Базовые конструкции

курс

Программирование для мобильных платформ

Макет исходного файла Java

Базовый синтаксис исходного файла Java:

У каждого файла ***.java*** есть 4 составляющих, из которых мы в наших примерах в лекциях обсудили пока только одну – главный класс.

[<package_declaratlon>]
<Import_declaration>*
<class_declaratlon>+
<class_declaratlon>-

- ✓ ***одиночный оператор package (необязателен)***
- ✓ ***любое количество операторов import (необязательны)***
- ✓ ***одиночное объявление открытого (public) класса***
- ✓ ***любое количество закрытых (private) классов пакета (необязательны)***

Макет исходного файла Java

Оператор **package**

- ✓ Первое, что может появиться в исходном файле ***Java*** — это оператор ***package***, который сообщает транслятору, в каком пакете должны определяться содержащиеся в данном файле классы.
- ✓ Пакеты задают **набор раздельных пространств имен**, в которых хранятся имена классов. Если оператор ***package*** не указан, классы попадают в безымянное пространство имен, используемое по умолчанию.
- ✓ Если вы объявляете класс, как принадлежащий определенному пакету, например,

package java.awt.image;

то и исходный код этого класса должен храниться в каталоге ***java/awt/image***.

*Каталог, который транслятор **Java** будет рассматривать, как корневой для иерархии пакетов, можно задавать с помощью переменной окружения **CLASSPATH**.*
*С помощью этой переменной можно задать несколько корневых каталогов для иерархии пакетов (через ; как в обычном **PATH**).*

Макет исходного файла Java

Трансляция классов в пакетах

При попытке поместить класс в пакет, вы сразу натолкнетесь на жесткое требование **точного совпадения иерархии каталогов с иерархией пакетов**. Нельзя переименовать пакет, не переименовав каталог, в котором хранятся его классы. Эта трудность видна сразу, но есть и менее очевидная проблема.

Представьте себе, что вы написали класс с именем ***PackTest*** в пакете ***test***. Вы создаете каталог ***test***, помещаете в этот каталог файл ***PackTest.java*** и транслируете. Пока — все в порядке. Однако при попытке запустить его вы получаете от интерпретатора сообщение «***can't find class PackTest***» («Не могу найти класс ***PackTest***»). Ваш новый класс теперь хранится в пакете с именем ***test***, так что теперь надо указывать при компиляции всю иерархию пакетов, разделяя их имена точками - ***test.PackTest***. Кроме того Вам надо либо подняться на уровень выше в иерархии каталогов и снова набрать «***java test.PackTest***», либо внести в переменную ***CLASSPATH*** каталог, который является вершиной иерархии разрабатываемых вами классов.

И еще: не забудьте в переменную окружения ***PATH*** операционной системы внести ***путь к каталогу BIN JDK***,

например, ***C:\Program Files\JAVA\jdk1.8.0_20\bin***

Макет исходного файла Java

Оператор import

После оператора ***package***, но до любого определения классов в исходном ***Java-файле***, может присутствовать список операторов ***import***.

Пакеты являются хорошим механизмом для отделения классов друг от друга, поэтому все встроенные в ***Java*** классы хранятся в пакетах.

Общая форма оператора ***import*** такова:
import пакет1 [.пакет2].(имякласса|*);
Здесь ***пакет1*** — имя пакета верхнего уровня,
пакет2 — это необязательное имя пакета, вложенного в первый пакет и отделенное точкой.

Макет исходного файла Java

Оператор import

После указания пути в иерархии пакетов, указывается либо имя класса, либо метасимвол звездочка.

Звездочка означает, что, если Java-транслятору потребуется какой-либо класс, для которого пакет не указан явно, он должен просмотреть ***все содержимое пакета со звездочкой вместо имени конкретного класса.***

В приведенном ниже фрагменте кода показаны обе формы использования оператора ***import*** :

```
import java.util.Date
import java.io.*;
```

*Использовать без нужды форму записи оператора import **со звездочкой** не рекомендуется, т.к. это может значительно увеличить время трансляции кода (на скорость работы и размер программы это не влияет).*

Макет исходного файла Java

Оператор import

Все встроенные в ***Java*** классы, которые входят в комплект поставки, хранятся в пакете с именем ***java***. Базовые функции языка хранятся во вложенном пакете ***java.lang***. Весь этот пакет **автоматически импортируется** транслятором во все программы. Это эквивалентно размещению в начале каждой программы оператора

```
import java.lang.*;
```

Если в двух разных пакетах, подключаемых с помощью формы оператора ***import*** со звездочкой, есть классы с одинаковыми именами, однако вы их не используете, транслятор не отреагирует. А вот при попытке использовать такой класс, вы сразу получите сообщение об ошибке, и вам придется переписать операторы ***import***, чтобы явно указать, класс какого пакета вы имеете ввиду:

```
class MyDate extends Java.util.Date { }
```

Подробнее см: http://www.helloworld.ru/texts/comp/lang/java/java/08.htm

Пример работы с пакетами

Исходный файл Java:

```
public class HelloWorld {
public static void main(String[] args) {
          System.out.println("Hello World!");
           }
 }
```

Пока у нас нет пакетов, делаем так:

Переходим в каталог, где лежит данный файл ***HelloWorld.java***, и выполняем команду:

javac HelloWorld.java

В данной папке появится файл ***HelloWorld.class***. Значит программа скомпилирована. Чтобы запустить:

java –classpath . HelloWorld

Пример работы с пакетами

Теперь поместим наш класс в пакет с именем ***com.greet.helloworld***. Для этого добавим в начало файла строчку:

```
package com.greet.helloworld;
```

Учтите, что имена пакетов должны быть в нижнем регистре. И избегайте использования спецсимволов. Проблемы возникают из-за разных платформ и файловых систем.

И создадим такую структуру каталогов:

```
HelloWorld
'---src
'       '---com
'               '---greet
'                       '---helloworld
'                                       '---HelloWorld.java
```

Путь к исходнику: src/com/greet/helloworld/HelloWorld.java.

Пример работы с пакетами

Компилируем:

```
javac -d classes src/com/greet/helloworld/HelloWorld.java
```

Указываем, что результат нужно положить в каталог **classes.**

! Директорию **classes** надо создать ручками, все остальное появится автоматически!

В каталоге *classes* автоматически создастся структура каталогов как и в *src* :

```
HelloWorld
'---classes
'       '---com
'               '---greet
'                       '---helloworld
'                                       '---HelloWorld.class
'---src
'       '---com
'               '---greet
'                       '---helloworld
'                                       '---HelloWorld.java
```

Обычно бинарные файлы хранятся в отдельной папке classes и не путаются с исходниками

Находясь в **HelloWorld**

Запускаем:

```
java -classpath ./classes com.greet.helloworld.HelloWorld
```

Либо из **classes**

Запускаем:

```
java com.greet.helloworld.HelloWorld
```

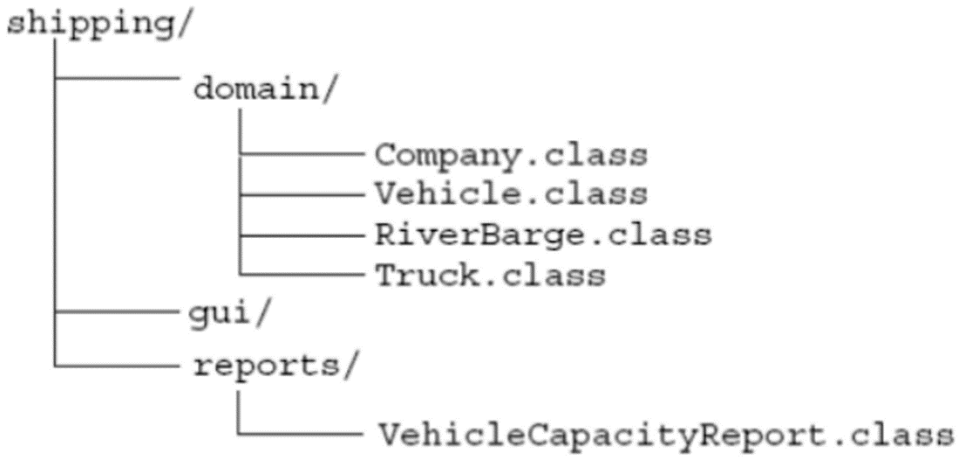

Больше информации см.: ***http://habrahabr.ru/post/125210/*** *и* ***http://life-prog.ru/view_zam.php?id=4***

Макет исходного файла Java

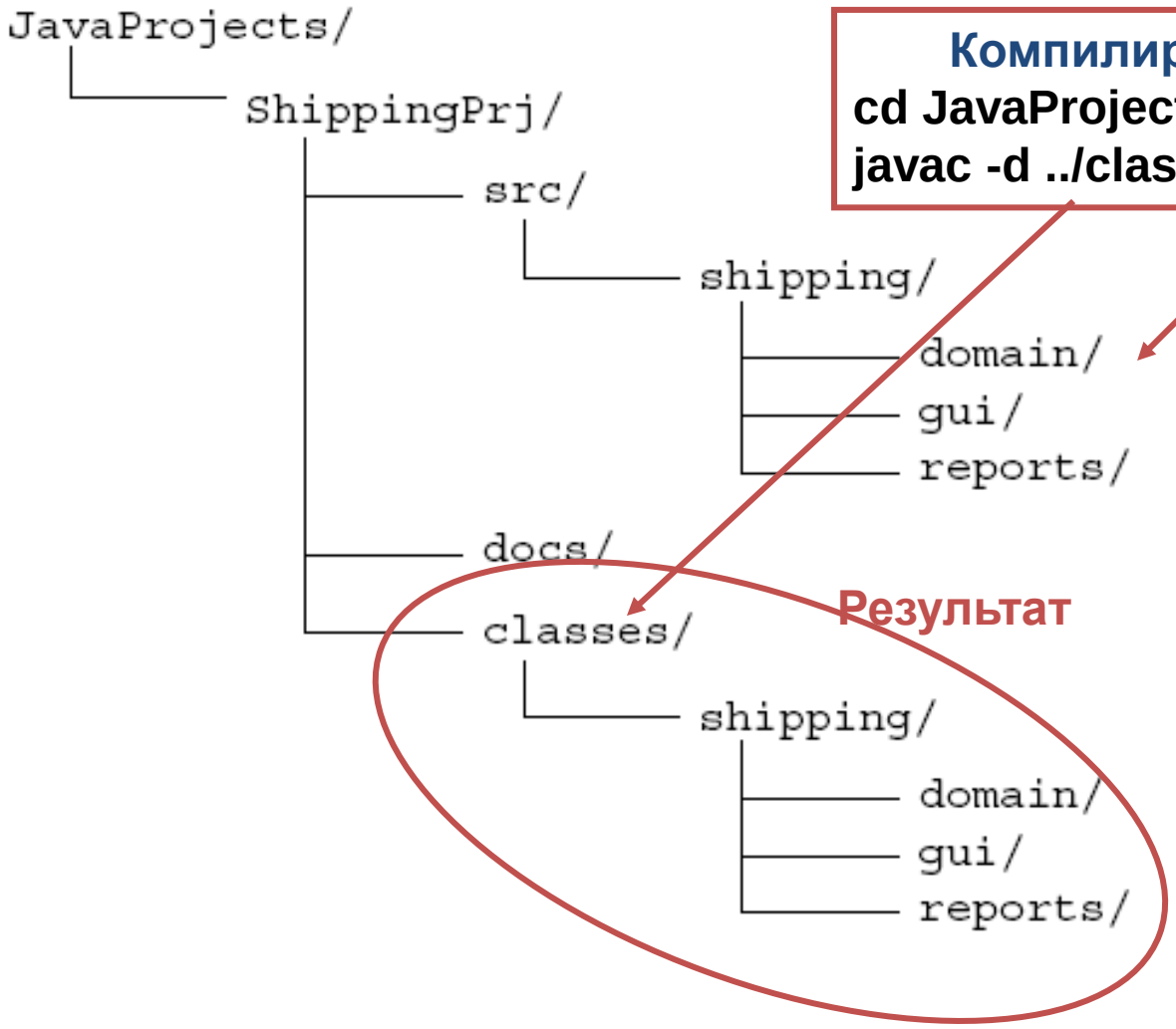
Например, файл **vehicleCapacityReport. java** такой:

```
package shipping.reports;
// означает, что файл помещен в пакет shipping.reports (он же
структура  директорий – путь к файлу-источнику)

import shipping.domain;
import java.util.List; // подключение внешнего пакета
//(import_declaratlon)
// подключение всех внешних классов, содержащихся в пакете java.io
import java.io.*;
   public class VehicleCapacityReport {
   private List vehicles;
   public void generateReport(Writer output) {...}
}
```



Development



Компилируем, используя опцию -d

```
cd JavaProjects/ShippingPrj/src
javac -d ../classes shipping/domain/*.java
```

Источник

Результат

Назначение архива jar

исполняемые пакеты

- Исторически языки типа .NET и C++ обладают преимуществом дружественного отношения к ОС: чтобы запустить приложение, достаточно просто набрать его имя в командной строке (helloWorld.exe) или дважды щелкнуть на соответствующий значок в оболочке графического интерфейса пользователя (GUI).
- В Java-программировании модуль запуска приложения —java— загружает в процесс JVM, и ему нужно передать аргумент командной строки (**com.greet.helloworld.HelloWorld**), указывающий класс, метод main() которого мы хотим запустить. Эти дополнительные шаги затрудняют создание удобных для пользователя приложений на языке Java. Все эти элементы нужно набрать в командной строке, чего многие конечные пользователи стараются избегать, и очень часто что-то выходит не так, и это приводит к странным сообщениям об ошибке.
- От этого можно уйти, сделав JAR-файл «исполняемым», так чтобы при его выполнении модуль запуска Java автоматически узнавал, какой класс нужно запустить. Для этого достаточно ввести в манифест JAR-файла (MANIFEST.MF в подкаталоге архива JAR META-INF) запись о главном классе.

Создание архива jar
исполняемые пакеты

- Классы можно объединять в jar-пакеты.
- Запуск программы из такого пакета происходит быстрее.
- Пакет может содержать ресурсы для приложения.
- Создание архива утилитой jar:

---

jar cvf HelloWorld.jar HelloWorld/.class*

Запуск приложения, запакованного в jar файл:

```
java -jar HelloWorld.jar
```

**Сходство имен класса и архива не обязательно*

Создание архива jar

исполняемые пакеты

```
// Создание нового jar-файла
>jar cvf file.jar список_файлов

// Просмотр содержимого архива
> jar tf file.jar

// Извлечение содержимого из jar-файла
> jar xf file.jar

// Открыть файл манифеста
> MANIFEST.MF
```

Создание архива jar

исполняемые пакеты

// Создание нового jar-файла

>jar cvf file.jar список_файлов

```
M:\JAVA\1\Calc\classes>jar cvf Calc.jar calc/Calc.class calc/*
adding: META-INF/ (in=0) (out=0) (stored 0%)
adding: META-INF/MANIFEST.MF (in=56) (out=56) (stored 0%)
adding: calc/Calc.class (in=726) (out=449) (deflated 38%)
adding: calc/Calc.class (in=726) (out=449) (deflated 38%)
adding: calc/Calculator.class (in=455) (out=355) (deflated 21%)
adding: calc/operation/ (in=0) (out=0) (stored 0%)
adding: calc/operation/Adder.class (in=420) (out=280) (deflated 33%)
Total:
------
(in = 2371) (out = 2373) (deflated 0%)
```

// Просмотр содержимого архива

> jar tf file.jar

```
M:\JAVA\1\Calc\classes>jar tf Calc.jar
META-INF/
META-INF/MANIFEST.MF
calc/Calc.class
calc/Calc.class
calc/Calculator.class
calc/operation/
calc/operation/Adder.class
```

Создание архива jar

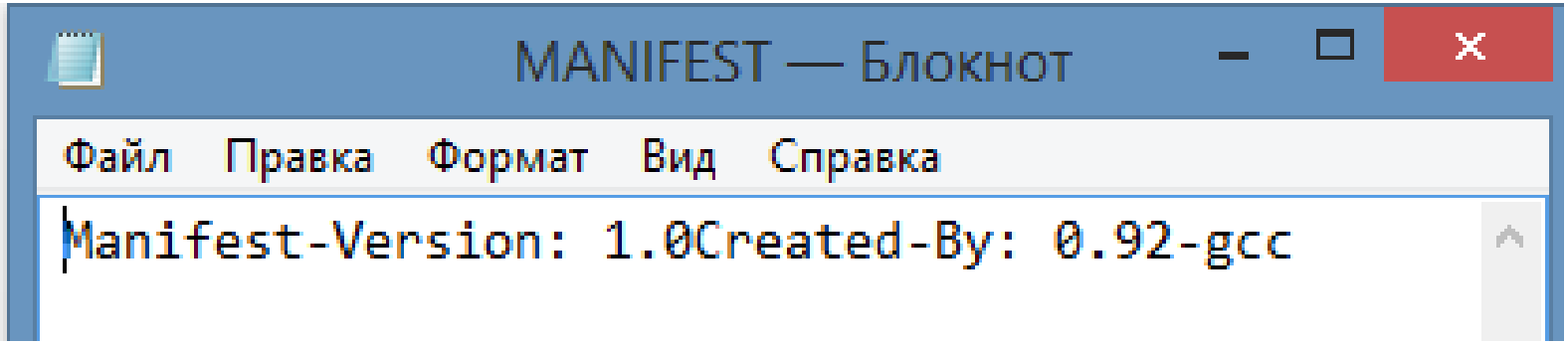
исполняемые пакеты

// Извлечение содержимого из jar-файла

> jar xf file.jar

```
M:\JAVA\1\Calc\classes>jar xf Calc.jar
```

> MANIFEST.MF



Создание архива jar

исполняемые пакеты

В редакторе отредактировать манифест, дописав в него главный класс и добавив перевод строки:
в конце манифеста должна быть пустая строка:

Main-Class: calc/Calc

! Редактируя манифест, Вы не должны писать расширение class

! Отредактированный файл-манифест должен быть сохранен в кодировке UTF-8

```
M:\...NIFEST.MF      1251  Ln
Manifest-Version: 1.0
Created-By: 0.92-gcc
Main-Class: calc.Calc
```

Создание архива jar

исполняемые пакеты

// создаем новый архив с обновленным Манифестом:

> jar cvfm Calc.jar MANIFEST.MF calc/*

```
M:\JAVA\1\Calc\classes>jar cvfm Calc.jar MANIFEST.MF calc/*
adding: META-INF/ (in=0) (out=0) (stored 0%)
adding: META-INF/MANIFEST.MF (in=66) (out=66) (deflated 0%)
adding: calc/Calc.class (in=726) (out=449) (deflated 38%)
adding: calc/Calculator.class (in=455) (out=355) (deflated 21%)
adding: calc/operation/ (in=0) (out=0) (stored 0%)
adding: calc/operation/Adder.class (in=420) (out=280) (deflated 33%)
Total:
------
(in = 1667) (out = 1840) (deflated -10%)
```

// запускаем архив на исполнение:

> java -jar Calc.jar

```
M:\JAVA\1\Calc\classes>java -jar Calc.jar
Hello World!
1+2+3=6
```

Создание архива jar

исполняемые пакеты

jar - это обычный ZIP-архив. Единственная особенность - наличие папки META-INF. Дополнительно в этой папке могут присутствовать другие файлы, о чём можно прочитать в разных обучающих курсах: Using JAR Files: The Basics, Jar File Overview, Lesson: Packaging Programs in JAR Files.

Краткая выжимка:
создать jar:

jar cf jar-file input-file(s)

создать jar со своим файлом манифеста:

jar cfm manifest jar-file input-file(s)

Создание архива jar

исполняемые пакеты

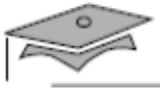
создать jar: *jar cf jar-file input-file(s)*

Слово "jar" ссылается на программу jar.exe, которая создает JAR-файл.
Опция "c" означает, что вы хотите создать JAR-файл
Опция "f" означает, что вы хотите указать имя файла.

input-file(s) - ***"входные_файлы" - разделенный пробелами список файлов, которые должны быть включены в JAR-файл, если все файлы находятся в одной директории, то можно написать имя_дерикторию/****

При запуске jar-файла двойным щелчком на самом деле выполняется команда **java -jar jar-file**. Для того, чтобы эта команда сработала, в манифесте должен быть указан запускаемый класс:

java -cp JAR-file.jar MainClass



ОСНОВЫ ЯЗЫКА JAVA

Идентификаторы, ключевые слова, типы

Комментарии

В стандарте языка Java имеются три типа комментариев:

```
/*Comment*/;
//Comment;
/** Comment*/
```

- ✓ Первые два полностью аналогичны C++.
- ✓ Третий тип называется документационный комментарий и введен в Java для автоматического документирования кода. После написания исходного текста утилита автоматической генерации документации javadoc собирает тексты таких комментариев в один файл.

Разделители

( )	Списки параметров в объявлении и вызове метода, задание приоритета операций в выражениях, выделение выражений в операторах управления выполнением программы и в операторах приведения типов.
{ }	Блоки кода, списки инициализации массивов
[ ]	Объявление массивов, доступ к элементам массивов
;	Разделяет операторы.
,	Разделяет идентификаторы в объявлениях переменных, а также цепочку выражений внутри оператора for.
.	Разделение имен пакетов и классов, обращение к члену или методу класса.

Semicolons, Blocks, and White Space

Определение класса заключается в блок:

```
public class MyDate
  { private int day;
    private int month;
    private Int year;
}
```

Можно формировать вложенные блоки:

```
while ( i < large ) {
 a = a + i;
// вложенный блок
    if ( a == max ) {
      b = b + a;
      a++;}
 i++;
}
```

Идентификаторы

Идентификаторы имеют следующие характеристики:

- Это имена, данные переменным, классам или методам
- Может начинаться с буквы в кодировке Unicode, подчеркивания (_), или знак доллара ($)
- Чувствительны к регистру и не ограничены максимальной длиной

Примеры:

```
identifier
userName
user_name
_sys_varl
$changе
```

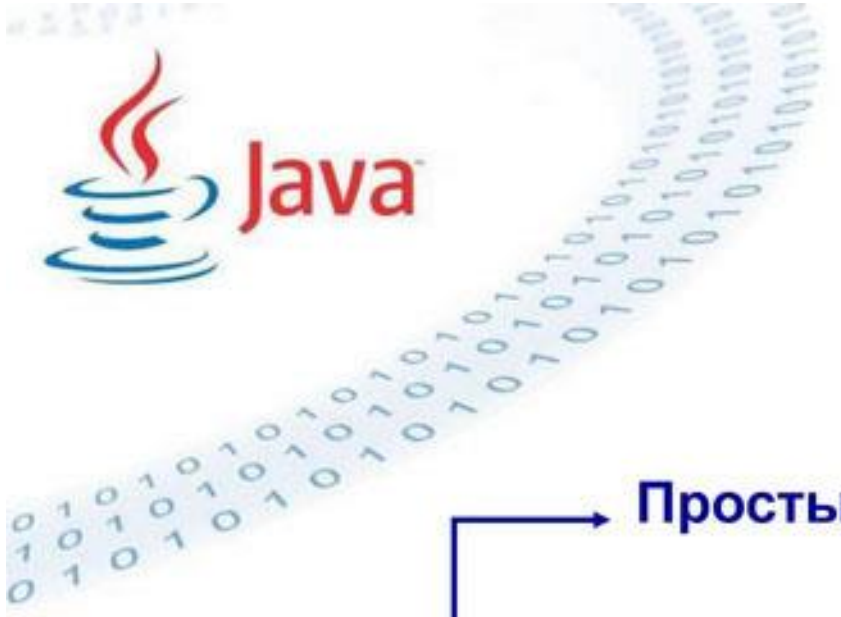
Ключевые слова

abstract	continue	for	new	switch
assert	default	goto	package	synchronized
boolean	do	if	private	this
break	double	implements	protected	throw
byte	else	import	public	throws
case	enum	instanceof	return	transient
catch	extends	int	short	try
char	final	interface	static	void
class	finally	long	strictfp	volatile
const*	float	native	super	while

Зарезервированы ключевые слова null, true, false

Ключевые слова и значения нельзя использовать в качестве имен переменных, классов или методов.

Типы данных



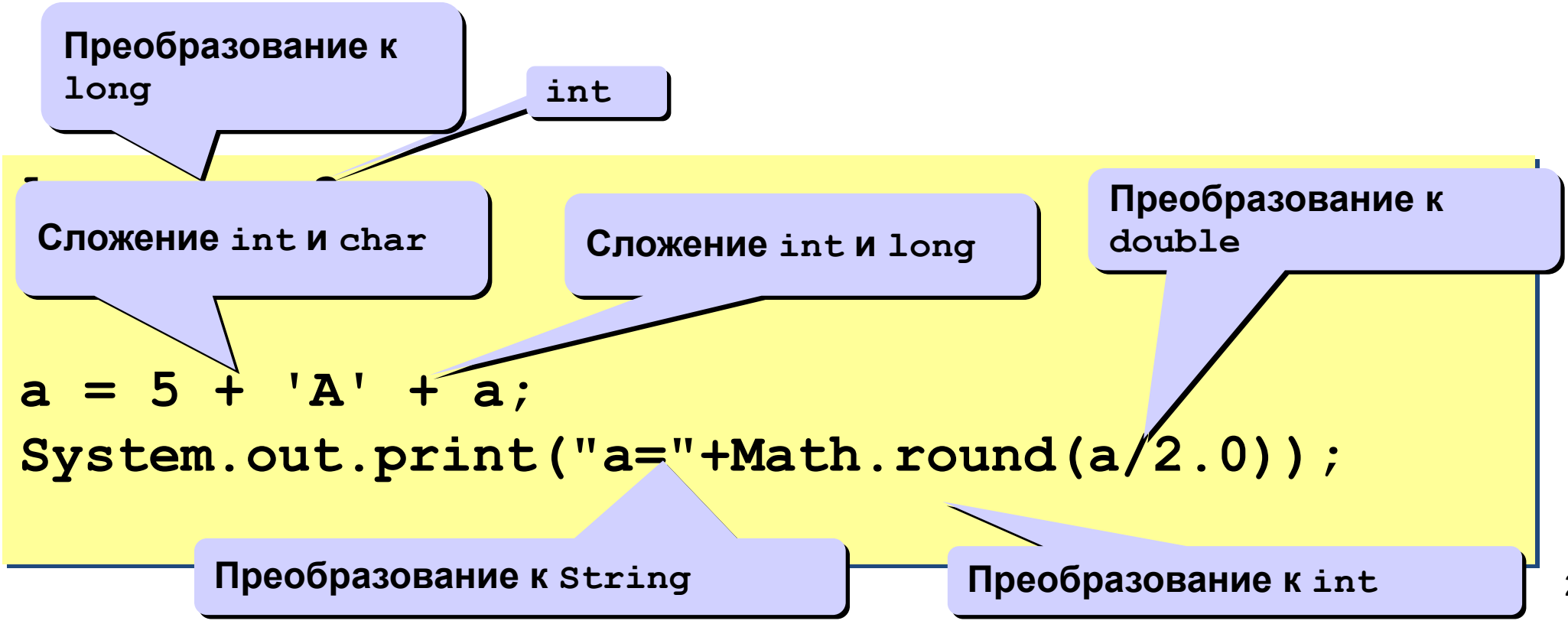
Типы данных в Java

- Типы данных
  - Простые
    - Целые (byte, short, int, long)
    - С пл.точкой (float, double)
    - Символьный (char)
    - Логический (boolean)
  - Ссылочные
    - Массивы
    - Классы
    - Интерфейсы

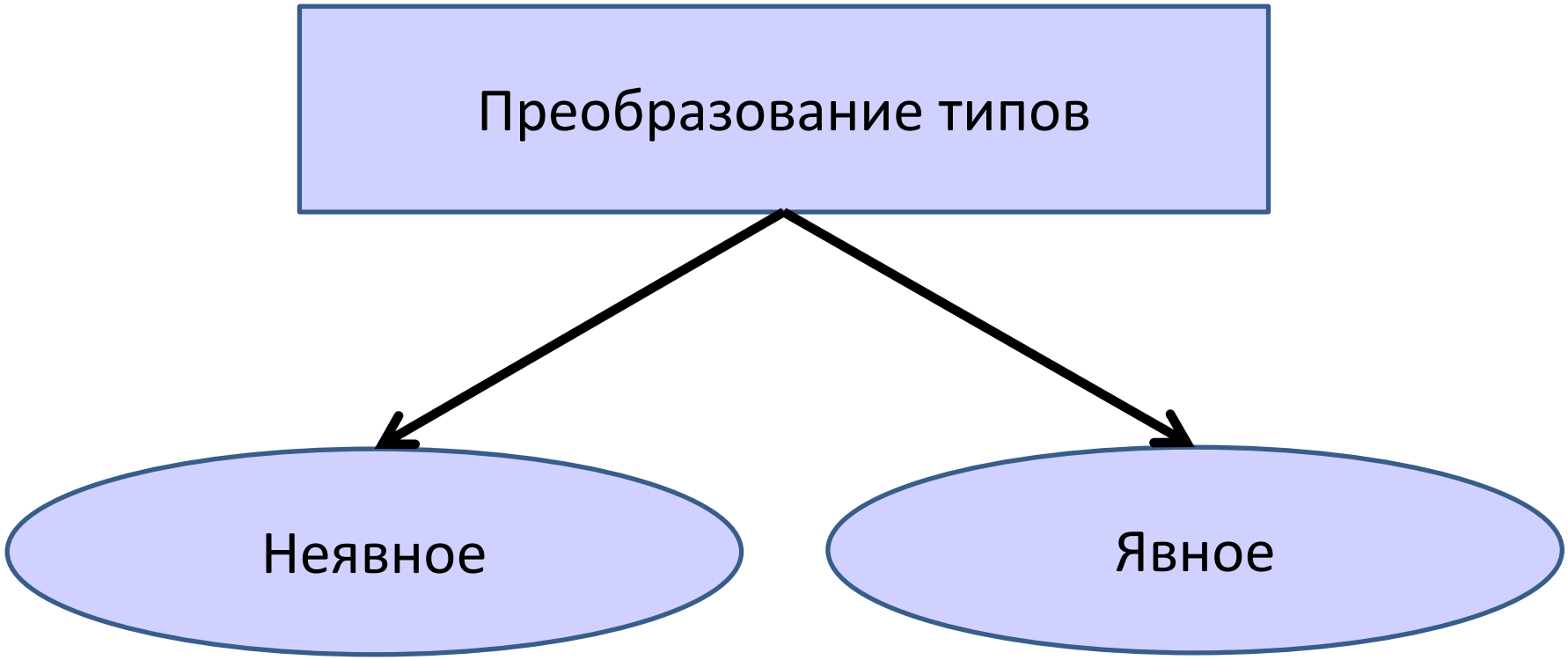
Java – строго типизированный язык

Java – язык строго типизированный язык.
Компилятор и виртуальная машина всегда следят за работой с типами, гарантируя надежность выполнения программы.

Однако, есть случаи, когда нужно осуществить то или иное преобразование для реализации логики программы.



Преобразование типов

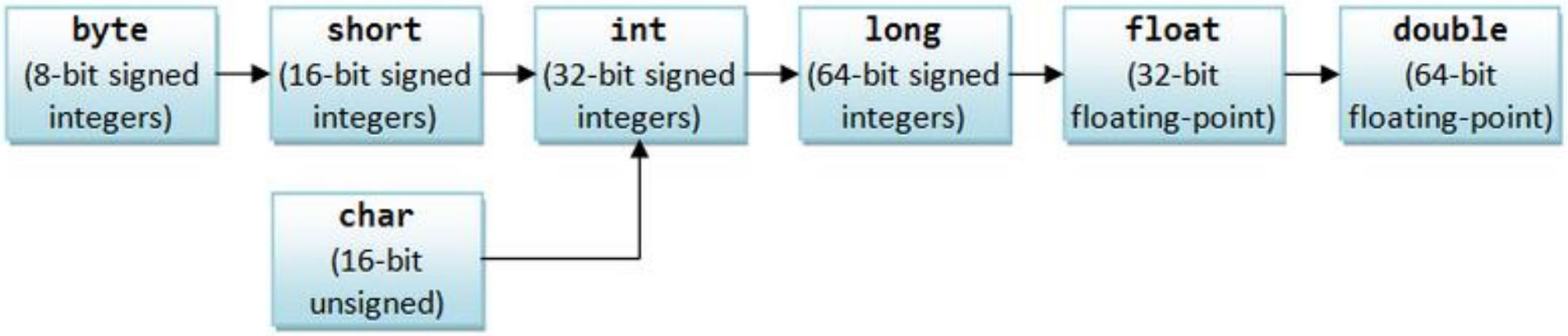


Неявное преобразование типов

Выполняется в случае, если выполняются условия:

1. Оба типа совместимы;
2. Длина целевого типа больше длины исходного типа.

Пример. Преобразование от **`byte`** к **`int`**.



Направление для неявного преобразования типов

Явное преобразование типов

Общая форма явного преобразования

```
(<целевой тип>) <значение>
```

1. Если длина целевого типа меньше длины исходного типа, то происходит **преобразование с сужением**.
2. Если значение переменной вещественного типа присваивается переменной целого типа, то выполняется **усечение** (отбрасывается дробная часть).

Пример 1. Преобразование от `int` к `short`.

Сужение

Пример 2. Преобразование от `float` к `int`.

Усечение

Преобразование типов. Пример

```
int b = 1;
byte a = b;
byte c = (byte)-b;
int i = c;
```

Ошибка!

Явное преобразование переменной типа `int` к типу `byte`

Автоматическое неявное преобразование переменной типа `byte` к типу `int`

Виды приведений

- тождественное (**`identity`**);
- расширение примитивного типа (**`widening primitive`**);
- сужение примитивного типа (**`narrowing primitive`**);
- расширение объектного типа (**`widening reference`**);
- сужение объектного типа (**`narrowing reference`**);
- преобразование к строке (**`String`**);
- запрещенные преобразования (**`forbidden`**).

Тождественное преобразование

В **Java** преобразование выражения любого типа к точно такому же типу **всегда** допустимо и успешно выполняется.

Для чего нужно такое преобразование?

- Любой тип в **Java** может участвовать в преобразовании, хотя бы в тождественном.

Тип **boolean** можно привести только к **boolean**.

- Облегчение чтения кода для разработчика.

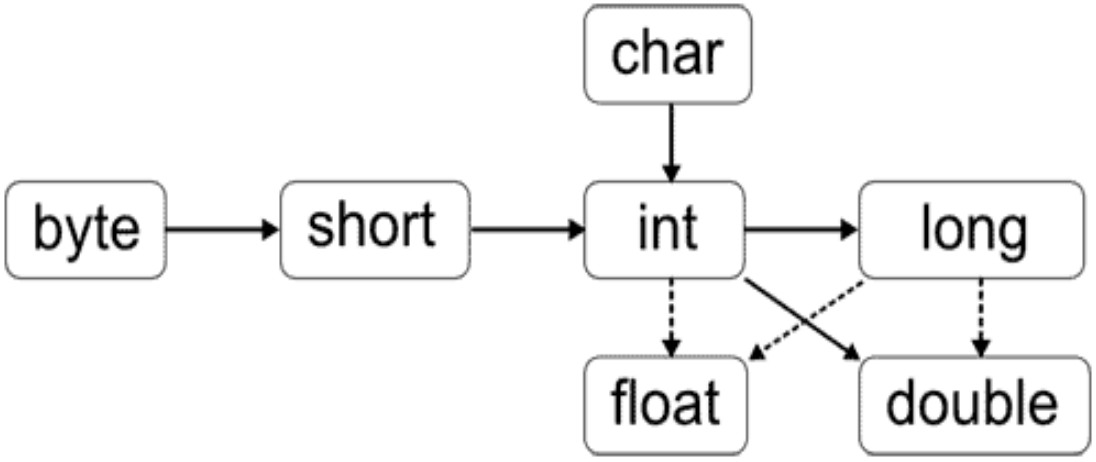
```
print(getCity().getStreet().getHouse().getFlat());

print((Flat)(getCity().getStreet().getHouse().
      getFlat()));
```

Преобразование типов

Простой тип, расширение	Объектный тип, расширение
Простой тип, сужение	Объектный тип, сужение

Схема допустимого преобразования типов



✓Шесть сплошных линий со стрелками обозначают преобразования, которые выполняются без потери информации.
✓Три штриховые линии, со стрелками, означают преобразования, при которых может произойти **потеря точности.**

Расширение простого типа

Расширение простого типа – переход от менее емкого типа к более емкому.

Это преобразование **безопасно** – новый тип вмещает все данные, которые хранились в старом.

Не происходит потери данных.

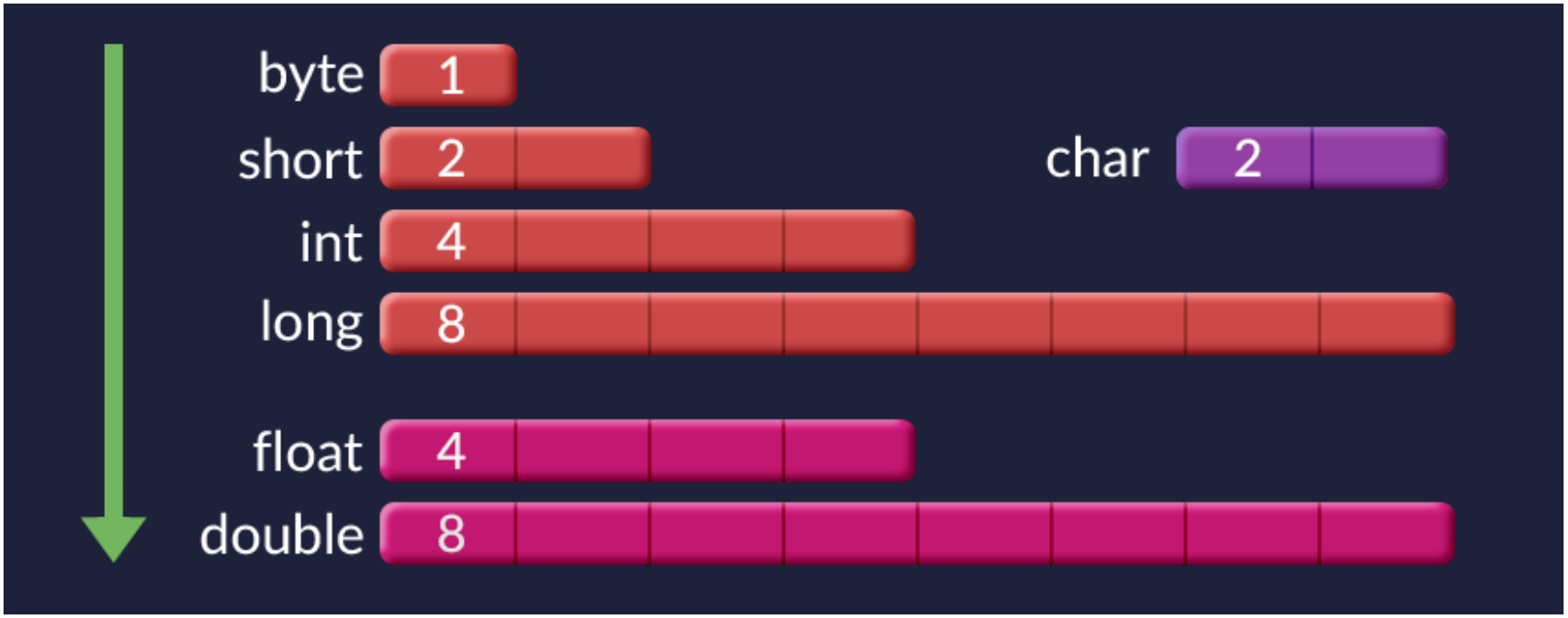
Например, от типа **`byte`** (1 байт) к типу **`int`** (4 байта)

```
byte b = 3;
int a = b;
```

Типы данных

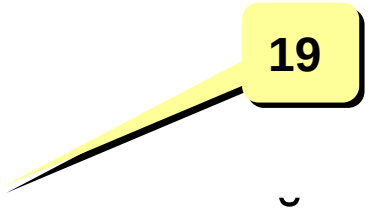
Тип	Размер	Min	Max
byte (байт)	8 бит	-128	+127
short (короткое целое)	16 бит	-2^{15}	$2^{15}-1$
int (целое число)	32 бита	-2^{31}	$2^{31}-1$
long (целое число двойного размера)	64 бита	-2^{63}	$2^{63}-1$
char (символьный)	16 бит	0	$2^{16}-1$
float (вещественные)	32 бита	3.4e-038	3.4e+038
double (вещественные двойной точности)	64 бита	1.7e-308	1.7e+308

Типы данных



Расширения простого типа

- от **byte** к **short, int, long, float, double**
- от **short** к **int, long, float, double**
- от **char** к **int, long, float, double**
- от **int** к **long, float, double**
- от **long** к **float, double**
- от **float** к **double**



Сколько существует расширяющих преобразований простого типа?

Расширение является рискованным

- от типов **byte** и **short** к типу **char**
- от типа **char** к типу **short**

Искажения при расширении

У дробных типов значащих разрядов (разрядов, отведенных на представление мантиссы) меньше, чем у некоторых целых типов, поэтому возможна **потеря точности** при:

- приведении значений **`int`** к типу **`float;`**
- приведении значений типа **`long`** к типу **`float`** или **`double`**.

```
long a=111111111111L;
float f = a;
System.out.println(f);
a = (long) f;
System.out.print(a);
```

1.11111111e11

111111110656

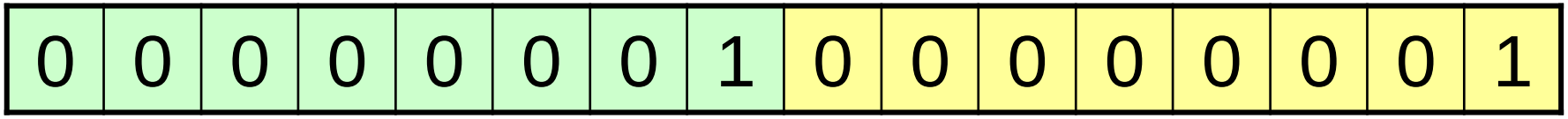
Сужения простого типа

Сужение простого типа – переход осуществляется от более емкого типа к менее емкому.

Риск потерять данные! При сужении целочисленного типа все старшие биты отбрасываются.

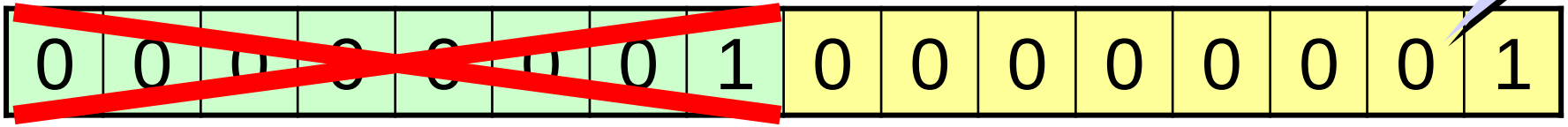
```
short a = 257;
```

16 бит




```
byte b = (byte)a;
```

8 бит




```
System.out.print(b);
```

1

Сужения простого типа

- от **byte** к **char**
- от **short** к **byte**, **char**
- от **char** к **byte**, **short**
- от **int** к **byte**, **short**, **char**
- от **long** к **byte**, **short**, **char**, **int**
- от **float** к **byte**, **short**, **char**, **int**, **long**
- от **double** к **byte**, **short**, **char**, **int**, **long**, **float**

Таблица преобразований

Преобразовать из:	Преобразовать в: boolean	byte	Short	Char	Int	long	float	double
boolean	–	N	N	N	N	N	N	N
byte	N	–	Y	C	Y	Y	Y	Y
short	N	C	–	C	Y	Y	Y	Y
char	N	C	C	–	Y	Y	Y	Y
int	N	C	C	C	–	Y	Y*	Y
long	N	C	C	C	C	–	Y*	Y*
float	N	C	C	C	C	C	–	Y
double	N	C	C	C	C	C	C	–

Буква **N** означает невозможность преобразования. Буква **Y** означает расширяющее автоматическое преобразование. Буква **C** означает сужающее преобразование, требующее явного приведения. Наконец, **Y*** означает автоматическое расширяющее преобразование, в процессе которого значение может потерять некоторые из наименее значимых разрядов.

Сужения простого типа. Примеры

```
System.out.print((byte)383);
```
127

```
System.out.print((byte)384);
```
-128

```
System.out.print((byte)-384);
```
-128

Кроме значения может быть потерян знак

```
char c=40000;
System.out.print((short)c);
```
-25536

Сужение дробного типа до целочисленного

1. Дробное значение преобразуется [illegible] типом является **`long`**, или в **`int`** – в противном случае, т.е. дробная часть отбрасывается.
   - Если исходное число – **`NaN`**, то результат 0.
   - Если исходное число – положительная или отрицательная бесконечность, то результат – максимальное или минимальное значение выбранного типа.
   - Если дробное число конечной величины, но после округления получилось слишком большое по модулю число выбранного типа, то результатом будет максимальное или минимальное число выбранного типа.
   - Если результат округления уложился в диапазон типа, то он и будет результатом.
2. Производится дальнейшее сужение от выбранного целочисленного типа к целевому, если это необходимо.

Не число (`Not A Number`)

Типы данных

✓ В отличие от некоторых других языков (C/C++), в Java из-за требования переносимости программ все типы данных имеют строго определенный диапазон, не зависящий от платформы, на которой выполняется программа.

✓ Все числовые типы (целые и с плавающей точкой) в Java являются знаковыми (в отличие от C/C++, где есть unsigned типы).

✓ **В Java простые типы всегда передаются по значению, а ссылочные по ссылке.**

✓ Для всех простых типов в языке Java есть соответствующие классы — **классы-оболочки (wrapper).** Конечно, они предназначены не для вычислений, а для действий, типичных при работе с классами, — создания объектов, преобразования типов объектов, получения численных значений объектов в разных формах и передачи объектов в методы по ссылке.

Дополнительная информация о простых типах
Логический - boolean

Логический тип имеет следующие характеристики:

• логический тип данных имеет два литерала, true и false.
Например, в выражении:

```
boolean truth = true;
```

объявлена переменная логического типа truth и ей присвоено значение истина (true).

При этом true!=1, а false!=0.
Преобразование в целый тип невозможно!

Символьный тип

Имя типа	Ширина	Диапазон
char	16	0 ...65 535

Для представления символов Java использует кодировку Unicode. Unicode определяет полный набор интернациональных символов, объединяющий множество наборов символов национальных алфавитов. По этой причине в Unicode для хранения каждого символа выделяется 16 бит. Таким образом, тип char в Java – 16 разрядный. Диапазон его значений варьируется от 0 до 65535. Стандартный набор символов ASCII располагается в интервале 0 – 127.

- Символ должен быть заключен в одинарные кавычки (' ')
- Использует следующие обозначения:

‘a‘ - буква (символ) a

‘ \ t ‘ - символ табуляции

'\ U ????' Конкретный Unicode символ, ***????***, заменяется четырьмя шестнадцатеричными цифрами. Например, код '\ U03A6' представляет греческую букву фи.

Текстовый тип - строка

Тип Текстовая строка имеет следующие характеристики:

- ✓ Не примитивный тип данных, не просто массив символов: **строка - это класс!**
  - ✓Заключена в двойные кавычки ("") :

```
"Строка в Java – это класс, а массив символов литерал!"
```

  - ✓Может использоваться следующим образом:

```
String greeting = "Good Morning !! \n";
String errorMessage = "Record Not Found!";
```

Типы данных

Целочисленные типы

Имя типа	Разрядность	Диапазон
byte	8	-128 ... 127
short	16	-32 768 ... 32 767
int	32	-2 147 483 648 ... 2 147 483 647
long	64	-9 223 372 036 854 775 808 ... 9 223 372 036 854 775 807

Целые типы могут быть представлены в трех формах: десятичное, восьмеричное или шестнадцатеричное:

2 - десятичная форма для целого 2.

077 - ведущий 0 указывает на восьмеричное значение.

0xBAAC - ведущий 0x указывает на шестнадцатеричное значение.

- литералы имеют тип по умолчанию ***int***.
- литералы с суффиксом ***L*** или ***l*** имеют тип ***long***.

Целочисленные типы

Ширина целочисленного типа может не совпадать с занимаемым им количеством памяти. Исполняющая система ***Java*** может использовать для хранения целочисленных переменных **большее количество памяти по соображениям эффективности** выполнения кода. Например, некоторые реализации ***Java*** фактически хранят переменные типа ***byte*** и ***short*** в виде 32-битовых значений, т.к. это число соответствует размеру машинного слова большинства современных процессоров.

Тип ***byte*** лучше всего подходит для хранения произвольного потока байтов, загружаемого из сети или из файла. **За исключением этих случаев, использования типа *byte*, как правило, следует избегать**.

Для обычных целых чисел, используемых в качестве счетчиков и в арифметических выражениях, гораздо лучше подходит тип *int*, т.к. все операции с переменными этого типа процессор выполняет наиболее быстро.

Целочисленные типы

Тип ***short*** – наиболее редко используемый в ***Java*** тип, поскольку он определен как тип, в котором старший байт хранится первым (в так называемом формате «***big-endian***»). Такой формат хранения данных принят в архитектурах ***SPARC*** и ***Power PC***, в то время как в архитектуре ***Intel x86*** первым хранится младший (менее значащий) байт (формат «***little-endian***»).

Чаще всего используется тип ***int***. Он наиболее универсален и эффективен, с точки зрения выполнения кода.

Тип ***long*** используется в тех случаях, когда ***int*** недостаточен (по ширине) для хранения желаемого значения.

С плавающей точкой

Имя типа	Разрядность	Диапазон
float	**32**	**3.4e-038 ... 3.4e+038**
double	**64**	**1.7e-308 ... 1.7e+308**

Тип по умолчанию - ***double***.

Числа типа ***float*** имеют суффикс ***F***, например ***3.402F***. Числа с плавающей точкой, не имеющие суффикса ***F*** (например, ***3.402***), **всегда рассматриваются как числа типа double**. Для их представления можно (но не обязательно) использовать суффикс ***D***, например ***3.402D***.

Variables, Declarations, and Assignments

```
   public class Assign {
public static void main (String args []) {

int x, y i;           // declare Integer variables
float z = 3.414f;     // declare and assign floating point
double w = 3.1415;    // declare and assign doubles

boolean truth = true; // declare and assign boolean
char c;               // declare character variable
String str;           // declare String variable
   String strl = "bye";   // declare and assign String variable
   c = 'A';                  // assign value to char variable
str = "Hello there!";// assign value to String variable

x = 6;  // assign values to int variables
y = 1000,
   }
}
```

Автоматическое преобразование и приведение типов

- все byte и *short*-операнды расширяются до *int*
- если один операнд в выражении имеет тип long, тип всех операндов и результата расширяется до *long*
- если один операнд в выражении имеет тип *float*, тип всех операндов и результата расширяется до *float*
- если один операнд в выражении имеет тип *double*, тип всех операндов и результата расширяется до *double*

В математических выражениях Java производит расширение примитивных типов автоматически

Преобразование и приведение типов

Для чего нужно автоматическое расширение типов?

```
byte b = 100;
int i = (b * 2)/2;  //без расширения типов даст -28 вместо 100!
```

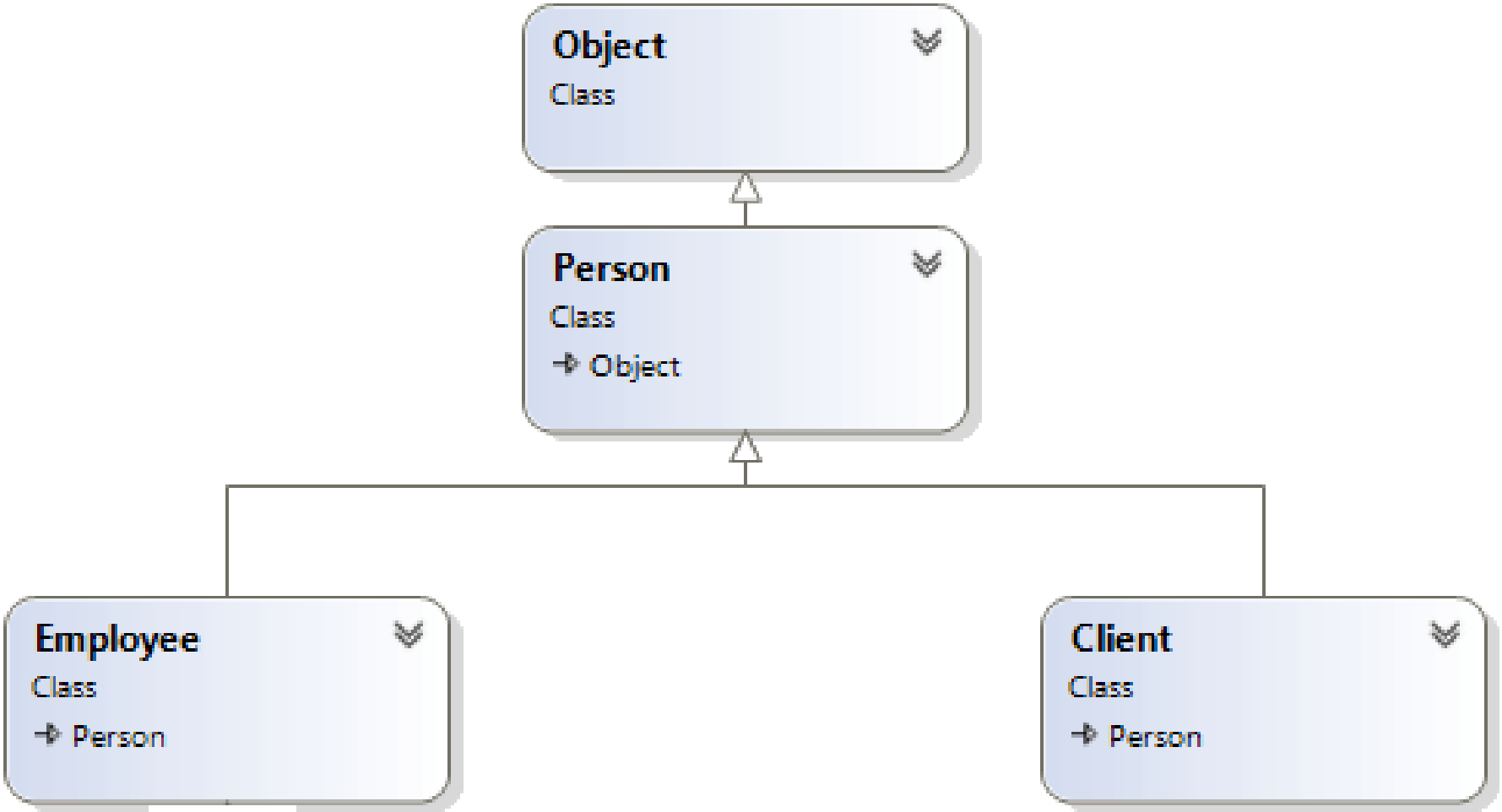
Бинарное числовое расширение может происходить при следующих операциях:

- арифметические операции +, -, *, /, %;
- операции сравнения <, <=, >, >=, ==, !=;
- битовые операции &, |, ^;
- в некоторых случаях для операции с условием ? :.

В математических выражениях Java производит расширение примитивных типов автоматически

Преобразование объектных типов

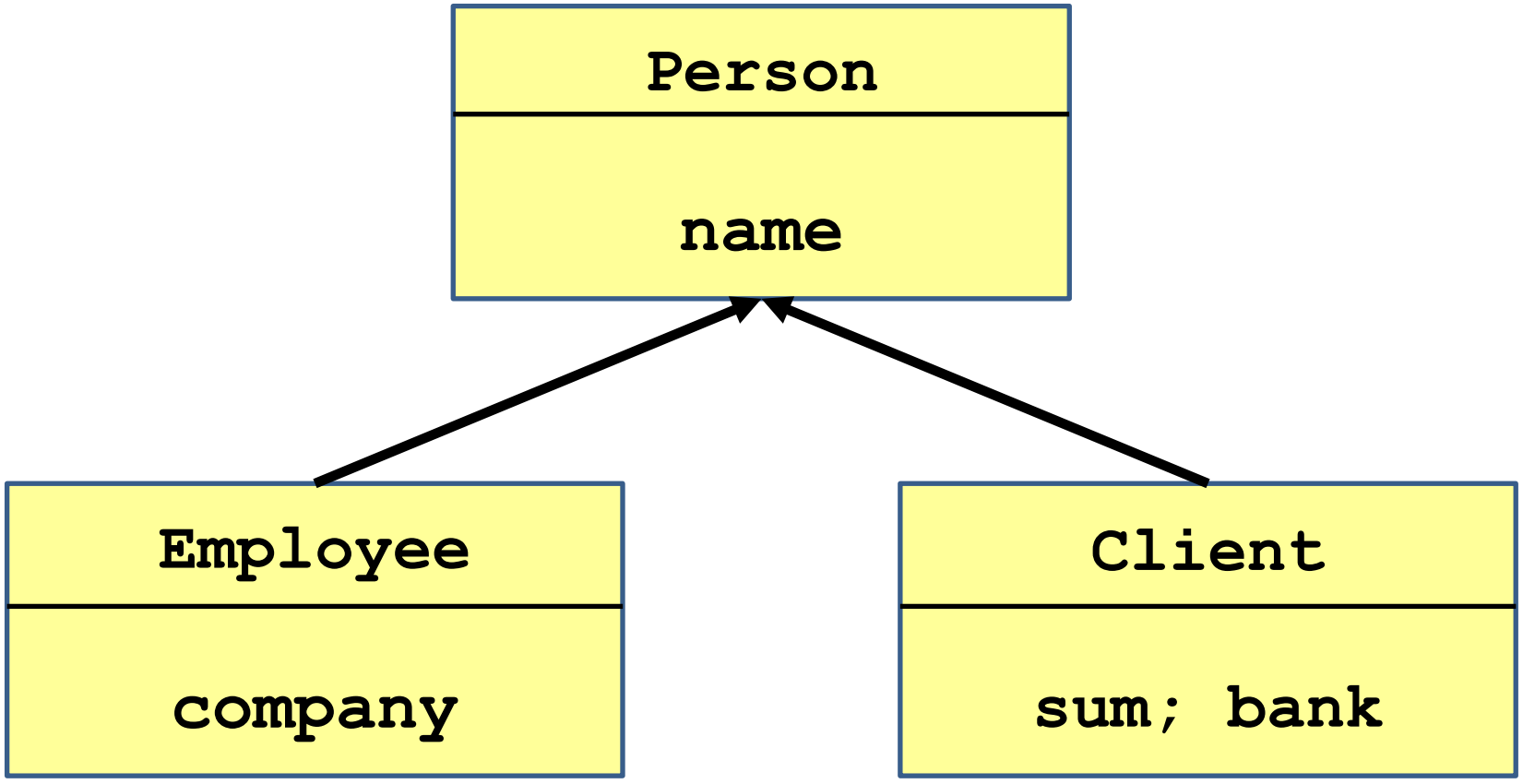
Преобразование объектных типов



Преобразование объектных типов

```
// Объявляем класс Person
class Person {
private String name;
public void display(){
        System.out.printf("Person %s \n", name);
    }}
// Объявляем класс Employee наследником от Person
class Employee extends Person{
private String company;
…}
// Объявляем второго наследника - класс Client
class Client extends Person{
private int sum;//Переменная для хранения суммы //на
счету
    private String bank;
…)
```

Схема наследования



Расширение объектных типов (восходящее преобразование)

Расширение – переход от более конкретного типа к менее конкретному, т.е. переход от детей к родителям.

Производится JVM автоматически.

Пример. Преобразование от наследника (**`Employee`**, **`Client`**) к родителю (**`Person`**).

```
Person p1 = new Employee("p1","Oracle");
Person p2 = new Client("p2", "DeutscheBank",
2000);
```

Внимание! С помощью ссылок **`p1`** и **`p2`** можно обращаться не только к полю **`name`**, но и к полям **`company`** и **`bank`**.

Восходящее преобразование

Мы можем преобразовать объект Employee по всей прямой линии наследования от Object к Employee.
Примеры нисходящих преобразований:

```
Object kate = new Client("Kate", "DeutscheBank", 2000);
((Person)kate).display();

Object sam = new Employee("Sam", "Oracle");
((Employee)sam).display();
```

Все преобразования сработают успешно

Сужение объектных типов - нисходящее преобразование.

Нисходящее преобразование – переход от менее конкретного типа к более конкретному, т.е. переход от родителей к детям.

Внимание! Такой переход не всегда возможен:.

```
Object kate = new Client("Kate", "DeutscheBank", 2000);
Employee emp = (Employee) kate;
emp.display();
 // или так
((Employee)kate).display();
```

Успешное преобразование

Ошибка!

Сужение объектных типов - нисходящее преобразование.

В данном случае переменная типа Object хранит ссылку на объект Client. Мы можем без ошибок привести этот объект к типам Person или Client. Но при попытке преобразования к типу Employee мы получим ошибку во время выполнения. Так как kate не представляет объект типа Employee.

Здесь мы явно видим, что переменная kate - это ссылка на объект Client, а не Employee. Однако нередко данные приходят извне, и мы можем точно не знать, какой именно объект эти данные представляют. Соответственно возникает большая вероятная столкнуться с ошибкой. И перед тем, как провести преобразование типов, мы можем проверить, а можем ли мы выполнить приведение с помощью оператора **instanceof:**

Оператор instanceof

Оператор **`instanceof`** проверяет принадлежность объекта указанному типу, возвращает значение типа **`boolean`**.

Объект

Класс

```
p instanceof Person
```

```
Object kate = new Client("Kate", "DeutscheBank", 2000);
if(kate instanceof Employee){

    ((Employee)kate).display();
}
else{
    System.out.println("Conversion is invalid");
}
```

Преобразование не выполнено

Преобразование к строке

Любой тип может быть преобразован к строке, т.е. к экземпляру класса **`String`**.

- Числовые типы записываются в текстовом виде без потери точности представления.
- Булевская величина приводится к строке "**`true`**" или "**`false`**" в зависимости от значения.
- Для объектных величин вызывается метод **`toString`**(). Если метод возвращает **`null`**, то результатом будет строка "**`null`**".
- Для **`null-значения`** генерируется строка "**`null`**".

Константы в Java

- **Символьные литералы**
  - 'a', 'z', '@'
  - управляющие последовательности

Управляющая последовательность	Описание
\ddd (три цифры!!!)	Восьмеричный символ UNICODE
\uxxxx (четыре цифры!!!)	Шестнадцатеричный символ UNICODE
\'	Одиночная кавычка
\"	Двойная кавычка
\\	Обратный слэш
\r	Возврат каретки
\n	Перевод строки
\f	Перевод страницы
\t	Символ табуляции (Tab)
\b	Возврат на один символ (backspace)

Ссылочные типы Java

В технологии Java, кроме простых типов все остальные типы ***ссылочные.***

Ссылочная переменная содержит дескриптор объекта.

Например:

```
public class MyDate {// декларация класса MyDate
   private int day = 1;
   private int month = 1;
   private int year = 2000;
    public MyDate(int day, int month, int year) // конструктор
   класса
   {…}
   public String toString() { ... }
   }
   public class TestMyDate {//класс для тестирования MyDate

   public static void main(String[] args)
   {
   MyDate today = new MyDate(22, 7, 1964);
   }
 }
```

Создание и инициализация объектов

Вызов ***new Obj1()*** выполняет следующие действия:

a) выделяется память для объекта.

б) выполняется явная инициализация атрибутов.

c) вызывается конструктор.

d) возвращается ссылка на объект оператором ***new***.

f) ссылка на объект присваивается переменной.

Пример:

```
MyDate my_birth = new  MyDate (22, 7, 1994);
```

Инициализация ссылок

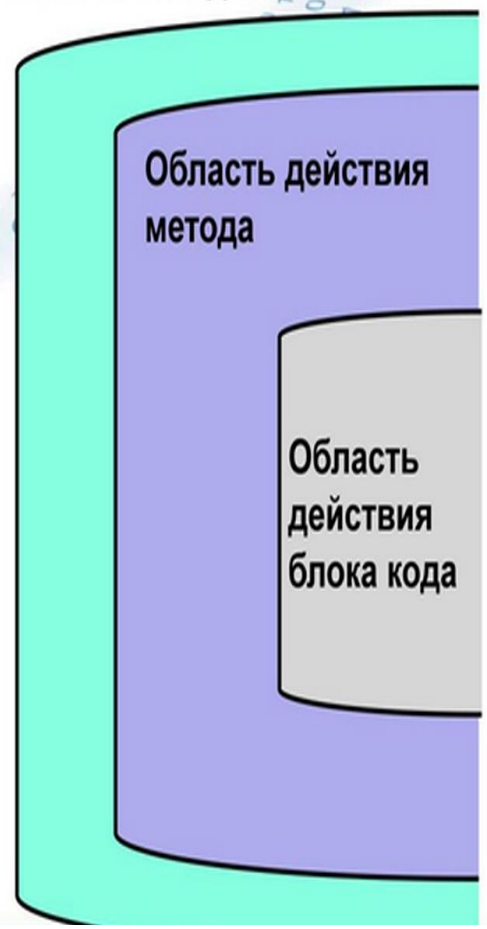
- Две переменные ссылаются на один и тот же объект:

```
1   int x = 7;
2   int y = x;
3   MyDate s = new MyDate(22, 7, 1964);
4   MyDate t = s;
```

- Переопределение ссылок создает две разные ссылки на два разных объекта:

```
5   t = new MyDate(22, 12, 1964);
```

Область действия и время жизни переменных




```
   class Scope
{public static void main(String args[])
   { int i;     //известна внутри main()
     int k=1;   //известна внутри main()

     for (i=0;i<3;i++)
   { //началась новая область действия
      int y =20; //y известна только в новой обл-ти
      int k; //ошибка - переменная к уже существует
    }
      y=100; //ошибка! y здесь не известна
}
}
```

В отличие от C/C++, в Java во внутренней области действия нельзя объявлять переменную с таким же именем как во внешней области действия.

Область действия и время жизни переменных

В Java-программах можно выделить две основных области действия: одна определяется классом, а другая методом.

Область действия, определяемая методом, начинается с его открывающей фигурной скобки и заканчивается закрывающей. Если метод имеет параметры, они также включаются в его область действия и работают как любая переменная метода.

Области действия могут быть вложены. Каждый раз когда создается новый блок кода ({ }), создается новая область действия. Внешняя область действия включает внутреннюю, т.е. объекты, объявленные во внешней области, будут действовать и во внутренней. Обратное – не верно, т.е. объекты, объявленные во внутренней области не действуют во внешней. Переменные можно объявлять в любой точке внутри блока кода, но область действия переменной начинается только после ее объявления.

При выполнении программы **переменные создаются в момент входа в их область действия и разрушаются при выходе из этой области**. Таким образом, время жизни переменной ограничено областью ее видимости.

Передача параметров по значению (Pass-by-Value)

- Язык Java передает аргументы только по значению.
- Когда **экземпляр объекта** передается в качестве аргумента методу, значение аргумента является ссылкой на объект.
- Содержимое объекта может быть изменено в вызываемом методе, при этом исходная ссылка на объект никогда не изменяется.

```
public class PassTest {
// Methods to change the current values
public static void changeInt(int value) {
value = 55;
}
public static void changeObjectRef(MyDate ref) {
ref = new MyDate(1, 1, 2000);
}
public static void changeObjectAttr(MyDate ref){
ref.setDay(4);
}
```

Передача параметров по значению (Pass-by-Value)

```
public static void main(String args[]) {
MyDate date;
int val;

// Assign the int
val = 11;
// Try to change it
chanqeInt(val);
// What is the current value?
System.out.println(" Int value is: " + val);
```

В результате получим:
Int value is: 11

Передача параметров по значению (Pass-by-Value)

```
// Назначим дату
date = new MyDate(22, 7, 1994);
// Попробуем ее изменить
changeObjectRef(date);
// Какое будет текущее значение?
System.out.println("MyDate: " + date);
```

В результате получим:
MyDate: 22.7.1994

✓Содержимое объекта может быть изменено в вызываемом методе, при этом исходная ссылка на объект никогда не изменяется.

```
// теперь попробуем изменить день
// передаем ссылку на объект
  changeObjectAttr(date);
// Какое будет текущее значение?
System.out.println("MyDate: " + date);
 }
}
```

В результате получим:
MyDate: 4.7.1994

✓Содержимое объекта было изменено с помощью метода setDay()

Ссылка this

Вот несколько способов использования этого ключевого слова:

• Чтобы устранить неоднозначность между экземпляром переменной и параметром

• Чтобы передать текущий объект в качестве параметра в другой метод или конструктор

```
public class MyDate {
private int day = 1;
private int month = 1;
private int year = 2000;

public MyDate(int day, int month, int
year) {
   this.day = day;
   this.month = month;
   this.year = year;
   }
public MyDate(MyDate date) {
   this.day = date.day;
   this.month = date.month;
   this.year = date.year;
  }
```

Ссылка this

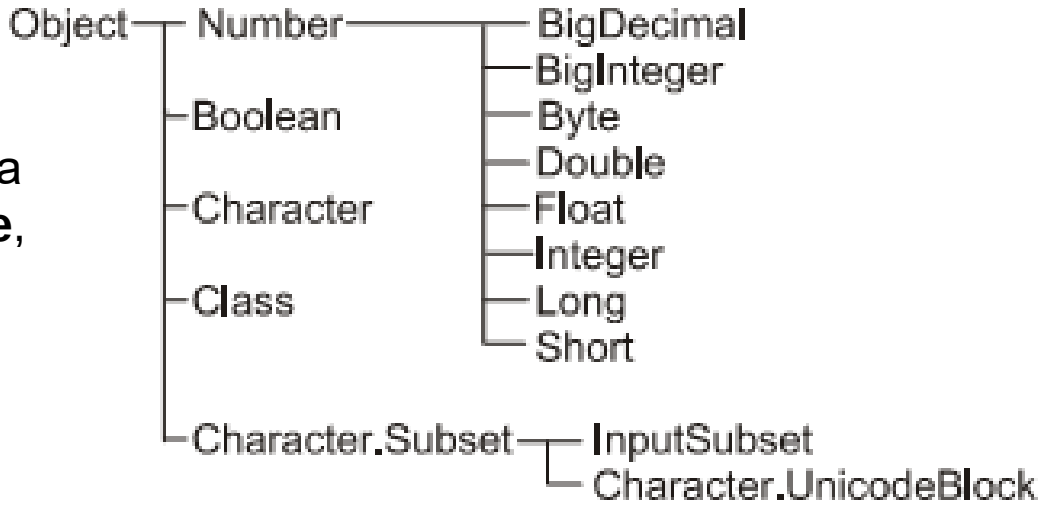
```
public MyDate addDays(int moreDays) {
MyDate newDate = new MyDate(this);
newDate.day = newDate.day + moreDays;
// Not Yet Implemented: wrap around code...
return newDate; }
public String toString() {
  return ("" + day + "." + month + "." + year);
}
}// end class MyDate
```

```
public class TestMyDate {
public static void main(String[] args) {
MyDate my_birth = new MyDate(22, 7, 1994);
MyDate the_next_week = my_birth.addDays(7) ;
  System.out.println(the_next_week); }
}
```

Классы - обертки

Кроме базовых типов данных, в языке Java широко используются соответствующие классы-оболочки (wrapper-классы) из пакета **java.lang**: **Boolean**, **Character**, **Integer**, **Byte**, **Short**, **Long**, **Float**, **Double**.

Объекты этих классов могут хранить те же значения, что и соответствующие им базовые типы.



Переменную базового типа можно преобразовать к соответствующему объекту, передав ее значение конструктору при объявлении объекта. Объекты класса могут быть преобразованы к любому базовому типу методами тип.**valueOf()** или обычным присваиванием:

```
Integer k1 = Integer.valueOf(55);
Integer k2 = Integer.valueOf(100);
Double  d1 = Double.valueOf(3.14);
```

В версии 5.0 введен процесс **автоматической инкапсуляции данных базовых типов в соответствующие объекты оболочки и обратно** (автоупаковка). При этом нет необходимости в создании соответствующего объекта с использованием оператора **new**. Например:

```
Integer k1 = 55;
```

Зачем нужны классы - обертки

Java — полностью объектно-ориентированный язык. Это означает, что все, что только можно, в Java представлено объектами.
Восемь примитивных типов нарушают это правило. Они оставлены в Java не только из-за многолетней привычки к числам и символам. Арифметические действия удобнее и быстрее производить с обычными числами, а не с объектами классов, которые требуют много ресурсов от компьютера.

Ссылаться на примитив мы не можем, значит, используем ссылку на "объект, похожий на примитив" - обёртку. Поэтому были созданы классы обертки Byte, Double, Float, Integer, Long, Short, Character, целью которых является предоставление унифицированного доступа к примитивным типам.

Классы - обертки обладают массой полезных функций, например: метод сравнения объектов equals (), переопределенный из класса object.

Все названные классы, кроме Boolean и class, имеют метод compareTo (), сравнивающий числовое значение, содержащееся в данном объекте, с числовым значением другого объекта.

Подробнее см. http://www.realcoding.net/article/view/1976

Полезные ссылки

- **Разбираемся с classpath в Java.**
  http://pr0java.blogspot.ru/2015/04/classpath-java.html
- **Управление Java classpath (Windows) (исходники)**
  http://www.interface.ru/home.asp?artId=7375