Часть 1 (Предисловие)

Спецификация языка

Историческое

Язык Java был разработан Джеймсом Гослингом и Патриком Нотоном (и прочими) в компании Sun Microsystems в 1995. Язык предназначался для написания программ для различной бытовой электроники (обычно оснащаемой различными процессорами с различными форматами опкода, что создает проблему совместимости). В силу специфики своего назначения язык Java был задуман как архитектурно- нейтральный. Эта нейтральность достигается за счет того, что этот язык интерпретируемый. Разработчик языка утверждает, что если для некой платформы реализован интерпретатор Java, то программа на Java будет на ней работать. Конечно, то же можно сказать и в отношении компиллируемых языков, если для целевой платформы реализован компиллятор. Однако, разработка интерпретаторов проще, чем разработка компилляторов. Поэтому язык Java разработан интерпретируемым.

В настоящее время применение языка Java шире, чем изначально. В течении последних лет это один из самых распространенных языков программирования. И применяется он, соответственно, в самых разных областях применения языков программирования. Такой распространенности языка поспособствовали различные исторические (теперь уже именно исторические) процессы. В особенности — развитие сети Интернет и соответствующего ПО (речь идет о разитии Web- программирования). Дело в том, что сеть Интернет объединяет самые разные устройства, и это вновь напоминает о проблеме совместимости. Поэтому язык Java был оснащен средствами для создания Web- приложений, и стал применяться в Web- программировании. В дальнейшем язык Java стал применяться и в остальных сферах применения

языков программирования.

Спецификация

Первая версия языка `Java` вышла в 1995. Это `Java 1.0`. Рассматриваемая версия — это `Java 1.8`. Она выпущена в 2014 (разработчик — теперь компания Oracle).

Часть 2 (ЯПВУ Java)

Об именах

Идентично тому, как в C++.
Отдельно следует оговорить, что в языке Java блоки кода, как обычно, создают область видимости имен.

О переменных

Идентично тому, как в C++. Разве что, имеют место некоторые различия.

Объявление переменных

Идентично тому, как в C++. Разве что, имеют место некоторые различия.
В случае совпадения имен в глобальной и локальной областях видимости не будет никакой маскировки глобальной переменной, а просто произойдет ошибка.
Как обычно, переменная должна быть объявлена до ее использования в исходном коде.
Запись объявления переменных выглядит так:
```
VarType VarName;
```

VarType может принимать следующие значения:

byte	1 байт;
char	2 байт;
short	2 байт;
int	4 байт;
long	8 байт;
float	4 байт;
double	8 байт;

`boolean` _ байт;

Численные типы интерпретируются как знаковые.
Тип `char` является численным.
Тип `boolean` как всегда, принимает значения `true`, `false`; и как всегда,говорить о размере такого операнда не целесообразно. Этот тип не приводим к численным типам, и численные типы тоже не приводимы к нему.

Инициализация переменных

Идентично тому, как в `C++`.

Приведение типов

Как обычно, приведение типов бывает автоматическим, и явным. Автоматическое приведение типов выполняется между численными типами (и притом, когда разрядность приемника позволяет вместить в него источник; притом, если емкость самого типа приемника меньше емкости типа источника, то принято считать, что разрядность приемника не позволяет вместить в него источник; также принято считать, что емкость типов с плавающей точкой больше емкости целочисленных типов). Иначе же, требуется явное приведение типов.
Когда требуется явное приведение типов, и при этом оно не производится, возникает ошибка.
Об автоматическом приведении типов необходимо знать еще следующее. При оперировании с целочисленными типами (точнее — целочисленными типами, менее емкими чем `int`), эти типы сразу же автоматически приводятся к типу `int`; а затем, при необходимости, они приводятся к более емким типам (и не только целочисленным); о необходимости такого приведения однозначно говорит наличие в выражении операнда соответствующего типа. При оперировании с нецелочисленными типами автоматическое приведение типов производится только при необходимости в этом; при необходимости дальнейшего приведения к целочисленным типам это должно выполняться явно —

иначе же возникает ошибка (ведь, как и говорилось выше, емкость целочисленных типов принято считать меньшей).
Явное приведение типов выполняется той же записью, что и в языке C++.
Вообще, механизм, совершающий полезную работу по приведению типов, тот же, что и в C++. В том плане, что приведение типов происходит по соответственным разрядам.
Необходимо помнить, что тип boolean не приводим к численным типам, а они к нему.

Указатели

Указатели в принципе не поддерживаются языком Java.

Составные типы данных

Составные типы данных в принципе не поддерживаются языком Java, так как собственно структур в нем нет, массивы — это объекты класса Array, а строки — это не массивы, а объекты класса String.

Операторы

Идентично тому, как в C++. Разве что, имеют место некоторые различия.
Сразу же следует оговорить применение запятой в исходном коде. В языке Java запятая применяется не так, как в C++. В языке C++ запятая — это оператор, и применяется в выражениях на усмотрение программиста. В языке Java запятая — это не оператор, а просто символ. Поэтому, в языке Java запятая применяется только так, как она и применяется, и только там, где она и применяется. Это рассматривается в данных записях.

Арифметические операторы

Арифметические операторы, как обычно, применимы к операндам численных типов.
Набор этих операторов тот же, что и в C++.
Операторы инкремента и декремента, как обычно, допускают запись в префиксной и постфиксной формах, смысл этих форм — в порядке выполнения.
При использовании арифметических операторов необходимо помнить следующее:
оператор деления выполняет целочисленное деление над целочисленными, и нецелочисленное — над нецелочисленными;
оператор деления по модулю применим как к целочисленным, так и к нецелочисленным — просто при применении к нецелочисленным он возвращает остаток в нецелочисленном виде.

Логические операторы

Логические операторы применимы к операндам типа boolean. И исключительно к ним. Ввиду того, что в языке Java значения типа boolean не приводимы к численным типам, и численные типы — к типу boolean, логические операторы должны применяться с повышенной осторожностью.
Для логических операций предусмотрены следующие знаки:

!	NOT
&	AND
|	OR
^	XOR
&&	CONDITIONAL AND
||	CONDITIONAL OR

Операции NOT, AND, OR, XOR могут быть использованы в составном присваивании.
Условные логические операторы (CONDITIONAL AND, CONDITIONAL OR) исполняются условно — они исполняются при условии истинности первого операнда, иначе же они просто возвра-

щают его значение.

Побитовые операторы

Побитовые операторы, как обычно, применимы к операндам всех целочисленных типов. Набор этих операторов — как и в C++:

!	NOT
&	AND
|	OR
^	XOR
>>	SHIFT RIGHT
<<	SHIFT LEFT

Вдобавок имеется оператор беззнакового сдвига вправо

>>>	SHIFT RIGHT LEADING ZERO

Обычный сдвиг вправо выполняется с заполнением знаком, а беззнаковый — с заполнением нулем. Для сдвига влево нет такого рода альтернативного оператора, так как сдвиг влево и так выполняется с заполнением нулем.
Все побитовые операторы могут быть использованы в составном присваивании, кроме оператора NOT – действительно, этот оператор и так применяется к своему единственному операнду, так что нет смысла использовать его еще и в записи составного присваивания.

Операторы сравнения

Операторы сравнения применимы к операндам численных типов, и к операндам типа `boolean` (хотя для них имеют смысл только операторы `==` и `!=`). Набор этих операторов тот же, что и в языке C++.
Необходимо помнить, что сравниваемые операнды должны быть одного типа, или же типов, приводимых автоматически. Операторы сравнения возвращают значения типа `boolean`.

Приоритет операторов, задаваемых знакими

Идентично тому, как в C++. Поэтому, привожу эти сведения исключительно для справки:
ЗАДАНИЕ ПРИОРИТЕТА;
ЛОГИЧЕСКОЕ И ПОБИТОВОЕ НЕТ, УНАРНЫЕ ПЛЮС- МИНУС, ИНКРЕМЕНТ- ДЕКРЕМЕНТ;
ПРИВЕДЕНИЕ ТИПА;
УМНОЖЕНИЕ- ДЕЛЕНИЕ, СУММА- РАЗНОСТЬ;
СДВИГИ;
ОПЕРАТОРЫ СРАВНЕНИЯ;
ПОБИТОВОЕ И, ПОБИТОВОЕ ИСКЛЮЧАЮЩЕЕ ИЛИ, ПОБИТОВОЕ ИЛИ;
ЛОГИЧЕСКОЕ И, ЛОГИЧЕСКОЕ ИСКЛЮЧАЮЩЕЕ ИЛИ, ЛОГИЧЕСКОЕ ИЛИ;
ТЕРНАРНЫЙ ОПЕРАТОР;
ПРИСВАИВАНИЕ;

Кстати говоря, таким образом выглядит список приоритетов во всех C- образных языках (по крайней мере, на момент написания этих записей). Хотя, разумеется, список имеющихся в них операторов может несколько различаться.

Операторы управления ходом вычислений

Идентично тому, как в C++. Разве что, имеют место некоторые различия.

Оператор goto

Оператор goto отсутствует в языке Java. Разработчик языка аргументирует это борьбой с макаронным кодом.

Операторы управления итерацией

break

Оператор `break` реализован в `Java` в общем идентично тому, как в `C++`. Вдобавок, он позволяет устанавливать метку выхода из цикла. Это делает его похожим на оператор `goto` в языке программирования `C++`. Сама метка ставится той же записью, что и в `C++`. Однако, она не может быть поставлена на произвольный участок кода (как- никак, борьба с макаронным кодом). Метка должна быть поставлена на цикл\функцию\блок кода, что инкапсулирует данный оператор (инкапсулирует либо непосредственно, либо опосредовано другими циклами\функциями\блоками кода). При установке метки она должна прописываться перед целевым оператором цикла\именем функции\блоком кода.
Однако, необходимо иметь ввиду, что при использовании оператора `break` таким способом, переход будет не на саму помеченную структуру исходного кода, а за нее (как- никак, это метка выхода из цикла).

`continue`

Оператор `continue`, в отличии от оператора `break`, применим только в циклах. Этот оператор осуществляет переход к следующей итерации цикла.
Необходимо помнить, что у цикла `for` это секция инкремента, у циклов `while`, и `do- while` это условие. Оператор `continue` также может содержать метку, она ставится по тем же правилам, что и у оператора `break`.

Операторы цикла

Идентично тому, как в `C++`. Разве что, имеют место некоторые различия. Условие этих операторов должно возвращать значение типа `boolean`. Тело этих операторов может быть и пустым.

Операторы цикла. Цикл while

Идентично тому, как в `C++`.

Операторы цикла. Цикл do- while

Идентично тому, как в C++.

Операторы цикла. Цикл for

Идентично тому, как в C++. Точно так же, выражения — секции условия цикла могут быть представлены рядом составляющих, идущих через запятую.
Вдобавок, языком Java поддерживается так называемый усовершенствованный цикл for. Его форма записи:

```
for (VarType VarName : CollectionName) … ;
```

Объявленная в условии цикла переменная служит для перебора заданной коллекции — она последовательно принимает значения ее элементов (передача происходит по значению). Разумеется, она должна иметь тип элементов коллекции. Если коллекция — это многомерный массив, то переменная условия должна объявляться как массив, вложенный в эту коллекцию. В таком случае переменная условия сама будет массивом, и при этом она будет принимать вложенные в коллекцию массивы (это осуществляется самим оператором, и поэтому это нормально; в остальных же случаях работа с массивами производится только поэлементно).
Кстати говоря, усовершенствованный цикл for может использоваться и для перебора параметров метода. Его целесообразно использовать для этого при переменной длине списка параметров. В этой цели он используется следующей записью:

```
RetType FuncName (VarType … VarName0)
{ …
// any code
for (VarType VarName1 : VarName0)
 …
// any code
}
```

Оператор условия if

Идентично тому, как в C++.

Оператор выбора switch

Идентично тому, как в C++. Разве что, имеют место некоторые различия. Условие цикла может возвращать тип byte, char, short, int, перечисление, или объект класса String. Значения, с которыми сравнивается условие, как обычно, могут быть того же типа, или автоматически приводимого.

Часть 3 (ЯПВУ Java. ООП)

О классах

О ссылочных типах

Говоря о языке `Java`, приходится оперировать таким понятием, как ссылочные типы.
Хотя в этом языке указателей (и в том числе ссылок) как таковых нет, в нем есть аналогичное (но не идентичное) средство — ссылочные типы.
Так называемые ссылочные типы — это классы, и все синтаксически сходные с ними средства. Ссылочные типы называются так потому, что имена их экземпляров используются для ссылки на них (то есть, для доступа к ним). Таким образом, обнаруживается дуализм в понятии «имя экземпляра ссылочного типа»: с одной стороны — это имя самого экземпляра, с другой стороны — это имя переменной, хранящей ссылку на экземпляр ссылочного типа. Этот дуализм осложняет понимание языка `Java`, однако этот дуализм однозначно есть (и преподносится разработчиком языка без сколь- нибудь негативного отношения).
Все ссылочные типы есть производные от класса `Object` (он располагается так: `java.lang.Object`).

Классы

Классы, как обычно, это классы объектов. А объекты — экземпляры своего класса.
Классы, как обычно, объявляют следующей записью:
```
class ClassName { … }
```

Объекты, как обычно, объявляются следующей записью:
```
ClassName ObjectName;
```

Однако, создание объявленных объектов производится следующей записью:
```
ObjectName = new ClassName( … );
```

Объекты могут быть созданы и при объявлении, тогда запись объявления объекта принимает следующий вид:
```
ClassName ObjectName = new ClassName( … );
```

Как видно из записи, для создания объектов используется конструктор объектов целевого класса. Оператор `new` принимает операнд — конструктор целевого класса. Сам оператор выделяет память под объект. Конструктор осуществляет формирование объекта целевого класса. Оператор `new` завершает свою работу возвратом в выражение ссылки на этот объект — ее значение и присваивается переменной `ObjectName`. Так и происходит создание объектов.
В дальнейшем переменная `ObjectName` может быть реинициализирована ссылкой на другой объект.
Можно и вообще разорвать связь переменной `ObjectName` с каким- либо объектом. Для этого ей нужно присвоить специальное значение `null`. Что касается типа этого значения — то это значение может быть присвоено экземплярам любых ссылочных типов. Вообще, использование любых ссылочных типов как ссылок происходит идентично.

Доступ к членам класса

Доступ к членам класса извне этого класса, как обычно, производится через оператор доступа к членам класса (через точку). Доступ к членам класса изнутри этого класса, как обычно, обходится и без оператора доступа к членам класса.
Доступ к членам класса может регулироваться спецификаторами доступа. Как обычно, применение спецификаторов доступа опционально. Набор спецификаторов доступа тот же, что в `C++`. Их действие — то же, что в `C++`. Различается лишь форма записи, в языке `Java` она следующая:
```
AccSpec VarType VarName;
```

```
/*AccSpec указывается перед объявлением каждого
члена, и без двоеточия*/
```

Завершенные члены класса

Члены класса могут быть объявлены как завершенные. Завершенные члены данных не могут быть реинициализированы (хотя конструкторы все равно могут инициализировать их целевыми значениями), поэтому они должны быть инициализированы при их объявлении:

```
final VarType VarName = … ;
```

если завершенными объявлены параметры метода, то они будут таковыми только в пределах этого метода. Если же завершенными объявлены прочие переменные, то они будут таковыми во всей программе.

Завершенными членами класса могут быть и методы. Завершенные методы не могут быть перегружены. Завершенные методы делаются подставляемыми (это делается автоматически, и на усмотрение компиллятора).

Завершенными членами класса могут быть и вложенные в него классы. Они не могут быть наследуемы; и, кроме того, их методы становятся завершенными.

Статические члены класса

Идентично тому, как в `C++`. Разве что, имеют место некоторые различия.

Статическими могут быть все члены класса — и члены данных, и методы, и вложенные в них классы. Статическим методам непосредственно доступны только статические члены класса. И недоступны ключевые слова `this`, `super`. Нестатические же члены класса доступны им опосредовано — через оператор доступа к членам класса. Это потому, что они находятся в общей для всего класса информации, и оперируют общей для всего класса информацией. Что касается обращения к статическим членам класса извне этого класса — то это производится иной записью, не-

жели в C++:
```
ClassName.VarName
/*то есть,
используется оператор доступа к членам класса,
а не оператор доступа к пространствам имен*/
```

Необходимо помнить, что статические члены класса формируются не в порядке, прописанном в исходном коде, а еще на этапе загрузки программы. Зная это, статические члены класса можно использовать в исходном коде еще до записи создания объектов их класса.

Статические блоки кода

Статическими можно объявить как члены класса, так и блоки кода (то есть, блоки кода внутри методов — ведь весь исполняемый код в языке Java должен храниться внутри методов). Статические блоки кода объявляются следующей записью:
```
static { … }
```

Статические блоки кода исполняются лишь один раз за все время исполнения программы — на этапе ее загрузки. Статические блоки кода целесообразно применять для начальной инициализации статических членов пользовательскими значениями.

О вложенных классах

Классы, вложенные в другие классы, могут быть как статическими, так и нет.
Если вложенный класс статический, то он пребывает в общей для класса информации, и ему непосредственно доступны только статические члены (нестатические доступны опосредованно, через оператор доступа к членам (так как речь идет о нестатических членах — то оператор применяется к членам конкретного объекта)).
Если вложенный класс не является статическим, то ему непосредственно доступны все члены инкапсулирующего его класса. Вложенные классы, не являющиеся статическими, принято назы-

вать внутренними классами.

Локальные классы методов

Идентично тому, как в C++.
На локальные классы методов налагаются те же ограничения, что и на такие же классы в C++:
их методы должны иметь полный прототип внутри класса;
их методы могут обращаться только к глобальным переменным, и статическим локальным переменным (в том методе, где локализован класс);
в самих этих классах нельзя объявить статические переменные;

Необходимо помнить об этих ограничениях.

Операции над объектами

Идентично тому, как в C++. Разве что, имеют место некоторые различия.
Оператор присваивания, как и в C++, применим к объектам одного класса. Однако, при этом необходимо помнить, работа происходит со ссылочными типами. Что и было оговорено выше.
Оператор приведения типа также применим к объектам. Оператор приведения типа, как обычно, указывается перед своим операндом, и запись приведения типа имеет вид:
```
(ClassName) ObjectName;
```

Однако, приведение типа в применении к объектам допускает только приведение к суперклассу или подклассу объекта. Хотя, приведение к суперклассу итак (неявно) выполняется самим языком при возникновении необходимости в этом. Преобразовать один подкласс к другому подклассу того же суперкласса нельзя — иначе все классы (как наследники Object) были бы приводимы друг к другу.
Операторы сравнения также применимы к объектам (хотя, для них имеют смысл только операторы проверки на равенство\неравенство).

Оператор instanceof

Оператор `instanceof` служит для проверки объекта на принадлежность к классу. Он записывается так:
```
ObjectName instanceof ClassName;
```

Этот оператор возвращает логическое значение (истина — если принадлежит, иначе — ложь).

Классы. Методы классов

Методы класса, как обычно, это функции- члены этого класса. В языке `Java` весь исполняемый код должен храниться в классах — а в классах он храниться в методах. Методы класса объявляются идентично тому, как в `C++`:
```
RetType FuncName ( … ) { … }
```

Точно так же, если метод не возвращает никакого значения, то в `RetType` нужно прописать `void`, а в операторе `return` не прописывать никаких операндов. Если метод возвращает объект — то это осуществляется также, как и в `C++`. Если у метода нет параметров, то область параметров остается пустой. Если принимаемый параметр — объект, то передача осуществляется той же записью, что и в `C++`. У метода может быть и неопределенное количество параметров, тогда область параметров принимает вид:
```
(VarType0 VarName0, VarType1 … VarName1)
```

Как видно из записи, у метода может быть произвольное число параметров только одного типа. То есть, в области параметров может быть только одна запись вида `VarType1 … VarName1`. И, кроме того, такая запись должна быть последней в области параметров. Кстати говоря, список параметров переменной длины, заданный в такой записи, принято называть varargs. Переменное количество параметров можно задать и так:
```
(VarType0 VarName0, VarType1 VarName1[])
```

Список параметров в этой записи уже не принято называть varargs. Необходимо помнить, что если список параметров задан именно так, то метод принимает его в виде массива (то есть объекта).
Что еще следует знать опередаче параметров:
переменные элементарных типов данных передаются методам по значению;
объекты передаются методам по ссылке — сама форма записи предусматривает это;
однако, принято считать, что и объекты передаются по значению — разработчик языка аргументирует это следующим — при передаче объектов передается ссылка на них, а она передается по значению.

Рекурсия методов

Идентично тому, как в C++.

Ключевое слово this

Хотя в языке Java нет указателей, методы объекта могут ссылаться на свой объект посредством ключевого слова this. Оно может быть полезно когда имя конкретного объекта еще не известно (например, в конструкторе).

Маскировка членов класса

Коллизии имен в языке Java приводят к ошибке.
Однако, при использовании классов действует исключение из этого правила, и оно заключается в следующем:
если внутри класса происходит коллизия имен его членов и имен параметров его методов, то эта коллизия разрешается следующим образом — непосредственно внутри класса происходит работа с его членами, а внутри метода класса их маскируют его параметры;

Именно так и происходит маскировка членов класса. Регулиро-

вать обращение к замаскированным\не замаскированным переменным можно применением\не применением их уточненного имени (в уточненном имени может быть прописано имя объекта, или ключевое слово `this`).

О сигнатуре методов

Сигнатура метода — это его тип возвращаемого значения и тип параметров.
В отношении перегрузки методов это понятие сужается до типа параметров — ведь именно по параметрам и происходит разрешение перегрузки.
Надо заметить, что понятие сигнатуры метода используется во всех `C-` образных языках.

О перегрузке методов

Идентично тому, как в `C++`. При этом необходимо помнить о следующем :
метод может быть переопределен следующими способами — перегрузкой и собственно переопределением;
при перегрузке сигнатура методов должна различаться, а при собственно переопределении — нет;
перегрузка доступна не только для унаследованных методов, а собственно переопределение методов доступно только для унаследованных методов (но не для методов, объявленных как `static`, или `final`);
при перегрузке метод может быть прописан с любым спецификатором доступа, а при собственно переопределении нельзя задавать более строгий спецификатор доступа, чем это прописано у наследуемого метода (таким образом, приходим к следующим фактам: спецификатор `protected` нельзя прописать для метода, у которого он уже прописан — так как явное указание считается избыточным, а более строгий спецификатор не допустим; спецификатор `private` нельзя прописать вообще — так как более строгий спецификатор не допустим, а сами `private-` члены не наследуются; то есть, из всех спецификаторов доступа

при переопределении может быть явно прописан только спецификатор `public`);
при перегрузке можно вводить какие- либо проверяемые исключения, а при собственно переопределении нельзя;

Хотя, перегрузка и собственно переопределение — это разные понятия, они зачастую используются как полные синонимы. Если (далее в этих записях) они используются не как полные синонимы, то это будет оговорено явно.

Пользовательские конструкторы

Как обычно, тело конструктора может быть любой последовательностью операторов, несмотря на то, что конструктор — это специфическое средство языка, решающее свою специфическую задачу. И, как обычно, конструктор имеет имя своего класса; и в его объявлении не допускается прописывать тип возвращаемого значения.
Необходимо помнить, что если определен пользовательский конструктор, то в дальнейшем будет использоваться именно пользовательский конструктор. То есть, пользовательский конструктор не перегружает конструктор по умолчанию (дополняя его перегруженной версией — самим собой), а заменяет его.
Также необходимо помнить, что конструктор не может быть объявлен завершенным — действительно, завершенность бы ограничила его дальнейшее применение.

О порядке вызова конструкторов

При создании объекта, состоящего в той или иной наследственной иерархии, происходит вызов его конструктора.
С точки зрения реализации языка этот процесс не тривиален. Поэтому он происходит поэтапно — сначала вызываются конструкторы его суперклассов, а затем конструктор данного класса. Именно в таком порядке — конструкторы вызываются в порядке наследования.
Этот процесс происходит именно так, даже если вызов конструк-

тора суперкласса не прописан явно. Однако, если вызов конструктора суперкласса не прописан явно, значит вся полезная работа явно прописывается в коде конструктора данного класса. И тогда его код будет больше, и будет делать лишнюю работу — то, что и так делают конструкторы суперклассов.

Перегрузка конструкторов

Перегрузка конструкторов осуществляется так же, как и перегрузка методов.

Классы. Наследование классов

В языке Java наследуемый класс принято называть суперклассом, а наследующий класс — подклассом.
Подклассы тоже могут быть наследуемыми своими подклассами, и производные от них — тоже, и так далее.
Для одного подкласса лишь один класс может быть суперклассом. Об этом принято говорить так — множественное наследование языком не поддерживается.
Класс, наследующий от другого класса, объявляется так:

```
class DerivedClassName extends BasicClassName
{ … } // собственные члены подкласса
```

Как видно из записи, в языке Java нет никаких спецификаторов наследования — их функцию целиком и полностью выполняют спецификаторы доступа (их набор тот же, что в C++, и действие тоже). Сам механизм наследования работает (по части полезной работы) так же, как в C++.

Ссылка на объект подкласса

Переменная суперкласса может ссылаться на объект подкласса, но по этой ссылке будут доступны лишь унаследованные члены. Методы, унаследованные от суперкласса, и затем перегруженные в подклассе, тоже будут доступны (причем в перегруженном варианте).

Ключевое слово super

Аналогично ключевому слову `this`, однако ключевое слово `super` служит для ссылки на непосредственный суперкласс. В записи оно используется аналогично:

```
super.VarName
```

Конструктор super

Из подкласса можно вызвать конструктор его непосредственного суперкласса, это делается следующей записью:

```
super( … );
```

Если конструктор непосредственного суперкласса вызывается в конструкторе подкласса, то его вызов должен быть первым прописан в коде конструктора подкласса.

Абстрактные классы

Абстрактные классы — это классы, содержащие абстрактные методы. Отдельно следует оговорить следующее — класс есть абстрактный, если он содержит один или более абстрактных методов. Это высказывание итак следует из определения, однако оно должно быть дано именно в этих словах (это необходимо для избежания споров, действительно ведущихся касательно данного понятия).

Абстрактные методы — это методы, абстрактные от реализации (то есть, имеющие лишь объявления, но не определения). Эта абстрактность не позволяет такому методу состояться в конкретной версии без перегрузки. Соответственно, и абстрактный класс не может состояться в конкретных объектах. Объекты могут быть лишь у его подкласса, перегружающего его абстрактные методы. Если же подкласс не производит перегрузку абстрактных методов своего суперкласса, то и сам он остается абстракным классом. Необходимо помнить, что хотя объекты абстрактного класса не могут быть созданы, они все же могут быть объявлены — это позволяет использовать их как ссылки.

Абстрактный класс должен объявляться специфической записью:

```
abstract class ClassName
{ …
abstract RetType FuncName (…);
 … }
```

В этой записи используется ключевое слово `abstract`, в подклассах оно уже не применяется (если только они не абстрактные). Необходимо помнить, что абстрактные методы не могут быть статическими (действительно, зачем включать абстрактные от реализации методы в информацию, общую для всего класса). Конструкторы не могут быть абстрактными (действительно, абстрактные конструкторы просто не смогли бы решать свою характерную задачу).

Классы. Массивы

Все массивы, несмотря на свой специфический синтаксис, являются объектами класса `Array`.
Массивы объявляются следующей записью:
```
VarType VarName [];
```

Альтернативная запись:
```
VarType[] VarName;
```

Массивы можно объявлять и списком:
```
VarType VarName0[], … , VarType VarNameN[];
```

Альтернативная запись также доступна:
```
VarType[] varName0, … , VarNameN;
```

Область индексов в записи объявления задается именно так — она задается пустой. Это потому, что при объявлении массива не происходит его создания (и соответственно выделения памяти). Создаются массивы следующей записью:
```
VarName = new VarType[ … ];
```

Здесь область индексов уже указывает размер массива (как обычно, в количестве элементов). Действительно, чтобы выде-

лить память под массив, необходимо указать количество элементов определенного типа, по этим сведениям и будет происходить выделение памяти.
Инициализация уже созданного массива происходит как обычно — по элементам. Для этого прибегают к тем же записям, что и в C++. При использовании так называемой «записи инициализатора массива» необходимость в операторе `new` отпадает.
Объявление массива и его создание можно произвести в одной записи:

```
VarType VarName[] = new VarType[ … ];
```

Эта запись также имеет альтернативную форму:

```
VarType[] VarName = new VarType[ … ];
```

Языком Java поддерживаются и многомерные массивы, они объявляются следующей записью:

```
VarType VarName[][] … [];
```

При объявлении многомерных массивов альтернативная запись также доступна.
Создаются многомерные массивы следующей записью:

```
VarName = new VarType[ … ][ … ] … [ … ];
```

Для создания многомерных массивов также доступна запись инициализатора массива — просто она принимает очень громоздкий вид.
При объявлении (и создании) многомерного массива достаточно указать лишь первую размерность (первую в записи — она и есть верх иерархии). Для прочих размерностей размер задается лишь при необходимости (то есть в дальнейшем). Это позволяет выделять память динамически — то есть при необходимости, и в необходимых количествах.
Выход за пределы массива проверяется самим языком, в случае выхода возникает ошибка.

Классы. Классы- обертки

Классы- обертки — это классы, поддерживаемые языком для инкапсуляции значений элементарных типов данных. Что, в свою очередь, служит для возможности объектного представления значений элементарных типов данных. В целях беспрепятственного использования этой возможности языком поддерживается автоматическое преобразование между обертками и соответствующими им значениями (непосредственными значениями и переменными) элементарных типов данных. Это преобразование происходит в выражениях в случае необходимости, и производится неявно, самим языком.
Языком поддерживаются следующие оболочки:

```
Byte        оболочка для типа byte
Short       оболочка для типа short
Character   оболочка для типа char
Integer     оболочка для типа int
Long        оболочка для типа long
Float       оболочка для типа float
Double      оболочка для типа double
Boolean     оболочка для типа boolean
```

Ввиду применения классов- оберток обнаруживается следующая особенность — для создания объекта- обертки нет необходимости создавать его явно. Для этого достаточно выполнить присваивание значения (элементарного типа данных) объявленному (еще не созданному) объекту- обертке , и создание этого объекта произойдет автоматически.
И вдобавок, ввиду применения классов- оберток обнаруживается следующее — допустима запись:

```
RetType FuncName()
{return( … )}
```

Где `RetType` и `( … )` являются не однотипными, а типом и его оберткой. То есть, обертки и соответствующие элементарные типы данных взаимно приводимы, даже когда речь идет о типе возвращаемого методом значения (`RetType`) и самом возвра-

щаемом значении ((…)).
И еще, в операторах управления ходом вычислений типы и их обертки могут использоваться так же, как и в выражениях вообще — то есть, как автоматически приводимые типы.
При использовании оберток необходимо помнить следующее: в выражениях обертки автоматически приводимы лишь к соответствующим элементарным типам данных; иначе — требуется явное приведение типов (если при этом оно не выполняется — то, как обычно, возникает ошибка).

Классы. Пакеты

Пакеты — это, собственно, пакеты классов. Это средство, аналогичное пространстранствам имен в C++. Как и пространства имен, пакеты могут образовывать иерархии.
По умолчанию, все классы из файла исходного кода (после компилляции) хранятся в безымянном пакете (хотя каждый класс хранится в отдельном файле). Чтобы все классы из файла исходного кода (после компилляции) оказались в пакете с пользовательским именем, в первой строке исходного кода должна быть запись:

```
package PackageName;
```

Разумеется, классы из разных файлов исходного кода также можно сбить в один пакет (то есть, прописать вышеописанным способом их принадлежность к одному пакету).
Пакеты хранятся в каталогах файловой системы с именем пакета. Иерархии пакетов должна соответствовать иерархия их каталогов. Когда пакеты образуют иерархии первая строка исходного кода принимает вид:

```
package PackageName.NestedPackageName;
```

При этом, имена, образующие иерархию, не должны повторяться. Для доступа к классу, лежащему в пакете, извне этого пакета прибегают к записи:

```
PackageName.ClassName
```

О пути к пакетам

Пакеты должны храниться в файловой системе вышеописанным образом. Это требование продиктовано загрузчиком. Загрузчик ищет пути относительно текущего рабочего каталога. Если же относительно него не удается найти путь, то альтернативные пути берутся из значения переменной `CLASSPATH`.
Путь можно указать и явно — в параметрах загрузчика или компиллятора.

Пакеты и управление доступом

Член класса, объявленный без спецификатора доступа, будет доступен в пределах пакета своего класса.
Член класса, объявленный со спецификатором доступа `public`, будет доступен и вне пакета своего класса.
Член класса, объявленный со спецификатором `protected`, будет доступен в своем классе и наследующих его классах (даже если наследующий его класс будет в другом пакете).
Член класса, объявленный со спецификатором `private`, будет доступен только в своем классе.

Специфика спецификатора public

Этот спецификатор доступа применим не только к членам класса, но и к самим классам (даже не являющимся членами какого-либо класса). Во всем файле исходного кода лишь одному такому классу (не являющемуся подклассом) можно применить этот спецификатор — и при этом имя скомпиллированного файла должно быть именем этого класса.

Импорт пакетов

Как и говорилось выше, доступ к классу в пакете осуществляется через имя пакета. И эта запись в случае глубоко вложенных пакетов может быть очень долгой. Эту запись можно сократить, для этого прибегают к импорту пакетов:

```
import PackageName.ClassName;
```

Эта запись приведет к импорту указанного класса из указанного пакета. Если же необходимо импортировать каждый пакет (из заданного уровня иерархии), то в этой записи имя класса (или соответствующего уровня иерархии) заменяется на символ астериск (символ `*`).
Импорт каждого пакета осуществляется отдельным оператором, и с новой строки. Операторы `import` должны непрерывно следовать друг за другом, а самый первый из них должен быть непосредственно за оператором `package`.
Имеется возможность импортировать из указанного пакета избирательно статические члены, это делается записью:

```
import static PackageName;
```

Классы из импортированных пакетов доступны непосредственно — без всей этой избыточной полной записи. Однако, избыточная запись по прежнему допустима. Более того, она необходима, если имена классов из разных пакетов совпадают.
Импортированные классы и пакеты по прежнему подчиняются правилам доступа (они были рассмотрены выше).

Интерфейсы

Интерфейс — это средство языка, позволяющее создать интерфейс (к членам данных какого- либо класса), абстрактный от реализации. Интерфейс к членам данных — это методы класса; таким образом, интерфейс — это средство, предназначенное для создания абстрактных методов, реализуемых затем в классах. То есть, интерфейсы в языке `Java` в целях применения аналогичны функциям- друзьям в языке `C++`.
Интерфейсы объявляются следующей записью:

```
interface InterfaceName
{ …
RetType FuncName ( … )
 … }
```

Как видно из записи, интерфейсы синтаксически сходны с классами. Необходимо помнить, что создать экземпляр интерфейса невозможно — это потому, что интерфейсы должны допускать абстрактность от реализации. Однако, экземпляр интерфейса может быть объявлен — интерфейсы, являясь ссылочными типами, могут использоваться в качестве ссылок.
Интерфейсы используются (по прямому назначению) специфическим способом — они не создаются в экземплярах, а реализуются в классах.
Так как интерфейсы должны допускать абстрактность от реализации, их методы могут быть объявлены кратко. Если же методы интерфейса объявлены полно (так называемые методы с реализацией по умолчанию), то они должны объявляться с помощью ключевого слова `default`, и запись их объявления принимает следующий вид:

```
default RetType FuncName ( … ) { … }
```

Если методы объявлены с реализацией по умолчанию, то они не требуют последующего определения в классе, реализующем данный интерфейс. Если же метод с реализацией по умолчанию имеет последующее определение в классе, то это определение и используется в данном классе.
Членами интерфеса могут быть как методы, так и члены данных. Все члены данных интерфейса должны объявляться как завершенные и статические.

Статические члены интерфейсов

Статическими членами интерфейсов могут быть как члены данных (в интерфейсах они всегда должны быть статическими), так и методы (как обычные методы, так и с реализацией по умолчанию). Необходимо помнить, что статические методы интерфейсов вызываются аналогично статическим методам классов:

```
InterfaceName.FuncName( … );
```

На статические методы интерфейсов налагаются те же ограничения, что и на статические методы классов.

Реализация интерфейсов

Интерфейсы предназаначены для реализации в классах. Перед реализацией в классах интерфейсы должны быть объявлены.
Один интерфейс может быть реализован в одном или нескольких классах; при этом, в одном классе может быть реализован один или несколько интерфейсов.
Реализация интерфейсов осуществляется следующей записью:

```
class ClassName
implements InterfaceName
{ … }
```

Если класс реализует несколько интерфейсов, то они идут списком, через запятую:

```
class ClassName
implements InterfaceName0, … , InterfaceNameN
```

При реализации интерфейсов их методы должны иметь полные определения в реализующих их классах. Притом, в классах должны иметься определения всех методов интерфейса — и не важно, какие из них будут реально использоваться. Кроме того, эти полные определения методов должны предваряться спецификатором доступа `public`. Это потому, что все члены интерфейсов итак (неявно) объявляются как `public`. Если же при реализации интерфейсов их методы не имеют полных определений в реализующем их классе, то такой класс должен объявляться как абстрактный:

```
abstract class implements InterfaceName
{ … }
```

Вложенные интерфейсы

Вложенные интерфейсы — это интерфейсы, объявленные в других интерфейсах и классах. Ко вложенным интерфейсам за пределами инкапсулирующего их интерфейса\класса обращаются через оператор доступа к членам.
Вложенные интерфейсы могут быть объявлены с любым спецификатором доступа; не вложенные интерфейсы могут быть объ-

явлены только со спецификатором доступа `public`. При этом необходимо помнить, что члены интерфесов (неявно) объявляются как `public`.

Наследование интерфейсов

Наследование доступно интерфейсам — интерфейсы могут наследовать от интерфейсов. Наследование интерфейсов осуществляется следующей записью:

```
interface InterfaceName0 extends InterfaceName1
{ … } // собственные члены производного
```

В случае наследования интерфейсов необходимо помнить, что в реализующем их классе должны быть определены все методы реализуемых интерфейсов.

Перечисления

Перечисления — это соответствующее средство языка (пожалуй, все современные ЯПВУ поддерживают перечисления). Как обычно, экземпляры перечислений представляют собой один из объявленных в списке перечисления пунктов. В языке `Java` пунктом списка перечисления является класс. Классы в списке перечисления (неявно) объявляются как `public static final`. Перечисления объявляются следующей записью:

```
enum EnumName
{ ClassName0, … , ClassNameN; // список перечисл.
VarType VarName;              // члены данных
RetType FuncName( … ) { … }}  // методы
```

Как видно из записи, перечисления синтаксически сходны с классами. Все перечисления наследуют от (служебного класса) `enum` (он располагается так `java.lang.Enum`), однако явное использование наследования для перечислений не доступно. Из записи также видно следующее: перечисления включают в себя список перечисления, и еще могут включать в себя собственные

члены данных, и собственные методы.
Объявленные в перечислении члены данных и методы добавляются в классы из списка перечисления. Это делается самим языком. Необходимо помнить, что несмотря на форму записи, перечисления есть перечисления, а экземпляры перечислений — это, соответственно, экземпляры классов из списка перечисления (и ни что иное).
В записи объявления перечисления можно задать передачу параметров для конструкторов классов из списка перечисления. Эта специфическая возможность реализуется в списке перечисления следующей записью:

```
ClassName0( … ), … , ClassNameN( … );
```

Кроме того, в объявлении перечисления может быть объявлен конструктор классов списка перечисления. Это делается следующей записью:

```
enum EnumName
{ …
EnumName ( … )  { … }
 … }
```

Экземпляры перечисления объявляются следующей записью:

```
EnumName ExemplarName;
```

И создаются следующей записью:

```
ExemplarName = EnumName.ClassName;
```

Экземпляры могут быть созданы и при их объявлении:

```
EnumName ExemlarName = EnumName.ClassName;
```

Отдельно следует оговорить, что создание экземпляров перечислений посредством оператора `new` не допускается.

Ограничения, налагаемые на перечисления

На перечисления налагаются следующие ограничения:
перечисления обладают специфическим синтаксисом (что и было рассмотрено выше);

перечислениям не доступно наследование;
в списке перечисления должны объявляться собственные классы, а не использоваться уже существующие (действительно, перечисления — это специфическое средство языка, так что обычные, уже существующие классы не годятся);
если в объявлении перечисления не указать его собственных членов данных и методов, то классы из списка перечисления будут пустыми;

Обобщения

Обобщение — это класс, обобщающий ряд классов (интерфейсов). Обобщения служат для дальнейшего применения в качестве шаблонов.
Обобщение объявляется следующей записью:

```
class GenClassName<GenType>
{ … } // тело оперирует с классом GenType
```

Экземпляры обобщения объявляются следующей записью:

```
GenClassName<ClassName> ObjectName;
//ClassName это уже конкретный класса
```

Как видно из записи, чтобы объявить экземпляр, нужно конкретизировать обобщающий класс. Объявленный экземпляр будет иметь класс `GenClassName<ClassName>` (по крайней мере, в исходном коде). Экземпляры этого класса можно использовать в исходном коде как экземпляры `ClassName`; кроме того, им доступны как члены `ClassName`, так и собственные члены `GenClassName<ClassName>`. Они даже могут использоваться как экземпляры `ClassName` без явного приведения. Но, необходимо помнить, что запрос их класса вернет класс `ClassName<ClassName>`.
Класс- обобщение может быть конкретизирован в любом конкретном классе, в том числе и в классе- обертке. И использоваться, в том числе, и как этот класс.
Что же касается использования разных обобщенных классов, то

необходимо помнить, что это и есть разные классы (и один из них не может использоваться как другой из них).
Создание экземпляров производится следующей записью:
```
ObjectName = new GenClassName<ClassName> ( … );
```

Как обычно, создать экземпляры можно и при объявлении:
```
GenClassName<ClassName> ObjectName = new
GenClassName<ClassName> ( … );
```

Обобщения могут оперировать как с одним обобщенным классом, так и с несколькими. Тогда запись объявления обобщения принимает вид:
```
class GenClassName<GenType0, … , GenTypeN>;
```

Метасимвольный аргумент типа

Речь идет не об аргументе метода, а об аргументе самого обобщения — то есть `GenType`. Метасимвольный аргумент — это аргумент типа, который подразумевает использование любого конкретного типа, и служит для организации последующей работы с любым конкретным типом. Метасимвольный аргумент типа задается знаком вопроса, и использование его имеет вид:
```
class GenClassName<?>
```

Такой класс- обобщение может быть конкретизирован в любом конкретном классе, если только на конкретизацию не наложено ограничение.

Ограничения конкретизации

При объявлении обобщений можно ограничить их последующую конкретизацию. Конкретизация обобщения может быть ограничена конкретным классом (или же интерфейсом).
Когда последующая конкретизация ограничивается определенным классом и его подклассами, то это называется ограничением конкретизации сверху. Ограничение конкретизации сверху задается следующей записью:
```
Class GenClassName<GenType extends ClassName>;
```

```
//ClassName – конкретный класс
```

Если интерфейсы используются совместно с классом, то запись принимает вид:

```
class GenClassName<GenType extends ClassName & In-
terfaceName0 & … & IntrfaceNameN>;
```

Как видно из записи, допустимо указывать лишь один класс, и целый ряд интерфейсов. В этом смысле ограничение конкретизации походит на наследование.

Обобщения, использующие метасимвольный аргумент, могут быть ограничены в конкретизации как сверху (было рассмотрено выше), так и снизу. Ограничение снизу задается записью:

```
class GenClassName<? super ClassName>;
```

Конкретизация будет ограничена суперклассами данного класса, не включая его самого.

Обобщенные интерфейсы

Обобщенными могут быть как классы, так и интерфейсы. Обобщенные интерфейсы объявляются следующей записью:

```
interface GenInterfaceName<GenType>
{ … } // тело оперирует с классом GenType
```

Как обычно, можно оперировать как с одним обобщенным классом, таки с несколькими — тогда они идут списком.

Обобщенные интерфейсы реализуются следующей записью:

```
class GenClassName<GenType> implements
GenInterfaceName<GenType>
{ … } // тело класса, реализующего интерфейс
```

Затем этот класс конкретизируется — это делается обычной для этого записью. Объявление этого класса (реализующего обобщенный интерфейс) должно включать `GenType` (используемый в интерфейсе), чтобы он мог оперировать с ним. При этом, если на `GenType` в интерфейсе наложено ограничение, то при реализации в классе это ограничение прописывается в области `<>`

класса, и опускается в области <> интерфейса. Но, необходимо помнить, что в записи объявления интерфейса оно разумеется прописывается.
Обобщенный интерфейс может быть реализован и с конкретизацией, тогда реализующий его класс будет оперировать с конкретным (не обобщенным) классом, и никакой обобщенный класс ему не будет необходим. Запись релизации примет вид:

```
class ClassName implements
InterfaceName< … > // … конкретный класс
```

Базовый тип обобщения

В интересах работы с унаследованным кодом (написанным до появления в языке обобщений) в язык введен так называемый базовый (сырой) тип. Использование базового типа обозначается тем, что область <> остается пустой.
Базовый тип используется следующим образом:
объявляется обобщение;
затем обобщение конкретизируется (при возникновении необходимости в этом) в базовом типе, и для этого прибегают к специфической записи (`GenClassName ObjectName = new GenClassName(new ClassName())`), причем, в записи конкретизации базовым типом область <> вовсе не прописывается, как это и видно из записи;
затем экземпляр обобщения используется в исходном коде с явным приведением к типу в котором он конкретизирован;
однако, при конкретизации (с использованием целевого конструктора) объект базового типа может быть использован лишь как объект этого конкретного типа, и к другим типам он не приводим;

При использовании базового типа необходимо помнить, что использоваться он должен только для сращивания унаследованного кода с современным (такова задумка разработчика языка). Использование базового типа в других целях порицается разработчиком языка.

Особенности иерархии обобщенных классов

Как и обычные классы, обобщения могут образовывать наследственные иерархии.
Вслучае иерархии, подклассы должны содержать обобщенные типы своих суперклассов — чтобы иметь возможность их конкретизировать, и самим состояться в конкретизации. Разумеется, подклассы в этой иерархии могут вводить и собственные обобщенные типы.
Иерархии обобщенных классов могут венчаться и обычным классом (не обобщением).

Явное приведение типа для обобщений

Явное приведение типа для обобщений ограничено — обобщение можно явно привести лишь к их конкретному типу их же обобщения.

О последующем стирании типов

При использовании обобщений (после компилляции в байт- код) происходит так называемое стирание типов. Стирание типов - это стирание сведений об обобщениях. С точки зрения байт-кода сведения об обобщениях избыточны, и поэтому они в последствии стираются.
Стирание типов происходит так:
обобщающий тип приводится к ограничивающему его типу (если ограничения не прописаны явно, то ограничивающим типом считается `Object`);
затем обобщающий тип приводится к конкретизирующему типу, и само обобщение приводится к этому классу (конкретизирующему типу);

Таким образом, при работе с обобщениями целесообразно помнить, что с точки зрения байт- кода обобщений вовсе не существует, и обобщения являются средством языка, имеющим смысл только с точки зрения исходного кода.

Обобщенные методы

Строго говоря, обобщенным может быть как метод обобщенного, так и метод обычного класса. Обобщенные методы обычных классов объявляются следующей записью:

```
<GenClassName, … > RetType FuncName( … )
{ … } // тело оперирует с GenClassName
```

Такие методы конкретизируются при вызове:

```
FuncName( … );
```

В этой записи отсутствует область <>. Классическая запись, где присутствует область <> также доступна, и имеет вид:

```
< … > FuncName( … );
```

О перегрузке методов обобщений

Перегрузка методов обобщений производится так же, как и перегрузка методов обычных классов.

Обобщенные конструкторы

Обобщенными могут быть не только обычные методы, но и конструкторы. Они объявляются следующей записью:

```
class ClassName
{ …
<GenType> ClassName( … ) { … }
 … }
```

Конкретизация, как обычно происходит при вызове:

```
ClassName ObjectName = new ClassName( … );
```

Ограничения, налагаемые на обобщения

При работе с обобщениями необходимо помнить о налагаемых на них ограничениях:
статические члены обобщений не могут оперировать с `GenType`;
нельзя создать массив элементов типа `GenType`;

нельзя создать массив обобщений;
обобщения не могут наследовать от `Throwable`;

Прерывания

Как обычно, выброс прерываний происходит в ходе исполнения программы, либо производится соответствующей записью в исходном коде.
Выброшенные прерывания должны быть обработаны пользовательским обработчиком, иначе же они будут обработаны системным обработчиком (а он просто выводит сообщение об ошибке, и завершает программу).
В языке `Java` прерывания сопоставляются с соответствующими им объектами прерываний. Все эти объекты — это объекты классов, производных от класса `Throwable`.
У класса `Throwable` есть ряд подклассов, в особенности следующие подклассы.

`Exception`

прерывания этого класса предполагаются обрабатываемыми в пользовательском обработчике; у класса `Exception` есть подкласс — `RuntimeException`;

`Error`

прерывания этого подкласса предполагаются обрабатываемыми в исполняющей системе `Java`;

Если выброс прерываний делается записью в исходном коде, то эта запись имеет следующий вид:

```
throw new ClassName( … );
/*ClassName – это должен быть класс Throwable, или же его подкласс*/
```

Выбросы прерываний должны производиться в блоках `try`.

Впринципе, выбросы прерываний могут прописываться и в методах, которые имеют полные определения вне блока try, но вызываются в нем. Если выброс прерывания предполагается из метода, то требуется задать разрешенные классы выбрасываемых прерываний (для классов Error, RuntimeException, и их подклассов разрешается этого не делать). В таком случае полный прототип метода принимает вид:

```
RetType FuncName( … )
throws ClassName0, … , ClassNameN
{ … }
```

В общем, язык Java обладает теми же средствами работы с прерываниями, что и язык C++. Хотя, имеются некоторые различия.

В языке Java структура кода, осуществляющая работу с прерываниями имеет следующий вид:

```
try { … }
/* блок try */
catch( … ) { … }
/* пользовательский обработчик */
 …
catch( … ) { … }
/* очередной пользовательский обработчик */
finally { … }
/* блок finally выполняется после блока try, причем независимо от выполнения каких- либо блоков пользовательских обработчиков; этот блок предназначен для выполнения каких- либо последующих действий после выполнения каких- либо обработчиков; этот блок может использоваться вместо обработчика, если же обработчик присутствует — этот блок опционален */
```

Как обычно, участок кода, в котором предполагается выброс прерываний, должен быть заключен в блок try. Этот блок должен быть именно блоком, а к единственному оператору он не применим. Разумеется, блоков try в исходном коде может быть и несколько. Блоки обработчиков должны следовать непосредственно за этим блоком. Если же заданным блоком try нет

обработчика, то он ищется за тем блоком try, что инкапсулирует данный блок try (эти блоки могут быть вложенными; что касается вложенных блоков — то сами эти блоки, и блоки (образованные телами вложенных в try методов) воспринимаются языком как взаимозаменимые конструкции — за тем исключением, что блоки обработчиков могут располагаться только за блоками try). Если же нет никакого такого инкапсулирующего блока try, либо, так или иначе, нет никакого пользовательского обработчика, то применяется системный обработчик (системный обработчик просто выводит сообщение об ошибке, и завершает программу).
Поскольку все выбрасываемые прерывания имеют сопоставленные им объекты прерываний, то пользовательский обработчик должен иметь вид:

```
catch(ClassName ObjectName)
{ … } // код обработчика
```

По выбросу прерывания тот или иной обработчик вызывается в зависимости от класса объекта выброшенного прерывания. При этом, необходимо помнить о следующем — в языке Java требуется, чтобы обработчики подклассов прерываний были прописаны раньше обработчиков суперклассов прерываний, иначе возникает ошибка.
По исполнении блока обработчика происходит переход на код, расположенный за блоком try. Именно так!

Многократный перехват исключений

Это перехват ряда исключений в одном обработчике. Это средство служит для тех случаев, когда перехват разных исключений требует одних и тех же действий. Таким образом, применение многократного перехвата исключений уменьшает объем исходного кода. Многократный перехват исключений задается следующей записью:

```
catch(ClassName0 | … | ClassNameN)
{ … } // код обработчика
```

try с ресурсами

Блок `try` позволяет автоматически управлять закрытием ресурсов. Под этими ресурсами подразумеваются все потоки, в том числе потоки ввода- вывода. При использовании `try` с автоматическим управлением ресурсами эти ресурсы будут автоматически закрыты по исполнении блока `try`. Блок `try` с ресурсами в записи выглядит так:

```
try(ObjectName0 = new ClassName0(); …; … )
/* ( … ) - это и есть ресурсы */
{ … }
/* тело, оперирующее с объектами — объявленными ресурсами */
```

Список прерываний

Прерывания, поставляемые самим языком, хранятся в пакете `java.lang`. Кроме того, ряд исключений располагается в прочих пакетах. Некоторые из них проверяются на этапе компилляции — это проверяемые исключения, некоторые из них не проверяются на этапе компилляции — это непроверяемые.
Имеются следующие непроверяемые исключения (`java.lang`), производные от `RuntimeException`.

`ArithmeticException`

арифметическая ошибка (деление на ноль, …);

`ArrayIndexOutOfBoundsException`

выход за пределы массива;

`ArrayStoreException`

сохранение в массиве несовместимого типа;

`ClassCastException`

ошибка приведения типа;

`EnumConstantNotPresentException`

использование неопределенного значения перечисления;

`IllegalArgumentException`

передача недопустимого аргумента;

`IllegalMonitorStateException`

недопустимая контрольная операция;

`IllegalStateException`

недопустимое состояние приложения/среды;

`IllegalThreadStateException`

операция не совместима с состоянием потока;

`IndexOutOfBoundsException`

выход за пределы типа;

`NegativeArraySizeException`

отрицательный размер массива;

`NullPointerException`

использование ссылки на ничтоже;

`NumberFormatExcetion`

ошибка преобразования строки в число;

`SecurityException`

нарушение безопасности;

`StringIndexOutOfBounds`

выход за пределы строки;

`TypeNotPresentException`

тип не найден;

`UnsupportedOperationException`

неподдерживаемая операция;

Имеются следующие проверяемые исключения (`java.lang`).

`ClassNotFoundException`

класс не найден;

`CloneNotSupported`

клонирование не поддерживается (`Cloneable` не реал.);

`IllegalAccessException`

ошибка доступа к классу;

`InstantiationException`

создание экземпляра абстрактного класса (интерфейса);

`InterruptedException`

поток исполнения прерван другим потоком;

`NoSuchFieldException`

поле не существует;

`NoSuchMethodException`

метод не существует;

`ReflectiveOperationException`

ошибка, связанная с рефлексией;

Лямбда- выражения

Лямбда- выражения — это специфическое средство языка.
Для его задействования потребуются следующие составляющие — само лямбда- выражение, и функциональный интерфейс (так же именуемый как Single Abstract Method Interface, SAM- Interface, SAM- Type).
Функциональный интерфейс — это специфический интерфейс, объявляющий один единственный абстрактный метод.
Лямбда- выражение дает определение методу функционального интерфейса. При применении лямбда- выражений этот метод может быть вызван непосредственно как член функционального интерфейса — реализация в классах для этого не используется.
Лямбда- выражениямогут переопределять метод функционального интерфейса столько раз, сколько потребуется — это делается просто употреблением этого выражения в целевом участке исходного кода.
Само лямбда- выражение имеет следующие составляющие:
левая составляющая — называется «параметры выражения» — здесь объявляются имена, которые используются в правой составляющей выражения;
лямбда- оператор — это оператор `->` - он соединяет левую и правую составляющие выражения;
правая составляющая — называется «тело выражения» — это полезная нагрузка выражения;

Тело выражения может быть единственным оператором, или блочным. Причем, если тело выражения блочное — то оно счи-

тается самим телом метода, и поэтому оно должно быть полной последовательностью его операторов. Параметры выражения опциональны — действительно, тело выражения может и не оперировать какими- либо новыми именами. Если выражение не содержит новых имен, то они просто не прописываются; но, сама левая составляющая выражения присутствует все равно. Параметры выражения (если их область не пуста) не требуют объявления их типов. Их типы выводятся из параметров метода функционального интерфейса. Но если типы все же прописываются, то они должны прописываться для всех имен.

Запись самого лямбда- выражения имеет вид:

```
(VarName, … ) -> … ;
/* левая и правая составляющие; причем, если в левой составляющей используется лишь один параметр, то можно и без скобок */
```

Запись объявления функционального интерфейса:

```
interface InterfaceName
{ RetTypeFuncName(); }
```

Задействование лямбда- выражения в исходном коде имеет вид:

```
interface InterfaceName {RetType FuncName();}
/* объявляется функциональный интерфейс */
InterfaceName ExemplarName;
/* оъявляется его экземпляр, для него это можно */
ExemplarName = ( … ) -> … ;
/* этой записью осуществляется переопределение метода функционального интерфейса; как и говорилось выше, его в последствии можно и переопределить */
ExemplarName.FuncName( … );
/* этой записью метод вызывается */
ExemplarName = ( … ) -> … ;
/* этой записью осуществляется переопределение; переопределить можно тело; но необходимо помнить, что параметры выражения должны соответствовать по типу, количеству, и имени параметрам метода функционального интерфейса (то есть, параметры могут быть переопределены лишь в плане их значений — передача значений параметров происходит при вызове метода);
```

```
это же касается и возвращаемых типов */
```

Как видно из приведенной записи, параметры выражения — это и есть параметры метода функционального интерфейса. Если они есть у метода — то они должны быть и у выражения.

Применение обобщенных функциональных интерфейсов

Для применения лямбда- выражений могут применяться как обычные функциональные интерфейсы, так и обобщенные функциональные интерфейсы. Когда применяются обобщенные функциональные интерфейсы, то требуется их конкретизация. Их конкретизация происходит в следующей записи:

```
interface InterfaceName< … > { … }
/* объявление */
InterfaceName ExemplarName<ClassName> =
( … ) -> … ;
/* и конкретизация;
причем, конкретизация должна происходить непременно
— в лямбда- выражениях не допустимо оперировать с
обобщенным типом */
```

Лямбда- выражения в параметрах методов

Есть и альтернативный способ использования лямбда- выражений. Этот способ задействуется не при вызове метода из экземпляра функционального интерфейса, а при вызове метода, не являющегося методом функционального интерфейса. К этому методу выдвигаются специфические требования. Он должен вызывать метод экземпляра функционального интерфейса; в его аргументах должен быть экземпляр функционального интерфейса — на место него будет подставлено лямбда- выражение. В исходном коде это будет выглядеть так:

```
interface InterfaceName {RetType FuncName( … );}
// объявлен интерфейс
RetType FuncName(InerfaceName ExemplarName, … )
{ …
ExemplarName.FuncName( … );
```

```
 … }
// объявлен метод
FuncName(( … ) -> …, … );
// вызван метод
```

В этой записи вместо лямбда- выражения методу можно передать и объявленный экземпляр, если ему предварительно присвоено лямбда- выражение.

О захвате переменных

В лямбда- выражениях доступны для использования имена, глобальные относительно выражения. Но, если в лямбда- выражении используются переменные из непосредственно инкапсулирующей его области видимости, то это называется захватом переменных. В этой области видимости оказываются доступны только те переменные, что фактически используются как завершенные (но при этом не обязательно объявленные как завершенные). Это же означает, что переменные из этой области видимости доступны в лямбда- выражении только не для присваивания. Присваивание им, будучи запрещенным языком, называется недопустимым захватом. А использование их без присваивания им называется допустимым захватом. Необходимо помнить, что это правило распространяется только на эту область видимости.

Ссылки на методы

Ссылка на метод позволяет использовать в исходном коде имя метода, не выхывая его. При создании ссылки на метод необходимо помнить, что из области видимости (в которой она создается) должен быть видим функциональный интерфейс.
Ссылки на статические методы создаются следующей записью:
`ClassName::FuncName`

И использование этой ссылки выглядит так:

```
interface InterfaceName
{RetType FuncName0( … );}
// объявлен функциональный интерфейс
```

```
static RetType FuncName1( … ) { … }
//объявлен некий статический метод
static RetType FuncNameN(InterfaceName ExemplarName, … )
{ExemplarName.FuncName0( … );}
//объявлен статический метод,
//принимающий функциональный интерфейс
FuncNameN(ClassName::FuncName1, … );
//используется ссылка на метод,
//она передается методу вместо лямбда- выражения
```

Разумеется, типы возвращаемых значений методов `FuncName1()` и `FuncNameN()` должны быть те же, что у метода `FuncName0()`.

Ссылки на нестатические методы создаются записью:

```
ObjectName::FuncName
```

И используются вышеописанным образом. В принципе, нестатические методы могут быть указаны и как статические (записью `ClassName::FuncName`). Кроме того, можно использовать и метод из суперкласса (записью `super::FuncName`).

Ссылки на обобщенные методы создаются записью:

```
interface InterfaceName< … >
{RetType FuncName0 ( … );}
//объявлен обобщенный функциональный интерфейс,
//а в нем — обобщенный метод
static < … > RetType FuncName( … ) { … }
//оъявлен обобщенный метод в программе,
//а в нем — вызов метода интерфейса
static < … > RetType FuncNameN( … ) { … }
// объявлен еще один обобщенный метод в программе,
//но он уже не вызывает метод интерфейса
FuncNameN(ClassName::< … >FuncName0, … );
// используется ссылка на метод,
//при этом происходит конкретизация,
// если это метод обобщенного класса,
// то запись такова:
// ClassName< … >::FuncName0
```

Ссылки на конструкторы могут быть переданы вместо лямбда-выражения, как и обычные методы.
Кроме того, они могут быть переданы записью:

```
ClassName::new
//VarType::new – запись для массивов
```

Предикаты

Предикаты — это собственно предикаты (в формальной логике это высказывания, значения которых заведомо известны). В яыке `Java` они применяются для проверки исполнения тех или иных участков кода программы. Применять их следует в целях тестирования и отладки.
Предикаты используются следующей записью:

```
assert … ;
// … - выражение
```

Если выражение истинно, то программа просто продолжит исполнение. Если же оно ложно — то произойдет соответствующее исключение.
Оператор `assert` допускает альтернативную запись:

```
assert … :StringOfText;
/* … – выражение, проверяемое на истинность;
StringOfText - строка передаваемая конструктору соответствующего исключения (если оно возникает, то эта строка принимается конструктором, и используется им в сообщении об ошибке)*/
```

Необходимо помнить, что предикаты по умолчанию не проверяются, несмотря на то, что они прописываются в исходном коде (то есть, операторы `assert` не исполняются). Для того, чтобы они проверялись, это надо явно указать (в параметрах командной строки при запуске приложения, о них — в справочниках).

Аннотации

Аннотации — это средство языка, позволяющее включать в исходный код посторонние сведения. Это предусмотрено для применения в средствах разработчика и интерпретаторе Java. Аннотации бывают языковые (поставляемые самим языком), и пользовательские (определяемые пользователем). Языковые аннотации имеют определенные имена, формы записи, и способы применения. О языковых аннотациях — в ТомеII. Пользовательские аннотации применяются с помощью методов работы с аннотациями. Об этих методах — в ТомеII.
Пользовательские аннотации объявляются так:

```
@interface AnnotationName { … }
/* тела аннотаций состоят из прототипов методов
(причем, именно кратких);
эти методы не могут иметь параметров;
эти методы могут возвращать этд, объекты String,
объекты Class, перечисления, массив из вышеперечис-
ленных, аннотации;
эти методы не могут оперировать обобщ.типом;
этим методам не доступен оператор throws */
```

Как видно из записи, аннотации синтаксически сходны с классами (интерфейсами). Все аннотации (неявно) наследуют от интерфейса java.lang.annotation. Явное наследование для аннотаций не доступно; кроме того, невозможно создать экземпляр аннотации.
После объявления аннотации могут быть приписаны к целевому участку исходного кода (и таким образом аннотировать его). Целевым участком исходного кода может быть: объявление класса его членов (и все вложенные в них структуры исходного кода), объявление перечислений, аннотация. Приписывание аннотаций к целевым участкам исходного кода осуществляется записью:

```
@AnnotationName ( … )
/* ( … ) - инициализация членов аннотации */
 …
/* … это целевой участок исходного кода*/
```

При приписывании аннотации к целевому участку исходного кода происходит инициализация ее членов. Инициализация членов аннотации выглядит в записи как инициализация переменных элементарных типов данных:

```
(VarName0 = Value0, … , VarNameN = ValueN)
```

Правила удержания аннотаций

Правила удержания аннотаций определяют, когда аннотации удерживаются, а когда отбрасываются. Эти правила хранятся в перечислении `java.lang.annotation.RetentionPolicy`. Есть следующие правила удержания аннотаций.

`SOURCE`

аннотации с этим правилом удерживаются в исходном коде, а после компилляции отбрасываются; необходимо помнить, что аннотации, приписываемые к локальным переменным, в любом случае следуют именно этому правилу — и удерживаются не далее, чем в исходном коде;

`CLASS`

аннотации с этим правилом удерживаются после компилляции, но при исполнении отбрасываются;

`RUNTIME`

аннотации с этим правилом удерживаются после компилляции, и при исполнении;

Эти правила (опционально) прописываются для каждой аннотации отдельно. Причем, если правило не прописано, то по умолчанию задействуется правило `CLASS`. Если же правило прописывается, то это делается в объявлении аннотации, и следующей записью:

```
@Retention ( … )
/* ( … ) - SOURCE\CLASS\RUNTIME */
```

```
@interface AnnotationName { … }
```

`@Retention` – это языковая аннотация, при помощи нее и прописываются правила.

Значения по умолчанию

Для членов аннотаций можно задать значения по умолчанию, тогда тело аннотации примет вид:

```
{RetType FuncName() default … ;}
/* … - значение */
```

Члены со значением по умолчанию могут быть реинициализированы — при приписывании аннотации.

Аннотации с одним членом

Если тело аннотации состоит из одного единственного члена, то такую аннотацию принято называть одночленной аннотацией. Одночленным аннотациям присущ особый синтаксис — единственный их член должен иметь имя `value`. И их тело будет иметь вид:

```
{RetType value();}
```

В записи приписывания такой аннотации достаточно просто указать значение инициализации ее члена, а запись присваивания для этого будет избыточной (но допустимой). То есть, запись приписывания такой аннотации может иметь вид:

```
@AnnotationName ( … )
/* … - это просто значение инициализации*/
 …
/* … - целевой участок кода */
```

Этот синтаксис доступен для одночленных аннотаций, и не одночленных аннотаций — когда все их члены, кроме одного, объявлены со значениями по умолчанию. И тогда тело не одночленных аннотаций должно иметь вид:

```
{RetType value();
```

```
RetType FuncName0() defoult … ;
 …
RetType FuncNameN() defoult … ;}
```

Если же при приписывании такой аннотации происходит реинициализация членов со значениями по умолчанию, то запись приписывания принимает обычный вид (рассмотрено выше).

Пустые аннотации

Если тело аннотации пусто, то такую аннотацию принято называть аннотацией- маркером. Такая аннотация объявляется простейшей записью:

```
@AnnotationName {}
```

И приписывается записью:

```
@AnnotationName()
 … // целевой участок кода
```

Причем, в этой записи круглые скобки можно и опустить.

Часть 4 (Приложения)

Литералы

Идентично тому, как в C++. Разве что, имеют место некоторые различия.
Численные литералы в шестнадцатеричном виде записываются через регистронезависимый префикс (0X или 0x).
Определение типа литерала, как обычно, осуществляется самим компиллятором языка.
Численные литералы в целочисленном виде по умолчанию считаются int. А в нецелочисленном виде — double. Но программист может и явно задать тип литерала. Для этого надо просто указать за записью литерала соответствующий суффикс — он представляет собой первую литеру имени типа, и он регистронезависим.
Численные литералы в целочисленном виде можно указывать и в двоичном виде — через префикс 0b.
Численные литералы в нецелочисленном виде можно указывать в десятичном или шестнадцатеричном виде. В любой системе счисления они могут быть как в обычном, так и в научном виде. При записи в шестнадцатеричной системе счисления в научном виде запись имеет специфический вид — например, 0x2.1p789, где p (неважно, p или P) обозначает степень числа 2, а не 10.
Все численные литералы допускают верстку их записи при помощи символа _, причем допускается употреблять несколько идущих подряд символов _. Эта возможность предусмотрена просто для наглядности записи.

Эскейп- последовательности

Так же, как и в C++.

О языке и библиотеках

Говоря о языке `Java` следует иметь ввиду — библиотеки, включаемые в стандартную поставку языка, считаются средствами самого языка. Однако средства языка, являющиеся библиотечными, не рассматриваются в этом томе. Сведения о них — в Томе II.

О некоторых средствах языка

Об апплетах

Это средство задействуется применением библиотек, и поэтому сведения о нем — в Томе II.

О сервлетах

Это средство задействуется применением библиотек, и поэтому сведения о нем — в Томе II.

О многопоточности

Это средство задействуется применением библиотек, и поэтому сведения о нем — в Томе II.