Лекция 7. Массивы

Оглавление

Массивы как тип данных в Java

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.

Работа с любым массивом включает обычные операции, уже описанные для других типов, - объявление, инициализация и т.д. Начнем последовательно изучать их в приложении к массивам.

Объявление массивов

В качестве примера рассмотрим объявление переменной типа "массив, основанный на примитивном типе int ":

```
int a[];
```

Как мы видим, сначала указывается базовый тип. Затем идет имя переменной, а пара квадратных скобок указывает на то, что используемый тип является именно массивом. Также допустима запись:

```
int[] a;
```

Количество пар квадратных скобок указывает на размерность массива. Для многомерных массивов допускается смешанная запись:

```
int[] a[];
```

Переменная a имеет тип "двумерный массив, основанный на int ". Аналогично объявляются массивы с базовым объектным типом:

```
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];
```

Переменная инициализируется ссылкой, указывающей на только что созданный массив. После его создания можно обращаться к элементам, используя ссылку на массив, далее в квадратных скобках указывается индекс элемента. Индекс меняется от нуля, пробегая всю длину массива, до максимально допустимого значения, на единицу меньшего длины массива.

```
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=0
1*1=1
```

```
2*2=4
3*3=9
4*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
```

Ошибка возникнет только на этапе выполнения программы.

Хотя при создании массива необходимо указывать его длину, это значение не входит в определение типа массива, важна лишь размерность. Таким образом, 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. В последнем примере для присвоения переменной ссылки на массив большей длины потребовалось создать новый экземпляр.

Поскольку для экземпляра массива длина является постоянной характеристикой, для всех массивов существует специальное поле length, позволяющее узнать ее значение. 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.

Соответственно, и поле length имеет тип int, а теоретическая максимально возможная длина массива равняется 2^{31}-1, то есть немногим больше 2 млрд.

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

Теперь, когда мы выяснили, как создавать экземпляры массива, рассмотрим, какие значения принимают его элементы.

Если создать массив на основе примитивного числового типа, то изначально после создания все элементы массива имеют значение по умолчанию, то есть 0. Если массив объявлен на основе примитивного типа boolean, то и в этом случае все элементы будут иметь значение по умолчанию false. Выше рассматривался пример инициализации элементов с помощью цикла for.

Рассмотрим создание массива на основе ссылочного типа. Предположим, это будет класс Point. При создании экземпляра массива с применением ключевого слова new не создается ни один объект класса Point, создается лишь один объект массива. Каждый элемент массива будет иметь пустое значение null. В этом можно убедиться на простом примере:

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

Результатом будут лишь слова null.

Далее нужно инициализировать элементы массива по отдельности, например, в цикле. Вообще, создание массива длиной n можно рассматривать как

заведение n переменных и работать с элементами массива (в последнем примере p[i]) по правилам обычных переменных.

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. Например, для числового массива явная инициализация записывается следующим образом:

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

Длина массива вычисляется автоматически, исходя из количества введенных значений. Далее создается массив такой длины и каждому его элементу присваивается указанное значение.

Аналогично можно порождать массивы на основе объектных типов, например:

```
Point p=new Point(1,3);
Point arr[]={p, new Point(2,2), null, p};
// или
String sarr[]={"aaa", "bbb", "cde"+"xyz"};
```

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

Например:

```
public class Parent {
   private String[] values;

   protected Parent(String[] s) {
      values=s;
   }
}

public class Child extends Parent {

   public Child(String firstName,
                String lastName) {
      super(???);
      // требуется анонимное создание массива
   }
}
```

В конструкторе класса Child необходимо осуществить обращение к конструктору родителя и передать в качестве параметра ссылку на массив. Теоретически можно передать null, но это приведет в большинстве случаев к некорректной работе классов. Можно вставить выражение new String[2], но

тогда вместо значений firstName и lastName будут переданы пустые строки. Попытка записать {firstName, lastName} приведет к ошибке компиляции, так можно только инициализировать переменные.

Корректное выражение выглядит так:

```
new String[]{firstName, lastName}
```

Что является некоторой смесью выражения, создающего массивы с помощью new, и инициализатора. Длина массива определяется количеством указанных значений.

Многомерные массивы

Теперь перейдем к рассмотрению многомерных массивов. Так, в следующем примере

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

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). Пример заполнения двумерного массива через цикл:

```
int pithagor_table[][]=new int[10][10];

for (int i=1; i<10; i++) {

   for (int j=1; j<10; 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
```

Однако такой взгляд на двумерные и многомерные массивы является неполным. Более точный подход заключается в том, что 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". Поясним такую точку зрения.

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;
```

После таких операций массив, на который ссылается переменная x, назвать прямоугольным никак нельзя. Зато хорошо видно, что это просто набор одномерных массивов или значений null.

Полезно подсчитать, сколько объектов порождается выражением new int[3][5]. Правильный подсчет таков: создается один массив массивов (один объект) и три массива чисел, каждый длиной 5 (три объекта). Итого, четыре объекта.

В рассмотренном примере три из них (массивы чисел) были тут же переопределены новыми значениями. Для таких случаев полезно использовать упрощенную форму выражения создания массивов:

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

Такая запись порождает один объект – массив массивов – и заполняет его значениями null. Теперь понятно, что 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, при условии, что i неотрицательно и меньше x.length, а также x[i] не равно null. Иначе будут возникать ошибки во время выполнения программы.

Вообще, при создании многомерных массивов с помощью new необходимо указывать все пары квадратных скобок, соответственно количеству измерений. Но заполненной обязательно должна быть лишь крайняя левая

пара, это значение задаст длину верхнего массива массивов. Если заполнить следующую пару, то этот массив заполнится не значениями по умолчанию null, а новыми созданными массивами с меньшей на единицу размерностью. Если заполнена вторая пара скобок, то можно заполнить третью, и так далее.

Аналогично, для создания многомерных массивов можно использовать инициализаторы. В этом случае применяется столько вложенных фигурных скобок, сколько требуется:

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

В этом примере порождается четыре объекта. Это, во-первых, массив массивов длиной 4, а во-вторых, три массива чисел с длинами 2, 1, 0, соответственно.

Все рассмотренные примеры и утверждения одинаково верны для многомерных массивов, основанных как на примитивных, так и на ссылочных типах.

Класс массива

Поскольку массив является объектным типом данных, можно попытаться представить себе, как выглядело бы объявление класса такого типа. На самом деле эти объявления не хранятся в файлах, или еще каком-нибудь формате. Учитывая, что массив может быть объявлен на основе любого типа и иметь произвольную размерность, это физически невыполнимо, да и не требуется. Вместо этого во время выполнения приложения виртуальная машина генерирует эти объявления динамически на основе базового типа и размерности, а затем они хранятся в памяти в виде таких же экземпляров класса Class, как и для любых других типов.

Рассмотрим гипотетическое объявление класса для массива, основанного на неком объектном типе Element.

Объявление класса начинается с перечисления модификаторов, среди которых особую роль играют модификаторы доступа. Класс массива будет иметь такой же уровень доступа, как и базовый тип. То есть если Element объявлен как public -класс, то и массив будет иметь уровень доступа public. Для любого примитивного типа класс массива будет public. Можно также указать модификатор final, поскольку никакой класс не может наследоваться от класса массива.

Затем следует имя класса, на котором можно подробно не останавливаться, т.к. к типу массива обращение идет не по его имени, а по имени базового типа и набору квадратных скобок.

Затем нужно указать родительский класс. Все массивы наследуются напрямую от класса Object. Далее перечисляются интерфейсы, которые реализует класс. Для массива это будут интерфейсы Cloneable и Serializable. Первый из них подробно рассматривается в конце этой лекции, а второй будет описан в следующих лекциях.

Тело класса содержит объявление одного public final поля length типа int. Кроме того, переопределен метод clone() для поддержки интерфейса Cloneable.

Сведем все вышесказанное в формальную запись класса:

```
[public] class A implements Cloneable,
                java.io.Serializable {
  public final int length;
    // инициализируется при создании
  public Object clone() {
    try { return super.clone();}
    catch (CloneNotSupportedException e) {
      throw new InternalError(e.getMessage());
   }
 }
}
```

Таким образом, экземпляр типа массив является полноценным объектом, который, в частности, наследует все методы, определенные в классе Object, например, toString(), hashCode() и остальные.

Например:

```
// результат работы метода toString()
System.out.println(new int[3]);
System.out.println(new int[3][5]);
System.out.println(new String[2]);

// результат работы метода hashCode()
System.out.println(new float[2].hashCode());
```

Результатом выполнения программы будет:

```
[I@26b249
[[I@82f0db
[Ljava.lang.String;@92d342
7051261
```

Преобразование типов для массивов

Теперь, когда массив введен как полноценный тип данных в Java, рассмотрим, какое влияние он окажет на преобразование типов.

Ранее подробно рассматривались переходы между примитивными и обычными (не являющимися массивами) ссылочными типами. Хотя массивы являются объектными типами, их также будет полезно разделить по базовому типу на две группы – основанные на примитивном или ссылочном типе.

Имейте в виду, что переходы между массивами и примитивными типами являются запрещенными. Преобразования между массивами и другими объектными типами возможны только для класса Object и интерфейсов Cloneable и Serializable. Массив всегда можно привести к этим трем типам, 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.

Пока не будем останавливаться на этом подробно, однако заметим, что 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;
```

Вообще, 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. Например, массив Child[][] можно привести к Parent[][], так как их базовые типы приводимы (Child[] к Parent[]) также на основе этого правила (поскольку базовые типы Child и Parent приводимы в силу правил наследования).

Как обычно, расширения можно проводить неявно (как в предыдущем примере), а сужения – только явным приведением.

Вернемся к массивам, основанным на примитивном типе. Невозможность их участия в преобразованиях типов связана, конечно, с различиями между

простыми и ссылочными типами данных. Поскольку элементами объектных массивов являются ссылки, они легко могут участвовать в приведении. Напротив, элементы простых типов действительно хранят числовые или булевские значения. Предположим, такое преобразование осуществимо:

```
// пример вызовет ошибку компиляции
byte b[]={1, 2, 3};
int i[]=b;
```

В таком случае, элементы b[0] и i[0] хранили бы значения разных типов. Стало быть, преобразование потребовало бы копирования с одновременным преобразованием типа всех элементов исходного массива. В результате был бы создан новый массив, элементы которого равнялись бы по значению элементам исходного массива.

Но преобразование типа не может порождать новые объекты. Такие операции должны выполняться только явным образом с применением ключевого слова new. По этой причине преобразования типов массивов, основанных на примитивных типах, запрещены.

Если же копирование элементов действительно требуется, то нужно сначала создать новый массив, а затем воспользоваться стандартной функцией System.arraycopy(), которая эффективно выполняет копирование элементов одного массива в другой.

Ошибка ArrayStoreException

Преобразование между типами массивов, основанных на ссылочных типах, может стать причиной одной довольно неочевидной ошибки.

Рассмотрим пример:

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

С точки зрения компилятора код совершенно корректен. Преобразование во второй строке допустимо. В третьей строке элементу массива типа Parent присваивается значение того же типа.

Однако 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. С учетом этих замечаний становится ясно, что в третьей строке делается попытка добавить в массив Child значение типа Parent, что некорректно.

Действительно, ведь 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.

Может сложиться впечатление, что разобранная ситуация является надуманной,– зачем преобразовывать массив и тут же задавать для него неверное значение? Однако преобразование при присвоении значений является лишь примером. Рассмотрим объявление метода:

```
public void process(Parent[] p) {
   if (p!=null && p.length>0) {
      p[0]=new Parent();
   }
}
```

Метод выглядит абсолютно корректным, все потенциально ошибочные ситуации проверяются if -выражением. Однако следующий вызов этого метода все равно приводит к ошибке:

```
process(new Child[3]);
```

И это будет как раз ошибка ArrayStoreException.

Переменные типа массив и их значения

Завершим описание взаимосвязи типа переменной и типа значений, которые она может хранить.

Как обычно, массивы, основанные на простых и ссылочных типах, мы описываем раздельно.

Переменная типа массив примитивных величин может хранить значения только точно такого же типа, либо null.

Переменная типа "массив ссылочных величин" может хранить следующие значения:

1. null ;
2. значения точно такого же типа, что и тип переменной;
3. все значения типа массив, основанный на типе, приводимом к базовому типу исходного массива.

Все эти утверждения непосредственно следуют из рассмотренных выше особенностей приведения типов массивов.

Еще раз напомним про исключительный класс Object. Переменные такого типа могут ссылаться на любые объекты, порожденные как от классов, так и от массивов.

Сведем все эти утверждения в таблицу.

Таблица Табл. 7.1. Тип переменной и тип ее 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, как следует из названия, 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(), выражение

```
x != x.clone()
```

должно быть истинным, как и выражение

```
x.clone().getClass() == x.getClass()
```

Наконец, выражение

```
x.equals(x.clone())
```

также верно. 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.

Поэтому было реализовано следующее решение.

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

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

Именно он используется для клонирования. Далее возможны два варианта.

Первый вариант: 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.

Второй вариант предполагает 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(), то он должен реализовать в своем классе интерфейс Cloneable. В этом интерфейсе нет ни одного элемента, он служит лишь 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. При этом 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.

Обратите внимание, что сам класс Object не реализует интерфейс Cloneable, а потому попытка вызова new Object().clone() будет приводить к ошибке. Метод clone() предназначен скорее 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.

Напомним, что все массивы реализуют интерфейс Cloneable и, таким образом, доступны для клонирования.

Важно помнить, что все поля клонированного объекта приравниваются, их значения никогда не клонируются. Рассмотрим пример:

```
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);
```

```
  }
}
```

Результатом работы программы будет:

```
t2.p.x=-1, t2.height=3
```

Из примера видно, что примитивное поле было скопировано и далее существует независимо в исходном и клонированном объектах. Изменение одного не сказывается на другом.

А вот ссылочное поле было скопировано по ссылке, оба объекта ссылаются на один и тот же экземпляр класса Point. Поэтому изменения, происходящие с исходным объектом, сказываются на клонированном.

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;
}
```

Обратите внимание, что результат метода Object.clone() приходится явно приводить к типу Test, хотя его реализация гарантирует, что клонированный объект будет порожден именно от этого класса. Однако тип возвращаемого значения в данном методе для универсальности объявлен как Object, поэтому явное сужение необходимо.

Теперь метод main можно упростить:

```
public static void main(String s[]) {
   Test t1=new Test(1, 2, 3);
   Test t2=(Test) t1.clone();
   t1.p.x=-1;
   t1.height=-1;
   System.out.println("t2.p.x=" + t2.p.x +
     ", t2.height=" + t2.height);
}
```

Результатом будет:

```
t2.p.x=1, t2.height=3
```

То есть теперь все поля исходного и клонированного объектов стали независимыми.

Реализация такого "неглубокого" клонирования в методе Object.clone() необходима, так как в противном случае клонирование второстепенного объекта могло бы привести к огромным затратам ресурсов, ведь этот объект может содержать ссылки на более значимые объекты, а те при клонировании также начали бы копировать свои поля, и так далее. Кроме того, типом поля клонируемого объекта может быть класс, не реализующий Cloneable, что приводило бы к дополнительным проблемам. Как показано в примере, при необходимости дополнительное копирование можно добавить самостоятельно.

Клонирование массивов

Итак, любой массив может быть клонирован. В этом разделе хотелось бы рассмотреть особенности, возникающие из-за того, что Object.clone() копирует только один объект.

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]);
}
```

Разберем, что будет происходить в этих двух случаях. Начнем с того, что в первой строке создается двумерный массив, состоящий из двух одномерных. Итого три объекта. Затем, на следующей строке при клонировании будет создан новый двумерный массив, содержащий ссылки на те же самые одномерные массивы.

Теперь несложно предсказать результат обоих вариантов. В первом случае в исходном массиве меняется ссылка, хранящаяся в первом элементе, что не

принесет никаких изменений для клонированного объекта. На консоли появится 1.

Во втором случае модифицируется существующий массив, что скажется на обоих двухмерных массивах. На консоли появится 0.

Обратите внимание, что если из примера убрать условие if-else, так, чтобы отрабатывал первый вариант, а затем второй, то результатом будет опять 1, поскольку в части второго варианта модифицироваться будет уже новый массив, порожденный в части первого варианта.

Таким образом, в Java предоставляется мощный, эффективный и гибкий механизм клонирования, который легко применять и модифицировать под конкретные нужды. Особенное внимание должно уделяться копированию объектных полей, которые по умолчанию копируются только по ссылке.

Наконец, изучается механизм клонирования, существующий с самых первых версий Java и позволяющий создавать точные копии объектов, если их классы позволяют это делать, реализуя интерфейс Cloneable. Поскольку стандартное клонирование порождает только один новый объект, это приводит к особым эффектам при работе с объектными полями классов и массивами.