На этой страничке собираются материалы, которые могут помочь при подготовке к экзамену по языкам программирования.

ЯП из курса: C, C++, Java, C#, Pascal, Delphi, Оберон-2, Модула-2, Ада (83 и 95 стандарты).

Полезные ссылки:

- Сравнение ЯП в википедии
- Энциклопедия языков программирования
- Книжка Страуструпа
- Ликбез по типизации в языках программирования / Хабрахабр

План (краткий, взят из методички Головина, подробный см. в самой методичке):

Table of Contents

Базисные типы данных в языках программирования: простые и составные типы данных, операции над ними

ADA

Integer: Размер не фиксирован.
Character: Как я понял, существует несколько разновидностей (зависит от размера) и является особым перечислимым типом (Enumeration)
String: Массив Character фиксированной длины. Так же есть стандартные пакеты, реализующие строки квазистатической и динамической длины.
Floating point: Эти типы обычно определяются вручную в виде конструкции, где Num_Digits указывает максимальную погрешность:

```
digits Num_Digits
```

Fixed Point: Эти типы также обычно определяются вручную в виде конструкции, где Delta означает погрешность:

```
delta Delta range Low .. High
```

Boolean: Перечислимый тип с особой семантикой, состоящий из значений true и false
Access: Тип указателя в языке ADA, с некоторой своей особой семантикой. В отличие от других языков (таких как C/C++) тип Access может указывать только на объекты в динамической области памяти. Кроме того этот тип лишен адресной арифметики. Эти два факта якобы позволяют избежать ошибок, связанных с указателями в C/C++. Однако, тип Access в Аде не лишен проблем висячих ссылок и мусора, ситуаций из-за которых возникает подавляющее количество очень неприятных ошибок в языках C/C++. Однако стоит отдать должное языку Ада: большинство ситуаций, в которых можно использовать адрес, зачастую решаются другими способами.

```
use Smth_Package.Entity;
type Entity_Access is access Entity;
A1, A2 : Entity_Access;
begin
   A1   := new Entity;
   A1   := new Entity; -- Образовался мусор
   A2   := A1;
   Free_Entity (A1); ''-- A1 теперь null ''
                     ''-- A2 теперь - висячая ссылка ''
end
```

Поподробнее об Access

Здесь будем пользоваться следующим примером:

```
type Person is record
   First_Name : String (1..30);
   Last_Name  : String (1..20);
end record;

type Person_Access is access Person;
```

Так как в Аде указатели могут указывать на объекты только из динамической памяти, тип Access очень тесно связан с кучей (в Аде вместо понятия кучи используется понятие пула). Такая связь позволяет для каждого типа объекта держать свой отдельный пул, который может управляться программистом, например можно вручную изменять размер пула:

```
for Person_Access'Size use New_Size; -- 0 запрещает создание новых объектов в пуле
```

Создадим пару экземпляров нашей структуры:

```
Father: Person_Access := new Person;
 -- неинициализовано
Mother: Person_Access := new Person'(Mothers_First_Name, Mothers_Last_Name);
 -- инициализованно
```

Тип Access - это высокоуровневый указатель, представленный записью с полями. Например, разыменовать указатель можно следующим образом:

```
Mother.all.Last_Name = Father.all.Last_Name ''--
 здесь, Mother.all имеет тип Person;
```

Кроме того типы указателей различаются уровнями доступа (read-only и read-write):

```
 type Person_Read_Access is access constant Person; -- read-only
 type Person_RW_Access is access all Person; -- read-write
```

Кроме оператора **new** в пуле переменные можно размещать модификатором **aliased** (у таких переменных есть атрибут 'Access):

```
 Child: aliased Person;
 Child_Access: Person_Access := Child'Access
```

Есть еще очень много интересного по этой теме. Все было взято тут.

ОБЕРОН и ОБЕРОН-2

Отличия:
В Оберон-2 добавлены:
связанные с типом процедуры;

экспорт только для чтения;

открытые массивы в качестве базового типа для указателей;

оператор with с вариантами;

оператор for.

Основные типы:
1. BOOLEAN логические значения TRUE и FALSE

2. CHAR символы расширенного набора ASCII (0X .. 0FFX)

3. SHORTINT целые в интервале от MIN(SHORTINT) до MAX(SHORTINT)

4. INTEGER целые в интервале от MIN(INTEGER) до MAX(INTEGER)

5. LONGINT целые в интервале от MIN(LONGINT) до MAX(LONGINT)

6. REAL вещественные числа в интервале от MIN(REAL) до MAX(REAL)

7. LONGREAL вещественные числа от MIN(LONGREAL) до MAX(LONGREAL)

8. SET множество из целых от 0 до MAX(SET)

Типы от 3 до 5 - целые типы, типы 6 и 7 - вещественные типы, а вместе они называются числовыми типами. Эти типы образуют иерархию; больший тип поглощает меньший тип:

LONGREAL >= REAL >= LONGINT >= INTEGER >= SHORTINT

Примеры объявлений переменных:
i, j, k: INTEGER

a: ARRAY 100 OF REAL

Операции

+, -, *, /, DIV, MOD

Операции над множествами
+ объединение

- разность (x - y = x * (-y))

* пересечение

/ симметрическая разность множеств (x / y = (x-y) + (y-x))

Отношения

, # (неравенство), <, <

, >, >=, IN (принадлежность множеству), IS (проверка типа)

Пример присваивания:

i := 0

Тип Запись

```
RECORD
    day, month, year: INTEGER
END
```

Modula-2

Порядковые: CARDINAL CHAR INTEGER BOOLEAN
Битовое множество BITSET (Величина может зависеть от реализации. Например, 32 бита)
Плавающая точка: REAL LONGREAL (Подчиняются IEEE, но зависят от реализации. К примеру, возможно REAL = LONGREAL = double)
Процедурный тип PROC

Java

Java — это язык со статической типизацией. Это значит, что каждой переменной и каждому выражению соответствует тип, известный на этапе компиляции.

Типы подразделяются на две категории (если не считать null, см. чуть дальше).

Примитивные (primitive) типы:

- boolean (true или false)
- Числовые типы:
  - Целые числа:
    - Знаковые: byte, short, int, long (8-, 16-, 32-, 64-битные).
    - Беззнаковый 16-битный: char (код символа в UTF-16[1]).
  - Числа с плавающей точкой: float, double (32-, 64-битные; IEEE 754).

Ссылочный (reference) тип:

- Классы, интерфейсы, массивы.

Также существует специальный тип null.

Объектом в Java считается экземпляр класса или массив. Значением ссылочного типа является ссылка на объект (или специальное значение null). Значение переменной ссылочного типа можно изменить, в отличии от C++.

Все объекты (включая массивы) обладают методами класса Object (java.lang.Object) (иначе говоря, с т. з. Java-программиста все объекты *наследуют* методы класса Object). Строковые литералы (например, "Hello world!") являются объектами типа String (иначе говоря, экземплярами класса String).

---

[1] Была правка UTF-16 → Unicode. Откатил. Пояснил на странице обсуждения.

Операции над простыми типами почти идентичны C/C++, однако могут выбрасывать исключения. Конкатенация строк: "Hello " + "world!" (может принимать в качестве одного из аргументов не только строку, но и любой из целых типов).

Больше информации о типах, значениях и переменных:
http://docs.oracle.com/javase/specs/jls/se7/html/jls-4.html .

C++

Целые типы:
char - размер как для хранения символа, определяется машиной (обычно байт)

short - размер, соответствующий целой арифметике на данной машине (обычно, слово)

int

long

long long

Для всех целых типов есть unsigned аналоги. По умолчанию - знаковые (так что, например, signed int - то же самое, что int). Исключение - char. Его знаковость/беззнаковость зависит от реализации.

С плавающей точкой:
float

double

long double

1 = sizeof(char) <= sizeof(short) <= sizeof(int) <= sizeof(long)
sizeof(float) <= sizeof(double) <= sizeof(long double)

```
const   = ;
// Значение переменной не может изменяться после инициализации (инициализация
 обязательна).
```

Арифметические операции:
+ (плюс, унарный и бинарный)

- (минус, унарный и бинарный)

* (умножение)

/ (деление)

Над целыми - операция % получения остатка: 7%2 = 1

Операции сравнения:
== (равно)

!= (не равно)

<, >, <=, >=

Также логические операции (&&, ||), побитовые операции(&, |, ^, <<, >>), а так же сокращённые формы многих операций: +=, -=, %=, &&=, ||=, &=, ++, -- (последние два имеют префиксные и постфиксные формы) и им подобные.

При присваивании и арифметических операциях C++ выполняет все осмысленные преобразования между основными типами, чтобы их можно было сочетать без ограничений.

Производные типы:
* - указатель на

*const - константный указатель на

& - ссылка на

[] - вектор (одномерный массив), индексация с 0

() - функция, возвращающая

Унарное & - операция взятия адреса

Например:
char* p // указатель на символ
char *const q // константный указатель на символ
char v[10] // вектор из 10 символов
char c; p = &c; // p указывает на c

C#

Встроенные типы:
1. sbyte, byte - 8-битное целое число

2. short, ushort - 16-битное целое число

3. int, uint - 32-битное целое число

4. long, ulong - 64-битное целое число

5. float, double - 32- и 64-битные числа с плавающей запятой

6. bool - 8-битное логическое значение

7. char - 16-битный знак Юникода

8. decimal - 128-битный точный дробный или целочисленный, который может представлять десятичные числа с 29 значащими цифрами.

Особняком стоят (являются встроенными типами, но при этом с т.з. программиста являются классами?):

9. string - последовательность знаков

10. object - базовый тип для всех остальных типов

Структуры и классы

В C# структуры являются урезанной версией класса. Память под структуры наравне с простыми типами выделяется на стеке (если, конечно, они не являются частью объекта — память под объекты всегда выделяется на куче). Ограничения на структуры:

1. Структура не может быть явно унаследована ни от какого класса или структуры, при этом все структуры неявно наследуют object.

2. В структуре нельзя явно определить конструктор умолчания, неявно определяемый конструктор заполняет структуру неопределенными значениями.

Типы значений (хранятся на стеке, если не являются полем класса):
Все базовые, кроме object и string.

Перечисления (enum).

Структуры (struct).

Ссылочные типы[2] (значения, на которые указывает ссылочный тип (в частности, экземпляры классов), располагаются в куче):
Класс.

Массив (является объектом, экземпляром абстрактного класса Array).

Интерфейс.

И некоторые другие.

Массивы

Длина массива — статический или динамический атрибут

Си++: длина массива - только статический атрибут.
Оберон, **Модула-2**: динамический атрибут только для формальных параметров, в остальных случаях - статический.
Ада, **Java**, **C#**: может быть и тем и другим.

---

[2] Иногда говорят «референциальные типы», здесь использован вариант перевода из msdn.

Замечание из методички:
в языках Оберон и Модула-2 длина формальных параметров — открытых массивов является динамическим атрибутом.
В других случаях длина массива — статический атрибут.
В Аде формальные параметры неограниченных типов-массивов также имеют динамический атрибут-длину (равно как и динамические массивы-локальные переменные).

Пример динамического массива в языке Java (или C#):

```
void f(int N) {
    byte [] dynArray = new byte [N];
    // ...обработка ...
}
```

ОБЕРОН И ОБЕРОН-2

Элементы массива обозначаются индексами, которые являются целыми числами от 0 до длины массива минус 1.

ТипМассив = ARRAY [Длина {"," Длина}] OF Тип.

Длина = КонстантноеВыражение.

Тип вида

```
ARRAY L0, L1, ..., Ln OF T
```

понимается как сокращение

```
ARRAY L0 OF
     ARRAY L1 OF
        ...
          ARRAY Ln OF T
```

Массивы, объявленные без указания длины, называются открытыми массивами. Они могут использоваться только в качестве базового типа указателя, типа элементов открытых массивов и типа формального параметра. Длина формальных параметров — открытых массивов является динамическим атрибутом.

Modula-2

Массивы как в Паскале (только помним про регистрозависимость: все ключевые слова IN CAPITALS).
Пример объявления массива

```
VAR
    a: ARRAY [-1..100] OF ARRAY [1..10] OF INTEGER;
```

Для передачи массива в качестве параметра (и только для этого) используется т.н. открытый (или гибкий) массив X: ARRAY OF T. Индексы будут отображены на 0..HIGH(X).

```
  PROCEDURE SCAL(VAR x,y: ARRAY OF REAL): REAL; (* Зачем здесь VAR, не знаю.
Должно быть можно без него. *)
  ========================================
  VAR
      a: ARRAY [0..N - 1] OF REAL;
      b: ARRAY [1..N] OF REAL;
  ========================================
  SCAL(a, b);
```

Управление памятью

Классы памяти

Статическая Всё что связано со словом «статический» размещается в блоке статической памяти. Время жизни – от начала программы либо момента первого входа в блок, и до конца программы.

Квазистатическая Квазистатическая память связана с понятием блока: переменные связана с этим блоком (от момента объявления, точнее, прохода code flow через объявление, до выхода из блока). Размещаются в стеке. Еще автоматической называют (автоматическая переменная).

Динамическая Время жизни зависит от процедур (ручное управление памятью) или от сборщика мусора.

Указатели

Ада, C, C++, C#, Delphi, Оберон, Модула

Указатели языка Ада 83 ссылаются только на объекты из динамической памяти. Указатель языка Си++ может ссылаться на любой объект данных (динамический, локальный, статический), что может приводить к труднообнаружимым ошибкам.

Также в языке Ада отсутствует адресная арифметика (арифметические операции над указателями), что также уменьшает вероятность появления ошибки работы с памятью.

В языке Оберон(и Оберон-2) указатель может быть объявлен только на массив или запись. ТипУказатель = POINTER TO Тип. Любая переменная-указатель может принимать значение NIL, которое не указывает ни на какую переменную вообще.

В Modula-2 можно объявлять указатель на что угодно. При том как на переменные в динамической памяти, так и в статической. Есть адресная арифметика (ADDADR, SUBADR, DIFADR), преобразование типов указателей (CAST), разыменование (^), функция взятия адреса (ADR), аллокация и деаллокация (NEW-DISPOSE или

ALLOCATE-DEALLOCATE - два варианта), размер типа в байтах (TSIZE) и NIL. В случае получения невалидного адреса, разыменования NIL и т.п. выдаётся ошибка.

```
TYPE
    TreePtr = POINTER TO TreeGlue;
    TreeGlue = RECORD
        key     : KeyType;
        left    : TreePtr; (* left child *)
        right   : TreePtr; (* right child *)
    END;
```

В Java явных указателей нет.

Преобразование типов

Явное преобразование типов

Преобразование типов называется *явным,* если оно указано программистом в исходном коде.

Неявное преобразование типов

Преобразование типов называется *неявным*, если оно не указывается программистом, но, тем не менее, выполняется (в языках со статической типизацией — соответствующие конструкции подставляются на этапе компиляции). В языках **C**[3], **C#**, **Java**, **Pascal**, **Delphi**, **Modula-2**, **Оберон**, **Оберон-2** неявными могут быть только расширяющие преобразования (иначе говоря, преобразования, к более общему типу[4]); в **C++** — любые преобразования. В языке **Ада** неявных преобразований почти[5] нет.

Неявное преобразование для пользовательских классов

Язык Java запрещает любые неявные преобразования между объектами классов (исключение составляют только неявные преобразования к стандартному типу String, разрешенные в некоторых контекстах).

Языки C++ и C# разрешают неявные преобразования для классов, определяемых пользователем.

---

[3] Если не считать (T *) → (void *).

[4] Следует иметь ввиду, что в случае простых типов не всегда более общий тип может отобразить менее общий. К примеру, вещественный тип single стандарта IEEE 754 не может точно представить число 16777217, в то время как 32-битный целочисленный тип может.

[5] Исключение составляет, например, неявное приведение числового литерала к конкретному типу. Подробнее: 1.

В **C++** преобразования определяются специальными функциями-членами: конструкторами преобразования и функциями преобразования. Конструктор преобразования имеет прототип вида:

```
X(T) // или X(T&) или X (const T&)
```

Функция преобразования имеет вид:

```
class X {
   operator T();
};
```

В языке **C#** область применения пользовательских преобразований уже, чем в языке C++. Можно определять свои преобразования только между двумя классами, нельзя определять преобразования в типы значений или из них. Преобразование из класса X в класс Y реализуется с помощью специального метода — функции преобразования:

```
static operator Y (X x) { ... }
```

Функция преобразования может быть только статическим методом либо класса x, либо класса Y. Если такая функция преобразования есть, то она вызывается с использованием обычного синтаксиса преобразований: (Y) x. Компилятор вставляет неявное преобразование из X в Y только, если соответствующая функция преобразования снабжена модификатором implicit:

```
static implicit operator Y (X x) { ... }
```

Если же используется модификатор explicit, то функция преобразования может вызываться только явно. По умолчанию принимается модификатор explicit, что снижает вероятность случайной ошибки.

В **Delphi** есть неявное преобразование типов, работает для функций. Пример (не работает для Writeln, потому что это не функция, а конструкция языка):

```
program Overloads;
{$APPTYPE CONSOLE}
type
  TRec = record
  private
    function GetOrd: Integer;
  public
    class operator Implicit(const Value: TRec): Integer;
    property ord: Integer read GetOrd;
  end;

class operator TRec.Implicit(const Value: TRec): Integer;
begin
  Result := 0;
end;
```

```
function TRec.GetOrd: Integer;
begin
  Result := 0;
end;

procedure Foo(i: Integer);
begin
end;

var
  R: TRec;
  a: array[0..0] of Integer;

begin
  Writeln(R);//E2054 Illegal type in Write/Writeln statement
  Writeln(Integer(R));//explicit cast, provided by class operator Implicit
  Writeln(R.ord);//my preferred option, a property
  a[R] := 0;//E2010 Incompatible types: 'Integer' and 'TRec'
  a[Integer(R)] := 0;//again, explicit cast is fine
  a[R.ord] := 0;//or using a property
  Foo(R);//implicit cast used for actual parameters
end.
```

Понятия *conversion* и *casting*

В большинстве языков, основанных на базе **Algol** и обладающих механизмом вложенных функций - например, в **Ada, Delphi, Modula 2** и **Pascal,** понятия *conversion* и *casting* принципиально различны. Понятие *conversion* относится к явному или неявному изменению значения одного типа данных на значение другого типа данных (например, расширение 16-битного целого до 32-битного). В этом случае, могут измениться требования к объёму выделенной памяти; могут возникнуть потери точности или округления. Понятие *casting,* напротив, обозначает *явное* изменение интерпретации *последовательности бит.* Например, последовательность из 32 бит может быть интерпретирована как целое без знака, как массив из 32 значений типа boolean или как вещественное число с одинарной точностью, соответствующее стандарту IEEE. В **C-подобных** языках, понятием *casting* обозначается явное приведение типа в независимости от того, является ли оно изменением интерпретации последовательности бит, либо же настоящим преобразованием типа.

Упаковка и распаковка

Данные понятия определены для языков **C#** и **Java.**

Упаковкой (boxing) называется процесс преобразования значения простого типа значения в экземпляр соответствующего класса-оболочки.

Распаковкой (unboxing) называется, очевидно, процесс преобразования экземпляра класса-оболочки в значение соответствующего простого типа.

C#

В **C#** упаковка и распаковка выполняются автоматически.

Пример упаковки и распаковки:

```
int i = 123;
// The following line boxes i.
object o = i;

o = 123;
i = (int) o;  // unboxing
```

Java

В **Java** автоупаковка и автораспаковка поддерживаются начиная с J2SE 5.0 (сентябрь 2004 года).

Пример упаковки и распаковки:

```
int i = 123;
Integer boxedI = Integer.valueOf(i); // boxing

Integer boxedI = Integer.valueOf(123);
int i = boxedI.intValue(); // unboxing
```

Другие языки

В некоторых языках упаковка и распаковка отсутствуют. Например, в **Smalltalk** любое значение принадлежит некоторому классу (т.е. даже значения простых типов являются экземплярами классов).

В JavaScript ситуация несколько иная. Например, есть примитивный тип *Number* для чисел. В отличие от Java и C#, это тип является полноценным (к объектам этого типа можно применить операцию `typeof`). Однако при вызове методов примитивный тип упаковывается в объект, прототипом которого является `Number.prototype` (это не тот же самый *Number*-примитивный тип!), и в котором уже определены нужные методы. Пример кода, который это демонстрирует:

```
Number.prototype.test = function() { return this; }
var x = 5;
alert(x + " " + typeof x);
x = x.test();
alert(x + " " + typeof x);
```

Операторный базис языков программирования. Управление последовательностью вычислений

ADA

Примечание: в отличие от нижеследующих языков (таких как Pascal / Delphi и Modula-2), в ADA конструкция **with** не входит в операторный базис, а служит для подключения пакетов; см. раздел 7.

```
if condition then
 statement;
elseif enother_condition then
 enother_statement;
...
else
 last_one_statement;
end if;
```

```
case X is
   when constant1 =>
      stetement1;
   when constant2 | constant3 =>
      statement2_and3;
   ...
   when others =>
      other_statement;
end case;
```

```
return; -- for procedures
return Value; -- for functions
```

```
goto Label
   Dont_Do_Smth;
<<Label>>
```

Циклы устроены посложнее:

```
Endless_Loop :
  loop
     Do_Something;
  end loop Endless_Loop;
```

```
While_Loop :
  while X <= 5 loop
     X := Calculate_Something;
  end loop While_Loop;
```

```
Until_Loop :
  loop
     X := Calculate_Something;
```

```
     exit Until_Loop when X > 5;
  end loop Until_Loop;

Exit_Loop :
  loop
     X := Calculate_Something;
     exit Exit_Loop when X > 5;
     Do_Something (X);
  end loop  Exit_Loop;

For_Loop :
  for I in Integer range 1 .. 10 loop
     Do_Something (I)
  end loop For_Loop;
```

(Замечание: именовать циклы необязательно.)

ОБЕРОН И ОБЕРОН-2

Операторы в последовательности операторов идут через точку с запятой. В конце последовательности должна стоять точка.

Пайп-символы «|» — это обязательная деталь конструкций CASE и WITH (они разделяют альтернативные варианты, за исключением умолчательного, задаваемого после ключевого слова ELSE).

```
IF Выражение THEN ПоследовательностьОператоров
{ELSIF Выражение THEN ПоследовательностьОператоров}
[ELSE ПоследовательностьОператоров]
END.
```

```
CASE ch OF
"A" .. "Z": ReadIdentifier |
"0" .. "9": ReadNumber |
"'", '"': ReadString
ELSE SpecialCharacter
END.
```

```
WHILE Выражение DO
ПоследовательностьОператоров
END.
```

```
REPEAT
ПоследовательностьОператоров
UNTIL Выражение.
```

```
FOR v := начало TO конец BY шаг DO
ПоследовательностьОператоров
END.
(* Присутствует только в Оберон-2 *)
```

```
LOOP
ПоследовательностьОператоров
END.
(* Вечный цикл; для выхода используется ключевое слово EXIT *)

WITH
Дискриминант: Значение1 DO ПоследовательностьОператоров1 |
Дискриминант: Значение2 DO ПоследовательностьОператоров2
ELSE ПоследовательностьОператоров3
END.
(* Является подобием оператора CASE-OF по дискриминанту
''   размеченного объединения; присутствует только в Оберон-2 *)''
```

Modula-2

Полностью совпадает с Обероном-2, за исключением WITH, определяемого так:

```
WITH ПеременнаяСтруктурногоТипа DO
    ПоследовательностьОператоров
    Поле1:= Значение1;
    ПоследовательностьОператоров
    Поле2:= Значение2;
    …
END.
(* Поле1, Поле2, итд. — поля переменной «ПеременнаяСтруктурногоТипа»;
''   конструкция с аналогичным поведением есть в Pascal и Delphi *)''
```

Pascal и Delphi

```
if condition1 then begin
    statement1;
    statement2;
    …
end { заметьте, точки с запятой нет }
else if condition2 then begin
    …
end
else begin
    …
end; { if-блок закончен, точка с запятой есть }

case value1 of
    constant1: begin
        statement1;
        statement2;
        …
    end;

    constant2: begin
        …
    end;
```

```
    …

    else begin
       …
    end;
end;

goto label1; { да, в стандарте Паскаля это было }

never_executed_statement1;
never_executed_statement2;
…

label1:

for variable1 := constant1 to constant2 do begin
    statement1;
    statement2;
    …
end; { если constant1 > constant2, используется downto }

while continuation do begin
    statement1;
    statement2;
    …
end;

repeat
    statement1;
    statement2;
    …
until discontinuation;
{ здесь, discontinuation означает условие останова, а не
''  продолжения (как, например, в цикле предыдущего типа);''
''  также заметьте, что скобки begin / end тут не нужны }''

with struct1.substruct1.subsubstruct1 do begin
    field1 = value1;
    field2 = value2;
    …
end;
{ field1 и field2 — это поля структуры struct1.substruct1.subsubstruct1;
''  в этом блоке они перекрывают собой любые переменные с теми же именами }''
```

C и C++

```
if (condition1) {
    statement1;
    statement2;
    …
```

```
}
else if (condition2) {
    …
}
else {
    …
}
```

```
return;        // возврат из void-функции
return value1; // возврат из функции, отдающей значение
```

```
switch (value1) {
    case constant1:
        statement1;
        statement2;
        …
        break;

    case constant2:
        …
        break;

    …

    default:
        …
        break;
}
// также вместо break можно использовать return (выходим не только из switch-блока, но и из функции).
```

```
goto label1;

never_executed_statement1;
never_executed_statement2;
…

label1:
```

```
for (initializer; continuation; increment) {
    statement1;
    statement2;
    …
}
// любые из трёх выражений в заголовке цикла могут пустовать.
// например, for ( ; ; ) {} даст вечный цикл.
```

```
do {
    statement1;
    statement2;
```

```
    …
} while (continuation);

while (continuation) {
    statement1;
    statement2;
    …
}
```

for в C# и Java

В **Java** используется 2 формы оператора цикла for. Первая форма полностью соответствует оператору for языка Си++:

```
for (e1; e2; e3) S
```

Вторая форма появилась в J2SE 5.0 (вместе с generic'ами) и используется для поэлементного просмотра коллекций (цикл for-each). Она имеет вид;

```
for (T v: Coll) S
```

Здесь Coll — коллекция элементов (типа T или приводимых к типу T). Переменная v на каждой итерации цикла принимает значение очередного элемента коллекции. Для того, чтобы объекты класса-коллекции могли появляться в цикле for-each, класс должен реализовать интерфейс Iterable.

Для того, чтобы for можно было аналогичным образом использовать в **C#**, коллекция должна реализовывать интерфейс IEnumerable (конкретно метод GetEnumerator)

```
// метод класса, который мы хотим сделать итеративным
public System.Collections.IEnumerator GetEnumerator()
{
    for (int i = 0; i < 10; i++)
    {
        yield return i;
    }
}
```

Здесь yield преобразует int в объект класса IEnumerator. Кроме того, внутреннее состояние функции запоминается и в следующий раз выполнение функции будет продолжено с того состояния и места, где функция вышла в прошлый раз. yield может вызываться только в теле for.

Процедурные абстракции

Передача параметров в подпрограммы

Для каждой подпрограммы указывается набор формальных параметров. Можно рассматривать формальные параметры как локальные переменные тела подпрограммы. При вызове подпрограммы указывается список фактических параметров. Соответствие между фактическими и формальными параметрами

выполняется по позиции в списке: первый фактический параметр соответствует первому формальному параметру и т. д. Такой способ называется *позиционным*. Язык C#, начиная с версии 4, предусматривает альтернативный — *ключевой* способ отождествления, в котором используются имена формальных параметров, но мы не будем его рассматривать. Существует три вида формальных параметров:

- входные параметры (параметры, от которых требуется только значение). Мы используем только значения фактических параметров, которые не меняются при выходе из тела функции;
- выходные параметры (эти параметры не обязаны иметь начальное значение, но могут быть изменены в теле функции);
- изменяемые параметры (требуется и исходное значение, и возможность его изменения).

С входным параметром может связываться произвольное выражение, а выходным или изменяемым — только объекты, которые могут стоять в левой части оператора присваивания. В большинстве языков программирования вместо указания вида параметра указывается способ (механизм) связывания параметра, называемый способом передачи параметра.

Существует два основных способа передачи параметров: *по значению* и *по ссылке*.

Передача параметров по значению

Формальный параметр есть некоторая локальная переменная. Место для локальных переменных отводится в стеке. При вызове подпрограммы значение фактического параметра копируется в соответствующий формальный параметр. Все изменения формального параметра связаны с изменением локальной переменной и не сказываются на фактическом параметре. Перед копированием может потребоваться приведение типа, если типы фактического и формального параметров не совпадают.

Передача параметров по ссылке

Фактически этот способ есть передача ссылки по значению. Формальный параметр — это ссылка на объект. (Существует мнение, что данное «определение» не только не отражает сути явления, но и неверно в корне. В дискуссии вокруг передачи аргументов в Java Dale King дал следующее определение. *Передача по ссылке — это когда lvalue формального параметра устанавливается в lvalue фактического параметра.*) В момент вызова происходит инициализация ссылки фактическим параметром. Преобразования типов в этот момент не происходит: типы формального и фактического параметров должны совпадать. Поскольку ссылка после инициализации отождествляется с объектом, то любые изменения формального параметра подразумевают изменения фактического параметра. Очевидно, что способ передачи по значению соответствует семантике входных формальных параметров. По ссылке можно передавать выходные и изменяемые параметры.

Аргументы в C/C++ всегда передаются по значению

В C++ есть ссылочный тип. Переменная ссылочного типа может ссылаться на значение любого типа, должна быть инициализирована и не может менять значения. С помощью передачи переменной ссылочного типа можно имитировать все возможности контрукции var из **Pascal**. Но можно действовать в стиле C — передавать указатель. В свою очередь, чтобы менять указатель, можно передавать в функцию/метод указатель или ссылку на него.

Аргументы в Java всегда передаются по значению

Существует распространённое **заблуждение** о том, что «объекты передаются по ссылке, а примитивные типы — по значению». **На самом деле** ситуация иная:

i. Аргументы любого типа передаются по значению. Объекты, однако, не передаются вообще.
ii. Значения переменных всегда примитивы или ссылки (или null), но никак не объекты.

Подробнее см. http://www.yoda.arachsys.com/java/passing.html .

В соответствии с изложенным выше, метод может изменить объект через аргумент-ссылку. С примитивным типом это не пройдёт, так как в Java нет ссылок на значения примитивных типов. Чтобы иметь возможность изменить из метода значение некоторой внешней переменной примитивного типа, нужно чтобы эта переменная была полем некоторого объекта.

В связи с этим для примитивных типов были введены классы-обёртки. Объект такого класса содержит в себе значение примитивного типа, которое можно как прочитать, так и поменять. См. также Упаковка и распаковка

Перегрузка операций

Ада 83, Ада 95, Си++, Java, Delphi, C#

Понятие «перегрузка» (англ. overloading) означает, что одному имени в одной области видимости может соответствовать несколько определений. В современных языках программирования перегружаться могут только имена подпрограмм, но не типов, переменных, модулей.

Пример на языке C++:

```
class X {
public:
    void f();
    void f(int)
};

X a;
```

```
a.f(); // первая функция
a.f(0); // вторая функция
```

Отличие перегрузки от замещения (скрытие, англ. hide) состоит во-первых, в том, что перегрузка обрабатывается статически (на этапе трансляции) (а замещение?), в во-вторых, при замещении речь идет о разных областях видимости: базовый класс с объявлением виртуального метода (объемлющая область видимости) и производный класс с замещающим методом (вложенная область видимости).

ADA

В аде есть процедуры и функции. Первые не возвращают значения, вторые - возвращают. Самое интересное здесь - это виды передачи параметров. В аде их 4.

in: Фактический параметр не изменяется и передается как константа read-only (этот модификатор используется по умолчанию)
out: Фактический параметр передается только на изменение. Внутри подпрограммы этот параметр не доступен на чтение.
in out: Фактический параметр передается с read-write семантикой.
access: Фактический параметр является указателем. Это не модификатор, а тип и может использоваться с 3-мя вышеперечисленными модификаторами.

До 2012-го года функции в Аде могли принимать только **in** и **access** параметры.

Имеются типы-указатели на подпрограмму, определяются в виде.

```
type PFunction is access function (X : in Param_Type) return Return_Type;
```

ОБЕРОН И ОБЕРОН-2

Имеются два вида процедур: собственно процедуры и процедуры- функции. Последние активизируются обозначением функции как часть выражения и возвращают результат, который является операндом выражения. Собственно процедуры активизируются вызовом процедуры. Процедура является процедурой-функцией, если ее формальные параметры задают тип результата. Тело процедуры-функции должно содержать оператор возврата, который определяет результат.

```
PROCEDURE log2 (x: INTEGER): INTEGER;
 VAR y: INTEGER; (*предполагается x>0*)
BEGIN
 y := 0; WHILE x > 1 DO x := x DIV 2; INC(y) END;
 RETURN y
END log2
```

Определение новых типов данных. Логические модули. Классы

ADA

В Аде вообще типы принято определять вручную, пользуясь конструкциями **type** и **subtype**. Например:

```
type Short is range -128 .. +128
type Index is range 0 .. 256
type My_String is array ( Index range <> ) of Character -- My_String - это массив, индексируемый типом Index в неизвестных пока границах, состоящий из Character
subtype String_10 is My_String ( 0 .. 10 ); -- Подтип типа My_String, состоящий из 11 элементов.
type Money is delta 1/100 range 0.0 ... 10000.0
```

Ключевое слово **Limited** означает, что у новоявленного типа нет стандартных операций до тех пор, пока программист сам их не опишет.

В языке Ada есть мощный механизм пакетов, и этим он, несмотря на то, что основан на Pascal, совсем на него не похож. Пакеты заимствованы из других языков. Они похожи на классы тем, что являются средством абстракции, позволяющем инкапсулировать сложность внутри пакета, предоставив наружу только интерфейс. Ладно, к сути.

Каждый пакет физически разделён на две части.

```
-- Интерфейс
package Points is
   type Point is record
       x: Integer;
       y: Integer;
   end record;
   zero: constant Point := (0, 0);
   procedure setToZero(point: in out Point);
   function dotProduct(point1: in Point; point2: in Point) return Integer;
end Points;
```

```
-- Реализация
package body Points is
   procedure setToZero(point: in out Point) is
       begin
       point := zero;
       end setToZero;
   function dotProduct(point1: in Point; point2: in Point) return Integer is
       begin
       return point1.x * point2.x + point1.y * point2.y;
       end setToZero;
end Points;
```

Ada имеет средства разграничения доступа. Всё, что следует за спецификатором **private**, не будет доступно при импорте пакета.

```
-- Интерфейс
package Points is
   type Point is private; -- Это ключевое слово как бы намекает.
   zero: constant Point; -- Тут тоже неявный private.
   procedure setToZero(point: in out Point);
```

```
   function dotProduct(point1: in Point; point2: in Point) return Integer;
private
   type Point is record
       x: Integer;
       y: Integer;
   end record;
   zero: constant Point := (0, 0);
end Points;
```

Вне пакета можно будет объявить переменные типа «Point», но все действия будут ограничены присваиваниями, сравнениями на равенство и проверками принадлежности (*нужно описать подробнее*). Поэтому в пакет надо бы добавить «конструктор», если, конечно, предполагается возможная инициализация этих структур вне пакета. Если стандартная реализация этих операций не устраивает, можно использовать **limited private**, который в остальном ничем от **private** не отличается, но этих операций не позволяет.

Реализация может также иметь часть-инициализацию. Это всё, что после **begin**.

```
package body Points is
    procedure setToZero(point: in out Point) is
        begin
        point := zero;
        end setToZero;
    function dotProduct(point1: in Point; point2: in Point) return Integer is
        begin
        return point1.x * point2.x + point1.y * point2.y;
        end setToZero;
begin
    Ada.Text_IO.Put_Line("Package ready!");
end Points;
```

Пакеты можно расширять. Взять и определить пакет *Points.RandomDistributions*, в нём можно пользоваться всем, что объявлено в пакете-родителе. Пакет можно сделать недоступным вне своей иерархии.

```
private package Points.internalThing is
    <...>
```

При подключении потомка родитель-пакет подключается автоматически.

О подключениях. Подключить пакет можно с помощью конструкции **with**.

```
with Points.RandomDistributions;
```

После этого можно будет пользоваться ввсем, что в пакете лежит, через точечную нотацию. Если же вызовов слишком много, то можно влить содержимое пакета в текущую область видимости с помощью **use** (только после подключения!).

```
use Points.RandomDistributions;
```

В случае конфликтов имён придётся-таки предварять каждое обращение именем пакета.

ОБЕРОН И ОБЕРОН-2

В Обероне-2 стандартная процедура NEW используется, чтобы распределить блоки данных в свободной памяти. Нет, однако, никакого способа явно освободить распределенный блок. Взамен Оберон-среда использует сборщик мусора чтобы найти блоки, которые больше не используются и сделать их снова доступными для распределения.

Расширение типов

Наиболее важным добавлением(от-но Modula-2) является механизм расширенных типов записи. Он позволяет конструировать новые типы на основе уже существующих и задает определенную степень совместимости между новыми и старыми типами. Предположим, что у нас имеется тип

```
T = RECORD x,y: INTEGER END
```

В дополнение к существующим полям могут быть заданы новые:

```
T0 = RECORD (T) z: REAL END
T1 = RECORD (T) w: LONGREAL END
```

Здесь определяются типы с полями x,y,z и x,y,w соответственно. Мы говорим, что тип T'

```
T' = RECORD (T)  END
```

является (прямым) расширением типа T, и наоборот, T называется (прямым) базовым (base) типом для T'.

Modula-2

Перечислимый тип

```
TYPE DaysOfWeek = (sunday, monday, tuesday, wednesday, thursday, friday, s
aturday);
(* тогда ORD(friday) = 5, MAX(DaysOfWeek) = saturday *)
```

Тип поддиапазон

```
TYPE SubType = [min .. max];
```

Дополнительно можно указывать хостовой тип

```
TYPE
    SubType1 = [0..7]; (* host type is CARDINAL *)
    SubType2 = INTEGER [0..7]; (* host type is INTEGER *)
    WeekDays = [mon .. fri]; (* host type is DaysOfWeek *)
```

Говорят, что тип T1 совместим с типом T0, если либо он описан как T1 = T0, либо как диапазон T0, либо если T0 является диапазоном T1, либо если и T0, и T1 - оба диапазоны одного и того же (базового) типа.

Записи

Как в Паскале.

```
TYPE
    myRec = RECORD
        val1: type1;
        valn: typen;
    END;
```

Namespace в C#

Ключевое слово namespace используется для объявления области, которая содержит набор связанных объектов. Можно использовать пространство имён для организации элементов кода, а также для создания глобально уникальных типов.

```
namespace SampleNamespace
{
    class SampleClass { }

    interface SampleInterface { }

    struct SampleStruct { }

    enum SampleEnum { a, b }

    delegate void SampleDelegate(int i);

    namespace SampleNamespace.Nested
    {
        class SampleClass2 { }
    }
}
```

Классы

Конструкторы

Обычно выделяют следующие типы конструкторов: конструктор по умолчанию, конструктор копирования и конструктор преобразования. Конструкторы имеют пользовательскую (тело объявленного конструктора) и системную (сгенерированный код) части.

Деструкторы и финализаторы

Деструкторы и финализаторы — это специальные функции-члены класса, автоматически вызываемые при уничтожении объекта класса. Роль деструкторов и финализаторов — освобождение захваченных объектом ресурсов. Если объект не захватывает ресурсы или ресурсы освобождаются автоматически, то нужды в деструкторе (финализаторе) нет. Деструкторы *не имеют* параметров. В языке Java деструкторов нет, их роль (до определенной степени) играет метод void finalize() — финализатор. В языке **C#** формально деструкторы есть, однако их поведение аналогично поведению финализаторов в языке **Java**. Деструкторы в **C++** отличаются от финализаторов в **Java** и **C#** тем, что в **C++** можно достаточно точно установить момент вызова деструктора.

i. Деструкторы статических объектов вызываются после выхода из функции main.
ii. Деструкторы квазистатических объектов выполняются при выходе из блока. Причём, деструкторы вызываются строго в обратном порядке относительно порядка вызова конструкторов квазистатических объектов данного блока.
iii. Деструкторы динамических объектов выполняются при вызове операции delete.
iv. Деструкторы временных объектов выполняются при выходе из конструкции, в контексте которой был создан объект.

В **Java** и **C#** время вызова финализатора не определено из-за наличия сборщика мусора. Как и конструкторы, деструкторы имеют пользовательскую и системную части.

Неявные контрукторы/деструкторы/операции в C++

Следующие элементы класса могут быть определены неявно:

- Конструктор по умолчанию (если нет ни одного явно определённого конструктора).
- Конструктор копирования (если конструктор копирования не описан явно). Осуществляет т.н. поверхностное копирование полей класса (бит в бит).
- Деструктор (если деструктор не описан явно). Вызывает деструкторы объектов-полей (не полей-указателей!).
- Операция присваивания (если не определена явно). Записывает значения полей объекта-аргумента в целевой объект (тоже поверхностное копирование), возвращает ссылку на целевой объект.

Фишки:

- Запрет копирования объекта: определить конструктор копирования в приватной части класса.
- Запрет создания экземпляровобъекта — глобальных и локальных переменных, иначе говоря данные запрещено размещать в секциях .data/.bss и на стеке, разрешено только в динамической памяти. Нужно определить декструктор в

приватной части класса. Освобождение ресурсов объекта при таком подходе осуществляется через метод данного класса.

- Запрет присваивания одного объекта другому: определение операции присваивания в приватную часть класса.
- Запрет неявного вызова конструктора копирования, конструктора преобразования (иначе говоря, конструкторов с одним параметром) или операции преобразования (последнее — только в C++11). Нужно добавить в начало заголовка ключевое слово explicit.

Свойства

В C# и Delphi есть поддержка свойств (property).

Свойство - способ доступа к внутреннему состоянию объекта, имитирующий переменную некоторого типа. Обращение к свойству объекта выглядит так же, как и обращение к структурному полю (в структурном программировании), но, в действительности, реализовано через вызов функции. При попытке задать значение данного свойства вызывается один метод, а при попытке получить значение данного свойства — другой.

Свойства повышают гибкость и безопасность программирования, поскольку, являясь частью (открытого) интерфейса, позволяют менять внутреннюю реализацию объекта без изменения его свойств. Свойства предназначены для того, чтобы свести программирование к операциям над свойствами, скрывая вызовы методов.

Свойства в **C#** — поля с логическим блоком, в котором есть ключевые слова get и set.

```
class MyClass
{
    private int p_field;
    public int Field
    {
        get
        {
            return p_field;
        }
        private set
        {
            p_field = value;
        }
    }
}
```

Для описания свойства в **Delphi** служит слово property.

```
TMyClass = class
private
  FMyField: Integer;
  procedure SetMyField(const Value: Integer);
```

```
  function GetMyField: Integer;
public
  property MyField: Integer read GetMyField write SetMyField;
end;

function TMyClass.GetMyField: Integer;
begin
  Result := FMyField;
end;

procedure TMyClass.SetMyField(const Value: Integer);
begin
  FMyField := Value;
end;
```

Статические классы

Статический класс — это, в общем случае, достаточно неформальное понятие. Обычно так называют класс, все члены которого объявлены статическими. Для работы с таким классом не требуется создавать экземпляр и часто, говоря о статическом классе, подразумевают невозможность создания экземпляра. Такой класс, как правило, не имеет конструкторов и деструкторов, либо они объявлены приватными (в зависимости от языка).[6]

Иногда понятие *статического класса* закрепляется на уровне языка программирования. В этом случае язык добавляет некоторые требования, кроме статичности всех членов и невозможности создания экземпляров: например, невозможность наследования от данного класса или невозможность реализации таким классом интерфейса (см. далее про C#).

Следует понимать, что в контексте разных языков программирования могут подразумеваться разные понятия статического класса, разной степени формальности (вплоть до строго закреплённого языком понятия).

В C++

В C++ понятие статического класса отсутствует. Для реализации статического класса нужно объявить члены класса статическими (static) и запретить создание его экземпляров, поместив конструктор по умолчанию и конструктор копирования в приватную (private) часть класса (а по-хорошему, ещё и деструктор). Похожих свойств можно добиться, используя шаблон проектирования singleton — для класса-singleton'а будет существовать *ровно один* экземпляр данного класса.

---

[6] Стоит уточнить, что статический класс в C# может иметь статический конструктор, см. далее раздел про C#.

В Java

В Java также нет понятия соответствующего статическому классу в данном выше понимании (статические вложенные классы — совершенно другая вещь, о них далее в этом разделе). Ситуация аналогична ситуации в C++. Для реализации статического класса нужно объявить все члены статическими (static) и запретить создание его экземпляров, объявив конструктор приватным (private). Чтобы ближе имитировать статический класс C# можно отказаться от наследования с помощью ключевого слова *final*. Похожих свойств можно добиться используя шаблон проектирования singleton.

В Java существует модификатор static для класса, но применяться он может **только** ко вложенным (nested) классам. Статический вложенный класс нельзя считать статическим в том смысле, который вкладывается в это понятие в начале данного раздела и далее в разделе про Java этот смысл не подразумевается.

Вложенные классы делятся на две категории — статические и нестатические. Вложенные классы, объявленные статическими называются статическими вложенными классами (static nested classes). Нестатические вложенные классы называются внутренними (inner classes). Ко вложенному классу могут быть применены любые модификаторы доступа (тогда как к классу верхнего уровня — только *public* или *package private*).

```
class OuterClass {
    ...
    static class StaticNestedClass {
        ...
    }
    class InnerClass {
        ...
    }
}
```

Статический вложенный класс может использоваться без экземпляра объемлющего класса. Он взаимодействует с членами объемлющего класса (и любого другого) точно так же как любой класс верхнего уровня (не вложенный). В сущности, он ведёт себя как класс верхнего уровня, вложенный в другой лишь с точки зрения логической группировки классов.

```
// Создание экземпляра статического вложенного класса.
OuterClass.StaticNestedClass nestedObject =
    new OuterClass.StaticNestedClass();
```

Внутренний класс имеет доступ ко всем членам объемлющего класса, даже если они объявлены с модификатором *private* и не может содержать статические члены. Экземпляр внутреннего класса можно получить только от конкретного экземпляра объемлющего класса. Можно сказать, что каждый из экземпляров внутреннего класса ведёт себя как часть соответствующего экземпляра объемлющего класса.

```
// Создание экземпляра внутреннего класса.
OuterClass.InnerClass innerObject =
    outerObject.new InnerClass();
```

Больше информации:

- http://docs.oracle.com/javase/tutorial/java/javaOO/nested.html
- http://stackoverflow.com/questions/3584113/java-static-class

В C#

В C# существует понятие статического класса. Статический класс создаётся с помощью ключевого слова *static*. Компилятор предоставляет относительно такого класса некоторые гарантии.

- Может содержать только статические члены.
- Нельзя создать экземпляр такого класса.
- Неявно определяется как *sealed*, т.е. наследоваться от такого класса нельзя.
- Не может иметь instance constructors.
- Может иметь статический конструктор, который выполняется перед первым обращением к данному классу.
- Не может быть унаследован ни от какого объекта, исключая Object, от которого, как и все классы, наследуется неявно.

Также можно обратить внимание на тот факт, что к члену интерфейса не может быть применено ключевое слово *static*, равно как и к соответствующему члену реализующего интерфейс класса. Напрямую к статическим классам это не относится, но означает, что реализовать интерфейс с хотя бы одним членом статический класс не может. Вопрос о том, может ли статический класс быть реализацией интерфейса без членов остаётся в рамках данной статьи открытым, профессионалы могут добавить ответ и ссылку на него.

Больше информации:

- Msdn: «Static Classes and Static Class Members (C# Programming Guide)».
- Msdn: «Interfaces (C# Programming Guide)».
- http://stackoverflow.com/questions/259026/why-doesnt-c-sharp-allow-static-methods-to-implement-an-interface

Объединение типов (запись с вариантами)

Объединение типов (или запись с вариантами) — это конструкция, объединяющая в один тип несколько различных структур (вариантов). Все варианты в объединении начинаются с одного адрес и занимают одну и ту же память.

Размеченное объединение типов содержит одно выделенное поле (дискретного типа данных) — общее для всех вариантов. Такое поле называется дискриминантом.

Значение дискриминанта определяет, по какому варианту выделена память в переменной-экземпляре размеченного объединения.

Семантика копирования

Проблема копирования состоит в том, что копируемый объект может содержать ссылки на другие объекты. Проблема копирования состоит в том, что объекты могут содержать ссылки на другие объекты. При копировании таких ссылок возникает дилемма: копировать либо только ссылку, либо полностью содержимое объекта, на который указывает ссылка. Первый вид копирования называется *поверхностным*, а второй — *глубоким.* В общем случае транслятор не в состоянии выбрать нужную семантику копирования, поэтому в языках программирования, которые мы рассматриваем, принята следующая схема: по умолчанию реализуется *поверхностное копирование,* но программисту предоставляются средства для указания *глубокой семантики копирования.* В **C++** этими средствами являются переопределение конструктора копирования и операции присваивания, в **Java** и **C#** — наследование и реализация специального интерфейса.

Особенности копирования объектов в Java

Все массивы реализуют интерфейс Cloneable и могут копироваться. Поэтому далее рассматриваются только экземпляры классов.

Для того, чтобы разрешить копировать экземпляр класса в Java нужно сделать следующее:

i. Реализовать интерфейс Cloneable. Этот интерфейс не содержит методов и служит только для того, чтобы указать, что вы разрешаете копировать этот объект.
ii. Переопределить метод clone класса Object со спецификатором доступа public. Заголовок метода в Object выглядит так:
```
protected Object clone() throws CloneNotSupportedException
```
Object.clone() выбрасывает исключение CloneNotSupprotedException только в случае, если не реализован интерфейс Cloneable. Однако, переопределяющий метод может выбрасывать исключение CloneNotSupportedException для того, чтобы показать, что объект не может или не должен быть скопирован.
iii. По соглашению, переопределяющий метод должен получать копию объекта с помощью вызова метода clone суперкласса (конструкция super.clone()). Object.clone() возвращает новый объект со значениями полей, в точности равными соответствующим значениям переданного в параметре объекта (поверхностное копирование).
iv. Если требуется глубокое копирование, то перед тем, как вернуть объект-копию, нужно привести его к соответствующему типу и заменить значения необходимых полей вручную созданными копиями.

Больше информации:

- http://docs.oracle.com/javase/7/docs/api/java/lang/Cloneable.html

- http://docs.oracle.com/javase/7/docs/api/java/lang/Object.html#clone%28%29
- http://docs.oracle.com/javase/7/docs/api/java/lang/CloneNotSupportedException.html

И. Г. Головин выделяет не два (копируется / не копируется) вида копирования в Java, а четыре. ЕМНИП, добавляется вариант с перманентным выбросом исключения CloneNotSupportException для явного запрета копирования. А четвёртый не помню. *TODO: описать все варианты.*

В общем случае всё это говорит о сложности проблемы копирования. Проблемы у программистов на **C#** — такие же.

Модульность и раздельная трансляция

Раздельная трансляция

C, C++

Раздельная трансляция означает то, что программа разбивается на части — физические модули или единицы компиляции. Каждая единица может или обязана транслироваться отдельно от остальных. Независимая раздельная трансляция означает то, что транслятор не обладает информацией об уже оттранслированных единицах и поэтому не может проверить корректность межмодульных связей.

Модульность

ОБЕРОН И ОБЕРОН-2

Модуль - совокупность объявлений констант, типов, переменных и процедур вместе с последовательностью операторов, предназначенных для присваивания начальных значений переменным. Модуль представляет собой текст, который является единицей компиляции.

```
Модуль   =  MODULE идент ";" [СписокИмпорта] ПоследовательностьОбъявлений
[BEGIN ПоследовательностьОператоров] END идент ".".
СписокИмпорта    =  IMPORT Импорт {"," Импорт} ";".
Импорт   =  [идент ":="] идент.
```

Список импорта определяет имена импортируемых модулей. Последовательность операторов после символа BEGIN выполняется, когда модуль добавляется к системе (загружается). Это происходит после загрузки импортируемых модулей. Отсюда следует, тот циклический импорт модулей запрещен. Отдельные (не имеющие параметров и экспортированные) процедуры могут быть активированы из системы. Эти процедуры служат командами.

```
MODULE Trees; (* экспорт: Tree, Node, Insert, Search, Write, Init *)
IMPORT Texts, Oberon; (* экспорт только для чтения: Node.name *)

TYPE
     Tree* = POINTER TO Node;
```

```
          Node* = RECORD
               name-: POINTER TO ARRAY OF CHAR;
               left, right: Tree
          END;

     VAR w: Texts.Writer;

     PROCEDURE (t: Tree) Insert* (name: ARRAY OF CHAR);
          VAR p, father: Tree;
     BEGIN p := t;
          REPEAT father := p;
               IF name = p.name^ THEN RETURN END;
               IF name < p.name^ THEN p := p.left ELSE p := p.right END
          UNTIL p = NIL;
          NEW(p); p.left := NIL; p.right := NIL; NEW(p.name, LEN(name)+1); COP
Y(name, p.name^);
          IF name < father.name^ THEN father.left := p ELSE father.right := p
END
     END Insert;

     PROCEDURE (t: Tree) Search* (name: ARRAY OF CHAR): Tree;
          VAR p: Tree;
     BEGIN p := t;
          WHILE (p # NIL) & (name # p.name^) DO
               IF name < p.name^ THEN p := p.left ELSE p := p.right END
          END;
          RETURN p
     END Search;

     PROCEDURE (t: Tree) Write*;
     BEGIN
           IF t.left # NIL THEN t.left.Write END;
           Texts.WriteString(w, t.name^); Texts.WriteLn(w); Texts.Append(Obero
n.Log, w.buf);
           IF t.right # NIL THEN t.right.Write END
     END Write;

     PROCEDURE Init* (t: Tree);
     BEGIN NEW(t.name, 1); t.name[0] := 0X; t.left := NIL; t.right := NIL
     END Init;

     BEGIN Texts.OpenWriter(w)
     END Trees.
```

Загруженный модуль может вызывать команду незагруженного модуля, задавая ее имя как строку. Специфицированный модуль при этом динамически загружается и выполняется заданная команда. Динамическая загрузка позволяет пользователю запустить программу как небольшой набор базисных модулей и расширять ее, добавляя последующие модули во время выполнения по мере необходимости. Интерфейс модуля (объявления экспортируемых объектов) извлекается из модуля

так называемым смотрителем, который является отдельным инструментом среды Оберон.

Modula-2

Программа представляет собой набор модулей — самостоятельных единиц компиляции, которые могут компилироваться раздельно. При этом программный модуль может быть (но не обязан) разделён на две части: модуль определений и модуль реализации. Модуль определений — это внешний интерфейс модуля, то есть набор экспортируемых им имён констант, переменных, типов, заголовков процедур и функций, которые доступны внешним модулям. Модуль реализации содержит программный код, в частности, конкретизацию описаний всего, что перечислено в модуле определений. Например, некоторый тип «запись» может быть объявлен в модуле определений с указанием лишь его имени, а в модуле реализации — с полной структурой. В этом случае внешние модули могут создавать значения данного типа, вызывать процедуры и функции, работающие с ним, выполнять присваивание переменных, но не имеют прямого доступа к структуре значений, поскольку эта структура не описана в модуле определений. Если для этого же типа описать в модуле определений структуру, то она станет доступна. Помимо модулей глобального уровня в Модуле-2 допускается создавать локальные модули. Импорт определений, описанных в прочих модулях, полностью контролируется. Можно импортировать модули определений целиком, но синтаксис позволяет существенно уточнять списки импорта, например, импортировать из модуля конкретные константы, переменные, процедуры и функции, только те, которые необходимы.

```
DEFINITION MODULE ModuleName;  (* Модуль определений *)
{Import}
{Declaration}
END ModuleName.
=======================
Где Import это:
[FROM ModuleName] IMPORT
    identifier {,identifier};

IMPLEMENTATION MODULE ModName;  (* Модуль реализаций *)
{Import}
{Declaration}
[ BEGIN
    ListOfStatements
[EXCEPT
    ListOfStatements]
[
 FINALLY
    ListOfStatements
 EXCEPT
    ListOfStatements
]
```

```
   ]
   END ModName.

   MODULE ModName; (* Должен существовать единственный на весь проект. Это ma
in. *)
   {Import}
   {Declaration}
   BEGIN
       ListOfStatements
   [EXCEPT
       ListOfStatements]
   END ModName.
```

Для обеспечения видимости в других модулях объявления функций, переменных и типов, описанные в модулях определений, можно импортировать. Остановимся на типах. При импортировании их внутренняя структура становится видна импортирующему модулю (т.н. прозрачный экспорт). Существует скрытый экспорт (opaque export), где видно становится только имя типа. Он возможен только для указательного типа (но сделаем указатель на структуру - и вуаля, получим инкапсуляцию и абстракцию данных).

ADA

Ада, наверное - единственный язык со вложенностью модулей, их раздельной компиляцией и двойной связью одновременно. Опишем вложенную спецификацию, тут ничего сложного:

```
package Outer is
   ...
   procedure some_proc (X: Some_type) is private;
   package Inner is
   ... ''-- Тут, в общем то видно все, что есть в пакете Outer.
   end Inner;
   ...
end Outer;
```

Тела этих пакетов, как и тело функции, можно разнести по разным файлам (единицам компиляции), используя "заглушку" **separate**:

```
package body Outer is
   ...
   package Inner is separate;
   ...
   procedure some_proc (X: some_type) is separate;
end Outer;
```

Теперь опишем тела модуля Inner и процедуры.

```
separate (Outer) ''--Тут нет `;' ''
package body Inner is
```

```
   ...
end Inner;

separate (Outer)
procedure some_proc (X: some_type) is
   ...
end some_proc;
```

Вот здесь заглушка **separate** и является двойной модульной связью.

Исключительные ситуации и обработка ошибок

Зачечание
Кроме перечисленных в итоговой таблице языков исключения поддерживает Visual Basic.

Исключения и блоки try {} catch {} finally {}. Семантика возобновления и семантика завершения.

Семантика возобновления: после обработки исключения управление может вернуться непосредственно в точку, где возникло исключение (варианты: на следующий оператор или на любой оператор из того же блока). Пример языка с семантикой возобновления: Visual Basic.

Моделирование семантики возобновления на C++:

```
bool need_restart = true;
while (need_restart) {
    need_restart = false;
    try {
        // Some code here
    } catch (...) {
        // C# - просто catch, без круглых скобок
        // Java - catch (Throwable e)

        need_restart = true;
    }
}
```

Семантика завершения: после возникновения исключения блок, в котором оно возникло, обязательно завершается. Обработка исключения происходит в блоках, вызвавших блок с исключением. Пример языка с семантикой завершения: Си++.

catch (в delphi - **except**) - то что будет выполнено в случае ошибки в блоке try. **finally** - то что будет выполнено в любом случае, вне зависимости от того что произошло в блоке try.

Конструкция **try** ... **finally** ... есть в C#, Java, Delphi. Декларируется, что в C++ в **finally** нет необходимости в виду RAII и, как следствие, выполнении деструкторов на выходе из блока.

throw (C++) и throws (Java)

Текст раздела построен с т.з. программиста Java (иначе говоря, обозначены отличия синтаксиса и семантики конструкции C++ от конструкции Java, а не наоборот). Причина — одно из заданий экзамена по ЯПам звучит примерно как «опишите конструкцию throws в Java (зачем нужно и как работает). Как моделируется на C++, Delphi[7]??».

Примеры употребления конструкций:

```
// Java

void someMethod() throws IOException, SomeOtherException { … }
// IOException – стандартный класс исключения, наследник Exception.
// SomeException должен быть наследником Throwable.

void someMethod() { … }
// Запись «не выбрасывает исключения»: отсутствие throws и списка исключений.

// C++

void someMethod() throw (SomeException1, SomeException2) { … }
// SomeException1, SomeException2 – вообще говоря, любые типы.
// Декларациии throw должны совпадать как при определении, так и при описании
.

void someMethod() throw () { … }
// Запись «не выбрасывает исключения»: throw ().

void someMethod() { … }
// Может выбрасывать любые исключения.
```

Данные конструкции служат для того, чтобы показать программисту и компилятору, что данный метод (или, в случае C++, метод или функция) может выбрасывать исключения соответствующих типов. Насколько я понимаю, всё это влияет только на статические проверки компилятора и эстетические чувства программиста. В runtime эти декларации никак себя не проявляют, поэтому употребляемые здесь «может / не может выбрасывать исключение данного типа» и тому подобные обороты следует понимать в контексте статических проверок. *(Замечание: не совсем верно. Исключения могут возникнуть в виртуальной функции или в отдельно оттранслированной. Memento std::unexpected().)*

В Java считается, что метод, выбрасывающий исключение должен обозначить это с помощью конструкции throws. Иначе говоря, считается, что если директивы throws нет, то метод не выбрасывает исключений. Компилятор делает некоторые

---

[7] Видимо, без использования «родных» аналогичных конструкций, нужно уточнить.

статические проверки, по крайней мере, запрещает выбрасывать исключения, не перечисленные в throws, явно — с помощью оператора throw.

В C++, в отличие от Java, если директива throw не задана, то считается, что данный метод или функция может выбрасывать любые исключения. Статические (времени компиляции) проверки делаются только для тех методов/функций, для которых указан (возможно пустой) список исключений.

Одно из заданий экзамена по ЯПам

2004 г, задание 8. Смоделируйте на языке Си++ функцию

```
void f() throw (E1,E2,E3) { g(); h(); }
```

предполагая, что конструкция throw[8] не допускается компилятором.

```
void f()
{
    try {
        g(); h();
    } catch (E1) {
        throw;
    } catch (E2) {
        throw;
    } catch (E3) {
        throw;
    } catch (...) {
        unexpected();
    }
}
```

ADA

Создание исключения:

```
New_Exception: exception;
```

Поднятие/возбуждение исключения[9]:

```
raise New_Exception;
```

Отлов исключения:

---

[8] Видимо, подразумевается конструкция throw (…) в заголовке метода/функции, а не выбрасывание (или повторное выбрасывание) исключения с помощью оператора throw.

[9] Терминология отличается от принятой в мире языков с C++-подобным синтаксисом. Там: выбрасывание/выброс (throw), здесь — поднятие/возбуждение (raise).

```
begin
   Rise_Exception;
exception
   when New_Exception =>
      Do_Smth;
end;
```

Отлов исключения так же может иметь такой вид:

```
when Error: Error_Name =>
```

Однако, что он означает я до конца не понял, но на экзамене это вряд ли нужно будет. Кому интересно, брал здесь.

Наследование типов и классов

C# и Java

В C# и Java можно в дочернем классе сослаться на экземпляр родительского с помощью ключевых слов **base** (C#) и **super** (Java).

Кроме того, в этих языках есть ключевые слова **sealed** (C#) и **final** (Java). Они могут находиться в заголовке[10] метода (в случае C# — только виртуального) или класса (**final** также может находиться в описании поля, но это «другая песня»).

В C# **sealed** в заголовке виртуального метода означает запрет перегрузки (англ. override) метода в производных классах. В заголовке класса — запрет наследования от данного класса.

В Java **final** в заголовке метода означает, запрет замещения (скрытия, англ. hide) или перегрузки (англ. override) метода в производных классах. В заголовке класса — запрет наследования от данного класса.

ADA

Наследование в Аде происходит путем создания нового пакета и расширения в нем базовой записи:

```
package Person is
   type Object is tagged private;
   procedure Put (O : Object);
private
   type Object is
   record
      Name   : String (1 .. 10);
      Gender : Gender_Type;
```

---

[10] Вероятно, только перед определением типа.

```
   end record;
end Person;

with Person;
package Programmer is
   type Object is new Person.Object with private;
   private
      type Object is
      record
         abuility   : Language_List;
      end record;
end Person;
```

ООП в Аде является вполне полноценным с Динамическим полиморфизмом, RTTI, абстрактными типами и интерфейсами.

Динамический полиморфизм

В объектно-ориентированных языках программирования динамический полиморфизм реализуется с помощью наследования классов и виртуальных функций и/или виртуальных членов. Класс-потомок наследует сигнатуры методов класса-родителя, а реализация, в результате переопределения метода, этих методов может быть другой, соответствующей специфике класса-потомка. Другие функции могут работать с объектом как с экземпляром класса-родителя, но если при этом объект на самом деле является экземпляром класса-потомка, то во время исполнения будет вызван метод, переопределенный в классе-потомке. Это называется поздним связыванием.

Позднее связывание есть в C++, C#, Java, Delphi, Ада 95, Оберон-2.

C#

В **C#** к динамическому полиморфизму имеют отношение 3 важных ключевых слова (модификатора):

1. **virtual**

2. **override**

3. **new**

virtual

Ключевое слово **virtual** используется для изменения объявлений методов, свойств, индексаторов и событий и разрешения их переопределения в производном классе. Например, этот метод может быть переопределен любым производным классом. Модификатор **virtual** нельзя использовать с модификаторами **static, abstract, private** или **override.**

```
public virtual double Area()
{
    return x * y;
}
```

override

Модификатор **override** требуется для расширения или изменения абстрактной или виртуальной реализации унаследованного метода, свойства, индексатора или события. Иными словами, модификатор **override** *расширяет* метод базового класса. Метод, переопределенный с использованием **override,** называется переопределенным базовым методом. Переопределенный базовый метод должен иметь ту же сигнатуру, что и метод **override.** Невиртуальный или статический метод нельзя переопределить. Переопределенный базовый метод должен иметь тип **virtual, abstract** или **override.** Объявление **override** не может изменить уровень доступа метода **virtual.** Методы **override** и **virtual** должны иметь одинаковый модификатор уровня доступа. Модификаторы **new, static** и **virtual** нельзя использовать для изменения метода **override.** Переопределяющее объявление свойства должно задавать такие же модификаторы уровня доступа, тип и имя, как и имеющиеся у унаследованного свойства, а переопределенное свойство должно иметь тип **virtual, abstract** или **override.**

```
class BaseClass
{
    public virtual void Method1()
    {
        Console.WriteLine("Base - Method1");
    }
    public void Method2()
    {
        Console.WriteLine("Base - Method2");
    }
}
class DerivedClass : BaseClass
{
    public override void Method1()
    {
        Console.WriteLine("Derived - Method1");
    }
    public void Method2()
    {
        Console.WriteLine("Derived - Method2");
    }
}
// ...
BaseClass bc = new BaseClass();
DerivedClass dc = new DerivedClass();
BaseClass bcdc = new DerivedClass();
bc.Method1();
```

```
bc.Method2();
dc.Method1();
dc.Method2();
bcdc.Method1();
bcdc.Method2();
// Output:
// Base - Method1
// Base - Method2
// Derived - Method1
// Derived - Method2
// Derived - Method1
// Base - Method2
```

new

Модификатор **new** *скрывает* члены, унаследованные от базового класса. При сокрытии унаследованного члена его производная версия заменяет версию базового класса. (На самом деле, члены можно скрыть и без модификатора **new,** но в результате возникнет предупреждение. Если же для явного скрытия члена используется **new,** то модификатор отключает вывод предупреждений и документирует тот факт, что производная версия предназначена для замены.

```
class BaseClass
{
    public void Method1()
    {
        Console.WriteLine("Base - Method1");
    }
    public void Method2()
    {
        Console.WriteLine("Base - Method2");
    }
}
class DerivedClass : BaseClass
{
    public override void Method1()
    {
        Console.WriteLine("Derived - Method1");
    }
    public new void Method2()
    {
        Console.WriteLine("Derived - Method2");
    }
}
// ...
BaseClass bc = new BaseClass();
DerivedClass dc = new DerivedClass();
BaseClass bcdc = new DerivedClass();
bc.Method1();
bc.Method2();
```

```
dc.Method1();
dc.Method2();
bcdc.Method1();
bcdc.Method2();
// Output:
// Base - Method1
// Base - Method2
// Base - Method1
// Derived - Method2
// Base - Method1
// Base - Method2
```

Абстрактные типы данных, классы и интерфейсы

Абстрактный тип данных (АТД) — это тип с полностью инкапсулированной структурой. Использовать объекты АТД возможно только при помощи явно определенных в интерфейсе типа операций. Абстрактный класс (АК) — это класс, содержащий хотя бы один абстрактный метод. Он предназначен только для того, чтобы быть базовым классом.

Прямой связи между АК и АТД нет. АТД может быть абстрактным классом, а может и не быть. Аналогично, АК может иметь инкапсулированную структуру, а может и не иметь.

Абстрактный класс

В объектно-ориентированных языках программирования абстрактный класс реализуются следующими тремя способами

1. Модификатор abstract перед классом. Используется в C# и Java.

2. Класс содержит хотя бы один абстрактный метод. В C# и Java абстрактный метод обозначается модификатором abstract перед объявлением метода. В C++ абстрактные методы называются чистыми виртуальными. Примеры:

```
// тело отсутствует
virtual void draw() = 0; // C++
abstract void draw(); // Java, C#
```

3. Если в классе, производном от абстрактного класса с абстрактными методами, не замещен хотя бы один абстрактный метод, то класс тоже является абстрактным. В C# и Java незамещенные абстрактные методы должны быть явно объявлены как абстрактные.

В языке C# абстрактными могут быть и свойства:

```
abstract int Length { get;}
```

Пример абстрактного класса (**Java** или **C#**):

```
abstract class ShapesClass
{
```

```
    abstract public int Area();
}
```

Абстрактные методы (функции) есть в C++, Java, Delphi, C#, Ада 95.

Абстрактный тип данных

В большинстве языков абстрактный тип данных реализуется с помощью интерфейсов. См. пример в соответствующем разделе.

В Ада используется ключевое слово **limited:**

```
type Stack is limited private;
```

В ОБЕРОН И ОБЕРОН-2 АТД позволяют открывать поля структуры.

Пример абстрактного ТД и абстрактных функций в Ада95

Мы продемонстрируем абстрактные классы, описывая несколько реализаций одной и той же абстракции; абстрактный класс будет определять структуру данных Set, и производный класс — реализовывать множество в виде булевого массива. В языке Ada 95 слово abstract обозначает абстрактный тип и абстрактные подпрограммы, связанные с этим типом:

```
package Set_Package '''is '''
    type Set is abstract tagged null record;
    function Union(S1, S2: Set) return Set is abstract;
    function Intersection(S1, S2: Set) return Set is abstract;
end Set_Package;
```

Вы не можете объявить объект абстрактного типа и не можете вызвать абстрактную подпрограмму. Тип служит только каркасом для порождения конкретных типов, а подпрограммы должны замещаться конкретными подпрограммами.

Рассмотрим производный тип, в котором множество представлено булевым массивом:

```
with Set_Package;
package Bit_Set_Package is
    type Set is new Set_Package.Set with private;
    function Union(S1, S2: Set) return Set;
    function Intersection(S1, S2: Set) return Set;
private
    type Bit_Array is array(1..100) of Boolean;
    type Set is new Set_Package.Set with
        record
            Data: Bit_Array;
        end record;
end Bit_Set_Package;
```

Конечно, необходимо тело пакета, чтобы реализовать операции.

Интерфейс

Интерфейс состоит только из абстрактных методов. В нем нет реализации методов (как нет и невиртуальных методов), нет нестатических полей (статические поля, например константы, допустимы).

Пример моделирования интерфейса на C++:

```
class Set {
public:
    virtual void Incl(T & x) = 0
    virtual void Excl(T & x) = 0
    virtual bool IsIn(T & x) = 0

    virtual ~Set () {} // деструктор
};
```

Явная и неявная реализация интерфейса

Явная реализация интерфейса означает, что вызов метода интерфейса может происходить только через ссылку на интерфейс, но не может происходить через ссылку на класс, реализующий интерфейс. Перед вызовом интерфейсного метода необходимо явно преобразовать ссылку на объект реализующего класса к ссылке на интерфейс. Концепция явной реализации полезна, например, при конфликте имен между унаследованными интерфейсами. Используется только в C#. Явная реализация всегда статична, неявная же может быть виртуальной. Неявная реализация может быть абстрактной и реализовываться только в классе-потомке. Явная реализация не может быть абстрактной, но сам класс может иметь другие абстрактные методы и сам быть абстрактным.

```
interface ISomeInterface
{
    void F();
}
class CoClass: ISomeInterface
{
    void ISomeInterface.F() { //в случае неявного интерфейса, здесь стояло бы
 public void F() {
        System.Console.WriteLine("Явно реализованный метод");
    }
// ...
}
// ...
CoClass c = new CoClass();

c.F(); // ошибка: нельзя вызывать явно реализованный метод
       // интерфейса через ссылку на объект

((ISomeInterface)c).F(); // все нормально
```

Множественное наследование

Полностью реализовано только в C++. В Ada, C#, Delphi, Java множественное наследование поддерживается только для интерфейсов[11].

Динамическая идентификация типа

Динамическая идентификация типа данных (Run-time type identification, RTTI) — механизм, который позволяет определить тип данных переменной или объекта во время выполнения программы.

C++

В C++ для динамической идентификации типов применяются операторы dynamic_cast и typeid (определён в файле typeinfo.h), для использования которых информацию о типах во время выполнения обычно необходимо добавить через опции компилятора при компиляции модуля.

Оператор dynamic_cast пытается выполнить приведение к указанному типу с проверкой. Целевой тип операции должен быть типом указателя, ссылки или void*.

Оператор typeid возвращает ссылку на структуру type_info, которая содержит поля, позволяющие получить информацию о типе.

Delphi

Компилятор Delphi сохраняет в исполняемом файле программы информацию обо всех классах, используемых в ней. При создании любого объекта в памяти перед ним располагается заголовок, в котором есть в том числе ссылка на структуру-описатель класса этого объекта. Встроенные в язык функции работают с этой информацией прозрачно для программиста. Оператор is позволяет проверить, является ли объект или тип наследником определённого типа, а оператор as является аналогом dynamic_cast в C++.

C#

В C# для определения типа объекта во время исполнения используется метод GetType, а также ключевые слова is и as, которые являются аналогами для typeid и dynamic_cast в C++ соответственно.

Оберон-2

В Оберон-2 есть два средства для идентификации типа: операция IS и охрана типа.

---

[11] По крайней мере в контексте Java более корректно будет говорить не о наследовании (**extends**), а реализации (**implements**) или частичной реализации интерфейса (последнее возможно только для абстрактных классов)

Проверка типа

v IS T означает "динамический тип v есть T (или расширение T)" и называется проверкой типа. Проверка типа применима, если

1. v - параметр-переменная типа запись, или v - указатель, и если

2. T - расширение статического типа v

Охрана типа

Операторы with выполняют последовательность операторов в зависимости от результата проверки типа и применяют охрану типа к каждому вхождению проверяемой переменной внутри этой последовательности операторов. Если v - параметр-переменная типа запись или переменная-указатель, и если ее статический тип T0, оператор

```
WITH v: T1 DO S1 | v: T2 DO S2 ELSE S3 END
```

имеет следующий смысл: если динамический тип v - T1, то выполняется последовательность операторов S1 в которой v воспринимается так, будто она имеет статический тип T1; иначе, если динамический тип v - T2, выполняется S2, где v воспринимается как имеющая статический тип T2; иначе выполняется S3. T1 и T2 должны быть расширениями T0. Если ни одна проверка типа не удовлетворена, а ELSE отсутствует, программа прерывается.

Java

В Java тип объекта может быть получен при помощи метода getClass(), объявленного в классе java.lang.Object и потому реализуемого каждым классом. Для проверки принадлежности объекта определенному типу используется оператор instanceof (obj instanceof SomeClass), он заменяет dynamic_cast из C++. Также принадлежность объекта классу может быть определена с помощью оператора приведения типа, который в случае несоответствия типов выбрасывает исключение ClassCastException.

ADA

В Аде для определения типа существует ключевое слово **in** работающее аналогично **is** в, скажем, Oberon-2. После этого можно приводить типы, пользуясь мощным механизм конвертирования Ады (в аде вместо понятия приведения типов/type casting используется понятие конвертирование типов/type conversion):

```
Derived_Object: Derived := Derived (Base_Object) -- Здесь будет производится проверка в run-time
```

Понятие о родовых объектах. Обобщенное программирование

ADA

Note to C++ programmers: generic units are similar to C++ templates.

Объявляем шаблон:

```
generic
 type Element_T is private;  -- Generic formal type parameter
procedure Swap (X, Y : in out Element_T);
```

Для его использования необходимо создать объект нужного типа:

```
procedure Swap_Integers is new Swap (Integer);
```

Возможна перегрузка:

```
procedure Instance_Swap is new Swap (Float);
procedure Instance_Swap is new Swap (Day_T);
```

Можно так же указывать, какие подпрограммы должны быть определены для обобщенного типа:

```
generic
 type Multiplable_T is private;  -- Generic formal type parameter
 with function "*" ( X, Y: Multiplable_T) return Multiplable_T;
```

Оператор loop определяет повторное выполнение последовательности операторов.

Операторы with выполняют последовательность операторов в зависимости от результата проверки типа и применяют охрану типа к каждому вхождению проверяемой переменной внутри этой последовательности операторов.

C#

В C# используется конструкция where для определения ограничений на параметры родовых (другое название - обобщенных) конструкций. Ее вид: where имя_типа : список_ограничений.

Виды ограничений
интерфейс — означает, что параметр-тип должен реализовывать этот интерфейс;

имя класса (может быть только одно такое ограничение в списке) означает, что параметр-тип должен быть наследником этого класса;

struct или class – означает, что параметр-тип должен быть структурой или классом;

new() - означает, что параметр-тип должен иметь конструктор умолчания (без параметров).

Пример универсального шаблона (generic), реализующего односвязный список.

```
// type parameter T in angle brackets
public class GenericList
{
    // The nested class is also generic on T.
    private class Node
    {
        // T used in non-generic constructor.
        public Node(T t)
        {
            next = null;
            data = t;
        }
        private Node next;
        public Node Next
        {
            get { return next; }
            set { next = value; }
        }
        // T as private member data type.
        private T data;
        // T as return type of property.
        public T Data
        {
            get { return data; }
            set { data = value; }
        }
    }
    private Node head;
   // constructor
    public GenericList()
    {
        head = null;
    }
    // T as method parameter type:
    public void AddHead(T t)
    {
        Node n = new Node(t);
        n.Next = head;
        head = n;
    }
    public IEnumerator GetEnumerator()
    {
        Node current = head;
        while (current != null)
        {
            yield return current.Data;
            current = current.Next;
        }
    }
}
```

Параллельное программирование

ADA

Ада реализует концепцию так называемых задач (**task**), что по сути является синонимом потока. Задача оформляется совершенно аналогично модулю, но может быть объявлена и описана где угодно, даже в теле подпрограммы:

```
package Tasked is
   procedure Tasked_proc is
      task Untyped_task_1 is
         Some declarations    -- Здесь объявляем все так, как будто объявляем
пакет
      end task;
      task Untyped_task_2;    -- Мы можем ничего и не открывать в задаче
внешнему миру
                              -- Мы можем объявлять как отдельные задачи
      task type Task_Type is  -- Так и целые типы задач
            Some declarations
      end Task_Type;

      Typed_task_1, Typed_task_2 : Task_Type;   -- Две одинаковые задачи

      task body Untyped_task_1 is separate; -- Можно описать задачу отдельно
      task body Untyped_task_2 is
         something is separate;                   -- Или часть задачи
отдельно
         begin
            ...      -- Собственно то, что будет делать задача
         end;
      end Untyped_task_2;

      task body Task_Type is
         ...                              -- Тело типа задачи
      end Task_Type;

                     -- Тут начинают работать наши 4 задачи параллельно:
Untyped_task_1, Untyped_task_2, Typed_task_1, Typed_task_2
                     -- На самом деле их 5, так как главный процесс
рассматривается как неявная задача
   begin      -- Tasked_proc
      Do_Something;
      ...
            -- Здесь задача, запустившая потомков будет ожидать их завершения
   end Tasked_proc;
end Tasked;
```

В Аде есть множество методов синхронизации задач. Рассмотрим один из них - *entry*. Это что-то на подобии точек входа в задачу извне. По своему объявлению, это - самые обычные процедуры, однако описываются и работают они по-другому. Когда

одна задача хочет запросить у другой какую-либо услугу, она вызывает **entry**, которая реализовывает данную услугу. В теле задачи (непосредственно в выполняемом коде), реализующей услугу есть участки кода, описывающих тела **entry**. Описание начинается со слова **accept** . Задача, вызвавшая **entry** будет ожидать, когда задача, реализующая этот **entry** дойдет до его тела. Этот момент называется **Rendezvou** (рандеву, свидание). Если несколько задач, одновременно (это слово, естественно, не стоит воспринимать буквально) вызвали **entry**, то они становятся в очередь (тут уже речь идет, наверное, о названии Orgy xD). Как это все выглядит:

```
task type Service is
   type Some_Type is range -10 .. 10;
   entry Plus ( X: in Some_Type );
   entry Minus ( X: in Some_Type );
   entry Get ( X: out Some_Type ); -- Повторюсь, entry - те же процедуры у
которых могут быть in/out параметры.
end Service;
```

Кроме того для того чтобы задача не простаивала, тела **accept** можно объединить в блок **select**, который будет выбирать запрошенные entry,

```
task body Service is
   Some_Var: Some_Type;
begin
   Some_Var := 0;
   loop

      accept Get ( X: out Some_Type ) do ''--
 Здесь будем простаивать, пока кто-нибудь не запросит Get;
         X := Some_Var;
      end Get;

      select                                        ''--
 Здесь либо сразу выполняем одно из запрошенных (Plus, Minus), либо ждем, пок
а придет запрос на один из этих entity ''
         accept Plus ( X: in Some_Type ) do
            Some_Var := Some_Var + X;
         end Plus;
      or
         accept Minus ( X: in Some_Type ) do
            Some_Var := Some_Var - X;
         end Minus;
      end select;
   end loop;
end Service;
```

В этом примере, естественно лучше было все 3 **accept** запихнуть в **select** - было бы меньше простоев, но для примера я оставил так.

Modula-2

В языке Modula-2 есть низкоуровневый механизм *сопрограмм*.

Отличия сопрограммы от процесса:

i. Известно, что сопрограммы выполняются квазипараллельно. Следовательно, их использование исключает трудную проблему взаимодействия истинно параллельных процессов.
ii. Переключение процессора от одной сопрограммы к другой осуществляется явным *оператором передачи управления*. Выполнение сопрограммы, которой передаётся управление, возобновляется с той точки, где она была приостановлена последним таким оператором.

Примеры кода

Примеры кода на Java

```
package MyPackage; // Всё, что будет написано в этом файле, считается лежащим
 в этом пакете.
import java.lang.*; // Импорт пакетов.

public class MyClass
{
    private static int m_i = 3;
    private static int m_j;
    static {
        // Демонстрация статического блока в Java. Операторы в этом блоке буд
ут выполнены в самом начале программы.
        m_j = m_i * 4;
    }
    public static void main(String[] args)
    {
        final int N = 5; // final — аналог констант в языках C/C++
        final int M = 3;
        double a[][] = new double[N][M];
        double [][]b = new double[N][M];
        for (int i = 0; i < a.length; ++i) {
            for (int j = 0; j < a[i].length; ++j) {
                a[i][j] = i * j;
            }
        }
        for (int i = 0; i < b.length; ++i) {
            for (int j = 0; j < b[i].length; ++j) {
                b[i][j] = i + j;
            }
        }
        double res[][] = sumMatrices(a, b);
        for (int i = 0; res != null && i < res.length; ++i) {
```

```
            for (int j = 0; res[i] != null && j < res[i].length; ++j) {
                System.out.print(res[i][j] + " ");
            }
            System.out.println();
        }
    }
    public static double[][] sumMatrices(double a[][], double b[][])
    {
        label1: if (a != null && b != null && a.length == b.length) {
            label2: {
                for (double x[] : a) { // аналог for-each
                    for (double y[] : b) {
                        if (x.length != y.length) {
                            break label2; // прервать внешний цикл
                        }
                    }
                }
                break label1;
            }
            System.out.println("Bad Arrays...");
            return null;
        } else {
            System.out.println("Bad Arrays...");
            return null;
        }
        double [][]res = new double[a.length][a[0].length];
        for (int i = 0; i < a.length; ++i) {
            for (int j = 0; j < a[i].length; ++j) {
                res[i][j] = a[i][j] + b[i][j];
            }
        }
        return res;
    }
}
```

```
package MyPackage;

import java.sql.SQLException;

public class Example
{
    public static void main(String [] argv)
    {
        // демонстрация работы внутренних классов:
        Outer outer = new Outer(10);
        outer.test();
        // демонстрация работы динамической диспетчеризации методов:
        A refA;
        refA = new A(5);
        refA.callMe();
```

```
        refA = new B(10, 15);
        refA.callMe();
        // демонстрация работы с абстрактными классами:

        //AbstrA abstra = new AbstrA(5); – нельзя, т.к. AbstrA –
 абстрактный класс.
        AbstrA refAbstrA;
        refAbstrA = new AbstrB(5, 10);
        refAbstrA.callMe();
        // демонстрация работы с интерфейсами:
        Client client = new Client();
        client.someFunc();
        // демонстрация работы с исключениями:
        try {
            client.badFuncEx();
            // Мы не можем оставить потенциальную посылку функцией исключения
 без внимания:
            // либо окружим её вызов конструкцией try-catch,
            // либо напишем, что эта функция тоже может бросать исключения (б
ольшое отличие от языка C++).
        } catch (Exception ex) { // Ловушка для всех исключений, ибо Exceptio
n – это базовый класс.
            //do Smth...
        }
    }
}

class Outer
{
    private int x = 0;
    public Outer(int x)
    {
        this.x = x;
    }
    private void display()
    {
        System.out.println(x);
    }
    public void test()
    {
        Inner in = new Inner();
        in.useOuterFunc();
    }
    class Inner
    {
        public void useOuterFunc()
        {
            // Внутренний класс имеет непосредственный доступ к приватным и п
убличным полям внешнего класса.
            // Обратное неверно.
```

```
            display();
            x = 15;
            display();
        }
    }
}

// Реализация динамической диспетчеризации методов:
class A
{
    private int i;
    A(int i)
    {
        this.i = i;
    }
    public void callMe()
    {
        System.out.println("In A.callMe(). " + i);
    }
}

final class B extends A
// final означает, что никакой класс не сможет отнаследоваться от B.
// final, стоящий перед описанием метода запрещает его переопределение.
{
    private int i; // Поле i класса B перекрывает поле i суперкласса.
    B(int i, int j)
    {
        super(i); // вызывается метод A(int i) суперкласса.
        this.i = j;
    }
    public void callMe()
    // Сигнатура функций должна совпадать. В противном же случае –
 это обычная перегрузка.
    // Сам метод называется переопределённым (аналог виртуальным методам в C+
+).
    {
        System.out.println("In B.callMe(). " + i); // Здесь выведется значени
е поля подкласса.
        // Для доступа к полю суперкласа нужно использовать super.i (но здесь
 поле недоступно, ибо private).
    }
}

// Использование абстрактных классов. Если мы объявляем какой-
то метод в классе абстрактным, то
// автоматически должны сделать абстрактным и сам класс. С точки зрения динам
ической диспетчеризации это означает лишь то,
// что подклассы должны будут явно реализовать все абстрактные методы или тож
е быть абстрактными.
```

```
abstract class AbstrA
{
    private int i;
    AbstrA(int i)
    {
        this.i = i;
    }
    abstract public void callMe();
}

final class AbstrB extends AbstrA
{
    private int i;
    AbstrB(int i, int j)
    {
        super(i);
        this.i = j;
    }
    public void callMe()
    {
        System.out.println("In AbstrB.callMe(). " + i);
    }
    protected void finalize()
    {
        // Считаю нужным упомянуть о методе finalize. Он должен вызываться пр
и удалении объекта сборщиком мусора.
        // Используется для освобождения занятых ресурсов (закрытия файлов и
т.п.).
        // Однако его поведение настолько туманно, что применять его не стоит
, а позаботиться об освобождении отдельно. :)
        System.out.println("Wow! You're lucky to see it in your console! :D")
;
    }
}

// Использование интерфейсов и исключений. :)
// Все методы и члены интерфейсов по умолчанию делаются публичными. Все члены
 – final static.

interface MyInterface
{
    int pubStatFinalMember = 121;
    // Все переменные должны быть инициализированы.
    void someFunc();
}
/*
Доступ class имя_класса [extends суперкласс]
    [implements интерфейс [, интерфейс...]]
```

```
    //тело класса
*/
class Client implements MyInterface
// В классе должны быть определены все функции интерфейса. Спецификатор досту
па у них обязан быть public.
// Если в классе определяются не все функции интерфейса, то он должен быть об
ъявлен абстрактным.
// Кроме того интерфейс может быть вложен в класс или же другой интерфейс, а
также интерфейсы можно наследовать.
{
    public void someFunc()
    {
        //pubStatFinalMember = 601; ——— Нельзя, т.к. переменная — final.

        System.out.println("In someFunc() from interface.");
    }
    public void badFuncEx() throws SQLException
    {
        //Здесь же и рассмотрим исключения. :)
        try {
            int i = 0;
            i = pubStatFinalMember / i;
            System.out.println("You shouldn't see it.");
        } catch (ArithmeticException ex) { // Тут мы поймаем исключение делен
ия на нуль
            System.out.println(ex);
            throw new SQLException();
            // Пошлём другое исключение выше. >:D Но в таком случае мы должны
 у функции пометить, что она может
            // разбрасываться исключениями (см. throws список_исключений).
        } finally {
            // Этот блок кода обязательно получит управление после блока try/
catch вне зависимости от исключения.
            System.out.println("You will always see it.");
        }
    }
}
```

Вывод:

```
10
15
In A.callMe(). 5
In B.callMe(). 15
In AbstrB.callMe(). 10
In someFunc() from interface.
java.lang.ArithmeticException: / by zero
You will always see it.
```

Пример кода на Delphi

Описание юнита, наследования и конструкторов

```
unit Unit1;

interface

uses
  Forms, Dialogs, Classes, Controls, StdCtrls;

type
  // Определение родительского класса основанного, по умолчанию, на TObject
  TFruit = class
  public
    name   : string;
    Constructor Create; overload;   // Этот конструктор использует умолчания
    Constructor Create(name : string); overload;
  end;

  // Определение типа потомка
  TApple = class(TFruit)
  public
    diameter : Integer;
  published
    Constructor Create(name : string; diameter : Integer);
  end;

  // Класс формы используемой этим модулем
  TForm1 = class(TForm)
    procedure FormCreate(Sender: TObject);
  end;

var
  Form1: TForm1;

implementation
{$R *.dfm} // Вложение определений формы

// Создание объекта fruit - версия без параметров
constructor TFruit.Create;
begin
  // Сначала выполняется родительский конструктор (TObject)
  inherited;  // Вызов родительского метода Create

  // Теперь устанавливаем имя fruit, по умолчанию
  self.name := 'Fruit';
end;
```

```
// Создание объекта fruit - версия с параметрами
constructor TFruit.Create(name: string);
begin
  // Не сможет выполниться родительский конструктор - параметры отличаются

  // И сохраняем имя fruit
  self.name := name;
end;

// Создание объекта apple
constructor TApple.Create(name: string; diameter : Integer);
begin
  // Сначала выполняется родительский конструктор (TFruit)
  inherited Create(name);  // Вызов родительского метода

  // Теперь сохраняем переданный apple диаметр
  self.diameter := diameter;
end;

// Основная линия кода
procedure TForm1.FormCreate(Sender: TObject);
var
  fruit  : TFruit;
  banana : TFruit;
  apple  : TApple;

begin
  // Создание 3-х различных объектов fruit
  fruit  := TFruit.Create;
  banana := TFruit.Create('Banana');
  apple  := TApple.Create('Pink Lady', 12);

  // Смотрим какие из наших объектов являются fruit
  if fruit  Is TFruit then ShowMessage(fruit.name  +' - fruit');
  if banana Is TFruit then ShowMessage(banana.name +' - fruit');
  if apple  Is TFruit then ShowMessage(apple.name  +' - fruit');

  // Смотрим какие объекты являются apple
  if fruit  Is TApple then ShowMessage(fruit.name   +' - apple');
  if banana Is TApple then ShowMessage(banana.name  +' - apple');
  if apple  Is TApple then ShowMessage(apple.name   +' - apple');
end;
end.
```

```
Fruit - fruit
Banana - fruit
Pink Lady - fruit
Pink Lady - apple
```

Примеры на C++, Ада и Java с использованием шаблонов

Стек:

```
GENERIC
    TYPE T IS PRIVATE; SIZE : INTEGER;
PACKAGE Stacks IS
    TYPE Stack IS LIMITED PRIVATE;
    PROCEDURE Push(S: IN OUT Stack; X : IN T);
    PROCEDURE Pop(S: IN OUT Stack; X : OUT T);
    FUNCTION IsEmpty(S : IN Stack) RETURN BOOLEAN;
    FUNCTION IsFull(S : IN Stack) RETURN BOOLEAN;
PRIVATE
    TYPE Stack is RECORD
        Body : ARRAY (1..SIZE) OF T;
        Top : INTEGER := 1;
    END RECORD;
END Stacks;
```

```
template <typename T, int size> class Stack
{
public:
    Stack() {top = 0;}
    void Push(T x);
    T Pop(T& x);
    bool IsEmpty();
    bool IsFull();
private:
    Stack (const Stack& s);
    T body[N];
    int top;
};
```

Очередь:

```
generic
  type T is private;
  Size : integer;
package G_Queue is
  type Queue is limited private;
  procedure Enqueue(Q: inout Queue; X:T);
  procedure Dequeue(Q: inout Queue; X:T);
  procedure Init(Q: out Queue);
  procedure Destroy(Q: inout Queue);
  function IsFull(Q: Queue);
  function IsEmpty(Q: Queue);
  -- другие процедуры ...
private
  type Queue is record
Left, Right: integer;
     body : array(1..Size) of T;
```

```
  end record;
end G_Queue;
```

Дек:

```
generic
  type T is private;
package G_Deque is
  type Deque is limited private;
  procedure PushRight(Deq: inout Deque; X:T);
  procedure PushLeft(Deq: inout Deque; X:T);
  procedure PopRight(Deq: inout Deque; X: out T);
  procedure PopLeft(Deq: inout Deque; X: out T);
  procedure Init(Deq: out Deque);
  procedure Destroy(Deq: inout Deque);
  function IsFull(Deq: Deque);
  function IsEmpty(Deq: Deque);
  -- другие процедуры ...
private
  type PLink is access;
  type Link is record inf : T; next, prev : PLink; end record;
  type PLink is access Link;
  type Deque is record Left, Right: PLink; end record;
end G_Deque;
```

```
interface IDeque
{
    void PushLeft(T x);
    void PushRight(T x);
    T PopLeft();
    T PopRight();
    bool IsFull();
    bool IsEmpty();
    // другие функции
}
```

Функция перемножения матриц:

```
template  Matrix& MatMult (Matrix& A, Matrix& B);

Matrix b,c;
// конкретизация функции
Matrix a = MatMult(b,c);
```

```
generic
  type T is private;
  with function “+”(x,y:T) return T (<>);
  with function “*”(x,y:T) return T (<>);
  type Matrix is private;
function G_MatMult(A,B: Matrix) return Matrix;
```

Примеры кода на C#

Пример обработки исключений в **C#.**

```
class NegativeValueException : Exception
{
    public NegativeValueException(string message) : base(message)
    {
    }
}
// ...
string path = @"c:\users\public\test.txt";
var file = new System.IO.StreamReader(path);
try
{
    var buffer = file.ReadLine();
    var time = Int32.Parse(buffer);
    if ( time < 0)
    {
        throw new NegativeValueException("Time must be a positive value.");
    }
    MyClassInstance.Foo(time);
}
catch (System.IO.IOException e)
{
    Console.WriteLine("Error reading from {0}. Message = {1}", path, e.Messag
e);
}
catch (NegativeValueException e)
{
    Console.WriteLine("Value error. Message = {0}", e.Message);
}
catch (Exception e)
{
    if (e is ArgumentNullException || e is FormatException || e is OverflowEx
ception)
    {
        Console.WriteLine("Parsing error. Message = {0}", e.Message);
    }
    else
    {
        Console.WriteLine("Oops, something went wrong. Message = {0}", e.Mess
age);
    }
}
finally
{
    if (file != null)
    {
                file.Close();
```

```
    }
}
```

Ключевое слово *event* в **C#** позволяет уменьшить объём кода, необходимого для реализации событийной модели взаимодействия на делегатах. Иначе говоря, это такой синтаксический сахар для упрощения работы с обратными вызовами.

```
using System;
namespace wildert
{
    public class Metronome
    {
        public event TickHandler Tick; // объявляем событие Tick
        public EventArgs e = null;
        public delegate void TickHandler(Metronome m, EventArgs e);
        public void Start()
        {
            while (true)
            {
                System.Threading.Thread.Sleep(3000);
                if (Tick != null)
                {
                    Tick(this, e); // генерируем событие Tick
                }
            }
        }
    }
        public class Listener
        {
            public void Subscribe(Metronome m)
            {
                // регистрируем обработчик события Tick
                m.Tick += new Metronome.TickHandler(HeardIt);
            }
            private void HeardIt(Metronome m, EventArgs e)
            {
                System.Console.WriteLine("HEARD IT");
            }

        }
    class Test
    {
        static void Main()
        {
            Metronome m = new Metronome();
            Listener l = new Listener();
            l.Subscribe(m);
            m.Start();
        }
```

```
    }
}
```

Замечание
Делегат — это тип, который определяет сигнатуру метода. При создании экземпляра делегата можно связать этот экземпляр с любым методом с совместимой сигнатурой. Метод можно запустить (или вызвать) с помощью экземпляра делегата. Делегаты похожи на указатели на функции в C++.[12]

Моделирование приватных типов данных из Ады в C++

```
''-- ADA ''

-- head
package P is
   type T is limited private;
   function Convert (X: in T) return Integer;
end P;

-- body
package body P is
   type T is record
      ...
   end record;
   function Convert (X: in T) return Integer is
      ...
   end Convert;
end P;
```

```
'' //C++''

'' //P.hpp''
class P {
public:
   P();
   operator int () const;
private:
   P(const P&);
   ...
}
```

Эмуляция в Java private и limited private из Ады

В Java нет перегрузки операций, поэтому эмуляция limited невозможна на Java. Эмуляция private реализуется таким же способом, что сверху:

---

[12] Процитировано из msdn.

```
//P.java
class P {
   public P() {
      ...
   }
   public int convert () {
      ...
   }
   private ...
}
```

Итоговая таблица

Если в таблице указан знак вопроса, то либо этого языка не было в списке языков в задании, либо информация отсутствует.

	ANSI/ISO C (1989/1990)	C++98	C#	Java	Pascal	Delphi	Оберон	Оберон-2	Modula-2	Ada83	Ada95
Оператор перехода «goto метка»	есть	есть	есть	нет[13]	есть	есть	нет	нет	нет	есть	есть
Конструкция "свойс	нет	нет	есть	нет	нет	есть	нет	нет	нет	нет	нет

[13] Есть break **label**; и continue **label**;, где **label** ставится с двоеточием перед началом цикла и указывает, какой именно цикл (в случае continue — итерацию какого именно цикла) среди тех, в которые вложен данный оператор, нужно прервать. В Java есть зарезервированное слово goto, но оно не несёт никаких функций — оператора безусловного перехода в языке нет (однако переход осуществить можно, см. 2).

тво" (property)											
Абстрактные методы	нет	есть	есть	есть	нет	есть	нет	нет	нет	нет	есть
Виртуальные методы (a.k.a. динамическое связывание методов)	нет[14]	есть	есть	есть	нет	есть	нет	есть	нет[15]	нет	есть
Перегрузка (overloading)	нет	есть	есть	есть	нет	есть	нет	нет	нет	есть	есть
Исключения (exceptions)[16]	нет	есть	есть	есть	нет	есть	нет	нет	нет	есть	есть

[14] Обычно реализуется через указатели на функции и указатели на структуры с указателями на функции.

[15] Комитет ISO утвердил т.н. «объектное расширение» (OO extension). (См.: стандартизация ISO Modula-2.) В получившемся языке каждый метод виртуален (взято отсюда). Однако в канонической Modula-2 даже наследования-то нет (собственно, это одна из фич Oberon).

[16] Исключения также есть и в Visual Basic.

Конструкция try-finally	нет[17]	нет[18]	есть	есть	нет	есть	нет	нет	нет	нет	нет
Раздельная независимая трансляция	есть	есть	нет	нет	нет	есть[19]	нет	нет	нет	нет	нет
Тип запись (struct, record)	есть	есть[20]	есть	нет	есть	есть	есть	есть	есть	есть	есть
Размеченные объединения	ч/и[21]	и[22]	нет	нет	есть	есть	нет	нет[23]	есть	есть[24]	есть[25]
Перечисление	есть	есть	есть	есть	есть	есть	нет	нет	есть	есть	есть

[17] Имитируется макросами через `goto cleanup;`

[18] Декларируется отстутвие необходимости ввиду налиия RAII; есть в качестве расширения в некоторых реализациях (GCC, MSVC)

[19] С версии 4.0.

[20] Является классом с публичной областью видимости для полей; если не использовать виртуальные методы, близок по использованию к структурам в C.

[21] Обычный **union** + детерминант + ручная проверка

[22] Или как в C, или используя boost::variant

[23] Нет в Оберон; учитывая характер изменений, внесённых в Оберон-2, делаем вывод о том, что размеченных объединений в нём нет.

[24] Есть в Ада; вероятно, есть и в конкретной реализации.

[25]

ислен ия		т ь	т ь	т ь			т	т	ть	т ь	
Запре щение замещ ения метод а в произ вольн ых класса х или наслед ования класса	нет	н е т	ес т ь [26]	ес т ь [27]	нет	нет	не т	не т	не т	н е т	нет
Проце дурно е програ ммиро вание	есть	ес т ь	ес т ь	ес т ь	есть	есть	ес ть	ес ть	ес ть	ес т ь	есть
Модул ьное програ ммиро вание	ч/и	ч / и	и	и	и	и	ес ть	ес ть	ес ть	ес т ь	есть
Вложе нные модул и	нет	ес т ь [28]	ес т ь	н е т (?	нет[29]	нет[30]	не т [31]	не т[32]	ес ть	ес т ь	есть

[26] Используется ключевое слово **sealed**.

[27] Используется ключевое слово **final**.

[28] См. статью «Модули в C++».

[29] Вложенными могут быть только подпрограммы, которые не являются библиотечными модулями.

				)							
АТД	нет	есть	есть	есть	нет	есть	есть	есть	ч/и [33]	есть [34]	есть[35]
Расширяющее программирование	нет	и	и	и	нет	и	есть	есть			
ООП	нет	есть	есть	есть	нет	есть	и	есть	нет	нет	естьСм., например, [https://en.wikibooks.org/wiki/Ada_Programming/Object_Orientation
Компонентное программирование	и	и	и	и	нетСтандарт Паскаля — не поддерживает	w:Компонентный Паскаль и далее — поддерживают.	и	и	и		
Композиционное программиро	нет	нет	нет	нет	нет	нет	ч/и	ч/и			

30

31

32

[33] Делаем структуру, скрыто экспортируем указатель на неё в другой модуль.

[34] См. статью «Абстрактные типы данных в Ada».

35

вание											
Обобщенное программирование	нет	и	и	нет	нет	нет	нет	нет	нет	есть	есть
Параллельное программирование	нет	нет[36]	есть	есть	нет	нет	нет	нет	ч/и?	есть	есть
Рефлексивное программирование	нет	и	есть	и	нет	и	и	и	нет	нет	нет
Передача параметров по значению	есть	есть	есть[37]	есть	есть	есть	есть	есть	есть	есть[38]	есть[39]
Передача параметров	и[40]	и	есть[41]	и	есть[42]	есть[43]	есть[44]	есть[45]	есть[46]	есть[47]	есть[48]

[36] В C++11 появился std::thread. Подробнее.

[37] см. MSDN.

[38] Ключевое слово in (подразумевается по умолчанию).

[39]

[40] Моделируется с помощью указателей.

[41] Модификаторы **ref** и **out**, также см. MSDN.

по ссылке											

ч/и - частично имитируется

и - имитируется

Примечания

---

[42] Ключевое слово var.

[43]

[44]

[45]

[46]

[47] Ключевое слово access.

[48]