БФУ им. И.Канта

Лекция 3:
«Типы данных. Приведение типов. Массивы»

Дисциплина: «Язык программирования Java»

Преподаватель:
доцент кафедры математического моделирования и информационных систем,
к.т.н. Листопад Сергей Викторович

Калининград, 2013

Типы данных

Java является строго типизированным языком. Это означает, что любая переменная и любое выражение имеют известный тип еще на момент компиляции. Такое строгое правило позволяет выявлять многие ошибки уже во время компиляции.

Все типы данных разделяются на две группы. Первую составляют 8 простых, или примитивных (от английского primitive), типов данных. Они подразделяются на три подгруппы:

Целочисленные
- byte
- short
- int
- long
- char (также является целочисленным типом)

Дробные
- float
- double

Булевые
- boolean

Вторую группу составляют объектные, или ссылочные (от английского reference), типы данных. Это все классы, интерфейсы и массивы.

Переменные

Переменные используются в программе для хранения данных. Любая переменная имеет три базовых характеристики:

- Имя уникально идентифицирует переменную и позволяет обращаться к ней в программе;
- Тип описывает, какие величины может хранить переменная;
- Значение - текущая величина, хранящаяся в переменной на данный момент.

Работа с переменной всегда начинается с ее объявления (declaration). В Java любая переменная имеет строгий тип, который задается при объявлении и никогда не меняется. Значение может быть указано сразу (это называется инициализацией), а в большинстве случаев задание начальной величины можно и отложить. Некоторые примеры объявления переменных примитивного типа int с инициализаторами и без таковых:

```
int a;          int b = 0, c = 3+2;      int d = b+c;    int e = a = 5;
```

Объявление переменных и возможная инициализация при объявлении описываются следующим образом. Сначала указывается тип переменной, затем ее имя и, если необходимо, инициализатор, который может быть константой или выражением, вычисляемым во время компиляции или исполнения программы. В частности, можно пользоваться уже объявленными переменными. Далее можно поставить запятую и объявить новую переменную точно такого же типа.

При объявлении переменной может быть использовано ключевое слово final. Его указывают перед типом переменной, и тогда ее необходимо сразу инициализировать и уже больше никогда не менять ее значение. Простейший пример объявления final -переменной:

```
final double pi=3.1415;
```

Различия примитивных и ссылочных типов

Возьмем простой тип int:

```
int a=5;    // объявляем первую переменную и  инициализируем ее
int b=a;    // объявляем вторую переменную и приравниваем ее к
   первой
a=3;        // меняем значение первой, результат: a=3; b=5
```

Теперь рассмотрим ссылочный тип данных. Переменные таких типов всегда хранят ссылки на объекты:

```
class Point { int x, y;}
Point p1 = new Point(3,5);
Point p2=p1;
p1.x=7; // результат: p1.x=7, p1.y=5, p2.x=7, p2.y=5
```

Таким образом, примитивные переменные являются действительными хранилищами данных. Ссылочные же переменные хранят лишь ссылки на объекты, причем различные переменные могут ссылаться на один и тот же объект. Если же переменная сохраняет ссылку на другой объект она перестает видеть изменения, происходящие с прежним объектом:

```
p1 = new Point(4,9);
print(p2.x); // результат: p1.x=4, p1.y=9, p2.x=7, p2.y=5
```

Теперь легко понять смысл литерала null. Такое значение может принять переменная любого ссылочного типа. Это означает, что ее ссылка никуда не указывает, объект отсутствует. Любая попытка обратиться к объекту через такую переменную приведет к ошибке. Также значение null можно передать в качестве любого объектного аргумента при вызове функций.

Целочисленные типы

Название типа	Длина (байты)	Область значений
byte	1	-128 .. 127
short	2	-32.768 .. 32.767
int	4	-2.147.483.648 .. 2.147.483.647
long	8	-9.223.372.036.854.775.808 .. 9.223.372.036.854.775.807 (примерно 10^{19})
char	2	'\u0000' .. '\uffff', или 0 .. 65.535

Целочисленные литералы имеют тип int по умолчанию, или тип long, если стоит буква L или l. Именно поэтому корректным литералом считается только такое число, которое укладывается в 4 или 8 байт, соответственно. Иначе компилятор сочтет это ошибкой. Таким образом, следующие литералы являются корректными: 1 -2147483648 2147483648L 0L 1111111111111111L. Над целочисленными аргументами можно производить следующие операции:

- операции сравнения (возвращают булево значение)
  - <, <=, >, >=
  - ==, !=
- числовые операции (возвращают числовое значение)
  - унарные операции + и -
  - арифметические операции +, -, *, /, %
  - операции инкремента и декремента (в префиксной и постфиксной форме): ++ и --
  - операции битового сдвига <<, >>, >>>
  - битовые операции ~, &, |, ^
- оператор с условием ?:
- оператор приведения типов
- оператор конкатенации со строкой +

Целочисленные типы. Особенности

Хотя целочисленные типы имеют длину 8, 16, 32 и 64 бита, вычисления проводятся только с 32-х и 64-х битной точностью. А это значит, что перед вычислениями может потребоваться преобразовать тип одного или нескольких операндов. Если хотя бы один аргумент операции имеет тип long, то все аргументы приводятся к этому типу и результат операции также будет типа long. Вычисление будет произведено с точностью в 64 бита, а более старшие биты, если таковые появляются в результате, отбрасываются. Если же аргументов типа long нет, то вычисление производится с точностью в 32 бита, и все аргументы преобразуются в int (это относится к byte, short, char). Результат также имеет тип int. Все биты старше 32-го игнорируются. Рассмотрим пример:

```
int i=300000;
print(i*i);   // умножение с точностью 32 бита: -194313216
long m=i;
print(m*m);   // умножение с точностью 64 бита: 90000000000
print(1/(m-i)); // попробуем получить разность значений int и long: ошибка деления на ноль
```

Примеры, которые могут создать проблемы при написании программ:

```
double x = 1/2; // x=0
```

Попробуем вычислить, сколько миллисекунд содержится в неделе и в месяце:

```
print(1000*60*60*24*7);       // вычисление для недели: 604800000
print(1000*60*60*24*30);      // вычисление для месяца: -1702967296
print(1000*60*60*24*30L);     // вычисление для месяца: 2592000000
```

Понятно, что типы большей длины могут хранить больший спектр значений, а потому Java не позволяет присвоить переменной меньшего типа значение большего типа. Например, такие строки вызовут ошибку компиляции:

```
int x=1;          byte b=x;      // пример вызовет ошибку компиляции
byte b=1;         byte c=b+1; // пример вызовет ошибку компиляции
byte b=5;         byte c=-b;    // пример вызовет ошибку компиляции
```

Целочисленные типы. Особенности

Итак, все числовые операторы возвращают результат типа int или long. Однако существует два исключения:операторы инкрементации и декрементации:

```
byte x=5;       byte y1=x++;   // на момент начала исполнения x равен 5
```

и оператор с условием ?:. Если второй и третий операнды имеют одинаковый тип, то и результат операции будет такого же типа:

```
byte x=2;       byte y=3;          byte z=(x>y) ? x : y;        // верно, x и y одинакового типа
byte abs=(x>0) ? x : -x;      // неверно!
```

Наконец, рассмотрим оператор конкатенации со строкой. Оператор + может принимать в качестве аргумента строковые величины. Если одним из аргументов является строка, а вторым – целое число, то число будет преобразовано в текст и строки объединятся.

```
int x=1;                    print("x="+x); // результатом будет:x=1
print(1+2+"text");          print("text"+1+2); //результатом будет: 3texttext12
```

Отдельно рассмотрим работу с типом char. Значения этого типа могут полноценно участвовать в числовых операциях:

```
char c1=10;                 char c2='A';      // латинская буква A (\u0041, код 65)          int i=c1+c2-'B';
```

//Переменная i получит значение 9.

Рассмотрим следующий пример:

```
char c='A';       print(c);    print(c+1);       print("c="+c);    print('c'+'='+c);  //A66c=A225
```

Для каждого примитивного типа существуют специальные вспомогательные классы-обертки (wrapper classes). Для типов byte, short, int, long, char это Byte, Short, Integer, Long, Character. Эти классы содержат многие полезные методы для работы с целочисленными значениями. Например, преобразование из текста в число.

Единственные операции с целыми числами, при которых Java генерирует ошибки,– это деление на ноль (операторы / и %).

Дробные типы

Название типа	Длина (байты)	Область значений
float	4	3.40282347e+38f ; 1.40239846e-45f
double	8	1.79769313486231570e+308 ; 4.94065645841246544e-324

Для целочисленных типов область значений задавалась верхней и нижней границами, весьма близкими по модулю. Для дробных типов добавляется еще одно ограничение – насколько можно приблизиться к нулю, другими словами – каково наименьшее положительное ненулевое значение. Таким образом, нельзя задать литерал заведомо больший, чем позволяет соответствующий тип данных, это приведет к ошибке overflow. И нельзя задать литерал, значение которого по модулю слишком мало для данного типа, компилятор сгенерирует ошибку underflow.

```
float f = 1e40f; // значение слишком велико, overflow
double d = 1e-350;   // значение слишком мало, underflow
```

Над дробными аргументами можно производить следующие операции:

- операции сравнения (возвращают булево значение)
  - <, <=, >, >=
  - ==, !=
- числовые операции (возвращают числовое значение)
  - унарные операции + и -
  - арифметические операции +, -, *, /, %
  - операции инкремента и декремента (в префиксной и постфиксной форме): ++ и --
- оператор с условием ?:
- оператор приведения типов
- оператор конкатенации со строкой +

Практически все операторы действуют по тем же принципам, которые предусмотрены для целочисленных операторов. Операторы сравнения корректно работают и в случае сравнения целочисленных значений с дробными.

Дробные типы. Особенности

Java никак не сообщает о переполнениях. Деление на ноль не приводит к некорректной ситуации.Такая свобода связана с наличием специальных значений дробного типа:

- положительная и отрицательная бесконечности (positive/negative infinity);
- значение "не число", Not-a-Number, сокращенно NaN ;
- положительный и отрицательный нули.

Положительную и отрицательную бесконечности можно получить:

```
1f/0f    // положительная бесконечность, тип float
-1d/0d   // отрицательная бесконечность, тип double
```

Также в классах Float и Double определены константы POSITIVE_INFINITY и NEGATIVE_INFINITY. Как видно из примера, такие величины получаются при делении конечных величин на ноль. Значение NaN можно получить, например, в результате следующих действий:

```
0.0/0.0         // деление ноль на ноль
(1.0/0.0)*0.0   // умножение бесконечности на ноль
```

Эта величина также представлена константами NaN в классах Float и Double. Величины положительный и отрицательный ноль записываются очевидным образом:

```
0.0    // дробный литерал со значением   положительного нуля
+0.0   // унарная операция +, ее значение - положительный ноль
-0.0   // унарная операция -, ее значение - отрицательный ноль
```

Все дробные значения строго упорядочены. Отрицательная бесконечность меньше любого другого дробного значения, положительная – больше. Значения +0.0 и -0.0 считаются равными. Однако другие операторы различают их, например, выражение 1.0/0.0 дает положительную бесконечность, а 1.0/-0.0 – отрицательную.

Единственное исключение - значение NaN. Если хотя бы один из аргументов операции сравнения равняется NaN, то результат заведомо будет false (для оператора != соответственно всегда true). Таким образом, единственное значение x, при котором выражение x!=x истинно,– именно NaN.

Дробные типы. Особенности

Если результат слишком велик по модулю (overflow), результат – +/- бесконечность.

```
print(1e20f*1e20f); //Infinity
print(-1e200*1e200); //-Infinity
```

Если результат слишком мал (underflow), он округляется до нуля:

```
print(1e-40f/1e10f);   // underflow для float: 0.0
print(-1e-300/1e100);  // underflow для double: -0.0
float f=1e-6f;   print(f); //1.0E-6
f+=0.002f;       print(f); // 0.002001
f+=3;            print(f); // 3.002001
f+=4000;         print(f); // 4003.002    //4003,001953125
```

Если хотя бы один аргумент имеет тип double, то значения всех аргументов приводятся к этому типу и результат операции также будет иметь тип double. Вычисление будет произведено с точностью в 64 бита. Если же аргументов типа double нет, а хотя бы один аргумент имеет тип float, то все аргументы приводятся к float, вычисление производится с точностью в 32 бита и результат имеет тип float. Эти утверждения верны и в случае, если один из аргументов целочисленный. Если хотя бы один из аргументов имеет значение NaN, то и результатом операции будет NaN.

```
print(1/2); // 0
print(1/2.); // 0.5
```

Более сложный пример:

```
int x=3;      int y=5;
print (x/y); //0
print((double)x/y);  //0.6
print(1.0*x/y); //0.6
```

При приведении дробных значений к целым типам дробная часть просто отбрасывается. При приведении значений int к типу float и при приведении значений типа long к типу float и double возможны потери точности.

Булев тип

Булев тип boolean может хранить два возможных значения – true и false . Величины этого типа получаются в результате операций сравнения.

Над булевыми аргументами можно производить следующие операции:

- операции сравнения (возвращают булево значение)
  - ==, !=
- логические операции (возвращают булево значение)
  - !
  - &, |, ^
  - &&, ||
- оператор с условием ?:
- оператор конкатенации со строкой +

В операторе с условием ?: первым аргументом может быть только значение типа boolean. Также допускается, чтобы второй и третий аргументы одновременно имели булев тип.

Операция конкатенации со строкой превращает булеву величину в текст "true" или "false" в зависимости от значения.

Только булевы выражения допускаются для управления потоком вычислений, например, в качестве критерия условного перехода if.

Никакое число не может быть интерпретировано как булево выражение. Если предполагается, что ненулевое значение эквивалентно истине (по правилам языка C), то необходимо записать x!=0. Ссылочные величины можно преобразовывать в boolean выражением ref!=null.

Объекты и правила работы с ними

Объект – экземпляр некоторого класса, или экземпляр массива. Класс – описание объектов одинаковой структуры, и если в программе такой класс используется, то описание присутствует в единственном экземпляре. Объектов этого класса может не быть вовсе, а может быть создано сколь угодно много.

Объекты создаются с использованием ключевого слова **new**, причем одно слово **new** порождает строго один объект (или ни одного, если происходит ошибка). После ключевого слова указывается имя класса, от которого мы собираемся породить объект. Создание объекта всегда происходит через вызов одного из конструкторов класса (их может быть несколько), поэтому в заключение ставятся скобки, в которых перечислены значения аргументов, передаваемых конструктору. Определение класса Point:

```
class Point {
  int x, y;
   /**
      * Конструктор принимает 2 аргумента,
      * которыми инициализирует поля объекта.
      */
   Point (int newx, int newy){
      x=newx;      y=newy;
   }}
```

Объекты и правила работы с ними

Если конструктор отработал успешно, то выражение new возвращает ссылку на созданный объект. Эту ссылку можно сохранить в переменной, передать в качестве аргумента в какой-либо метод или использовать другим способом. JVM всегда занимается подсчетом хранимых ссылок на каждый объект. Как только обнаруживается, что ссылок больше нет, такой объект предназначается для уничтожения сборщиком мусора (garbage collector). Восстановить ссылку на такой "потерянный" объект невозможно.

```
Point p=new Point(1,2);    // Создали объект, получили на него ссылку
Point p1=p;                // теперь есть 2 ссылки на точку (1,2)
p=new Point(3,4);          // осталась одна ссылка на точку (1,2)
p1=null; // Ссылок на объект-точку (1,2) больше нет, он будет уничтожен сборщиком мусора.
```

Любой объект порождается ключевым словом new, исключение – экземпляры класса String:

```
String s="abc"+"def“; // создано три объекта класса String: два - литералы, третий - результат конкатенации.
```

Операция создания объекта – одна из самых ресурсоемких в Java. Кроме того, в версии Java 1.1 была введена технология reflection, которая позволяет обращаться к классам, методам и полям, используя лишь их имя в текстовом виде. С ее помощью также можно создать объект без ключевого слова new, однако эта технология довольно специфична, применяется в редких случаях, а кроме того, довольно проста и потому в данном курсе не рассматривается.

Операции над объектами

Над ссылочными значениями можно производить следующие операции:

- обращение к полям и методам объекта (осуществляется она с помощью символа . (точка))
- оператор instanceof (возвращает булево значение)
- операции сравнения == и != (возвращают булево значение)
- оператор приведения типов
- оператор с условием ?:
- оператор конкатенации со строкой +

Объект всегда "помнит", от какого класса он был порожден. С другой стороны, как уже указывалось, можно ссылаться на объект, используя ссылку другого типа:

```
// Объявляем класс Parent
class Parent {}
// Объявляем класс Child и наследуем его от класса Parent
class Child extends Parent {}
Parent p = new Child();
```

instanceof

Используя оператор instanceof, можно узнать, от какого класса произошел объект. Этот оператор имеет два аргумента. Слева указывается ссылка на объект, а справа – имя типа, на совместимость с которым проверяется объект. Например:

```
Parent p = new Child();// проверяем переменную p типа Parent
// на совместимость с типом Child
print(p instanceof Child); // true.
```

Оператор instanceof опирается не на тип ссылки, а на свойства объекта, на который она ссылается. Но этот оператор возвращает истинное значение не только для того типа, от которого был порожден объект:

```
// Объявляем новый класс и наследуем его от класса Child
class ChildOfChild extends Child { }
Parent p = new ChildOfChild();
print(p instanceof Child); //true
```

Добавим еще один класс:

```
class Child2 extends Parent {}
Parent p=new Child();
print(p instanceof Child); //true
print(p instanceof Child2); //false
```

Для ссылки, равной null, оператор instanceof всегда вернет значение false.

Операторы сравнения == и !=

Операторы сравнения == и != проверяют равенство (или неравенство) объектных величин по ссылке. Однако часто требуется альтернативное сравнение – по значению. При сравнении по ссылке ни тип объекта, ни значения его полей не учитываются, true возвращается только в том случае, если обе ссылки указывают на один и тот же объект:

```
Point p1=new Point(2,3);
Point p2=p1;
Point p3=new Point(2,3);
print(p1==p2);      //true
print(p1==p3);      //false
```

Если один из аргументов оператора == равен null, а другой – нет, то значение такого выражения будет false. Если же оба операнда null, то результат будет true. Для корректного сравнения по значению существует специальный метод equals:

```
String s = "abc";
s=s+1;
print(s.equals("abc1")); //true
```

Операция с условием ?: работает как обычно и может принимать второй и третий аргументы, если они оба одновременно ссылочного типа. Результат такого оператора также будет иметь объектный тип.

Как и простые типы, ссылочные величины можно складывать со строкой. Если ссылка равна null, то к строке добавляется текст "null". Если ссылка указывает на объект, то у него вызывается специальный метод toString() и текст, который он вернет, будет добавлен к строке.

Класс Object

Класс Object – базовый класс для всех классов - именно от него наследуются все классы, в объявлении которых явно не указан другой родительский класс. А значит, любой класс напрямую, или через своих родителей, является наследником Object. Отсюда следует, что методы этого класса есть у любого объекта (поля в Object отсутствуют), а потому они представляют особенный интерес.

Рассмотрим основные из них.

getClass() - метод возвращает объект класса Class, который описывает класс, от которого был порожден этот объект. У него есть метод getName(), возвращающий имя класса:

```
String s = "abc";
Class cl=s.getClass();
System.out.println(cl.getName()); //java.lang.String
```

В отличие от оператора instanceof, метод getClass() всегда возвращает точно тот класс, от которого был порожден объект.

equals() - метод имеет один аргумент типа Object и возвращает boolean. Как уже говорилось, equals() служит для сравнения объектов по значению, а не по ссылке. Сравнивается состояние объекта, у которого вызывается этот метод, с передаваемым аргументом.

```
Point p1=new Point(2,3); Point p2=new Point(2,3); print(p1.equals(p2)); // false
```

Поскольку сам Object не имеет полей, а значит, и состояния, в этом классе метод equals возвращает результат сравнения по ссылке. Однако при написании нового класса можно переопределить этот метод и описать правильный алгоритм сравнения по значению:

```
public boolean equals(Object o) {// Нужно убедиться, что объект «o» совместим с типом Point
  if (o instanceof Point)  {
    Point p = (Point)o;         // Типы совместимы, можно провести преобразование
    return p.x==x && p.y==y; // Возвращаем результат сравнения координат
  }   // Если объект не совместим с Point, возвращаем false
  return false;}
```

Класс Object

hashCode() - метод возвращает значение int. Цель hashCode() – представить любой объект целым числом. Особенно эффективно это используется в хэш-таблицах (в Java есть стандартная реализация такого хранения данных). Конечно, нельзя потребовать, чтобы различные объекты возвращали различные хэш-коды, но необходимо, чтобы объекты, равные по значению, возвращали одинаковые хэш-коды.
В классе Object этот метод реализован на уровне JVM. Сама виртуальная машина генерирует число хеш-кодов, основываясь на расположении объекта в памяти.

toString() - метод позволяет получить текстовое описание любого объекта. Создавая новый класс, данный метод можно переопределить и возвращать более подробное описание. Для класса Object и его наследников, не переопределивших toString(), метод возвращает следующее выражение: getClass().getName()+"@"+hashCode(). Например:

print(new Object()); // java.lang.Object@92d342

Именно этот метод вызывается при конвертации объекта в текст, когда он передается в качестве аргумента оператору конкатенации строк.

finalize() - метод вызывается при уничтожении объекта сборщиком мусора. В классе Object он ничего не делает, однако в классе-наследнике позволяет описать все действия, необходимые для корректного удаления объекта, такие как закрытие соединений с БД, сетевых соединений, снятие блокировок на файлы и т.д. В обычном режиме напрямую этот метод вызывать не нужно, он отработает автоматически. Если необходимо, можно обратиться к нему явным образом. Нужно описывать только дополнительные действия, связанные с логикой работы программы. Все необходимое для удаления объекта JVM сделает сама.

Класс String

Как уже указывалось, класс String занимает в Java особое положение. Экземпляры только этого класса можно создавать без использования ключевого слова new. Каждый строковый литерал порождает экземпляр String, и это единственный литерал (кроме null), имеющий объектный тип. Затем значение любого типа может быть приведено к строке с помощью оператора конкатенации строк, который был рассмотрен для каждого типа, как примитивного, так и объектного. Еще одним важным свойством данного класса является неизменяемость. Это означает, что, породив объект, содержащий некое значение-строку, мы уже не можем изменить данное значение – необходимо создать новый объект.

```
String s="a";
s="b"; // переменная сменила свое значение, но только создав новый объект
```

Поскольку каждый строковый литерал порождает новый объект, что есть очень ресурсоемкая операция в Java, зачастую компилятор стремится оптимизировать эту работу. Во-первых, если используется несколько литералов с одинаковым значением, для них будет создан один и тот же объект.

```
String s1 = "abc";
String s2 = "abc";
String s3 = "a"+"bc";
print(s1==s2); //true
print(s1==s3); //true
```

Если же строка создается выражением, которое может быть вычислено только во время исполнения программы, то оно будет порождать новый объект:

```
String s1="abc";
String s2="ab";
print(s1==(s2+"c")); //false
```

В классе String определен метод intern(), который возвращает один и тот же объект-строку для всех экземпляров, равных по значению. То есть если для ссылок s1 и s2 верно выражение s1.equals(s2), то верно и s1.intern()==s2.intern(). Разумеется, в классе переопределены методы equals(), hashCode(), toString().

Класс Class

Класс Class - метакласс для всех классов Java. Когда JVM загружает файл .class, который описывает некоторый тип, в памяти создается объект класса Class, который будет хранить это описание. Например, если в программе есть строка:

```
Point p=new Point(1,2);
```

то это означает, что в системе созданы следующие объекты:

- объект типа Point, описывающий точку (1,2) ;
- объект класса Class, описывающий класс Point ;
- объект класса Class, описывающий класс Object. Поскольку класс Point наследуется от Object, его описание также необходимо;
- объект класса Class, описывающий класс Class. Это обычный Java-класс, который должен быть загружен по общим правилам.

Одно из применений класса Class уже было рассмотрено – использование метода getClass() класса Object. Если продолжить последний пример с точкой:

```
Class cl=p.getClass();        // это объект №2 из списка
Class cl2=cl.getClass();      // это объект №4 из списка
Class cl3=cl2.getClass();     // опять объект №4, выражение cl2==cl3 верно.
```

Приведения

Java является строго типизированным языком, каждое выражение и переменная имеет строго определенный тип уже на момент компиляции. Тип устанавливается на основе структуры применяемых выражений и типов литералов, переменных и методов, используемых в этих выражениях. Например:

```
long a=3;
a = 5+'A'+a;
print("a="+Math.round(a/2F));
```

В первой строке литерал 3 имеет тип по умолчанию, то есть int. При присвоении этого значения переменной типа long необходимо провести преобразование.

Во второй строке сначала производится сложение значений типа int и char. Второй аргумент будет преобразован так, чтобы операция проводилась с точностью в 32 бита. Второй оператор сложения потребует преобразования, так как наличие переменной a увеличивает точность до 64 бит.

В третьей строке сначала будет выполнена операция деления, для чего значение long надо будет привести к типу float, так как второй операнд - дробный литерал. Результат будет передан в метод Math.round, который произведет математическое округление и вернет целочисленный результат типа int. Это значение необходимо преобразовать в текст, чтобы осуществить дальнейшую конкатенацию строк.

Пример показывает, что даже простые строки могут содержать многочисленные преобразования, зачастую незаметные для разработчика. Часто бывают и такие случаи, когда программисту необходимо явно изменить тип некоторого выражения или переменной, например, чтобы воспользоваться подходящим методом или конструктором.

Пример:

```
int b=1;
byte c=(byte)-b;      // необходимо провести явное преобразование
int i=c;              // обратное приведение производится автоматически
```

Виды приведений

В Java предусмотрено семь видов 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).
- Рассмотрим их по отдельности.
- Тождественное преобразование

Тождественное преобразование

Зачем нужно тождественное приведение? Есть две причины для того, чтобы выделить такое преобразование в особый вид.

- Во-первых, с теоретической точки зрения теперь можно утверждать, что любой тип в Java может участвовать в преобразовании, хотя бы в тождественном. Например, примитивный тип boolean нельзя привести ни к какому другому типу, кроме него самого.
- Во-вторых, иногда в Java могут встречаться такие выражения, как длинный последовательный вызов методов:

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

При исполнении такого выражения сначала вызывается первый метод getCity(). Можно предположить, что возвращаемым значением будет объект класса City. У этого объекта далее будет вызван следующий метод getStreet(). Чтобы узнать, значение какого типа он вернет, необходимо посмотреть описание класса City. У этого значения будет вызван следующий метод (getHouse()), и так далее. Чтобы узнать результирующий тип всего выражения, необходимо просмотреть описание каждого метода и класса.

Компилятор без труда справится с такой задачей, однако разработчику будет нелегко проследить всю цепочку. В этом случае можно воспользоваться тождественным преобразованием, выполнив приведение к точно такому же типу. Это ничего не изменит в структуре программы, но значительно облегчит чтение кода:

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

Расширение примитивных типов

Для простых типов расширение означает, что осуществляется переход от менее емкого типа к более емкому. Например, от типа byte (длина 1 байт) к типу int (длина 4 байта). Такие преобразования безопасны в том смысле, что новый тип всегда гарантированно вмещает в себя все данные, которые хранились в старом типе, и таким образом не происходит потери данных. Именно поэтому компилятор осуществляет его сам, незаметно для разработчика:

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

В последней строке значение переменной b типа byte будет преобразовано к типу переменной a (то есть, int) автоматически, никаких специальных действий для этого предпринимать не нужно.

Следующие 19 преобразований являются расширяющими:

- 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

Нельзя провести преобразование к типу char от типов меньшей или равной длины (byte, short), или, наоборот, к short от char без потери данных. Это связано с тем, что char, в отличие от остальных целочисленных типов, является беззнаковым. Даже при расширении данные все-таки могут быть в искажены. Это приведение значений int к типу float и приведение значений типа long к типу float или double. Хотя эти дробные типы вмещают гораздо большие числа, чем соответствующие целые, но у них меньше значащих разрядов:

```
long a=111111111111L;
float f = a;
a = (long) f;
print(a); // 111111110656
```

Сужение примитивных типов

Сужение означает, что переход осуществляется от более емкого типа к менее емкому. При таком преобразовании есть риск потерять данные. Например, если число типа int было больше 127, то при приведении его к byte значения битов старше восьмого будут потеряны. В Java такое преобразование должно совершаться явным образом, т.е. программист в коде должен явно указать, что он намеревается осуществить такое преобразование и готов потерять данные.

Следующие 23 преобразования являются сужающими:

- 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

При сужении целочисленного типа к более узкому целочисленному все старшие биты, не попадающие в новый тип, просто отбрасываются. Не производится никакого округления или других действий для получения более корректного результата:

```
print((byte)383); //127
print((byte)384); //-128
print((byte)-384); //-128
char c=40000;
print((short)c); //-25536
```

Сужение примитивных типов

Сужение дробного типа до целочисленного является более сложной процедурой. Она проводится в два этапа.

- Дробное значение преобразуется в long, если целевым типом является long, или в int - в противном случае (целевой тип byte, short, char, int). Для этого исходное дробное число сначала математически округляется в сторону нуля. При этом могут возникнуть особые случаи:
  - исходное дробное значение - NaN, результат - 0 типа int или long;
  - если исходное дробное значение является положительной или отрицательной бесконечностью, результатом - максимально или минимально возможное значение выбранного типа int или long;
  - если дробное значение - конечная величина, но в результате округления получилось слишком большое по модулю число для выбранного типа int или long, результат - максимальное или минимальное значение этого типа.
  - если результат округления укладывается в диапазон значений выбранного типа, то он и будет результатом первого шага.
- На втором шаге производится дальнейшее сужение от выбранного целочисленного типа к целевому, если таковое требуется, то есть может иметь место дополнительное преобразование от int к byte, short или char.

```
float fmin = Float.NEGATIVE_INFINITY;          float fmax = Float.POSITIVE_INFINITY;
print("long: "+(long)fmin+".."+(long)fmax);
                    //long: -9223372036854775808..9223372036854775807
print("int: " + (int)fmin + ".." +       (int)fmax);          //int: -2147483648..2147483647
print("short: " + (short)fmin + ".." +       (short)fmax);           //short: 0..-1
print("char: " + (int)(char)fmin + ".." +       (int)(char)fmax);   //char: 0..65535
print("byte: " + (byte)fmin + ".." +       (byte)fmax);        // byte: 0..-1
```

Расширение ссылочных типов

class Parent { int x;} // Объявляем класс Parent

class Child extends Parent {int y;} // наследуем класс Child от класса Parent

class Child2 extends Parent {int z;} // наследуем класс Child2 от класса Parent

Объекты класса Parent обладают одним полем x, только ссылки Parent могут ссылаться на такие объекты. Объекты Child обладают полем y и полем x, полученным по наследству от Parent. На такие объекты могут указывать ссылки типа Child или Parent:

Parent p = new Child();

С помощью такой ссылки p можно обращаться лишь к полю x созданного объекта. Поле y недоступно, так как компилятор, проверяя корректность выражения p.y, не может предугадать, что ссылка p будет указывать на объект типа Child во время исполнения программы. Он анализирует лишь тип самой переменной, а она объявлена как Parent, но в этом классе нет поля y, что и вызовет ошибку компиляции.

Ссылки типа Parent могут указывать на объект любого из трех рассматриваемых типов, а ссылки типа Child и Child2 - только на объекты точно такого же типа.

Расширение означает переход от более конкретного типа к менее конкретному, т.е. переход от детей к родителям. В нашем примере преобразование от любого наследника (Child, Child2) к родителю (Parent) есть расширение, переход к более общему типу. Подобно случаю с примитивными типами, этот переход производится самой JVM при необходимости и незаметен для разработчика, то есть не требует никаких дополнительных усилий, так как он всегда проходит успешно: всегда можно обращаться к объекту, порожденному от наследника, по типу его родителя.

Parent p1=new Child(); Parent p2=new Child2();

В обеих строках переменным типа Parent присваивается значение другого типа, а значит, происходит преобразование. Поскольку это расширение, оно производится автоматически и всегда успешно. При подобном преобразовании с самим объектом ничего не происходит.

Следующие преобразования являются расширяющими:

- от класса A к классу B, если A наследуется от B;
- от null -типа к любому объектному типу.

Сужение ссылочных типов

Обратный переход, то есть движение по дереву наследования вниз, к наследникам, является сужением. Например, для рассматриваемого случая, переход от ссылки типа Parent, которая может ссылаться на объекты трех классов, к ссылке типа Child, которая может ссылаться на объекты лишь одного из трех классов, очевидно, является сужением. Такой переход может оказаться невозможным. Если ссылка типа Parent ссылается на объект типа Parent или Child2, то переход к Child невозможен, ведь в обоих случаях объект не обладает полем y, которое объявлено в классе Child. Поэтому при сужении разработчику необходимо явным образом указывать на то, что необходимо попытаться провести такое преобразование. JVM во время исполнения проверит корректность перехода. Если он возможен, преобразование будет проведено. Если же нет - возникнет ошибка.

```
Parent p=new Child();
Child c=(Child)p;      // преобразование будет успешным.
Parent p2=new Child2();
Child c2=(Child)p2;      // во время исполнения возникнет ошибка!
```

Чтобы проверить, возможен ли желаемый переход, можно воспользоваться оператором instanceof:

```
Parent p=new Child();
if (p instanceof Child) {Child c = (Child)p;}
Parent p2=new Child2();
if (p2 instanceof Child) {Child c = (Child)p2;}
Parent p3=new Parent();
if (p3 instanceof Child) {Child c = (Child)p3;}
```

На данный момент можно назвать лишь одно сужающее преобразование: от класса A к классу B, если B наследуется от A (важным частным случаем является сужение типа Object до любого другого ссылочного типа).

Преобразование к строке

Любой тип может быть приведен к строке, т.е. к экземпляру класса String. Такое преобразование является исключительным в силу того, что охватывает абсолютно все типы, в том числе и boolean, про который говорилось, что он не может участвовать ни в каком другом приведении, кроме тождественного.

Числовые типы записываются в текстовом виде без потери точности представления. Формально такое преобразование происходит в два этапа. Сначала на основе примитивного значения порождается экземпляр соответствующего класса-"обертки", а затем у него вызывается метод toString(). Но поскольку эти действия снаружи незаметны, многие JVM оптимизируют их и преобразуют примитивные значения в текст напрямую.

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".

Запрещенные преобразования

Не все переходы между произвольными типами допустимы. Например, к запрещенным преобразованиям относятся: переходы от любого ссылочного типа к примитивному, от примитивного - к ссылочному (кроме преобразований к строке). Уже упоминавшийся пример - тип boolean - нельзя привести ни к какому другому типу, кроме boolean (как обычно - за исключением приведения к строке). Затем, невозможно привести друг к другу типы, находящиеся не на одной, а на соседних ветвях дерева наследования. В примере, который рассматривался для иллюстрации преобразований ссылочных типов, переход от Child к Child2 запрещен. В самом деле, ссылка типа Child может указывать на объекты, порожденные только от класса Child или его наследников. Это исключает вероятность того, что объект будет совместим с типом Child2.

Этим список запрещенных преобразований не исчерпывается. Он довольно велик, и в то же время все варианты достаточно очевидны. Разумеется, попытка осуществить запрещенное преобразование вызовет ошибку компиляции.

Применение приведений

Ситуации в коде, где могут встретиться или потребоваться приведения:

- Присвоение значений переменным (assignment). Не все переходы допустимы при таком преобразовании - ограничения выбраны таким образом, чтобы не могла возникнуть ошибочная ситуация.
- Вызов метода. Это преобразование применяется к аргументам вызываемого метода или конструктора. Допускаются почти те же переходы, что и для присвоения значений. Такое приведение никогда не порождает ошибок. Так же приведение осуществляется при возвращении значения из метода.
- Явное приведение. В этом случае явно указывается, к какому типу требуется привести исходное значение. Допускаются все виды преобразований, кроме приведений к строке и запрещенных. Может возникать ошибка времени исполнения программы.
- Оператор конкатенации производит преобразование к строке своих аргументов.
- Числовое расширение (numeric promotion). Числовые операции могут потребовать изменения типа аргумента(ов). Это преобразование имеет особое название - расширение (promotion), так как выбор целевого типа может зависеть не только от исходного значения, но и от второго аргумента операции.

Массивы

В отличие от обычных переменных, которые хранят только одно значение, массивы (arrays) используются для хранения целого набора значений. Количество значений в массиве называется его длиной, сами значения – элементами массива. Значений может не быть вовсе, в этом случае массив считается пустым, а его длина равной нулю.

Элементы не имеют имен, доступ к ним осуществляется по номеру индекса. Если массив имеет длину n, отличную от нуля, то корректными значениями индекса являются числа от 0 до n-1. Все значения имеют одинаковый тип и говорится, что массив основан на этом базовом типе. Массивы могут быть основаны как на примитивных типах (например, для хранения числовых значений 100 измерений), так и на ссылочных (например, если нужно хранить описание 100 автомобилей в гараже в виде экземпляров класса Car).

Сразу оговоримся, что в Java массив символов char[] и класс String являются различными типами. Их значения могут легко конвертироваться друг в друга с помощью специальных методов, но все же они не относятся к идентичным типам.

Как уже говорилось, массивы в Java являются объектами (примитивных типов в Java всего восемь и их количество не меняется), их тип напрямую наследуется от класса Object, поэтому все элементы данного класса доступны у объектов-массивов.

Базовый тип также может быть массивом. Таким образом конструируется массив массивов, или многомерный массив.

Объявление массивов

В качестве примера рассмотрим объявление переменной типа массив:

```
int a[];
int[] a;
int[] a[];
Point p, p1[], p2[][];
```

Создание переменной типа массив еще не создает экземпляры этого массива. Такие переменные имеют объектный тип и хранят ссылки на объекты, однако изначально имеют значение null (если они являются полями класса; напомним, что локальные переменные необходимо явно инициализировать). Чтобы создать экземпляр массива, нужно воспользоваться ключевым словом new, после чего указывается тип массива и в квадратных скобках – длина массива.

```
int a[]=new int[5];
Point[] p = new Point[10];
int array[]=new int[5];
for (int i=0; i<5; i++) {
    array[i]=i*i;
}
for (int j=0; j<5; j++) {
   System.out.println(j+"*"+j+"="+array[j]);
}
```

Результатом выполнения программы будет: 0*0=01*1=12*2=43*3=94*4=16. Если бы индекс превысил максимально возможное для такого массива значение, то появилась бы ошибка времени исполнения. Проверка, не выходит ли индекс за допустимые пределы, происходит только во время исполнения программы, т.е. компилятор не пытается выявить эту ошибку даже в таких явных случаях, как:

```
int i[]=new int[5];
i[-2]=0;   // ошибка! индекс не может быть отрицательным
```

Особенности работы с массивами

Хотя при создании массива указывается его длина, она не входит в определение типа массива, т.е. одна переменная может ссылаться на массивы разной длины:

```
int i[]=new int[5];
i=new int[7];   // переменная та же, длина массива другая
```

Однако для объекта массива длина обязательно должна указываться при создании и уже никак не может быть изменена. Поскольку для экземпляра массива длина является постоянной характеристикой, для всех массивов существует поле length. Например:

```
Point p[]=new Point[5];
for (int i=0; i<p.length; i++) { p[i]=new Point(i, i);}
```

Значение индекса массива всегда имеет тип int. При обращении к элементу можно также использовать byte, short или char, поскольку эти типы автоматически расширяются до int. Попытка задействовать long приведет к ошибке компиляции.

Продолжая рассматривать тип массива, подчеркнем, что в качестве базового типа может использоваться любой тип Java, в том числе:

- интерфейсы. В таком случае элементы массива могут иметь значение null или ссылаться на объекты любого класса, реализующего этот интерфейс;
- абстрактные классы. В этом случае элементы массива могут иметь значение null или ссылаться на объекты любого неабстрактного класса-наследника.

Поскольку массив является объектным типом данных, его значения могут быть приведены к типу Object или, что то же самое, присвоены переменной типа Object. Например,

```
Object o = new int[4];
```

Это дает интересную возможность для массивов, основанных на типе Object, хранить в качестве элемента ссылку на самого себя:

```
Object arr[] = new Object[3];
arr[0]=new Object();
arr[1]=null;
arr[2]=arr;   // Элемент ссылается на весь массив!
```

Инициализация массивов

Если создать массив на основе примитивного числового типа, то изначально после создания все элементы массива имеют значение 0 (false для boolean). При создании массива на основе ссылочного типа не создается ни один объект класса Point, создается лишь один объект массива. Каждый элемент массива будет иметь пустое значение null:

```
Point p[]=new Point[5];
for (int i=0; i<p.length; i++) {System.out.println(p[i]);} // результат - слова null.
```

Далее нужно инициализировать элементы массива по отдельности, например, в цикле. Кроме того, существует и другой способ создания массивов – инициализаторы. В этом случае ключевое слово new не используется, а ставятся фигурные скобки, и в них через запятую перечисляются значения всех элементов массива:

```
int i[]={1, 3, 5};
int j[]={};   // эквивалентно new int[0]
```

Аналогично можно порождать массивы на основе объектных типов:

```
Point p=new Point(1,3);
Point arr[]={p, new Point(2,2), null, p};
```

Однако инициализатор нельзя использовать для анонимного создания экземпляров массива, то есть не для инициализации переменной, а, например, для передачи параметров метода или конструктора:

```
public class Parent {
  private String[] values;
  protected Parent (String[] s) {      values=s;   }}
public class Child extends Parent {
  public Child (String firstName, String lastName) {
    super(new String[]{firstName, lastName}); }}        // требуется анонимное создание массива
```

В конструкторе класса Child необходимо осуществить обращение к конструктору родителя и передать в качестве параметра ссылку на массив.

Многомерные массивы

Переменная i ссылается на двумерный массив, который можно представить себе в виде таблицы 3x5. Суммарно в таком массиве содержится 15 элементов, к которым можно обращаться через комбинацию индексов от (0, 0) до (2, 4):

```
int i[][]=new int[3][5];
```

Пример заполнения двумерного массива через цикл:

```
int pithagor_table[][]=new int[5][5];
for (int i=0; i<5; i++) {
   for (int j=0; j<5; j++) {
      pithagor_table[i][j]=i*j;
      System.out.print(pithagor_table[i][j] +        "\t");
   }
   System.out.println();}
```

Результатом выполнения программы будет:

```
0    0    0    0    0
0    1    2    3    4
0    2    4    6    8
0    3    6    9    12
0    4    8    12  16
```

Многомерные массивы

В Java нет двумерных, и вообще многомерных массивов, а есть массивы, базовыми типами которых являются также массивы. Например, тип int[] означает "массив чисел", а int[][] означает "массив массивов чисел".

Если создать двумерный массив и определить переменную x, которая на него ссылается, то, используя x и два числа в паре квадратных скобок каждое (например, x[0][0]), можно обратиться к любому элементу двумерного массива. Но в то же время, используя x и одно число в паре квадратных скобок, можно обратиться к одномерному массиву, который является элементом двумерного массива. Его можно проинициализировать новым массивом с некоторой другой длиной и таблица перестанет быть прямоугольной – она примет произвольную форму. В частности, можно одному из одномерных массивов присвоить даже значение null.

```
int x[][]=new int[3][5];      // прямоугольная таблица
x[0]=new int[7];
x[1]=new int[0];
x[2]=null;
```

Выражением new int[3][5] создается один массив массивов (один объект) и три массива чисел, каждый длиной 5 (три объекта). Итого, четыре объекта. Для рассмотренного случая полезно использовать упрощенную форму выражения создания массивов:

```
int x[][]=new int[3][];
```

И в этом, и в предыдущем варианте выражение x.length возвращает значение 3 – длину массива массивов. Далее можно с помощью выражений x[i].length узнать длину каждого вложенного массива чисел.

Для создания многомерных массивов можно использовать инициализаторы:

```
int i[][] = {{1,2}, null, {3}, {}};
```

Преобразование типов для массивов

Переходы между массивами и примитивными типами являются запрещенными. Преобразование массива к другим объектным типам возможны только для класса Object и интерфейсов Cloneable и Serializable. Обратный же переход является сужением и должен производиться явным образом по усмотрению разработчика. Таким образом, интерес представляют только переходы между разными типами массивов. Очевидно, что массив, основанный на примитивном типе, принципиально нельзя преобразовать к типу массива, основанному на ссылочном типе, и наоборот. Преобразования между типами массивов, основанных на различных примитивных типах, невозможны ни при каких условиях.

Для ссылочных же типов такого строгого правила нет. Например, если создать экземпляр массива, основанного на типе Child, то ссылку на него можно привести к типу массива, основанного на типе Parent.

```
Child c[] = new Child[3];
Parent p[] = c;
```

Существует универсальное правило: массив, основанный на типе A, можно привести к массиву, основанному на типе B, если сам тип A приводится к типу B.

```
// если допустимо такое приведение:
B b = (B) new A();
// то допустимо и приведение массивов:
B b[]=(B[]) new A[3];
```

Применяя это правило рекурсивно, можно преобразовывать многомерные массивы. Как обычно, расширения можно проводить неявно (как в предыдущем примере), а сужения – только явным приведением.

Ошибка ArrayStoreException

Преобразование между типами массивов, основанных на ссылочных типах, может стать причиной ошибки:

```
Child c[] = new Child[5];
Parent p[]=c;
p[0]=new Parent();
```

При выполнении такой программы возникнет ошибка. Нельзя забывать, что преобразование не меняет объект, изменяется лишь способ доступа к нему. В свою очередь, объект всегда "помнит", от какого типа он был порожден. В третьей строке делается попытка добавить в массив Child значение типа Parent, что некорректно.

Переменная «c» продолжает ссылаться на этот массив, а значит, следующей строкой может быть такое обращение: c[0].onlyChildMethod(); где метод onlyChildMethod() определен только в классе Child. Данное обращение совершенно корректно, а значит, недопустима ситуация, когда элемент c[0] ссылается на объект, несовместимый с Child.

Таким образом, несмотря на отсутствие ошибок компиляции, виртуальная машина при выполнении программы всегда осуществляет дополнительную проверку перед присвоением значения элементу массива. Необходимо удостовериться, что реальный массив, существующий на момент исполнения, действительно может хранить присваиваемое значение. Если это условие нарушается, то возникает ошибка, которая называется ArrayStoreException.

Переменные типа массив и их значения

Тип переменной	Допустимые типы ее значения
Массив простых чисел	• null • в точности совпадающий с типом переменной
Массив ссылочных значений	• null • совпадающий с типом переменной • массивы ссылочных значений, удовлетворяющих следующему условию: если тип переменной – массив на основе типа A, то значение типа массив на основе типа B допустимо тогда и только тогда, когда B приводимо к A
Object	• null • любой ссылочный, включая массивы

Клонирование

Механизм клонирования позволяет порождать новые объекты на основе существующего, которые обладали бы точно таким же состоянием, что и исходный. То есть ожидается, что для исходного объекта, представленного ссылкой x, и результата клонирования, возвращаемого методом x.clone(), выражения истинны:

- x != x.clone()
- x.clone().getClass() == x.getClass()
- x.equals(x.clone())

Реализация такого метода clone() осложняется целым рядом потенциальных проблем, например:

- класс, от которого порожден объект, может иметь разнообразные конструкторы, которые к тому же могут быть недоступны (например, модификатор доступа private);
- цепочка наследования, которой принадлежит исходный класс, может быть довольно длинной, и каждый родительский класс может иметь свои поля – недоступные, но важные для воссоздания состояния исходного объекта;
- в зависимости от логики реализации возможна ситуация, когда не все поля должны копироваться для корректного клонирования; одни могут оказаться лишними, другие потребуют дополнительных вычислений или преобразований;
- возможна ситуация, когда объект нельзя клонировать, дабы не нарушить целостность системы.

Клонирование

Класс Object содержит метод clone(). Рассмотрим его объявление:

```
protected native Object clone()  throws CloneNotSupportedException;
```

Он и используется для клонирования. Далее возможны два варианта.

- разработчик может в своем классе переопределить этот метод и реализовать его по своему усмотрению, решая перечисленные проблемы так, как того требует логика разрабатываемой системы. Упомянутые условия, которые должны быть истинными для клонированного объекта, не являются обязательными и программист может им не следовать, если это требуется для его класса.
- предполагается использование реализации метода clone() в самом классе Object. То, что он объявлен как native, говорит о том, что его реализация предоставляется виртуальной машиной. Естественно, перечисленные трудности легко могут быть преодолены самой JVM, ведь она хранит в памяти все свойства объектов.

  При выполнении метода clone() сначала проверяется, можно ли клонировать исходный объект. Если разработчик хочет сделать объекты своего класса доступными для клонирования через Object.clone(), он должен реализовать в своем классе интерфейс Cloneable. В этом интерфейсе нет элементов, он служит признаком для виртуальной машины, что объекты могут быть клонированы. Если проверка не выполняется успешно, метод порождает ошибку CloneNotSupportedException.

  Если интерфейс Cloneable реализован, то порождается новый объект от того же класса, от которого был создан исходный объект. Копирование выполняется на уровне виртуальной машины, никакие конструкторы не вызываются. Затем значения всех полей, объявленных, унаследованных либо объявленных в родительских классах, копируются. Полученный объект возвращается в качестве клона.

Клонирование

Класс Object не реализует интерфейс Cloneable, а потому попытка вызова new Object().clone() будет приводить к ошибке. Метод clone() предназначен для использования в наследниках, которые могут обращаться к нему с помощью выражения super.clone(). При этом могут быть сделаны следующие изменения:

- модификатор доступа расширен до public ;
- удалено предупреждение об ошибке CloneNotSupportedException ;
- результирующий объект может быть модифицирован любым способом, на усмотрение разработчика.

Все массивы реализуют интерфейс Cloneable и доступны для клонирования. Важно помнить, что все поля-ссылки клонированного объекта приравниваются, их значения никогда не клонируются.

Пример клонирования

```
public class Test implements Cloneable {
Point p;
int height;
public Test(int x, int y, int z) {
    p=new Point(x, y);
    height=z;
    }
public static void main(String s[]) {
    Test t1=new Test(1, 2, 3), t2=null;
    try {
      t2=(Test) t1.clone();
    } catch (CloneNotSupportedException e) {}
    t1.p.x=-1;
    t1.height=-1;
    System.out.println("t2.p.x=" + t2.p.x + ", t2.height=" + t2.height);
}} // Результат работы программы: t2.p.x=-1, t2.height=3
```

Примитивное поле было скопировано и далее существует независимо в исходном и клонированном объектах, а ссылочное поле было скопировано по ссылке, оба объекта ссылаются на один и тот же экземпляр класса Point. Этого можно избежать, если переопределить метод clone() в классе Test:

```
public Object clone() {
   Test clone=null;
   try {
      clone=(Test) super.clone();
   } catch (CloneNotSupportedException e) {
          throw new InternalError(e.getMessage());
          }
   clone.p=(Point)this.p.clone();
   return clone;
} // Результат работы программы: t2.p.x=1, t2.height=3
```

Клонирование массивов

Рассмотрим пример:

```
int a[]={1, 2, 3};
int b[]=(int[])a.clone();
a[0]=0;
System.out.println(b[0]);
```

Результатом будет единица, что вполне очевидно, так как весь массив представлен одним объектом, который не будет зависеть от своей копии. Усложняем пример:

```
int a[][]={{1, 2}, {3}};
int b[][]=(int[][]) a.clone();
if (...) {
   // первый вариант:
   a[0]=new int[]{0};
   System.out.println(b[0][0]);
}
else {
   // второй вариант:
   a[0][0]=0;
   System.out.println(b[0][0]);
}
```