightweight, yet provides the developer with all the tools they need, such as a built-in debugger, Intellisense tools (point hint, parameter hint, name hint), navigation to subprogram definition and implementation, code templates, and autoformatted code.

PascalABC.NET also has *a* built-in *form designer that* lets you create complete RAD (Rapid Application Development) style windows applications.

Unlike many professional environments, the **PascalABC.NET** development environment doesn't have a cumbersome interface and doesn't create a lot of auxiliary files on disk when you compile your program. For small programs this allows you to follow the "one program - one file on disk" principle.

In the **PascalABC.NET** environment, great care has been taken to connect the running program to the shell: A console program launched from the shell carries out I/O in a special window built into the shell. You can also run several programs at the same time - all of which will be controlled by the shell.

The integrated **PascalABC.NET** environment allows you to switch between Russian and English in the settings, and not only the interface elements, but also the error messages are localized.

In addition, internal views of **PascalABC.NET** allow you to create compilers for other programming languages and integrate them into the development environment with plugins.

Specialized modules for training

The Microsoft.NET platform provides **PascalABC.NET** with a standard library of huge numbers of classes to do just about anything - from algorithmic to application-specific. This is why **PascalABC.NET** does not need to develop a large number of its own modules.

The proprietary modules that **PascalABC.NET** has focused specifically on introductory programming instruction.

To teach programming to schoolchildren implemented modules of

the classic school executors Robot and Draughtsman, containing about two hundred automatically verifiable tasks on the basic structures of the programming language.

Besides, **PascalABC.NET** contains a module of electronic ProgProgramming g Taskbook (by M. Abramyan), which allows you to automatically set and check tasks. There are also the modules for teacher, which allow to create tasks for executors Robot, Draughtsman and Electronic Taskbook.

The raster graphics module GraphWPF and the vector graphics module WPFObjects and the 3D graphics module Graph3D can be used to create simple graphics as well as interactive animation applications controlled by events.

We should also mention the "student" modules: the Collections module for simplified collections, the Arrays module for simple operations with dynamic arrays, and the Forms module for manually creating simple applications with a windowed user interface.

Differences between PascalABC.NET and Delphi

Added

1. The `+= -=` operations for .NET events and for procedural variables.
2. The operations `+= -= *=` for integers and `+= -= *= /=` for real ones.
3. The `+=` operation for strings.
4. Subprograms with a variable number of parameters.
5. The **`new`** operation to call the constructor (`ident := new type_name(params);`).
6. The **`new`** operation to create a dynamic array.
7. Operation **`typeof`** .
8. Using **`uses`** to connect .NET namespaces (implemented in Delphi Prism).
9. Type of access **`internal`** (along with **`public`**, **`private`**, **`protected`**).
10. Variable initialization: **`var`** `a: integer := 1J`
11. Initialize variables: **`var`** `a := 1`;
12. Declaring local variables in a block.
13. Declare the loop parameter in the loop header: **`for var`** `i := 1`

`to 10 do`, `foreach var x in a do`.

14. The `lock` operator, which provides stream synchronization.
15. Methods in Records.
16. Field initializers in classes and records.
17. Generalized classes (generics).
18. Implemented typed files (unlike Delphi Prism, where they are removed).
19. Simplified module syntax.
20. Description of methods within a class or record interface.
21. Interface write implementation.
22. Expansion methods.
23. Lambda expressions.

Modified

1. Only abbreviated calculation of logical expressions.
2. A different `foreach` syntax.
3. .NET-style `interfaces.`
4. Another syntax for overloading operations.
5. Static class methods instead of class methods. Absence of `Tciass` type.
6. Destructors are left only for compatibility and do not perform any actions.
7. The `object` type is synonymous with `System.Object`.
8. The type `exception` is synonymous with `System.Exception`.
9. Index `string` from 1, switch directive to index from 0.
10. The `write` procedure outputs all types.
11. Structural type equivalence for procedural variables, dynamic arrays, typed pointers, and sets (in Delphi Object Pascal, name type equivalence except for open arrays).
12. Arbitrary type-based sets (`set of string`).
13. Prohibits the use of pointers to managed memory.
14. Procedural variables (delegates) instead of `procedures of object`.
15. You can use the `Read`, `Write` procedures to manipulate the `file` without any type of file.
16. Array arrays differ in type from two-dimensional arrays (in particular, the entries `a[i][j]` and `a[i,j]` are not equivalent).
17. Overload is performed without the `overload` keyword.

18. All constructors are named `Create`.
19. Automatic memory management with garbage collector (except for pointers to unmanaged memory).

Not available

1. Keywords and directives **`packed threadvar inline asm exports unsafe resourcestring dispinterface in out absolute dynamic local platform requires export message resident assembler safecall automated far near stdcall cdecl published stored contains implements varargs default deprecated package register dispid pascal writeonly`** and related features.
2. The typecasting for variables: `char(b) := 'd'`.
3. Ability to assign a subroutine address to a `pointer`.
4. Notes with options.
5. `PChar` strings.
6. Ability to use `@` operation for procedural variables.
7. Variant types.
8. Bestiary parameters (**`var`** `a;` **`const`** `b`).
9. Open arrays (not to be confused with dynamic arrays!).
10. Methods related to **`messages`**.
11. Nested class definitions.

PascalABC.NET language description

PascalABC.NET is a next-generation Pascal language that includes all the features of the standard Pascal language, extensions to Delphi Object Pascal, and some native extensions, as well as some features that make it compatible with other .NET languages. PascalABC.NET is a multi-paradigm language - you can program in a variety of styles: structured programming, object-oriented programming, functional programming.

In addition, the presence of a large number of standard .NET class libraries forms a style that is noticeably different from that of standard Pascal.

This section **describes the PascalABC.NET language**.

Basics

- Program structure
- Expressions and operations

Data types

- Overview of types
- Entire types
- Substance types
- Logic type
- Symbol type
- String type
- Enumerated and range types
- Static arrays
- Dynamic arrays
- Records
- Corteges
- Multitudes
- Files
- Sequences
- Signposts
- Procedural type
- Classes
- Dimensional and reference types
- Memory management and garbage collection

Operators

- Assignment operators
- Compound operator
- Variable description operator
- Loop statement
- Operator of the cycle for
- The foreach loop operator
- The while and repeat loop operators
- Conditional if statement
- Variant case operator
- Procedure call operator
- The try except statement
- The try finally operator
- Operator raise
- The break, continue, and exit operators
- yield operator
- The yield sequence operator
- The goto operator
- Operator lock
- Operator with
- Empty operator

Structural programming

- Procedures and functions
- Modules
- Dll Libraries
- Documentary comments

Object-oriented programming

- Overview of classes and objects
- Inheritance
- Polymorphism
- Generalized types
- Abstract methods and classes
- Anonymous classes
- Autoclasses
- Exception handling
- Expansion methods
- Interfaces
- Attributes (in development)

Functional programming

- Lambda expressions
- Capturing variables
- Sequences
- Sequence methods

Standard modules

- PABCSystem module

Additional questions

- Scope of the identifier
- The compiler's initiatives
- Open MR

Program structure: overview

The program contains keywords, identifiers, and comments. Keywords are used to highlight syntactic constructions and are highlighted in bold in the editor. Identifiers are names of program objects and cannot coincide with keywords.

The program in **PascalABC.NET** looks like this:

```
program NAME;
uses section
descriptions section begin
   end operators.
```

The first line is called ***the program header*** and is optional.

The **uses** section consists of several consecutive **uses** sections, each starting with the keyword **uses**, followed by a list of module names and .NET namespaces, listed separated by commas.

The descriptions section may include the following subsections:

- variable description section
- constant description section
- type description section
- tag description section
- Procedures and functions section

These subsections follow one another in no particular order.

This is followed by the **begin/end** block, inside which are operators separated one from the other by a semicolon. Among the operators there may be a variable description operator, which allows you to describe variables within the block.

The **uses** section and the descriptions section may be missing.

For example:

```
program MyProgram;
var a,b: integer; x: real;
begin
  readln(a,b);
  x := a/b; writeln(x);
end.
```

or

```
uses GraphABC;
begin
  var x := 100;
  var y := 100;
  var r := 50;
  Circle(x,y,r);
end.
```

Identifiers and keywords

Identifiers are names of programs, modules, procedures, functions, types, variables and constants. An identifier is any sequence of Latin letters or numbers beginning with a letter. A letter is also the underscore character "_".

For example, `ai`, `_h`, `bi23` are identifiers and ia is not.

Each identifier has an identifier scopeassociated with it.

The following words are ***keywords***, used to formalize language constructs and cannot be used as identifiers:

```
and array as auto begin case class constructor constructor
destructor divor do downto else end event except
extensionmethod file finalization finally for foreach function
goto if implementation inherited initialization interface is
label lock loop mod nil not of operator or procedure property
program raise record repeat sealed set sequence shl shr sizeof
template then to try typeof until uses using var where while
with xor
```

A number of words are ***contextually key*** (they are key only in some context):

```
abstract default external forward internal on overload override
params private protected public read reintroduce unit virtual
write
```

Contextual keywords can be used as names.

Some keywords coincide with the most important names of the .NET platform. So in **PascalABC.NET** it is possible to use these names without conflicting with the keywords.

The first way is to use a qualified name. For example:

```
var a: System.Array;
```

In this context, the word `Array` is a name within the `System` namespace, and there is no conflict with the `array` keyword.

The second way is to use the special symbol `&` before the name. In this case the name can coincide with the keyword. For example:

```
uses System;
var a: &Array;
```

Comments Off on

Comments are portions of code ignored by the compiler and used by the programmer to explain the program text. There are several types of comments in **PascalABC.NET.**

The sequence of characters between curly braces `{ }` or characters `(*` and `*)` is considered a comment:

```
{ This is a commentary }
(* This is also a commentary.)
```

Any sequence of characters after `//` and before the end of the string is also considered a comment:

```
var Version: integer; // Product version
```

Comments of different types can be nested:

```
{ This is another (* comment *)}
```

Variable description

Variables can be described in the descriptions section as well as directly inside any **begin/end** block.

The variable description section begins with the **var** keyword, followed by elements of the description kind

list of names: *type* ;

or

name: *type* := *expression* ;

or

name: *type* = *expression*; // for Delphi compatibility

or

name := *expression* ;

The names in the list are listed, separated by commas. For example:

```
var a,b,c: integer; d: real := 3.7; s := 'PascalABC forever';
  al := new List<integer>; p1 := 1;
```

In the last three cases, the type of the variable is automatically determined by the type of the right part.

Variables can be described directly within a block. Such descriptions are called intrablock descriptions and represent a variable description operator.

In addition, loop variables can be described in the header of the **for** and **foreach** statements.

Global variables are initialized with zero values. This is not guaranteed for local variables - they must be initialized explicitly.

Combining variable description and tuple assignment

You can combine tuple assignment (unpacking a tuple into variables) with variable description:

```
var t := (1,2);
(var a, var b)    (1,2);
```

or

```
var (a,b)    (1,2);
```

Unpacking a tuple into variables is often used when returning a tuple function:

```
function SP(a,b: real)    (a*b,2*(a+b));

var (S,P) := SP(2,3);
```

Constants description

The section describing named constants begins with the service word **const**, followed by elements of the description kind

constant name = *value* ;

or

constant name: *type* = *value* ;

For example:

```
const Pi = 3.14; Count = 10; Name = 'Mike'; DigitsSet =
  ['0'...'9']; Arr: array [1...5] of integer = (1,3,5,7,9);
  Rec: record name: string; age: integer end = (name:
'Ivanov'; age: 23);
  Arr2: array [1..2,1..2] of real = ((1,2),(3,4));
```

Label Description

The label section starts with the reserved word **label**, followed by a comma-separated list of labels. Identifiers and positive integers can be used as labels:

```
label a1,l2,777777;
```

Labels are used to jump in thegotostatement.

Description of types

The type section begins with the service word **type**, followed by lines like

type name = type ;

For example, `type arr10 = array [1...10] of integer; myint = integer;`

```
pinteger = Ainteger;
IntFunc = function(x: integer): integer;
```

Usually a description is used for composite types (static arrays, procedural variables, records, classes) to give a name to a complex type. If named type equivalence is defined for a type, this is the only way to pass variables of that type to a subroutine.

It is obligatory to use type descriptions for classes:

```
type A = class
    1: integer;
    constructor Create(ii: integer);
    begin i:=ii; end;
  end;
```

If the type description is used simply to replace one name with another, these types are called *type synonyms*:

```
type int = integer; double = real;
```

Type descriptions can be generalized, i.e., include parameters-types in angle brackets after the type name.

```
type
  Dict<K,V> = Dictionary<K,V>;
  Arr<T> = array of T;
```

Using such a type with a specific type parameter is called type *instantiation:*

```
var
  a: Arr<integer>;
  d: Dict<string,integer>;
```

When describing recursive data structures, a type pointer may appear before the type itself in the definition of another type:

```
type
  PNode = ATNode;
  TNode = record
    data: integer;
    next: PNode;
```

```
  end;
```

It is important that the definitions of both types are in the same **type** section.

Unlike Delphi Object Pascal the following recursive description is correct:

```
type
  TNode = record
    data: integer;
    next: ATNode;
  end;
```

Note that for reference types (classes) it is allowed to describe a field with a type that matches the type of the current class:

```
type Node = class data: integer; next: Node;
  end;
```

Scope of the identifier

Any identifier used in the program must be described beforehand. Identifiers are described in the descriptions section. Identifiers for variables can also be described within a block.

The main program, subprogram, block, module, class form the so-called ***namespace*** - an area in a program in which the name must have a single description. Thus, two identical names cannot be described in the same namespace (except for overloadedsubroutine names). In addition, there are explicit namespace definitions in .NET assemblies.

The scope of an identifier (i.e., the place where it can be used) extends from the moment of description to the end of the block in which it is described. The scope of a global identifier described in a module extends to the entire module, as well as to the main program to which the module is connected in the **uses** section.

In addition, there are variables defined in the block and associated with some constructs (**for**, **foreach**). In this case the action of variable `i` extends to the end of the corresponding construct. So, the following code is correct:

```
var a: array of integer := (3,5,7); for i: integer := 1 to 9
do
  write(a[i]);
foreach i: integer in a do write(i);
```

An identifier with the same name defined in a nested namespace *hides* an identifier defined in an external namespace. For example, in code

```
var i: integer;
procedure p;
var i: integer;
begin
  i := 5;
end;
```

value 5 will be assigned to the variable `i` described in the procedure `p`; inside the procedure `p` it is not possible to refer to the global variable `i`.

Variables described within a block cannot have the same names as variables in the descriptions section of that block. For example, the

following program is wrong:

```
var i: integer;
begin
  var i: integer; // error end.
```

In contrast, derived classes can define members with the same names as in base classes, and their names hide the corresponding names in base classes. The keyword **inherited** is used to refer to the same name member of the base class from the method of the derived class:

```
type
  A=class
    i: integer; procedure p; begin
      i := 5;
    end;
  end;
  B=class(A)
    i: integer; procedure p; begin
      i := 5;
      inherited p;
    end;
  end;
```

The algorithm for finding a name in a class is as follows: first, the name is searched in the current class, then in its base classes, and if not found, then in the global scope.

The algorithm for finding a name in the global scope when there are several connected modules is as follows: first, the name is searched for in the current module, then, if not found, by the chain of connected modules in the order from right to left. For example, in the program

```
uses unit1,unit2;
begin
  id := 2;
end.
```

The description of the variable `id` will be searched first in the main program, then in the `unit2` module, then in the `uniti` module. In this case, different modules may have different `id` variables described. This situation means that `uniti` forms an external namespace, the `unit2` namespace is directly nested in it, and the main program namespace is nested in `unit2`.

If, in the last example, both `uniti` and `unit2` define `id` variables, it is recommended to specify the variable name by the name of the module using the *ImModule* construct`.` *Name*:

```
uses unit1,unit2;
begin
  unitl.id := 2;
end.
```

Overview of types

Types in **PascalABC.NET** are divided into simple types, structured types, pointer types, procedural types and sequences.

Simple types include integer and real types, logical, character, enumerated, and range types.

A data type is called ***structured*** if one variable of that type can contain many values.

Structured types include arrays, strings, records, tuples, sets, files, and classes.

A special type of data is the sequence, which stores in essence the algorithm for obtaining sequence data one by one.

All simple types except the real type are called ***chunk types***. Only values of these types can be indexes of static arrays and parameters of the `for` loop. In addition, for ordinal types, the functions `Ord`, `Pred`, and `Succ` are used, as well as the procedures `Inc` and `Dec`.

All types except the pointer types are derived from the `object` type. Each type in **PascalABC.NET** has a mapping to a.NETtype.The pointer type belongs to unmanageable code and is modeled by the `void*` type.

Most types in **PascalABC.NET** are subdivided into dimensional types, reference types, and pointer types. A comparison of dimensional and reference types is given here.

In addition, PascalABC.NET has several types inherited from Delphi Object Pascal that are difficult to categorize as dimensional or reference types. These are static arrays, sets, dimensional strings and files. In terms of their representation in memory, they are of reference type, but in terms of behavior, they are of dimensional type.

Dimensional and reference types

Most types in **PascalABC.NET** fall into two large groups: ***dimensional*** and ***reference***.

Dimensional types include all simple types and records. More precisely, all dimensional types are inherited from the .NET type System.ValueType.

Reference types include strings, dynamic arrays, tuples, classes, sequences, and proceduraltypes. More precisely, all reference types are inherited from the .NET type System.Object and are not inherited from System.ValueType.

Dimensional and reference types differ in the following characteristics:

- memory allocation
- memory management
- assignment
- comparison
- subprogramming

In addition, PascalABC.NET has several types inherited from Delphi Object Pascal that are difficult to categorize as dimensional or reference types. These are static arrays, sets, dimensional strings and files. In terms of their representation in memory, they are of reference type, but in terms of behavior (assignment, comparison, subroutines), they are of dimensional type.

Dimensional types are more efficient in calculations: they occupy less memory and operations performed on small dimensional types are most efficient. Reference types are more flexible: memory is allocated for them dynamically while the program is running and is released automatically when the object of reference type is no longer used.

Memory allocation

The memory for a variable of dimension type is allocated on the program stack. For global variables, memory is allocated at program start, for local variables - at the moment of subprogram call. A variable of the dimensional type stores the value of this type.

```
var i: integer; // allocate memory for i on the program stack
i := 5;
```

A reference type variable stores a reference to an object of some class in dynamic memory. If it is not initialized, it stores a special value nil (null reference). To initialize reference variables a constructor call of the corresponding class is used:

```
type Person = auto class name: string;
  age: integer;
end;

var p: Person; // p stores the value nil, no memory for the object is allocated
p := new Person('Ivanov',2 0),- // the constructor allocates memory for the object and writes a reference to it in a variable
```

Features of PascalABC.NET

Strings in **PascalABC.NET** are not initialized with nil by default, but with an empty string.

Assignment

When assigning variables of a dimensional type, the value of that type is copied.

```
type Point3 = record x,y,z: real;
end;

var p1,p2: Point3;
pl.x := 1; pl.y := 2; p1.z := 3; p2 := pl; // all fields
are copied Print(p2); // (1,2,3)
p1.x := 4; p1.y := 5; p1.z := 6;
Print(p2); // (1,2,3) - p2 does not change, because it
occupies another memory on the stack
```

When you assign variables of reference type, a reference is copied, so after assignment, both references end up referring to the same object in dynamic memory:

```
type Point3 = auto class x,y,z: real;
end;

var p1,p2: Point3; // variables store nil reference p1 :=
new Point3(1,2,3);
p2 := p1;  // the reference is copied, after which p2 is
referenced
to the same object as p1
Print(p2); // (1,2,3)
p1.x := 4; p1.y := 5; p1.z := 6;
Print(p2); // (4,5,6) - the object has changed, p2 refers
to the same object as p1
```

Static arrays, sets, and dimension strings behave like dimension types when assigned. Thus, when assigning one static array to another, all elements are copied:

```
var a,a1: array [1...1000000] of integer;
a1 := a; // all 1000000 elements are copied (long)
```

Equality comparison

The equality and inequality comparison of objects of dimensional type compares their values. In particular, two variables of record type are equal if all fields of those records are equal.

```
type PersonRec = record name: string;
  age: integer;
end;
var p,p1: PersonRec;
p.name := 'Ivanov'; p.age := 20;
pl.name := 'Ivanov'; p1.age := 20;
writeln(p1 = p); // True
```

Two variables of reference type are equal by default if they refer to the same object.

```
type Person = auto class
  name: string;
  age: integer;
end;
var p := new Person('Ivanov',20);
var p1 := new Person('Ivanov',20);
writeln(p1 = p); // False
```

However, the comparison operation can be overloaded. For example, for strings and tuples the equality comparison is redefined so that not references but values are compared.

Transfer to subprograms

When transferring dimensional types by value, the value of the actual parameter is copied to a variable - a formal parameter. If the dimensional type has a large size, it can take a long time, so the dimensional type in this case is transferred by reference to the constant:

```
type Point3 = record x,y,z: real;
end;

procedure PrintPoint(const p: Point3);
begin
  Print(p.x,p.y,p.z) end;
```

Reference types are usually passed to a subprogram by value. When such parameters are passed, the reference is copied, resulting in the formal and actual parameters referring to the same object.

```
procedure Change666(a: array of integer);
begin
  a[0] := 6 6 6;
end;
```

In this case, as a result of changing a formal parameter within a subprogram, the content of the corresponding actual parameter when the subprogram is called also changes.

Reference types are passed to a subprogram by reference only if the reference itself changes within the subprogram:

```
procedure CreateA(var a: array of integer); begin
  a := new integer[10];
end;
```

Static arrays, dimensional strings, and sets behave like dimensional types when passed to subroutines. For example, it is inefficient to try to pass a static array by value into a subroutine, because a lot of data is copied. Therefore, static arrays are always passed by reference:

```
type Arr = array [1...100] of integer;

procedure PrintArray(const a: Arr; n: integer);
begin
  for var i:=1 to n do Print(a[i]) end;
```

Memory management

Dimensional types are allocated on the program stack, so they do not need special memory management. Global dimensional variables are allocated memory the whole time of program operation. Local dimensional variables are allocated memory at the moment a subprogram is called, and are released at the moment the subprogram is finished.

The memory managementfor reference types is done automatically by the garbage collector. The garbage collector is started at an unspecified point in time, when the managed memory is no longer sufficient. It returns to the pool of unused memory those objects that are no longer referenced, and then defragments the remaining memory, resulting in dynamic memory is always defragmented and its allocation when the constructor is called is almost instantaneous.

Static arrays, dimensional strings, sets and files are referential in terms of memory allocation and the memory occupied by values of this type is also managed by the garbage collector.

Entire types

Below is a table of integer types that also contains their size and range of valid values.

Type	Size, byte	Value range
`shortint`	1	-128..127
`smallint`	2	-32768..32767
`integer`, `longint`	4	-2147483648..2147483647
`int64`	8	-
`byte`	1	0..255
`word`	2	0..65535
`longword`, `cardinal`	4	0..4294967295
`uint64`	8	0..18446744073709551615
`Biginteger`	variable	unrestricted

The types `integer` and `longint` as well as `longword` and `cardinal` are

synonymous.

The maximum values for each integer type are defined as external стандastandard constants: `MaxInt64`, `Maxint`, `MaxSmallint`, `MaxShortInt`, `MaxUInt64`, `MaxLongWord`, `MaxWord`, `MaxByte`.

For each integer type `t` except `Biginteger,` the following constants are defined as static members:

- `T.Minvalue` is a constant representing the minimum value of type `t`;
- `T.Maxvalue` is a constant representing the maximum value of type `t`;

Static functions are defined for each integer type `t: T.Parse(s)` - a function that converts a string representation of a number to a value of type `t`. If the conversion is not possible, an exception is generated;

- `T.TryParse(s,res)` is a function that converts a string representation of a number to a value of type `t` and writes it to the `res` variable. If the conversion is possible, it returns `True`, otherwise it returns `False`.

In addition, an instance function `ToString` is defined for `t`, which returns a string representation of a variable of that type.

Constants of integer type can be represented both in decimal and hexadecimal form, with a $ sign in front of the hexadecimal constant:

```
253456 $FFFF
```

Substance types

Below is a table of real types with their size, number of significant digits and range of valid values:

Type	Size, byte	Quantity significant digits	Value range
`real`	8	15-16	-1.810308.. 1,8-10308
`double`	8	15-16	-1.810308.. 1.8-10308
`single`	4	7-8	-3.41038.. 3.4-1038
`decimal`	16	28-29	-79228162514264337593543950335 .. 79228162514264337593543950335

The types `real` and `double` are synonymous. The smallest positive number of type `real` is approximately 5L10-324, for type `single` it is approximately 1.4-10^{45}.

The maximum values for each real type are defined as external [стандаstandard constants](): MaxReal, MaxDouble and MaxSingle.

For each real type `R,` in addition to `decimal,` the following constants are also defined as static class members:

- `R.Minvalue` is a constant representing the minimum value of type `R`;
- `R.Maxvalue` is a constant representing the maximum value of type `R`;
- `R.Epsilon` is a constant representing the smallest positive number of type `R`;
- `R.NaN` is a constant that does not represent a number (occurs, for example, when dividing 0/0);
- `R.NegativeInfinity` is a constant that represents negative infinity (occurs, for example, when division -2/0);
- `R.positiveinfinity` is a constant representing positive infinity (occurs, for example, when dividing 2/0).

The following static functions are defined for each real type `R` except `decimal`:

- `R.isNaN(r)` - returns `True` if `r` stores the value `R.NaN`, and `False` otherwise;
- `R.isinfinity(r)` - returns `True` if `R.Positiveinfinity` or `R.NegativeInfinity` is stored in `r`, and `False` otherwise;
- `R.IsPositiveInfinity(r)` - returns `True` if `r` contains the value `R.PositiveInfinity`, and `False` otherwise;
- `R.IsNegativeInfinity(r)` - returns `True` if `R.NegativeInfinity` is stored in `r`, and `False` otherwise;

The following static functions are defined for each real type `R:`

- `R.Parse(s)` is a function that converts a string representation of a number to a value of type `R`. If the conversion is not possible, an exception is generated;
- `R.TryParse(s,res)` is a function that converts a string representation of a number to a value of type `R` and writes it to the `res` variable. If the conversion is possible, it returns `True`, otherwise it returns `False`.

In addition, an instance function `ToString` is defined that returns a string representation of a variable of type `R`.

Real constants can be written in either floating-point or exponential form:

`1.70.0132` `.5e3` (2500) `1.4e-1` (0.14)

Logic type

Values of the `boolean` type take up 1 byte and take one of the two values given by the predefined constants `True` and False.

Static methods are defined for the logical type:

- `boolean.Parse(s)` is a function that converts a string representation of a number to a `boolean` value. If the conversion is not possible, an exception is generated;
- `boolean.TryParse(s,res)` is a function that converts a string representation of a number to a value of `boolean` type and writes

it to `res` variable. If the conversion is possible, it returns `True`, otherwise it returns `False`.

In addition, an instance function `Tostring` is defined that returns a string representation of a variable of type `boolean`.

The logical type is ordinal. Specifically, `False< True,` Ord`(False) =0,` Ord`(True) = 1`.

Symbol type

The `char` character type occupies 2 bytes and stores a Unicode character. Characters are implemented by the `System.Char` type of the .NET platform.

The `+` operation for characters means concatenation (merging) of strings. For example: `'a'+'b' = 'аь'`. As for strings, if you add a number to a character, the number is preconverted to a string representation:

```
var s: string := ' '+15; // s = ' 15'
var s1: string := 15+' '; // s = '15 '
```

Over the characters are defined comparison operations `< > <= >= = <> ,` which compare character codes:

```
'a'<'b' // True '2'<'3' // True
```

The standard functions `Chr` and `Ord are used` to convert between characters and their Unicode codes:

`Chr(n)` is a function that returns a character with the code `n` in Unicode;
`Ord(c)` is a function that returns a `word` type value representing the Unicode code of the `c` character.

The standard functions `ChrAnsi` and `OrdAnsi are used` to convert between characters and their codes in Windows encoding:

`ChrAnsi(w)` - returns the character with the code `w` in Windows encoding;
`OrdAnsi(c)` - returns a `byte` value representing the code of the `c` character in Windows encoding.

In addition, the `#number` expression returns a Unicode character with a *number* code (the number must be in the range 0 to 65535).

Explicit type conversions play a similar role:

`char(w)` returns the Unicode character code `w`; `word(c)` returns the Unicode character code `c`.

СтандаStandard subroutinesfor working with symbols.

Static `char` methods.

Enumerated and range types

An enumerated type is defined by an ordered set of identifiers.

```
typeName = (valuel, value2, ..., valuen);
```

Values of an enumerated type occupy 4 bytes. Each value `value is` a constant of `typeName` that falls into the current namespace.

For example:

```
type
  Season = (Winter,Spring,Summer,Autumn);
  DayOfWeek = (Mon,Tue,Wed,Thi,Thr,Sat,Sun);
```

A constant of an enumerated type can be referred to directly by name, or you can use the `typeName.value` entry, in which the name of the constant is specified by the name of the enumerated type to which it belongs:

```
var a: DayOfWeek;
a := Mon;
a := DayOfWeek.Wed;
```

Values of the enumerated type can be compared by <:

```
DayOfWeek.Wed < DayOfWeek.Sat
```

The functions `Ord`, `Pred` and `Succ`, and the procedures `Inc` and `Dec` can be used for values of the enumerated type. The `Ord` function returns the ordinal number of the value in the list of constants of the corresponding enumerated type, with numbering starting from zero.

For an enumerated type, an instance function `ToString` is defined that returns a string representation of a variable of the enumerated type. When you output a value of the enumerated type with the `write` procedure, it also outputs a string representation of the enumerated type value.

For example:

```
type Season = (Winter,Spring,Summer,Autumn);
var s: Season;
begin
        Summer;
  writeln(s.ToString); // Summer writeln(s); // Summer
end.
```

A range type is a subset of values of an integer, character, or enumerated type and is described in the form `a..y`, where `a` is the lower bound, y is the upper bound of the interval type, `a<` y:

```
var
  intI: 0...10;
  intC: 'a'.. 'z';
  intE: Mon;
```

The type, on the basis of which a range type is built, is called *the base type* for that range type. Values of a range type occupy the same amount of memory as values of the corresponding base type.

String type

Strings are of type `string`, consist of a set of consecutive `char` characters and are used to represent text.

Strings can be of any length. Characters in a string can be accessed using an index: `s[i]` denotes `the i-th` character in the string, numbering starts with one. If index `i exceeds` the string length, an exception is generated.

Comparison operations are definedoverstrings: `< > <= >= = <>.` Comparison of strings for inequality is performed **lexicographically**: `si < s2` if for the first non-matching character with number i si[`i]<s2[i] or` all characters of strings are matched, but si is shorter than s2.

The `+` operation for strings means concatenation (merging) of strings. For example: `'Petya'+'Masha' = 'PetyaMasha'`.

The extended assignment operator `+=` for strings adds a string - the right operand - to the end of the string - the left operand.
For example:

```
var s: string := 'Petya';
s += 'Masha'; // s = 'PetyaMasha'
```

A string can be added to a number, with the number being preconverted to a string representation:

```
s := 'Width: ' + 15; // s = 'Width: 15'
s := 20.5 + ''; // s = '20.5'
s += 1; // s = '20.51'
```

The operation `*` is defined over strings and integers: `s*n` and `n*s` means the string formed from the string `s` repeated n times:

```
s := '*'*10; // s = '**********'
s := 5*'ab' // s = 'abababababab' s := 'd'; s *= 3; // s =
'ddd'
```

A slice taking operation is also defined over the lines.

Strings are implemented by the `System.String` type of the .NET platform and are a reference type. Thus, all operations on strings are inherited from the `System.string` type. Unlike .NET strings, however, strings in **PascalABC.NET** are modifiable. For example, you can change `s[i]` (you can't in .NET). Furthermore, `strings` in **PascalABC.NET behave** like dimensional strings: after

```
var s2 := 'Hello';
var si := s2; si[2] := 'a';
```

string `s2` will not change. Here is what is called Copy On Write - when you change a character of the string, it creates a copy, so s1 and s2 start referring to different parts of the memory.

By default, strings are initialized with an empty string (in .NET with nil). However, it is possible to assign `nil` to a string, which is necessary to work with NET code.

In addition, for compatibility with Delphi Object Pascal, **PascalABC.NET** implements short strings like `string[n]`.

СтандаStandard subroutinesfor working with strings.

Members of the`string`class.

Short Lines

To be compatible with Delphi Object Pascal, **PascalABC.NET** implements short strings. The type `string[n]` is used to describe it, where `n` is an integer type constant indicating the string length (n <= 255). For compatibility with Delphi Object Pascal, the standard module describes the type `shortstring=string[255]` .

Short strings behave like normal strings except for a few things.

1. Short strings, unlike normal strings, can be used as components of typed files.
2. If a short string is assigned to a string that exceeds its size, it is truncated to the size of the original string. For example:

```
var s: string[3] := 'ABCD';
Print(s); // ABC
```

3. A short string, unlike a normal string, cannot be assigned nil.
4. For efficiency, short strings should be passed to the subprogram by reference, using the modifier **`var`** or **`const`**.

СтандаStandard subroutinesfor working with strings.

Members of the`string`class.

Methods of type string

The `string` type in PascalABC.NET is a class and contains a number of properties, static and instance methods, and extension methods.

The methods of the `string` class assume that strings are indexed from zero. In addition, no method changes the string, because strings in .NET are immutable.

String class properties

Property	Description
`s[i]`	Index Property. Returns or allows you to change the i-th character of the string s. Strings in PascalABC.NET are indexed from 1.
`Length: integer`	Returns the string length

String class static methods

The method	Description
`String.Compare(s1,s2: string): integer`	Compares the lines si and s2. Returns a number <0 if s1<s2, =0 if s1=s2 and >0 if s1>s2
`String.Compare(s1,s2: string; ignorecase: boolean): integer`	Same. If ignoreorecase=True, then strings are compared without case of letters
`String.Format(fmtstr: string, params arr: array of objects): string;`	Formats arr parameters according to the format string fmtstr
`String.Join(ss: array of string; delim: string): string`	Returns the string obtained by merging ss strings using delim as a delimiter

Instance methods of the String class

Note that all instance methods do not change the string, as it may seem at first sight, but return the changed string if necessary. In addition, the characters in the term are considered to be indexed from zero.

The method	Description
`Contains(s: string): boolean`	Returns True if the current string contains s, and False otherwise
`EndsWith(s: string): boolean`	Returns True if the current string ends with s, and False otherwise
`IndexOf(s: string): integer`	Returns the index of the first occurrence of substring s in the current string or -1 if no substring is found
`IndexOf(s: string; start,count: integer): integer`	Returns index of the first occurrence of substring s in the current string or -1 if no substring is found. The search starts with the character number start and extends to

	to the following counts of characters
`IndexOfAny(cc: array of char): integer`	Returns the index of the first occurrence of any character in the array cc
`Insert(from: integer; s : string): string`	Returns the string obtained from the original string by inserting a substring of s at the from
`LastIndexOf(s: string): integer`	Returns the index of the last occurrence of substring s in the current string
`LastIndexOf(s: string; start,count: integer): integer`	Returns index of the last occurrence of substring s in the current string or -1 if no substring is found. The search starts with the character number start and extends to the following count of characters
`LastIndexOfAny (a: array of char): integer`	Returns the index of the last occurrence of any character in the array cc
`PadLeft(n: integer): string`	Returns a string,

	obtained from the original string by right-aligning and filling it with spaces on the left side up to the length of n
`PadRight(n: integer): string`	Returns a string that was derived from the original string by left-aligning and filling it with spaces to the right up to the length of n
`Remove(from,len: integer): string`	Returns the string obtained from the original string by deleting the len of the simoles from the from position
`Replace(s1,s2: string): string`	Returns the string obtained from the original string by replacing all occurrences of substring s1 with string s2
`Split(params delim: array of char): array of string`	Returns an array of strings, obtained by splitting the original string into words, with the delimiters

	any of the characters is used (the default is a space)
`StartsWith(s: string): boolean`	Returns True if the current line begins with s, and False otherwise
`Substring(from,len integer) : string`	Returns a substring of the original string from the position from the length of len
`ToCharArray: array of char`	Returns a dynamic array of source string characters
`ToLower: string`	Returns a lowercase string
`ToUpper: string`	Returns the string, converted to uppercase
`Trim: string`	Returns a string derived from the original string by removing leading and trailing spaces
`TrimEnd(params cc: array of char)  string`	Returns a string derived from the original

	by removing the terminating characters from the cc array
`TrimStart(params cc: array of char): string`	Returns a string derived from the original string by removing the leading characters from the cc array

Methods for extending the String class

Some extension methods are standard for .NET, some are implemented only in PascalABC.NET.

The method	Description
`Inverse: string`	Returns the string inversion
`Print`	Outputs the letters of the string, separated by a space
`Println`	Outputs the letters of a line separated by a space, and jumps to a new line
`ReadInteger(var from: integer): integer`	Reads an integer from the from position and returns it. The from position is incremented by the read element
`ReadReal(var from: integer): real`	Reads from a string a real number from the from position and returns it. The from position is incremented by the read element
`ReadWord(var from: integer): string`	

	Reads a word from a string before a space or before the end of the string from the from position and returns it. The from position is incremented by the element read
`ToInteger: integer`	Converts the string to an integer and returns it. If this is not possible, an exception is generated
`ToIntegers: array of integer`	The string must contain integers separated by spaces. An array of integers is returned. If this is not possible, an exception is generated
`ToReal: real`	Converts the string to a real and returns it. If it impossible, an exception is generated
`ToReals: array of real`	The line should

	stored real, separated by spaces. An array of real ones is returned. If this is not possible, an exception is generated
`ToWords(params delim: array of char): array of string`	Returns an array of strings, obtained by splitting the original string into words, with the delimiters any of the characters is used according to The default is. space). Unlike s.Split, it does not include empty strings in the resulting array. In particular, this means that words can be separated by several separators

Arrays

An array is a set of elements of the same type, each with its own

number, called *an index* (there can be several indexes, then the array is called *multidimensional*).

Arrays in **PascalABC.NET** are divided into static and dynamic arrays.

An exception is always generated in **PascalABC.NET when an** index change is out of bounds.

Dynamic arrays

Dynamic array description

A dynamic array type is constructed as follows:

> **`array of`** *element type* (one-dimensional array) **`array [,] of`** *element type* (two-dimensional array)
> etc.

A variable of dynamic array type is a reference. Therefore, a dynamic array needs to be initialized (allocated memory for elements).

Allocating memory for a dynamic array

There are two ways to allocate memory for a dynamic array. The first method uses the `new` operation in the style of a class constructor call:

```
var
  a: array of integer;
  b: array [,] of real;
begin
  a := new integer[5];
  b := new real[4,3]; end.
```

The good thing about this method is that it allows you to combine array description and memory allocation:

```
var
  a: array of integer := new integer[5];
  b: array [,] of real := new real[4,3];
```

You can omit the type description in this case - the type is autodetected:

```
var
  a := new integer[5]; b := new real[4,3];
```

The second way to allocate memory for a dynamic array uses the standard `setLength` procedure:

```
SetLength(a,10);
SetLength(b,5,3);
```

The elements of the array are filled with default values.

The `SetLength` procedure has the advantage that when it is called again, the old contents of the array are preserved.

Initializing a dynamic array

You can initialize a dynamic array when allocating memory for it with the new operation:

```
a := new integer[3](1,2,3);
b := new real[4,3] ((1,2,3),(4,5,6),(7,8,9),(0,1,2));
```

Initialization of a dynamic array at the time of description can be done in abbreviated form:

```
var
  a: array of integer := (1,2,3);
  b: array [,] of real := ((1,2,3),(4,5,6),(7,8,9),
(0,1,2));
  c: array of array of integer := ((1,2,3),(4,5),(6,7,8));
```

This allocates memory for the number of elements specified on the right.

The easiest way to initialize a one-dimensional array is to use the standard functions Seq..., which allocate memory of the desired size and fill the array with the specified values:

```
var a := Arr(1,3,5,7,8);                    //
integer
var s := Arr('Ivanov','Petrov','Sidorov'),-   array of
// string
var b := ArrFill(777,5);                    // array of
[777,777,777,777,777]
var r := ArrRandom(10);                     // b =

10 random integers in the range from 0 to 99  fill in
```

In the same style you can initialize arrays of arrays:

```
var a := Arr(Arr(1,3,5),Arr(7,8),Arr(5,6)); // array of
array of integer
```

Length of the dynamic array

A dynamic array remembers its length (an n-dimensional dynamic array remembers the length for each dimension). The length of an array (the number of elements in it) is returned by the standard `Length` function or the `Length` property:

```
l := Length(a);
l := a.Length;
```

For multidimensional arrays, the length for each dimension is returned by the standard `Length` function with two parameters or by the `GetLength(i)` method :

```
l := Length(a,0);
l := a.GetLength(0);
```

Output a dynamic array

After memory allocation, the input of a dynamic array can be done traditionally in a loop:

```
for var i:=0 to a.Length-1 do read(a[i]);
```

You can enter a dynamic array using the standard function ReadSeqInteger:

```
var a := ReadSeqInteger(10);
```

In this case, a dynamic array is allocated memory of the required size.

Output a dynamic array

The write procedure outputs a dynamic array by enclosing the elements in square brackets and separating them with commas:

```
var a := Arr(1,3,5,7,9);
writeln(a); // [1,3,5,7,9]
```

An n-dimensional dynamic array is output so that each dimension is enclosed in square brackets:.

```
var m := new integer[3,3] ((1,2,3),(4,5,6),(7,8,9));
writeln(m); // [[1,2,3],[4,5,6],[7,8,9]]
```

A dynamic array can also be output using the Print or Println extensions:

```
a.Println;
```

The elements are separated by spaces by default, but you can change this by setting the Print parameter, which is the element separator. For example:

```
a.Print(NewLine);
```

displays each element on a separate line.

Array arrays

If an array of arrays is declared

```
var c: array of array of integer;
```

then it can only be initialized with `SetLength`:

```
SetLength(c,5);
for i := 0 to 4 do
  SetLength(c[i],3);
```

To initialize such an array with `new`
enter the type name for `an array of integer`:

```
type IntArray = array of integer;
var c: array of IntArray; - --
c := new IntArray[5];
for i := 0 to 4 do
  c[i] := new integer[3];
```

Array initialization can also be done in abbreviated form:

```
var
  c: array of array of integer := ((1,2,3),(4,5),(6,7,8));
```

Assigning dynamic arrays

Dynamic arrays of the same type can be assigned to each other, with both reference variables pointing to the same memory:

```
var a1: array of integer;
var a2: array of integer; a1 := a2;
```

Note that structuralequivalence of types is adopted for dynamic arrays: dynamic arrays with the same structure can be assigned to each other and passed as parameters to subroutines.

To assign one dynamic array to a copy of another array, use the standard `copy` function:

```
a1 := Copy(a2);
```

Passing a dynamic array to a subprogram

A dynamic array is usually passed to a subroutine by value because the variable itself is already a reference:

```
procedure Squares(a: array of integer); begin for var i:=0 to
a.Length-1 do
    a[i] := Sqr(a[i]);
end;

begin var a := Arr(1,3,5,7,9); Squares(a);
end.
```

A dynamic array is passed by reference only in one case: if it is created or recreated within a subprogram. In particular, this must be done if SetLength is called for a dynamic array within a subroutine:

```
procedure Add(var a: array of integer; x: integer); begin
  SetLength(a,a.Length+1);
  a[a.Length-1] := x; end;

begin var a := Arr(1,3,5,7,9); Add(a, 666); writeln(a);
end.
```

Subroutinesпрограммы дfor working with дdynamic arrays

Subroutinesпрограммы дfor generating дdynamic arrays

Extension methods for sequences

methodsды for дdynamic arrays

Static arrays

Static array description

Static arrays, unlike dynamic ones, set their size directly in the type. The memory for such arrays is allocated immediately when describing them.

The static array type is constructed as follows:

array [*index type1, ... , type UHdeKcaN*] **of** *basic type*

The index type must be ordinal. Usually the index type is a range type and is represented as `a..y`, where `a` and y are constant

expressions of integer, character or enumerated type. For example:

```
type MyEnum = (w1,w2,w3,w4,w5);
  Arr = array [1...10] of integer; var a1,a2: Arr;
  b: array ['a'...'z',w2...w4] of string;
  c: array [1...3] of array [1...4] of real;
```

Initializing a static array

You can also specify the initialization of the array with values in the description:

```
var
  a: Arr := (1,2,3,4,5,6,7,8,9,0);
  cc: array [1..3,1..4] of real := ((1,2,3,4), (5,6,7,8),
(9,0,1,2));
```

Assigning a static array

Static arrays of the same type can be assigned to each other and the contents of one array will be copied to the other:

```
a1 := a2;
```

Output static array

The write procedure outputs a static array by enclosing the elements in square brackets and separating them with commas:

```
var a: Arr := (1,2,3,4,5,6,7,8,9,0);
var m := array [1..3,1..3] of integer := ((1,2,3),(4,5,6),
(7,8,9));
writeln(a); // [1,2,3,4,5]
writeln(m); // [[1,2,3],[4,5,6],[7,8,9]]
```

Passing a static array to a subprogram

When you pass a static array to a subroutine by value, you also copy the contents of the array - the actual parameter into the array - the formal parameter:

```
procedure p(a: Arr); // pass a static array by value - bad!
- • ••
p(a1);
```

This is extremely wasteful, so it is recommended to pass static arrays by reference. If the array does not change within a subroutine, it should be passed as a reference to a constant; if it changes, it should be passed as a reference to a variable:

```
type Arr = array [2...10] of integer;

procedure Squares(var a: Arr);
begin
  for var i:= Low(a) to High(a) do a[i] := Sqr(a[i]);
end;

procedure PrintArray(const a: Arr);
begin
  for var i: = Low(a) to High(a) do
    Print(a[i]) end;

var a: Arr := (1,3,5,7,9,2,4,6,8);

begin
  Squares(a);
  PrintArray(a);
end.
```

The Low and High functions are used to access the lower and upper bounds of the dimensionality of the one-dimensional array.

Records

A record is a set of elements of different types, each with its own name and called a record field.

Description of records

The record type in the classical Pascal language is described as follows:

record *field descriptions* **end**

where the field descriptions have the same appearance as the variable description section without the **var** keyword.

For example:

```
type
  Person = record Name: string; Age: integer;
  end;
```

Variables of record type

Record type variables store the values of all record fields in a contiguous memory block.

To access record fields, dot notation is used:

```
var p: Person;
begin
  p.Name := 'Ivanov';
  p.Age := 20;
  writeln(p); // (Ivanov,20)
end.
```

By default, the `write` procedure prints the contents of all the fields in the record in parentheses, separated by commas.

Methods and access modifiers for records

In **PascalABC.NET** you can define methods and properties within records, and use access modifiers. Thus, the description of a record in**PascalABC.NET** looks like this

```
record
    section1
    section2
     ... end
```

Each section has a view:

access modifier of declaration fields description or methods description and properties description

Theaccessmodifier in the first section may be missing, in which case the modifier `public` (all members are open) is implied.

For example:

```
type
  Person = record
  private
     Name: string;
     Age: integer;
  public
     constructor Create(Name: string; Age: integer);
     begin
       Self.Name := Name;
       Self.Age := Age;
     end;
     procedure Print;
  end;

procedure Person.Print;
begin
   writelnFormat('Name: {0} Age: {1}', Name, Age); end;
```

As with classes, methods can be described both inside and outside the record body. In the example above, the constructor is described inside the record, while the Print method is declared inside and described outside the record body. The constructor method is always named Create and is intended to initialize the fields of a record.

Initializing records

When describing a variable or constant of record type, you can use a record initializer (as in Delphi Object Pascal):

```
const p: Person = (Name: 'Petrova'; Age: 18);
var p: Person := (Name: 'Ivanov'; Age: 20);
```

Record constructors have the same syntax as classes. However, unlike classes, calling a record constructor does not create a new object in dynamic memory, but only initializes the record fields:

```
var p: Person := new Person('Ivanov',2 0),-
```

More traditionally, a record defines a normal method, traditionally named **`init`**, that initializes the fields of the record:

```
type Person = record - -- public procedure Init(Name: string;
Age: integer); begin
      Self.Name := Name;
      Self.Age := Age;
    end;
    -- end; - --
var p: Person;
p.Init('Ivanov',20);
```

Distinguishing records from classes

A list of the differences between records and classes is given below: 1. A record is a dimensional type (variables of record type are placed on the stack).

2. Records cannot be inherited; records cannot be inherited from either (note that records can, however, implement interfaces). In .NET, record type is implicitly assumed to be a descendant of `System.ValueType` and is implemented by struct-type.
3. If no access modifier is specified in the record, the modifier **`public`** (all members open) is implied by default, and **`internal`** is implied in the class.

Output a variable of record type

By default, the `write` procedure for a variable of record type outputs the contents of all its public properties and fields in parentheses, separated by commas. To change this behavior, you should override the `Tostring` virtual method of the `object` class, in which case it will be called when outputting the object.

For example:

```
type
  Person = record

    function ToString: string; override;
    begin
      Result := string.Format('Name: {0} Age: {1}', Name,
Age);
    end;
  end;

var p: Person := new Person('Ivanov',2 0),- writeln(p); //
Name: Ivanov Age: 20
```

Assigning and passing as parameters to subroutines

Since a record, unlike a class, is a dimensional type, assigning records copies the contents of the fields of one variable record to another:

```
d2 := d1;
```

Name equivalence of types is adopted for records: records that only have the same name can be assigned to each other and passed as parameters to subroutines.

To avoid copying, those records that contain more than one field are passed to subprograms by reference. If the record does not change within a subroutine, then a reference to a constant is used, if it changes, then a reference to a variable:

```
procedure PrintPerson(const p: Person);
begin
  Print(p.Name, p.Age);
end;

procedure ChangeName(var p: Person; NewName: string);
begin
  p.Name := Name;
end;
```

Equality comparison

Records of the same type can be compared for equality, and records are considered equal if the values of all fields are the same:

```
type Person = record name: string; age: integer;
end;

var p1,p2: Person;

begin
  p1.age := 20;
  p2.age := 20;
  p1.name := 'Ivanov';
  p2.name := 'Ivanov';
  writeeln(p1=p2); // True end.
```

Note

Unlike Delphi Object Pascal, **PascalABC.NET** has no variant entries.

Corteges

A tuple is a data type analogous to a record or class. Like a record or class, a tuple is a collection of elements of different types, but it is much easier to describe. In addition, the fields of a tuple have predefined names and are immutable: it is impossible to change the fields of a tuple after it has been created.

Type of motorcade

The following entry is used for the tuple type: `var t:`

```
(string,integer);
```

This entry is similar to a declaration of an enumerated type. If the parentheses are new names, it is an enumerated type, and if the type names, it is a tuple type.

Tuples are represented by the System.Tuple type of the .NET platform:

```
var t: System.Tuple<string,integer>;
```

However, this perception may change in the future.

Constructing values of tuple type

Values of tuple type can be constructed as a comma-separated enumeration of the values that make up a tuple. For example:

```
t := ('Ivanov',23);
```

For values constructed in this way, auto-drawing of type works:

```
var t1 := ('Ivanov',(5,3,4)); // tuple, second element
of which is the motorcade
```

Withdrawal of motorcades

As with the record output, the tuple output consists of parentheses and a comma-separated list of elements:

```
writeln(t); // (Ivanov,23)
writeln(tl); // (Ivanov,(5,3,4))
```

Access to tuple elements

The elements (fields) of a tuple are named Iteml, Item2, etc:

```
Print(t.Item1,t.Item2);
```

The elements of a tuple can also be referred to by an index:

```
Print(t[0],t[1]);
```

The indices must be constant expressions.

Once created, the tuple is unchanged: its fields cannot be changed:

```
t[1] := 20; // error
```

Tuple assignment (unpacking tuple into variables)

Values of tuple type can be unpacked into variables of corresponding types using tuple assignment: `var t := ('Ivanov',23); var name: string; var age: integer; (name,age) := t;`

The last assignment that uses a parenthetical variable enumeration on the left side of the assignment operator is called **a tuple assignment**. The compiler replaces the tuple assignment with several consecutive single assignments: `name := t[0]; age := t[1];`

Using a tuple assignment changes the programming style. For example, to swap the values of two variables a and b, just write the following tuple assignment:

```
(a,b) := (b,a);
```

In a tuple assignment, the number of elements on the right is greater than the number of variables on the left: `(a,b) := (1,2,3);`

A tuple assignment can be combined with a variable description:

```
(var a, var b) := (1,2);
```

or

```
var (a,b)     (1,2);
```

Using tuples in functions

Tuples allow you to package multiple values into one. This is useful if you need to pass multiple related values as a parameter or return multiple values from a function.

For example, a function that calculates the area and perimeter of a rectangle can be written in the form:

```
function SP(a,b: real) := (a*b,2*(a+b));
```

To take advantage of the result of such a function, it is convenient to use the unpacking of the tuple into variables with a description:

```
var (S,P) := SP(2,3);
```

Multitudes

A set is a set of elements of the same type. The elements of a set are considered disordered; each element can enter the set no more than once.

PascalABC.NET has **built-in** and .NET library sets: HashSet<T> (unordered, insertion, deletion, accessory operations rate is O(1)) and SortedSet<T> (ordered, insertion, deletion, accessory operations rate is O(log(n)).

The type of embedded set is described as follows: `set of` *basic type*

Any type can be a base type, including string and class types. The exception is pointer types.

For example:

```
type ByteSet = set of byte;
  StringSet = set of string;
  Digits = set of '0'...'9';
  SeasonSet = set of (Winter,Spring,Summer,Autumn);
  PersonSet = set of Person;
```

Elements of a basic type are compared for equality as follows: for simple types, strings, and pointers, the values are compared; for structured types and classes, the values of all elements or fields are compared. However, if fields are of reference type, only their addresses are compared (shallow comparison).

To construct an embedded set type value, the so-called set constructor is used, which has the form:

`[` *list of values* `]`

where the list can list either expressions of the basic type separated by commas, or (for ordinal types) their ranges in the form `a..y`, where `a` and y are expressions of the basic type. For example:

```
var bs: ByteSet := [1,3,5,20...25]; fios: StringSet :=
  ['Ivanov', 'Petrov', 'Sidorova'];
```

Values in the list can be absent, then the set is empty:

```
bs: = [];
```

An empty constant set `[]` is assignment-compatible with a set of any type. However, the type of the empty constant set is not

automatically output:

```
var bs: = []; // Error!
```

The set defined by the set constructor can have elements of different types, for example:

```
[1..4,5.5,'c','xyz',Winter..Autumn]
```

In this case, the most common type is calculated and declared as the base type of the set. For example:

```
[1...4,5.5] // set of real ['1','abc'] // set of string
[1,'1'] // set of object
```

Structural equivalence of types takes place for sets. Sets of integers and sets based on a type and its range subtype or based on two range types of one basic type are implicitly converted to each other. If, when s := s1 is assigned, the set s1 contains elements that are not in the range of values of the basic type for the set s, they are cut off.

For example:

```
var st: set of 3...9;

st := [1...5,8,10,12]; // st will contain values [3...5,8]
```

The `in` operation checks if an element belongs to a set:

```
If Wed in best days then ...
```

The operations `+` (union), `-` (difference), `*` (intersection), = (equality), `<> (inequality)`, <= (non-strict nesting), < (strict nesting), >= (non-strict contains) and > (strict contains) are defined for sets.

The `write` procedure outputs all elements of a set. For example,

```
Write(['Ivanov', 'Petrov', 'Sidorova']);
```

outputs `['Ivanov', 'Petrov', 'Sidorova']`, with the data sorted in ascending order, if possible.

You can use a `foreach` loop to search for all elements of the set, and the data are searched in some internal order: `foreach var s in fios do Write(s,' ');`

To add an element `x` to the set `s`, use the construction `s += [x]` or the standard procedure `include`: `Include(s,x)`. To remove an element `x` from the set `s, use` the construction `s -= [x]` or the standard procedure `Exclude`: `Exclude(s,x)` .

File Types

A file is a sequence of items of the same type stored on disk. In **PascalABC.NET** there are two types of files - *binary* and *text*. Text files store characters separated on lines by #13#10 (Windows) and #10 (Linux). The sequence of characters to jump to a new line is stored in the NewLine constant. Binary files, in turn, are divided into typed and untyped files.

To describe a text file, the standard name `text` is used, untyped files are of type **`file`**, and to describe a typed file, the construction **`file of`** *type elements* is used:

```
  var fl: file of real; f2: text;
     f3: file;
```

Pointers, reference types, and record types containing reference fields or pointers cannot appear as element types in a typed file.

Standard file procedures and functions are described in

- subprogramsпрограммы вводInput
- subroutinesпрограммы выводOutput
- Common subroutinesпрограммы дfor working with files
- Subroutinesпрограммы дfor working with text files
- Subroutinesпрограммы дfor working with дbinary files
- Subroutinesпрограммы дfor working with file names
- Common file methods
- Text file methods
- Typed file methods
- Binary methods
- methodsды Typed file extension

In addition, there are a number of classes in .NET related to working with files. These are found in the System.Text and System. 10.

Sequences

A sequence is a set of data that can be searched one by one in some order. Varieties of sequences include one-dimensional dynamic arrays `an array of T`, lists `List<T> ,` LinkedList<T> `, sets` HashSet<T> `and SortedSet<T>.`

The sequence type is constructed as follows:

The `sequence of` *TYPE Elements*

The sequences are **read-only**. If a sequence needs to be changed, a new sequence is generated and returned.

The type `sequence of T` is synonymous with the .NET `System.Collections.Generic.IEnumerable<T>` type`, and` sequence is synonymous with the type object that supports the interface `System.Collections.Generic.IEnumerable<T>.`

Initializing the sequence

The sequence is initialized using the стандаstandard functions `Seq`, `SeqGen`, `SeqFill`, `SeqWhile`, `SeqRandom`, `SeqRandomReal`, `ReadSeqInteger`, `ReadSeqReal`, `ReadSeqString`. For example:

```
var s: sequence of integer;
s := Seq(1,3,5);
s.Println;
s := SeqGen(1,x->x*2,10);
writeln(s);
```

Storing the sequence

The sequence is **not stored entirely in memory**.
The sequence elements are generated algorithmically and are returned one at a time as they are traversed.

Thus, in the code

```
var s := SeqFill(1,10000000);
writeln(s.Sum());
```

the second line will take most of the execution time, and the execution of the first line will be reduced only to remembering the algorithm for generating the sequence in the s variable.

Connecting sequences

Two sequences of the same type can be joined by the + operation, with the second sequence added to the end of the first. For example:

```
Seq(1,2,3) + Seq(5,6,7)
Seq(1,2,3) + Arr(5,6,7)
```

In addition, you can join a sequence of some type to a value of that type with the + operation as the first or last element of the sequence, for example:

```
Seq(1,2,3) + 5
3 + Seq(5,6,7)
3 + Seq(5,6,7) + 9
```

The + operation is a shortened version of the Concat operation.

The multiplication operation by a number is also available for sequences:

```
Seq(1,2,3) * 3
```

means repeating the sequence 1 2 3 three times: 1 2 3 1 2 3 1 2 3

Cycle by sequence

The sequence elements can be bypassed with a **`foreach`** loop:

```
foreach var x in s do if x>2 then Print(x);
```

Assignment compatibility

A sequence variable with elements of type `t` can be assigned a one-dimensional array `of t`, a list `List<T> , a` linked list LinkedList<T> `,` a set HashSet<T> `or sortedset<T>` , as well as an object of any class that supports the `System.Collections.Generic.IEnumerable<T>.`

Standard subprograms and methods

For sequences available:

- Sequence processing methods
- Subroutinesпрограммы дfor sequence generationдовательностей
- Subroutinesпрограммы дfor generating infinite sequences
- methodsды Extension дfor sequencesдовательностей

Indicators

A pointer is a memory cell that stores an address. In **PascalABC.NET** pointers are divided into ***typed*** (contain the address of a memory cell of a given type) and ***untyped*** (contain a memory address not associated with data of any particular type).

The type of the pointer to type `t` is in the form of `^t`, for example:

```
type pinteger = Ainteger;
var p: Arecord r,i: real end;
```

A bestiary pointer is described with the word `pointer`.

To access a memory cell whose address stores a typed pointer, ***the dereferencing operation*** is used:

```
var
  i: integer; pi: Ainteger; - --
pi := @i; // the pointer was assigned the address of the i variable
piA := 5; // the variable i is assigned 5
```

The dereferencing operation cannot be applied to a type-free pointer.

A typed pointer can be implicitly converted to a typeless pointer:

```
var p: pointer; pr: Areal; - --
p := pr;
```

The inverse transformation can also be performed implicitly:

```
pr := p; prA := 3.14;
```

Pointers can be compared on equality (`=`) and inequality (`<> ). To` mark the fact that the pointer points nowhere, the standard constant **`nil`** (null pointer) is used : `p :=` **`nil`**.

Warning! Due to the peculiarities of the .NET platform, the `t` typed pointer type must not be referential or contain referential types at some level (for example, pointers to records that have one of the fields with a referential type are prohibited). The reason for this restriction is simple: pointers are implemented by unmanaged code that is not managed by the garbage collector. If the memory the pointer points to contains references to managed variables, they become invalid after the next garbage collection. Exceptions are dynamic arrays and strings that are handled in a special way. That is, you can make pointers to records that contain strings and dynamic arrays as fields.

Procedural type

A type designed to store references to procedures or functions is called a procedural type, and a variable of this type is called a procedural variable. The main purpose of procedural variables is to store and indirectly call actions (functions) during program execution and pass them as parameters.

Description of the procedural type

The description of the procedural type matches the header of the corresponding procedure or function without the name. For example:

```
type ProcI = procedure (i: integer);
  FunI = function (x,y: integer): integer;
```

A procedure variable can be assigned a procedure or function with a compatible type, for example:

```
function Mult(x,y: integer): integer;
begin
  Result := x*y;
end;

var f: FunI := Mult;
```

You can also assign a lambda expression to a procedure variable with the appropriate number of parameters and type of return value:

```
var f2: FunI := (x,y) -> x+2*y;
```

You can then call the procedure or function through this procedure variable using the normal call syntax:

```
write(f(2)); // 8 write(f2(3)); // 10
```

Synonyms for procedural types

A number of synonyms are defined in the system module for the most common proceduretypes. Here are examples of their use:

```
var f3: IntFunc := x -> 2*x-1;
var f4: Func<integer,real> := x -> 2.5*x;
var f3: Action<real> := x -> write(x,' ');
var pr: Predicate<string> := s -> s.Length>0;
```

Abbreviated constructions for procedural types

Abbreviated constructs are also defined for procedural types:

```
   () -> T; //      parameterless function returning T
   T1 -> T; //      function with parameter T1, returning
   T
   (T1,T2) -> T // function with parameters T1 and T2,
   returning T
   (T1,T2,T3) -> T // function with parameters T1, T2 and T3,
   returning T, etc.
                                   () -> (); //procedure
withoutparameters
   T1                    -> (); //procedure    T1
   (T1,T2)               -> ()//procedure     T1 and T2
   (T1,T2,T3)           -> () //procedure parameters    T1,   T2
and            T3
     etc.
```

Abbreviated constructs cannot describe procedural variables with parameters passed by reference. Structural equivalence of types is accepted for procedural variables: it is possible to assign to each other and pass as parameters procedural variables that coincide in structure (types and number of parameters, type of return value).

Procedural variables as parameters

Usually procedural variables are passed as parameters to implement *a callback* - a subprogram call through a procedural variable passed as a parameter to another subprogram:

```
procedure forall(a: array of real; f: real->real); begin
  for var i := 0 to a.Length-1 do
    a[i] := f(a[i]);
end;

forall(a,x->x*2); // multiply array elements by 2
forall(a,x->x+3); // increase array elements by 3
```

A procedure variable can store a null value, which is set by the constant `nil`. Calling a subroutine with a null procedural variable causes an error.

The += and -= operations for procedural variables

Procedural variables are implemented through .NET delegates. This means that they can store multiple subroutines. The `+=` and `-=` operators are used to add/disconnect subroutines:

```
pl += mult2;
pl += add3;
forall(a,p1);
```

The subroutines in this case are called in attachment order: multiplication first, then addition.

Disconnecting unattached subroutines does not perform any action:

```
pl -= print;
```

Besides, you can attach/unattach class and instance methods of classes to a procedural variable. In the latter case, a procedural variable in the object fields remembers some state, which changes between calls to the method associated with this procedural variable.

Example

```
type
  A = class
    x0: integer := 1;
    h: integer := 2;
    procedure PrintNext;
    begin
      Print(x0); x0 *= h; end;
  end;
begin
  var p: procedure;
  var al := new A();
  p := al.PrintNext;
  for var i:=1 to 10 do p;
  // 1 2 4 8 16 32 64 128 256 512 end.
```

This behavior is much easier to implement by capturing a variablewith a lambda expression:

```
begin
  var x0 := 1;
  var p: Action0 := procedure -> begin Print(x0); x0 *= 2
end;
  for var i:=1 to 10 do p;
end.
```

Equivalence and compatibility of types

Type Matching

Types `TI` and `T2 are` said to coincide if they have the same name or are defined in the **`type`** section as `TI` = `T2`. Thus, in the descriptions

```
type IntArray = array [1...10] of integer;
  IntArrayCopy = IntArray;
var
  al: IntArray;
  a2: IntArrayCopy;
  b1,c1: array [1...15] of integer;
  b2: array [1...15] of integer;
```

variables `a1` and `a2` and variables `b1` and `c1` have the same type, and variables y1 and y2 have different types.

Equivalence of types

Types `T1` and `T2 are` said to be equivalent if one of the following conditions holds:

1. `T1` and `T2` are the same
2. `T1` and `T2` are dynamic arrays with matching element types
3. `T1` and `T2` are pointers with matching basic types
4. `T1` and `T2` are sets with matching basic types
5. `T1` and `T2` are procedural types with the same formal parameter list (and return value type for functions)

If types are equivalent only if their names coincide, then we say that there is a ***name equivalence of types***. If types are equivalent if their names are the same, they are said to be ***structurally equivalent***. Hence, in **PascalABC.NET we have** name equivalence for all types except dynamic arrays, sets, typed pointers and procedural types, for which we have structure equivalence. Only if types `T1` and `T2` are equivalent, an actual parameter of type `T1` can be substituted for a formal parameter of type `T2`.

Type compatibility

Types `T1` and `T2` are said to be compatible if one of the following conditions holds:

1. `T1 AND T2` are equivalent to
2. `T1` and `T2 belong to` integer types
3. `T1` and `T2 belong to` the real types
4. One of the types is a subrange of the other, or both are subranges of some type
5. `T1` and `T2` are sets with compatible basic types

Type Compatibility by Assignment

A value of type `T2` is said to be assignable to a variable of type `t1` or type `T2` is assignment-compatible with type `TI` if one of the following conditions holds:

1. `TI` and `T2` are compatible
2. `TI` - real type, `T2` - integer type
3. `TI` - string type, `T2` - character type
4. `T1` - `pointer`, `T2` - typed pointer
5. `TI` - pointer or procedural variable, `T2=nii`
6. `TI` - procedure variable, `T2` - procedure or function name with the corresponding parameter list
7. `TI`, `T2` are class types, one of which is a descendant of the other. Since in **PascalABC.NET** all types except pointers are descendants of the `object` type, you can assign a value of any type (except for pointers) to a variable of the `Object` type
8. `TI` is the interface type, `T2` is the class type that implements the interface

If type `T2` is assignment-compatible with type `TI`, it is also said that type `T2` ***does not bring in*** type `TI`.

Mapping to .NET types

The **PascalABC.NET** standard types are implemented by the .NET class library types. Here is a table that lists the **PascalABC.NET** standard types and the .NET types.

Type PascalABC.NET	Type .NET
int64	System.Int64
uint64	System.UInt64
integer, longint	System.Int32
longword, cardinal	System.UInt32
BigInteger	System.BigInteger
smallint	System.Int16
word shortint	System.UInt16
	System.SByte
byte boolean	System.Byte
	System.Boolean
	System.Double
real double	System.Double
char string	System.Char System.String
object	System.Object
array of T	T[]
record	struct

Expressions and operations: an overview

An expression is a construction that returns a value of some type. Simple expressions are variables and constants. More complex expressions are constructed from simple ones using operations, function calls, and brackets. Data to which operations are applied are called *operands*.

The following operations are available in **PascalABC.NET:** `@`, **`not`**, `!`, `*`, `/`, **`div`**, **`mod`**, **`and`**, **`shl`**, **`shr`**, `+`, `-`, **`or`**, **`xor`**, `=`, `> , < , <> , <=, >=, as, is,` in, `as` well as the new operation and the type cast operation. The operations `@`, `-`, `+`, `!`, **`not`**, type conversion and `new` are unary (have one operand), the others are binary (have two operands), the operations `+` and `-` are both binary and unary.

PascalABC.NET operations help

- Arithmetic operations
- Logical operations
- Comparison operations
- String operations
- Bitwise operations
- Operations with sets
- Explicit type conversion operation
- The operations is and as
- Operation new
- Operation @ get address
- Operations with pointers
- The typeof and sizeof operations
- Cuts
- Conditional operation

The order in which operations are performed is determined by their priority. In PascalABC.NET there are four levels of priority of operations, defined in the priority table.

- Table of operations priorities

A number of operations for user-defined types can be overloaded. You can also overload operations for .NET-types if they have not been overloaded.

- Overload operations

Arithmetic operations

Arithmetic operations include binary operations `+`, `-`, `*`, `/` for real and integer numbers, binary operations **`div`** and **`mod`** for integers, and unary operations `+` and `-` for real and integer numbers. The type of expression `x` *op* `y`, where *op* is the sign of the binary operation `+`, `-`, or `*`, is determined from the following table:

	shortint	**byte**	**smallint**	**word**	**integer**	**longword**	**int64**	**uint64**
shortint	integer	integer	integer	integer	integer	int64	int64	uint64
byte	integer	integer	integer	integer	integer	longword	int64	uint64
smallint	integer	integer	integer	integer	integer	int64	int64	uint64
word	integer	integer	integer	integer	integer	longword	int64	uint64
integer	integer	integer	integer	integer	integer	int64	int64	uint64
longword	int64	longword	int64	longword	int64	longword	uint64	uint64
int64	int64	int64	int64	int64	int64	uint64	int64	uint64
uint64	uint64	uint64	uint64	uint64	uint64	uint64	uint64	uint64
BigInteger	BigInteger	BigInteger	BigInteger	BigInteger	BigInteger	BigInteger	BigInteger	BigInteger
single	single	single	single	single	single	single	single	single
real	real	real	real	real	real	real	real	real
decimal	decimal	decimal	decimal	decimal	decimal	decimal	decimal	decimal

That is, if the operands are integers, the result is the shortest integer type required to represent all the values we get.

When performing a binary operation with `uint64` and a signed integer, the resulting type will be `uint64`, and an overflow may occur without causing an exception.

For the `/` operation this table is corrected as follows: the result of dividing any integer by an integer is of type `real`.

The same rules apply to **`div`** and **`mod`** operations, but the operands can only be integers. The rules for calculating **`div`** and **`mod`** operations are as follows:

- `x` **`div`** `y` is the result of integer division of `x` by `y`. More precisely, `x` **`div`** `y = x / y`, rounded to the nearest integer towards 0; `x` **`mod`** `y` is the remainder of an integer division of `x` by `y`. More precisely, `x` **`mod`** `y = x - (x` **`div`** `y) * y`.

The unary arithmetic operation `+` for any integer type returns that type. Unary arithmetic operation `-` returns for integer types smaller or equal to `integer`, for `longword` and `int64` - value of `int64` type, for `uint64` unary operation is not applicable, for `single` and `real` types - types `single` and `real` respectively. That is, the same result is the shortest type required to represent all obtained values.

Logical operations

Binary operations include the binary operations **`and`**, **`or`** AND **`xor`**, and the unary operation **`not`**, which have operands of the `boolean` type and return values of the `boolean` type. These operations obey the standard rules of logic: `a` **`and`** is true only when `a` and b are true, `a` **`or`** is true only when either `a` or b is true, `a` **`xor`** is true only when only one of `a` and b is true, **`not`** `a` is true only when `a is` false.

Expressions with **`and`** and **`or`** are calculated according to ***a short scheme***:

in the expression `x` **`and`** `y` if `x` is false, then the whole expression is false, and `y` is not evaluated;
in the expression `x` **`or`** `y` if `x` is true, then the whole expression is true, and `y` is not evaluated.

Bitwise operations

Bitwise operations include binary operations **and**, **or**, **not**, **xor**, **shl**, **shr**. They perform bitwise manipulations with operands of integer type. The resulting type for **and**, **or**, **xor** will be the smallest integer, including all possible values of both operand types. For **shl**, **shr** the resulting type is the same as the left operand type, for **not the** operand type. Bitwise operations are performed as follows: With each bit (0 is taken as False, 1 as True) a corresponding logical operation is performed. For example:

```
00010101 and 00011001 = 00010001
00010101 or 00011001 = 00011101
00010101 xor 00011001 = 00001100 not 00010101 = 11101010
```

(operands and result are in binary form).

The **shl** and **shr** operations have the form:

```
a shl n a shr n
```

where n is a positive integer, a is an integer.

`a` **shl** `n is` a positive integer obtained from the binary representation of the number a by shifting it to the left by n positions. Positions added to the right are filled with zeros. `a` **shr** `n is` a positive integer obtained from the binary representation of a by shifting it to the right by n positions.

For example:

```
3 shl 2 = 12 12 shr 2 = 3
```

since 3=112, after a shift to the left by 2 positions 112 is converted to 11002=12, and 12=11002 after a shift to the right by 2 positions is converted to 112=3.

Comparison operations

The comparison operations `<` , `>` , `<=`, >=, `=`, <> `return` the value of boolean type and apply to operands of simple type and to strings. The `=` and `<>` operations also apply to all types. By default, values are compared for dimensional types, and references are compared for reference types. You can override this behavior by overloading the `=` and `<>` operations. Similarly, you can overload all comparison operations for record types and classes entered by the user.

String operations

All comparison operations `< , > , <=, >=, =,` <> apply to strings. Comparison of strings on inequality is done **lexicographically**: `si < s2` if for the first non-matching character with number i si[`i]<s2[i]` `or` all characters of strings coincide, but si is shorter than s2.

In addition, the concatenation (merge) `+` operation is applicable to strings and characters, and its result has a string type.

For example, `'a'+'b'='ab'`.

The operation `+=` is also applicable to strings:

```
s += si; // s := s + si;
```

A string can be added to a number, with the number being preconverted to a string representation:

```
s := 'Width: '+i5'; // s = 'Width: i5'
s := 20.5+'; // s = '20.5'
s += i; // s = '20.5i'
```

The operation `*` is defined over strings and integers: `s*n` and `n*s` means the string formed from the string `s` repeated n times:

```
s := '*'*i0; // s = '**********
s := 5*'ab' // s = 'abababababab
s := 'd'; s *= 3; // s = 'ddd'
```

An **`in`** operation is defined over strings to check if a substring is embedded in a string:

```
'bc' in 'abcde' // True 'c' in 'abd' // False
```

Operations with pointers

For all pointers the comparison operations `=` and `<>` are applicable.

The `dereferencing` ***operation*** applies to typed pointers: if `p` is a pointer to type `t`, then `p1` is an element of type `t` pointed to `by p`. Pointers `of pointer type` cannot be dereferenced.

Operations with sets

The operations `in` (belonging) `+` (union), `-` (difference) and `*` (intersection) as well as the operators `+=`, `-=` and `*=` apply to sets with basic elements of the same type:

```
                var s1,s2,s: set of byte;
            begin
              s1 := [1..4];
              s2 := [2..5];
              s := s1 + s2; // s = [1...5]
              s := s1 - s2; // s = [1]
              s := s1 * s2; // s = [2...4]
            // s += s1 is equivalent to s := s
            // s -= s1 is equivalent to s := s
            // s *= s1 is equivalent to s := s
              s += [3..6];  // s = [2..6]
              s -= [3];     // s = [2,4..6]
              s *= [1...5];     // s =
Print(3 in s1); // True
    end.
```

Comparison operations = (equality), `<>` (inequality), <= (not strictly nested), < (strictly nested), >= (not strictly contains) and > (strictly contains) are also applicable to sets with basic elements of the same type:

```
[1..3] = [1,2,3]
['a'...'z'] <> ['0'...'9']
[2..4] < [1..5]
[1..5] <= [1..5]
[1..5] > [2..4]
[1..5] >= [1..5]
```

But it is not true that `[1...5] < [1...5]`.

Finally, the operation `in` determines whether an element belongs to the set: `3 in [2..5]` will return `True`, `1 in [2..5]` will return `False`.

The types HashSet<T> and SortedSet<T> also belong to sets.
The same operations apply to variables of this type as to built-in sets:

```
    begin
      var s1 := new HashSet<integer>;
      s1 := HSet(Range(1,4));
      var s2 := HSet(Range(2,5));
     var          s:=si+s2;//s=              [1...5]
     var          s:=si-s2;//s=         [1]
     var          s:=si*s2;//s=              [2...4]
```

```
Print(3 in si); // True end.
```

Operation @

The `@` operation is applied to a variable and returns its address. The result type is a typed pointer to the variable type. For example:

```
var r: real; pr: Areal := @r;
```

The operations is and as

The `is` operation is designed to check whether a class variable has the specified dynamic type. The `as` operation allows you to safely convert a variable of one class type to another class type (as opposed to explicitly casting the class type).

The operation `is of` the form:

```
a is ClassType
```

and returns `True` if `a belongs to` the *ClassType* class or one of its descendants.

For example, if `Base` and `Derived` are classes, and `Derived` is a descendant of `Base`, and the variables b and d are `Base` and `Derived` respectively, then the expressions `b is Base` and `d is Base` return `True`, and b `is Derived` returns `False`.

The `as` operation has the form:

```
a as ClassType
```

and returns a reference to an object of *ClassType* type if the conversion is possible, otherwise it returns `nil`.

For example, in the program

```
type Base = class end;
  Derived = class(Base) procedure p; begin end;
  end;
var b: Base;
begin
  b := new Base;
  writeeln(b is Derived);
  b := new Derived;
  writeeln(b is Derived);
end.
```

the first time it displays `False`, the second time it displays `True`.

The `is` AND `as` operations are used to manipulate a base class variable that contains a derived class object.

1 way.

```
if b is Derived then Derived(b).p;
```

2 way.

```
var d: Derived := b as Derived;
```

```
d.p;
```

Operation new

The **new** operation has the form:

newClassName (*Constructor Parameters*)

It calls the Class *Name* constructor and returns the created object.

For example:

```
type My = class constructor Create(i: integer); begin
    end;
  end;

var m: My := new My(5);
```

The equivalent way to create an object is to call the Object Pacal style constructor:

```
var m: My := My.Create(5);
```

It is easier to create a class object when a variable is initialized using ***type auto-definition***:

```
var m := new My(5);
```

A record can also have constructors defined, which are called in the same way. But unlike a class, calling a record constructor does not allocate memory (it is already allocated) and only fills field values.

The typeof and sizeof operations

The **sizeof** (*type name*) operation returns for that type its size in bytes.

The **typeof**(*type name*) operation returns an object of the system.Type class for that type. Here is an example of using typeof:

```
type Base = class ... end;
  Derived = class(Base) ... end;
var b: Base := new Derived;
begin
  writeln(b.GetType = typeof(Derived)); end.
```

Explicit type conversion operation

The explicit type conversion operation looks like

NameType (*expression*)

and allows you to convert an expression to a type *called NameType*. An expression type and a type named *NameType* must both belong to either an ordinal type or a pointer type, or one type must be an heir of the other, or the expression type must support an interface with the *NameType* name. In the case of pointers, it is forbidden to convert a typed pointer to a pointer type of another type.

Example.

```
type pinteger = ^integer;
  Season = (Winter,Spring,Summer,Autumn);
var i: integer;
    b: byte;
    p: pointer := @i;
    5: Season;
begin
  i := integer('z');
  b := byte(i);
  1 := pinteger(p);
  s := Season(1);
end.
```

When dimensional types are converted to Object type, packing takes place.

Example.

```
var i: integer := 5;
begin
  var o: Object := Object(i); end.
```

word operation

The conditional operation (**first form**) has the following form:

condition `?` *expression1* `:` *expression2*

If the condition is satisfied, the result is the value of expression1, otherwise the value of expression2.

For example:

```
var min := a < b ? a : b;
```

It is easy to see that the conditional operation allows in simple cases not to use the conditional operator. For example, the previous code for real a and b is equivalent to the following:

```
var min: real;
if a<b then min := a else min := b;
```

There is also a **second form of conditional operation** in Pascal syntax:

`if` *condition* **`then`** *expression1* **`else`** *expression2*

For example:

```
var min := if a < b then a else b;
```

It is recommended to use the second form of conditional operation.

Limitation. The second form of a conditional operation cannot be used as a top-level expression in lambda. The following code is wrong:

```
var l: real -> real := x -> if x<0 then 0 else x;
```

To solve this problem, you can put a conditional operation in brackets:

```
var l: real -> real := x -> (if x<0 then 0 else x);
```

Cuts

A slice is a set of elements of a dynamic array, List<T> or string, arranged sequentially or with some step.

The cut looks like:

```
a[from:to]
```

or

```
a[from:to:step]
```

or

```
a?[from:to]
```

or

```
a?[from:to:step]
```

and contains a copy of the elements of the original container in the range [`from`, `to`) with `the` step step. The entry[`from`, `to`) means that the element with the index `from` is included in the range, and the element with the index `to` is not included. The values `from`, `to`, `step` must be integers.

If `step` is not specified, then it is assumed that `step=1`, i.e. the elements are arranged continuously.

The slice type is the same as the container type: the array element slice is an array, the string slice is a string and the slice for List<T> also belongs to the List<T> type.

For normal slices an exception is generated in the following cases. If the `from` index is out of bounds, an exception is generated during program execution. The `to` index can be out of bounds by 1, i.e. in the range [-1..a.Count] for dynamic arrays and lists List<T> and in the range [0..s.Length+1] for strings, otherwise an exception also arises.

If `step=0`, an exception is generated for all slices.

Slices of type `a?` are called *safe slices* and do not generate exceptions in all other cases. They work as follows. First, the container is considered infinite in both directions, and the slice selects certain elements in it. After that, only those elements that belong to the original container are left in the slice.

Here are some examples:

```
var a := Arr(0,1,2,3,4,5,6);
Println(a[2:5]); // [2,3,4]
Println(a[2:a.Length]); // [2,3,4,5,6] Println(a[2:1000]);
// Exception!
Println(a?[2:1000]); // [2,3,4,5,6]
Println(a[2:7:2]); // [2,4,6]
Println(a[2:0]); // empty array var s := '0123456';
Println(s[2:5]); // 123
Println(s[0:2]); // Exception!
Println(s?[0:2]); // 12
Println(s?[-2::2]); // 135
```

```
Println(s[2:s.Length]); // 12345
```

The step `step` can be negative. For example:

```
var a := Arr(0,1,2,3,4,5,6);
Println(a[5:2:-1]); // [5,4,3]
Println(a[2:5:-1]); // empty array Println(a[a.Length-1:-1:-
2]); // [6,4,2,0]
```

In a slice entry, the `from` or `to` expression can be skipped, in which case all elements from the corresponding side get into the slice. If `step>0`, then missing `from` is assumed to be equal to the index of the first element, missing to is assumed to be equal to the index of the last element + 1. If `step<0`, then the missing `from is assumed to be equal to the index of` the last element, and the missing to `is assumed to be equal` to the index of the first element - 1.

For example:

```
var s := '0123456';
Println(s[2:]); // 23456
Println(s[:4]); // 012
Println(s[::-1]); // 6543210
```

Related to slices is the Slice method defined for dynamic arrays, List<T> lists, strings and sequences. However, the meaning of its parameters is different: a.Slice(from,step,count).

Priority of operations

Priority determines the order of operations in an expression. The operations with the highest priority are executed first. Operations with the same priority are performed from left to right.

Table of operations priorities

`**`	1 (highest)
`@`, **`not`**, `^`, `+`, `-` (unary), `new`	2
`*`, `/`, **`div`**, **`mod`**, **`and`**, **`shl`**, **`shr`**, **`as`**, **`is`**	3
`+`, `-` (binary), **`or`**, **`xor`**	4
`..`	5
`=` , `<>` , `<` , `>` , `<=`, `>=`, `in`	6
`?:`	7 (lowest)

Brackets are used to change the order of operations in expressions.

Operators: overview

The following operators are defined in **PascalABC.NET.**

- Assignment operators
- Compound operator
- Variable description operator
- Operator of the cycle for
- The foreach loop operator
- The while and repeat loop operators
- Conditional if statement
- Variant case operator
- Procedure call operator
- The try except statement
- The try finally operator
- Operator raise
- The break, continue, and exit operators
- The goto operator
- Operator lock
- Operator with
- Empty operator

Assignment operator

The assignment operator has the form:

variable `:=` *expression*

A variable can be a simple variable, a dereferenced pointer, a variable with indexes, or a variable component of the record type. The `:=` symbol is called an assignment character. The expression must beassignment-compatible with the variable.

The assignment operator replaces the current value of a variable with the value of an expression.

For example:

```
i := i + 1; // increments i by 1
```

PascalABC.NET also defines assignment operators with `+=`, `-=`, `*=`, `/=`. For numeric types these operators are described here. In addition, use of operators `+=` and `*=` for strings is described here and operators `+=`, `-=` and `*=` for sets are described here. Their actions for procedural variables are described here.

The operators `+=`, `-=`, `*=`, `/=` have the following meaning: `a #= y` means `a := a # y`, where `#` is the sign of the operation `+`, `-`, `*`, `/`.

For example:

```
a += 3; // increase a by 3
b *= 2; // increase b by 2 times
```

The operator `/=` is not applicable if the expression on the left is an integer.

The operators `+=`, `-=`, `*=`, `/=` can also be used with class properties of the corresponding types on the left side.

Compound operator (block)

A compound operator is designed to combine several operators into one. It has the form:

```
begin
  end operators
```

In **PascalABC.NET** a compound operator is also called ***a block***. (In Pascal, traditionally a block is a section of definitions followed by a compound operator; **PascalABC.NET has gone one step** further by allowing variables to be described directly within the compound operator.)

Operators are separated from each other by "`;` ". The keywords `begin` and `end` that border operators are called ***operator brackets***.

For example:

```
s := 0;
p := 1;
for var i:=1 to 10 do begin
  p := p * i; s := s + p end
```

`End` can also be preceded by "`;` ". In this case, the last operator before `end` is assumed to be an empty operator that does not perform any action.

In addition to operators, there may beintra-block variable descriptions in a block:

```
begin
  var a,b: integer; var r: real;
  readln(a,b);
  x := a/b; writeln(x);
end.
```

Empty operator

The empty operator does not include any characters, does not perform any actions and is used in two cases:

1. To use the ";" character after the last statement in a block:

```
begin
  a := 1;
  b := a;
end
```

Since in Pascal the ";" separates operators, in the above code it is assumed that there is an empty operator after the last ";". Thus, the ";" before end in a block can either be placed or not.

2. For marking the position following the last operator in the block:

```
label a;
begin goto a;
  x := 1;
a: end
```

word operator

The conditional operator has a *full* and *a short* form.

The full form of the conditional operator is as follows:

```
if condition then onepamopl else operator2
```

Some logical expression is specified as a condition. If the condition is true, then *onepamopl* is executed, otherwise *operator2* is executed.

The short form of the conditional operator looks like this:

```
if condition then operator
```

If the condition is true, *the operator* is executed, otherwise the program passes to the next operator.

In the case of a design of the form

```
If condition1 then
  if condition2 then operator1
  else operator2
```

`else` always refers to the nearest preceding `if` statement, for which the `else` branch is not yet specified. If the previous example requires `else` to refer to the first `if` operator, then a compound operator must be used:

```
If condition1 then
begin
  if condition2 then operator1 end
else operator2
```

For example:

```
if a<b then min := a else min := b;
```

Variable description operator

In **PascalABC.NET** you can describe variables inside a compound begin-end statement in a special variable description statement. Such descriptions are called intra-block descriptions.

The intra-block description has one form:

```
var list of names: type ;
```

or

```
var name: type := expression ;
```

or

```
var name: type = expression; // For compatibility with Delphi
```

or

```
var name := expression ;
```

The names in the list are listed, separated by commas. For example:

```
begin
  var a1,a2,a3: integer;
  var n: real := 5;
  var s := ' ';
  - • ••
end.
```

In the latter case, the type of the variable is autodetected by the type of the expression in the right part. Auto typing is actively used when initializing a variable by calling a constructor or function that returns an object:

```
begin
  var l := new List<integer>;
  var a := Seq(1,3,5); // type a is output by the type of
value returned Seq: array of integer end.
```

Quartet assignment with variable description

You can combine tuple assignment (unpacking a tuple into variables) with variable description:

```
var t := (1,2);
(var a, var b) := (1,2);
```

or

```
var (a,b)     (1,2);
```

Unpacking a tuple into variables is often used when returning a tuple function:

```
function SP(a,b: real)     (a*b,2*(a+b));

var (S,P) := SP(2,3);
```

Initialization by lambda expression

Auto-type in description is not possible when initializing a variable with a lambda-expression:

```
// var f := x -> x*x; // you can't do that! var f :
Func<integer,integer> := x -> x*x;
```

Intra-block descriptions are used to avoid cluttering the descriptions section with descriptions of auxiliary variables. In addition, intra-block descriptions allow you to enter variables exactly when they are first needed. Both of these factors greatly increase the readability of the program.

Selection operator

The select operator performs one action out of several, depending on the value of some expression called *a switch*. It has the following form:

```
case switch of
  selection list 1: onepamopl;
  ...
  selection list N: onepamopN ;
  else a list of end statements;
```

The switch is anordinal or string type expression, and the selection lists contain constants of an assignment-compatible type. As in the `if` statement, the `else` branch may be missing.

The `case` statement works as follows. If the current switch value is found in one of the selection lists, the operator corresponding to that list is executed. If the switch value is not found in any list, the list of `else` branch operators is executed, or, if there is no `else` branch, the `case` operator does not perform any action.

The selection list consists either of a single constant or, for an enumerated type, of a range of values of the form `a..y` (the constant `a` must be smaller than the constant y); you can also list several constants or ranges separated by commas. For example:

```
Country of
  'Russia': Capital := 'Moscow';
  'France': Capital := 'Paris';
  'Italy': Capital := 'Rome';
  else Capital := 'No country in the database'; end;
case DayOfWeek of
  1..5: writeln('Weekday');
  6,7: writeln('Day off');
end;
```

The selection lists must not overlap. For example, the following fragment

```
case i of
  2.5: write(1) ;
  4..6: write(2);
end;
```

will cause a compilation error.

Operator of the cycle for

The **for** loop operator has one of two forms:

for *variable* := *start value* **to** *end value* **do** *operator*

or

for *variable* := *start value* **downto** *end value* **do**
operator

Alternatively, a variable can be described directly in the loop header:

for *variable*: *type* := *start value* **to** *or* **downto** *end value* **do**
operator

or

for var *variable* := *start value* **to** *or* **downto** *end value* **do**
operator

In the latter case, ***auto-definition of the*** variable ***type*** by the type of its initial value is used. In the last two cases, the scope of the declared variable extends to the end of the loop body, which in this case forms an implicit block. Outside the loop body, such a variable is not available, so the next loop can use a variable with the same name:

```
for var i := 1 to 10 do Print(i);
for var i := 1 to 5 do Print(i*i);
```

The text from the word **for** to the word **do**, inclusive, is called ***the loop header***, and the statement after **do** is called ***the loop body***.
The variable after the word **for is** called ***the loop parameter***. For the first form of the loop with the **to** keyword, the loop parameter changes from the initial value to the final value, increasing each time by one, and for the second form with the **downto** keyword, decreasing by one. The loop body is executed for each value of the variable-parameter.
A single repetition of the loop body is called ***a loop iteration***. The value of the loop parameter is considered undefined after the loop ends.

The loop parameter variable can be of anyordinal type. The start and end values must be assignment-compatible with the loop parameter

variable.

For example:

```
var en: (red,green,blue,white);
-- for en := red to blue do write(Ord(en):2);
for var c := 'a' to 'z' do write(c);
```

If for a loop **for** ... **to the** initial value of the loop variable is greater than the final value or for the **for** ... **downto the** initial value of the loop variable is less than the final value, the loop body will not be executed at all.

If the loop is used in a subprogram, the variable parameter of the loop must be described as local.
The best solution in PascalABC.NET is to describe the variable in the loop header.

Changing the variable-parameter of a loop within a loop is a logical error. For example, the following fragment with a nested **for** statement is an error:

```
for i := 1 to 10 do i -= 1;
```

Loop statement

The **loop** statement has the form:

loop *expression* **do** *operator*

The expression must be of integer type and indicates the number of times the loop body is repeated. If the value of the expression <= 0, the loop body is not executed at all.

A loop is used in simple situations where the loop body does not depend on the loop iteration number:

```
loop 5 do
  Print(1);
var x := 1; loop 5 do begin
  Print(x);
  x += 2;
end;
```

The foreach loop operator

The **foreach** loop operator has one of the following forms:

foreach *variable* **in** *container* **do** *operator*

or

foreach *variable*: *type* **in** *container* **do** *operator*

or

foreach var *variable* **in** *container* **do** *operator*

A container can be a dynamic array, a string, a set, or any container that satisfies the `IEnumerable` or `IEnumerable<T>` interface (for example, List<T>, Dictionary<Key,vaiue>, etc.). The loop variable must have the **same** type as the container elements (if the container satisfies the `IEnumerable` interface, then it is the `object` type). In the last form of `foreach`, the type of the loop variable is autodetected by the type of the container elements.

The loop variable runs through all values of the container elements and the loop body is executed for each value of the loop variable. Changing the loop variable inside the loop body does not change the container elements, i.e., they are read-only.

For example:

```
var
  ss: set of string := ['Ivanov', 'Petrov', 'Sidorov'];
  a: array of integer := (3,4,5);
  b: array [1...5] of integer := (1,3,5,7,9);
  l := new List<real>;
begin
  foreach s: string in ss do write(s,' ');
  writeln;
  foreach x: integer in a do
    write(x,'
```

```
  writein;
  foreach var x in b do
    write(x, ' ') ;
  writein;
  foreach var r in l do
    write(r,' ');
end.
```

The while and repeat loop operators

The **while** loop statement has the following form:

while *condition* **do** *operator*

A condition is a boolean type expression, and the operator after **do** is called *the loop body*. Before each iteration of the loop, the condition is evaluated, and if it is true, the loop body is executed, otherwise the loop is exited.

If *the condition is* always true, *a loop* may occur:

```
while 2>1 do write(1);
```

The **repeat** loop operator has the following form: **repeat** *operators* **until** *condition*

Unlike the **while** loop, the condition is evaluated after the next loop iteration, and if it is true, the loop is exited. Thus, the operators that form the loop body of the **repeat** operator are executed at least once.

The repeat operator is usually used in situations where the condition cannot be checked without executing the loop body. For example:

```
read(x); until x=0;
```

If *the condition is* always false, *a loop* may occur:

```
repeat
  write(1);
until 2=1;
```

Operator with

The **with** operator allows you to reduce access to the fields of a record, as well as to the fields, methods, and properties of an object. It has the form:

with *record or object name* **do** *operator*

or

with *list of names* **do** *operator*

You can omit the name of a record when accessing the field of a specified record or the name of an object when accessing the field, method, or property of a specified object. For example, let a variable be described as

```
var DateOfBirthday = record
    Day: Integer;
```

```
    Month: Integer;
    Year: Integer;
  end;
```

Then assigning values to its fields without using the **with** operator looks like this:

```
DateOfBirthday.Day := 23;
DateOfBirthday.Month := 2; DateOfBirthday.Year := 1965;
```

Using the **with** operator allows you to shorten the previous entry:

```
with DateOfBirthday do
begin
  Day := 23;
  Month := 2;
  Year := 1965;
end;
```

If the external variable has the same name as the field (method, property), then the field (method, property) is preferred. If there are nested **with** operators, first the attempt is made to consider the variable as a field of the record or object of the internal operator **with** itself, then directly the enclosing **with** operator, and so on. If a **with** statement contains a list of objects, they are considered from right to left. For example, if there are descriptions of

```
var x,y,z: integer; a: record
       x,y: integer;
    end;
  b: record x: integer;
    end;
```

then the program fragment

```
with a,b do begin
  x := 1;
  y := 2; z := 3;
end;
```

is equivalent to the fragment

```
with a do with b do begin
   x := 1;
   y := 2; z := 3;
 end;
```

as well as the fragment

```
b.x:=1;
a.y:=2; z:=3;
```

The operator with is obsolete and is now practically not used.

The unconditional goto operator

The unconditional **goto** operator has the following form:

goto *tag*

It moves the execution of the program to the operator marked with *the label*.

A label is an identifier or an unsigned integer. To label a statement with a label, you must precede the statement with a label followed by a colon:

labeii: *operator*

Labels must be described in the labels section using the service word **label**:

```
label 1,2,3;
```

For example, as a result of executing the program

```
label 1,2;
begin
  var i := 5;
2: if i<0 then goto 1;
  write(i);
  Dec(i);
  goto 2;
1: end.
```

will be output 543210.

The label must mark the operator in the same block in which it is described. A label cannot label more than one operator.

The transition to the label can be performed either on the operator in the same block or on the operator in the enclosing construction. Thus, it is forbidden to jump to a label inside a loop from outside the loop.

The use of goto operator in a program is considered a sign of bad programming style. For the main variants of **goto** use there are special procedures introduced in Pascal language: **break** - transition to the operator following the loop, **exit** - transition behind the last operator of the procedure, **continue** - transition behind the last operator in the loop body.

One of the few examples of the `goto` operator's appropriate use in a program is to exit several nested loops simultaneously. For example, when searching for element `k` in a two-dimensional array:

```
var a: array [1..10,1..10] of integer;

  var found := False;
  for var i:=1 to 10 do
  for var j:=1 to 10 do if a[i,j]=k then begin
      found := True;
      goto c1;
    end;
c1: writeln(found);
```

The break, continue, and exit operators

The **break** AND **continue** operators are only used inside loops.

The **break** operator is designed to terminate the loop prematurely. It immediately exits the current loop and proceeds to the next loop iteration. The **continue** operator terminates the current loop iteration by jumping to the end of the loop body. For example:

```
flag := False; for var i:=1 to 10 do begin read(x);
  if x<0 then continue; // skip the current loop iteration
  if x=5 then
  begin
    flag := True; break; // exit the loop end;
end;
```

Using **break** and **continue** operators outside the loop body is wrong.

The **exit** statement is designed to terminate a procedure or function prematurely. For example

```
function Analyze(x: integer): boolean;
begin if x<0 then begin
    Result := False; exit end; - -- end;
```

Calling **exit** in the operators section of the main program causes it to end immediately.

More precisely, **break**, **continue** and **exit** in **PascalABC.NET** are special internal procedure calls.

yield operator

The yield operator is used in functions that generate sequences and has the form:

yield *expression*

A function that contains iterations is called *an iterator*. At each call such a function executes code up to the next iterator, then ends its work by returning the value of the expression specified in the iterator, and saves its state until the next call.

For example:

```
function Squares(n: integer): sequence of integer; begin
  for var i:=1 to n do yield i*i
end;

begin
```

```
  var q := Squares(5);

  foreach var x in q do
    Print(x);
  Println;

  q.Println;
end.
```

In this example, the variable q stores a sequence, i.e. the algorithm for calculating the squares of the first n numbers, which will be run either in the foreach loop on sequence q, or by calling the Println extension method for sequence q. These sequences are returned one by one by calls to the yield operator in the body of the Squares function.

After each call to the operand yield, the function returns the next value i*i, finishes its work and saves values of all its variables in the internal context. The next time you call this function, its body starts executing as if from the point where we were at the end of the previous call.

A yield statement can contain variables external to the function. The yield operator is said to capture such variables. For example:

```
var a := 2;

function Squares(n: integer): sequence of integer;
begin
  for var i:=1 to n do yield i*a
end;

begin
  var q := Squares(5);

  q.Println; a := 3;
  q.Println;
end.
```

In this code, the yield operator captures the variable a from external context. The capture is done by reference: if you change the a variable in the main program and call the iterator function again, the changed value of the a variable will be used to generate the sequence. As a result, the output of this program will look like:

```
2 4 6 8 10
3 6 9 12 15
```

There are a number of limitations for functions that have yields in their bodies:

1. Functions containing yield can only return sequences.
2. Among the parameters of iterator functions there cannot be const, var, params-parameters and default parameters.
3. If a function uses yield, it is forbidden to use the Result variable, and vice versa.
4. Functions with yield cannot contain lock, try...except, try...finally operators.
5. Yield cannot be nested in the with statement.
6. Yield cannot be used inside lambda expressions.
7. Functions with yield cannot contain nested subroutines and cannot be nested subroutines themselves.
8. Functions with yield cannot contain local type definitions
9. The methods of extending cueld cannot be recursive.

The yield sequence operator

The yield sequence operator is used in sequence-generating functions along with theyieldoperator, and has the form:

yield sequence *expression*

Unlike theyieldoperator, the yield sequence operator tries elements of the sequence specified in the expression and returns those elements as values of the main iterator function. For example, the following code:

```
function f: sequence of real;
begin
  yield sequence Seq(1,2,3);
  yield 4;
end;

begin
  f.Println;
end.
```

will output the sequence

```
1 2 3 4
```

The following example illustrates the formation of a sequence of elements when traversing a binary tree in infix order:

```
function InfixPrintTree<T>(root: Node<T>): sequence of T;
begin
  if root = nil then exit;
  yield sequence InfixPrintTree(root.left);
  yield root.data;
  yield sequence InfixPrintTree(root.right);
end;
```

The same restrictions apply to functions that have yield sequences in their bodies as to functions with yield.

The operator try ... except

The operator **try** ... **except statement** has the form:

try *operators* **except**
  exception handling block **end;**

The **try** block is called *a protected block*. If an error occurs during program execution, it is terminated and execution is passed to the **except** block. If an exception is handled in the **except** block, then

after it is handled the program continues to run from the statement following **try** ... **except** ... **end**. If the exception is left unhandled and there is a comprehensive **try** block, its execution is passed to the **except** block. If the **try** block does not exist, then the program will terminate with an error. Finally, if there is no error in the **try** block, the **except** block is ignored and the program continues.

If another exception occurs during exception handling (in the **except** block), the current **except** block is terminated, the first exception is considered unhandled, and handling of the new exception is transferred to the comprehensive **try** block. Thus, there is at most one unhandled exception at any given time.

The exception handling block is either a sequence of statements separated by semicolons, or a sequence of exception handlers of the form

on *name*`:` *type* **do** *operator*

Handlers are separated by '`;` ', the last handler can also be followed by '`;` '. Here *type* is the exception type (must be derived from the standard `Exception` type), *name* is the name of the exception variable (the name followed by a colon can be omitted). In the first case all operators from the **exceptions** block are executed while processing the exception. In the second case the current exception type is searched among the handlers (the handlers are enumerated sequentially from the first to the last one), and if the handler is found the corresponding exception handling statement is executed, otherwise the exception is considered unhandled and is passed to the enclosing **try** block. In the latter case all **on** handlers can be followed by **else** branch which will definitely handle exception if none of the handlers is executed.

Note that the exception variable name can be the same in different handlers, i.e. it is local to the handler.

Searching for exception types in handlers is done with respect to inheritance: an exception will be handled if it belongs to the type specified in the handler or derived from it. Therefore, it is customary to write derived class handlers first, and base class handlers second (otherwise the derived class exception handler will never work). The

`Exception` handler handles all possible exceptions and must therefore be written last.

Example.

```
var a: array [1...10] of integer;
try
  var i: integer;
  readln(i);
  writeln(a[i] div i);
  -- except on System.DivideByZeroException do
writeln('Divide by 0');
  on e: System.IndexOutOfRangeException do
    writeln(e.Message);
  on System.FormatException do writeln('Invalid input
    format');
  else writeln('Some other exception'); end;
```

The operator try ... finally

The operator **`try`** ... **`finally`** **`statement`** has the form:

```
try operators
finally
  operators
end;
```

Operators in the **`finally`** block are executed regardless of whether or not an exception occurs in the **`try`** block. The exception itself is not handled.

The **`finally`** block is used to return previously allocated resources.

Example 1. Closing an open file.

```
reset(f);
try

finally close(f);
end;
```

The file will be closed whether or not an exception occurs in the **`try`** block.

Example 2. Returns the allocated dynamic memory.

```
New(p);
try

finally
  Dispose(p);
```

```
end;
```

The dynamic memory controlled by the `p` pointer will be returned regardless of whether or not an exception occurs in the **`try`** block.

Operator raise

The **raise** operator is designed to raise an exception and has the form:

```
raise object
```

Here *the object* is an object of a class derived from `Exception`.
For example:

```
raise new Exception('Error');-
```

When you raise a specific exception, it is desirable to define your type of exception.

To re-generate an exception inside the **except** section, a **raise** call without parameters is also used:

```
raise;
```

The += and -= operators for procedural variables

The assignment operator `+=` is for attaching to a procedure variable and the assignment operator `-=` is for detaching. Subprograms are called in the order of accession. For example:

```
procedure mult2(var r: real);
begin r := 2 * r;
end;

procedure add3(var r: real);
begin r := r + 3;
end;

var
  p: procedure (var x: real);
  r: real;

begin r := 1; p := mult2; p += add3;
  p(r); // r := 2 * r; r := r + 3;
  p -= mult2;
  p(r); // r := r + 3;
end.
```

Disconnecting non-attached subroutines does not perform any action.

You can also attach/unattach static and instance class methods to a procedural variable. For an example, see the topic on proceduralvariables.

The `+=` and `-=` operators are also used to add/remove handlers for .NET events. For example:

```
procedure OnTimer1(sender: object; e:
System.Timers.ElapsedEventArgs);
begin
  write(1);
end;
```

```
begin
  var Timer1 := new System.Timers.Timer(1000
  Timer1.Elapsed += OnTimer1;
  Timer1.Start;
  while True do
Sleep(1000); end.
```

Operator lock

The lock operator looks like this:

lock *object* **do** *operator*

The object necessarily belongs to the reference type.

The **lock** statement ensures that *the statement* will only be executed by one thread. *The object* here stores the lock, and *the statement* that represents the body of the **lock** statement is called the synchronization block. When the first thread enters the lock block it locks *the object*, when it leaves the lock block it unlocks it. If *the object is* locked, no other thread can enter the synchronization block and suspends until *the object* is unlocked.

Operator

```
lock obj do oper;
```

is completely equivalent to the next section of code:

```
Monitor.Enter(obj); try oper;
finally
  Monitor.Exit(obj); end;
```

Procedures and functions: overview

What are procedures and functions

A procedure or function is a sequence of statements that has a name, a list of parameters, and can be called from various parts of a program. Functions, unlike procedures, as a result of their execution return a value that can be used in an expression. For consistency, functions and procedures are called *subroutines*.

Description of procedures and functions

Any procedure or function used in the program must be previously described in the descriptions section.

The description of the procedure is as follows:

```
procedure name (a list of formal parameters);
descriptions section begin
  end operators;
```

The description of the function is as follows:

```
function name (list of formal parameters): type of the return value ;
descriptions section begin
  end operators;
```

The operators of a subprogram bordered by **begin/end** operator brackets are called *the body* of that subprogram.

A list of formal parameters along with surrounding parentheses may not exist. It consists of one or more sections, separated by "; ". Each section consists of a comma-separated list of variables, followed by a colon and a type. Each section may be preceded by the keyword **var** or **const**, indicating that the parameters are passed by reference (see Parameters of Procedures and Functions). The type of a formal parameter must be either a name, a dynamic array, a set, or a procedural variable (structuralequivalence of types takes place for the last three types).

The procedure or function descriptions section is structured similarly to the main program descriptions section. Here, the so called ***local*** variables and constants, types (except for classes - classes can only be described globally), as well as nested procedures and functions are described. All such local objects are accessible only within a given subprogram and are not visible from the outside.

In the subprogram descriptions section, you can describe other subprograms. The exception is class methods described directly in the class body: they cannot describe nested subprograms due to syntactic ambiguity.

For example:

```
procedure DoAdd(a,b: real; var res: real);
begin
```

```
  res := a + b;
end;
```

Subprogram call

A subprogram is described once and can be called many times. The procedure call operator is used to call the procedure:

```
begin
  var x := Readinteger; var y := Readinteger; var res:
  integer;
  DoAdd(x,y,res);
  Print(res);
  DoAdd(2*x,y,res); Print(res);
end;
```

A function call expression is used to call a function.

Result variable

Inside the body of any function, a special variable named `Result` is defined, which stores the result of the function's calculation. Its type is the same as the type of the function's return value. For example:

```
function Sum(a,b: real): real;
begin
  Result := a + b; end;
function MinElement(a: array of real): real;
begin Result := real.MaxValue; foreach var x in a do if x <
Result then
      Result := x;
end;
begin var a := Seq(1,5,3); writeln(MinElement(a) + Sum(2,3));
end.
```

If you do not assign a value to the `Result` variable within a function, the function will return an unpredictable value as a result of its call.

Simplified syntax for describing subroutines

PascalABC.NET has a simplified syntax for describing single-operator procedures:

```
procedure WriteStar := write('*');
```

A similar syntax is available for functions that calculate a single expression:

```
function Add(a,b: real): real := a + b;
```

In some cases, it is possible for the return value of a function to have an auto-retrieval of types:

```
function Add(a,b: real) := a + b;
```

Procedure and function parameters

Parameters specified when describing a subprogram are called ***formal parameters***. Parameters that are specified when calling a subprogram are called ***actual parameters***.

If a formal parameter is described with the qualifier keyword **`var`** or **`const`**, it is called ***a variable parameter*** and is said to be passed by ***reference***. If the parameter is described without **`var`** or **`const`**, it's called ***a value parameter*** and is said to be passed ***by value***. The word reference is also used in PascalABC.NET for reference types.

If a parameter is passed by value, the values of the actual parameters are assigned to the corresponding formal parameters when the subprogram is called. The types of actual parameter-values must be compatible in assignment with the types of the corresponding formal parameters.

For example, have the following description of the procedure:

```
procedure PrintSquare(i: integer);
begin
  writeln(i*i); end;
```

Then when you call `PrintSquare(5*a-b)` the value `5*a-b` will be calculated and assigned to the `i` variable, and then the body of the procedure will be executed.

If a parameter is passed by reference, then when a subprogram is called, the actual parameter replaces the corresponding formal parameter in the body of the procedure. As a result, any changes to a formal parameter-variable within a procedure result in corresponding changes to the actual parameter. The actual parameter-variables must be variables, and their types must be equivalent to the types of the corresponding formal parameters.

For example, if the procedure described is

```
procedure Mult2(var a: integer);
begin a := a*2;
end;
```

then after calling `Muit2(d)` the value of `d` will increase by a factor of 2.

As an actual parameter-value, you can specify any expression

whose type is the same as the type of a formal parameter or is implicitly reduced to it. Only a variable whose type is exactly the same as the formal parameter can be specified as an actual parameter-variable.

When a parameter is passed by reference, the address of the actual parameter is passed to the subroutine. Therefore, if a parameter occupies a lot of memory (array, record, string), it is usually also passed by reference. As a result, it is not the parameter itself that is passed to the procedure, but its address, which saves memory and running time. If the parameter changes within a subprogram, it is passed with the **var** keyword; if it does not change, it is passed with the **const** keyword:

For example:

```
type
  Person = record name: string; age,height,weight: integer;
  end;
procedure Print(const p: Person);
begin
  write(p.name,',p.age,',p.height,',p.weight); end;
procedure IncAge(var p: Person);
begin
  Inc(p.age); end;
```

Note the peculiarities of passing dynamic arrays as parameters of subroutines.

Since a dynamic array is a reference, when you change a formal parameter-dynamic array inside a subroutine, the corresponding actual parameter changes. For example, as a result of running a program

```
procedure p(a: array of integer);
begin
  a[1] := 2;
end;

var b: array of integer := (1,1);

begin
  p(b);
  writeln(b[1]); end.
```

will be output 2. Passing dynamic arrays by reference only makes sense if the memory for a dynamic array is redistributed within a subroutine:

```
procedure q(var a: array of integer);
begin
  SetLength(a,10);
end;
```

Variable number of parameters

To indicate that a subprogram should have a variable number of parameters, the keyword **params** is used, followed by a description of a dynamic array. For example: `function Sum(params a: array of integer): integer; begin Result := 0; for i: integer := 0 to a.Length do Inc(Result,a[i]);`

```
   end;
```

When the subroutine is called, any nonzero number of actual parameters of compatible type, listed separated by commas, can be in place of the formal parameter **params**:

```
   var s: integer := Sum(1,2,3,4,5); s := s + Sum(6,7);
```

In the parameter list, the **params** keyword can only be specified for the last parameter, and that parameter must not be the default parameter. Parameters **params** are always passed by value only.

Default settings

You can use the default parameters in the subprogram header. To do this, just put an assignment sign and the value after the parameter. If you do not specify the default value of the parameter when you call it, the value specified in the subroutine description will be used.
The default parameters should be passed by value and come last in the parameter list.

For example:

```
procedure PrintTwo(a,b: integer; delim: char := ' ');
begin
  write(a,delim,b);
end;


PrintTwo(3,5);
PrintTwo(4,2,';');
```

All options for calling a subprogram with default parameters can participate in allowing overloading.

Advance Announcement

In some situations it may be necessary to call a subprogram described later in the program. For example, this is the case with indirect recursion (subprogram `A` calls subprogram `C`, which in turn calls subprogram `A`). IN THIS CASE, THE subprogram's leading declaration is used, consisting of its header followed by the **`forward`** keyword. For example:

```
procedure B(i: integer); forward;
procedure A(i: integer);
begin

  B(i-1);
end;
procedure B(i: integer);
begin

  A(i div 2); end;
```

It is forbidden to make a leading announcement for an already described subprogram.

For methods, the **`forward`** keyword is not allowed. It is not necessary because you can call methods defined in the class body later.

Subroutine name overloading

Several procedures or functions with the same name, but different numbers or types of parameters can be defined in one namespace. The names of such procedures and functions are called overloaded, and their creation is called ***name overloading***. A kind of name overloading isoverloading of operations.

When you call an overloaded procedure or function, you choose the version that has formal parameter types that match or are closest to the actual parameter types. For example, if there are descriptions

```
procedure p(b: byte); begin end;
procedure p(r: real);
begin end;
```

then calling `p(i.o)` will select the overloaded version with a parameter of type `real` (exact match), and calling `p(i)` will select the overloaded version with a parameter of type `byte` (this will convert the actual parameter of type `integer` to type `byte`).

Note that, unlike Object Pascal, you don't need to use the **`overload`** service word when overloading.

If no version in the current namespace fits the call, a compilation error occurs. If two or more versions fit the call equally well, there is also a compilation error of ambiguous subroutine selection. For example, if there are descriptions of

```
procedure p(i: integer; r: real);
begin end;
procedure p(r: real; i: integer);
begin end;
```

then when you call `p(i,2)` they both fit the same, which leads to ambiguity.

It is forbidden to overload a subprogram with another subprogram with the same number and type of parameters, differing only in whether the parameter is passed by value or by reference. For example, the descriptions of

```
procedure p(i: integer);
```

и

```
procedure p(var i: integer);
```

are considered to be the same.

The return value of a function is not involved in the overloading resolution, i.e., overloaded functions cannot differ only in the types of return values.

The name overloading algorithm for multiple connected modules and the method name overloading algorithm have peculiarities. The main feature of these algorithms is that they work across namespace boundaries.

The search for the overloaded name of a global subprogram in the presence of several connected modules is performed in all modules. At that, first the current module is searched, and then all modules connected in the **uses** section are searched, in the order *from right to left*. If during this search an object is found that cannot overload the previous ones (for example, a procedure is overloaded and a variable name is found), the overloading chain ends, and the search for the best overloaded subroutine goes on among the ones found up to that point. If a module compiled later has a subprogram with exactly the same parameters, it hides the version from the module compiled earlier.

For example, let the main program connect two modules, `uni` and `un2`:

main.pas

```
uses un2,un1;
procedure p(i: integer);
begin
  write(i);
end;
begin
  p(2.2);
  p(2);
end.
```

un2.pas

```
unit un2;
procedure p(r: real);
begin
  write(3);
end;
end.
```

unl.pas

```
unit uni;
procedure p(r: real);
begin
  write(2);
end;
end.
```

The result will be `21`, which means that procedure `p` from the `uni` module was called first.

The search for an overloaded method name is performed similarly: first the current class is browsed, then its base class, and so on up to the `object` class, or up to the point where an object is encountered which cannot overload the previous ones (field or property name). From all methods of the same name found in this way, the best one is chosen. There may be methods with identical parameters in different classes; in this case the first encountered method from the given class to the `object` class is called.

Subprograms with a variable number of parameters also participate in overloading, but normal subprograms have priority over them. For example, in the situation

```
procedure p(i: integer);
begin
  write(1);
end;
procedure p(params a: array of integer);
begin
  write(2);
end;
begin
  p(1) end.
```

the first procedure will be called.

Calling subroutines from an unmanaged dll

To call a subroutine from an unmanaged dll (containing normal, not .NET code), use a construction of the form:

function header **external** ' *dll's NAME*' **name** ' *dll's FUNCTION NAME*';

For example:

```
function MessageBox(h: integer; m,c: string; t: integer):
integer;
  external 'User32.dll' name 'MessageBox';
- • ••
MessageBox(0,'Hello!','Message',0);
```

Module structure

Modules are used to divide the program text into several files. Modules describe variables, constants, types, classes, procedures, and functions. To be able to use these objects in the calling module (which can also be the main program), you should specify the name of the module file (without the .pas extension) in the **uses** section of the calling module. The module file (.pas) or compiled module file (.pcu) must be located either in the same directory as the main program or in the **Lib** subdirectory of the **PascalABC.NET** system directory.

The module has the following structure:

unit *module name*;

interface

interface **implementation** *section*

initialization *section*

initialization section

the finalization section of the **end.**

There is also a simplified module syntax without interface and implementation sections.

The first line is mandatory and is called ***the module header*** . The module name must be the same as the file name.

The interface section and the module implementation section can start with the **uses** section of external modules and .NET namespaces. The names in the two **uses** sections must not overlap.

The interface section includes a declaration of all the names that are exported by this module to other modules (when you connect it in the **uses** section). These can be constants, variables, procedures, functions, classes, interfaces. The implementation of class methods can be given directly in the interface section, but this is not recommended.

The implementation section contains the implementation of all procedures, functions and methods declared in the interface section. In addition, the implementation section may contain descriptions of internal names that are not visible outside the

module and are used only as auxiliary.

The initialization and *finalization sections* are a sequence of operators separated by `;`. Operators from the module initialization section are executed before the main program starts, operators from the module finalization section are executed after the main program ends. The order of execution of initialization and finalization sections of connected modules is unpredictable. Both the initialization and finalization sections may be missing.

Instead of initialization and finalization sections, there can only be an initialization section in the form of

begin *sequence of* **end** *statements*.

For example:

```
unit Lib;
interface
uses GraphABC;
const Dim = 5; var Colors: array [1...Dim] of integer;
function RandomColor: integer; procedure FillByRandomColor;
implementation
function RandomColor: integer;
begin
  Result := RGB(Random(255),Random(255),Random(255));
end;
procedure FillByRandomColor;
begin
            for i: integer := 1 to Dim doColors
                    [i] := RandomColor;
end;
initialization
  FillByRandomColor;
end.
```

Cyclic links between modules are possible under certain constraints.

Section uses

The **uses** section consists of several consecutive **uses** sections, each of which looks like:

uses *list of names* ;

The names in the list are listed, separated by commas, and can either be names of **PascalABC.NET** plug-ins or .NET namespaces. For example:

```
uses System, System.Collections.Generic, MyUnit;
```

Here `MyUnit` is a **PascalABC.NET** module represented as a source or compiled .pcu module, `System` and `System.Collections.Generic` are .NET namespaces.

In a module or main program that contains a **uses** section you can use all the names from the **PascalABC.NET** plug-ins and .NET namespaces. The main difference between modules and .NET namespaces is that a module contains code, while .NET namespaces contain only names - to use code you must include it using the compiler directive *{$referenceBuild Name}*, *whereBuild Name* is the name of the dll file containing the .NET code. Another equally important difference is that you cannot use names defined in another module or main program without connecting that module in the **uses** section. In contrast, if a .NET assembly is connected with the `$reference` directive, you can use its names by explicitly specifying them with the namespace without connecting that namespace in the uses section. For example:

```
begin System.Console.WriteLine('PascalABC.NET'); end.
```

By default, the first **uses** section implicitly includes the PABCSystem module, which contains the standard constants, types, procedures and functions. Even if the **uses** section is missing, the PABCSystem module is implicitly included. In addition, the System.dll, System.Core.dll and mscorlib.dll assemblies containing the basic .NET types are included by default using the implicit `$ reference` directive. Global names are searched first in the current module or main program, then in all connected modules and namespaces, beginning with the rightmost in the **uses** section and ending with the

leftmost. The namespace of the rightmost module is assumed to be nested within the namespace of the leftmost module. Thus, there is no name conflict. If you need to use a name from a particular module or namespace, you should use the notation

ImModule name **.** *Name*

or

NameSpaceNameNET **.** *Name*

The module name can also be the name of the main program if it has a **program** header.

t the simplified syntax of the module

The simplified syntax of modules without interface and implementation sections is as follows:

```
unit module name;

 descriptions section
end.
```

or

```
unit module name;

descriptions section
begin

initialization section end.
```

The descriptions section describes constants, variables, procedures, functions, classes, interfaces. All names are exported. The simplified module syntax is convenient to use for initial training - the module differs from the program only in the header and possibly in the absence of the statement section.

Cyclic links between modules

Cyclic module references in interface parts are forbidden. For example, the following situation is incorrect:

```
unit A; interface uses B;
implementation end.

unit B; interface uses A;
implementation end.
```

Thus, it is not possible to define two public classes in different modules with object fields that refer to each other.

However, if one reference is in the interface part and a second reference is in the implementation part, or both are in the implementation parts, then cyclic references are allowed in this case:

```
unit A; interface implementation uses B;
end.

unit B; interface uses A;
implementation end.
```

Dll Libraries

Dll libraries (dynamically linked libraries):

- contain a group of interconnected subprograms
- are in the compiled file
- are designed to be accessed from various programs

They are located in a file with a .dll extension either in the application's current directory (*local libraries*) or in the system directory (*global libraries*). Global libraries can be used by several applications simultaneously.

In their purpose, libraries are very similar to modules, but have a number of important differences.

Differences between libraries and modules

1. When you create an executable .exe file from modules, the linker program puts into it TOJ types and constants that are used (called) in the main program. When compiling, all subroutines, because it is not known which subroutines will be needed for a particular prim
2. The `.dii` libraries are completely loaded into RAM when the program runs.
3. `.dii` libraries are often used simultaneously by multiple programs.
4. A `.dii` library can be written and compiled in one language, and can be called from programs written in other languages. For example, a PascalABC.NET program can call a function from a library written in C# and vice versa. In this way, the libraries support ***cross-language communication***.

Library structure

The library has almost the same structure as the module:

```
library name;
interface

interface section
implementation

implementation section
end.
```

The name of the library must match the name of the pas file in which the library is located. There is also a simplified syntax for libraries - without interface and implementation sections, which is the same as the simplified syntax for modules (except for the header).

Compiling the library creates `a .dii file` containing the compiled library in the current directory.

Connecting the library to the main program

The compiler directive *preferenceLibraryName}* is used to connect the library to the main program. For example:

```
preference ABC.dll}
preference ABC1.dll}

begin
  writeln(a.GetType);
end.
```

The library can be connected anywhere in the source file.

Libraries ABC and ABC1 look like this, respectively:

```
library ABC;
var a: integer;
end.
```

и

```
library ABC1;
var a: real;
end.
```

Algorithm for finding names in libraries

The name is searched first in the source module, then in the modules connected in the uses section in order from right to left, and only then in the connected libraries in order of connection.

According to this rule in the example from the previous paragraph, the variable a will be of type integer.

In case of a name conflict, the name used can be preceded by the library name followed by a period:

```
preference ABC.dll}
preference ABC1.dll}

begin
  writeeln(ABCl.a.GetType);
end.
```

Sharing modules and libraries

Sharing modules and libraries has a number of limitations.

Modules are used to create an executable .exe file or a .dll library (an assembly in .NET terms). Libraries can then be connected to other libraries or to the main program using the preference} directive. Using the same module for two libraries or the same module for a library and the main program that connects that library is prone to a number of collisions if careless programming.

Here are examples of such conflicts.

Example 1.

Module Unit1.pas

```
unitl;

type Point = auto class x,y: integer;
end;
end.
```

Library lib1.pas

```
library libl;

Unitl;

function Origin: Point := new Point(0,0);

end.
```

Main program

```
preference libl.dll} uses Unit1;

begin
  var p: Point; // type Point from Unit1
  p := Origin; // Origin returns type libl.Point end.
```

Compilation error: Cannot convert Unitl.Point type to Point

The reason: The lib1.dll library and the main program have different versions of the Point type, which are not compatible in assignment.

Example 2.

Module Unit1.pas

```
unit1;

type Point = auto class x,y: integer;
```

```
end;

procedure pp; begin
  Print(12345); end; end.
```

Library lib1.pas `library lib1;`

```
uses Unit1;

end.
```

Main program `preference lib1.dll}`

```
var p: Point;

begin pp; end.
```

Compilation error: Unknown name 'Point', Unknown name 'PP'

The reason: the library lib1.dll does not "sew" types and procedures not used in it. That is, lib1.dll will be empty.

Example 3. The correct solution to the code from Example 2

Module Unit1.pas

```
unit1;

type Point = auto class x,y: integer;
end;

procedure pp;
begin
  Print(12345);
end;
end.
```

Library lib1.pas

```
library libl;

Unitl;

type Point = Unitl.Point; // type synonym

procedure pp := Unit1.pp; // call procedure pp again


end.
```

Main program

```
preference lib1.dll}
```

```
// do not connect module Unit1!!!

var p: Point;

begin pp; end.
```

Result: Everything compiles

Example 4. Specifics of using some types of standard PABCSystem module.

lib1.pas `library libl; procedure q1(s: set of integer); begin end;`

```
procedure q2(f: Text);
begin end;

procedure q3(f: file of integer); begin end;

end.
```

Main program `preference libl.dll} var fl: Text; f2: file of integer;`

```
begin q1([1,2,3]); q2(f1); q3(f2);
end.
```

Compilation errors:

Cannot convert set of integer type to PABCSystem.TypedSet
Cannot convert Text type to PABCSystem.Text
You cannot convert file type of integer to PABCSystem.TypedFile

Reason: the type of embedded set, Text type, and types of typed and untyped files cannot be passed through the boundaries of assemblies (in different assemblies these will be different types)

Documentary comments

You can mark the titles of procedures, functions, methods, class names, types, constants, and variables with so-called document comments. Documenting comments pop up in the editor's tooltips when you hover the mouse over a word, when you open a parenthesis after a subprogram name, and when you select a field from the list of fields that drop down when you click a dot after the name. The system of tooltips in the editor is called Intellisense.

The documenting comment is placed on the line preceding the object to be marked and starts with `///`. For example:

```
const /// Constant Pi Pi = 3.14;
```

```
type
  /// TTT is a synonym of integer type
  TTT = integer;
  /// Documenting comment of class XXX
  XXX = class
  end;

/// Documenting comment of procedure p procedure p(a :
integer);
begin
end;

var
  /// Documenting comment of variable t1
  t1: TTT;
```

Documenting comments can take several lines, each of which must begin with `///` . For commenting subprograms you can use a documenting comment `///-` in the first line, then its contents change the subprogram header in the tooltip when the mouse pointer is moved. For example:

```
///- Exclude(var s : set of T; el : T) /// Removes element el
from s procedure Exclude(var s: TypedSet; el: object);
```

If the first line of the documenting comment has the form ///--, then the tooltip does not pop up. This is done for items that you want to hide from the tooltip system.

Overview of classes and objects

Class description

A class is a composite type consisting of fields (variables), methods (procedures and functions) and properties.
The description of the class has the form:

```
type Class NAME = class
    section1
    section2
     -- end;
```

Each section has a view:

access modifier of declaration fields description or methods description and properties description

The access modifier in the first section may be missing, which implies the **internal** modifier (visibility everywhere inside the assembly).

Methods can be described either inside or outside the class. When describing a method inside a class, its name is prefaced by the class name, followed by a period. For example:

```
type Person = class private
    fName: string;
    fAge: integer;
  public
    constructor Create(Name: string; Age: integer); begin
    fName := Name;
      fAge := Age;
    end;
    procedure Print;
    property Name: string read fName;
    property Age: integer read fAge;
  end;

procedure Person.Print;
begin
  Writeln($'Name: {Name} Age: {Age}');
end;
```

The ancestor class name (see Inheritance) may be given after the word **class** in parentheses, as well as a comma separated list of supported interfaces.

The word `class` can be preceded by the keyword `sealed`, in which case it is forbidden to inherit from the class.

All descriptions and declarations within a class form ***the body of the class***. Fields and methods form ***the interface*** of the class.
Field initializers are described here.

Classes can only be described at the global level.
Local class definitions (i.e., definitions in the description section of subprograms) are forbidden.

Variable class type

In **PascalABC.NET,** classes are reference types. This means that a variable of type class actually stores a reference to an object.

Variables of class type are called ***class objects*** or ***class instances***. They are initialized by calling the class constructor, a special method that allocates memory for the class object and initializes its fields:

```
var p: Person := new Person('Ivanov',2 0);-
```

After initialization, public class members (fields, methods, properties) can be accessed through a class type variable using dot notation:

```
Print(p.Name,p.Age);
p.Print;
```

Output a variable of class type

By default, the `write` procedure for a variable of class type outputs the contents of its public fields and properties in parentheses, separated by commas:

```
Write(p); // Ivanov 20
```

To change this behavior, the virtual `Tostring` method of the `object` class must be overridden in the class, in which case it will be called when the object is output.

For example:

```
type
  Person = class
    - • ••
    function ToString: string; override;
    begin
      Result := $'Name: {Name} Age: {Age}';
    end;
  end;

var p: Person := new Person('Ivanov',20);
Writeln(p); // Name: Ivanov Age: 20
```

Assigning and passing as parameters to subroutines

A variable of class type is a reference and stores a reference to the object created by the constructor call.

As a reference a variable of type class can store the value **nil**:

```
p := nil;
• ••
if p = nil then ...
```

When you assign variables of type class, only the reference is copied. After assignment, both variables of the class type will reference and jointly modify the same object:

```
var p1,p2: Person;
• ••
pl := new Person('nempoe',20);
p2 := pl;
pl.IncAge;
p2.Print; // Name: Petrov Age: 21
```

Equality comparison

When comparing variables of class type for equality, references are compared, not values.

```
var p1 := new Person('nempoe',20);
var p2 := new Person('nempoe',20);
writeln(p1=p2); // False
p2 := p1;
writeln(p1=p2); // True
```

This behavior can be changed by overloading the = operation on the class.

Visibility of class members and access modifiers

Each field, method, or property of a class has an access modifier (attribute) that defines the rules for its visibility. In **PascalABC.NET** there are four kinds of access modifiers: **public**, **private**, **protected** and internal. Class members with **public** attribute can be accessed from anywhere in the program, class members with **private attribute** are available only inside methods of the class, class members with **protected attribute** are available inside methods of the class and all its subclasses, class members with **internal** attribute are available inside assembly (term .NET, assembly in our understanding is a set of files needed to generate an .exe or .dll file). In addition, **private** AND **protected** members ARE VISIBLE from everywhere within the module in which the class is defined.

The body of a class is divided into sections. Each section begins with an access modifier, followed by fields, and then by methods and properties with access defined by this modifier. The first section may not have an access modifier, in which case the modifier is **internal**. A class can have any number of sections in any order.

For example, let this code be located in one module:

```
type A = class private
    x: integer; protected
    a: integer; public
    constructor Create(xx: integer); begin x := xx; // true,
because inside a class method you can access its closed
field x a      := 0;// true
    end;
  end;
```

Let the following code be located in another module:

```
type
  B = class(A)
  public
    procedure print;
    begin
     writeln(a); // true, because a is a protected field
       writeln(x); // incorrect, because x is a closed field
    end;
  end;

var b1: B := new B(5);
```

```
writeln(b1.x); // incorrect, because x is a closed field
writeln(b1.a); // incorrect, because a is a protected field
b1.print; // correct, because print is an open method
```

Comments on the program text describe correct and incorrect access to fields and methods.

Methods

Methods are procedures and functions declared within a class or record. Special types of methods are constructors, destructors and overloaded operations.

Methods can be defined either inside a class (Java, C#, C++ style) or outside a class (Delphi, C++ style). When defining a method outside the class interface, its name is preceded by the class name, followed by a period. For example:

```
type Rectangle = class x1,y1,x2,y2: integer; constructor
    Create(xx1,yy1,xx2,yy2: integer); begin
      x1 := xx1; x2 := xx2;
      y1 := YU1; y2 := yy2;
    end;
    function Square: integer;
  end;
function Rectangle.Square: integer;
begin
  Result := abs(x2-x1) * abs(y2-y1);
end;
```

Usually, when a class is defined in the interface part of a module, only methods are declared in the class interface, and class methods are implemented in the module's implementation section.

Methods are divided into *class* methods and *instance methods*. Class methods in .NET are called *static methods*. The declaration of a class method starts with the **`class`** keyword. Instance methods can only be called through a class object variable. Class methods, on the other hand, are not associated with a specific instance of the class; they should be called as: *class name*`.` *method name* `(`*parameters*`)`

Within a class method there can be no access to the fields of the class, but only to other class methods

methods. In contrast, an instance method can call a class method.

For example:

```
type Rectangle = class - -- class procedure Move(var r:
Rectangle; dx,dy: integer);
     begin
       r.xl += dx; r.x2 += dx;
       r.yl += dy; r.y2 += dy;
     end; end;

var r := new Rectangle(10,10,100,100);
Rectangle.Move(r,5,5);
```

Essentially, class methods are a kind of global subroutines, but they are inside a class, which emphasizes that they perform actions related to that class. The class in this case acts only as a namespace.

It is not uncommon to create classes that consist entirely of class methods. This is, for example, the `system.Math` class, which contains definitions for mathematical subroutines.

Field initializers

When an object is created***, its genders are automatically initialized with null values*** if they are not initialized explicitly. They can be initialized either in the constructor or directly in the description. Initializing a field in description causes the initialization code to be inserted at the beginning of ALL constructors.

For example:

```
type A = class private
    x: integer := 1; y: integer;
    l := new List<integer>;
  public constructor Create(xx,yy: integer); begin
      x := xx; y := yy; end;
    constructor Create;
    begin end; end;
```

In this example, the code `x:=1; l : = new List<integer> is inserted` at the beginning of each constructor.

Constructors

Objects are created using special methods called ***constructors***.

A constructor is a function that creates an object in dynamic memory, initializes its fields, and returns a reference to the created object. This reference is usually immediately assigned to a variable of class type. When describing a constructor, the `constructor` keyword is used instead of the `function` keyword. In addition, the type of the return value is not specified for the constructor.

For example:

```
type Person = class private nm: string; ag: integer;
  public constructor Create(name: string; age: integer);
  end;
-- constructor Person.Create(name: string; age: integer);
begin nm := name; ag := age;
end;
```

In **PascalABC.NET** the constructor must always be named `Create`. When describing a constructor within a class, you can omit the constructor name:

```
type
  Person = class constructor (name: string; age: integer);
  begin nm := name; ag := age;
    end; end;
```

Because of **PascalABC.**NET's implementation of constructor calls, a constructor without parameters is always created in **PascalABC.NET** (regardless of whether another constructor is defined). This constructor initializes all fields with null values (string fields with empty strings, logical fields with False values).

There are two ways to call the constructor.

1 method. Object Pascal style.

To call a constructor, specify the class name followed by a separator point, the constructor name, and a list of parameters. For example:

```
var p: Person;
p := Person.Create('Ivanov',2 0);-
```

2 method. With `new` operation - C# style (preferred).

```
var p: Person;
p := new Person('Ivanov',2 0),-
```

A destructor in Object Pascal is a special procedure that destroys an object and frees the dynamic memory it was occupying. When describing the destructor the **destructor** service word **destructor** is used instead of **procedure**.

For example: `destructor Destroy; begin - -- end;`

Since memory in **PascalABC.NET** is managed by the garbage collector, the destructor in **PascalABC.NET** plays no role and is just a normal procedure-method.

Preliminary announcement of classes

Two or more classes can contain objects of other classes as fields, cyclically referring to each other.

For example:

```
type AAA = class b: BBB;
  end; BBB = class a: AAA;
  end;
```

This code will cause a compilation error because the type `BBB IS` not yet defined when field b is described. In such a situation you should use the preliminary description of the class in the form of

Class name `= class`;

A pre-described class must be fully described in the same **`type`** section:

```
type
  BBB = class;
  AAA = class b: BBB;
  end;
  BBB = class a: AAA;
  end;
```

Variable Self

Inside each non-static method, a variable `self` is defined implicitly, referring to the object that called the method.

For example:

```
type A = class
    1: integer;
    constructor Create(i: integer);
    begin
      Self.i := i;
    end;
  end;
```

At the time the `Create` constructor is called, the object will already have been created. The construct `self.i` refers to the field `i` of this object, not to the parameter `i` of the `Create` function. In fact, any non-static method implicitly has `Self` before any field name and method of that class.

Properties

A property looks like a class field, but allows you to perform some actions when accessing it for reading or writing. The property is described in the class or record as follows:

```
property Prop: type read PropertyReader write Propertywriter;
```

The PropertyReader can be:

- *is the name of the function to read the property*;
- *field* of the corresponding type;
- *expression* of the corresponding type (in this case the property is called *extended*).

The Propertywriter can be:

- *is the name of the procedure for recording a property*;
- *field* of the corresponding type;
- *operator* (in this case the property is called *extended*). A predefined variable value is available in the operator, in which the value to change the property is placed.

One of the sections - read or write - can be omitted, in which case we have a write-only or read-only property, respectively.

When accessing a property for reading, *PropertyReader* is called; when accessing it for writing, *Propertywriter* is called.

As a rule, each property is associated with some field of the class and returns the value of that field using *the read function*, and changes it using *the write procedure*. The read function and write procedure must be methods of this class and have the following form:

```
function getProp: type; procedure setProp(value: type);
```

If a property's read function simply returns the value of a field, you can specify the field's name instead of its name. Similarly, if the write procedure simply assigns a value to a field, you can replace its name with the name of that field.

Any of the **read** or **write** sections can be omitted, in which case we get a write-only or read-only property.

Usually, the read function and write procedure are described in the private section of the class. They can be virtual, in which case it is

appropriate to describe them in the protected class section.

First, let's look at an example that uses a read function and a write procedure for a property:

```
type Person = class private
    fName: string;
    fAge: integer;
    procedure setAge(value: integer);
    begin if value<0 then value := 0; fAge := value end;
    function getAge: integer;
    begin
      Result := fAge;
    end;
    function getName: string;
    begin
      Result := fName;
    end;
    function getId: string;
    begin
      Result := fName + fAge.ToString; end;
  public
    constructor (name: string; age: integer);
    begin
      fName := name;
      fAge := age;
    end;
    property Age: integer read getAge write setAge;
    property Name: string read getName;
    property Id: string read getId; end;
begin
  var p: Person;
  p := new Person('Ivanov',2 0);-
  p.Age := -3; // p.Age = 0 !
  p.Age := p.Age + 1; // the compiler replaces this code with
p.setAge(p.getAge + 1);
  writeln(p.Id);
end.
```

Whenever we assign a new value to the `Age` property, the `setAge` procedure is called with the corresponding parameter. Whenever we read the value of the `Age` property, the `getAge` function is called.

As already noted, in trivial cases, the procedure name in the **`write`** property section and the function name in the **`read`** property section can be replaced with the names of the corresponding fields. Here is the code with this remark in mind:

```
type
  Person = class
```

```
  private
    fName: string;
    fAge: integer;
    procedure setAge(value: integer);
    begin if value<0 then value := 0; fAge := value
    end;
    function getId: string;
    begin
      Result := fName + fAge.ToString;
    end;
  public constructor (name: string; age: integer) :=
(fName,fAge) := (name,age);
    property Age: integer read fAge write setAge; property
    Name: string read fName;
    property Id: string read getId;
  end;
```

Finally, let's use *the advanced properties*, replacing `getId` with the expression `fName + fAge.ToString`, and `setAge` with the operator that implements the body of this procedure:

```
type Person = class private
    fName: string;
    fAge: integer;
  public constructor (name: string; age: integer) :=
(fName,fAge) := (name,age);
    property Age: integer read fAge write fAge := value<0 ? 0
: value;
    property Name: string read fName;
    property Id: string read fName + fAge.ToString; end;
```

Note that advanced properties are only available in PascalABC.NET, introduced in the language so that you don't have to write a separate read function or write procedure, and not in other versions of Pascal.

Extended properties are convenient to use in many situations. For example, when ordinary properties are referenced by similar field properties of this class:

```
type MyList<T> = class private
    l := new List<T>;
  public
    property Capacity: integer read l.Capacity write
l.Capacity := value;
  end;
begin
  var ml := new MyList<integer>;
  ml.Capacity := 5; // write access: value 5 is copied to the
value variable
```

```
Println(ml.Capacity); // read access end.
```

Properties cannot be passed by reference to procedures and functions. For example, the following code is wrong:

```
Inc(p.Age); // error!
```

If you want to handle the value of a property by passing it by reference, you must use an auxiliary variable:

```
a := p.Age; Inc(a);
p.Age := a;
```

However, the properties of the corresponding types can be used in the left part of the assignment operations `+= -= *= /=`:

```
p.Age += 1;
```

Properties are very useful when working with visual objects because they allow you to automatically redraw the object if you change any of its visual characteristics. For example, if you create a button of type `Bi Button`, then to visually change its width it is sufficient to assign a value to its `width` property:

```
bi.Width := 100;
```

The procedure to write this property to the `fwidth` private field will look something like this:

```
procedure SetWidth(w: integer);
begin
  if (w>0) and (w<>fWidth) then begin fWidth := w;
      end button redraw code;
```

Note the second part of the condition in the **`if`** statement: `w<>fwidth`. Adding this check avoids unnecessary redrawing of the button if its width does not change.

Index properties

Index properties behave similarly to array fields and are usually used to access container elements. As with normal properties, when using index properties, some actions can be performed in passing.

The index property is described in the class as follows:

```
property Prop[index description]: type read IndexedPropertyReader
write IndexedPropertyWriter;
```

The *IndexedPropertyReader* can be:

- *is the name of the function to read the index property*;
- *an expression* of the corresponding type (in this case the index property is called *extended*).

The IndexedPropertyWriter can be:

- *is the name of the procedure for writing the index property*;
- *operator* (in this case the index property is called *extended*). A predefined variable value is available in the operator, into which a value is placed to change the property.

The read function and write procedure must be methods of this class and have the following form:

```
function GetProp(index description): type; procedure
SetProp(index description; value: type);
```

In the simplest case of a single index, the description of the index property looks like this:

```
property Prop[ind: index type]: type read GetProp write
SetProp;
```

Whenever we assign `a.Prop[ind] := value` to an object `a` containing the `Prop property`, the procedure `a.setprop(ind,vaiue)` is called, and when we read the value `of a.Prop[ind]`, the function `a.GetProp(ind)` is called.

An index property followed by **`default`** keyword followed by `;` , is called ***default index property*** and allows to use class objects as arrays, i.e. to use a`[ind]` *instead of* `a.prop[ind]`. The fundamental difference between index properties and array fields is that the index type can be arbitrary (in particular, string). This makes it easy to

implement so-called associative arrays whose elements are indexed by strings.

In the following example, the index property is used to paint the checkerboard cells white or in graphic mode.

```
uses GraphWPF; const n = 8; sz = 50;
type ChessBoard = class private a: array [,] of boolean :=
new boolean[n,n]; procedure setCell(x,y: integer; value:
boolean); begin if value then
        Brush.Color := Colors.White else Brush.Color :=
      Colors.Gray; FillRectangle((x-1)*sz+1,(y-
      1)*sz+1,sz,sz); a[x-1,y-1] := value;
    end;
    function getCell(x,y: integer) := a[x-1,y-1];
  public property Cells[x,y: integer]: boolean read getCell
write setCell; default;
end;

var c: ChessBoard := new ChessBoard;

begin
  for var x:=1 to n do
  for var y:=1 to n do
    c[x,y] := Odd(x+y);
end.
```

Instead of writing separate `getCell` and `setCell` methods for read and write access to the property, you can use *the advanced index properties*:

```
type
  ChessBoard = class
  private
    a: array [,] of boolean := new boolean[n,n];
  public
    property Cells[x,y: integer]: boolean
      read a[x-1,y-1]
      write begin if value then Brush.Color := Colors.White
      else Brush.Color := Colors.Gray; FillRectangle((x-
      1)*sz+1,(y-1)*sz+1,sz,sz); a[x-1,y-1] := value;
      end; default;
end;
```

Inheritance

A class can be inherited from another class. The class that is inherited is called ***a base class*** (***superclass***, ***ancestor***), and the class that is inherited is called ***a derived class*** (***subclass***, ***descendant***). When inheriting, all fields, methods and properties of a base class pass to the derived class; besides, new fields, methods and properties can be added and old methods can be overridden (replaced). Constructors are inherited by special rules that are discussed here.

When describing a class, its base class is specified in parentheses after the word **`class`**.

For example:

```
type BaseClass = class procedure p;
    procedure q(r: real); end;
  MyClass = class(BaseClass) procedure p;
    procedure r(i: integer); end;
```

In this example, the procedure `p` is overridden and the procedure `r` is added to the `MyClass` class.

If you don't specify a base class name, it is assumed that the class inherits from the `object` class, the ancestor of all classes. For example, `BaseClass` is inherited from `Object`.

Method redefinition in inheritance is discussed here.

The word **`class`** can be preceded by the keyword **`sealed`**, in which case it is forbidden to inherit from the class.

Overriding methods

A method of a base class can be overridden (replaced) in subclasses. If you want to call a method of a base class, use the inherited command word. For example:

```
type Person = class private
    name: string;
    age: integer;
  public
    constructor Create(nm: string; ag: integer);
    begin name := nm; age := ag;
    end;
    procedure Print;
    begin
      Writeln('Name: ',name,' Age: ',age);
    end; end;

  Student = class(Person)
  private
    course, group: integer;
  public constructor Create(nm: string; ag,c,gr: integer);
    begin inherited Create(nm,ag); course := c; group := gr;
    end;
    procedure Print;
    begin
      Inherited Print;
      Writeln('Course: ',course,' Group: ',group); end;
  end;
```

Here the `Print` method of the `Student` derived class first calls the `Print` method inherited from the `Person` base class using the **`inherited`** `Print` construct. Similarly, the `Create` constructor of the `Student` class calls the `Create` constructor of the `Person` base class first, also using the **`inherited constructor`** word.
The rules of constructor inheritance are discussed here.

Note that in this case the constructor of the base class is called as a procedure, not as a function, and no new object is created.

If a base class method with the same parameters is called in a method, you can use the inherited notation without specifying the method name and parameters. For example, the `student.print` method can be written like this:

```
    procedure Print;
```

```
begin
  inherited;
  writeeln('Курс: ',course,' Группа: ',group); end;
```

Constructor inheritance

The rules of constructor inheritance are quite complex. Different programming languages have different solutions for this. In particular, in Delphi Object Pascal, all constructors are inherited. In .NET, by contrast, constructors are not inherited. The reason for this is that each class must be responsible for initializing its instances. The only exception in .NET is that if a class does not define constructors at all, a constructor without parameters, called the default constructor, is automatically generated.

PascalABC.NET has an intermediate solution. If a class does not define constructors, then all ancestor constructors are automatically generated in the descendant by calling the corresponding ancestor constructors (we can also say that they are inherited). If the class defines constructors, the ancestor constructors are not generated. The default constructor, if not explicitly defined, is automatically generated anyway and is `protected`.

Also, in .NET it is mandatory that the ancestor constructor be called first in the descendant constructor; in Object Pascal this is optional. If in **PascalABC.NET** the ancestor constructor is called from the descendant constructor, this call must be the first operator. If the ancestor constructor is not explicitly called from the descendant constructor, the default ancestor constructor (i.e., without parameters) is called implicitly as the first operator in the descendant constructor. If the ancestor does not have such a constructor (it can be a class compiled by another .NET compiler or it can be a member of the standard class library - all classes compiled by **PascalABC.NET** have a default constructor), a compilation error occurs.

For example:

```
type A = class
    i: integer;
    // the default constructor is not explicitly defined, so it
is generated automatically
    constructor Create(i: integer);
    begin Self.i := i;
    end;
  end;
  B = class(A)
```

```
    j: integer;
    constructor Create;
    begin
      // the default constructor of the base class is called
automatically
      // the default constructor is explicitly defined, so it
is not automatically generated
      j := 1;
    end;
    constructor Create(i,j: integer);
    begin
      inherited Create(i); Self.j := j;
    end;
  end;
  C = class(B)
  // the class does not define constructors, so
  // default constructor and constructor Create(i,j: integer)
  // are generated automatically by calling the corresponding
ancestor constructors in their bodies
  end;
```

Virtual methods and polymorphism

Polymorphism (from *the Greek* "many forms") is a property of classes related by inheritance to have different implementations of methods included in them, and the ability of a base class variable to call methods of the class whose object is contained in that variable at the time the method is called.

Polymorphism is used in a situation where a group of interrelated objects needs to perform a single action, but each of these objects needs to perform the specified action in its own way (i.e. the action has many forms). To do this, a base class is defined for all objects with virtual methods provided for changing behavior, and then these methods are overridden in descendants.

To explain, let's look at method overrides in a subclass:

```
type Base = class public procedure Print; begin
      writeeln('Base');
    end; end; Derived = class(Base) public
    procedure Print;
    begin
      writeln('Derived'); end; end;
```

Let's assign an object of the derived `Derived` class `to` the Base class variable and call the `Print` method.

```
var b: Base := new Derived; b.Print;
```

Which version of the `Print` method is called - the `Base` class or the `Derived` class? In this case, the decision will be made at the compilation stage: the `Print` method of the `Base` class declared in the description of the y variable will be called. They say that there is ***an early association of*** the method name with its body. If, however, the decision about which method to call is made at runtime, depending on the actual type of object to which the Y variable refers, then the `Derived.Print` method is called (also said to be ***late***). Methods with late binding are called ***virtual methods, and*** the base class variable through which a virtual method is invoked is called ***a polymorphic variable***. Thus, polymorphism is implemented by calling virtual functions through a base class variable. The class type that is stored in this variable at runtime is called ***a dynamic type*** of this variable.

In order to make a method virtual, in the declaration of that method you should specify the keyword **virtual** after the header followed by ;. To override a virtual method you should use the **override** keyword:

```
type Base = class public procedure Print; virtual; begin
      writeeln('Base'); end;
  end;
  Derived = class(Base) public
    procedure Print; override; begin
      writeln('Derived'); end;
  end;
```

Now in a similar section of code.

```
var b: Base := new Derived; b.Print;
```

Print method of Derived class is called because the decision to call the method is postponed to the stage of program execution. Print methods are said to be tied in ***a chain of virtuality***. The keyword **reintroduce** is used to break it (not to call methods in subclasses virtually):

```
type
  DerivedTwicel = class(Derived)
  public
    procedure Print; reintroduce;
    begin
      writeeln('DerivedTwicel');
    end;
  end;
```

If we want to start a new chain of virtuality, we should use both **virtual** and **reintroduce**:

```
type
  DerivedTwice2 = class(Derived) public procedure Print;
  virtual; reintroduce; begin
      writeeln('DerivedTwice2');
    end; end;
```

If you redefine a virtual function as non-virtual without the **reintroduce** keyword, there will be no error, just a warning that the virtuality chain is broken. So the **reintroduce** keyword in this situation only suppresses the warning output.

When overriding a virtual method in a subclass, its access level must not be lower than in the base class. For example, a **public** virtual

method cannot be overridden in a **private-method** subclass.

Abstract methods and classes

Methods intended to be overridden in subclasses are declared with the **abstract** keyword and are called abstract. These methods are virtual, but the **virtual keyword** need not be used. For example:

```
type Shape = class private x,y: integer;
  public constructor Create(xx,yy: integer); begin
      x := xx; y := yy; end;
    procedure Draw; abstract; end;
```

Classes containing abstract methods are also called *abstract classes*. Instances of these classes cannot be created.

Classes with abstract methods are used as "semi-finished products" to create other classes. For example:

```
type
  Point = class(Shape)
  public procedure Draw; override; begin
      PitPixel(x,y,Color.Black);
    end; end;
```

Using **override** when overriding abstract methods is mandatory because abstract methods are a kind of virtual methods.

You can explicitly declare a class as abstract by using the **abstract** keyword. Usually abstract classes contain abstract methods, but not necessarily:

```
type A = abstract class(Shape) end;
```

Overload operations

Operation overloading is a language tool that allows you to enter operations on user-defined types. In **PascalABC.NET** you can use only predefined operation icons. Operation overloading for type `t`, which is a class or a record, is done by a static (class) method function with a special name *operatorSignOperations*. Overloading of special operations `+=`, `-=`, `*=`, `/=` is performed with the help of static procedure-method, the first parameter of which is passed by reference.

For example:

```
type Complex = record re,im: real; static function
operator+(a,b: Complex): Complex; begin Result.re := a.re +
b.re;
      Result.im := a.im + b.im; end; static function
    operator=(a,b: Complex): boolean; begin
      Result := (a.re = b.re) and (a.im = b.im); end;
 end;
```

The following rules apply to overloading operations:

1. You can overload all operations except `@` (address taking), **`as`**, **`is`**, `new`. You can also overload special binary operations `+=`, `-=`, `*=`, `/=` that do not return values.
2. You can only overload operations that have not yet been overloaded.
3. The type of at least one operand must match the type of the class or record within which the operation is defined.
4. Overloading is performed using a static function-method, the number of parameters of which coincides with the number of parameters of the corresponding operation (2 - for binary, 1 - for unary).
5. Overloading operations `+=`, `-=`, `*=`, `/=` for the corresponding operators is done by a static procedure-method whose first parameter is passed by reference and has the type of record or class in which the given operation is defined, the second is passed by value and is compatible with the first by assignment. The rest of the operations are overloaded using static function-methods.
6. Type conversion operations are defined by static functions that

use **operator implicit** (for implicit type conversion) or **operator explicit** (for explicit type conversion) instead of a name.

For example:

```
type
  Complex = record - -- static function operator implicit(d:
     real): Complex; begin
       Result.re := d;
       Result.im := 0;
     end;
     static function operator explicit(c: Complex): string;
     begin
       Result := Format('({0},{1})',c.re,c.im);
     end;
     static procedure operator+=(var c: Complex; value:
Complex);
     begin
       c.re += value.re;
       c.im += value.im;
     end;
     static function operator+(c,c1: Complex): Complex;
     begin
       Result.re := c.re + c1.re;
       Result.im := c.im + c1.im;
     end;
  end;
```

You can overload operations using extension methods - in this case you should not write the word class when describing a subroutine. For example, this is how the system module implements adding a number to a string:

```
function operator+(str: string; n: integer): string;
extensionmethod;
begin
  result := str + n.ToString;
end;
```

You can overload advanced assignment operations +=, -=, etc:

```
type t0 = class
    x: integer;
    static function operator+=(a: t0; i: integer): t0; begin
      a.x += i;
    end;
  end;
begin var t := new t0; t += 2;
end.
```

It is important to note that if there is a property to the left of the extended assignment, the overloaded form of the extended assignment is ignored and the extended assignment itself is expanded into an assignment and the corresponding operation:

```
type t0 = class
    x: integer;
    static function operator+=(a: t0; i: integer): t0;
    begin a.x += i;
    end;
  end; t1 = class property p1: t0 read ... write ...;
  end;
begin var t := new t1; t.p1 += 2;
end.
```

In this code, the last extended assignment operator will be forcibly expanded to

```
t.p1 := t.p1 + 2;
```

and since the + operation is not overloaded for t0, the compiler will generate an error.

Static classes, fields, methods, properties and constructors

You can declare so-called *static* fields, properties and methods in a class. They do not belong to a specific instance of the class, but are associated with the class. To call them, use dot notation, and use the class name before the dot instead of the object name. To make a field or property or method static, the keyword **static** must precede its name. When describing static properties, only static fields or methods can be specified in read and write sections.

For example, let's define for class Person the number of created objects of this class as a static field and organize access to this field for reading using static function. After each constructor call the value of static field will increase by 1:

```
type Person = class
  private
    name: string;
    age: integer;
    static cnt: integer := 0;
  public
    static property Coun: integer read cnt; constructor (n:
    string; a: integer);
    begin
      cnt += 1;
      name := n;
      age := a;
    end;
    static function Count: integer;
    begin
      Result := cnt;
    end;
  end;
begin
  var p: Person := new Person('Ivanov',20);
  var p1: Person := new Person('Petrov',18);
  Writeln(Person.Count); // call the class method Count end.
```

Unlike static fields and methods, regular fields and methods are called instance fields. You can access both instance and static fields from conventional methods, but only static fields can be accessed from static methods.

Similarly, you can also define a static constructor to automatically initialize class fields. The static constructor is described with the

static keyword and is guaranteed to be called before calling any static method and creating the first object of that class.

For example, let's define in `Person` class a static field - an array of objects of `Person` type - and initialize it in static constructor. Then you can use that array to implement `RandomPerson` function, which returns random object of `Person` type:

```
type
  Person = class
  private
    static arr: array of Person;
    name: string;
    age: integer;
  public
    static constructor;
    begin
      SetLength(arr,3);
      arr[0] := new Person('Ivanov',2 0),-
      arr[1] := new Person('nempoea',19);
      arr[2] := new Person('nonoe',35);
    end;
    //...
    static function RandomPerson: Person;
    begin
      Result := arr[Random(3)];
    end;
  end;
const cnt = 10;
begin
  var a := new Person[cnt];
  for var i:=0 to a.Length-1 do
    a[i] := Person.RandomPerson;
end.
```

The class can also be described as static:

```
type
  MyStatic = static class
    static Pi: real := 3.14;
    static function Pi2 := Pi * Pi; end;
```

In this case all its methods, fields, properties and constructors must be static. It is forbidden to create instances of static classes. In addition, you cannot inherit from static classes, a static class cannot be an ancestor, and you cannot instantiate a generalized class with a static class. For Delphi compatibility, static class members can also be declared with the keyword **class**, which in this context is synonymous to **static**:

```
type
  MyStatic = static class
    class Pi: real := 3.14;
    class function Pi2 := Pi * Pi; end;
```

Expansion methods

Any existing type stored in the external dll and all types in the standard .NET library can be extended with new methods. An extension method is defined as a procedure or function with the **extensionmethod** modifier. The first parameter of an extension method must necessarily be named Self and belong to an extensible type. Let us compare the two procedures described below:

```
procedure MyPrint(Self: integer);
begin
  writeeln(Self) end;

procedure MyPrintEx(Self: integer); extensionmethod; begin
  writeeln(Self) end;

begin
  MyPrint(1);
  1.MyPrintEx; end.
```

Here MyPrint is a normal procedure with an integer type parameter, MyPrintEx is an extension method of integer type. When called, the first parameter MyPrintEx becomes the object that calls MyPrintEx as a method.

You can extend the type of a sequence, then all classes that are sequences (dynamic one-dimensional arrays, lists List<T>, sets HashSet<T> and SortedSet<T>) will get this method. For example, the PABCSystem module introduces the ForEach extension method for sequences in this way:

```
procedure &ForEach<T>(Self: sequence of T; action: T -> ();
extensionmethod;
begin foreach x: T in Self do action(x);
end;
```

You can use extension methods to overloadoperations:

```
procedure operator+=<T>(a: List<T>; x: T): List<T>;
extensionmethod;
begin
  a.Add(x);
end;
```

In this case the first parameter does not have to be named Self.

There are a number of limitations for extension methods:

- Extension methods cannot be virtual.

- If an extension method has the same name as a normal method, the normal method is preferred.

Attributes

The language has a limited number of standard keyword attributes - for example: public, private and protected access level attributes, static, virtual and override attributes for methods.

Custom attributes (hereafter simply attributes) are special language constructs that allow you to label classes, methods, subroutines, and parameters with some name (possibly with parameters).

For example, for serialization mechanism the class is marked with [Serializable] attribute, and the fields that should not be serialized are marked with [NonSerialized] attribute. For Unit-testing, the test method is marked with the [Test] attribute or the [TestCase] attribute. The attributes are recognized by reflection method (reflexion).

An attribute is a regular class inherited from System.Atribute. For example:

```
type AuthorAttribute = class(System.Attribute) auto
  property Name: string;
  constructor (n: string);
  begin name := n; end;
end;
```

Note that the attribute class usually ends in `Attribute`.

When marking an entity with an attribute, the attribute name is enclosed in square brackets. The attribute constructor also provides the ability to use attributes with parameters. For example, you can mark a class with an `AuthorAttribute` attribute.

```
[Author('Alex')] type My = class
  // ...
end;
```

Note that the `Attribute` ending can be omitted.

You can tell if a class is marked with an attribute by using reflection:

```
begin
  var t := typeof(My);
  var attrs := t.GetCustomAttributes(false);
  foreach var attr in attrs do if attr is AuthorAttribute
    (var auth) then
      Print(auth.Name);
```

```
end.
```

Trace

Anonymous classes

Sometimes it is necessary to generate a class object on the fly without describing the class. Such a class has no name (it is anonymous), but a set of fields is known.

An object of an anonymous class is created as follows:

```
var p := new class(Name := 'Ivanov', Age := 20);
Println(p.Name,p.Age);
```

The p object automatically generates public Name and Age fields of the appropriate types.

Two objects belong to the same anonymous class if they have the same set of fields and those fields belong to the same types. For example:

```
var p1 := new class(Name := 'Petrov', Age := 21); p1 := p;
```

If fields of an unnamed class are initialized by variables, you don't have to write field names - they are generated automatically and their names and types coincide with the variable names and types. For example:

```
var Name := 'Popova';
var Age := 23;
var p := new class(Name, Age);
Println(p.Name,p.Age);
```

The fields of an unnamed class can also be initialized by a variable with a compound name that has dot notation. In this case, the last names in the dot notation are taken as field names. For example:

```
var d := new DateTime(2015,5,15);
var p := new class(d.Day, d.Month, d.Year);
Println(p.Day, p.Month, p.Year);
Println(p);
```

Autoclasses

When describing a class, you can put the word **auto** before the word **class**. Such classes are called autoclasses. For autoclasses, a constructor is automatically generated with parameters that initialize all class fields, as well as a Tostring method that outputs values of all class fields. For example:

```
type Person = auto class name: string; age: integer;
end;
var p := new Person('Ivanov',2 0),- // the autoclass
constructor is automatically generated
Writeln(p); // the automatically generated ToString method
is called
```

Here, unlike in the writeln action, by default the values of all the fields, not just the public ones, are output.

Exception handling: overview

When an error occurs during the execution of a program, a so-called *exception* is generated which can *be caught* and *handled*. An exception is an object of a class derived from the `Exception` class that is generated when an exceptional situation occurs.

There are a number of standard exception types available. You can also define custom exception types.

If an exception is not handled, the program will end with an error. To handle exceptions the operator **`try ... except`**.

Exceptions are usually raised in subroutines because the subroutine developer usually does not know how to handle an erroneous situation. At the point where the subroutine is called, it is usually already known how the exception should be handled. For example, let the following function be developed:

```
function mymod(a,b: integer): integer;
begin
  Result := a - (a div b) * b; end;
```

If you call `mymod(1,0)`, the `System.DivideByZeroException` of integer division by 0 will be thrown.

Consider a naive attempt to handle an error situation within the `mymod` function:

```
function mymod(a,b: integer): integer;
begin if b = 0 then
    writeeln('mymod function: divide by 0');
  Result := a - (a div b) * b; end;
```

Such a solution is bad, because the programmer developing the `mymod` function does not know how it will be used. For example, when calling the `mymod` function in a loop, we will see a repeated error message on the screen.

The easiest way is to leave the original version of the function and handle a `System.DivideByZeroException`:

```
try
  readln(a,b);
  writeln(mymod(a,b) mod (a-1));
  • -- except on System.DivideByZeroException do
writeln('Divide by 0');
end;
```

The difference from the output inside a function is that when we

design a program, we ourselves define the action to be performed when an exception is handled. This can be a specific error message, output to an error file, or an empty statement (in case we want to silently extinguish an exception).

However, this solution has a significant drawback: `System.DivideByZeroException` will be thrown even if a=1 and will not be associated with `mymod` function. To eliminate this drawback, let's define our own exception class and raise it in the `mymod` function:

```
type MyModErrorException = class(System.Exception) end;
function mymod(a,b: integer): integer;
begin if b = 0 then raise new MyModErrorException('Function
mymod: division by 0');
  Result := a - (a div b) * b; end;
```

Then the error handling will look like this:

```
try
  readln(a,b);
  writeln(mymod(a,b) mod (a-1));
  • -- except on System.DivideByZeroException do
writeln('Divide by 0');
  on e: MyModErrorException do writeln(e.Message);
  else writeln('some other exception') end;
```

IF TO MAKE `MYMODErrorException` A UNDERSTANDING CLASS `System.ArithmeticException`, As And `System.DivideByZeroException`, then the LAST CODE can be simplified:

```
type MyModErrorException = class(System.ArithmeticException)
end; - --
try
  readln(a,b);
  writeln(mymod(a,b) mod (a-1));
  -- except on e: System.ArithmeticException do
    writeeln(e.Message);
  else writeln('Some other exception') end;
```

Finally, WE CAN DO the following. We intercept in FUNCTIONS `mymod` exception `System.DivideByZeroException` and In response generate a new one - `MyModErrorException`:

```
function mymod(a,b: integer): integer;
begin
  try
    Result := a - (a div b) * b;
  except
    on e: System.DivideByZeroException do
```

```
      raise new MyModErrorException('Function mymod:
division by 0');
  end;
end;
```

Standard exception classes

All exception classes are descendants of the `system.Exception` class which includes the following interface:

```
type Exception = class
  public constructor Create; constructor Create(message:
    string); property Message: string; // read only property
    StackTrace: string; // read only
  end;
```

The `Message` property returns a message associated with the exception object.

The `StackTrace` property returns the subroutine call stack at the time of exception generation.

Below are some exception classes defined in the `System` namespace and derived from the `System.SystemException` class:

`System.OutOfMemoryException` - not enough memory to execute the program;

`System.StackOverflowException` - stack overflow (usually with multiple nested subroutine calls);

`System.AccessViolationException` - an attempt to access protected memory;

`System.ArgumentException` - invalid value of a subprogram parameter;

`System.ArithmeticException` is a base class of all arithmetic exceptions. Heirs:

- `System.DivideByZeroException` - integer division by 0;
- `System.OverflowException` - Overflow when performing an arithmetic operation or type conversion;

`System.FormatException` - Incorrect parameter format (for example, when converting string to number);

`System.IndexOutOfRangeException` - EXCEPTION of the range of the array index change;

`system.invaiidcastException` - incorrect type conversion;

`System.NullReferenceException` - an attempt to call a method for a null object or to dereference a null pointer;

`System.IO.IOException` - IO error. Heirs:

`System.IO.IOException.DirectoryNotFoundException` - directory not found;

`System.IO.IOException.EndOfStreamException` - an attempt to read beyond the end of the stream;

`System.IO.IOException.FileNotFoundException` - file not found.

User-defined exceptions

To define your own type of exception, it is sufficient to spawn a class that is a descendant of the `Exception` class:

```
type MyException = class(Exception) end;
```

The body of an exception class can be empty, but nevertheless, a new name for the exception type will distinguish it from the rest of the exceptions:

```
try ...
except
  on MyException do
    writeeln('Integer division by 0');
  on Exception do writeln('File is missing');
end;
```

The exception may contain additional information related to the point at which the exception occurred:

```
type
  FileNotFoundException = class(Exception) fname: string;
    constructor Create(msg,fn: string);
    begin inherited Create(msg); fname := fn;
    end;
  end;

• ••

procedure ReadFile(fname: string);
begin
  if not FileExists(fname) then
    raise new FileNotFoundException('File not
found',fname);
end;

• ••

try
  -- except on e: FileNotFoundException do writeln('File
'+e.fname+' not found'); end;
```

Re-generating an exception

To re-generate an exception in the **`except`** block, use the **`raise`** operator without parameters:

```
raise;
```

For example:

```
try ...
except
```

```
  on FileNotFoundException do
  begin
    log.WriteLine('File not found'); // Write to the error
file
    raise;
  end;
end;
```

Examples of exception handling

Example 1. Handling incorrect data entry.

Consider the program.

```
var i: integer;
begin
  readln(i);
  writeln(i);
  writeeln('Program execution continues');
end.
```

If an error occurs while entering data (for example, we enter a wrong number), the program will terminate with an error (input error) and the following `writeln` statements will not be executed.

Let's catch an exception in the **`try`** block:

```
var i: integer;
begin try readln(i); writeln(i); except
    writeeln('Input error');
  end;
  writeeln('Program execution continues'); end.
```

This time, if an input error occurs, the program will not be terminated, but will be passed to the **`except`** block, after which the program will continue. Thus, in the last program only the `writeln(i)` operator will not be executed.

If various exceptions can occur in the **`try`** block, a second form of the **`except`** block with several exception handlers is usually used.

Example 2. Handling various exceptions.

```
var a,b: integer;
assign(f,'a.txt');
try
  readln(a,b);
  reset(f);
  c:=a div b;
except
  on System.DivideByZeroException do writeln('Integer
    division by 0');
  on System.IO.IOException do writeln('No file');
end;
```

It is often necessary to combine exception handling and resource release, regardless of whether an exception occurs or not. In this case the nested operators **`try`** ... **`except`** and **`try`** ... **`finally`**.

Example 3. Nested operators **try** ... **except** and **try** ... **finally**.

```
assign(f,'a.txt');
try
  reset(f);
  try c:=a div b; except on System.DivideByZeroException do
  writeln(,Integer division by 0'); end;
  finally close(f);
  end; except
  on System.IO.IOException do writeln('No file'); end;
```

Note that in this example the exception related to integer division by 0 is handled in the inner **try** block itself, and the exception related to a missing file is handled in the outer try block itself. At the same time, if a file was open, it will be closed regardless of the division by 0 exception.

Interfaces: overview

An interface *is* a data type containing a set of method and property headers intended to be implemented by some class. Interfaces are described in the **type** section as follows:

Interface name = **interface** *declaration of methods and properties*
end;

For the method, only the header is given, for the property the necessary read and write access modifiers are given after the return type.

For example:

```
type
  IShape = interface procedure Draw; property X: integer read;
  property Y: integer read; end;
  ICloneable = interface function Clone: Object; end;
```

Fields and static methods cannot be part of an interface.

A class implements an interface if it implements all interface methods and properties in **the public section**. If the class does not implement at least one method or property of the interface, a compilation error occurs. A class can also implement multiple interfaces. The list of implemented interfaces is specified in brackets after the **class** keyword (if an ancestor name is specified, then after the ancestor name).

For example:

```
type
  Point = class(IShape,ICloneable) private xx,yy: integer;
  public constructor Create(x,y: integer); begin
```

```
    xx    x; yy := y;
  end; procedure Draw; begin; property X: integer read xx;
  property Y: integer read yy; function Clone: Object;
  begin
    Result := new Point(xx,yy); end;
  procedure Print;
  begin write(xx,' ',yy);
  end; end;
```

Interfaces can be inherited from each other:

```
type
  IPosition = interface property X: integer read; property Y:
  integer read; end;
  IDrawable = interface procedure Draw; end;
  IShape = interface(IPosition,IDrawable) end;
```

An interface is essentially an abstract class without any implementation of its methods. For interfaces, in particular, all the rules of object type casting apply: the type of an object implementing an interface can be implicitly cast to an interface type, but the reverse conversion is only done explicitly and can cause an exception if the conversion is not possible:

```
var ip: IShape := new Point(20,30);
ip.Draw;
Point(ip).Print;
```

All methods of the class that implements the interface are virtual without using the **virtual** or **override** keywords. In particular, `ip.Draw` will call the `Draw` method of the `Point` class. However, the virtuality chain of such methods is broken. To continue the chain of virtual methods implementing the interface in subclasses, you should use the keyword **virtual**: **type**

```
Point = class(IShape,ICloneable)

  function Clone: Object; virtual; begin
    Result := new Point(xx,yy);
  end;
end;
```

For interfaces, like for classes, you can also use the **is** and **as** operations:

```
if ip is Point then

var p: Point := ip as Point;
if p<>nil then
  writeeln('npeo6pa3oeaHue successful');
```

Implementing multiple interfaces

Several interfaces may contain the same methods or properties. When inheriting from such interfaces, such identical methods or properties are merged into one:

```
type IShape = interface procedure Draw; property X: integer
read; property Y: integer read; end; IBrush = interface
    procedure Draw;
    property Size: integer read;
  end;
  Brush = class(IShape,IBrush) procedure Draw;
    begin end; end;
```

To solve the problem with identical names in interfaces, in .NET classes can implement interface methods in a so-called **explicit** way, so that an interface method call for a class variable is only possible after an explicit conversion to an interface type. For example:

```
type
  IWindow = interface procedure Menu; end;
  IRestaurant = interface procedure Menu;
  end;
  RestaurantSystem = class(IWindow,IRestaurant) public
procedure IWindow.Menu; // explicit implementation of the
interface method
    begin
      Println('IWindow.Menu');
    end;
    procedure IRestaurant.Menu;
    begin
      Println('IRestaurant.Menu');
    end;
  end;
begin
  var r := new RestaurantSystem;
  IWindow(r).Menu;
  IRestaurant(r).Menu;
  r.Menu; // compilation error!
end.
```

Generalized types: overview

A generic type is a template for creating a class, record or interface, parameterized by one or more types. A class (record, interface) is formed from a class (record, interface) template by substituting specific types as parameters. Parameters are specified after the name of the generalized type in angle brackets. For example, `stack<T> is a template` class of a list of elements of type t, parameterized by type t, and stack<integer> `is a` list `class` with elements of type integer.

Generalized subprogramsare described here.

The following syntax is used to declare a class template:

```
type Node<T> = class
    data: T;
    next: Node<T>;
  public constructor Create(d: T; nxt: Node<T>);
    begin
      data := d; next := nxt;
    end;
  end;
  Stack<T> = class
    tp: Node<T>;
  public procedure Push(x: T);
    begin
      tp := new Node<T>(x,tp);
    end;
    function Pop: T;
    begin
      Result := tp.data;
      tp := tp.next;
    end;
    function Top: T;
    begin
      Result := tp.data;
    end;
    function IsEmpty: boolean;
    begin
      Result := tp = nil;
    end;
  end;
```

The use of the class template is illustrated below:

```
var
  si: Stack<integer>;
```

```
  sr: Stack<real>;
begin
  si := new Stack<integer>;
  sr := new Stack<real>;
  for var i := 1 to 10 do
    si.Push(Random(100));
  while not si.IsEmpty do sr.Push(si.Pop);
  while not sr.IsEmpty do write(sr.Pop,' ');
end.
```

Substitution of a particular type-parameter into a generalized type is called instantiation.

Generalized subprograms: overview

A generic subprogram is a subprogram that is parameterized by one or more types. A subprogram is formed from a generic subprogram by substituting specific types as parameters. Parameters are specified after the subprogram name in angle brackets.

For example, the following generalized function is parameterized with one parameter:

```
function FindFirstInArray<T>(a: array of T; val: T):
integer;
begin
  Result := -1;
  for var i:=0 to a.Length-1 do
    if a[i]=val then
    begin
      Result := i;
      exit;
    end;
end;

var x: array of string;

begin
  SetLength(x,4);
  x[0] := 'Vanya';
  x[1] := 'Kolya';
  x[2] := 'Seryozha';
  x[3] := 'Sasha';
  writeeln(FindFirstInArray(x,'Seryozha'));
end.
```

When calling a generalized subroutine, the type-parameter generalization can be omitted, because the compiler ***outputs*** the types of the template parameters by the types of the actual parameters. In this case, after the output obtained: `T=string`.

The exact match of types is required in the derivation; type conversions are not allowed. For example, when compiling the following code

```
var x: array of real;

begin
  SetLength(x,3);
  x[0] := 1;
  x[1] := 2.71;
  x[2] := 3.14;
  writeeln(FindFirstInArray(x,1)); end.
```

an error will occur. The reason is that the first parameter is of type array of real and the second is of type integer, which does not correspond to any type T in the header of the generalized function. To solve the problem, either change the type of the second parameter to real:

```
FindFirstInArray(x,1.0)
```

or explicitly after the function name in angle brackets specify the type name with which the call is parameterized:

```
FindFirstInArray&<real>(x,1)
```

The use of the & sign here is mandatory, because otherwise the compiler interprets the < sign as "less than".

Not only ordinary subroutines can be generalized, but also methods of classes and methods of another generalized class. For example:

```
type
  Pair<T,Q> = class
    first: T;
    Q;
    function ChangeSecond<S>(newval: S): Pair<T, S>; end;
function Pair<T,Q>.ChangeSecond<S>(newval: S): Pair<T,S>;
begin
  result := new Pair<T,S>;
  result.first := first;
  result.second := newval;
end;
var
  x: Pair<integer,real>;
  y: Pair<integer,string>;
begin
  x := new Pair<integer,real>;
  x.first := 3;
  y := x.ChangeSecond('abc');
  writeeln(y.first, y.second);
end.
```

When finished, this program will output 3abc.

Generalized subprograms as parameters

A generalized subprogram can act as a formal parameter of another generalized subprogram.

For example, the `system.Array` class has several static generalized methods with generalized routines as parameters. For example, `System.Array.Find has` the following prototype:

```
System.Array.FindAll<T>(a: array of T; pred: Predicate<T>):
array of T;
```

and returns a subarray of the array `a of` elements `t` satisfying the condition `pred`.

Here is an example of how to call this function:

```
function f(x: integer): boolean;
begin
  Result := ;
end;

var a := Seq(1,3,6,5,8);
var b := System.Array.FindAll(a,x -> x mod 2 = 0);
```

This returns an array `b` containing all even values of array `a in the` same order.

Restrictions on parameters of generalized subprograms and classes

By default, you can do only a limited set of actions with variables of the parameter type of a generalized class or subprogram inside methods of generalized classes and generalized subprograms: assignment and equality comparison (note that in NET equality comparison inside generalizations is forbidden!).

For example, this code will work:

```
function Eq<T>(a,b: T): boolean;
begin
  Result := a = b;
end;
```

You can also use assigning a default value to a variable that has the parameter type of a generalized class or subprogram using the `default(T)` construct - the default value for type `t` (`nil` for referential types and zero for dimensional types):

```
procedure Def<T>(var a: T);
begin
  a := default(T);
end;
```

However, this code

```
function Sum<T>(a,b: T): T;
begin
  Result := a + b;
end;
```

will cause a compilation error before instantiation (creating an instance with a particular type). This behavior in .NET is radically different from templates in C++, where any operations with template parameters can be used in the template code, and the error can occur only at the moment of instantiation with a specific type.

To allow certain actions on variables of the parameter type of a generalized class or subprogram, the restrictions on generalized parameters set in the section after the subprogram or class header are used:

```
type
  MyPair<T> = class
    where T: System.ICloneable;
  private x,y: T;
```

```
  public
    constructor (x,y: T);
    begin
      Self.x := x;
      Self.y := y;
    end;
    function Clone: MyPair;
    begin
      Result := new MyPair<T>(x.Clone,y.Clone);
    end;
  end;
```

The following restrictions are listed in the **where** section, separated by commas:
In 1st place: the word **class** or the word **record** or the name of the ancestor class.
In 2nd place: a comma-separated list of interfaces to be implemented. In 3rd place: the word **constructor**, indicating that this type should have a default constructor.

In this case, each of the seats, except one, can be empty.

For each type-parameter there can be a different **where** section, each **where** section ends with a semicolon.

Example. A generalized function to find the minimum element in an array. The elements must implement the `IComparable<T>` interface.

```
function MinElem<T>(a: array of T): T;
  where T: IComparable<T>;
begin
  var min := a[0];
  for var i := 1 to a.High do
    if a[i].CompareTo(min)<0 then min := a[i];
  Result := min;
end;
```

Unfortunately, there is no way to use the entry a[i]<min because the operations are not part of the interfaces.

La mbda-o-rare

A lambda-expression is a special kind of expression that is replaced at the compilation stage with the name of a subprogram corresponding to the lambda-expression and generated by the compiler "on the fly".

The full syntax of lambda expressions is laid out here.

Here we talk aboutcapturing lambda-expression variables from an external context.

Lambda expressions may not be used when initializing class or record fields, within nested subprograms, in a subprogram if there is a nested subprogram, in a module initialization section.

Lambda expressions may not be used together with label labels and goto statements in the same subroutine.

The syntax of lambda expressions is quite complex and is illustrated in this paragraph by examples.

Example 1.

```
var f: integer -> integer := x -> x*x;
f(2);
```

The entry `x -> x` is a lambda expression, which is a function with one parameter x of type `integer` that returns x*x of type `integer.` Based on this entry the compiler generates the following code:

```
function #fun1(x: integer): integer;
begin
  Result := x*x;
end;
- • ••
var f: integer -> integer := #fun1;
f(2);
```

Here `#funi` is the name generated by the compiler. In addition, the `#funi` function code is also generated by the compiler.

Example 2. Filtering of even

Usually a lambda expression is passed as a subroutine parameter. For example, in the following code

```
var a := Seq(3,2,4,8,5,5);
a.Where(x -> x mod 2 = 0).Print;
```

The lambda expression `x -> x mod 2 = 0` sets the condition for

selecting even numbers from the array a.

Example 3. Sum of squares

```
var a := Seq(1,3,5);
writeeln(a.Aggregate(0,(s,x)->s+x*x));
```

Sometimes it is necessary to explicitly specify the type of parameters in the lambda-expression.

Example 4. Selecting an overloaded version of a procedure with a lambda parameter.

```
procedure p(f: integer -> integer);
begin
  write(f(1));
end;

procedure p(f: real -> real);
begin
  write(f(2.5));
end;

begin
  p((x: real)->x*x);
end.
```

In this example, calling p(x -> x) will cause a compilation error because the compiler cannot choose which version of procedure p to choose. Specifying the type of the lambda parameter helps eliminate this ambiguity.

Example 5. Lambda procedure.

```
procedure p(a: integer -> ();
begin
  a(1) end;

begin
  p(procedure(x) -> write(x));
end.
```

Capturing variables in a lambda expression

A lambda expression can use variables from an external context. Such variables are called captured lambda expressions.

Example 1. Capturing a variable in a Select query.

```
begin var a := Seq(2,3,4);
  var z := 1;
  var q := a.Select(x->x+z);
  q.Println;
  z := 2;
  q.Println;
end.
```

Here the lambda expression `x->x+z` captures the external variable z. It is important to note that when the value of the variable z changes, the query `a.Select(x->x+z),` stored in the variable q, is executed with the new value of z.

Example 2. Accumulation of the sum in an external variable.

```
begin
  var sum := 0;
  var AddToSum: integer -> () := procedure (x) -> begin sum
+= x; end;

  AddToSum(1);
  AddToSum(3);
  AddToSum(5);

  writeln(sum); end.
```

Sequence methods

All sequences have many sequence processing methods implemented as extensionmethods.

List of sequence methods

- Printmethods (PascalABC.NET only)
- Filtering method Where
- Select projection method
- SelectMany projection method
- methodsды Take,TakeWhile,SkiSkip,SkipWhile
- Sorted,SortedDescendingmethods (PascalABC.NET only)
- methodsды OrderBy,OrderByDescendingg
- The ThenBy,ThenByDescending methods
- ForEach method (PascalABC.NET only)
- Concat Method
- JoinIntoStringmethod (PascalABC.NET only)
- *Zip* Method
- Distinct method
- Union,Intersect,Except methods
- Reverse method
- The SequenceEqual method
- methodsды First,FirstOrDefault
- methodsды Last,LastOrDefault
- methodsды SinSingle,SingSingleOrDefault
- DefaultIfEmpty method
- methodsды ElementAt,ElementAtOrDefault
- Methodsды AnAny, All
- Count Methods
- Method Contains
- Aggregate method
- methodsды Sum,AveraAverage
- methodsды Min,Max
- Join method
- GroupJoin method
- GroupBy method
- AsEnumerable method
- methodsды ToArray,ToList
- ToDictionary method
- ToLookup method
- The OfType method
- Cast Method

Print methods

Description of methods

The methods are given for a sequence `of t`. **`function`**
`Print(delim: string := ' '): sequence of T;`

Displays the sequence on the screen, using delim as a delimiter.

`function` `Println(delim: string := ' '): sequence of T;`

Displays the sequence on the screen, using delim as a delimiter, and jumps to a new line.

Example

```
begin
  var a := Arr(1,3,5);
  a.Println;
  ReadLines('a.txt').Println(NewLine); end.
```

Filtering method Where

Description of methods

The methods are given for a sequence `of t`. The `function Where(predicate: T->boolean): sequence of T;`

Performs filtering of a sequence of values based on a given predicate. Returns a subsequence of values of the original sequence that satisfy the predicate.

`function Where(predicate: (T,integer)->boolean): sequence of T;`

Performs filtering of the sequence of values based on the given predicate, taking into account the element index. Returns a subsequence of values of the original sequence that satisfy the predicate.

Example

```
begin
  var a := Arr(1,2,3,5,6);
  a.Where(x -> x mod 2 = 0).Println; // 2 6 end.
```

Select projection method

Description of methods

The methods are given for the sequence `of t`. The `function Select<Res>(selector: T->Res): sequence of Res;`

Projects each element of the sequence onto another element using the selector function. Returns the sequence of elements resulting from the projection.

`function Select<Res>(selector: (T,integer)->Res): sequence of Res;`

Projects each element of the sequence onto another element using the selector function, taking the index of the element into account. Returns the sequence of elements resulting from the projection.

Example

```
begin
  var a := Arr(1,2,3,4,5,6);
  a.Select(x -> x*x).Println; // 1 4 9 16 25 36 end.
```

SelectMany projection method

Description of methods

The methods are given for the sequence `of t`. The `function SelectMany<Res>(selector: T-> sequence of Res): sequence of Res; Projects` each element of the sequence into a new sequence and merges the resulting sequences into one sequence. Returns the merged sequence.

```
function SelectMany<Res>(selector: (T,integer)-> sequence of Res): sequence of Res;
```

Projects each sequence element into a new sequence, taking the element index into account, and combines the resulting sequences into one sequence. Returns the merged sequence.

```
function SelectMany<Coll,Res>(collSelector: (T,integer)- > sequence of Coll; resultselector: (T,Coll)->Res): sequence of Res;
```

Projects each sequence element into a new sequence, combines the resulting sequences into one, and calls the result selector function for each element of that sequence. The index of each element in the original sequence is used in the intermediate projected form of that element. Returns the combined sequence.

```
Function SelectMany<Coll,Res>(collSelector: T-> sequence of Coll; resultselector: (T,Coll)->Res): sequence of Res;
```

Projects each sequence element into a new sequence, merges the resulting sequences into one, and calls the result selector function for each element of that sequence. Returns the merged sequence.

Example

```
begin
  var a := Arr(Arr(1,2,3),Arr(4,5,6),Arr(7,8,9));
  a.SelectMany(x -> x).Println; // 1 2 3 4 5 6 7 8 9 end.
```

Take, TakeWhile, Skip, SkipWhile methods

Description of methods

The methods are given for a sequence `of t`. `function Take(count: integer): sequence of T;` Returns a sequence of count elements from the beginning of the sequence.

`function TakeWhile(predicate: T->boolean): sequence of T;`

Returns the chain of sequence elements satisfying the specified condition, up to the first non-satisfying one.

`function TakeWhile(predicate: (T,integer)->boolean): sequence of T;`

Returns the chain of sequence elements satisfying the specified condition, up to the first non-satisfying one (the element index is taken into account).

`function Skip(count: integer): sequence of T;`

Skips count elements in the sequence and returns the remaining elements.

`function SkipWhile(predicate: T->boolean): sequence of T;`

Skips through the elements in the sequence as long as they satisfy the given condition, and then returns the remaining elements.

`function SkipWhile(predicate: (T,integer)->boolean): sequence of T;`

Skips through the elements in the sequence as long as they satisfy the given condition, and then returns the remaining elements (the element index is taken into account).

Example

```
begin
var a := Arr(1,2,3,4,5,6);
     a.Take(3 I.Printin; // 1 2 3
)             l.Printin; // 4 5 6
     a.Skip(3 l.Take(3).Println; // 3 4 5
a.TakeWhile(x -> x<3).Println; // 1 2
a.SkipWhile(x -> x<5).Println; // 5 6 end.
```

Sorted, SortedDescending methods

Description of methods

The methods are given for a sequence `of t`. **The function** `Sorted(): sequence of T;` Returns an ascending sorted sequence.

`function SortedDescending(): sequence of T;`

Returns the sequence sorted in descending order.

Example

```
begin
     var a := Arr(6,2,7,4,8,1);
     a.Sorted.Println; // 1 2 4 6 7 8
     a.SortedDescending.Println; // 8 7 6 4 2 1
   end.
```

OrderBy, OrderByDescending methods

Description of methods

The methods are given for the sequence `of t`.

```
function OrderBy<Key>(keySelector: T->Key):
System.Linq.IOrderedEnumerable<T>;
```

Sorts sequence elements in ascending key order and returns the sorted sequence. KeySelector is a function that projects an element onto a key.

```
function OrderBy<Key>(keySelector: T->Key; comparer:
IComparer<Key>): System.Linq.IOrderedEnumerable<T>;
```

Sorts the elements of the sequence in ascending order using comparer and returns the sorted sequence. keySelector is a function that projects an element onto a key.

```
function OrderByDescending<Key>(keySelector: T->Key):
System.Linq.IOrderedEnumerable<T>;
```

Sorts the sequence elements in descending key order and returns the sorted sequence. keySelector is a function that projects an element onto the key.

```
function OrderByDescending<Key>(keySelector: T->Key; comparer:
IComparer<Key>): System.Linq.IOrderedEnumerable<T>;
```

Sorts the sequence elements in descending order using comparer and returns the sorted sequence. keySelector is a function that projects an element onto a key.

Example

```
begin
  var a := Agg(('Ivanov',20),('Popov',21),('Avilov',28));
  a.OrderBy(t -> t[0]).Println;           // (Avilov,28)
(Ivanov,20) (Popov,21)
  a.OrderByDescending(t -> t[1]).Println; // (Avilov,28)
(Popov,21) (Ivanov,20)
end.
```

The ThenBy,ThenByDescending methods

Description of methods

Methods are given for sequence `of t. function ThenBy<Key>(keySelector: T->Key): System.Linq.IOrderedEnumerable<T>; Performs` additional ordering of sequence elements in ascending key order and returns the sorted sequence. KeySelector is a function that projects an element onto a key.

```
function ThenBy<Key>(keySelector: T->Key; comparer:
IComparer<Key>): System.Linq.IOrderedEnumerable<T>;
```

Performs an additional ordering of the sequence elements in ascending order using comparer and returns the sorted sequence. keySelector is a function that projects an element to a key.

```
function ThenByDescending<Key>(keySelector: T->Key):
System.Linq.IOrderedEnumerable<T>;
```

Performs additional ordering of the sequence elements in descending key order and returns the sorted sequence. keySelector is a function that projects an element onto the key.

```
function ThenByDescending<Key>(keySelector: T->Key; comparer:
IComparer<Key>): System.Linq.IOrderedEnumerable<T>;
```

Performs additional ordering of the sequence elements in descending order using comparer and returns the sorted sequence. keySelector is a function that projects an element onto a key.

Example

```
begin
  var a := Agg((('Ivanov', 20), ('Popov', 21), ('Ivanov',
    18), ('Avilov', 28), ('Ivanov', 25));
  a.OrderBy(t -> t[0]).ThenBy(t -> t[1]).Println;
    // (Avilov,28) (Ivanov,18) (Ivanov,20) (Ivanov,25)
(Popov,21) end.
```

Concat Method

Description of methods

The methods are given for the sequence `of t`. The `function Concat(second: sequence of T): sequence of T;`

Connects the two sequences by appending the second to the end of the first and returning the resulting sequence.

Example

```
begin
var al := Lst(2,3,5);
     var a2 :=Seq(4,7,8);
   al.Concat   (a2).Println; // 2 3 5 4 7
   end.        8
```

Zip Method

Description of methods

The methods are given for a sequence `of t`. `function Zip<TSecond,Res>(second: sequence of TSecond; resultsselector: (T,TSecond)->Res): sequence of Res;`

Combines two sequences using the specified function, taking one element of each sequence and returning the element of the resulting sequence.

Example

```
begin
  var a := Arr(1,2,3);
  var b := Lst(4,5,6);
  a.Zip(b,(x,y) -> x+y).Println; // 5 7 9 end.
```

Distinct method

Description of methods

The methods are given for the sequence `of t`.

`function Distinct(): sequence of T;` Returns the divergent elements of the sequence.

`function Distinct(comparer: IEqualityComparer<T>): sequence of T;`

Returns the differing elements of the sequence, using the comparer to compare values.

Example

```
begin
  var a := Arr('aaa','bbb','ccc','aaa','ccc');
  a.Distinct.Println; // aaa bbb ccc
end.
```

Union,Intersect,Except methods

Description of methods

The methods are given for the sequence **of** t. The **function**

```
Union(second: sequence of T): sequence of T;
```

Finds the union of sets represented by two sequences.

```
function Union(second: sequence of T; comparer:
IEqualityComparer<T>): sequence of T;
```

Finds the union of sets represented by two sequences using the specified comparator.

```
function Intersect(second: sequence of T): sequence of T;
```

Finds the intersection of sets represented by two sequences.

```
function Intersect(second: sequence of T; comparer:
IEqualityComparer<T>): sequence of T;
```

Finds the intersection of sets represented by two sequences, using the specified comparator to compare values.

```
function Except(second: sequence of T): sequence of T;
```

Finds the difference of sets represented by two sequences.

```
function Except(second: sequence of T; comparer:
IEqualityComparer<T>): sequence of T;
```

Finds the difference of sets represented by two sequences, using the specified comparator to compare values.

Example

```
begin
  var a := Range(1,5);
  var b := Range(3,7);
  a.Union(b).Println;      // 1 2 3 4 5 6 7
  a.Intersect(b).Println; // 3 4 5
  a.Except(b).Println;     // 1 2
end.
```

Reverse method

Description of methods

The methods are given for a sequence `of t`. `Function Reverse(): sequence of T;` Returns an inverted sequence.

Example

```
begin
  var a := Range(1,9);
  a.Reverse.Println; // 9 8 7 6 5 4 3 2 1 end.
```

The SequenceEqual method

Description of methods

The methods are given for a sequence `of t`. `function SequenceEqual(second: sequence of T): boolean;`

Determines if two sequences match. `function SequenceEqual(second: sequence of T; comparer: IEqualityComparer<T>): boolean;`

Determines if two sequences match, using the specified comparator to compare elements.

Example

```
begin
  var a := Arr(1,2,3);
  var b := Lst(1,2,3); a.SequenceEqual(b);
end.
```

First, FirstOrDefault methods

Description of methods

The methods are given for a sequence `of t`. `Function First(): t;`
Returns the first element of the sequence.

```
function First(predicate: T->boolean): T;
```

Returns the first element of the sequence that satisfies the specified condition.

```
function FirstOrDefault(): T;
```

Returns the first element of the sequence or the default value if the sequence contains no elements.

```
function FirstOrDefault(predicate: T->boolean): T;
```

Returns the first element of the sequence that satisfies the condition, or the default value if no such elements are found.

Example

```
begin
  var a := Arr(1,2,3,4);
  Println(a.Skip(2).First); // 3
  Println(a.First(x -> x mod 2 = 0); // 2
  Println(a.FirstOrDefault(x -> x>5); // 0 end.
```

Last, LastOrDefault methods

Description of methods

The methods are given for a sequence `of t`. `function Last(): T;`
Returns the last element of the sequence.
`function Last(predicate: T->boolean): T;`

Returns the last element of the sequence that satisfies the specified condition.
`function LastOrDefault(): T;`

Returns the last element of the sequence or the default value if the sequence contains no elements.
`function LastOrDefault(predicate: T->boolean): T;`

Returns the last element of the sequence that satisfies the specified condition, or the default value if no such elements are found.

Example

```
begin
  var a := Arr(1,2,3,4);
  Println(a.Last); // 4
  Println(a.Last(x -> x mod 2 = 0); // 4
  Println(a.LastOrDefault(x -> x>5); // 0 end.
```

Single, SingleOrDefault methods

Description of methods

The methods are given for a sequence `of t. function Single(): T;` Returns a single element of the sequence and generates an exception if the number of elements in the sequence is different from 1.

```
function Single(predicate: T->boolean): T;
```

Returns the only sequence element that satisfies the given condition, and generates an exception if there is more than one such element.

```
function SingleOrDefault(): T;
```

Returns a single sequence element or the default value if the sequence is empty; an exception is generated if there is more than one element in the sequence.

```
function SingleOrDefault(predicate: T->boolean): T;
```

Returns the only sequence element that satisfies the given condition, or the default value if no such element exists; if more than one element satisfies the condition, an exception is generated.

Example

```
begin
var a := Arr(1,2,3,4);
     Println(a.Single); //exception
     Println(a.Single(x -> x>3); // 4
   Println(a.SingleOrDefault(x -> x>5); // 0 end.
```

DefaultIf Empty method

Description of methods

The methods are given for a sequence `of t`. `function DefaultIfEmpty(): sequence of T;` Returns the elements of the specified sequence or a single-element collection containing a default type parameter value if the sequence is empty.
`function DefaultIfEmpty(defaultValue: T): sequence of T;`

Returns the elements of the specified sequence or a single-element collection containing the specified value if the sequence is empty.

Example

```
begin
  var a := Arr(1,2,3,4);
  a.Skip(4).DefaultIfEmpty.Println; // 0 end.
```

ElementAt, ElementAtOrDefault methods

Description of methods

The methods are given for a sequence `of t`. `function ElementAt(index: integer): T;` Returns an element at the specified index in the sequence.

`function ElementAtOrDefault(index: integer): T;`

Returns the item at the specified index in the sequence or the default value if the index is outside the valid range.

Example

```
begin
  var a := Arr(1,2,3,4);
  Println(a.ElementAt(2)); // 3
  Println(a.ElementAtOrDefault(10)); // 0 end.
```

Methods Any, All

Description of methods

The methods are given for a sequence `of t. function Any(): boolean;` Checks if the sequence contains any elements.

`function Any(predicate: T->boolean): boolean;`

Checks if any element of the sequence satisfies the given condition.

`function All(predicate: T->boolean): boolean;`

Checks if all elements of the sequence satisfy the condition.

Example

```
begin
     var a := Lst(1,3,5);
     Println(a.All(x -> x mod 2 <> 0)); // True
     Println(a.Any(x -> x mod 2 = 0)); //  False
   end.
```

Count Methods

Description of methods

The methods are given for a sequence `of t. function Count(): integer;` Returns the number of elements in the sequence.

```
function Count(predicate: T->boolean): integer;
```

Returns a number representing the number of sequence elements satisfying the given condition.

```
function LongCount(): int64;
```

Returns a value of type Int64 representing the total number of elements in the sequence.

```
function LongCount(predicate: T->boolean): int64;
```

Returns a value of type Int64, representing the number of sequence elements satisfying the given condition.

Example

```
begin
  var a := Lst(1,3,5,6);
  Println(a.Count(x -> x mod 2 <> 0)); // 3 end.
```

Method Contains

Description of methods

Methods are given for a sequence `of t`. `function Contains(value: T): boolean;` Determines whether the specified element is contained in the sequence, using the default equality check comparer. `function Contains(value: T; comparer: IEqualityComparer<T>): boolean;`

Determines if the sequence contains a given element using the specified comparator.

Example

```
begin
  var a := Lst(1,3,5,6);
  Println(a.Contains(666)); // False
  Println(666 in a); // False end.
```

Aggregate method

Description of methods

The methods are given for a sequence `of t`. `function Aggregate(func: (T,T)->T): T;` Applies an aggregate function to the sequence. Returns the final aggregate value.

`function Aggregate<Accum>(seed: T; func: (Accum,T)->Accum): T;`

Applies an aggregate function to the sequence. The specified initial value is used as the initial value of the aggregate operation. Returns the final aggregate value.

`function Aggregate<Accum,Res>(seed: T; func: (Accum,T)->Accum; resultsselector: Accum->Res): T;`

Applies an aggregate function to the sequence.The specified initial value serves as the initial value for the aggregate operation, and the specified function is used to select the resulting value. Returns the final aggregate value.

Example

```
begin
  var a := Seq(2,3,5,6);
  Println(a.Aggregate(1,(p,x) -> p*x));
end.
```

Sum, Average methods

Description of methods

The methods are given for a sequence `of ts`. `function Sum(): number;` Calculates the sum of a sequence of numeric type values.

`function Sum(selector: t->number): number;`

Calculates the sum of a sequence of numeric type values resulting from applying a conversion function to each element of the input sequence.

`function Average(): real;`

Calculates the average for a sequence of numeric type values.

`function Average(selector: t->number): real;`

Calculates the average for a sequence of numeric type values resulting from applying a conversion function to each element of the input sequence.

Example

```
begin
  var a := Lst(1,3,5,6);
  Println(a.Sum);
  var b := Agg(('Ivanov',20),('Popov',21),('Avilov',28));
  Println(b.Average(x -> x[1]));
end.
```

Min, Max methods

Description of methods

The methods are given for a sequence `of values of t. function Min(): number;` Calculates the minimum element of a sequence of numeric type values.

```
function Min(selector: t->number): number;
```

Calls a conversion function for each element of the sequence and returns the minimum value of the numeric type.

```
function Max(): number;
```

Calculates the maximal element of a sequence of numeric type values.

```
function Max(selector: t->number): number;
```

Calls a conversion function for each element of the sequence and returns the maximum value of the numeric type.

Example

```
begin
  var a := Lst(1,3,5,6);
  Println(a.Min, a.Max);
  var b := Agg(('Ivanov',20),('Popov',21),('Avilov',28));
  Println(b.Min(x -> x[1]));
end.
```

Join methods

Description of methods

The methods are given for a sequence `of t`. `function Join<TInner,Key,Res>(inner: sequence of Tinner; outerKeySelector: T->Key; innerKeySelector: TInner->TKey; resultselector: (T,TInner)->Res): sequence of Res;`

Combines two sequences based on key mapping into a third sequence. The function resultSelector sets the projection of the elements of two sequences with the same key values into the element of the third sequence.

`function Join<TInner,Key,Res>(inner: sequence of Tinner; outerKeySelector: T->Key; innerKeySelector: TInner->TKey; resultselector: (T,TInner)->Res; comparer: System.Collections.Generic.IEqualityComparer<Key>): sequence of Res;`

Combines two sequences based on key mapping into a third sequence. The function resultSelector sets the projection of elements of two sequences with the same key values into the element of the third sequence. The comparer is used for key comparison.

Example

```
begin
  var people := Agg((1,'Ivanov'),(2,'Popov'),
(3,'Sidorov'));
  var subjects := Agg((1,'History'),(1,'Math'),
(2,'History')
    ,(3,'Math'),(1,'Russian'),(2,'Physics'));

  people.Join(subjects,p->p[0],s->s[0],(p,s)->
(p[1],s[1])).Println(NewLine);
end.
```

```
Conclusion:
(Ivanov,History)    (Ivanov,Mathematics)    (Ivanov,Russian)
(Popov,History) (Popov,Physics) (Sidorov,Mathematics)
```

GroupJoin method

Description of methods

The methods are given for the sequence `of T`. `function GroupJoin<TInner,Key,Res>(inner: sequence of Tinner; outerKeySelector: T->Key; innerKeySelector: TInner->TKey; resultselector: (T,sequence of TInner)->Res): sequence of Res;`

Combines two sequences based on key equality and groups the results. The resultSelector function then projects the key and the sequence of corresponding values onto the element of the resulting sequence. `function GroupJoin<TInner,Key,Res>(inner: sequence of Tinner; outerKeySelector: T->Key; innerKeySelector: TInner->TKey; resultselector: (T,sequence of TInner)->Res; comparer: IEqualityComparer<Key>): sequence of Res;`

Combines two sequences based on key equality and groups the results. The specified comparator is used to compare the keys. The resultSelector function then projects the key and the sequence of corresponding values onto an element of the resulting sequence.

Example

```
begin
  var people := Agg((1,'Ivanov'),(2,'Popov'),
(3,'Sidorov'));
  var subjects := Agg((1,'History'),(1,'Math'),
(2,'History')
        ,(3,'Math'),(1,'Russian'),(2,'Physics'));

  people.GroupJoin(subjects,p->p[0],s->s[0],(p,ss)->
(p[1],ss.Select(x->x[1]))).Println(NewLine);
end.

Conclusion:
(Ivanov,[History,Mathematics,Russian])
(Popov,[History,Physics])
(Sidorov,[Mathematics])
```

GroupBy method

Description of methods

Methods are given for sequence `of t. function GroupBy<Key>(keySelector: T->Key): IEnumerable<IGrouping<Key,T>>; Groups` sequence elements according to the given key selector function and returns a sequence of groups; each group corresponds to one key value. `function GroupBy<Key>(keySelector: T->Key; comparer: System.Collections.Generic.IEqualityComparer<Key>): IEnumerable<IGrouping<Key,T>>;`

Groups the elements of the sequence according to the specified key selector function, compares the keys with the specified comparator, and returns a sequence of groups; each group corresponds to one key value.

```
function GroupBy<Key,Element>(keySelector: T->Key;
elementselector: T->Element): IEnumerable<IGrouping<Key,T>>;
```

Groups the elements of the sequence according to the specified key selector function and projects the elements of each group using the specified function. Returns a sequence of groups; each group corresponds to one key value.

```
function GroupBy<Key,Element>(keySelector: T->Key;
elementselector: T->Element; comparer:
IEqualityComparer<Key>): IEnumerable<IGrouping<Key,Element>>;
```

Groups the elements of the sequence according to the key selector function.The keys are compared using the comparator, the elements of each group are projected using the specified function.

```
function GroupBy<Key,Res>(keySelector: T->Key; resultselector:
(Key,sequence of T)->Res): sequence of Res;
```

Groups the sequence elements according to the specified key selector function and creates a resulting value for each group and its key.

```
function GroupBy<Key,Element,Res>(keySelector: T->Key;
elementsselector: T->Element; resultsselector: (Key,sequence of
Element)->Res): sequence of Res;
```

Groups the elements of a sequence according to the specified key selector function and creates a resulting value for each group and its key.The elements of each group are projected using the

specified function.

```
function GroupBy<Key,Res>(keySelector: T->Key; resultselector:
(Key,sequence of T)->Res; comparer: IEqualityComparer<Key>):
sequence of Res;
```

Groups the elements of the sequence according to the specified key selector function and creates a resulting value for each group and its key.The keys are compared using the specified comparator.

```
function GroupBy<Key,Element,Res>(keySelector: T->Key;
elementselector: System.T->Element; resultselector:
(Key,sequence of Element)->Res; comparer:
IEqualityComparer<Key>): sequence of Res;
```

Groups the elements of the sequence according to the specified key selector function and creates a resulting value for each group and its key.The key values are compared using the specified comparator, and the elements of each group are projected using the specified function.

Example

```
begin
var a := Agg(('Ivanov',3),('Popov',1),('Avilov',1)
  ('Kozlov',3) var,('Donkeys',2),('Rogues',1));
        groups :=a.GroupBy(s->s[1]);

    foreach var g In groups do
    begin
  Print(g.Key+':');
  g.Select(x->x[0]).Println;
    end;
  end.

  Conclusion:
  3: Ivanov Kozlov
  1: Popov Avilov Rogov
  2: Donkeys
```

AsEnumerable method

Description of methods

The methods are given for a sequence `of t`. `function AsEnumerable(): sequence of T;` Returns input data reduced to the type IEnumerable.

Example

```
function Print<T>(Self: array of T): array of T;
extensionmethod;
begin
  Self.AsEnumerable.Print;
  Result := Self;
end;

begin
  Arr(1,2,3).Print end.
```

ToArray, ToList methods

Description of methods

The methods are given for a sequence `of t`. `function ToArray(): array of T;` Creates an array from the sequence.
`function ToList(): List<T>;`

Creates a List from the sequence.

Example

```
begin
  var a := Arr(1,2,3);
  a := a.Select(x->x*x).ToArray;
  var l := Lst(1,2,3);
  l := l.Select(x->x*x).ToList; end.
```

ToDictionary method

Description of methods

The methods are given for a sequence `of t`. `function ToDictionary<Key>(keySelector: T->Key): Dictionary<Key,T>;`

`Creates` a Dictionary from a sequence according to the given key selector function.

`function ToDictionary<Key>(keySelector: T->Key; comparer: IEqualityComparer<Key>): Dictionary<Key,T>;`

Creates a Dictionary from the sequence according to the specified key selector and key comparator function.

`function ToDictionary<Key,Element>(keySelector: T->Key; elementselector: T->Element): Dictionary<Key,Element>;`

Creates a Dictionary from a sequence according to the specified key selector and item selector functions.

`function ToDictionary<Key,Element>(keySelector: T->Key; elementselector: T->Element; comparer: IEqualityComparer<Key>): Dictionary<Key,Element>;`

Creates a Dictionary from the sequence according to the specified comparator and the functions of the key selector and item selector.

Example

```
begin
  var a := Agg(('crocodile',3),('hippo',1),('tiger',2));
  var d := a.ToDictionary(x->x[1],x->x[0]);
  d.Println; // (3,crocodile) (1,hippo) (2,tiger) end.
```

ToLookup method

Description of methods

The methods are given for the sequence `of t`. `function ToLookup<Key>(keySelector: T->Key): System.Linq.ILookup<Key,T>; Creates` a System.Linq.Lookup object from the sequence according to the specified key selector function.

```
function ToLookup<Key>(keySelector: T->Key; comparer:
IEqualityComparer<Key>): System.Linq.ILookup<Key,T>;
```

Creates a System.Linq.Lookup object from the sequence according to the specified key selector and key comparator function.

```
function ToLookup<Key,Element>(keySelector: T->Key;
elementselector: T->Element):
System.Linq.ILookup<Key,Element>;
```

Creates a System.Linq.Lookup object from the sequence according to the specified key selector and item selector functions.

```
function ToLookup<Key,Element>(keySelector: T->Key;
elementselector: T->Element; comparer:
IEqualityComparer<Key>): System.Linq.ILookup<Key,Element>;
```

Creates a System.Linq.Lookup object from the sequence according to the specified comparator and key selector and item selector functions.

Example

Without an example

The OfType method

Description of methods

The methods are given for the sequence `of t`. `Function OfType<Res>(): sequence of Res;` Performs filtering of elements of System.Collections.IEnumerable object by the given type. Returns a subsequence of the given sequence. in which all elements belong to the given type.

Example

```
begin
  var a := new object[](1,2.5,'d','ff',3.4);
  a.OfType&<real>().Println;
end.
```

Cast Method

Description of methods

The methods are given for the sequence `of t`. `function Cast<Res>(): sequence of Res;` `Converts` elements of a System.Collections.IEnumerable object to the specified type.

Example

```
begin
  var a: sequence of integer;
  var b: sequence of real;
  a := Seq(1,3,5);
  b := a.Cast&<real>();
end.
```

JoinIntoString method

Description of methods

The methods are given for a sequence `of t`. The `function JoinIntoString(delim: string := ' '): string;`

Converts the sequence elements to a string representation, then combines them to a string using delim as a delimiter.

Example

```
begin
  var a := Arr('aaa','bbb','ccc');
  var s: string := a.JoinIntoString('');
  Println(s); // aaabbbccc
end.
```

memory management

All reference types in .NET are managed by a so-called ***garbage collector***. This means that the memory allocated by a constructor call is never explicitly returned by a destructor call. Once an object is no longer needed, it should be assigned `nil`.

If there is a shortage of dynamic memory, the program execution is paused and a special procedure called garbage collection is started. It detects all so-called reachable objects. If no one else points to this object, it is considered unreachable and will be collected by the garbage collector. The time when the garbage collector is called is considered indefinite.

For example, when executing the code section

```
type Person = class - -- end;
var p: Person := new Person('Ivanov',2 0);-
-- p := nil;
```

memory allocated to `p`, after assigning it `nil,` will become unreachable and will be collected at an unpredictable moment.

Note that the dynamic memory allocated by the New procedure and controlled by pointers is not under the control of Garbage Collector, so it needs to be freed explicitly by calling Dispose. That's why working with standard pointers is considered deprecated in **PascalABC.NET** and isn't recommended for use.

Overview of the PABCSystem module

The PABCSystem module is called ***the system module*** and is automatically the first to be attached to any program or module. It contains a number of procedures, functions, constants, types, extension methods, and overloaded operations.

Standard types and constants

Standard subprograms

- General subprogramsпрограммы
- Mathematical subroutinesпрограммы
- subprogramsпрограммы вводInput
- subroutinesпрограммы выводOutput
- Common subroutinesпрограммы дfor working with files
- Subroutinesпрограммы дfor working with text files
- Subroutinesпрограммы дfor working with дbinary files
- Subroutinesпрограммы дfor working with file names
- System subroutinesпрограммы
- Subroutinesпрограммы дfor working with symbols
- Subroutinesпрограммы дfor working with strings
- Subroutinesпрограммы дfor working with дastandard sets
- Subroutinesпрограммы дfor working with дdynamic arrays

Generating objects of structured types

- Subroutinesпрограммы дfor sequence generationдовательностей
- Subroutinesпрограммы дfor generating infinite sequences
- Subroutinesпрограммы дfor generating дdynamic arrays
- Subroutinesпрограммы дfor generating matrices
- ShortfunctionsLst,HSet,SSet,Dict,KV

Extension methods defined in the PABCSystem module

- HYPERLINK "https://calibre-pdf-anchor.n/%23Extension%20methods%20for%20sequence%20of%20T.html"Methodsды extending the typesequence of T
- HYPERLINK "https://calibre-pdf-anchor.n/%23Extension%20methods%20for%20array%20of%20T.html"Methodsды typesarray of T
- methodsды of typearraarray[,]of T
- methodsды type extensionList<T>
- Methods forды extending the typeintegerger
- methodsды typeBiBigIntegerInteger
- HYPERLINK "https://calibre-pdf-anchor.n/%23Extension%20methods%20for%20real.html"Methodsды extending the typereal
- Methods forды typechar
- HYPERLINK "https://calibre-pdf-anchor.n/%23Extension%20methods%20for%20string.html"Methodsды typestrinstring
- methodsды extensionFunc
- methodsды extensionComplex
- methodsды extensionIDictionaryy

File type methods

Standard constants

```
AllDelimiters = ' <>=l'|~$ §!"#%''()*,+-./:;@[\]_{}"-
-"'#9#io#i3;
```
Constant - word delimiter characters

```
E = 2.718281828459045;
```
Constant E

```
MaxByte = byte.MaxValue;
```
Maximum value of byte type

```
MaxDouble = real.MaxValue;
```
Maximum value of type double

```
MaxInt = integer.MaxValue;
```
Maximum value of type integer

```
MaxInt64 = int64.MaxValue;
```
Maximum value of int64 type

```
MaxLongWord = longword.MaxValue;
```
Maximum value of the longword type

```
MaxReal = real.MaxValue;
```
Maximum value of type real

```
MaxShortInt = shortint.MaxValue;
```
Maximum value of shortint type

```
MaxSingle = single.MaxValue;
```
Maximum value of type single

```
MaxSmallInt = smallint.MaxValue;
```
Maximum value of the smallint type

```
MaxUInt64 = uint64.MaxValue;
```
Maximum value of type uint64

```
MaxWord = word.MaxValue;
```
Maximum value of word type

```
MinDouble = real.Epsilon;
```
Minimum positive value of type double

```
MinReal = real.Epsilon;
```
Minimum positive value of type real

```
MinSingle = single.Epsilon;
```
Minimum positive value of type single

```
NewLine = System.Environment.NewLine;
```
Newline constant

```
Pi = 3.141592653589793;
```

Constant Pi

Standard types

`Action<T> = system.Action<T>; Represents` an action with one parameter

`ActionO = System.Action;`

Represents an action without parameters

`Action2<T1, T2> = System.Action<T1, T2>;`

Represents an action with two parameters

`Action3<T1, T2, T3> = System.Action<T1, T2, T3>;`

Represents an action with three parameters

`BigInteger = System.Numerics.BigInteger;`

Represents an arbitrarily large integer `cardinal = System.UInt32;`

cardinal = longword

`Comparer<T> = System.Collections.Generic.Comparer<T>;`

Represents a base class for implementing the IComparer interface

`Complex = System.Numerics.Complex;`

Represents a complex number

`Console = System.Console;`

A class that manages the console window and the console I/O

`DateTime = System.DateTime;`

Represents the date and time

`decimal = System.Decimal;`

Represents a 128-bit real number

`Dictionary<Key, Value> = System.Collections.Generic.Dictionary<Key, Value>;`

Represents an associative array (a set of key-value pairs) implemented based on the hash table `double = System.Double;`

double = real

`Encoding = System.Text.Encoding;`

Character encoding type

`Exception = System.Exception;`

Basic exception type

`Func<T, Res> = System.Func<T, Res>;`

Represents a function with one parameter

`Func0<Res> = System.Func<Res>;`

Represents a function without parameters

```
Func2<T1, T2, Res> = System.Func<T1, T2, Res>;
```

Represents a function with two parameters

```
Func3<T1, T2, T3, Res> = System.Func<T1, T2, T3, Res>;
```

Represents a function with three parameters

```
GC = System.GC;
```

The class that controls garbage collection

```
HashSet<T> = System.Collections.Generic.HashSet<T>;
```

Represents a set of values implemented on the basis of a hash table

```
ICollection<T> = System.Collections.Generic.ICollection<T>;
```

Presents the interface for the collection

```
IComparable<T> = IComparable<T>;
```

Represents a base class for implementing the IComparer interface

```
IComparer<T> = System.Collections.Generic.IComparer<T>;
```

Represents an interface for comparing two elements

```
IDictionary<Key, Value> =
System.Collections.Generic.IDictionary<Key, Value>;
```

Represents the interface for the set of key-value pairs

```
IEnumerable<T> = System.Collections.Generic.IEnumerable<T>;
```

Represents an interface that provides an enumerator to enumerate items in the collection

```
IEnumerator<T> = System.Collections.Generic.IEnumerator<T>;
```

Represents an interface for enumerating elements of a collection `IEqualityComparer<T> = System.Collections.Generic.IEqualityComparer<T>;`

Provides an interface to support equality comparisons

```
IList<T> = System.Collections.Generic.IList<T>;
```

Represents the interface for a collection with index access

```
IntFunc = Func<integer, integer>;
```

Represents a function with one parameter of integer type that returns an integer

```
ISet<T> = System.Collections.Generic.ISet<T>;
```

Represents the interface for `KeyValuePair<Key, Value> =`

```
System.Collections.Generic.KeyValuePair<Key, Value>;
```

Represents a Key-Value pair for an associative array

```
LinkedList<T> = System.Collections.Generic.LinkedList<T>;
```

Represents a bilaterally linked list

```
LinkedListNode<T> =
System.Collections.Generic.LinkedListNode<T>;
```

Represents a node of a doubly linked list

```
List<T> = System.Collections.Generic.List<T>;
```

Represents a list based on a dynamic array

```
longint = System.Int32;
```

longint = integer

```
Match = System.Text.RegularExpressions.Match;
```

Represents results from a single match of the regular expression `MatchCollection = System.Text.RegularExpressions.MatchCollection;`

Represents the set of successful matches of the regular expression `MatchEvaluator = System.Text.RegularExpressions.MatchEvaluator;`

Represents the method called when a match is found in Regex.Replace

```
NonSerialized = System.NonSerializedAttribute;
```

Indicates that the field of the serializable class should not be serialized

```
Object = System.Object;
```

Basic type of objects

```
Predicate<T> = System.Predicate<T>;
```

Represents a function with one parameter that returns a boolean

```
Predicate2<T1, T2> = function(x1: T1; x2: T2): boolean;
```

Represents a function with two parameters that returns a boolean

```
Predicate3<T1, T2, T3> = function(x1: T1; x2: T2; x3: T3):
boolean;
```

Represents a function with three parameters that returns a boolean

```
Queue<T> = System.Collections.Generic.Queue<T>;
```

Represents the queue - a set of elements implemented on the principle of "first in first out"

```
RealFunc = Func<real, real>;
```

Represents a function with one parameter of the real type that returns a real

```
Regex = System.Text.RegularExpressions.Regex;
```

Represents a regular expression

```
RegexGroup = System.Text.RegularExpressions.Group;
```

Presents results from the same group when Regex.Match is run

```
RegexGroupCollection =
System.Text.RegularExpressions.GroupCollection;
```

Presents results from a set of groups when Regex.Match is run

```
RegexOptions = System.Text.RegularExpressions.RegexOptions;
```

Represents parameters of a regular expression

```
Serializable = System.SerializableAttribute;
```

Indicates the ability to serialize the class

```
ShortString = string[255];
```

Represents the type of a short string of fixed length 255 characters

```
SortedDictionary<Key, Value> =
System.Collections.Generic.SortedDictionary<Key, Value>;
```

Represents an associative array based on a binary search tree

```
SortedList<Key, Value> =
System.Collections.Generic.SortedList<Key, Value>;
```

Represents an associative array (a set of key-value pairs) based on a dynamic array of pairs

```
SortedSet<T> = System.Collections.Generic.SortedSet<T>;
```

Represents a set of values implemented on the basis of a binary search tree

```
Stack<T> = System.Collections.Generic.Stack<T>;
```

Represents the stack - a set of elements, implemented on the principle of "last in first out"

```
Stopwatch = System.Diagnostics.Stopwatch;
```

Provides methods for accurate measurement
time spent

```
StringBuilder = System.Text.StringBuilder;
```

Represents a modifiable character string

```
StringFunc = Func<string, string>;
```

Represents a function with one string type parameter that returns a string

```
Tuple = System.Tuple;
```

Represents the motorcade

General subprograms

```
function CommandLineArgs: array of string;
```
Returns the command string arguments with which the program was started

```
procedure Dec(var i: integer);
```
Reduces the value of variable i by 1

```
procedure Dec(var i: integer; n: integer);
```
Decreases the value of the variable i by n

```
procedure Dec(var e: enumerated type);
```
Decreases the value of the enumerated type by 1

```
procedure Dec(var e: enumerated type; n: integer);
```
Decreases the value of the enumerated type by n

```
function Eof: boolean;
```
Returns True if the end of the input stream is reached

```
function Eoln: boolean;
```
Returns True if the end of the string is reached

```
procedure Inc(var i: integer);
```
Increases the value of variable i by 1

```
procedure Inc(var i: integer; n: integer);
```
Increases the value of the variable i by n

```
procedure Inc(var e: enumerated type);
```
Increases the value of the enumerated type by 1

```
procedure Inc(var e: enumerated type; n: integer);
```
Increases the value of the enumerated type by n

```
function Ord(a: integer): integer;
```
Returns the sequence number of the value a

```
function Ord(a: enumerated type): integer;
```
Returns the sequence number of the value a

```
function Pred(x: integer): integer;
```
Returns the preceding x value

```
function Pred(x: enumerated type): enumerated type;
```
Returns the preceding x value

```
function SeekEof: boolean;
```
Skips whitespace, then returns True if the end of the input stream is reached

```
function SeekEoln: boolean;
```
Skips whitespace characters, then returns True if the end of the string is reached

```
function Succ(x: integer): integer;
```
Returns the next value after x

```
function Succ(x: enumerated type): enumerated type;
```

Returns the next value after x

```
procedure Swap<T>(var a, b: T);
```

Swaps values of two variables

Input subprograms

procedure Read(a,b,...); Enters values a,b,... from the keyboard
procedure Read(f: file; a,b,...);

Enters the values a,b,... from the file f

function ReadBigInteger: BigInteger;

Returns a value of type BigInteger entered from the keyboard

function ReadBoolean: boolean;

Returns a boolean value entered from the keyboard **function** ReadBoolean(prompt: string): boolean;

Outputs an input prompt and returns a boolean value entered from the keyboard

function ReadBoolean(f: Text): boolean;

Returns the value of boolean type entered from the text file f

function ReadChar: char;

Returns a char value entered from the keyboard **function** ReadChar(prompt: string): char;

Outputs an input prompt and returns a char value entered from the keyboard

function ReadChar(f: Text): char;

Returns the char value entered from the text file f

function ReadChar2: (char, char);

Returns a tuple of two char values entered from the keyboard

function ReadChar2(prompt: string): (char, char);

Returns a tuple of two char values entered from the keyboard

function ReadChar3: (char, char, char);

Returns a tuple of three char values entered from the keyboard

function ReadChar3(prompt: string): (char, char, char);

Returns a tuple of three char values entered from the keyboard

function ReadChar4: (char, char, char, char);

Returns a tuple of four char values entered from the keyboard

function ReadChar4(prompt: string): (char, char, char, char);

Returns a tuple of four char values entered from the keyboard

function ReadInt64: int64;

Returns a value of int64 type entered from the keyboard

function ReadInt64(prompt: string): int64;

Outputs an input prompt and returns an int64 value entered from the keyboard

function ReadInt64(f: Text): int64;

Returns a value of int64 type entered from the text file f

```
function Readinteger: integer;
```

Returns an integer value entered from the keyboard **function**

```
Readinteger(prompt: string): integer;
```

Outputs an input prompt and returns an integer value entered from the keyboard

```
function Readinteger(f: Text): integer;
```

Returns an integer value entered from the text file f

```
function Readinteger2: (integer, integer);
```

Returns a tuple of two integer values entered from the keyboard

```
function Readinteger2(prompt: string): (integer, integer);
```

Returns a tuple of two integer values entered from the keyboard **function** `Readinteger3: (integer, integer, integer);`

Returns a tuple of three integer values entered from the keyboard **function** `Readinteger3(prompt: string): (integer, integer, integer);`

Returns a tuple of three integer values entered from the keyboard **function** `ReadInteger4: (integer, integer, integer, integer);`

Returns a tuple of four integer values entered from the keyboard **function** `ReadInteger4(prompt: string): (integer, integer, integer, integer);`

Returns a tuple of four integer values entered from the keyboard **function** `ReadLexem: string;`

Returns the following token **function** `ReadLexem(f: Text): string;`

Returns the next token from the text file f **procedure**

```
Readln(a,b,...);
```

Enters the values a,b,... from the keyboard and moves to the next line

```
procedure Readln(f: Text; a,b,...);
```

Enters values a,b,... from the text file f and moves to the next line of the **function** `ReadlnBigInteger: BigInteger;`

Returns a value of type BigInteger entered from the keyboard

and moves to the next line of input `function ReadlnBoolean: boolean;`

Returns a boolean value entered from the keyboard and moves to the next line of input `function ReadlnBoolean(prompt: string): boolean;`

Outputs an input prompt and returns a boolean value entered from the keyboard and moves to the next line of input
`function ReadlnBoolean(f: Text): boolean;`

Returns the boolean value entered from the text file f and skips to the next line
`function ReadlnChar: char;`

Returns a char value entered from the keyboard and moves to the next line of input `function ReadlnChar(prompt: string): char;`

Outputs an input prompt and returns a char value entered from the keyboard and moves to the next line of input `function ReadlnChar(f: Text): char;`

Returns the char value entered from the text file f and skips to the next line
`function ReadlnChar2: (char, char);`

Returns a tuple of two char values entered from the keyboard and moves to the next input line `function ReadlnChar2(prompt: string): (char, char);`

Returns a tuple of two char values entered from the keyboard and moves on to the next line of input `function ReadlnChar3: (char, char, char);`

Returns a tuple of three char values entered from the keyboard and moves on to the next line of input `function ReadlnChar3(prompt: string): (char, char, char);`

Returns a tuple of three char values entered from the keyboard and moves on to the next line of input `function ReadlnChar4: (char, char, char, char);`

Returns a tuple of four char values entered from the keyboard and moves on to the next line of input `function ReadlnChar4(prompt: string): (char, char, char, char);`

Returns a tuple of four char values entered from the keyboard

and skips to the next line of input
```
function ReadlnInt64: int64;
```
Returns int64 type value entered from the keyboard and moves to the next line of input `function ReadlnInt64(prompt: string): int64;`

Outputs an input prompt and returns an int64 value entered from the keyboard and moves to the next line of input
```
function ReadlnInt64(f: Text): int64;
```
Returns an int64-type value entered from the text file f, and skips to the next line of `the function ReadlnInteger: integer;`

Returns an integer value entered from the keyboard and moves to the next line of input `function ReadlnInteger(prompt: string): integer;`

Outputs an input prompt and returns an integer value entered from the keyboard and moves to the next input line `function ReadlnInteger(f: Text): integer;`

Returns an integer value entered from the text file f, and skips to the next line of `the function ReadlnInteger2: (integer, integer);`

Returns a tuple of two integer values entered from the keyboard and moves on to the next line of input `function ReadlnInteger2(prompt: string): (integer, integer);`

Returns a tuple of two integer values entered from the keyboard and moves to the next line of input `function ReadlnInteger3: (integer, integer, integer);`

Returns a tuple of three integer values entered from the keyboard, and moves on to the next line of input `function ReadlnInteger3(prompt: string): (integer, integer, integer);`

Returns a tuple of three integer values entered from the keyboard and moves on to the next line of input `function ReadlnInteger4: (integer, integer, integer, integer);`

Returns a tuple of four integer values entered from the keyboard, and moves on to the next line of input
```
function ReadlnInteger4(prompt: string): (integer, integer,
integer, integer);
```
Returns a tuple of four integer values entered from the

keyboard and moves on to the next line of input **function** `ReadlnReal: real;`

Returns a value of type real, entered from the keyboard, and moves to the next line of input **function** `ReadlnReal(prompt: string): real;`

Outputs an input prompt and returns a value of type real, entered from the keyboard, and moves to the next input line **function** `ReadlnReal(f: Text): real;`

Returns the value of type real entered from the text file f, and moves to the next line of **the function** `ReadlnReal2: (real, real);`

Returns a tuple of two values of type real, entered from the keyboard, and moves to the next line of input **function** `ReadlnReal2(prompt: string): (real, real);`

Returns a tuple of two values of type real, entered from the keyboard, and moves to the next line of input **function** `ReadlnReal3: (real, real, real);`

Returns a tuple of three values of type real, entered from the keyboard, and moves to the next line of input **function** `ReadlnReal3(prompt: string): (real, real, real);`

Returns a tuple of three values of type real, entered from the keyboard, and moves to the next line of input **function** `ReadlnReal4: (real, real, real, real);`

Returns a tuple of four values of type real, entered from the keyboard, and moves to the next line of input **function** `ReadlnReal4(prompt: string): (real, real, real, real);`

Returns a tuple of four values of type real, entered from the keyboard, and goes to the next line of input **function** `ReadlnString: string;`

Returns a value of type string entered from the keyboard and moves to the next line of input **function** `ReadlnString(prompt: string): string;`

Outputs an input prompt and returns a value of type string entered from the keyboard, and moves to the next input line **function** `ReadlnString(f: Text): string;`

Returns the value of string type entered from the text file f, and

moves to the next line of **the function** `ReadlnString2: (string, string);`

Returns a tuple of two string values entered from the keyboard and moves to the next input line **function** `ReadlnString2(prompt: string): (string, string);`

Returns a tuple of two string values entered from the keyboard and moves on to the next line of input **function** `ReadlnString3: (string, string, string);`

Returns a tuple of three string values entered from the keyboard and moves on to the next line of input **function** `ReadlnString3(prompt: string): (string, string, string);`

Returns a tuple of three string values entered from the keyboard and moves on to the next line of input **function** `ReadlnString4: (string, string, string, string);`

Returns a tuple of four string values entered from the keyboard and moves on to the next line of input **function** `ReadlnString4(prompt: string): (string, string, string, string);`

Returns a tuple of four string values entered from the keyboard, and skips to the next line of input

function `ReadReal: real;`

Returns a value of type real entered from the keyboard

function `ReadReal(prompt: string): real;`

Outputs the input prompt and returns a value of type real entered from the keyboard **function** `ReadReal(f: Text): real;`

Returns a value of type real, entered from text file f

function `ReadReal2: (real, real);`

Returns a tuple of two values of type real entered from the keyboard **function** `ReadReal2(prompt: string): (real, real);`

Returns a tuple of two values of type real, entered from the keyboard **function** `ReadReal3: (real, real, real);`

Returns a tuple of three values of type real entered from the keyboard **function** `ReadReal3(prompt: string): (real, real, real);`

Returns a tuple of three values of type real, entered from the keyboard **function** `ReadReal4: (real, real, real, real);`

Returns a tuple of four values of type real entered from the

keyboard `function ReadReal4(prompt: string): (real, real, real, real);`

Returns a tuple of four values of type real, entered from the keyboard `function ReadString: string;`

Returns a value of type string entered from the keyboard `function ReadString(prompt: string): string;`

Outputs an input prompt and returns a value of type string entered from the keyboard `function ReadString(f: Text): string;`

Returns value of type string entered from text file f `function ReadString2: (string, string);`

Returns a tuple of two string values entered from the keyboard `function ReadString2(prompt: string): (string, string);`

Returns a tuple of two string values entered from the keyboard `function ReadString3: (string, string, string);`

Returns a tuple of three string values entered from the keyboard `function ReadString3(prompt: string): (string, string, string);`

Returns a tuple of three string values entered from the keyboard
`function ReadString4: (string, string, string, string);`

Returns a tuple of four string values entered from the keyboard `function ReadString4(prompt: string): (string, string, string, string);`

Returns a tuple of four string values entered from the keyboard `function TryRead(var x: number): boolean;`

Enters the numeric value of x from the keyboard. Returns False if an error occurred while entering
`function TryRead(var x: number; message: string): boolean;`

Outputs an input prompt and enters the numeric value of x from the keyboard. Returns False if an error occurs
`function TryRead(var x: boolean; message: string := ''): boolean;`

Enters the boolean value x from the keyboard. Returns False if an error occurred while entering

Output subroutines

```
function Deserialize(filename: string): object;
```

Deserializes an object from a file

```
procedure Print(a,b,...);
```

Outputs values a,b,... on the screen, after each value prints a space

```
procedure Print(f: Text; a,b,...);
```

Outputs values a,b,... to the text file f, after each value prints a space

```
procedure Println(a,b,...);
```

Outputs values a,b,... on the screen, after each value prints a space and jumps to a new line

```
procedure Println(f: Text; a,b,...);
```

Outputs values a,b,... to the text file f, after each value prints a space and jumps to a new line

```
procedure Serialize(filename: string; obj: object);
```

Serializes the object to a file (the object must have the [Serializable] attribute)

```
procedure Write(a,b,...);
```

Outputs the values a,b,... on the screen

```
procedure Write(f: file; a,b,...);
```

Outputs values a,b,... into file f `procedure WriteFormat(formatstr: string; params args: array of objects);`

Outputs args values according to formatstr

```
procedure WriteFormat(f: Text; formatstr: string; params args:
array of objects);
```

Outputs args values to the text file f according to the formatstr

```
procedure Writeln(a,b,...);
```

Outputs the values a,b,... onto the screen and jumps to a new line

```
procedure Writeln(f: Text; a,b,...);
```

Outputs values a,b,... to the text file f and moves to a new line

```
procedure WritelnFormat(formatstr: string; params args: array
of objects);
```

Outputs args values according to the formatstr format string and jumps to a new string `procedure WritelnFormat(f: Text; formatstr: string; params args: array of objects);`

Outputs args values to the text file f according to the formatstr format string and jumps to a new line

Mathematical subroutines

```
function Abs(x: number): number;
```
Returns the modulus of number x
```
function ArcCos(x: real): real;
```
Returns the angle in radians, cosine of which is equal to x, -1<=x<=1
```
function ArcSin(x: real): real;
```
Returns the angle in radians, the sine of which is equal to x, -1<=x<=1
```
function ArcTan(x: real): real;
```
Returns the angle in radians, the tangent of which is equal to x
```
function Ceil(x: real): integer;
```
Returns the smallest integer >= x
```
function Cos(x: real): real;
```
Returns the cosine of the angle x measured in radians
```
function Cosh(x: real): real;
```
Returns the hyperbolic cosine of angle x measured in radians
```
function DegToRad(x: real): real;
```
Converts degrees to radians
```
function Exp(x: real): real;
```
Returns the exponent of the number x
```
function Floor(x: real): integer;
```
Returns the largest integer <= x
```
function Frac(x: real): real;
```
Returns the fractional part of the number x
```
function Int(x: real): real;
```
Returns the integer part of the number x
```
function Ln(x: real): real;
```
Returns the natural logarithm of the number x
```
function Log(x: real): real;
```
Returns the natural logarithm of the number x
```
function Log10(x: real): real;
```
Returns the decimal logarithm of the number x
```
function Log2(x: real): real;
```
Returns the logarithm of the number x on base 2
```
function LogN(base, x: real): real;
```
Returns the logarithm of the number x on base base
```
function Max(a: number, b: number): number;
```
Returns the maximum of the numbers a,b
```
function Max(a,b,...: T): T;
```

Returns the maximum of a,b,...

```
function Min(a: number, b: number): number;
```

Returns the minimum of the numbers a,b

```
function Min(a,b,...: T): T;
```

Returns the minimum of a,b,...

```
function Odd(i: integer): boolean;
```

Returns True if i is odd, and False otherwise

```
function Power(x, y: real): real;
```

Returns x to the power of y

```
function Power(x: real; n: integer): real;
```

Returns x to the integer power of n

```
function Power(x: BigInteger; y: integer): BigInteger;
```

Returns x to the power of y

```
function RadToDeg(x: real): real;
```

Converts radians to degrees

```
function Random(maxValue: integer): integer;
```

Returns a random integer in the range 0 to maxValue-1

```
function Random(maxValue: real): real;
```

Returns a random real in the range [0,maxValue)

```
function Random(a, b: integer): integer;
```

Returns a random integer in the range a to b

```
function Random(a, b: real): real;
```

Returns a random real in the range [a,b] `function Random: real;`

Returns a random real in the range [0..1] `function Random2(maxValue: integer): (integer, integer);`

Returns a tuple of two random integers between 0 and maxValue-1 `function Random2(maxValue: real): (real, real);`

Returns a tuple of two random realities in the range [0,maxValue) `function Random2(a, b: integer): (integer, integer);`

Returns a tuple of two random integers in the range a to b

```
function Random2(a, b: real): (real, real);
```

Returns a tuple of two random realities in the range [a,b]

```
function Random2: (real, real);
```

Returns a tuple of two random realities in the range [0..1]

```
function Random3(maxValue: integer): (integer, integer, integer);
```

Returns a tuple of three random integers between 0 and

maxValue-1 `function Random3(maxValue: real): (real, real, real);`

Returns a tuple of three random realities in the range [0,maxValue)

`function Random3(a, b: integer): (integer, integer, integer);`

Returns a tuple of three random integers in the range a to b

`function Random3(a, b: real): (real, real, real);`

Returns a tuple of three random realities in the range [a,b]

`function Random3: (real, real, real);`

Returns a tuple of three random realities in the range [0..1)

`procedure Randomize(seed: integer);`

Initializes the pseudorandom number sensor using the seed value. The same seed generates the same pseudorandom sequences

`procedure Randomize;`

Initializes a pseudorandom number sensor

`function Round(x: real): integer;`

Returns x rounded to the nearest integer. If the real is in the middle between two integers, it is rounded to the nearest even number (bank rounding): Round(2.5)=2, Round(3.5)=4

`function Round(x: real; digits: integer): real;`

Returns x rounded to the nearest real with digits after the decimal point

`function RoundBigInteger(x: real): BigInteger;`

Returns x rounded to the nearest long integer

`function Sign(x: number): integer;`

Returns -1,0 or +1 depending on the sign of number x

`function Sin(x: real): real;`

Returns the sine of the angle x measured in radians

`function Sinh(x: real): real;`

Returns the hyperbolic sine of the angle x measured in radians

`function Sqr(x: number): number;`

Returns the square of the number x

`function Sqrt(x: real): real;`

Returns the square root of the number x

`function Tan(x: real): real;`

Returns the tangent of the angle x measured in radians

`function Tanh(x: real): real;`

Returns the hyperbolic tangent of angle x measured in radians

```
function Trunc(x: real): integer;
```

Returns the integer part of the real number x

```
function TruncBigInteger(x: real): BigInteger;
```

Returns the integer part of the real number x as long whole

System subroutines

`procedure Assert(cond: boolean);` Outputs in a special window the subroutine call stack if the condition is not met `procedure Assert(cond: boolean; message: string);`

Outputs in a special window diagnostic message and subroutine call stack if the condition is not fulfilled `function ChangeFileNameExtension(name, newext: string): string;`

Changes the extension of the file named name to newext

`procedure ChDir(s: string);`

Changes the current directory

`function CreateDir(s: string): boolean;`

Creates directory. Returns True if the directory was successfully created

`function DeleteFile(s: string): boolean;`

Deletes the file. If the file cannot be deleted, it returns False.

`function DiskFree(diskname: string): int64;`

Returns the free space in bytes on the disk named diskname

`function DiskFree(disk: integer): int64;`

Returns the free space in bytes on disk. disk=0 is the current disk, disk=1 is disk A: , disk=2 - disk B:, etc.

`function DiskSize(diskname: string): int64;`

Returns the size in bytes on the disk named diskname

`function DiskSize(disk: integer): int64;`

Returns the size in bytes of the disk. disk=0 - current disk, disk=1 - disk A: , disk=2 - disk B:, etc.

`procedure Dispose<T>(var p: LT);`

Frees up the dynamic memory pointed to by p `function EnumerateAllDirectories(path: string): sequence of string;`

Returns a sequence of directory names for a given path, including subdirectories `function EnumerateAllFiles(path: string; searchPattern: string := '*.*'): sequence of string;`

Returns a sequence of file names in the given path, matching the search pattern, including subdirectories `function EnumerateDirectories(path: string): sequence of string;`

Returns the sequence of directory names for the given path

`function EnumerateFiles(path: string; searchPattern: string :=`

```
'*.*'): sequence of string;
```

Returns a sequence of file names on the specified path, matching the search pattern

```
procedure Exec(filename: string);
```

Starts a program or document named filename **procedure**

```
Exec(filename: string; args: string);
```

Launches a program or document with filename and command line parameters args

```
procedure Executefilename: string);
```

Starts a program or document named filename **procedure**

```
Executefilename: string; args: string);
```

Launches a program or document with filename and command line parameters args

```
function FileExists(name: string): boolean;
```

Returns True if the file named name exists

```
function GetCurrentDir: string;
```

Returns the current directory

```
function GetDir: string;
```

Returns current directory **function** `GetEXEFileName: string;`

Returns the name of the running .exe file

```
procedure Halt(exitCode: integer);
```

Terminates the program, returning the error code exitCode

```
procedure Halt;
```

Completes the program

```
function Milliseconds: integer;
```

Returns the number of milliseconds since the program started

```
function MillisecondsDelta: integer;
```

Returns the number of milliseconds since the last Milliseconds or MillisecondsDelta call or program start

```
procedure MkDir(s: string);
```

Creates directory

```
procedure New<T>(var p: LT);
```

Allocates dynamic memory of sizeof(T) and returns a pointer to it in the p variable. The type T must be of size

```
function ParamCount: integer;
```

Returns the number of command line parameters

```
function ParamStr(i: integer): string;
```

Returns the i-th command line parameter

```
function PascalABCVersion: string;
```

Returns the version of PascalABC.NET

```
function PointerToString(p: pointer): string;
```

Converts the pointer to a string representation

```
function RemoveDir(s: string): boolean;
```

Deletes a directory. Returns True if the directory was successfully deleted

```
function RenameFile(name, newname: string): boolean;
```

Renames file name, giving it newname. Returns True if the file was successfully renamed

```
procedure RmDir(s: string);
```

Deletes directory

```
function SetCurrentDir(s: string): boolean;
```

Sets the current directory. Returns True if the directory was successfully deleted

```
procedure Sleep(ms: integer);
```

Pauses for ms milliseconds

Common subroutines for working with files

```
procedure Assign(f: file; name: string);
```
Assigns file variable with a file on disk

```
procedure AssignFile(f: file; name: string);
```
Associates a file variable with a file on disk

```
procedure Close(f: file);
```
Closes the file

```
procedure CloseFile(f: file);
```
Closes the file

```
function Eof(f: file): boolean;
```
Returns True if the end of the file is reached

```
procedure Erase(f: file);
```
Deletes the file associated with the file variable

```
procedure Rename(f: file; newname: string);
```
Renames the file associated with the file variable, giving it the name newname.

Subroutines for working with text files

`procedure Append(f: Text);` Opens the TEXT file for addition in Windows encoding

`procedure Append(f: Text; en: Encoding);`

Opens a text file for addition in the specified encoding

`procedure Append(f: Text; name: string);`

Associates the file variable f with the file name and opens the text file in Windows encoding

`procedure Append(f: Text; name: string; en: Encoding);`

Associates the file variable f with the file name and opens the text file for addition in the specified encoding

`function Eoln(f: Text): boolean;`

Returns True if the file reaches the end of the line

`procedure Flush(f: Text);`

Writes the contents of the file buffer to disk

`function OpenAppend(fname: string): Text;`

Returns a text file named fname opened on a Windows encoding addition

`function OpenAppend(fname: string; en: Encoding): Text;`

Returns a text file named fname, opened on the addition in the specified encoding

`function OpenRead(fname: string): Text;`

Returns a text file named fname, open for reading in Windows encoding

`function OpenRead(fname: string; en: Encoding): Text;`

Returns a text file named fname, opened for reading in the specified encoding

`function OpenWrite(fname: string): Text;`

Returns a text file named fname, open for writing in Windows encoding

`function OpenWrite(fname: string; en: Encoding): Text;`

Returns a text file named fname, opened for writing in the specified encoding

`function ReadAllLines(path: string): array of string;`

Opens file, reads lines from it in Windows encoding as an array of lines, then closes file `function ReadAllLines(path: string; en: Encoding): array of string;`

Opens file, reads strings from it in specified encoding as array of strings, then closes file `function ReadAllText(path: string):`

```
string;
```

Opens a file, reads its contents in Windows encoding as a string, then closes the file with **function** `ReadAllText(path: string; en: Encoding): string;`

Opens the file, reads its contents in the specified encoding as a string, and then closes the file

function `ReadLines(path: string): sequence of string;`

Opens file, reads lines from it in Windows encoding and closes file. At each moment only the current line is stored in memory **function** `ReadLines(path: string; en: Encoding): sequence of string;`

Opens file, reads lines from it in specified encoding and closes file. At each moment only the current line **procedure** `Reset(f: Text)` is stored in memory;

Opens a text file for reading in Windows encoding **procedure** `Reset(f: Text; en: Encoding);`

Opens a text file for reading in the specified encoding **procedure** `Reset(f: Text; name: string);`

Associates the file variable f with the file name and opens the text file for reading in Windows encoding

procedure `Reset(f: Text; name: string; en: Encoding);`

Associates the file variable f with the file name and opens the text file for reading in the specified encoding

procedure `Rewrite(f: Text);`

Opens a text file for writing in Windows encoding- If the file existed - it is zeroed, if not - it is created empty

procedure `Rewrite(f: Text; en: Encoding);`

Opens a text file for writing in the specified encoding. if the file existed - it is reset, if not - an empty one is created

procedure `Rewrite(f: Text; name: string);`

Associates a file variable with file name and opens text file f for writing in Windows encoding-if the file existed, it is zeroed, if not, it is created empty

procedure `Rewrite(f: Text; name: string; en: Encoding);`

Associates the file variable f with the file name and opens the text file f for writing in the specified encoding.If the file existed - it is zeroed, if not - it is created empty

```
function SeekEof(f: Text): boolean;
```

Skips whitespace characters, then returns True if the end of the file is reached

```
function SeekEoln(f: Text): boolean;
```

Skips whitespace characters, then returns True if the file reaches the end of the line

```
procedure WriteAllLines(path: string; ss: array of string);
```

Creates a new file, writes to it the strings from the array in Windows encoding, and then closes the file

```
procedure WriteAllLines(path: string; ss: array of string; en:
Encoding);
```

Creates a new file, writes to it the strings from the array in the specified encoding, and then closes the file

```
procedure WriteAllText(path: string; s: string);
```

Creates a new file, writes a line in it in Windows encoding, and then closes the file

```
procedure WriteAllText(path: string; s: string; en: Encoding);
```

Creates a new file, writes a line to it in
the specified encoding, and then closes the file

```
procedure WriteLines(path: string; ss: sequence of string);
```

Creates a new file, writes to it the lines from the sequence in Windows encoding, and then closes the file

```
procedure WriteLines(path: string; ss: sequence of string; en:
Encoding);
```

Creates a new file, writes into it the lines from the sequence in the specified encoding, and then closes the file

Subroutines for working with typed and untyped files

`function CreateBinary(fname: string): file;` Creates or nulls an untyped file and returns a value to initialize the file variable

`function CreateBinary(fname: string; en: Encoding): file;`

Creates or nulls a zip file in the specified encoding and returns a value to initialize the file variable

`function CreateFile<T>(fname: string): file of T;`

Creates or nulls typed file and returns value to initialize file variable `function CreateFile<T>(fname: string; en: Encoding): file of T;`

Creates or nulls typed file in specified encoding and returns value to initialize file variable `function CreateFileInteger(fname: string): file of integer;`

Creates or nulls a typed integer file and returns a value to initialize the file variable

`function CreateFileReal(fname: string): file of real;`

Creates or nulls a typed real file and returns a value to initialize the file variable

`function OpenBinary(fname: string): file;`

Opens a bespoke file and returns a value to initialize the file variable

`function OpenBinary(fname: string; en: Encoding): file;`

Opens a zipless file in a given encoding and returns the value to initialize the file variable

`function OpenFile<T>(fname: string): file of T;`

Opens a typed file and returns the value
to initialize the file variable

`function OpenFile<T>(fname: string; en: Encoding): file of T;`

Opens a typed file in a given encoding and returns a value to initialize the file variable

`function OpenFileInteger(fname: string): file of integer;`

Opens a typed integer file and returns a value to initialize the file variable

`function OpenFileReal(fname: string): file of real;`

Opens a typed real file and returns a value to initialize the file

variable

```
procedure WriteElements<T>(fname: string; ss: sequence of T);
```

Opens a typed file, writes a sequence of ss elements to it and closes it

Subroutines for working with binary files

```
function FilePos(f: binary): int64;
```
Returns the current position of the file pointer in the binary file

```
function FileSize(f: binary file): int64;
```
Returns the number of elements in the binary file

```
procedure Reset(f: binary file);
```
Opens a binary file for reading and writing.A binary file is either a typed file of T or a typeless file of file

```
procedure Reset(f: binary file; name: string);
```
Associates the file variable f with the file name on disk and opens a binary file for reading and writing.A binary file is either a typed file of T or a typeless file file

```
procedure Reset(f: binary; en: Encoding);
```
Opens a binary file for reading and writing in a given encoding.A binary file is either a typed file of T or a typeless file of file

```
procedure Reset(f: binary; name: string; en: Encoding);
```
Associates the file variable f with the file name on disk and opens a binary file for reading and writing in a given encoding.A binary file is either a typed file of T or a typeless file of file

```
procedure Rewrite(f: binary file);
```
Opens a binary file for reading and writing, while zeroing its contents. If the file existed, it is zeroed.A binary file is either a typed file of T or a typeless file of file

```
procedure Rewrite(f: binary file; name: string);
```
Associates the file variable f with the name file on disk and opens the binary file for reading and writing, while zeroing its contents.A binary file is either a typed file of T or an untyped file of file

```
procedure Rewrite(f: binary; en: Encoding);
```
Opens a binary file for reading and writing in a given encoding, while zeroing its contents. If the file existed, it is zeroed.A binary file is either a typed file of T or a type-free file file `procedure Rewrite(f: binary file; name: string; en: Encoding);`

Associates the file variable f with the name file on disk and opens the binary file for reading and writing in the specified encoding, while zeroing its contents.A binary file is either a typed file of T or a typeless file file of file

```
procedure Seek(f: binary; n: int64);
```

Sets the current position of the file pointer in the binary file to the element with the given number

```
procedure Truncate(f: binary file);
```

Truncates a binary file by discarding all elements from the file pointer position.A binary file is either a typed file of T or a typeless file of file

Subroutines for working with file names

```
function ExpandFileName(fname: string): string;
```

Returns the full name of the file fname

```
function ExtractFileDir(fname: string): string;
```

Selects the drive name and path from the full file name fname

```
function ExtractFileDrive(fname: string): string;
```

Extracts the path from the full file name fname **function**

```
ExtractFileExt(fname: string): string;
```

Extracts extension from the full file name fname **function**

```
ExtractFileName(fname: string): string;
```

Extracts file name from the full file name fname **function**

```
ExtractFilePath(fname: string): string;
```

Selects the path from the full file name fname

Subroutines for working with symbols

```
function Chr(a: word): char;
```
Converts code to Unicode character

```
function ChrAnsi(a: byte): char;
```
Converts the code to a Windows encoded character

```
function ChrUnicode(a: word): char;
```
Converts the code into a Unicode character

```
procedure Dec(var c: char);
```
Reduces the c character code by 1

```
procedure Dec(var c: char; n: integer);
```
Reduces the c character code by n

```
procedure Inc(var c: char);
```
Increases the c character code by 1

```
procedure Inc(var c: char; n: integer);
```
Increases the c character code by n

```
function LowCase(ch: char): char;
```
Converts character to lower case

```
function LowerCase(ch: char): char;
```
Converts character to lower case

```
function Ord(a: char): word;
```
Converts the character into Unicode code

```
function OrdAnsi(a: char): byte;
```
Converts character to Windows encoding

```
function OrdUnicode(a: char): word;
```
Converts the character into Unicode code

```
function Pred(x: char): char;
```
Returns the preceding x character

```
function Succ(x: char): char;
```
Returns the next character after x

```
function UpCase(ch: char): char;
```
Converts character to uppercase

```
function UpperCase(ch: char): char;
```
Converts character to uppercase

Subroutines for working with strings

```
function CompareStr(s1, s2: string): integer;
```
Compares strings. Returns < 0 if s1<s2, > 0 if s1>s2 and = 0 if s1=s2

```
function Concat(s1,s2,...): string;
```

Returns the string that is the result of merging the strings s1,s2,...

```
function Concat(s1, s2: string): string;
```

Returns the string that is the result of merging s1 and s2

```
function Copy(s: string; index, count: integer): string;
```

Returns a substring of string s of length count from position index

```
procedure Delete(var s: string; index, count: integer);
```

Removes from the string s count of characters from the index position

```
function FloatToStr(a: real): string;
```

Converts a real number to a string representation `function Format(formatstring: string; params pars: array of objects): string;`

Returns a formatted string constructed from the format string and the list of formatted parameters `procedure Insert(source: string; var s: string; index: integer);`

Inserts a substring of source into the string s from the index position

```
function IntToStr(a: integer): string;
```

Converts an integer to a string representation

```
function IntToStr(a: int64): string;
```

Converts an integer to a string representation

```
function LastPos(subs, s: string): integer;
```

Returns the position of the last occurrence of the substring subs in the string s. If not found, returns 0

```
function LastPos(subs, s: string; from: integer): integer;
```

Returns the position of the last occurrence of the substring subs in the string s starting from the from position. If not found, it returns 0

```
function LeftStr(s: string; count: integer): string;
```

Returns the first count of characters of the string s

```
function Length(s: string): integer;
```

Returns the string length

```
function LowerCase(s: string): string;
```

Returns a lowercase string

```
function Pos(subs, s: string; from: integer := 1): integer;
```

Returns position of substring subs in string s. If not found, it returns 0

```
function PosEx(subs, s: string; from: integer := 1): integer;
```

Returns position of substring subs in string s starting from position from. If not found, returns 0 `function ReadIntegerFromString(s: string; var from: integer): integer;`

Reads an integer from a string starting from and sets from behind the read value `function ReadRealFromString(s: string; var from: integer): real;`

Reads the real from the string starting from and sets from after the read value `function ReadWordFromString(s: string; var from: integer): string;`

Reads from a string a sequence of characters up to the space character starting from the from position and sets from after the read value

```
function ReverseString(s: string): string;
```

Returns the inverted string

```
function ReverseString(s: string; index,length: integer): string;
```

Returns an inverted string in length range starting from index

```
function RightStr(s: string; count: integer): string;
```

Returns the last count of characters of the string s `procedure SetLength(var s: string; n: integer);`

Sets the length of string s to n `procedure Str(i: integer; var s: string);`

Converts an integer value i to a string representation and writes the result to s `procedure Str(r: real; var s: string);`

Converts a real value of r to a string representation and writes the result to s `procedure Str(r: single; var s: string);`

Converts a real value of r to a string representation and writes the result to s `function StringIsEmpty(s: string; var from: integer): boolean;`

Returns True if the end of the string is reached or the string contains only whitespace characters, and False otherwise `function StringOfChar(ch: char; count: integer): string;`

Returns a string consisting of count characters ch **function** `StrToFloat(s: string): real;`

Converts string representation of a real number to a numeric value **function** `StrToInt(s: string): integer;`

Converts string representation of an integer to a numeric value **function** `StrToInt64(s: string): int64;`

Converts a string representation of an integer to a numeric value **function** `StrToReal(s: string): real;`

Converts the string representation of a real number to a numeric value **function** `Trim(s: string): string;`

Returns a string with deleted leading and trailing spaces **function** `TrimLeft(s: string): string;`

Returns a string with the initial spaces removed **function** `TrimRight(s: string): string;`

Returns a string with deleted end spaces **function** `TryReadIntegerFromString(s: string; var from: integer; var res: integer): boolean;`

Reads an integer from a string starting from and sets from behind the read value.Returns True if the read was successful and False otherwise **function** `TryReadRealFromString(s: string; var from: integer; var res: real): boolean;`

Reads the real from the string starting from and sets from behind the read value.Returns True if the read is successful and False otherwise **function** `TryStrToFloat(s: string; var value: real): boolean;`

Converts the string representation s of a real number to a numeric value and writes it to value.If the conversion fails it returns False **function** `TryStrToFloat(s: string; var value: single): boolean;`

Converts the string representation s of a real number to a numeric value and writes it to value.If the conversion fails it returns False **function** `TryStrToInt(s: string; var value: integer): boolean;`

Converts the string representation s of an integer to a numeric value and writes it to value.If the conversion fails it returns False

```
function TryStrToInt64(s: string; var value: int64): boolean;
```

Converts the string representation s of an integer to a numeric value and writes it to value.If the conversion fails it returns False

```
function TryStrToReal(s: string; var value: real): boolean;
```

Converts the string representation s of a real number to a numeric value and writes it to value.If the conversion fails it returns False

```
function TryStrToSingle(s: string; var value: single): boolean;
```

Converts the string representation s of a real number to a numeric value and writes it to value.If the conversion fails it returns False

```
function UpperCase(s: string): string;
```

Returns an uppercase string

```
procedure Val(s: string; var value: number; var err: integer);
```

Converts string representation s of an integer or real number to a numeric value and writes it to the variable value-if the conversion is successful, then err=0, otherwise err>0

Subroutines for working with dynamic arrays

```
function ArrEqual<T>(a, b: array of T): boolean;
```

Returns whether the arrays match or not

```
function Copy(a: array of T): array of T;
```

Creates a copy of a dynamic array

```
function High(a: array of T): integer;
```

Returns the upper bound of the dynamic array

```
function Length(a: array of T): integer;
```

Returns the length of the dynamic array

```
function Length(a: array of T; dim: integer): integer;
```

Returns the length of a dynamic array by dim dimension

```
function Low(a: array of T): integer;
```

Returns 0

```
function MatrEqual<T>(a, b: array [,] of T): boolean;
```

Compares matrices for equality

```
function NextPermutation(a: array of integer): boolean;
```

Returns the next permutation in the array

```
procedure Reverse<T>(a: array of T);
```

Reverses the order of elements in a dynamic array

```
procedure Reverse<T>(a: array of T; index, count: integer);
```

Reverses the order of the elements in the range of the dynamic array of length count, starting from the index

```
procedure Reverse<T>(a: List<T>);
```

Reverses the order of the items in the list

```
procedure Reverse<T>(a: List<T>; index, count: integer);
```

Reverses the order of the elements in the range of the list of length count, starting from the index

```
procedure Reverse(var s: string);
```

Reverse `procedure Reverse(var s: string; index, count: integer);`

Reverses the order of the characters in the count part of the string length, starting from the index

```
procedure SetLength(var a: array of T; n: integer);
```

Sets the length of a one-dimensional dynamic array. Old content is retained

```
procedure SetLength(var a: array [,...,] of T; n1,n2,...:
```

```
integer);
```

Sets the size of an n-dimensional dynamic array. The old content is saved

```
procedure Shuffle<T>(a: array of T);
```

Shuffles the dynamic array randomly

```
procedure Shuffle<T>(l: List<T>);
```

Shuffles the list at random

```
procedure Sort<T>(a: array of T);
```

Sorts a dynamic array in ascending order **procedure**

```
Sort<T>(a: array of T; cmp: (T,T)->integer);
```

Sorts a dynamic array according to a sorting criterion specified by the cmp comparison function

```
procedure Sort<T>(a: array of T; less: (T,T)->boolean);
```

Sorts a dynamic array according to a sorting criterion specified by the comparison function less

```
procedure Sort<T>(l: List<T>);
```

Sorts the list in ascending order

```
procedure Sort<T>(l: List<T>; cmp: (T,T)->integer);
```

Sorts the list according to the sorting criterion specified by the cmp comparison function

```
procedure Sort<T>(l: List<T>; less: (T,T)->boolean);
```

Sorts the list according to the sorting criterion specified by the less comparison function

```
procedure SortDescending<T>(a: array of T);
```

Sorts a dynamic array in descending order

```
procedure SortDescending<T>(l: List<T>);
```

Sorts the list in descending order

Subroutines for working with standard sets

```
procedure Exclude(var s: set of T;
```
element from the set s
```
procedure Include(var s: set of T;   element: T);
element: T);
```
Deletes

Adds an element to the set s

Subprograms for working with complex numbers

`function Abs(c: Complex): real;` Returns the modulus of a complex number

`function Conjugate(c: Complex): Complex;`

Returns the complex conjugate number

`function Cos(c: Complex): Complex;`

Returns the cosine of a complex number

`function Cplx(re, im: real): Complex;`

Constructs a complex number with a real part re and an imaginary part im

`function CplxFromPolar(magnitude, phase: real): Complex;`

Constructs a complex number using polar coordinates

`function Exp(c: Complex): Complex;`

Returns the exponent of a complex number

`function Ln(c: Complex): Complex;`

Returns the natural logarithm of a complex number `function Log(c: Complex): Complex;`

Returns the natural logarithm of a complex number `function Log10(c: Complex): Complex;`

Returns the decimal logarithm of a complex number

`function Power(c, power: Complex): Complex;`

Returns the degree of a complex number

`function Sin(c: Complex): Complex;`

Returns the sine of a complex number

`function Sqrt(c: Complex): Complex;`

Returns the square root of a complex number

Subroutines for sequence generation

function PartitionPoints(a, b: real; n: integer): **sequence of** real; Returns a sequence of real points to divide the segment [a,b] into n equal parts

function Range(a, b: integer): **sequence of** integer;

Returns a sequence of integers from a to b **function** Range(a, b, step: integer): **sequence of** integer;

Returns a sequence of integers from a to b with step

function Range(a, b: BigInteger): **sequence of** BigInteger;

Returns a sequence of long integers from a to b **function** Range(a, b, step: BigInteger): **sequence of** BigInteger;

Returns a sequence of long integers from a to b with step

function Range(c1, c2: char): **sequence of** char;

Returns a sequence of characters from c1 to c2

function Range(c1, c2: char; step: integer): **sequence of** char;

Returns a sequence of characters from c1 to c2 in increments of step

function ReadSeqInteger(n: integer): **sequence of** integer;

Returns a sequence of n integers entered from the keyboard

function ReadSeqInteger(prompt: string; n: integer): **sequence of** integer;

Outputs an input prompt and returns a sequence of n integers entered from the keyboard **function** ReadSeqIntegerWhile(cond: integer->boolean): **sequence of** integer;

Returns a sequence of integers entered from the keyboard as long as a certain condition is met. **function** ReadSeqIntegerWhile(prompt: string; cond: integer- >boolean): **sequence of** integer;

Outputs the input prompt and returns a sequence of integers entered from the keyboard as long as a certain condition is met

function ReadSeqReal(n: integer): **sequence of** real;

Returns a sequence of n real entered from the keyboard

function ReadSeqReal(prompt: string; n: integer): **sequence of** real;

Outputs an input prompt and returns a sequence of n real ones entered from the keyboard **function** ReadSeqRealWhile(cond: real-

>boolean): **sequence of** real;

Returns a sequence of real, typed from the keyboard as long as a certain condition is met. **function** ReadSeqRealWhile(prompt: string; cond: real- >boolean): **sequence of** real;

Outputs an input prompt and returns a sequence of real, entered from the keyboard as long as a certain condition is met. **function** ReadSeqString(n: integer): **sequence of** string;

Returns a sequence of n strings entered from the keyboard **function** ReadSeqString(prompt: string; n: integer): **sequence of** string;

Outputs an input prompt and returns a sequence of n strings entered from the keyboard **function** ReadSeqStringWhile(cond: string->boolean): **sequence of** strings;

Returns a sequence of strings entered from the keyboard as long as the specified condition is met. **function** ReadSeqStringWhile(prompt: string; cond: string- >boolean): **sequence of** strings;

Outputs an input prompt and returns a sequence of strings entered from the keyboard as long as a certain condition is met. **function** Seq<T>(params **a**: array **of T): sequence of T;**

Returns the sequence of specified elements **function** SeqFill<T>(count: integer; x: T): **sequence of T;**

Returns a sequence of x elements count **function** SeqGen<T>(count: integer; f: integer->T): sequence **of T;**

Returns a sequence of count elements filled with values f(i) **function** SeqGen<T>(count: integer; f: integer->T; from: integer): sequence **of T;**

Returns a sequence of count elements filled with values f(i), starting with i=from **function** SeqGen<T>(count: integer; first: T; next: T->T): sequence **of T;**

Returns a sequence of count elements, starting with first, with the function next moving from the previous to the next **function** SeqGen<T>(count: integer; first, second: T; next: (T,T) ->T): sequence **of T;**

Returns a sequence of count elements, starting with first and second, with the next function moving from the previous two to the next **function** SeqRandom(n: integer := 10; a: integer := 0; b:

```
integer := 100): sequence of integer;
```

Returns a sequence of n random integer elements

```
function SeqRandomInteger(n: integer := 10; a: integer := 0; b: integer := 100): sequence of integer;
```

Returns a sequence of n random integer elements

```
function SeqRandomReal(n: integer := 10; a: real := 0; b: real := 10): sequence of real;
```

Returns a sequence of n random real elements

```
function SeqWhile<T>(first: T; next: T->T; pred: T->boolean): sequence of T;
```

Returns the sequence of elements with the initial value first, the function of the next transition from the previous to
to the next one and a condition for the continuation of the sequence

```
function SeqWhile<T>(first, second: T; next: (T,T) ->T; pred: T->boolean): sequence of T;
```

Returns a sequence of elements starting with first and second, with a function next to go from the previous two to the next and a condition pred to continue the sequence

Subroutines for creating dynamic arrays

```
function Arr<T>(params a: array of T): array of T;
```

Returns an array filled with the specified values of the `function Arr<T>(a: sequence of T): array of T;`

Returns an array filled with values from the sequence

```
function Arr(a: IntRange): array of integer;
```

Returns an array filled with a range of values

```
function Arr(a: CharRange): array of char;
```

Returns an array filled with a range of values

```
function ArrFill<T>(count: integer; x: T): array of T;
```

Returns an array of count elements x `function ArrGen<T>(count: integer; gen: integer->T): array of T;`

Returns an array of count elements filled with gen(i) values

```
function ArrGen<T>(count: integer; gen: integer->T; from: integer): array of T;
```

Returns an array of count elements filled with gen(i) values, starting with i=from `function ArrGen<T>(count: integer; first: T; next: T->T): array of T;`

Returns an array of count elements starting with first, with the function next moving from the previous to the next `function ArrGen<T>(count: integer; first, second: T; next: (T,T) ->T): array of T;`

Returns an array of count elements beginning with first and second, with the next function moving from the previous two to the next `function ArrRandom(n: integer := 10; a: integer := 0; b: integer := 100): array of integer;`

Returns an array of size n filled with random integer values

```
function ArrRandomInteger(n: integer := 10; a: integer := 0; b: integer := 100): array of integer;
```

Returns an array of size n filled with random integer values

```
function ArrRandomReal(n: integer := 10; a: real := 0; b: real := 10): array of real;
```

Returns an array of size n filled with random real values

```
function ReadArrInt64(n: integer): array of int64;
```

Returns an array of n int64 integers entered from the keyboard

```
function ReadArrInteger(n: integer): array of integer;
```

Returns an array of n integers entered from the keyboard

```
function ReadArrInteger(prompt: string; n: integer): array of integer;
```

Outputs an input prompt and returns an array of n integers entered from the keyboard `function ReadArrReal(n: integer): array of real;`

Returns an array of n real entered from the keyboard `function ReadArrReal(prompt: string; n: integer): array of real;`

Outputs an input prompt and returns an array of n real ones entered from the keyboard `function ReadArrString(n: integer): array of string;`

Returns an array of n strings entered from the keyboard

```
function ReadArrString(prompt: string; n: integer): array of string;
```

Outputs an input prompt and returns an array of n strings entered from the keyboard

Subroutines for creating two-dimensional dynamic arrays

```
function Matr<T>(m,n: integer; params data: array of T): array [,] of t;
```
Returns a two-dimensional array of size m x n, filled with the specified values on lines
```
function Matr<T>(params aa: array of array of T): array [,] of T;
```

Returns a two-dimensional array filled with values from one-dimensional arrays
```
function MatrByCol<T>(a: array of array of T): array [,] of T;
```

Generates a two-dimensional array from an array of arrays of columns
```
function MatrByCol<T>(a: sequence of array of T): array [,] of T;
```

Generates a two-dimensional array from a sequence of arrays of columns
```
function MatrByCol<T>(a: sequence of sequence of T): array [,] of T;
```

Generates a two-dimensional array from a sequence of column sequences
```
function MatrByCol<T>(m,n: integer; a: sequence of T): array [,] of T;
```

Generates a two-dimensional array by columns from the sequence
```
function MatrByRow<T>(a: array of array of T): array [,] of T;
```

Generates a two-dimensional array from an array of arrays of strings
```
function MatrByRow<T>(a: sequence of array of T): array [,] of T;
```

Generates a two-dimensional array from a sequence of arrays of rows
```
function MatrByRow<T>(a: sequence of sequence of T): array [,] of T;
```

Generates a two-dimensional array from a sequence of rows
```
function MatrByRow<T>(m,n: integer; a: sequence of T): array [,] of T;
```

Generates a two-dimensional array by strings from the sequence
```
function MatrFill<T>(m, n: integer; x: T): array [,] of T;
```

Returns a two-dimensional array of size m x n, filled with

elements of x

```
function MatrGen<T>(m, n: integer; gen: (integer,integer)->T):
array [,] of T;
```

Returns a two-dimensional array of size m x n, filled with elements gen(i,j)

```
function MatrRandom(m: integer := 5; n: integer := 5; a:
integer := 0; b: integer := 100): array [,] of integer;
```

Returns a two-dimensional array of size m x n filled with random integer values `function MatrRandomInteger(m: integer := 5; n: integer := 5; a: integer := 0; b: integer := 100): array [,] of integer;`

Returns a two-dimensional array of size m x n, filled with random integer values `function MatrRandomReal(m: integer := 5; n: integer := 5; a: real := 0; b: real := 10): array [,] of real;`

Returns a two-dimensional array of size m x n, filled with random real values `function ReadMatrInteger(m, n: integer): array [,] of integer;`

Returns matrix m by n integers entered from the keyboard

```
function ReadMatrReal(m, n: integer): array [,] of real;
```

Returns a matrix m by n real ones entered from the keyboard

```
function Transpose<T>(a: array [,] of T): array [,] of T;
```

Transforms a two-dimensional array

Short functions Lst, LLst, HSet, SSet, Diet, KV

```
function Dict<TKey, TVal>(params pairs: array of
KeyValuePair<TKey, TVal>): Dictionary<TKey, TVal>;
```

Returns a dictionary of pairs of elements (key, value) `function Dict<TKey, TVal>(params pairs: array of (TKey, TVal)): Dictionary<TKey, TVal>;`

Returns a dictionary of element pairs (key, value) `function DictStr(params pairs: array of (string, string)): Dictionary<string, string>;`

Returns a dictionary of element pairs (string, integer) `function DictStrInt(params pairs: array of (string, integer)): Dictionary<string, integer>;`

Returns a dictionary of pairs of elements (string, integer)

```
function HSet<T>(params a: array of T): HashSet<T>;
```

Returns a set based on a hash table filled with the specified values `function HSet<T>(a: sequence of T): HashSet<T>;`

Returns the set based on the hash table, filled with values from the sequence

```
function HSet(a: IntRange): HashSet<integer>;
```

Returns a set based on a hash table filled with a range of values

```
function HSet(a: CharRange): HashSet<char>;
```

Returns a set based on a hash table populated with a range of values `function KV<TKey, TVal>(key: TKey; value: TVal): KeyValuePair<TKey, TVal>;`

Returns a pair of elements (key, value) `function LLst<T>(params a: array of T): LinkedList<T>;`

Returns a doubly linked list filled with the specified values

```
function LLst<T>(a: sequence of T): LinkedList<T>;
```

Returns a doubly linked list filled with values from the sequence

```
function LLst(a: IntRange): LinkedList<integer>;
```

Returns a doubly linked list filled with a range of values

```
function LLst(a: CharRange): LinkedList<char>;
```

Returns a doubly linked list filled with a range of values

```
function Lst<T>(params a: array of T): List<T>;
```

Returns a list filled with the specified values

```
function Lst<T>(a: sequence of T): List<T>;
```
Returns a list filled with values from the sequence
```
function Lst(a: IntRange): List<integer>;
```
Returns a list filled with a range of values
```
function Lst(a: CharRange): List<char>;
```
Returns a list filled with a range of values
```
function SSet<T>(params a: array of T): SortedSet<T>;
```
Returns the set based on the binary search tree, filled with values from the sequence
```
function SSet<T>(a: sequence of T): SortedSet<T>;
```
Returns the set based on the binary search tree, filled with values from the sequence
```
function SSet(a: IntRange): SortedSet<integer>;
```
Returns a set based on a binary search tree, filled with a range of values
```
function SSet(a: CharRange): SortedSet<char>;
```
Returns a set based on a binary search tree, filled with a range of values

Sequence extension methods

Generation of infinite sequences

```
function Cycle<T>(Self: sequence of T): sequence of T;
```

Repeats the sequence an infinite number of times `function Step(Self: integer): sequence of integer;`

Returns an infinite sequence of integers from the current value with step 1 `function Step(Self: integer; step: integer): sequence of integer;`

Returns an infinite sequence of integers from the current step value

```
function Step(Self: real; step: real): sequence of real;
```

Returns an infinite sequence of real values from the current step value

{function Sum<T>(Self: **sequence of T; f:** T->BigInteger): BigInteger; Returns sum of sequence elements projected to a numeric value - does not yet work for Lst(1,2,3) function **AdjacentGroup**<T>(Self: sequence of T): **sequence of array of T;**

Groups identical consecutive elements, resulting in a sequence of arrays **function** Average(Self: **sequence of** BigInteger): real;

Returns the average of sequence elements **function** Batch<T>(Self: **sequence of T;** size: integer): **sequence of sequ**ences **of T;**

Splits the sequence into series of lengths size **function** Batch<T, Res>(Self: **sequence of T;** size: integer; proj: Func<IEnumerable<T>, Res>): sequence of Res;

Splits the sequence into series of length size and applies a projection to each series Cartesian<T, T1>(Self: **sequence of T; b: sequence of T1): sequence of (T, T1**);

Cartesian product of sequences **function** Cartesian<T, T1, T2>(Self: **sequence of T; b: sequence of T1;** func: (T,T1)->T2): sequence **of T2;**

Cartesian product of sequences **function** CountOf<T>(Self: **sequence of T; x:** T): integer;

Returns the number of elements equal to the specified value **procedure** ForEach<T>(Self: **sequence of T;** action: T -> ();

Applies an action to each element of the sequence **procedure** ForEach<T>(Self: **sequence of T;** action: (T,integer) -> ());

Applies an action to each element of the sequence, depending on the element number **function** Incremental(Self: **sequence of** integer): **sequence of** integer;

Returns the sequence of differences of neighboring elements of the original sequence **function** Incremental(Self: **array of** integer): **sequence of** integer;

Returns the sequence of differences of neighboring elements of the original sequence **function** Incremental(Self: List<integer>): **sequence of integer;**

Returns the sequence of differences of neighboring elements of the original sequence **function** Incremental(Self: LinkedList<integer>): **sequence of integer;**

Sequence extension methods

Returns the sequence of differences of neighboring elements of the original sequence `function Incremental(Self: sequence of real): sequence of real;`

Returns the sequence of differences of neighboring elements of the original sequence
`function Incremental(Self: array of real): sequence of real;`

Returns the sequence of differences of neighboring elements of the original sequence
`function Incremental(Self: List<real>): sequence of real;`

Returns the sequence of differences of neighboring elements of the original sequence `function Incremental(Self: LinkedList<real>): sequence of real;`

Returns the sequence of differences of neighboring elements of the original sequence `function Incremental<T, T1>(Self: sequence of T; func: (T,T)->T1): sequence of T1;`

Returns the sequence of differences of neighboring elements of the original sequence. The func `function Incremental<T, T1>(Self: sequence of T; func: (T,T,integer)->T1)` is used as the difference function: `sequence of T1;`

Returns the sequence of differences of neighboring elements of the original sequence. As the difference function func
`function Interleave<T>(Self: sequence of T; a: sequence of T): sequence of T;`

Alternates elements of two sequences `function Interleave<T>(Self: sequence of T; a, b: sequence of T): sequence of T;`

Alternates elements of three sequences of the `function Interleave<T>(Self: sequence of T; a, b, c: sequence of T): sequence of T;`

Alternates elements of four sequences `function JoinToString<T>(Self: sequence of T; delim: string): string;`

Converts the sequence elements to a string representation, then joins them to a string using delim as a delimiter `function JoinToString<T>(Self: sequence of T): string;`

Converts the sequence elements to a string representation, then combines them into a string using space as a separator
`function LastMaxBy<T, TKey>(Self: sequence of T; selector: T-`

```
>TKey): T;
```

Returns the last element of the sequence with the maximum key value `function LastMinBy<T, TKey>(Self: sequence of T; selector: T- >TKey): T;`

Returns the last element of the sequence with the minimal key value `function MaxBy<T, TKey>(Self: sequence of T; selector: T- >TKey): T;`

Returns the first element of the sequence with the maximum key value `function MinBy<T, TKey>(Self: sequence of T; selector: T- >TKey): T;`

Returns the first element of the sequence with the minimum key value `function Numerate<T>(Self: sequence of T): sequence of (integer, T);`

`Numerate<T>(Self: sequence of T; from: integer): sequence of (integer, T);`

Numerates sequence from number from `function Order<T>(Self: sequence of T): sequence of T;`

Returns an ascending sorted sequence of the `function OrderDescending<T>(Self: sequence of T): sequence of T;`

Returns the sequence sorted in descending order `function Pairwise<T>(Self: sequence of T): sequence of (T, T);`

Turns a sequence into a sequence of pairs of neighboring elements `function Pairwise<T, Res>(Self: sequence of T; func: (T,T)- >Res): sequence of Res;`

Turns a sequence into a sequence of pairs of adjacent elements, applies func to each pair of the resulting elements, and obtains a new sequence `function Partition<T>(Self: sequence of T; cond: T->boolean): (sequence of T, sequence of T);`

Splits a sequence into two by a given condition. It is implemented by two-pass algorithm `function Partition<T>(Self: sequence of T; cond: (T,integer)- >boolean): (sequence of T, sequence of T);`

Divides a sequence into two by a given condition that involves an index. Implemented by two-pass algorithm `function Print<T>(Self: sequence of T; delim: string): sequence of T;`

Prints the sequence on the screen, using delim as a delimiter

```
function Print<T>(Self: sequence of T): sequence of T;
```

Prints the sequence on the screen, using space as a separator

```
function PrintLines<T>(Self: sequence of T): sequence of T;
```

Prints a sequence, each element is printed on a new line

```
function PrintLines<T,T1>(Self: sequence of T; map: T->T1): sequence of T;
```

Outputs a sequence, each element is mapped using the map function and displayed on a new line `function Println<T>(Self: sequence of T; delim: string): sequence of T;`

Prints the sequence on the screen, using delim as a delimiter, and jumps to a new line of `the function Println<T>(Self: sequence of T): sequence of T;`

Outputs the sequence on the screen, using space as a separator, and jumps to a new line `function Product(Self: sequence of real): real;`

Returns the product of sequence elements `function Product(Self: sequence of integer): int64;`

Returns the product of the sequence elements

```
function Product<T>(Self: sequence of T; f: T->real): real;
```

Returns the product of sequence elements projected onto a numeric value `function Product<T>(Self: sequence of T; f: T->integer): int64;`

Returns the product of the elements of the sequence projected on the numeric value of `the function Product<T>(Self: sequence of T; f: T->BigInteger): BigInteger;`

Returns the product of the elements of the sequence projected on the numeric value of `the function Product(Self: sequence of BigInteger): BigInteger;`

Returns the product of the elements of the sequence `function SkipLast<T>(self: sequence of T; count: integer := 1): sequence of T;`

Returns a sequence without last count of elements `function Slice<T>(Self: sequence of T; from, step: integer): sequence of T;`

Returns a slice of the sequence from number from with step > 0 `function Slice<T>(Self: sequence of T; from, step, count:`

```
integer): sequence of T;
```

Returns a slice of the sequence from the number from step > 0 of length no more than count

```
function Sorted<T>(Self: sequence of T): sequence of T;
```

Returns `SortedDescending<T>(Self: sequence of T): sequence of T;`

Returns a descending sorted sequence of the `function SplitAt<T>(Self: sequence of T; ind: integer): (sequence of T, sequence of T);`

Splits the sequence into two at the ind position. It is implemented by a two-pass algorithm

```
function Sum(Self: sequence of BigInteger): BigInteger;
```

Returns the sum of sequence elements `function Tabulate<T, T1>(Self: sequence of T; F: T->T1): sequence of (T, T1);`

Tabulates a function with a sequence `function TakeLast<T>(Self: sequence of T; count: integer): sequence of T;`

Returns the last count of the sequence elements

```
function ToHashSet<T>(Self: sequence of T): HashSet<T>;
```

Returns the HashSet set for the given sequence

```
function ToLinkedList<T>(Self: sequence of T): LinkedList<T>;
```

Returns the LinkedList for the given sequence

```
function ToSortedSet<T>(Self: sequence of T): SortedSet<T>;
```

Returns the SortedSet set by the given sequence of `the function UnZipTuple<T, T1>(Self: sequence of (T, T1)): (sequence of T, sequence of T1);`

Unzips a sequence of two-element tuples into two sequences. It is implemented by two-pass algorithm `function UnZipTuple<T, T1, T2>(Self: sequence of (T, T1, T2)): (sequence of T, sequence of T1, sequence of T2);`

Unzips a sequence of three element tuples into three sequences. It is implemented by multipass algorithm `function UnZipTuple<T, T1, T2, T3>(Self: sequence of (T, T1, T2, T3)): (sequence of T, sequence of T1, sequence of T2, sequence of T3);`

Divides a sequence of four element tuples into four sequences. It is implemented by multi-pass algorithm `function WriteLines(Self: sequence of string; fname: string): sequence`

```
of string;
```

Outputs a sequence of lines to a file `function ZipTuple<T, T1>(Self: sequence of T; a: sequence of T1): sequence of (T, T1);`

Combines two sequences into a sequence of two element tuples `function ZipTuple<T, T1, T2>(Self: sequence of T; a: sequence of T1; b: sequence of T2): sequence of (T, T1, T2);`

Combines three sequences into a sequence of three element tuples `function ZipTuple<T, T1, T2, T3>(Self: sequence of T; a: sequence of T1; b: sequence of T2; c: sequence of T3): sequence of (T, T1, T2, T3);`

Combines four sequences into a sequence of four element tuples

Methods for extending one-dimensional dynamic arrays

```
function AdjacentFind<T>(Self: array of T; eq: (T,T)->boolean;
start: integer := 0): integer;
```
Finds the first pair of identical elements in a row using the eq comparison function, and returns the index of the first element of the pair.
If not found, returns -1

```
function ArrEqual<T>(Self,b: array of T): boolean;
```
Returns whether the arrays match or not

```
function BinarySearch<T>(Self: array of T; x: T): integer;
```
Performs a binary search in a sorted array

```
function Cartesian<T>(Self: array of T; n: integer): sequence of array
of T;
```
Returns the nth Cartesian degree of the set of elements given by the array

```
function Combinations<T>(Self: array of T; m: integer):
sequence of array of T;
```
Returns all combinations of m elements

```
function ConvertAll<T, T1>(Self: array of T; converter: T-
>T1): array of T1;
```
Converts the array elements and returns the converted array

```
function ConvertAll<T, T1>(Self: array of T; converter:
(T,integer)->T1): array of T1;
```
Converts the array elements and returns the converted array

```
procedure Fill<T>(Self: array of T; x: T);
```
Fills the elements of the array with the specified value

```
procedure Fill<T>(Self: array of T; f: integer->T);
```
Fills the array elements with values calculated by some rule

```
procedure FillRandom(Self: array of integer; a,b: integer);
```
Fills the array with random values in the range a to b

```
procedure FillRandom(Self: array of real; a,b: real);
```
Fills the array with random values in the range from a to b

```
function Find<T>(Self: array of T; p: T->boolean): T;
```
Performs a search for the first element in the array that satisfies the predicate. If not found, it returns null value of corresponding type

```
function FindAll<T>(Self: array of T; p: T->boolean): array of T;
```

Returns as an array all elements satisfying the predicate

```
function FindIndex<T>(Self: array of T; p: T->boolean): integer;
```

Finds the index of the first element in the array, which satisfies the predicate. If not found, returns -1

```
function FindIndex<T>(Self: array of T; start: integer; p: T- >boolean): integer;
```

Finds the index of the first element in the array that satisfies the predicate, starting from the index start. If not found, returns -1

```
function FindLast<T>(Self: array of T; p: T->boolean): T;
```

Performs search for the last element in the array that satisfies the predicate. If not found, it returns null value of corresponding type

```
function FindLastIndex<T>(Self: array of T; p: T->boolean): integer;
```

Performs a search for the index of the last element in the array that satisfies the predicate. If not found, returns -1

```
function FindLastIndex<T>(self: array of T; start: integer; p: T->boolean): integer;
```

Searches for the index of the last element in the array that satisfies the predicate, in the range of indexes from 0 to start. If not found, it returns -1

```
function High(Self: System.Array);
```

Returns the index of the last element of the array

```
function IndexMax<T>(Self: array of T; index: integer := 0): integer; where T: IComparable<T>;
```

Returns the index of the first maximal element from the index position

```
function IndexMin<T>(Self: array of T; index: integer := 0): integer; where T: IComparable<T>;
```

Returns the index of the first minimal element starting from the index position

```
function IndexOf<T>(Self: array of T; x: T): integer;
```

Returns the index of the first occurrence of the element or -1 if no element is found

```
function IndexOf<T>(Self: array of T; x: T; start: integer): integer;
```

Returns the index of the first occurrence of the element starting from the index start or -1 if the element is not found

```
function Indices<T>(Self: array of T): sequence of integer;
```

Returns sequence of indexes of one-dimensional array

```
function Indices<T>(Self: array of T; cond: T->boolean):
sequence of integer;
```

Returns the sequence of indices of elements of a one-dimensional array that satisfy the condition of `function Indices<T>(Self: array of T; cond: (T,integer) - >boolean): sequence of integer;`

Returns the sequence of indexes of elements of a one-dimensional array that satisfy the condition `LastIndexMax<T>(Self: array of T; index: integer): integer; where T: IComparable<T>;`

Returns the index of the last minimal element in the range [0,index-1] `function LastIndexMax<T>(Self: array of T): integer; where T: IComparable<T>;`

Returns the index of the last maximal element `function LastIndexMin<T>(Self: array of T; index: integer): integer; where T: IComparable<T>;`

Returns the index of the last minimal element in the range [0,index-1] `function LastIndexMin<T>(Self: array of T): integer; where T: IComparable<T>;`

Returns the index of the last minimal element

```
function LastIndexOf<T>(Self: array of T; x: T): integer;
```

Returns the index of the last occurrence of the element or -1 if no element is found `function LastIndexOf<T>(Self: array of T; x: T; start: integer): integer;`

Returns the index of the last occurrence of the element starting from the index start or -1 if the element is not found

```
function Low(Self: System.Array);
```

Returns the index of the first element of the array

```
function Max<T>(Self: array of T): T; where T: IComparable<T>;
```

Returns the maximal element

```
function Max(Self: array of integer): integer;
```

Returns the maximal element

```
function Max(Self: array of real): real;
```

Returns the maximal element

```
function Min<T>(Self: array of T): T; where T: IComparable<T>;
```

Returns the minimum element

```
function Min(Self: array of integer): integer;
```

Returns the minimum element

```
function Min(Self: array of real): real;
```

Returns the minimal element of the function

```
Permutations<T>(Self: array of T): sequence of array of T;
```

Returns all permutations of the set of elements defined by the array function

```
Permutations<T>(Self: array of T; m: integer): sequence of array of T;
```

Returns all partial permutations of n elements by m

```
function RandomElement<T>(Self: array of T): T;
```

Returns a random array element

```
function Shuffle<T>(Self: array of T): array of T;
```

Shuffles the elements of the array at random function

```
Slice<T>(Self: array of T; from, step: integer): array of T;
```

Returns an array slice from index from with step function

```
Slice<T>(Self: array of T; from, step, count: integer): array of T;
```

Returns a slice of the array from the index with a step length of no more than count

```
procedure Sort<T>(Self: array of T);
```

Sorts the array in ascending order

```
procedure Sort<T>(Self: array of T; cmp: (T,T) ->integer);
```

Sorts the array in ascending order using cmp as an element comparison function

```
procedure Transform<T>(Self: array of T; f: T->T);
```

Transforms array elements according to the given rule

```
procedure Transform<T>(Self: array of T; f: (T,integer)->T);
```

Converts the array elements according to a given rule

Methods for extending two-dimensional dynamic arrays

```
function Col<T>(Self: array [,] of T; k: integer): array of T;
```
the k-th column of a two-dimensional array

```
function ColCount<T>(Self: array [,] of T): integer;
```
Number of columns in a two-dimensional array

```
function Cols<T>(Self: array [,] of T): array of array of T;
```
Returns an array of columns of a two-dimensional array

```
function ColSeq<T>(Self: array [,] of T; k: integer): sequence of T;
```
the k-th column of the two-dimensional array as a sequence

```
function ColsSeq<T>(Self: array [,] of T): sequence of sequences of T;
```
Returns a sequence of columns of a two-dimensional array

```
function ConvertAll<T, T1>(Self: array [,] of T; converter: T->T1): array [,] of T1;
```
Converts elements of a two-dimensional array and returns the converted array

```
function ConvertAll<T, T1>(Self: array [,] of T; converter: (T,integer,integer)->T1): array [,] of T1;
```
Converts the elements of a two-dimensional array and returns the converted array

```
function ElementsByCol<T>(Self: array [,] of T): sequence of T;
```
Returns by the given two-dimensional array the sequence of its elements by columns

```
function ElementsByRow<T>(Self: array [,] of T): sequence of T;
```
Returns, for a given two-dimensional array, a sequence of its elements by strings

```
function ElementsWithIndices<T>(Self: array [,] of T): sequence of (T, integer, integer);
```
Returns the sequence (a[ij],ij) of the given two-dimensional array

```
procedure Fill<T>(Self: array [,] of T; f: (integer,integer) - >T);
```
Fills the elements of a two-dimensional array with values calculated according to some rule

```
procedure FillRandom(Self: array [,] of integer; a,b: integer);
```
Fills the elements of a two-dimensional array with random values in the range a to b

```
procedure FillRandom(Self: array [,] of real; a,b: real);
```

Fills the elements of a two-dimensional array with random values in the range a to b

```
procedure ForEach<T>(Self: array [,] of T; act: T -> ();
```

Applies an action to each element of a two-dimensional array

```
procedure ForEach<T>(Self: array [,] of T; act:
(T,integer,integer) -> ());
```

Applies an action to each element of a two-dimensional array

```
function Indices<T>(Self: array [,] of T; cond: T -> boolean):
sequence of (integer, integer);
```

Returns, for a given two-dimensional array, a sequence of indices of elements satisfying the given condition `function Indices<T>(Self: array [,] of T; cond: (T,integer,integer) -> boolean): sequence of (integer, integer);`

Returns, for a given two-dimensional array, a sequence of element indices that satisfy the given condition `function MatrEqual<T>(Self,b: array[,] of T): boolean;`

Is there an element in the matrix

```
function MatrSlice<T>(Self: array[,] of T; RowIndex: array of
integer; ColIndex: array of integer): array[,] of T;
```

Returns a slice of a two-dimensional array. RowIndex and ColIndex set the rows and columns to be sliced

```
function MatrSlice<T>(Self: array[,] of T; FromRow, ToRow,
FromCol, ToCol: integer): array[,] of T;
```

Returns a slice of a two-dimensional array between rows FromRow, ToRow and columns FromCol, ToCol `function operator in<T>(x: T; a: array[,] of T): boolean;`

Is there an element in the matrix

```
function Print<T>(Self: array [,] of T; w: integer := 4): array
[,] of T;
    Print(Self: array [,] of real; w: integer := 7; f: integer
:= 2): array [,] of real;
```

Output two-dimensional real array by format :w:f `function Println<T>(Self: array [,] of T; w: integer := 4): array [,] of T;`

```
    Println(Self: array [,] of real; w: integer := 7; f:
integer := 2): array [,] of real;
```

Output a two-dimensional real array in the format :w:f and skip to the next line

```
function Row<T>(Self: array [,] of T; k: integer): array of T;
```

the k-th row of the two-dimensional array

```
function RowCount<T>(Self: array [,] of T): integer;
```

Number of rows in a two-dimensional array

```
function Rows<T>(Self: array [,] of T): array of array of T;
```

Returns an array of rows of a two-dimensional array `function RowSeq<T>(Self: array [,] of T; k: integer): sequence of T;`

the k-th row of a two-dimensional array as a sequence

```
function RowsSeq<T>(Self: array [,] of T): sequence of sequences of T;
```

Returns the string sequence of a two-dimensional array

```
procedure SetCol<T>(Self: array [,] of T; k: integer; a: array of T);
```

Changes column k of a two-dimensional array to another column

```
procedure SetCol<T>(Self: array [,] of T; k: integer; a: sequence of T);
```

Changes column k of a two-dimensional array to another column of the `procedure SetRow<T>(Self: array [,] of T; k: integer; a: array of T);`

Changes row k of a two-dimensional array to another row

```
procedure SetRow<T>(Self: array [,] of T; k: integer; a: sequence of T);
```

Changes the string k of a two-dimensional array to another string

```
function Size<T>(Self: array [,] of T): (integer,integer);
```

Number of columns in a two-dimensional array

```
procedure SwapCols<T>(Self: array [,] of T; k1, k2: integer);
```

Swaps two columns of a two-dimensional array with numbers k1 and k2

```
procedure SwapRows<T>(Self: array [,] of T; k1, k2: integer);
```

Swaps two rows of a two-dimensional array with numbers k1 and k2

```
procedure Transform<T>(Self: array [,] of T; f: T->T);
```

Transforms elements of a two-dimensional array according to the given rule `procedure Transform<T>(Self: array [,] of T; f: (T,integer,integer)->T);`

Converts the elements of a two-dimensional array according to a given rule

Methods for expanding lists

`function AdjacentFind<T>(Self: IList<T>; start: integer := 0): integer; Finds` the first pair of identical elements in a row and returns the index of the first element of the pair. If not found, it returns -1

`function AdjacentFind<T>(Self: IList<T>; eq: (T,T)->boolean; start: integer := 0): integer;`

Finds the first pair of consecutive identical elements using the eq comparison function, and returns the index of the first element of the pair. If not found, it returns -1

`procedure Fill<T>(Self: List<T>; x: T);`

Fills the elements of the list with the specified value

`procedure Fill<T>(Self: List<T>; f: integer->T);`

Fills the list elements with values calculated by some rule

`function IndexMax<T>(Self: List<T>; index: integer := 0): integer; where T: IComparable<T>;`

Returns the index of the first maximal element starting from the index position

`function IndexMin<T>(Self: List<T>; index: integer := 0): integer; where T: IComparable<T>;`

Returns the index of the first minimal element starting from the index position

`function Indices<T>(Self: List<T>): sequence of integer;`

Returns a sequence of list indexes

`function Indices<T>(Self: List<T>; cond: T->boolean): sequence of integer;`

Returns the sequence of indices of the list elements satisfying the condition `function Indices<T>(Self: List<T>; cond: (T,integer) - >boolean): sequence of integer;`

Returns the sequence of indices of the list elements that satisfy the condition `LastIndexMax<T>(Self: List<T>; index: integer): integer; where T: IComparable<T>;`

Returns the index of the last minimal element in the range [0,index-1] `function LastIndexMax<T>(Self: List<T>): integer; where T: IComparable<T>;`

Returns the index of the last maximal element `function LastIndexMin<T>(Self: List<T>; index: integer): integer; where T: IComparable<T>;`

Returns the index of the last minimal element in the range [0,index-1] `function LastIndexMin<T>(Self: List<T>): integer; where T: IComparable<T>;`

Returns the index of the last minimal element of the `function RemoveLast<T>(Self: List<T>): List<T>;`

Deletes the last element. If there are no elements, generates an exception `procedure Replace<T>(Self: List<T>; oldValue, newValue: T);`

Replaces all occurrences of one value in an array with anotherReplaces all occurrences of one value in a list with another `function Shuffle<T>(Self: List<T>): List<T>;`

Shuffles the elements of the list at random `function Slice<T>(Self: List<T>; from, step: integer): List<T>;`

Returns a slice of the list from the index from step `function Slice<T>(Self: List<T>; from, step, count: integer): List<T>;`

Returns a slice of the list from the index with step length not more than count
`procedure Transform<T>(Self: List<T>; f: T->T);`

Converts the elements of an array or list according to a given rule
`procedure Transform<T>(Self: List<T>; f: (T,integer)->T);`

Converts the elements of an array or list according to a given rule

Methods for extending the type integer

```
function Between(Self: integer; a, b: integer): boolean;
```

Returns True if the value is between the other two `function Clamp(Self: integer; bottom,top: integer): integer;`

Returns a number bounded by the range from bottom to top inclusive `function ClampBottom(Self: integer; bottom: integer): integer;`

Returns the number bounded by the bottom value of the `function ClampTop(Self: integer; top: integer): integer;`

Returns the number bounded by top `function Divs(Self,d: integer): boolean;`

Returns True if the integer is divisible by the specified value `function DivsAll(Self: integer; params a: array of integer): boolean;`

Returns True if the integer is divisible by all values `function DivsAny(Self: integer; params a: array of integer): boolean;`

Returns True if the integer is divisible by one of the values of `function Downto(Self: integer; n: integer): sequence of integer;`

Generates a sequence of integers from the current value to n in descending order `function InRange(Self: integer; a,b: integer): boolean;`

Returns True if the value is between the other two `function IsEven(Self: integer): boolean;`

Returns whether an integer is even. `function IsOdd(Self: integer): boolean;`

Returns whether the integer is odd `function NotDivs(Self,d: integer): boolean; > 0;`

Returns True if the integer is not divisible by the specified value `function Range(Self: integer): sequence of integer;`

Returns a sequence of numbers from 1 to this `function Sqr(Self: integer): integer;`

Returns the square of the number `function Sqrt(Self: integer): real;`

Returns the square root of a number

```
function Times(Self: integer): sequence of integer;
```

Returns a sequence of integers 0,1,...n-1

```
function To(Self: integer; n: integer): sequence of integer;
```

Generates a sequence of integers from the current value to n

Extension methods of BigInteger type

```
function Sqrt(Self: BigInteger): real;
```

Returns square root of a number

Methods of extending the type real

```
function Between(Self: real; a, b: real): boolean;
```

Returns True if the value is between the other two

```
function Clamp(Self: real; bottom,top: real): real;
```

Returns a number limited by the range from bottom to top inclusive

```
function ClampBottom(Self: real; bottom: real): real;
```

Returns the number bounded by the bottom value

```
function ClampTop(Self: real; top: real): real;
```

Returns the number bounded by the top value

```
function InRange(Self: real; a,b: real): boolean;
```

Returns True if the value is between the other two

```
function Round(Self: real): integer;
```

Returns the number rounded to the nearest integer

```
function Round(Self: real; digits: integer): real;
```

Returns x rounded to the nearest real with digits after the decimal point

```
function RoundBigInteger(Self: real): BigInteger;
```

Returns the number rounded to the nearest long integer

```
function Sqr(Self: real): real;
```

Returns the square of the number

```
function Sqrt(Self: real): real;
```

Returns the square root of a number

```
function ToString(Self: real; frac: integer): string;
```

Returns real, formatted to a string with frac digits after the decimal point

```
function Trunc(Self: real): integer;
```

Returns the integer part of a real number **function**

```
TruncBigInteger(Self: real): BigInteger;
```

Returns the integer part of a real number as a long integer

Methods for extending char type

```
function Between(Self: char; a, b: char): boolean;
```

Returns True if the value is between the other two

```
function Code(Self: char): integer;
```

Unicode character code

```
function InRange(Self: char; a,b: char): boolean;
```

Returns True if the value is between the other two

```
function IsDigit(Self: char);
```

Is the symbol a number

```
function IsLetter(Self: char): boolean;
```
Is the symbol a letter
```
function IsLower(Self: char): boolean;
```
Whether the character belongs to the category of lowercase letters
```
function IsUpper(Self: char): boolean;
```
Whether the character belongs to the upper-case letter category
```
function Pred(Self: char);
```
Previous Symbol
```
function Succ(Self: char);
```
The next symbol
```
function ToDigit(Self: char): integer;
```
Converts symbol to a number
```
function ToLower(Self: char): char;
```
Converts character to lower case
```
function ToUpper(Self: char): char;
```
Converts character to uppercase

Methods for extending the string type

```
function Between(Self: string; a, b: string): boolean;
```

Returns True if the value is between two other **functions**

```
IndicesOf(Self, SubS: string; overlay: boolean := False):
sequence of integer;
```

Returns the sequence of indices of substring occurrences in the main stringThe overlay parameter determines whether overlapping of substring occurrences is allowed

```
function InRange(Self: string; a, b: string): boolean;
```

Returns True if the value is between two other **functions**

```
Inverse(Self: string): string;
```

Returns the string inversion

```
function IsMatch(Self: string; reg: string; options:
RegexOptions := RegexOptions.None): boolean;
```

Whether the string satisfies the regular expression **function**

```
Left(Self: string; length: integer): string;
```

Returns the substring obtained by cutting the leftmost characters from the length string **function**

```
Matches(Self: string;
reg: string; options: RegexOptions := RegexOptions.None):
sequence of Match;
```

Searches all occurrences of a regular expression in the specified string and returns them as a sequence of elements of type Match **function**

```
MatchValue(Self: string; reg: string; options:
RegexOptions := RegexOptions.None): string;
```

Searches for the first occurrence of a regular expression in the specified string and returns it as a string **function**

```
MatchValues(Self: string; reg: string; options: RegexOptions
:= RegexOptions.None): sequence of string;
```

Searches all occurrences of a regular expression in the specified string and returns them as a string sequence **function**

```
ReadInteger(Self: string; var from: integer): integer;
```

Reads an integer from a string starting at the from position and sets from after the read value

```
function ReadReal(Self: string; var from: integer): real;
```

Reads the real from the string starting at the from position and

sets from after the read value

```
function ReadWord(Self: string; var from: integer): string;
```

Reads a word from a string starting at the from position and sets from after the read value

```
function RegexReplace(Self: string; reg, repl: string; options: RegexOptions := RegexOptions.None): string;
```

Replaces all occurrences of a regular expression in the specified string with the specified replacement string and returns the transformed string

```
function RegexReplace(Self: string; reg: string; repl: Match- >string; options: RegexOptions := RegexOptions.None): string;
```

Replaces all occurrences of a regular expression in the specified string with the specified substitution transformation and returns the transformed string

```
function Remove(Self: string; params targets: array of string): string;
```

Deletes all occurrences of the specified strings in the string

```
function Replace(Self: string; oldStr,newStr: string; count: integer): string;
```

Replaces the count of occurrences of the oldStr substring with the newStr substring in the original string

```
function Right(Self: string; length: integer): string;
```

Returns the substring obtained by cutting the rightmost characters from the string

```
function Slice(Self: string; from, step: integer): string;
```

Returns a slice of the string from the index from with step

```
function Slice(Self: string; from, step, count: integer): string;
```

Returns a slice of the string from the index from with step length not more than count

```
function ToBigInteger(Self: string): BigInteger;
```

Converts string to BigInteger

```
function ToInteger(Self: string): integer;
```

Converts a string into an integer

```
function ToInteger(Self: string; defaultvalue: integer): integer;
```

Converts string to integerIf conversion fails it returns defaultvalue

```
function ToIntegers(Self: string): array of integer;
```

Converts string to an array of integers

```
function ToIntegers(Self: string; N: integer): array of integer;
```

Reads an array of N integers from a string

```
function ToReal(Self: string): real;
```

Converts the string to a real

```
function ToReal(Self: string; defaultvalue: real): real;
```

Converts a string to a real value If conversion is not possible the defaultvalue `function ToReals(Self: string): array of real;`

Converts string to an array of real

```
function ToWords(Self: string; params delim: array of char): array of string;
```

Converts string to an array of words

```
function ToWords(Self: string; delims: string := ' '): array of string;
```

Converts a string into an array of words, using delims characters from the string `function TryToInteger(Self: string; var value: integer): boolean;`

Converts a string to an integer and writes it to value-If the conversion fails, it returns False

```
function TryToReal(Self: string; var value: real): boolean;
```

Converts string to a real value and writes it to value-if the conversion fails, it returns False

Func extension methods

```
function Compose<T1, T2, TResult>(Self: T2->TResult; composer: T1->T2): T1->TResult;
```

superposition of fUNCTION

Complex extension methods

```
function Conjugate(Self: Complex): Complex;
```

Returns complex conjugate value

Dictionary expansion methods

`function Each<Key,Source,Res>(Self: sequence of System.Linq.IGrouping<Key,Source>; grOperation: System.Linq.IGrouping<Key,Source> -> Res): Dictionary<Key,Res>;` Returns the dictionary that maps the group key to the result of the group operation

`EachCount<Key,Source>(Self: sequence of System.Linq.IGrouping<Key,Source>): Dictionary<Key,integer>;`

Returns the dictionary that maps the number of elements with this key to the group key `function Get<Key, Value>(Self: IDictionary<Key, Value>; K: Key): Value;`

Returns the value associated with the specified key in the dictionary, and if there is no such key, the default `procedure operator-=<Key,Value>(Self: IDictionary<Key,Value>; k: Key);`

Operation of deleting a pair with a specified key value from the dictionary

Common methods of file types

The following methods are defined for all standard file types (`Text`, **`file`**, **`file of`** `T`):

`procedure` `Write(a,b,...);` Writes values a, b, ... to the file.
`function` `Name: string;`

Returns the file name
`function` `FullName: string;`

Returns the full name of the file **`function`** `Eof: boolean;`

Returns True if the end of the file is reached, and False otherwise **`procedure`** `Close;`

Closes the file **`procedure`** `Erase;`

Deletes the file **`procedure`** `Rename(newname: string);`

Renames the file, giving it the name newname

Text file methods

The following methods are defined for text files (type `Text`):

procedure `Write(a,b,...);` Writes values a, b, ... to the file.
procedure `Writeln(a,b,...);`
Writes values a, b, ... into the file. and moves to the next line
procedure `Print(a,b,...);`
Writes values a, b, ... into the file, separating them with spaces
procedure `Println(a,b,...);`
Writes values a, b, ... to the file, separating them with spaces, and skips to the next line of **the function** `Readinteger: integer;`
Returns the value of type integer entered from the text file
function `ReadReal: real;`
Returns the value of type real, entered from the text file
function `ReadChar: char;`
Returns the char value entered from the text file **function** `ReadString: string;`
Returns the value of string type entered from a text file, without skipping to the next line **function** `ReadBoolean: boolean;`
Returns the boolean type value entered from the text file
function `ReadWord: string;`
Returns the word entered from the text file **function** `Readlninteger: integer;`
Returns an integer value entered from a text file and skips to the next line of **the function** `ReadlnReal: real;`
Returns the value of type real, entered from the text file, and goes to the next line of **the function** `ReadlnChar: char;`
Returns a char value entered from a text file and skips to the next line of the **function** `ReadlnString: string;`
Returns the value of string type entered from a text file and skips to the next line of **the function** `ReadlnBoolean: boolean;`
Returns a boolean value entered from a text file and skips to the next line of **the function** `ReadlnWord: string;`
Returns the word entered from the text file and skips to the next line of **the function** `Eof: boolean;`
Returns True if the end of the file is reached, and False

otherwise `function Eoln: boolean;`

Returns True if the end of the string is reached, and False otherwise `function SeekEof: boolean;`

Skips whitespace, then returns True if the end of the file is reached `function SeekEoln: boolean;`

Skips whitespace characters, then returns True if the end of a line in the file `function Name: string` is reached;

Returns the file name

`function FullName: string;`

Returns the full file name

`procedure Reset;`

Opens a text file for reading in the Windows `procedure Reset(en: Encoding);`

Opens a text file for reading in the specified encoding `procedure Rewrite;`

Opens a text file for writing in Windows encoding `procedure Rewrite;`

Opens a text file for writing in the specified encoding `procedure Append;`

Opens a text file in Windows encoding

`procedure Append;`

Opens a text file for addition in the specified encoding

`procedure Close;`

Closes the file

`procedure Flush;`

Writes the contents of the file buffer to disk `function ReadToEnd: string;`

Returns as a string the contents of the file from the current position to the end of `the Erase procedure;`

Deletes file (file must be closed) `procedure Rename(newname: string);`

Renames the file, giving it the name newname (the file must be closed) `function Lines: sequence of string;`

Returns the string sequence of the opened text file

Methods of Binary Bespoke Files

The following methods are defined for the binary files without binary **files:**

procedure Reset; Opens an existing unsaved file for reading and writing
procedure Rewrite;

Opens an existing unsigned file for reading and writing. If the file did not exist it is created, if it existed it is reset **by the** Reset(en: Encoding) **procedure;**

Opens an existing unsigned file for reading and writing in the specified encoding **procedure** Rewrite(en: Encoding);

Opens an existing unsigned file for reading and writing in the specified encoding. If the file did not exist, it is created, if it did exist, it is zeroed **by function** Position: int64;

Returns or sets the current position of the file pointer in the zipless file **function** Size: int64;

Returns the number of bytes in the thumbnail file **procedure** Seek(n: int64);

Sets the current position of the file pointer in the placeholder file to the byte number n **procedure** Truncate;

Truncates a zipless file by discarding all elements from the position of the file pointer **procedure** WriteBytes(a: **array of** byte);

Writes data from a byte array to a plain file **function** ReadBytes(count: integer): **array of** byte;

Reads the specified number of bytes from the byte-free file to the byte array **function** ReadInteger: integer;

Reads an integer from a binary file
function ReadBoolean: boolean;

Reads the boolean from the batchless file **function** ReadByte: byte;

Reads a byte from a binary file with **function** ReadChar: char;

Reads a symbol from a symbol-free file **function** ReadReal: real;

Reads real from a binary file **function** ReadString: string;

Reads a string from a binspace file

Typed file methods

The following methods are defined for typed files `of t:`

`function Position: int64;` Returns the current position of the file pointer in a typed file
`function Size: int64;`

Returns the number of elements in a typed file `procedure Seek(n: int64);`

Sets the current position of the file pointer in the typed file to the element with the number n `procedure Truncate;`

Truncates a typed file by discarding all elements from the file pointer position

Typed file extension methods

```
function Elements<T>(Self: file of T): sequence of T;
```

Returns sequence of elements of an opened typed file

```
function Read<T>(Self: file of T): T;
```

Reads and returns next element of typed file `function Read2<T>(Self: file of T): (T,T);`

Reads and returns the following two elements of a typed file as a tuple `function Read3<T>(Self: file of T): (T,T,T);`

Reads and returns the following three elements of a typed file as a tuple `function ReadElements<T>(Self: file of T): sequence of T;`

Returns the sequence of elements of an open typed file from the current element to the end `function ReadElements<T>(fname: string): sequence of T;`

Opens a typed file, returns a sequence of its elements and closes the `procedure Reset<T>(Self: file of T);`

Opens an existing typed file `procedure Rewrite<T>(Self: file of T);`

Creates a new or nulls an existing typed file `function Seek<T>(Self: file of T; n: int64): file of T;`

Sets the current position of the file pointer in the typed file to the element with number n `procedure Write<T>(Self: file of T; params vals: array of T);`

Writes data to a typed file

OpenMP: overview

OpenMP *is* open standard for paralleling programs on multiprocessor systems with shared memory (for example, on multicore processors). OpenMP implements parallel computing using multithreading: the main thread creates a set of slave threads and the task is distributed between them.

OpenMP is a set of compiler directives that control the automatic allocation of threads and the data needed to run those threads.

In PascalABC.NET the following OpenMP elements are implemented:

- Constructions for creating and distributing work between threads (`parallel for` and `parallel sections` directives)
- Constructions for thread synchronization (`critical` directive)

Directives have the following form: `{$omp directive-name [option[,] option]...]}` Here `$omp` means that it is an OpenMP directive, `-directivename` is the name of the directive, for example `parallel`, after which there can be options. **The directive refers to the operator before which it is placed.**

Examples of using OpenMP are in the Samples/OMPSamples folder

The following is a description of the directives.

The parallel for

The `parallel for` directive provides paralleling of the loop that follows it.

```
{$omp parallel for}
for var i: integer:=1 to 10 do loop body
```

Several threads will be created here and different iterations of the loop will be distributed across these threads. The number of threads is usually the same as the number of processor cores, but in some

cases there may be differences, for example if a thread is waiting for user input, additional threads may be created in order to use all available cores if possible.

All variables defined outside the parallel loop will be shared, i.e. if the loop's body refers to such variables, all the threads will refer to the same memory location. All variables declared inside the loop will be private, i.e. each thread will have its own copy of this variable.

The `private` option allows variables described outside the loop to be private. The option is written as follows:

```
{$omp parallel for private(list of variables)}
```

Variable list - one or more variables, separated by commas.

```
var a,b: integer;
{$omp parallel for private(a, b)} for var i: integer:=1 to 10
do a :=...
```

In this case, variables a and b will be private, and assigning these variables in one thread will not affect other threads.

Limitation: the counters of paralleled loops and nested loops must be declared in the loop header.

Not all loops can be parallelized. If the same variable is accessed at different iterations and its value changes, paralleling such a loop will lead to errors; different runs may yield different results depending on the order in which the variable was accessed.

```
{$omp parallel for} for var i:=1 to 2 do a[i] := a[i+1];
```

Here the first iteration reads the second element of the array, and the second iteration writes the same element. If the first iteration occurs before the second iteration, the first element of the array will be written from the second, and if later, from the third element of the array.

```
var a:integer;
{$omp parallel for} for var i:=1 to 10 do begin a := i;
  ... := a; // by this point a can be changed by another thread
end;
```

The value of the variable a after this cycle can be anything from 1 to 10.

Loops are most effectively parallelized if each iteration of the loop takes quite a long time to execute. If the loop body consists of a

small number of simple statements, the cost of creating threads and distributing the load among them may exceed the benefit of parallel execution of the loop.

Example of parallel matrix multiplication

Matrix multiplication is a classic example to illustrate parallelism. The calculation of the different matrix elements takes place independently, so there is no need to provide for any means of synchronization.

```
  Arrays;
  procedure ParallelMult(a,b,c: array [,] of real; n: integer);
  begin
    {$omp parallel for } for var i:=0 to n-1 do for var j:=0 to
    n-1 do begin
      c[i,j]:=0;
      for var l:=0 to n-1 do
        c[i,j]:=c[i,j]+a[i,l]*b[l,j];
    end;
  end;

  procedure Mult(a,b,c: array [,] of real; n: integer); begin
  for var i:=0 to n-1 do for var j:=0 to n-1 do begin
      c[i,j]:=0;
      for var l:=0 to n-1 do
        c[i,j]:=c[i,j]+a[i,l]*b[l,j];
    end;
  end;

  const n = 400;

  begin
    var a := Arrays.CreateRandomRealMatrix(n,n);
    var b := Arrays.CreateRandomRealMatrix(n,n);
    var c := new real[n,n];
    ParallelMult(a,b,c,n);
    writeln('Parallel matrix multiplication:
  ',Milliseconds,'); var d := Milliseconds;
    Mult(a,b,c,n);
    writeeln('Non-parallel matrix multiplication:
  ',Milliseconds-d,' milliseconds');
  end.
```

Reduction in the parallel for directive

Often a variable is accumulated in a loop, this variable is initialized before the loop, and at each iteration some value is added to it or multiplied by some value. This variable must be declared outside the loop, so it will be shared. In this case, parallel execution errors are possible:

```
var a: integer:=0; {$omp parallel for} for var i:integer:=1
to 100 do a := a+1;
```

Two threads can read the old value, then the first thread will add one and write it to the a variable, then the second thread will add one to the old value and write the result to the a variable. In this case the changes made by the first thread will be lost. The program can work correctly with some runs, but errors are also possible.

The `reduction` option allows you to ensure the correct accumulation of results:

```
{$omp parallel for reduction(action : list of variables)}
```

In this case all variables from the list will be declared private, so different threads will work with their own instances of variables. These instances will be initialized depending on the action, and at the end of the loop the new value of the variable will be obtained from the value of that variable before the loop and all private copies of the action application from the option.

```
var a: integer := 1;
{$omp parallel for reduction(+:a)} for var i: integer:=1 to
2 do a := a+1;
```

Here the initial value of variable a is one, for action + local copies will be initialized with zeros, two iterations will be executed and for each thread the local copy of variable a will take value 1. After the loop ends, both local copies will be added to the initial value (1), and the resulting value of the a variable will equal 3, the same as in the sequential execution. The table lists allowable reduction operators and values with which the local copies of the reduction variable are initialized:

Partition operator reduction	Initialized value
+	0

*	1
-	0
and (beaten)	~0 (each bit set)
or (beaten)	0
xor (bitwise)	0
and (logical)	true
or(logical)	false

Parallel sections and the parallel sections directive

The `parallel sections` directive provides parallel execution of several operators, simple or compound.

```
{$omp parallel sections}
begin
  Section 1;
  Section 2;
  --; end;
```

Each operator in the **`begin ... end`** that follows the directive is a separate section.

```
{$omp parallel sections}
begin
  operator 1;
  operator 2;
  begin
    operator 3;
    operator 4;
    operator 5;
  end;
end;
```

There are three parallel sections described here, the first is operator 1, the second is operator 2, and the third is the `begin` ... `end`, consisting of operators 3-5.

All variables described outside the parallel sections will be shared, i.e. if the sections refer to such variables, the threads executing those sections will refer to the same memory location. All variables declared inside a section will be accessible only in the section in which they are declared.

Correct operation of parallel sections is possible only if the sections are independent of each other - if they can be executed in any order and do not access or change the same variables.

Synchronization and directive critical

The `critical` directive excludes the parallel execution of the operator that follows it.

```
{$omp critical name} operator;
```

This operator forms a critical section - a section of code that cannot

be executed simultaneously by multiple threads.

Only critical sections with the same name cannot be executed simultaneously. If one thread is already running a critical section and a second thread tries to enter a section with the same name, it will be blocked until the first thread leaves the critical section.

Critical sections can be used when accessing shared variables to avoid data loss.

```
var a:integer:=0;
{$omp parallel for} for var i:integer:=1 to 100 do {$omp
critical} a:=a+1;
```

Here the critical section can be used instead of reduction. The entire operator `a:=a+i` will be executed by one thread and only then by the other thread. However, the use of critical sections usually decreases efficiency by sequentially executing these sections. In this example, the whole loop body is a critical section, so the whole loop will be executed sequentially.

But not in all cases the use of critical sections helps to ensure the correct operation of parallel designs.

```
var a:integer := 0;
{$omp parallel sections}
begin
  {$omp critical}
  a:=1;
  {$omp critical}
  a:=a+1; end;
```

The value of a depends on the order in which the sections are executed. If the first section runs first, the value of a will be two, otherwise it will be one.

When critical sections are used, interlocks may occur. For example, the first thread executes code containing critical section A with critical section B inside. The second thread executes code containing critical section B, inside which critical section A exists. The first thread enters section A and fails to enter section B. The second thread enters section B and cannot enter section A, since that section is already being executed by another thread. The first thread cannot continue execution, since section B is already in execution by another thread. Both threads are blocked.

Compiler directives

Compiler directives are special commands to the compiler during compilation, written in the program text within the sequence {$ ... }. The curly brackets denote a comment, but the presence of the $ sign after { indicates that there is a compiler directive inside the comment.

General view of the compiler directive:

```
{$Directive parameter name}
```

List of compiler directives

{$apptype <application type>} - Set the type of the application (windows/console).

preference <file name>} - library connection.

{$gendoc <parameter>} -generates documentation in XML format. Parameters: **true**, **false**.

{$mainresource <file name>} - Connecting the .res file in as an unmanaged resource

{$resource <file name>} - Connecting the file as a Managed resource

{$region <region name>} - The beginning of the region (used in the editor in code collapse mode

{$endregion} - end region

{$title <description>} - The name of the assembly as an information product

{$description <description>} - A brief description of the assembly

{$product <product name>} - product info

{$version <product version>} - product version

{$company<company>} - company

{$copyright <copyright>} - copyright

{$trademark <trademark>} - trademark

{$include <file name>} - Inclusion of the contents of the specified file into the program text.

{$define <identifier>} -** definition of the name used in **$ifdef, **$ifndef** directives.

{$undef <identifier>} -** name exception, used to override the **$define directive.

{$ifdef <identifier>} - start of conditional compilation block (condition: "identifier defined" is checked).

{$ifndef <identifier>} - start of conditional compilation block (condition: "identifier not defined" is checked).

{$else} - "otherwise" directive in the conditional compilation block.

{$endif} - end of conditional compilation block.

{$faststrings} are strings with fast character access per write, but with reference semantics.

{$string_nullbased+} - including strings indexed from 0.

{$string_nullbased-}** - {$string_nullbased+} - turning off strings indexed from 0. **{$platformtarget x86} - compiling for 32-bit platform (required for 32-bit dlls)

{$platformtarget x64} - compilation for the 64-bit platform

The **$ifdef**, **$ifndef** directives together with the **$else** and **$endif** directives control conditional compilation of parts of the source file. Each **$ifdef**, **$ifndef** directive must correspond to the **$endif** directive which completes it. Any number of conditional compilation blocks (including nested ones) and no more than one **$else** directive are allowed between **$ifdef**, **$ifndef** and **$endif** directives.

Example. Using conditional compilation directives.

```
{$define DEBUG}
begin
  {$ifndef DEBUG}
    writeeln('The name DEBUG is not defined');
  {$else}
    writeeln('The name DEBUG is defined');
  {$endif}
end.
```

GraphWPF module: overview

The `GraphWPF` module is a simple graphics library and is designed to create graphics and animation programs in procedural and partially object style. Drawing is done in a special *graphics window*; it is not possible to draw in multiple windows. In addition, `GraphWPF has` simple mouse and keyboard events that can be used to create elementary event driven applications. The main use of `GraphWPF is` for training.

The `GraphWPF` module is based on the WPF graphics library and is a modern and improved version of the outdated `GraphABC` module.

The `GraphWPF` module defines a number of constants, types, procedures, functions and classes for drawing in *the graphics window*. They are divided into the following groups:

- ГGraphic primitives
- ФFunctions дforдtext output
- ФFunctions дforдgraph output
- ФFunctions дforд outputvideo
- ФFunctions д systemрдcoordinate
- proceduresFrameданimationии
- auxiliary functionsGraphWPF
- typesдmoduleGraphWPF
- variablesдmoduleGraphWPF
- Events дmoduleGraphWPF
- classBrushTypeype
- classPenTypeype
- classFontTypeype
- classWindowTypeype
- classWindowTypeWPFypeWPF
- classGraphWindowTypeype

Graphic primitives

procedure `Arc(x, y, r, angle1, angle2: real);` Draws an arc with center at (x,y) and radius r, enclosed between two rays that form angles angle1 and angle2 with the OX axis

procedure `Arc(x, y, r, angle1, angle2: real; c: Color);`

Draws an arc of a circle centered at (x,y) and with radius r, enclosed between two rays that form angles angle1 and angle2 with

the OX axis, in color c

```
procedure Circle(x,y,r: real);
```

Draws a circle with center at (x,y) and radius r

```
procedure Circle(x,y,r: real; c: Color);
```

Draws a circle with center at (x,y), radius r, and color c

```
procedure Circle(p: Point; r: real);
```

Draws a circle with center at point p and radius r

```
procedure Circle(p: Point; r: real; c: Color);
```

Draws a circle with center at point p, radius r, and color c

```
procedure DrawCircle(x,y,r: real);
```

Draws a circle with center at (x,y) and radius r

```
procedure DrawCircle(x,y,r: real; c: Color);
```

Draws a circle with center at (x,y), radius r, and color c

```
procedure DrawCircle(p: Point; r: real);
```

Draws the contour of a circle with center at point p and radius r

```
procedure DrawCircle(p: Point; r: real; c: Color);
```

Draws a circle with center at p, radius r, and color c

```
procedure DrawEllipse(x,y,rx,ry: real);
```

Draws an ellipse with center at (x,y) and radii rx and ry

```
procedure DrawEllipse(x,y,rx,ry: real; c: Color);
```

Draws an ellipse with center at (x,y), radii rx and ry, and color c

```
procedure DrawEllipse(p: Point; rx,ry: real);
```

Draws an ellipse with center at point p and radii rx and ry

```
procedure DrawEllipse(p: Point; rx,ry: real; c: Color);
```

Draws an ellipse with center at point p, radii rx and ry and color c

```
procedure DrawPixels(x,y: real; pixels: array [,] of Color);
```

Draws a two-dimensional array of pixels starting from the upper left corner with coordinates (x,y)

```
procedure DrawPixels(x,y: real; pixels: array [,] of Color; px,py,pw,ph: integer);
```

Draws a rectangular area (px,py,pw,ph) of a two-dimensional array of pixels starting from the upper left corner with coordinates (x,y)

```
procedure DrawPolygon(points: array of Point);
```

Draws the outline of a polygon defined by an array of points

```
procedure DrawPolygon(points: array of Points; c: GColor);
```

Draws the outline of a polygon defined by an array of points and color

```
procedure DrawRectangle(x,y,w,h: real);
```

Draws the outline of a rectangle with vertex coordinates (x,y) and (x+w,y+h)

```
procedure DrawRectangle(x,y,w,h: real; c: Color);
```

Draws a rectangle outline with vertex coordinates (x,y) and (x+w,y+h) color c `procedure DrawSector(x, y, r, angle1, angle2: real);`

Draws the contour of a sector of a circle centered at (x,y) and with radius r, enclosed between two rays that form angles angle1 and angle2 with the axis OX

`procedure DrawSector(x, y, r, angle1, angle2: real; c: Color);`

Draws the outline of a sector of a circle centered at (x,y) and with radius r, enclosed between two rays that form angles angle1 and angle2 with the OX axis, in color c

`procedure Ellipse(x,y,rx,ry: real);`

Draws an ellipse with center at (x,y) and radii rx and ry `procedure Ellipse(x,y,rx,ry: real; c: Color);`

Draws an ellipse with center at (x,y), radii rx and ry, and color inside c `procedure Ellipse(p: Point; rx,ry: real);`

Draws an ellipse with center at point p and radii rx and ry `procedure Ellipse(p: Point; rx,ry: real; c: Color);`

Draws an ellipse with center at p, radii rx and ry, and interior color c `procedure FillCircle(x,y,r: real);`

Draws the interior of a circle with center at (x,y) and radius r `procedure FillCircle(x,y,r: real; c: Color);`

Draws the interior of a circle with center at (x,y), radius r and color c `procedure FillCircle(p: Point; r: real);`

Draws the interior of a circle with center at point p and radius r `procedure FillCircle(p: Point; r: real; c: Color);`

Draws the interior of a circle with center at point p, radius r and color c `procedure FillEllipse(x,y,rx,ry: real);`

Draws the interior of an ellipse with center at (x,y) and radii rx and ry `procedure FillEllipse(x,y,rx,ry: real; c: Color);`

Draws the interior of an ellipse with center at (x,y), radii rx and ry, and color c `procedure FillEllipse(p: Point; rx,ry: real);`

Draws the interior of an ellipse with center at point p and radii rx and ry `procedure FillEllipse(p: Point; rx,ry: real; c: Color);`

Draws the interior of an ellipse with center at point p, radii rx and

ry, and color c `procedure FillPolygon(points: array of Points);`

Draws the interior of a polygon defined by an array of points

`procedure FillPolygon(points: array of Point; c: GColor);`

Draws the interior of a polygon defined by an array of points and color

`procedure FillRectangle(x,y,w,h: real);`

Draws the interior of a rectangle with vertex coordinates (x,y) and (x+w,y+h)

`procedure FillRectangle(x,y,w,h: real; c: Color);`

Draws the interior of a rectangle with vertex coordinates (x,y) and (x+w,y+h) in color c

`procedure FillSector(x, y, r, angle1, angle2: real);`

Draws the interior of a circle sector with center at (x,y) and radius r, enclosed between two rays that form angles angle1 and angle2 with the axis OX

`procedure FillSector(x, y, r, angle1, angle2: real; c: Color);`

Draws the interior of a sector of a circle centered at (x,y) and with radius r, enclosed between two rays that form angles angle1 and angle2 with the OX axis, color c

`procedure Line(x,y,x1,y1: real);`

Draws a line segment from point (x,y) to point (x1,y1)

`procedure Line(x,y,x1,y1: real; c: Color);`

Draws a line segment from point (x,y) to point (x1,y1) in color c

`procedure Line(p,p1: Point);`

Draws a line segment from p to p1

`procedure Line(p,p1: Point; c: Color);`

Draws a line segment from p to p1 in color c

`procedure LineBy(dx,dy: real);`

Draws a segment from the current position to a point shifted by a vector (dx,dy). The current position is moved to the new point

`procedure LineRel(dx,dy: real);`

Draws a segment from the current position to a point shifted by a vector (dx,dy). The current position is moved to the new point

`procedure Lines(a: array of (Point,Point));`

Draws segments defined by an array of pairs of points

`procedure Lines(a: array of (Point,Point); c: Color);`

Draws segments defined by an array of pairs of points in color c

`procedure LineTo(x,y: real);`

Draws a segment from the current position to the point (x,y). The current position is moved to (x,y)

```
procedure MoveBy(dx,dy: real);
```

Moves the current drawing position to the vector (dx,dy)

```
procedure MoveRel(dx,dy: real);
```

Moves the current drawing position to the vector (dx,dy)

```
procedure MoveTo(x,y: real);
```

Sets the current drawing position to (x,y)

```
procedure Pie(x, y, r, angle1, angle2: real);
```

Draws a sector of a circle centered at (x,y) and with radius r, enclosed between two rays that form angles angle1 and angle2 with the axis OX

```
procedure Polygon(points: array of Point);
```

Draws a polygon defined by an array of points

```
procedure Polygon(points: array of Point; c: Color);
```

Draws a polygon defined by an array of points and color

```
procedure PolyLine(points: array of Point);
```

Draws a polyline defined by an array of points

```
procedure PolyLine(points: array of Point; c: Color);
```

Draws a polyline defined by an array of points and color

```
procedure Rectangle(x,y,w,h: real);
```

Draws a rectangle with vertex coordinates (x,y) and (x+w,y+h)

```
procedure Rectangle(x,y,w,h: real; c: Color);
```

Draws a rectangle with vertex coordinates (x,y) and (x+w,y+h) in color c

```
procedure Sector(x, y, r, angle1, angle2: real);
```

Draws a sector of a circle centered at (x,y) and with radius r, enclosed between two rays that form angles angle1 and angle2 with the axis OX

```
procedure Sector(x, y, r, angle1, angle2: real; c: Color);
```

Draws a sector of a circle centered at (x,y) and with radius r, enclosed between two rays that form angles angle1 and angle2 with the OX axis, in color c

```
procedure SetPixel(x,y: real; c: Color);
```

Draws a pixel at (x,y) with color c

```
procedure SetPixels(x,y: real; w,h: integer; f:
(integer,integer)->Color);
```

Draws a rectangle of pixels of size (w,h) defined by the

mapping f, starting from the upper left corner with coordinates (x,y)

Functions for text output

```
procedure DrawText(x, y, w, h: real; text: string; alignment: Alignment := Alignment.Center; angle: real := 0.0);
```

Outputs a string in a rectangle with the coordinates of the top left corner (x,y) `procedure DrawText(x, y, w, h: real; text: string; c: GColor; align: Alignment := Alignment.Center; angle: real := 0.0);`

Outputs a string in a rectangle with the coordinates of the top left corner (x,y) `procedure DrawText(x, y, w, h: real; number: integer; align: Alignment := Alignment.Center; angle: real := 0.0);`

Outputs an integer into a rectangle with the coordinates of the top left corner (x,y) `procedure DrawText(x, y, w, h: real; number: real; align: Alignment := Alignment.Center; angle: real := 0.0);`

Outputs the real into a rectangle with the coordinates of the upper left corner (x,y)

```
procedure DrawText(r: GRect; text: string; align: Alignment := Alignment.Center; angle: real := 0.0);
```

Outputs a line into a rectangle `procedure DrawText(r: GRect; number: integer; align: Alignment := Alignment.Center; angle: real := 0.0);`

Outputs an integer into a rectangle `procedure DrawText(r: GRect; number: real; align: Alignment := Alignment.Center; angle: real := 0.0);`

Outputs the real into a rectangle `procedure DrawText(x, y, w, h: real; number: integer; c: GColor; alignment: Alignment := Alignment.Center; angle: real := 0.0);`

Outputs an integer into a rectangle with the coordinates of the top left corner (x,y) `procedure DrawText(x, y, w, h: real; number: real; c: GColor; align: Alignment := Alignment.Center; angle: real := 0.0);`

Outputs the real into a rectangle with the coordinates of the top left corner (x,y) `procedure DrawText(r: GRect; text: string; c: GColor; align: Alignment := Alignment.Center; angle: real := 0.0);`

Outputs the string in a rectangle

```
procedure DrawText(r: GRect; number: integer; c: GColor;
```

```
alignment: Alignment := Alignment.Center; angle: real := 0.0);
```

Outputs the integer into a rectangle

```
procedure DrawText(r: GRect; number: real; c: GColor;
alignment: Alignment := Alignment.Center; angle: real := 0.0);
```

Outputs the real into a rectangle `procedure DrawText(x, y, w, h: real; text: string; f: FontOptions; align: Alignment; angle: real);`

Outputs a string in a rectangle with the coordinates of the upper left corner (x,y) in the specified font

```
function TextHeight(text: string): real;
```

Height of text on output

```
function TextHeight(text: string; f: FontOptions): real;
```

Height of text when output with a given font `procedure TextOut(x, y: real; text: string; align: Alignment := Alignment.LeftTop; angle: real := 0.0);`

Outputs a string at position (x,y) `procedure TextOut(x, y: real; text: string; c: GColor; align: Alignment := Alignment.LeftTop; angle: real := 0.0);`

Outputs a string at position (x,y) in color c `procedure TextOut(x, y: real; text: integer; alignment: Alignment := Alignment.LeftTop; angle: real := 0.0);`

Outputs an integer to position (x,y) `procedure TextOut(x, y: real; text: integer; c: GColor; align: Alignment := Alignment.LeftTop; angle: real := 0.0);`

Outputs an integer at position (x,y) in color c `procedure TextOut(x, y: real; text: real; alignment: Alignment := Alignment.LeftTop; angle: real := 0.0);`

Outputs a real to position (x,y) `procedure TextOut(x, y: real; text: real; c: GColor; align: Alignment := Alignment.LeftTop; angle: real := 0.0);`

Outputs the real at position (x,y) in color c `procedure TextOut(x, y: real; text: string; f: FontOptions; align: Alignment := Alignment.LeftTop; angle: real := 0.0);`

Outputs a string at the (x,y) position in the specified font

```
function TextSize(text: string): Size;
```

Text size at output

```
function TextSize(text: string; f: FontOptions): Size;
```

Text size when outputting with the specified font

```
function TextWidth(text: string): real;
```

Width of text on output

```
function TextWidth(text: string; f: FontOptions): real;
```

Width of the text when outputting with the specified font

Functions for graph output

procedure `DrawGraph(f: real -> real; a, b, min, max, x, y, w, h: real; title: string := '');` Draws a graph of a function f, given on the segment [a,b] along the abscissa axis and on the segment [min,max] along the ordinate axis, in a rectangle defined by the parameters x,y,w,h

procedure `DrawGraph(f: real -> real; a, b, min, max, x, y, w, h: real; XTicks: real; YTicks: real; title: string := '');`

Draws a graph of a function f, given on the segment [a,b] on the abscissa axis and on the segment [min,max] on the ordinate axis, in a rectangle defined by the parameters x,y,w,h. The last two parameters set the grid spacing on OX and OY **procedure** `DrawGraph(f: real -> real; a, b, min, max: real; r: GRect; title: string := '');`

Draws a graph of a function f, given on the segment [a,b] on the abscissa axis and on the segment [min,max] on the ordinate axis, in a rectangle r **procedure** `DrawGraph(f: real -> real; a, b, min, max: real; r: GRect; XTicks, YTicks: real; title: string := '');`

Draws a graph of a function f, given on the segment [a,b] on the abscissa axis and on the segment [min,max] on the ordinate axis, in a rectangle r. The last two parameters specify the OX and OY grid spacing of the **procedure** `DrawGraph(f: real -> real; a, b, min, max: real; title: string := '');`

Draws a graph of a function f, given on the segment [a,b] on the abscissa axis and on the segment [min,max] on the ordinate axis, to a full graphics window **procedure** `DrawGraph(f: real -> real; a, b: real; x, y, w, h: real; title: string := '');`

Draws a graph of a function f, given on the interval [a,b], in a rectangle defined by the parameters x,y,w,h **procedure** `DrawGraph(f: real -> real; a, b: real; r: GRect; title: string := '');`

Draws the graph of a function f, given on the interval [a,b], in the rectangle r

procedure `DrawGraph(f: real -> real; r: GRect; title: string := '');`

Draws a graph of a function f, given on the interval [-5,5], in rectangle r **procedure** `DrawGraph(f: real -> real; a, b: real;`

```
title: string := '');
```

Draws a graph of a function f, given on the interval [a,b], on a full graphics window **procedure** `DrawGraph(f: real -> real; title: string := '');`

Draws the graph of the function f, given on the interval [-5,5], on the full graphics window

Functions for image and video output

```
procedure DrawImage(x,y: real; fname: string);
```
Draws an image from file fname at position (x,y)

```
procedure DrawImage(x,y,w,h: real; fname: string);
```
Draws an image from the fname file at the (x,y) position of size w by h

```
procedure DrawImageUnscaled(x,y: real; fname: string);
```
Draws the unscaled image from the fname file at the (x,y) position

```
procedure DrawVideo(x,y: real; fname: string);
```
Outputs video from file fname to position (x,y)

```
function ImageHeight(fname: string): integer;
```
Image height in pixels

```
function ImageSize(fname: string): (integer,integer);
```
Image size in pixels

```
function ImageWidth(fname: string): integer;
```
Image width in pixels

Functions for setting the coordinate system

procedure DrawGrid; Draws the coordinate system grid **procedure** SetMathematicCoords(x1: real := -10; x2: real := 10; drawgrid: boolean := true);

Sets the mathematical coordinate system with the range [x1,x2] on the OX axis.

procedure SetMathematicCoords(x1,x2,ymin: real; drawgrid: boolean := true);

Sets a mathematical coordinate system with range [x1,x2] on OX axis and minimum ymin coordinate on OY axis **procedure** SetStandardCoords(scale: real := 1.0; x0: real := 0; y0: real := 0);

Sets a standard coordinate system (OY axis pointing down) with center (x0,y0) **procedure** SetStandardCoordsSharpLines(x0: real := 0; y0: real := 0);

Sets a standard coordinate system with center (x0,y0). The image is not scaled to the monitor resolution and the lines are sharp

function XMax: real;

Maximum displayed x-coordinate **function** XMin: real;

Minimum displayed x-coordinate **function** YMax: real;

The maximum displayed y-coordinate **of the function** YMin: real;

Minimum displayed y-coordinate

Frame-based animation procedures `procedure BeginFrameBasedAnimation(Draw: procedure; frate: integer := 61);` Starts frame-based animation. Before drawing each frame, the contents of the window are erased, then the Draw `procedure BeginFrameBasedAnimation(Draw: procedure(frame: integer); frate: integer := 61)` is called;

Starts a frame-based animation. Before drawing each frame the contents of the window are erased, then the `procedure BeginFrameBasedAnimationTime(Draw: procedure(dt: real))` is called with a parameter equal to the frame number;

Starts a frame-based animation and passes to each frame handler the time dt that has elapsed since the last redraw `procedure EndFrameBasedAnimation;`

Completes the animation based on the frame

Auxiliary functions GraphWPF

```
function ARGB(a,r,g,b: byte): Color;
```

Returns color by red, green and blue component and transparency parameter (in range 0..255)

```
function clRandom: Color;
```

Returns a random color

```
function ColorBrush(c: Color): GBrush;
```

Returns a single-color brush, specified by color

```
function ColorPen(c: Color): GPen;
```

Returns a single-color pen specified by the color

```
function EmptyColor: Color;
```

Returns fully transparent color

```
function GrayColor(b: byte): Color;
```

Returns the gray color with intensity b

```
function Pnt(x,y: real): GPoint;
```

Returns a point with coordinates (x,y)

```
function RandomColor: Color;
```

Returns a random color

```
function RandomPoint(w: real := 0): Point;
```

The function of generating a random point in the boundaries of the screen. The optional parameter w specifies the minimum offset from the border

```
function RandomPoints(n: integer; w: real := 0): array of Point;
```

The function generates an array of random points in the boundaries of the screen. The optional parameter w specifies the minimum offset from the border

```
function Rect(x,y,w,h: real): GRect;
```

Returns a rectangle with corner coordinates (x,y), width w and height h

```
procedure Redraw(d: ()->());
```

A procedure to speed up the output. Refreshes the screen after all changes

```
function RGB(r,g,b: byte): Color;
```

Returns the color in the red, green and blue components (in the range 0...255)

```
function Vect(vx,vy: real): Vector;
```

Creates a vector with coordinates vx,vy

GraphWPF module types

```
Alignment =
(LeftTop,CenterTop,RightTop,LeftCenter,Center,RightCenter,LeftBottom,CenterBottom,RightBottom);
```
Text alignment constants relative to the point

```
Color = System.Windows.Media.Color;
```
Color type

```
Colors = System.Windows.Media.Colors;
```
Color constants

```
CoordType = (MathematicalCoords,StandardCoords);
```
Types of coordinate system

```
Fontstyle = (Normal,Bold,Italic,BoldItalic);
```
Font style type

```
GBrush = System.Windows.Media.Brush;
```
Brush type

```
GColor = System.Windows.Media.Color;
```
Color type

```
GPoint = System.Windows.Point;
```
Point type

```
GRect = System.Windows.Rect;
```
Rectangle type

```
GWindow = System.Windows.Window;
```
Window type

```
Key = System.Windows.Input.Key;
```
Key Type

```
Point = System.Windows.Point;
```
Point type

```
Vector = System.Windows.Vector;
```
Vector type

GraphWPF module variables

`Brush: BrushType;` Current Brush

`Pen: PenType;`

Current Feather

`Font: FontOptions;`

Current font

`Window: WindowTypeWPF;`

Main window

`GraphWindow: GraphWindowType;`

Graphic window

Events of the GraphWPF module

```
OnMouseDown: procedure(x, y: real; mousebutton: integer);
```

The event of clicking the mouse button. (x,y) - coordinates of the mouse cursor when the event occurs, mousebutton = 1 if the left mouse button is pressed, and 2 if the right mouse button is pressed

```
OnMouseUp: procedure(x, y: real; mousebutton: integer);
```

Mouse button release event. (x,y) - coordinates of the mouse cursor when the event occurs, mousebutton = 1 if the left mouse button is pressed, and 2 if the right mouse button is pressed

```
OnMouseMove: procedure(x, y: real; mousebutton: integer);
```

Mouse move event. (x,y) - coordinates of the mouse cursor when the event occurs, mousebutton = 0 if the mouse button is not pressed, 1 if the left mouse button is pressed, and 2 if the right mouse button is pressed

```
OnKeyDown: procedure(k: Key);
```

Key press event

```
OnKeyUp: procedure(k: Key);
```

Key release event

```
OnKeyPress: procedure(ch: char);
```

Symbol key press event

```
OnResize: procedure;
```

Event of changing the size of the graphics window

```
OnClose: procedure;
```

Event that occurs when the main window is closed

```
OnDrawFrame: procedure(dt: real);
```

The event of redrawing the graphic window. The dt parameter denotes the number of milliseconds since the last call of OnDrawFrame

BrushType class

```
/// Brush type; 6 /// Brush color;
```

shType = class

PenType class

Pen type.

PenType class `property Color: GCoior;` Pen color `property RoundCap: boolean;`

Rounding of the pen at the ends of the lines `property Width: real;`

Pen width `X: real;`

The current X coordinate of the pen `property Y: real;`

The current Y coordinate of the pen

FontType class

Font type.

FontOptions class `property Color: GColor;` Font color

`property Name: string;`

Font name

`property Size: real;`

Font size in units of 1/96 inch

`property Style: FontStyle;`

Font style

FontOptions class methods

`function WithColor(c: GColor): FontOptions;`

Font color decorator

`function WithName(name: string): FontOptions;`

Font style decorator

`function WithSize(sz: real): FontOptions;`

Font size decorator

`function WithStyle(fs: FontStyle): FontOptions;`

Font style decorator

GraphWindowType class

Type of graphic window.

GraphWindowType class properties

`property Height: real;` the height of the graphics window `property Left: real;`

The indent of the graphic window from the left edge of the main window `property Top: real;`

Indent of the graphics window from the upper edge of the main window

`property Width: real;`

Graphic window width

GraphWindowType class methods

`function Center: Point;`

Center of the graphics window

`function ClientRect: GRect;`

Graphic window client rectangle

`procedure Fill(fname: string);`

Fills the contents of the graphics window with the wallpaper from
of a file named fname

`procedure Load(fname: string);`

Restores the contents of the graphics window from a file named fname `procedure Save(fname: string);`

Saves the contents of the graphics window to a file with
by the name fname

WindowTypeWPF class

WindowTypeWPF is a type of graphical window. The base class is `WindowType`.

Methods of the WindowTypeWPF class

`procedure Clear(c: Color); override;` Clears the graphic window with color c

`procedure Clear; override;`

Clears the graphic window in white

`procedure Load(fname: string);`

Restores the contents of the graphics window from a file named fname

`procedure Save(fname: string);`

Saves the contents of the graphics window to a file named fname

Properties of the WindowType base class

`property Caption: string;`

Window title

`property Height: real;`

Height of the client part of the main window

`property IsFixedSize: boolean;`

Does the graphics window have a fixed size

`property Left: real;`

Main window indent from the left edge of the screen

`property Title: string;`

Window title

`Property Top: real;`

Main window indent from the top edge of the screen

`property Width: real;`

Width of the client part of the main window

Methods of the WindowType base class

`function Center: Point;`

Returns the center of the main window

`function ClientRect: GRect;`

Returns the rectangle of the client area of the window

```
function RandomPoint(w: real := 0): Point;
```

Returns a random point in the boundary of the screen. The optional parameter w specifies the minimum offset from the boundary

```
procedure CenterOnScreen;
```

Centers the main window in the center of the screen

```
procedure Clear(c: GColor); virtual;
```

Clears the graphic window with the specified color

```
procedure Clear; virtual;
```

Clears the graphic window in white

```
procedure Close;
```

Closes the main window and ends the application

```
procedure Maximize;
```

Maximizes the main window

```
procedure Minimize;
```

Minimizes the main window

```
procedure Normalize;
```

Returns the main window to its normal size

```
procedure SetPos(l, t: real);
```

Sets the indent of the main window from the upper left edge of the screen

```
procedure SetSize(w, h: real);
```

Sets the size of the client part of the main window

WindowType class

Type of main application window.

WindowType class `property Caption: string;` Window title

`property Height: real;`

The height of the client part of the main window `property IsFixedSize: boolean;`

Whether the graphic window has a fixed size `property Left: real;`

The indent of the main window from the left edge of the screen `property Title: string;`

Window title

`Property Top: real;`

The indent of the main window from the top edge of the screen `property Width: real;`

Width of the client part of the main window

WindowType class methods

`function Center: Point;`

Returns the center of the main window `function ClientRect: GRect;`

Returns the rectangle of the client window area `function RandomPoint(w: real := 0): Point;`

Returns a random point in the boundary of the screen. The optional parameter w specifies the minimum offset from the border of the `CenterOnScreen procedure;`

Centers the main window in the center of the screen `procedure Clear(c: GColor); virtual;`

Clears the graphic window with the specified color `procedure Clear; virtual;`

Clears the graphic window with the white color `procedure Close;`

Closes the main window and terminates the `Maximize`

procedure application;

Maximizes the main window

procedure Minimize;

Minimizes the main window

procedure Normalize;

Returns the main window to its normal size **procedure** SetPos(l, t: real);

Sets the offset of the main window from the upper left edge of the screen **procedure** SetSize(w, h: real);

Sets the size of the client part of the main window

WPFObj^cts module: overview

The `wPFobjects` module implements vector graphics objects with the ability to overlap each other, create compound graphics objects and nest them repeatedly in each other. Each vector graphic object redraws itself correctly when it is moved, resized and partially overlapped by other objects.

`WPFobjects` module is designed for early learning the basics of object-oriented programming, as well as for implementing animation and interactive projects of medium complexity. The `WPFobjects` module is based on the WPF graphical library and is a modern and improved version of the outdated `ABcobjects` module.

The `WPFobjects` module defines a number of constants, types, procedures, functions and classes. They are divided into the following groups:

- typesдmoduleWPFObjects
- variablesдmoduleWPFObjectsjects
- Events дmoduleWPFObjects
- classObjectWPF
- Class BoundedObjectWPF
- classEllipseWPF
- classCircleWPF
- classRectangleWPFgleWPF
- classSquareWPF
- classRoundRectWPF
- classRoundSquareWPF
- classLineWPF
- classRegRegularPolygonWPFygonWPF
- classStarWPF
- classPolPolygonWPFonWPF
- classPictureWPF
- ΦIntersection functions
- Graphic window class
- The class of the list of graphical objects
- auxiliary functionsWPFObjects

WPFObj^cts module types

```
Color = System.Windows.Media.Color;
```
Type
```
Colors = System.Windows.Media.Colors;
```

Color constants

```
Fontstyle = (Normal,Bold,Italic,BoldItalic);
```

Font style type

```
GBrush = System.Windows.Media.Brush;
```

Brush type

```
GColor = System.Windows.Media.Color;
```

Color type

```
GPen = System.Windows.Media.Pen;
```

Pen type

```
GPoint = System.Windows.Point;
```

Point type

```
GRect = System.Windows.Rect;
```

Rectangle type

```
GSize = System.Windows.Size;
```

Size type

```
GWindow = System.Windows.Window;
```

Window type

```
Key = System.Windows.Input.Key;
```

Key Type

```
Point = System.Windows.Point;
```

Point type

WPFObj^cts module variables

`Window: WindowType;` Main Window `Graphwindow: GraphWindowType;`

Graphic window

`Objects: ObjectsType;`

List of graphic objects

WPFObj^cts events

```
OnClose: procedure;
```
Event occurring at closing main window

```
OnDrawFrame: procedure(dt: real);
```
The event of redrawing the graphic window. The dt parameter denotes the number of milliseconds since the last call of OnDrawFrame

```
OnKeyDown: procedure(k: Key);
```
Key press event

```
OnKeyPress: procedure(ch: char);
```
Symbol key press event

```
OnKeyUp: procedure(k: Key);
```
Key release event

```
OnMouseDown: procedure(x, y: real; mousebutton: integer);
```
The event of clicking the mouse button. (x,y) - coordinates of the mouse cursor when the event occurs, mousebutton = 1 if the left mouse button is pressed, and 2 if the right mouse button is pressed

```
OnMouseMove: procedure(x, y: real; mousebutton: integer);
```
Mouse move event. (x,y) - coordinates of the mouse cursor when the event occurs, mousebutton = 0 if the mouse button is not pressed, 1 if the left mouse button is pressed, and 2 if the right mouse button is pressed

```
OnMouseUp: procedure(x, y: real; mousebutton: integer);
```
Mouse button release event. (x,y) - coordinates of the mouse cursor when the event occurs, mousebutton = 1 if the left mouse button is pressed, and 2 if the right mouse button is pressed

```
OnResize: procedure;
```
Event of changing the size of the graphics window

Class Ob jectWPF

A basic class of graphical objects.

ObjectWPF class properties

```
property Bottom: real;
```
The offset of the bottom of the graphical object from the top edge of the window

```
property Bounds: GRect;
```
Rectangle of a graphical object

```
Point: Point;
```

The center of a graphical object

```
property CenterBottom: Point;
```

The central bottom point of the graphical object

```
property CenterTop: Point;
```

The central upper point of the graphical object

```
property Color: GColor;
```

The color of the graphical object

```
property Direction: (real,real);
```

Direction of movement. Used with the Move method

```
property FontColor: Color;
```

The color of the text font inside the graphical object

```
property FontName: string;
```

The name of the text font inside the graphical object

```
property FontSize: real;
```

The font size of the text inside the graphical object

```
property Height: real;
```

The height of the graphical object

```
property Left: real;
```

Indent the graphical object from the left window edge

```
property LeftBottom: Point;
```

Lower left corner of the graphical object

```
property LeftTop: Point;
```

Left top corner of the graphical object

```
property Number: integer;
```

An integer number displayed in the center of the graphical object. The Text property is used

```
property RealNumber: real;
```

A real number displayed in the center of the graphical object. The Text property is used

```
Property Right: real;
```

Indent the right edge of the graphical object from the left edge of the window

```
property RightBottom: Point;
```

Lower right corner of the graphical object

```
property RightTop: Point;
```

Top right corner of the graphical object

```
property RotateAngle: real;
```

Angle of rotation of the graphical object (clockwise)

```
property ScaledHeight: real;
```
Scaled height of the graphical object
```
property ScaledSize: GSize;
```
The scaled size of the graphical object
```
property ScaledWidth: real;
```
Scaled width of the graphical object
```
property ScaleFactor: real;
```
Scaling multiplier of the object
```
property Size: GSize;
```
The size of the graphical object
```
property Text: string;
```
Text inside the graphical object `property TextAlignment: Alignment;`

Text alignment inside the graphical object `property Top: real;`

Indent the graphical object from the top edge of the window
```
property Visible: boolean;
```
Visibility of the graphical object `property Width: real;`

Width of the graphical object

ObjectWPF class methods

```
procedure AnimMoveBy(a,b: real; sec: real := 1);
```
Animates the movement of the graphical object on the vector (a,b) for sec seconds `procedure Move; virtual;`

Moves the graphical object to the vector (dx,dy) `procedure MoveBy(a,b: real);`

Moves the graphical object to the vector (a,b) `procedure MoveBy(v: (real,real));`

Moves the graphical object to the vector (a,b) `procedure MoveForward(r: real);`

Moves the graphical object in the RotateAngle direction (up when RotateAngle=0) `procedure MoveTime(dt: real); virtual;`

Moves the graphical object along the Direction vector with velocity Velocity in time dt
```
procedure MoveTo(x,y: real);
```
Moves the upper left corner of the graphical object to the point (x,y) `procedure Rotate(a: real);`

Rotates the graphical object clockwise by the angle a

```
procedure RotateToPoint(x,y: real);
```

Rotates the graphical object so that it "looks" at the point (x,y)

```
procedure Scale(r: real);
```

Scales a graphical object to r times its current size

The class of graphical objects with a border

A class of graphical objects with a boundary. The base class is `ObjectWPF`.

Properties of the BoundedObjectWPF class

`property BorderColor: GColor;` The color of the border of the graphical object

`property BorderWidth: real;`

Width of the graphical object border

`property Color: GColor;`

The color of the graphical object

Methods of the BoundedObjectWPF class

`function RemoveBorder: BoundedObjectWPF;`

Decorator for turning off the object boundary

`function SetBorder(w: real := 1; c: GColor := Colors.Black): BoundedObjectWPF;`

Decorator for turning on the object boundary

Properties of the ObjectWPF base class

`property Bottom: real;`

The indent of the bottom of the graphical object from the top edge of the window

`property Bounds: GRect;`

Rectangle of a graphical object

`Point: Point;`

The center of a graphical object

`property CenterBottom: Point;`

The central bottom point of the graphical object

`property CenterTop: Point;`

The central upper point of the graphical object

`property Color: GColor;`

The color of the graphical object

`property Direction: (real,real);`

Direction of movement. Used with the Move method

`property FontColor: Color;`

The color of the text font inside the graphical object `property`

```
FontName: string;
```
The name of the text font inside the graphical object **property**
```
FontSize: real;
```
The font size of the text inside the graphical object **property**
```
Height: real;
```
The height of the graphical object
property `Left: real;`
The indent of the graphical object from the left edge of the window **property** `LeftBottom: Point;`
Lower left corner of the graphical object
property `LeftTop: Point;`
Left top corner of the graphical object
property `Number: integer;`
An integer number displayed in the center of the graphical object. The Text **property** `RealNumber: real` is used;
A real number displayed in the center of the graphical object. The Text **property** `Right: real` is used;
Indent the right edge of the graphical object from the left edge of the window
property `RightBottom: Point;`
Lower right corner of the graphical object
property `RightTop: Point;`
Right top corner of the graphical object **property** `RotateAngle: real;`
Angle of rotation of the graphical object (clockwise) **property**
```
ScaledHeight: real;
```
The scaled height of the graphical object **property**
```
ScaledSize: GSize;
```
The scaled size of the graphical object **property** `ScaledWidth: real;`
Scaled width of the graphical object
property `ScaleFactor: real;`
Scaling multiplier of the object
property `Size: GSize;`
The size of the graphical object
property `Text: string;`

Text inside a graphical object

```
property TextAlignment: Alignment;
```

Text alignment inside a graphical object

```
Property Top: real;
```

Indent the graphical object from the top edge of the window

```
property Visible: boolean;
```

Visibility of a graphical object

```
property Width: real;
```

Width of the graphical object

ObjectWPF base class methods

```
procedure AnimMoveBy(a,b: real; sec: real := 1);
```

Animates the movement of the graphical object on the vector (a,b) for sec seconds

```
procedure Move; virtual;
```

Moves the graphical object to the vector (dx,dy)

```
procedure MoveBy(a,b: real);
```

Moves the graphical object to vector (a,b)

```
procedure MoveBy(v: (real,real));
```

Moves the graphical object to vector (a,b)

```
procedure MoveForward(r: real);
```

Moves the graphical object in the direction of RotateAngle (up when RotateAngle=0)

```
procedure MoveTime(dt: real); virtual;
```

Moves the graphical object along the Direction vector with velocity Velocity in time dt

```
procedure MoveTo(x,y: real);
```

Moves the upper left corner of the graphical object to the (x,y) point

```
procedure Rotate(a: real);
```

Rotates the graphical object clockwise by the angle a

```
procedure RotateToPoint(x,y: real);
```

Rotates the graphical object so that it "looks" at the point (x,y)

```
procedure Scale(r: real);
```

Scales a graphical object to r times its current size

EllipseWPF class

The class of graphical objects "Ellipse". The base class is `BoundedObjectWPF`.

EllipseWPF class constructors

`constructor` `(x,y,rx,ry: real; c: GColor);` Creates an ellipse with center at point (x,y), radii (rx,ry) and interior color with **`constructor`** `(x,y,rx,ry: real; c: GColor; borderwidth: real; borderColor: GColor := Colors.Black);`

Creates an ellipse with center at point (x,y), radii (rx,ry) and interior color c, with borderWidth and borderColor

`constructor` `(p: Point; rx,ry: real; c: GColor);`

Creates an ellipse with center at point p, radii (rx,ry) and interior color with **`constructor`** `(p: Point; rx,ry: real; c: GColor; borderWidth: real; borderColor: GColor := Colors.Black);`

Creates an ellipse with center at point p, radii (rx,ry) and interior color c, with borderWidth and borderColor

Properties of the EllipseWPF class

`property` `RadiusX: real;`

The radius of the ellipse along the OX axis `RadiusY: real;`

Radius of the ellipse along the OY axis

EllipseWPF class methods

`function` `RemoveBorder;`

Decorator for turning off the object boundary

`function` `SetBorder(w: real := 1; c: GColor := Colors.Black);`

Decorator for turning on the object boundary

`function` `SetRotate(da: real): EllipseWPF;`

Object rotation decorator

`function` `SetText(txt: string; size: real := 16; fontname: string := 'Arial'; c: GColor := Colors.Black): EllipseWPF;`

Object text decorator

Properties of the BoundedObjectWPF base class

`property` `BorderColor: GColor;`

The color of the border of the graphical object

```
property BorderWidth: real;
```

Width of the graphical object border

```
property Color: GColor;
```

The color of the graphical object

Methods of the BoundedObjectWPF base class

```
function RemoveBorder: BoundedObjectWPF;
```

Decorator for turning off the object boundary

```
function SetBorder(w: real := 1; c: GColor := Colors.Black):
BoundedObjectWPF;
```

Decorator for turning on the object boundary

Properties of the ObjectWPF base class

```
property Bottom: real;
```

The indent of the bottom of the graphical object from the top edge of the window

```
property Bounds: GRect;
```

Rectangle of a graphical object

```
Point: Point;
```

The center of a graphical object

```
property CenterBottom: Point;
```

The central bottom point of the graphical object

```
property CenterTop: Point;
```

The central upper point of the graphical object

```
property Color: GColor;
```

The color of the graphical object

```
property Direction: (real,real);
```

Direction of movement. Used with the Move method

```
property FontColor: Color;
```

The color of the text font inside the graphical object

```
property FontName: string;
```

The name of the text font inside the graphical object

```
property FontSize: real;
```

The font size of the text inside the graphical object

```
property Height: real;
```

The height of the graphical object

```
property Left: real;
```

Indent the graphical object from the left window edge

```
property LeftBottom: Point;
```
Lower left corner of the graphical object
```
property LeftTop: Point;
```
Left top corner of the graphical object
```
property Number: integer;
```
An integer number displayed in the center of the graphical object. The Text property is used
```
property RealNumber: real;
```
A real number displayed in the center of the graphical object. The Text property is used
```
Property Right: real;
```
Indent the right edge of the graphical object from the left edge of the window
```
property RightBottom: Point;
```
Lower right corner of the graphical object
```
property RightTop: Point;
```
Top right corner of the graphical object
```
property RotateAngle: real;
```
Angle of rotation of the graphical object (clockwise)
```
property ScaledHeight: real;
```
Scaled height of the graphical object
```
property ScaledSize: GSize;
```
The scaled size of the graphical object
```
property ScaledWidth: real;
```
Scaled width of the graphical object
```
property ScaleFactor: real;
```
Scaling multiplier of the object
```
property Size: GSize;
```
The size of the graphical object
```
property Text: string;
```
Text inside a graphical object
```
property TextAlignment: Alignment;
```
Text alignment inside a graphical object
```
Property Top: real;
```
Indent the graphical object from the top edge of the window
```
property Visible: boolean;
```
Visibility of a graphical object
```
property Width: real;
```
Width of the graphical object

ObjectWPF base class methods

```
procedure AnimMoveBy(a,b: real; sec: real := 1);
```

Animates the movement of the graphical object on the vector (a,b) for sec seconds

```
procedure Move; virtual;
```

Moves the graphical object to the vector (dx,dy)

```
procedure MoveBy(a,b: real);
```

Moves the graphical object to vector (a,b)

```
procedure MoveBy(v: (real,real));
```

Moves the graphical object to vector (a,b)

```
procedure MoveForward(r: real);
```

Moves the graphical object in the direction of RotateAngle (up when RotateAngle=0)

```
procedure MoveTime(dt: real); virtual;
```

Moves the graphical object along the Direction vector with velocity Velocity in time dt

```
procedure MoveTo(x,y: real);
```

Moves the upper left corner of the graphical object to the (x,y) point

```
procedure Rotate(a: real);
```

Rotates the graphical object clockwise by angle a

```
procedure RotateToPoint(x,y: real);
```

Rotates the graphical object so that it "looks" at the point (x,y)

```
procedure Scale(r: real);
```

Scales a graphical object to r times its current size

CircleWPF class

The class of graphical objects "Circle". The base class is `BoundedObjectWPF`.

CircleWPF class constructors

`constructor (x,y,r: real; c: GColor);` Creates a circle of radius r of a given color with coordinates of the center (x,y)

`constructor (p: Point; r: real; c: GColor);`

Creates a circle of radius r of a given color with center p

`constructor (x,y,r: real; c: GColor; borderwidth: real; borderColor: GColor := Colors.Black);`

Creates a circle of radius r of a given color with center coordinates (x,y), borderWidth and borderColor `constructor (p: Point; r: real; c: GColor; borderWidth: real; borderColor: GColor := Colors.Black);`

Creates a circle of radius r of a given color with center p, borderWidth and borderColor

CircleWPF class properties

`property Height: real;`

Circle height

`property Radius: real;`

Circle radius

`property Width: real;`

Circle width

CircleWPF class methods

`function RemoveBorder;`

Decorator for turning off the object boundary

`function SetBorder(w: real := 1; c: GColor := Colors.Black);`

Decorator for turning on the object boundary

`function SetRotate(da: real): CircleWPF;`

Object rotation decorator

`function SetText(txt: string; size: real := 16; fontname: string := 'Arial'; c: GColor := Colors.Black): CircleWPF;`

Object text decorator

Properties of the BoundedObjectWPF base class

```
property BorderColor: GColor;
```

The color of the border of the graphical object

```
property BorderWidth: real;
```

Width of the graphical object border

```
property Color: GColor;
```

The color of the graphical object

Methods of the BoundedObjectWPF base class

```
function RemoveBorder: BoundedObjectWPF;
```

Decorator for turning off the object boundary

```
function SetBorder(w: real := 1; c: GColor := Colors.Black):
BoundedObjectWPF;
```

Decorator for turning on the object boundary

Properties of the ObjectWPF base class

```
property Bottom: real;
```

The indent of the bottom of the graphical object from the top edge of the window

```
property Bounds: GRect;
```

Rectangle of a graphical object

```
Point: Point;
```

The center of a graphical object

```
property CenterBottom: Point;
```

The central bottom point of the graphical object

```
property CenterTop: Point;
```

The central upper point of the graphical object

```
property Color: GColor;
```

The color of the graphical object

```
property Direction: (real,real);
```

Direction of movement. Used with the Move method

```
property FontColor: Color;
```

The color of the text font inside the graphical object

```
property FontName: string;
```

The name of the text font inside the graphical object

```
property FontSize: real;
```

The font size of the text inside the graphical object

```
property Height: real;
```
The height of the graphical object
```
property Left: real;
```
Indent the graphical object from the left window edge
```
property LeftBottom: Point;
```
Lower left corner of the graphical object
```
property LeftTop: Point;
```
Left top corner of the graphical object
```
property Number: integer;
```
An integer number displayed in the center of the graphical object. The Text property is used
```
property RealNumber: real;
```
A real number displayed in the center of the graphical object. The Text property is used
```
Property Right: real;
```
Indent the right edge of the graphical object from the left edge of the window
```
property RightBottom: Point;
```
Lower right corner of the graphical object
```
property RightTop: Point;
```
Top right corner of the graphical object
```
property RotateAngle: real;
```
Angle of rotation of the graphical object (clockwise)
```
property ScaledHeight: real;
```
Scaled height of the graphical object
```
property ScaledSize: GSize;
```
The scaled size of the graphical object
```
property ScaledWidth: real;
```
Scaled width of the graphical object
```
property ScaleFactor: real;
```
Scaling multiplier of the object
```
property Size: GSize;
```
The size of the graphical object
```
property Text: string;
```
Text inside a graphical object
```
property TextAlignment: Alignment;
```
Text alignment inside a graphical object
```
Property Top: real;
```
Indent the graphical object from the top edge of the window
```
property Visible: boolean;
```
Visibility of a graphical object
```
property Width: real;
```

Width of the graphical object

ObjectWPF base class methods

```
procedure AnimMoveBy(a,b: real; sec: real := 1);
```

Animates the movement of the graphical object on the vector (a,b) for sec seconds

```
procedure Move; virtual;
```

Moves the graphical object to the vector (dx,dy)

```
procedure MoveBy(a,b: real);
```

Moves the graphical object to vector (a,b)

```
procedure MoveBy(v: (real,real));
```

Moves the graphical object to vector (a,b)

```
procedure MoveForward(r: real);
```

Moves the graphical object in the direction of RotateAngle (up when RotateAngle=0)

```
procedure MoveTime(dt: real); virtual;
```

Moves the graphical object along the Direction vector with velocity Velocity in time dt

```
procedure MoveTo(x,y: real);
```

Moves the upper left corner of the graphical object to the (x,y) point

```
procedure Rotate(a: real);
```

Rotates the graphical object clockwise by an angle a

```
procedure RotateToPoint(x,y: real);
```

Rotates the graphical object so that it "looks" at the point (x,y)

```
procedure Scale(r: real);
```

Scales a graphical object to r times its current size

RectangleWPF class

Rectangle" class of graphical objects. The base class is `BoundedObjectWPF`.

RectangleWPF class `constructor (x,y,w,h: real; c: GColor);` Creates a rectangle of size (w,h) of a given color with the coordinates of the upper left corner (x,y)

`constructor (p: Point; w,h: real; c: GColor);`

Creates a rectangle of size (w,h) of a given color with the coordinates of the upper left corner set by p `constructor (x,y,w,h: real; c: GColor; borderwidth: real; borderColor: GColor := Colors.Black);`

Creates a rectangle of size (w,h) of a given color with coordinates of the upper left corner (x,y), borderWidth and borderColor `constructor (p: Point; w,h: real; c: GColor; borderWidth: real; borderColor: GColor := Colors.Black);`

Creates a rectangle of size (w,h) of a given color with the coordinates of the upper left corner set by p, with borderWidth and borderColor

RectangleWPF class methods

`function RemoveBorder;`

Decorator for turning off the object boundary

`function SetBorder(w: real := 1; c: GColor := Colors.Black);`

Decorator to turn on the object boundary

`function SetRotate(da: real): RectangleWPF;`

Object rotation decorator `function SetText(txt: string; size: real := 16; fontname: string := 'Arial'; c: GColor := Colors.Black): RectangleWPF;`

Object text decorator

Properties of the BoundedObjectWPF base class

`property BorderColor: GColor;`

The color of the border of the graphical object

`property BorderWidth: real;`

Width of the graphical object border

`property Color: GColor;`

The color of the graphical object

Methods of the BoundedObjectWPF base class

```
function RemoveBorder: BoundedObjectWPF;
```

Decorator for turning off the object boundary

```
function SetBorder(w: real := 1; c: GColor := Colors.Black):
BoundedObjectWPF;
```

Decorator to turn on the object boundary

Properties of the ObjectWPF base class

```
property Bottom: real;
```

The indent of the bottom of the graphical object from the top edge of the window

```
property Bounds: GRect;
```

Rectangle of a graphical object

```
Point: Point;
```

The center of a graphical object

```
property CenterBottom: Point;
```

The central bottom point of the graphical object

```
property CenterTop: Point;
```

The central upper point of the graphical object

```
property Color: GColor;
```

The color of the graphical object

```
property Direction: (real,real);
```

Direction of movement. Used with the Move method

```
property FontColor: Color;
```

The color of the text font inside the graphical object

```
property FontName: string;
```

The name of the text font inside the graphical object

```
property FontSize: real;
```

The font size of the text inside the graphical object

```
property Height: real;
```

The height of the graphical object

```
property Left: real;
```

Indent the graphical object from the left window edge

```
property LeftBottom: Point;
```

Lower left corner of the graphical object

```
property LeftTop: Point;
```

Left top corner of the graphical object

```
property Number: integer;
```

An integer number displayed in the center of the graphical object. The Text property is used

```
property RealNumber: real;
```

A real number displayed in the center of the graphical object. The Text property is used

```
Property Right: real;
```

Indent the right edge of the graphical object from the left edge of the window

```
property RightBottom: Point;
```

Lower right corner of the graphical object

```
property RightTop: Point;
```

Top right corner of the graphical object

```
property RotateAngle: real;
```

Angle of rotation of the graphical object (clockwise)

```
property ScaledHeight: real;
```

Scaled height of the graphical object

```
property ScaledSize: GSize;
```

The scaled size of the graphical object

```
property ScaledWidth: real;
```

Scaled width of the graphical object

```
property ScaleFactor: real;
```

Scaling multiplier of the object

```
property Size: GSize;
```

The size of the graphical object

```
property Text: string;
```

Text inside a graphical object

```
property TextAlignment: Alignment;
```

Text alignment inside the graphical object `property Top: real;`

Indent the graphical object from the top edge of the window

```
property Visible: boolean;
```

Visibility of a graphical object

```
property Width: real;
```

Width of the graphical object

ObjectWPF base class methods

```
procedure AnimMoveBy(a,b: real; sec: real := 1);
```

Animates the movement of the graphical object on the vector

(a,b) for sec seconds `procedure Move; virtual;`

Moves the graphical object to the vector (dx,dy) `procedure MoveBy(a,b: real);`

Moves the graphical object to the vector (a,b) `procedure MoveBy(v: (real,real));`

Moves the graphical object to the vector (a,b) `procedure MoveForward(r: real);`

Moves the graphical object in the RotateAngle direction (up when RotateAngle=0) `procedure MoveTime(dt: real); virtual;`

Moves the graphical object along the Direction vector with velocity Velocity in time dt

`procedure MoveTo(x,y: real);`

Moves the upper left corner of the graphical object to the (x,y) point

`procedure Rotate(a: real);`

Rotates the graphical object clockwise by the angle a

`procedure RotateToPoint(x,y: real);`

Rotates the graphical object so that it "looks" at the point (x,y)

`procedure Scale(r: real);`

Scales a graphical object by a factor of three relative to its current size

SquareWPF class

The class of graphical objects "Square". The base class is `CircleWPF`.

Constructors class SquareWPF

`constructor` `(x,y,w: real; c: GColor);` Creates a square with side w of specified color with coordinates of the upper left corner (x,y)

`constructor` `(p: Point; w: real; c: GColor);`

Creates a square with side w of a given color with the coordinates of the upper left corner set by p

`constructor` `(x,y,w: real; c: GColor; borderwidth: real; borderColor: GColor := Colors.Black);`

Creates a square with side w of specified color with coordinates of the upper left corner (x,y), borderWidth and borderColor

`constructor` `(p: Point; w: real; c: GColor; borderWidth: real; borderColor: GColor := Colors.Black);`

Creates a square with side w of a given color with the coordinates of the upper left corner set by p, with borderWidth and borderColor

Methods of the SquareWPF class

`function` `RemoveBorder;`

Decorator for turning off the object boundary

`function` `SetBorder(w: real := 1; c: GColor := Colors.Black);`

Decorator to turn on the object boundary

`function` `SetRotate(da: real): SquareWPF;`

Object rotation decorator

`function` `SetText(txt: string; size: real := 16; fontname: string := 'Arial'; c: GColor := Colors.Black): SquareWPF;`

Object text decorator

CircleWPF base class properties

`property` `Height: real;`

Circle height

`property` `Radius: real;`

Circle radius

```
property Width: real;
```

Circle width

CircleWPF base class methods

```
function RemoveBorder;
```

Decorator for turning off the object boundary

```
function SetBorder(w: real := 1; c: GColor := Colors.Black);
```

Decorator to turn on the object boundary

```
function SetRotate(da: real): CircleWPF;
```

Object rotation decorator

```
function SetText(txt: string; size: real := 16; fontname:
string := 'Arial'; c: GColor := Colors.Black): CircleWPF;
```

Object text decorator

Properties of the BoundedObjectWPF base class

```
property BorderColor: GColor;
```

The color of the border of the graphical object

```
property BorderWidth: real;
```

Width of the graphical object border

```
property Color: GColor;
```

The color of the graphical object

Methods of the BoundedObjectWPF base class

```
function RemoveBorder: BoundedObjectWPF;
```

Decorator for turning off the object boundary

```
function SetBorder(w: real := 1; c: GColor := Colors.Black):
BoundedObjectWPF;
```

Decorator to turn on the object boundary

Properties of the ObjectWPF base class

```
property Bottom: real;
```

The indent of the bottom of the graphical object from the top edge of the window

```
property Bounds: GRect;
```

Rectangle of a graphical object

```
Point: Point;
```

The center of a graphical object

```
property CenterBottom: Point;
```

The central bottom point of the graphical object

```
property CenterTop: Point;
```

The central upper point of the graphical object

```
property Color: GColor;
```

The color of the graphical object

```
property Direction: (real,real);
```

Movement direction. It is used by the Move **property** `FontColor: Color` method;

The color of the text font inside the graphical object **property** `FontName: string;`

The name of the text font inside the graphical object **property** `FontSize: real;`

The font size of the text inside the graphical object

```
property Height: real;
```

The height of the graphical object

```
property Left: real;
```

Indent the graphical object from the left window edge

```
property LeftBottom: Point;
```

Lower left corner of the graphical object

```
property LeftTop: Point;
```

Left top corner of the graphical object

```
property Number: integer;
```

An integer number displayed in the center of the graphical object. The Text property is used

```
property RealNumber: real;
```

A real number displayed in the center of the graphical object. The Text **property** `Right: real` is used;

Indent the right edge of the graphical object from the left edge of the window

```
property RightBottom: Point;
```

Lower right corner of the graphical object

```
property RightTop: Point;
```

Top right corner of the graphical object

```
property RotateAngle: real;
```

Angle of rotation of the graphical object (clockwise) **property** `ScaledHeight: real;`

Scaled height of the graphical object

```
property ScaledSize: GSize;
```

The scaled size of the graphical object `property ScaledWidth: real;`

The scaled width of the graphical object `property ScaleFactor: real;`

Scaling multiplier of the object

`property Size: GSize;`

The size of the graphical object

`property Text: string;`

Text inside a graphical object

`property TextAlignment: Alignment;`

Text alignment inside the graphical object `property Top: real;`

Indent the graphical object from the top edge of the window

`property Visible: boolean;`

Visibility of a graphical object

`property Width: real;`

Width of the graphical object

ObjectWPF base class methods

`procedure AnimMoveBy(a,b: real; sec: real := 1);`

Animates the movement of the graphical object on the vector (a,b) for sec seconds

`procedure Move; virtual;`

Moves the graphical object to the vector (dx,dy)

`procedure MoveBy(a,b: real);`

Moves the graphical object to vector (a,b)

`procedure MoveBy(v: (real,real));`

Moves the graphical object to the vector (a,b) `procedure MoveForward(r: real);`

Moves the graphical object in the RotateAngle direction (up when RotateAngle=0) `procedure MoveTime(dt: real); virtual;`

Moves the graphical object along the Direction vector with velocity Velocity in time dt

`procedure MoveTo(x,y: real);`

Moves the upper left corner of the graphical object to the point (x,y) `procedure Rotate(a: real);`

Rotates the graphical object clockwise by the angle a

`procedure RotateToPoint(x,y: real);`

Rotates the graphical object so that it "looks" at the point (x,y)

```
procedure Scale(r: real);
```

Scales a graphical object to r times its current size

RoundRectWPF class

The class of graphical objects "Rectangle with rounded edges". The base class is `BoundedObjectWPF`.

RoundRectWPF class constructors

`constructor` `(x,y,w,h,r: real; c: GColor);` Creates a rectangle with rounded edges of size (w,h) with a rounding radius r of a given color with coordinates of the upper left corner (x,y)

`constructor` `(p: Point; w,h,r: real; c: GColor);`

Creates a rectangle with rounded edges of size (w,h) with rounding radius r of specified color with coordinates of the upper left corner set by the point p **`constructor`** `(x,y,w,h,r: real; c: GColor; borderWidth: real; borderColor: GColor := Colors.Black);`

Creates a rectangle with rounded edges of size (w,h) with rounding radius r of specified color with coordinates of the upper left corner (x,y), borderWidth and borderColor **`constructor`** `(p: Point; w,h,r: real; c: GColor; borderWidth: real; borderColor: GColor := Colors.Black);`

Creates a rectangle with rounded edges of size (w,h) with a rounding radius r of the specified color with the coordinates of the upper left corner set by p, with borderWidth and borderColor

RoundRectWPF class properties

`property` `RoundRadius: real;`

Rounding radius

Methods of the RoundRectWPF class

`function` `RemoveBorder;`

Decorator for turning off the object boundary

`function` `SetBorder(w: real := 1; c: GColor := Colors.Black);`

Decorator to turn on the object boundary

`function` `SetRotate(da: real): RoundRectWPF;`

Object rotation decorator

`function` `SetText(txt: string; size: real := 16; fontname: string := 'Arial'; c: GColor := Colors.Black): RoundRectWPF;`

Object text decorator

Properties of the BoundedObjectWPF base class

```
property BorderColor: GColor;
```

The color of the border of the graphical object

```
property BorderWidth: real;
```

Width of the graphical object border

```
property Color: GColor;
```

The color of the graphical object

Methods of the BoundedObjectWPF base class

```
function RemoveBorder: BoundedObjectWPF;
```

Decorator for turning off the object boundary

```
function SetBorder(w: real := 1; c: GColor := Colors.Black):
BoundedObjectWPF;
```

Decorator to turn on the object boundary

Properties of the ObjectWPF base class

```
property Bottom: real;
```

The indent of the bottom of the graphical object from the top edge of the window

```
property Bounds: GRect;
```

Rectangle of a graphical object

```
Point: Point;
```

The center of a graphical object

```
property CenterBottom: Point;
```

The central bottom point of the graphical object

```
property CenterTop: Point;
```

The central upper point of the graphical object

```
property Color: GColor;
```

The color of the graphical object

```
property Direction: (real,real);
```

Direction of movement. Used with the Move method