4. Классы и методы

4.1. Перегрузка методов

Перегрузкой называется возможность давать нескольким методам одинаковые имена. Методы с одинаковыми именами должны отличаться числом и типом аргументов. Тип возвращаемого значения не учитывается.

При вызове нужный метод выбирается из нескольких перегруженных по соответствию фактических аргументов формальным параметрам. Если точного соответствия нет, выбирается метод, для которого возможно автоматическое преобразование аргументов к нужному типу.

Программа 13. Перегрузка методов

```
// Автоматическое преобразование типов в применении к перегрузке.
class OverloadDemo{
   void test(){
      System.out.println ("Параметры отсутствуют") ;
   }
// Перегруженный test с двумя int-параметрами.
   void test (int a,  int b)   {
      System.out.println("a и b:  " + a + " " + b);
   }
// Перегруженный test с double-параметром.
   void test(double a) {
      System.out.println("Внутри test(double)  a:  " + a);
   }
}
class Overload {
   public static void main(String args[]) {
      OverloadDemo ob = new OverloadDemo();
      int i = 88;
      ob.test();
      ob.test (10, 20) ;
      ob.test(i);           // здесь будет вызван test(double)
      ob.test(123.2);       // здесь будет вызван test(double)
   }
}
```

Эта программа генерирует следующий вывод:

```
Параметры отсутствуют
a и b: 10 20
Внутри test(double) a: 88
Внутри test(double) a:  123.2
```

Эта версия `OverloadDemo` не определяет `test (int)` с одним целым параметром, поэтому, для `test(i)` никакого согласованного метода не находится. Однако Java может автоматически преобразовывать `int` в `double`, и это преобразование можно использовать для разрешения вызова. Поэтому, после того, как `test(int)` не находится, Java расширяет `i` до `double` и затем вызывает `test(double)`. Конечно, если бы `test(int)` был определен, то он вызывался бы вместо `test(double)`. Java использует эти автоматические преобразования типов только тогда, когда никакого точного соответствия не находится.

Перегрузка методов поддерживает полиморфизм, потому что это один из способов, с помощью которых Java реализует парадигму "один интерфейс, множество методов". В языках, которые не поддерживают перегрузку методов, каждому методу необходимо давать уникальное имя. Однако часто нужно реализовать, по существу, один и тот же метод для различных типов данных. Рассмотрим функцию абсолютного значения. На языках, которые не поддерживают

перегрузку, существует обычно три или более версий этой функции, каждая со слегка отличающимся именем. Например, в С, функция abs() возвращает абсолютное значение целого числа, labs() возвращает абсолютное значение длинного целого числа, а fabs() — абсолютное значение числа с плавающей точкой. Так как С не поддерживает перегрузку, каждая функция должна иметь свое собственное имя, даже при том, что все три функции выполняют, по существу, одно и то же. Это делает ситуацию более сложной, чем она фактически есть на самом деле. Хотя основная концепция каждой функции одна и та же, вам все еще нужно помнить три разных имени. Подобная ситуация отсутствует в Java, потому что метод получения абсолютного значения един для всех типов данных. Действительно, библиотека стандартных классов Java включает метод абсолютного значения, с именем abs(). Этот метод перегружен в Math-классе Java, чтобы обрабатывать все числовые типы. Java определяет, какую версию abs о вызывать, основываясь на типе аргумента. Значение перегрузки заключается в том, что она позволяет осуществлять доступ к связанным методам при помощи общего имени. Таким образом, имя abs представляет *общее выполняемое действие.* Право же выбирать правильную специфическую версию для конкретного обстоятельства предоставлено компилятору. Вы же, как программист, должны только помнить общую выполняемую операцию. При использовании полиморфизма несколько имен были сокращены до одного. Хотя этот пример довольно прост, но если расширить концепцию, то можно увидеть, как перегрузка может помочь вам управлять большей сложностью.

Когда вы перегружаете метод, каждая версия этого метода может выполнять любое действие, какое вы пожелаете. Нет никакого правила, заявляющего, как перегруженные методы должны быть связаны друг с другом. Однако, со стилистической точки зрения, перегрузка метода подразумевает некоторую их взаимосвязь. Таким образом, хотя можно использовать то же самое имя для разных задач, делать этого не следует. Например, можно использовать имя sqr для метода получения квадрата целого числа и для методы, извлекающего квадратный корень из числа с плавающей точкой, но это усложнит понимание программы, поэтому делать так не следует.

4.2. Перегрузка конструкторов

Программа 14. Три конструктора в классе Box

```
/* Класс Box: три конструктора для разных способов инициализации размеров блока.
class Box {
   double width;
   double height;
   double depth;
// конструктор для инициализации всех размеров
   Box(double w, double h, double d)   {
      width = w;
      height = h;
      depth = d;
   }
// конструктор для инициализации без указания размеров
   Box ()  {
      width = -1; // использовать--1 для указания
      height = -1; // не инициализированного
      depth = -1; // блока
   }
// конструктор для создания куба
   Box{double len)   {
      width = height = depth = len;
   }
// вычислить и возвратить объем
   double volume()   {
      return width * height * depth;
   }
```

```
}
class OverloadCons {
   public static void main(String args[])   {
// создать блоки, используя различные конструкторы
      Box mybox1 = new Box(10, 20, 15);
      Box mybox2 = new Box();
      Box mycube = new Box(7) ;
      double vol;
   // получить объем первого блока
      vol = mybox1.volume() ;
      System.out.println("Объем mybox1 равен " + vol);
// получить объем второго блока
      vol = mybox2.volume ();
      System.out.println("Объем mybox2 равен " + vol);
// получить объем куба
      vol = mycube.volume();
      System.out.println("Объем mycube равен " + vol);
   }
}
```

Вывод, выполненный этой программой:

```
Объем mybox1 равен 3000.0
Объем mybox2 равен -1.0
Объем mycube равен 343.0
```

Как вы видите, подходящий перегруженный конструктор вызывается, основываясь на параметрах, указанных при выполнении операции new.

4.3. Использование объектов в качестве параметров

Программа 15. Объекты как параметры методов

```
// Класс Box: Создание копии объекта
class Box {
   double width;
   double height;
   double depth;
   // построить клон объекта
   Box(Box ob){ // переслать объект конструктору
      width  = ob.width;
      height = ob.height;
      depth  = ob.depth;
   }
   // Сравнить объекты
   boolean equals(Box b){
      if(b.width == width && b.height == height && b.depth == depth)
         return true;
      else
         return false;
   }
// конструктор для инициализации всех размеров
   Box(double w, double h, double d)   {
      width = w;
      height = h;
      depth = d;
   }
// конструктор для инициализации без указания размеров
   Box ()  {
      width = -1; // использовать--1 для указания
      height = -1; // не инициализированного
```

```
      depth = -1; // блока
   }
// конструктор для создания куба
   Box(double len)   {
      width = height = depth = len;
   }
// вычислить и возвратить объем
   double volume()   {
      return width * height * depth;
   }
}

public class ObjectsAsParameters {
   public static void main(String args[])   {
// создать блоки, используя различные конструкторы
      Box mybox1 = new Box(10, 20, 15);
      Box mybox2 = new Box();
      Box copymybox1 = new Box(mybox1);
      // Сравнить mybox1 и mybox2
      System.out.println ("(mybox1 == mybox2) = " + mybox1.equals(mybox2));
// Сравнить mybox1 и copymybox1
      System.out.println ("(mybox1 == copymybox1) = " + myboxl.equals(copymybox1));
   }
}
```

Программа выдает

```
(mybox1 == mybox2) = false
(mybox1 == copymybox1) = true
```

4.4. Рекурсия

Java поддерживает *рекурсию.* Рекурсия — это определение чего-то в терминах самого себя. Метод, который вызывает сам себя называется рекурсивным.

Классический пример рекурсии — вычисление факториала числа. Фактори-1Л числа N есть произведение всех целых чисел между 1 и N. Например, факториал 3 равен 1x2x3 или 6. В приводимой ниже программе показано, как факториал может быть вычислен при помощи рекурсивного метода.

Программа 16. Рекурсивное вычисление факториала

```
// Простой пример рекурсии.
class Factorial {
// это рекурсивная функция
   int fact(int n)   {
      int result;
      if(n == 1)
         return 1;
      result = fact(n-1)  * n;
      return result;
   }
}
class Recursion {
   public static void main(String args[])   {
      Factorial f = new Factorial();
      System.out.println("Факториал 3 равен " + f.fact(3));
      System.out.println("Факториал 4 равен " + f.fact(4));
      System.out.println("Факториал 5 равен " + f.fact(5));
   }
}
```

Вывод этой программы:

```
факториал 3 равен 6
факториал 4 равен 24
факториал 5 равен 120
```

4.5. Статические элементы

Иногда возникает необходимость определить элемент (член) класса так, чтобы появилась возможность пользоваться им независимо от какого-либо объекта этого класса. Обычно к элементам класса нужно обращаться только через объект этого класса. Однако можно создать элемент для использования без ссылки на определенный объект. Чтобы это сделать, в начале его объявления указывается ключевое слово `static`. Когда элемент объявляется как `static`, к нему можно обращаться до того, как создаются какие-либо объекты его класса, и без ссылки на какой-либо объект. Статическими можно объявлять как методы, так и переменные. Примером `static`-элемента является метод `main()`. Он объявляется как `static`, потому что должен вызваться прежде, чем будут созданы какие-либо объекты.

Переменные экземпляра, объявленные как `static`, это, по существу, глобальные переменные. Когда создаются объекты их класса, никакой копии `static`-переменной не делается. Вместо этого, все экземпляры класса совместно используют (разделяют) одну и ту же `static`-переменную.

Методы, объявленные как `static` имеют несколько ограничений:

□ могут вызывать только другие `static`-методы;

□ должны обращаться только к `static`-данным;

□ никогда не могут ссылаться на `this` или `super`. (Ключевое слово `super` касается наследования и описано в следующей главе.)

Для организации вычислений с целью инициализировать статические переменные можно объявить *статический блок,* который выполняется только один раз, когда класс впервые загружается. Следующий программа демонстрирует класс, который имеет статические методы, несколько переменных и блок инициализации.

Программа 17. Статические члены класса

```
//Демонстрирует статические переменные, методы и блоки,
class StaticMembers {
   static int a = 3;
   static int b;
   static void meth(int x) {
      System.out.println("x = " + x);
      System.out.println("a = " + a);
      System.out.println("b = " + b);
   }
   static {              // Статический блок класса
      System.out.println ("Статический блок инициализирован.");
      b = a * 4;
   }
}
public class UseStatic{
   public static void main(String args[])  {
      System.out.println("Исходные значения статических членов");
      StaticMembers.meth(42);  // Вызов статического метода без объекта
      System.out.println("Измененные значения статических членов");
      StaticMembers.a = 100;
      StaticMembers.b = 200;
      StaticMembers.meth(300);
   }
}
```

Вывод этой программы:

```
Исходные значения статических членов
Статический блок инициализирован.
x = 42
a = 3
b = 12
Измененные значения статических членов
x = 300
a = 100
b = 200
```

Вне класса, в котором они определены, статические методы и переменные могут использоваться независимо от любого объекта. В этом случае для обращения к ним используются имена их класса и точечная операция. Например, если нужно вызвать static-метод вне его класса, можно воспользоваться следующей общей формой вызова:

```
classname.znethodname()
```

Здесь classname – имя класса, в котором объявлен static-метод; methodname — имя статического метода. Нетрудно заметить, что этот формат подобен вызову нестатических методов через переменные, ссылающиеся на объект. К переменной static можно обращаться таким же образом — при помощи точечной операции с именем класса (в качестве левого операнда). Так Java реализует управляемую версию глобальных функций и глобальных переменных.

4.6. Спецификатор final

Переменную можно объявить как final, предохраняя ее содержимое от изменения. Это означает, что нужно инициализировать final-переменную, когда она объявляется (в таком применении final подобен const в C/C++). Например:

```
final int FILE_NEW    = 1;
final int FILE_OPEN   = 2;
final int FILE_SAVE   = 3;
final int FILE_SAVEAS = 4;
final int FILE_QUIT   = 5;
```

После таких объявлений последующие части программы могут использовать FILE_OPEN и т.д., как константы, без опасения, что их значения были изменены.

Общее соглашение кодирования для final-переменных — выбирать все идентификаторы в верхнем регистре. Переменные, объявленные как final, не занимают память на "поэкземплярной" основе. Таким образом, finai-переменная — по существу константа.

4.7. Длина массивов

Массивы являются объектами, которые имеют переменную length, содержащую размер массива. Следующая программа демонстрирует указанное свойство.

Программа 18. Длина массивов

```
//Эта программа демонстрирует элемент длины массива.
public class LengthArray {
   public static void main(String args[]){
      int a1[] = new int[10];
      int a2[] = {3, 5, 7, 1, 8, 99, 44, -10};
      int a3[] = {4, 3, 2, 1};
      System.out.println("Размер a1 равен " + a1.length);
      System.out.println("Размер a2 равен " + a2.length);
      System.out.println("Размер a3 равен " + a3.length);
   }
}
```

Эта программа отображает (на экран) следующий вывод:

```
Размер a1 равен 10
Размер a2 равен 8
Размер a3 равен 4
```

Здесь видно, что отображается размер каждого массива. Имейте в виду, что значение length не имеет никакого отношения к числу элементов, которые фактически используются. Она отражает только число элементов, на которое массив рассчитан.

Ниже приведена программа, реализующая улучшенную версию класса stack. Предыдущие версии этого класса всегда создавали стек с десятью элементами. Новая версия позволяет создавать стеки любого размера. Чтобы предотвратить переполнение стека, используется значение stckLength.

Программа 19. Использование длины массивов в стеках

```
//Улучшенный Stack-класс, который использует length-элемент массива.
class Stack {
   private int stck[];
   private int tos;
   //выделить память и инициализировать стек
   Stack(int size){
      stck = new int[size];
      tos = -1;
   }
   //поместить элемент в стек
   void push(int item){
      if(tos == stck.length - 1)      //.использовать length-член
         System.out.println("Стек заполнен.");
      else
         stck[++tos] = item;
   }
   //вытолкнуть элемент из стека
   int pop(){
      if(tos < 0){
         System.out.println("Стек пуст.");
         return 0;
      }
      else
         return stck[tos--];
   }
}
public class TestStack2 {
   public static void main(String args[]){
      Stack mystack1 = new Stack(5);
      Stack mystack2 = new Stack(8);
      //поместить несколько чисел в стек
      for(int i = 0; i < 5; i++)
         mystack1.push(i);
      for(int i = 0; i < 8; i++)
         mystack2.push(i);
      //вытолкнуть эти элементы из стека
      System.out.println("Стек в mystack1:");
      for(int i = 0; i < 5; i++)
         System.out.println(mystack1.pop());
      System.out.println("Стек в mystack2:");
      for(int i = 0; i < 8; i++)
         System.out.println(mystack2. pop());
   }
}
```

Программа создает два стека: один глубиной в пять элементов, а другой — в восемь элементов. Тот факт, что массивы поддерживают информацию об их собственной длине, облегчает создание стеков произвольного размера.

4.8. Вложенные и внутренние классы

Существует возможность определения одного класса внутри другого. Такие классы известны как *вложенные* (nested) *классы.* Область видимости вложенного класса ограничивается областью видимости включающего класса. Таким образом, если класс `B` определен в классе `A`, то `B` известен внутри `A`, не вне его. Вложенный класс имеет доступ к членам класса (включая `private`-члены), в который он вложен. Однако включающий класс не имеет доступа к членам вложенного класса.

Существует два типа вложенных классов: *статические* и *нестатические.* Статический вложенный класс — это класс, который имеет модификатор `static` Согласно своей характеристике он должен обращаться к членам своего включающего класса через объект. То есть он не может обратиться к членам включающего класса напрямую. Из-за этого ограничения статические вложенные классы используются редко.

Наиболее важный тип вложенного класса — внутренний (inner) класс *Внутренний класс* — это нестатический вложенный класс, имеющий доступ ко всем переменным и методам своего внешнего класса и возможность обращаться к ним напрямую, таким же способом, как это делают другие нестатические члены внешнего класса. Итак, внутренний класс находится полностью в пределах видимости своего включающего класса.

Следующая программа показывает, как можно определять и использовать внутренний класс. Класс с именем `Outer` имеет одну переменную экземпляра с именем `outer_x`, один метод экземпляра с именем `test()` и определяет один внутренний класс с именем `inner`.

Программа 20. Вложенные классы

```
// Демонстрирует внутренний класс.
class Outer {                       // Это включающий класс
   int outer_x = 100;
   void test() {
      Inner inner = new Inner();
      inner.display();
   }

   class Inner {                    // Это внутренний класс
      void display(){
         System.out.println("В методе display: outer_x = " + outer_x);
      }
   }
}
class InnerClassDemo {
   public static void main(String args[]){
      Outer outer = new Outer();
      outer.test();
   }
}
```

Вывод этой программы:

```
В методе display: outer_x = 100
```

В этой программе внутренний класс с именем `inner` определен в пределах области видимости класса `outer`. Поэтому любой код в классе `inner` может прямо обращаться к переменной `outer_x`. Внутри `inner` определен метод с именем `display()`. Этот метод отображает `outer_x` в поток стандартного вывода. Метод `main()` класса `innerCiassDemo` создает экземпляр (объект) класса `outer` и вызывает его метод `test()`. Этот метод создает экземпляр класса `inner` и вызывает метод `display()`.

Класс inner известен только в пределах области действия (т. е. внутри) класса outer. Java-компилятор генерирует сообщение об ошибке, если какой-то код вне класса outer делает попытку создать экземпляр (объект) класса inner. Итак, вложенный класс ничем не отличается от любого другого программного элемента, но он известен только в пределах включающей его области.

Внутренний класс имеет доступ ко всем членам своего включающего класса, но обратное — не верно. Члены внутреннего класса известны только в пределах внутреннего класса и не могут использоваться внешним классом. Например:

```
// Эта программа компилироваться не будет,
class Outer {
   int outer_x - 100;
   void test(){
      Inner inner = new Inner();
      inner.display();
   }
// это внутренний класс
   class Inner {
      int y = 10;     //y – локальная для Inner
      void display(){
         System.out.println("В методе display: outer_x = " + outer_x);
      }
   }
   void showy(){
      System.out.println(y);    // ошибка, y здесь неизвестна!
   }
}
class InnerCiassDemo {
   public static void main(String args[]){
      Outer outer = new Outer();
      outer.test();
   }
}
```

Здесь y объявлена как переменная экземпляра класса inner. Таким образом, она не известна вне этого класса, и не может использоваться методом showy().

Существует возможность определять внутренние классы в пределах любого блока. Например, можно определить вложенный класс в блоке, установленном в методе или даже в теле цикла for, как показывает следующая программа.

Программа 21. Класс, вложенный в цикл

```
//Определение внутреннего класса в цикле for.
class Outer {
   int outer_x = 100;
   void test()  {
      for(int i = 0; i < 10; i++){
         class Inner{
            void display(){
               System.out.println("В методе display: outer_x = " + outer_x);
            }
         }
         Inner inner = new Inner();
         inner.display();
      }
   }
}
class InnerClass {
   public static void main(String args[]){
   Outer outer = new Outer();
   outer.test() ;
   }
}
```

Вывод этой версии программы:

```
В методе display: outer_x = 100
В методе display: outer_x = 100
В методе display: outer_x = 100
В методе display: outer_x = 100
В методе display: outer_x = 100
В методе display: outer_x = 100
В методе display: outer_x = 100
В методе display: outer_x = 100
В методе display: outer_x = 100
В методе display: outer_x = 100
```

4.9. Класс String

string — это очень часто используемый класс в библиотеке классов Java, так как строки являются очень важной частью программирования.

Каждая создаваемая строка в действительности является объектом типа string. Даже строчные константы — это фактически string-объекты. Например, в инструкции

```
System.out.println("This is a String, too");
```

строка "This is a String, too" является string-константой. Java обрабатывает string-константы так же, как другие машинные языки обрабатывают "нормальные" строки.

Кроме того, нужно понять, что объекты типа string неизменяемы. Если string-объект создан, то его содержимое не может быть изменено. Позднее будут рассмотрены способы работы со строками. В том числе и по их изменению.

Строки можно создать множеством способов. Самый простой — с помощью следующей инструкции:

```
String myString = "Это тест";
```

Созданные string-объекты, можно использовать везде, где допустима строка. Например, следующая инструкция выводит mystring-объект на экран дисплея:

```
System.out.println(myString);
```

Java определяет одну операцию для string-объектов, которая обозначается знаком плюс (+). Она используется для сцепления (конкатенации) двух строк. Например, такой оператор:

```
String myString = "Мне" + " нравится программировать на " + "Java.";
```

приводит к следующему содержимому объекта myString: "Мне нравится программировать на Java".

Класс string содержит несколько полезных методов.

Метод

```
boolean equals(String-object)
```

проверяет две строки на равенство.

Длину строки возвращает метод

```
int length()
```

Символ с заданным номером index возвращает метод

```
char charAt(int index)
```

Рассмотренные действия со строками демонстрирует следующая программа.

Программа 22. Строки и сравнение строк

```
// Действия со строками
class StringDemo{
   public static void main(String args[]){
      String strOb1 = "Первая строка";
```

```
      String strOb2 = "Вторая строка";
      String strOb3 = strOb1 + " и " + strOb2;
      System.out.println(strOb1);
      System.out.println(strOb2);
      System.out.println(strOb3);
      String strOb4 = strOb1;
      System.out.println("Длина strOb1: " + strOb1.length());
      System.out.println ("Символ с индексом 3 в strOb1: " + strOb1.charAt(3));
      if(strOb1.equals(strOb2))
         System.out.println("strOb1 == strOb2");
      else
         System.out.println("strOb1 != strOb2");
      if(strOb1.equals(strOb4))
         System.out.println("strOb1 == strOb4");
      else
         System.out.println("strOb1 != strOb4");
   }
}
```

Вывод, выполняемый этой программой:

```
Первая строка
Вторая строка
Первая строка и Вторая строка
Длина strOb1: 13
Символ с индексом 3 в strOb1: в
strOb1 != strOb2
strOb1 == strOb4
```

Можно, конечно, создавать и *массивы строк,* аналогично тому, как создаются массивы любого другого типа объектов. Пример использования массива строк приведен в следующей программе.

Программа 23. Массив строк

```
// Демонстрирует String-массивы
public class ArrayStrings {
   public static void main(String args[]){
      String str[] = { "один", "два", "три" };
      for(int i = 0; i < str.length; i++)
         System.out.println("str[" + i + "] : " + str[i]);
   }
}
```

Вывод этой программы:

```
str[0] : один
str[1] : два
str[2] : три
```

4.10. Использование аргументов командной строки

Иногда нужно переслать информацию в программу во время ее выполнения Это делается пересылкой аргументов командной строки методу `main()` *Аргументы командной строки* — это информация, которая следует непосредственно за именем программы в командной строке, используемой для запуска программы. Аргументы командной строки сохраняются как строки в `string`-массиве, пересылаемом в `main()`. Например, следующая программа отображает все аргументы командной строки, с которыми она вызывается.

Программа 24. Аргументы командной строки

```
// Показать все аргументы командной строки
```

```
public class ArgumentCommandLine {
   public static void main(String args[]){
      for(int i = 0; i < args.length; i++)
         System.out.println("args[" + i + "] : " + args[i]);
   }
}
```

Выполним компиляцию программы, а затем в командной строке наберем команду для ее запуска:

```
D:\>java -classpath D:\EclipseJunoWorkspace\Progr24_CommandLine\bin ArgunentCommandLine abc
123 xyz
```

После этого должен появиться такой вывод:

```
args[0] : abc
args[1] : 123
args[2] : xyz
```

Все аргументы командной строки пересылаются как строки. Если нужно передать через командную строку числа, их следует преобразовать из строкового представления в числовое.