4.Понятие класса

Класс определяет новый тип данных, используя который можно определять переменные этого типа, которые называются объектами или экземплярами класса. Объявление класса имеет вид:

```
class classname {
   type instance-variable1;
   type instance-variable2;
//…
   type instance-variableN;
   type methodnamel(parameter-list){
      // тело метода
   }
   type methodname2 (parameter-list){
      // тело метода
   }
//...
   type methodnameN(parameter-list){
      // тело метода
   }
}
```

Данные или переменные, определенные в классе, называются *переменными экземпляра* или *экземплярными переменными* (instance variables). Код содержится внутри *методов* (methods). Методы и переменные, определенные внутри класса, называются *членами класса* (class members).

Переменные, определенные в классе, называются *переменными экземпляра* потому, что каждый экземпляр класса (т. е. каждый объект класса) содержит свою собственную копию этих переменных. Таким образом, данные одного объекта отделены от данных другого.

Все методы имеют ту же общую форму, что метод `main()`, который мы использовали до сих пор. Однако большинство методов не будут определяться как `static` или `public`. Метод `main` определяется только в классе, который является стартовой точкой программы.

В отличие от языка C++ объявление класса и реализация методов хранятся в одном месте и не определяются отдельно то есть любой класс должен быть полностью определен в одном исходном файле.

Программа 7. Класс Box

```
/* Программа, которая использует Box-класс.
   Назовите этот файл DemoBox.java
*/
class Box{
   double width;
   double height;
   double depth;
}
// Этот класс объявляет объект типа Box.
class DemoBox {
   public static void main(String args[]){
      Box mybox = new Box();
      double vol;
   // Присвоить значения экземплярным переменным объекта mybox
      mybox.width = 10;
      mybox.height = 20;
      mybox.depth = 15;
   // Вычислить объем блока
      vol = mybox.width * mybox.height * mybox.depth;
      System.out.println("Объем равен " + vol);
   }
```

```
}
```

файл, который содержит эту программу следует назвать DemoBox.java, потому что метод main() находится в классе с именем DemoBox, а не в классе с именем Box. В результате компиляции создаются два файла с расширением .class — один для Box-класса и один для класса DemoBox. Java-компилятор автоматически помещает каждый класс в его собственный class-файл. Можно было поместить каждый класс в свой собственный файл с именами Box.java и DemoBox.java, соответственно.

Программа выводит:

```
Объем равен 3000.0
```

4.1. Объявление объектов

Класс - это новый тип данных, который можно использовать для объявления соответствующих объектов. Создание объектов класса - это двухшаговый процесс. Во-первых, нужно объявить переменную типа "класс". Она не определяет объект, но может ссылаться на него. Во-вторых, нужно создать объекта в памяти и назначать его этой переменной. Это делается с помощью оператора new, который распределяет динамически (т. е. во время выполнения) память для объекта и возвращает ссылку на нее. Данная ссылка является адресом ячейки памяти, выделенной объекту. Затем эта ссылка сохраняется в переменной. Таким образом, в Java все объекты класса должны быть распределены динамически.

В программе для объявления объекта типа Box использовалась строка вида:

```
Box mybox = new Box();
```

Эта инструкция объединяет оба шага создания объекта. Можно выполнять шаги отдельно:

```
Box mybox;            // Объявление ссылки на объект
mybox = new Box();    // Выделение памяти для Box-объекта
```

Первая строка объявляет mybox как ссылку на объект типа Box. После того как эта строка выполняется, mybox содержит значение null, которое означает, что переменная еще не указывает на существующий объект. Любая попытка использовать mybox в этой точке приведет к ошибке во время компиляции. Следующая строка создает объект и назначает ссылку на него переменной mybox. После того, как вторая строка выполнится, можно использовать mybox, как если бы это был объект Box. Но в действительности mybox просто содержит адрес ячейки памяти объекта Box. Действие этих двух строк программы изображено на рис. 1.

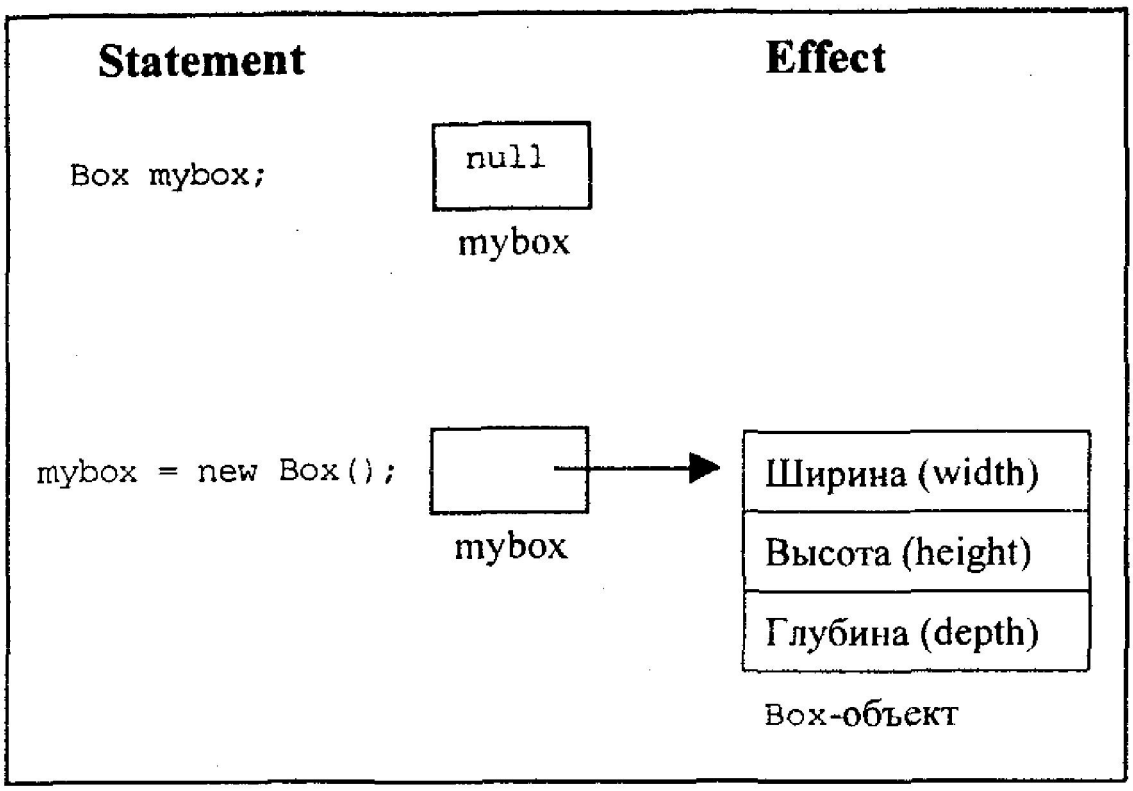

Рис. 1. Создание объекта типа Box

Невозможно направить объектную ссылку на произвольное место в памяти или манипулировать ею как целым числом.

4.2. Оператор new

Оператор new динамически распределяет память для объекта. Он имеет следующую общую форму:

```
class-var = new classname();
```

Здесь class-var — переменная типа "класс", которая создается; classname – имя класса, экземпляр которого создается. За именем класса следуют круглые скобки, указывающие на вызов *конструктора* класса. Конструктор (constructor) – это метод класса, вызываемый автоматически при создании объекта класса. Как правило, в классе определяются свои собственные конструкторы. Если явный конструктор не определен, то Java автоматически обеспечит так называемый "конструктор по умолчанию" (default constructor). Именно так обстоит дело с классом Box.

Простые типы Java (числа, символы) реализованы не как классы, а как "нормальные" переменные в целях эффективности.

4.3. Присваивание ссылочных переменных

Рассмотрим фрагмент

```
Box b1 = new Box();
Box b2 = b1;
```

Переменная b2 будет ссылаться на тот же объект, что и b1, копия объекта для b2 не создается.

Инструкция

```
b1 = null;
```

отключит b1 от исходного объекта без воздействия на объект или переменную b2.

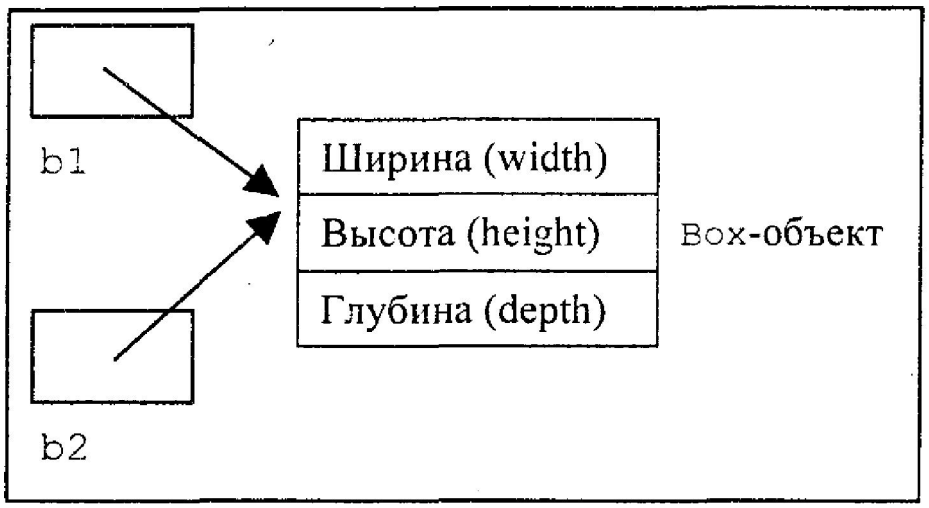

Рис. 2. Две ссылки на один и тот же объект

4.4. Добавление метода к классу Box

Методы определяют интерфейс с классом. Они позволяют разработчику класса скрывать специфическое размещение внутренних структур данных за более ясными абстракциями метода.

Добавим в класс Box метод, вычисляющий объем.

Программа 8. Включение метода в класс Box

```
// В этой программе класс Box содержит метод, выводящий объем.
class Box {
   double width;
   double height;
   double depth;
// Показать объем блока
   void volume()   {
      System.out.print ("Объем равен ") ;
      System.out.println(width * height * depth);
   }
}
class BoxDemo2 {
   public static void main(String args[])   {
      Box mybox1 = new Box();
      Box mybox2 = new Box();
   // Присвоить значения переменным экземпляра mybox1
      mybox1.width = 10;
      mybox1.height = 20;
      mybox1.depth = 15;
   /* Присвоить другие значения
      переменным экземпляра mybox2 */
      mybox2.width = 3;
      mybox2.height = 6;
      mybox2.depth = 9;
   // Показать объем первого блока
      mybox1.volume();
   // Показать объем второго блока
      mybox2.volume ();
   }
}
```

Эта программа выводит:

```
Объем равен 3000.0
Объем равен 162.0
```

Для вызова метода класса после имени объекта ставится точка и указывается имя метода:

```
mybox1.volume();
mybox2.volume ();
```

Метод — это способ реализации подпрограмм в языке Java.

Внутри метода `volume()` переменные `width`, `height` и `depth` указаны без предшествующих им имен объектов.

4.5. Возврат значений из метода

Методы класса могут возвращать в качестве результата своей работы некоторые значения, указываемые в инструкции `return`. Включим в класс `Box` метод, который будет вычислять объем и возвращать его значение.

Программа 9. Метод, возвращающий значение

```
class Box {
   double width;
   double height;
   double depth;
// Вычислить и возвратить объем
   double volume(){
      return width * height * depth;
   }
}
class DemoBox3 {
   public static void main(String args[]){
      Box mybox1 = new Box();
      Box mybox2 = new Box();
      double vol;
// Присвоить значения переменным экземпляра mybox1
   mybox1.width  = 10;
   mybox1.height = 20;
   mybox1.depth  = 15;
   /* Присвоить другие значения
      переменным экземпляра mybox2 */
      mybox2.width  = 3;
      mybox2.height = 6;
      mybox2.depth  = 9;
   // Получить объем первого блока
      vol = mybox1.volume();
      System.out.println("Объем равен " + vol);
   // Получить объем второго блока
      vol = mybox2.volume();
      System.out.println("Объем равен " + vol);
   }
}
```

4.6. Методы с параметрами

Добавим в класс `Box` метод, который будет устанавливать размеры блока, которые будем передавать через параметры метода.

Программа 10. Метод с параметрами

```
// Эта программа использует параметризованный метод.
class Box {
   double width;
   double height;
   double depth;
```

```
//Вычислить и возвратить объем
   double volume(){
      return width * height * depth;
   }
//Установить размеры блока
   void setDim(double w, double h, double d){
      width  = w;
      height = h;
      depth  = d;
   }
}
class DemoBox4{
   public static void main(String args[]){
      Box mybox1 = new Box();
      Box mybox2 = new Box();
      double vol;
   // Получить размеры первого блока
      System.out.println("Размеры первого блока");
      System.out.println(mybox1.width + ", " + mybox1.height + ", " +
                         mybox1.depth);
      System.out.println("Размеры второго блока");
      System.out.println(mybox2.width + ", " + mybox2.height + ", " +
                         mybox2.depth);
   //Установить размеры каждого блока
      mybox1.setDim(10, 20,  15);
      mybox2.setDim(3,  6,  9);
   //Получить объем первого блока
      System.out.println("Объемы блоков после установки размеров");
      vol = mybox1.volume ();
      System.out.println("Объем равен " + vol);
   //Получить объем второго блока
      vol = mybox2.volume();
      System.out.println("Объем равен " + vol);
   }
}
```

Метод `setDim()` имеет три *параметра* `w`, `h`, `d`. При вызове метода вместо параметров подставляются *аргументы*. Например,

```
myboxl.setDim(10, 20,  15);
```

Аргумент 10 копируется в параметр `w`, 20 копируется в `h` и 15 копируется в `d`. Внутри метода `setDim()` значения `w`, `h` и `d` затем присваиваются переменным `width`, `height` и `depth`, соответственно.

В программе сначала выводятся размеры блоков, которые устанавливаются при создании объектов `mybox1` и `mybox2`. Затем размеры блоков устанавливаются методом `setDim()`, вычисляются и выводятся объемы. Программа выводит:

```
Размеры первого блока
0.0, 0.0, 0.0
Размеры второго блока
0.0, 0.0, 0.0
Объемы блоков после установки размеров
Объем равен 3000.0
Объем равен 162.0
```

Из результатов работы программы видно, что размеры блоков после их создания оператором `new` равны нулю. Эту работу выполняет упоминавшийся конструктор по умолчанию.

5.Конструкторы

Оператор new, вызываемый при создании объекта выделяет память под данные класса. В случае с классом Box – память под переменные width, height и depth. Но, чтобы эти переменные получили определенные значения, выполнялось явное присваивание или, как в предыдущей программе, вызывался метод setDimo. То есть в предыдущих программах действия по созданию объекта и по его инициализации разнесены во времени. Желательно было бы выполнять инициализацию во время первоначального создания объекта. Эта автоматическую инициализация выполняться с помощью конструктора.

Конструктор – это специальный метод, который автоматически вызывается при создании объекта. Конструктор имеет такое же имя, как класс. Конструкторы ничего не возвращают, даже void. Происходит это от того, что неявным возвращаемым типом конструктора класса является тип самого класса.

Конструкторов в классе может быть несколько. Различаются они количеством и типом параметров.

Включим в класс Box конструктор без параметров, который будет устанавливать одинаковые размеры блока и конструктор с параметрами, который заменит метод setDim().

Программа 11. Конструкторы

```
class Box {
   double width;
   double height;
   double depth;
//Вычислить и возвратить объем
   double volume(){
      return width * height * depth;
   }
// Конструктор по умолчанию устанавливает одинаковые размеры блока
   Box(){
      width  = 10;
      height = 10;
      depth  = 10;
   }
// Конструктор с параметрами устанавливает заданные размеры блока
   Box(double w, double h, double d){
      width  = w;
      height = h;
      depth  = d;
   }
}
class DemoBox5{
   public static void main(String args[]){
      Box mybox1 = new Box();           // Используется конструктор по умолчанию
      Box mybox2 = new Box(10, 20, 30);  // Используется конструктор с параметрами
      double vol;
      //Получить объем первого блока
      vol = mybox1.volume ();
      System.out.println("Объем равен " + vol);
   //Получить объем второго блока
      vol = mybox2.volume();
      System.out.println("Объем равен " + vol);
   }
}
```

Программа выводит:

```
Объем равен 1000.0
Объем равен 6000.0
```

5.1. Ключевое слово this

Иногда у метода возникает необходимость обращаться к объекту, который его вызвал. Для этого Java определяет ключевое слово this. Его можно использовать внутри любого метода, чтобы сослаться на *текущий* объект. То есть this — это всегда ссылка на объект, метод которого был вызван. Можно использовать this везде, где разрешается ссылка на объект текущего класса.

Рассмотрим следующую версию конструктора по умолчанию Box():

```
// Избыточное использование this.
Box(double w, double h, double d){
   this.width = w;
   this.height = h;
   this.depth = d;
}
```

Эта версия Box() работает точно так же, как и более ранняя. Использование this избыточно, но совершенно корректно. Внутри Box() this всегда ссылается на вызывающий объект.

Скрытие переменной экземпляра

Как известно, в Java недопустимо объявление двух локальных переменных с одним и тем же именем внутри той же самой или включающей области действия идентификаторов. Заметим, что можно иметь локальные переменные, включая формальные параметры для методов, которые перекрываются с именами экземплярных переменных класса. Однако, когда локальная переменная имеет такое же имя, как переменная экземпляра, локальная переменная *скрывает* переменную экземпляра. Вот почему width, height и depth не использовались как имена параметров конструктора Box() внутри класса Box. Если бы они были использованы для именования этих параметров, то, скажем width, как формальный параметр, скрыл бы переменную экземпляра width. Хотя обычно проще указывать различные имена, существует другой способ обойти эту ситуацию. Поскольку this позволяет обращаться прямо к объекту, это можно применять для разрешения любых конфликтов пространства имен, которые могли бы происходить между экземплярными и локальными переменными. Ниже представлена другая версия Box(), которая использует width, height и depth для имен параметров и затем применяет this, чтобы получить доступ к переменным экземпляра с теми же самыми именами:

```
// Используйте этот вариант конструктора
// для разрешения конфликтов пространства имен.
Box(double width, double height, double depth){
   this.width  = width;
   this.height = height;
   this.depth  = depth;
}
```

Использование this в указанном контексте иногда может привести к путанице, так что некоторые программисты предпочитают не применять имена локальных переменных и формальных параметров, которые совпадают с переменными экземпляра. Конечно, другие программисты, наоборот, верят, что это хорошее соглашение — использовать одинаковые имена — для ясности, и this — чтобы преодолеть скрытие переменной экземпляра. Какой подход принимаете вы — это вопрос вкуса.

Хотя в только что показанных примерах this не имеет никакого существенного значения, в некоторых ситуациях он очень полезен.

5.2. Сборка "мусора"

Так как объекты распределяются динамически с помощью операции new, можно задать вопрос, как такие объекты ликвидируются и их память освобождается для более позднего перераспределения. В некоторых языках, таких как C++, от динамически распределенных

объектов нужно освобождаться вручную — при помощи оператора delete. Java использует другой подход: он выполняет освобождение памяти от объекта автоматически. Методика, которая реализует эту процедуру, называется сборкой "мусора". Она работает примерно так: когда никаких ссылок на объект не существует, предполагается, что этот объект больше не нужен, и память, занятая объектом, может быть освобождена. Нет никакой явной потребности уничтожать объекты как в C++. Сборка "мусора" происходит не регулярно (если вообще происходит) во время выполнения программы. Она не будет происходить просто потому, что существует один или более объектов, которые уже не используются. Кроме того, различные реализации исполняющей системы Java имеют разные подходы к сборке "мусора", но вам, по большей части, не придется думать об этом при записи своих программ.

Метод finalized

Иногда объекту нужно выполнять некоторые действия, когда он разрушается. Например, если объект содержит некоторый не-Java ресурс, такой как дескриптор файла или оконный шрифт, то нужно удостовериться, что до разрушения объекта эти ресурсы освобождаются. Для обработки таких ситуаций Java использует механизм, называемый *завершением* (finalization). Применяя завершение, можно определять специальные действия, которые будут выполняться примерно тогда, когда объект будет использоваться сборщиком мусора.

Чтобы добавить завершение к классу, вы просто определяете метод finalize(). Исполняющая система Java вызывает этот метод всякий раз, когда она собирается ликвидировать объект данного класса. Внутри метода finalized нужно определить те действия, которые должны быть выполнены прежде, чем объект будет разрушен. Сборщик мусора отрабатывает периодически, проверяя объекты, на которые нет больше ссылок ни из выполняющихся процессов, ни косвенных — из других действующих объектов. Непосредственно перед освобождением всех активов исполняющая система Java вызывает для объекта метод finalize().

Метод finalized имеет следующую общую форму:

```
protected void finalize()
{
   // Код завершения
}
```

Здесь ключевое слово protected — спецификатор, который запрещает доступ к finalized кодам, определенным вне этого класса.

Важно понять, что finalized вызывается только перед самой сборкой "мусора". Он не запускается, когда объект выходит из области действия идентификаторов, например. Это означает, что нельзя определить, когда finalized будет выполнен (и даже будет ли он выполнен вообще). Поэтому программа должна обеспечить другие средства освобождения системных ресурсов, используемых объектом. Для нормальной работы программы она не должна полагаться на finalized.

Язык C++ позволяет определять деструктор для класса, который вызывается, когда объект выходит из области действия идентификаторов. Java не поддерживает этой идеи и не использует деструкторов. Метод finalized только аппроксимирует функцию деструктора. Потребность в функциях деструктора минимальна из-за наличия в Java подсистемы сборки "мусора".

5.3. Класс Stack

Хотя класс Box полезен для иллюстрации существенных элементов класса, он имеет небольшое практическое значение. Чтобы показать действительную мощь классов, данную главу закончим более сложным примером. Как говорилось выше, одним из наиболее важных преимуществ объектно-ориентированного программирования (ООР), является инкапсуляция данных и кода, который манипулирует этими данными. Механизмом, с помощью которого достигается инкапсуляция, является класс. Создавая класс, мы организуем новый тип данных, который определяет как характер данных, так и подпрограммы, используемые для

манипулирования этими данными. Непротиворечивый и управляемый интерфейс с данными класса определяют методы. Таким образом, можно использовать класс через его методы, не беспокоясь о деталях его реализации или о том, как данные фактически управляются внутри класса. В некотором смысле, класс подобен "машине данных". Чтобы использовать машину через ее органы управления, никаких знаний о том, что происходит внутри машины, не требуется. Фактически, поскольку подробности скрыты, ее внутренняя работа может быть изменена так, как это необходимо. Пока код использует класс через его методы, внутренние подробности могут изменяться, не вызывая побочных эффектов вне класса.

Чтобы получить практическое приложение предшествующего обсуждения, давайте разработаем один из типичных примеров инкапсуляции — стек. *Стек* хранит данные, используя очередь типа LIFO ("Last-In, First-Out") — последним вошел, первым вышел. То есть стек подобен стопке тарелок на столе — последняя тарелка, поставленная на стопку, снимается со стопки первой. Стеки управляются через две операции, традиционно называемые *push* (поместить) и *pop* (извлечь, вытолкнуть). Чтобы поместить элемент в вершину стека, нужно использовать операцию *push*. Чтобы извлечь элемент из стека, нужно использовать операцию *pop*. Заметим, что инкапсуляция полного механизма стека — довольно простая задача.

Программа 12. Стек в виде массива

В следующей программе класс с именем stack реализует стек целых чисел:

```
// Класс стеков для хранения до 10 целых чисел 10
class Stack {
   int stck[] = new int[10];
   int tos;        // Позиция вершины стека
   // Конструктор: инициализировать вершину стека
   Stack(){
      tos = -1;
   }
   // Поместить элемент item в стек
   void push(int item){
      if (tos == 9)
         System.out.println("Стек заполнен.");
      else
         stck[++tos] = item;
   }
   // Извлечь элемент из стека
   int pop (){
      if (tos < 0){
         System.out.println("Стек пуст.");
         return 0;
      }
      else
         return stck[tos--];
   }
}
```

Класс stack определяет два элемента данных и три метода. Стек целых чисел содержится в массиве stck. Этот массив индексирован переменной tos, которая всегда содержит индекс вершины стека. Конструктор stack() инициализирует tos значением -1, которое указывает, что стек пуст. Метод push() помещает элемент в стек. Метод pop() извлекает элемент из стека. Так как доступ к стеку выполняется через push() и pop(), тот факт, что стек содержится в массиве, не мешает использованию стека. Например, стек мог бы храниться в более сложной структуре данных, скажем, типа связного списка, а интерфейс, определенный методами push() и pop(), остался бы тем же самым.

Показанный ниже класс Teststack, демонстрирует работу с классом stack. Он создает два целочисленных стека, помещает некоторые значения в каждый и затем выталкивает их.

```
class Teststack{
   public static void main(String args[]){
      Stack mystack1 = new Stack();
      Stack mystack2 = new Stack();
      System.out.println("Помещаем числа в 1-й стек");
      for(int i=0; i < 10; i++){
         System.out.print(i + " ");
         mystack1.push(i);
      }
      System.out.println("\nПомещаем числа во 2-й стек");
      for(int i = 10; i < 20; i++){
         System.out.print(i + " ");
         mystack2.push(i);
      }
      System.out.println("\nИзвлекаем числа из 1-го стека");
      for(int i = 0; i < 10; i++)
         System.out.print(mystack1.pop() + " ");
      System.out.println("\nИзвлекаем числа из 2-го стека");
      for(int i = 0; i < 10; i++)
         System.out.print(mystack2.pop() + " ");
   }
}
```

Эта программа генерирует следующий вывод:

```
Помещаем числа в 1-й стек
0 1 2 3 4 5 6 7 8 9
Помещаем числа во 2-й стек
10 11 12 13 14 15 16 17 18 19
Извлекаем числа из 1-го стека
9 8 7 6 5 4 3 2 1 0
Извлекаем числа из 2-го стека
19 18 17 16 15 14 13 12 11 10
```

Видно, что содержимое стеков различно.