1. Наследование

Наследование позволяет создавать новые классы на базе существующих классов. Класс, на основе которого создается новый класс называется *суперклассом* (superclass). Новый класс, создаваемый га основе суперкласса, называется *подклассом* (subclass). Подкласс — это специализированная версия суперкласса. Он наследует все переменные и методы, определенные суперклассом, и прибавляет свои собственные уникальные элементы.

1.1. Основы наследования

Наследуемый класс (суперкласс), указывается после ключевого слова extends при создании подкласса. Следующая программа создает суперкласс с именем A и подкласс с именем B.

Программа 25. Пример наследования

```
// Простой пример наследования.
// Создать суперкласс.
class A {
   int i, j;
   void showij() {
      System.out.println("i и j:  " + i + " " + j);
   }
}
// Создать подкласс расширением класса A.
class B extends A {
   int k;
   void showk() {
      System.out.println("k: " + k);
   }
   void sum() {
      System.out.println("i+j+k: " + (i + j + k));
   }
}
class Simplelnheritance {
   public static void main(String args[]){
      A superOb = new A();
      B subOb = new B();
      // Суперкласс может быть использован сам по себе.
      superOb.i = 10;
      superOb.j = 20;
      System.out.println("Содержимое superOb:  ");
      superOb.showij();
      System.out.println();
/* Подкласс имеет доступ ко всем public-членам
его суперкласса. */
```

```
        subOb.i = 7;
        subOb.j = 8;
        subOb.k = 9;
        System.out.println("Содержимое of subOb: ");
        subOb.showij();
        subOb.showk();
        System.out.println();
        System.out.println("Сумма i, j и k в subOb:");
        subOb.sum();
    }
}
```

Подкласс B включает все члены его суперкласса A. Вот почему объект subOb может обращаться к i и j и вызывать showij(). Поэтому же внутри sum() можно прямо ссылаться на i и j, как если бы они были частью B.

Хотя A — суперкласс для B, он тоже полностью независимый, автономный класс. Роль суперкласса для некоторого подкласса не означает, что этот суперкласс не может использоваться сам по себе. Более того, подкласс может быть суперклассом для другого подкласса.

Программы выводит:

```
Содержимое superOb:
i и j:  10 20

Содержимое of subOb:
i и j:  7 8
k: 9

Сумма i, j и k в subOb:
i+j+k: 24
```

Ниже показана общая форма объявления класса, который наследует суперкласс:

```
class subclass-name extends superclass-name {
// тело класса
}
```

где subclass-name — имя подкласса, superclass-name — имя суперкласса. Подкласс может создаваться на базе только одного суперкласса. Java не поддерживает наследования множества суперклассов в одиночном подклассе. (Это отличается от C++, в котором можно наследовать одновременно нескольким базовым классам.) Подкласс можно использовать как суперклассо другого подкласса.

1.2. Доступ к элементам и наследование

Хотя подкласс включает все элементы (члены) своего суперкласса, он не может обращаться к тем элементам суперкласса, которые были объявлены как private. Например, рассмотрим следующую простую иерархию классов:

```
/* В иерархии классов private члены остаются private для ее классов.
Эта программа содержит ошибку и не будет компилироваться.  */
// Создать суперкласс.
class A {
   int i;             // public по умолчанию
   private int j;     // private для A
   void setij(int x, int y) {
      i = x;
      j = y;
   }
}
// j класса A здесь не доступна.
class B extends A {
   int total;
   void sum () {
      total = i + j;  // ОШИБКА,  j здесь не доступна
   }
}
class Access {
   public static void main(String args[]) {
      B subOb = new B();
      subOb.setij(10, 12);
      subOb.sum();
      System.out.println("Всего " + subOb.total);
   }
}
```

Эта программа не будет компилироваться, потому что ссылка на j внутри метода sum() класса B вызывает нарушение правил доступа. С тех пор как j объявлена с помощью private, она доступна только другим членам его собственного класса. Подклассы не имеют доступа к ней.

1.3. Практический пример

Рассмотрим практический пример, который поможет проиллюстрировать мощь наследования. Расширим последнюю версию класса Box, включив четвертый компонент с именем weight. Таким образом, новый класс будет содержать ширину, высоту, глубину и вес блока.

Программа 26. Пример наследования

```
//Программа использует наследование для расширения Box.
```

```
class Box { // Класс из прогр. 15.
   double width;
   double height;
   double depth;
// создать клон объекта
   Box(Box ob){ // переслать объект конструктору
      width = ob.width;
      height = ob.height;
      depth = ob.depth;
   }
// конструктор, используемый, когда указаны все размеры
   Box(double w, double h, double d)   {
      width = w;
      height = h;
      depth = d;
   }
// конструктор, используемый, когда размеры не указаны
   Box ()   {
      width  = -1; // использовать -1 для указания
      height = -1; // неинициализированного
      depth  = -1; // блока
   }
// конструктор, используемый для создания куба
   Box(double len)   {
      width = height = depth = len;
   }
// вычислить и вернуть объем
   double volume()   {
      return width * height * depth;
   }
}

class BoxWeight extends Box {  // Box расширяется для включения веса.
   double weight;              // вес блока
// конструктор для BoxWeight
   BoxWeight(double w, double h, double d, double m){
      width = w; height = h; depth = d; weight = m;
   }
}

class DemoBoxWeight {
   public static void main(String args[])   {
      BoxWeight mybox1 = new BoxWeight(10, 20,  15,  34.3);
      BoxWeight mybox2 = new BoxWeight(2,  3,   4,  0.076);
      double vol;
      vol = mybox1. volume () ;
      System.out.println("Объем mybox1 равен " + vol);
      System.out.println("Вес mybox1 равен " + mybox1.weight);
      System.out.println();
      vol = mybox2.volume();
      System.out.println("Объем mybox2 равен " + vol);
      System.out.println("Вес mybox2 равен " + mybox2.weight);
   }
}
```

Вывод этой программы:

```
Объем mybox1 равен 3000.0
Вес mybox1 равен 34.3

Объем mybox2 равен 24.0
Вес mybox2 равен 0.076
```

BoxWeight наследует все характеристики Box и прибавляет к ним компонент weight. Для BoxWeight нет необходимости заново создавать все свойства Box. Можно просто расширить Box, чтобы достичь своих собственных целей.

Главное преимущество наследования состоит в том, что, как только вы создали суперкласс, который определяет общие для набора объектов атрибуты, его можно использовать для создания любого числа более специфичных подклассов. Каждый подкласс может добавить свою собственную классификацию. Например, следующий класс наследует Box и прибавляет цветовой атрибут:

```
// Box расширяется для включения цвета.

class ColorBox extends Box {
   int color;                          // color of box
   ColorBox(double w, double h, double d, int c) {
      width = w; height = h; depth = d; color = c;
   }
}
```

Как только создан суперкласс, который определяет общие аспекты объекта, он может наследоваться для формирования специализированных классов. Каждый подкласс просто прибавляет свои собственные, уникальные атрибуты. В этом сущность наследования.

1.4. Переменная суперкласса может ссылаться на объект подкласса

Ссылочной переменной суперкласса может быть назначена ссылка на любой подкласс, производный от этого суперкласса. Рассмотрим следующий пример:

```
class RefDemo {
   public static void main(String args[]) {
      BoxWeight weightbox = new BoxWeight(3, 5, 7, 8.37);
      Box plainbox = new Box();
      double vol;
      vol = weightbox.volume();
      System.out.println("Объем weightbox равен " + vol);
      System.out.println("Вес weightbox равен " + weightbox.weight);
      System.out.println();
      // назначить ссылку на BoxWeight ссылке на Box
      plainbox = weightbox;
```

```
        vol = plainbox.volume();        // OK, volume() определена в Box
        System.out.println{"Объем plainbox равен " + vol);
        /* Следующее утверждение не верно, потому что plainbox не определяет
           член weight. */
    // System.out.println("Вес plainbox равен " + plainbox.weight);
    }
}
```

Здесь weightbox — ссылка на объекты BoxWeight, а plainbox — ссылка на Box-объекты. Так как BoxWeight — подкласс Box, допустимо назначить plainbox ссылку на объект weightbox.

Важно, что тип ссылочной переменной, а не тип объекта, на который она ссылается, определяет, к каким членам можно обращаться. То есть, когда ссылка на объект подкласса указывает на ссылочную переменную суперкласса, вы будете иметь доступ только к тем частям объекта, которые определены суперклассом. Вот почему plainbox не может ссылаться на weight, даже когда она ссылается на BoxWeight-объект. Данный подход оправдан, потому что суперкласс не знает, что к нему добавляет подкласс. Вот почему последняя строка кода в предшествующем фрагменте закомментирована. У ссылки нет возможности обратиться к полю **weight,** потому что в **Box** оно не определено.

Хотя предыдущие рассуждения могут показаться немного экзотическими, с ним связаны некоторые важные практические приложения (два таких приложения обсуждаются позже в этой главе).

1.5. Использование ключевого слова super

В предшествующих примерах классы, производные от Box, не были реализованы так эффективно или устойчиво, как это могло бы быть. Например конструктор класса BoxWeight явно инициализирует поля width, height и depth метода Box(). Мало того, что он дублирует код своего суперкласса, что неэффективно, но это означает, что подклассу должен быть предоставлен доступ к этим членам. Однако наступит момент, когда вы захотите создать суперкласс, который сохраняет подробности своей реализации для себя (т. е хранит свои компоненты данных как private). В этом случае у подкласса не будет никакой возможности прямого доступа или инициализации этих переменных как своих собственных. Так как инкапсуляция это первичный атрибут ООР, не удивительно, что Java обеспечивает решение этой проблемы Всякий раз, когда подкласс должен обратиться к своему непосредственному суперклассу, он может сделать это при помощи ключевого слова super.

Ключевое слово super имеет две общие формы. Первая вызывает конструктор суперкласса. Вторая используется для доступа к элементу суперкласса, который был скрыт элементом подкласса. Далее рассматриваются обе формы.

Вызов конструктора суперкласса с помощью первой формы *super*

Подкласс может вызывать конструктор, суперкласса при помощи следующей формы super:

```
super (parameter-list);
```

Здесь parameter-list — список параметров, который определяет любые параметры, необходимые конструктору в суперклассе. Похожий по форме на конструктор super() должен всегда быть первым оператором, выполняемым внутри конструктора подкласса.

Чтобы посмотреть, как super() используется, приведем следующую улучшенную версию класса BoxWeight:

```
// BoxWeight теперь использует super для инициализации Box-атрибутов.
class BoxWeight extends Box {
   double weight; // вес блока
// инициализировать width, height и depth, используя super()
   BoxWeight(double w, double h, double d, double m){
      super(w, h, d);       // вызвать конструктор суперкласса
      weight = m;
   }
}
```

Здесь BoxWeight() вызывает super() с параметрами w, h и d. Используя эти параметры, super() вызывает конструктор Box(), который инициализирует width, height и depth. BoxWeight больше не инициализирует эти значения самостоятельно. Он нуждается в инициализации только уникальной для него переменной — weight. Box, если пожелает, может закрыть свои переменные (сделать их private).

В предыдущем примере, super() вызывался с тремя параметрами. Так как конструкторы могут быть перегружены, super() может вызывать любую их форму, определенную в суперклассе. Выполняться будет тот конструктор, который соответствует аргументам super(). Например, имеется законченная реализация BoxWeight, которая обеспечивает конструкторы для различных способов создания блока. В каждом случае вызывается super(), использующий соответствующие параметры. Обратите внимание, что width, height и depth объявлены частными (private) в классе Box.

Программа 27. Использование super для вызова конструктора

```
// Полная реализация BoxWeight.
class Box {           // Можно использовать код программы 15
   double width;
   double height;
   double depth;
```

```
// создать клон объекта
   Box(Box ob){     // передать объект конструктору
      width = ob.width;
      height = ob.height;
      depth = ob.depth;
   }
// конструктор, используемый, когда указаны все размеры
   Box(double w, double h, double d)   {
      width = w;
      height = h;
      depth = d;
   }
// конструктор, используемый, когда размеры не указаны
   Box() {
      width  = -1;    // использовать -1 для указания
      height = -1;    // неинициализированного
      depth  = -1;    // блока
   }
// конструктор, использутзый для создания куба
   Box(double len)   {
      width = height = depth = len;
   }
// вычислить и вернуть объем
   double volume()   {
      return width * height * depth;
   }
}

//BoxWeight теперь полностью реализует все конструкторы.
class BoxWeight extends Box {
   double weight;                  // вес блока
   //построить клон объекта
   BoxWeight(BoxWeight ob){       // передать объект конструктору
      super(ob);                   // Вызов конструктора суперкласса Box
      weight = ob.weight;
   }
//конструктор, используемый для всех размеров
   BoxWeight(double w, double h, double d, double m)   {
      super(w, h, d);              // вызвать конструктор суперкласса
      weight = m;
   }
//конструктор по умолчанию
   BoxWeight()   {
      super();              // Вызов конструктора по умолчанию суперкласса
      weight = -1;
   }
//конструктор, используемый для создания куба
   BoxWeight(double len, double m)  {
      super(len);
      weight = m;
   }
}
class DemoSuper {
   public static void main(String args[]){
      BoxWeight mybox1 = new BoxWeight(10, 20, 15, 34.3);
```

```
        BoxWeight mybox2 = new BoxWeight(2, 3, 4, 0.076);
        BoxWeight mybox3 = new BoxWeight (); //по умолчанию
        BoxWeight mycube = new BoxWeight(3, 2);
        BoxWeight myclone = new BoxWeight(mybox1);
        double vol;
        vol = mybox1.volume ();
        System.out. println ("Объем mybox1 равен " + vol);
        System.out.println ("Вее mybox1 равен " + mybox1.weight);
        System.out.println();
        vol = mybox2.volume ();
        System.out.println ("Объем mybox2 равен " + vol);
        System.out.println("Вее mybox2 равен " + mybox2.weight);
        System.out.println();
        vol = mybox3.volume();
        System.out.println("Объем туьох3 равен " + vol);
        System.out.println("Вес туьох3 равен " + mybox3.weight);
        System.out.println();
        vol = myclone.volume();
        System.out.println("Объем myclone равен " + vol);
        System.out.println("Вее myclone равен " + myclone.weight);
        System.out.println();
        vol = mycube.volume ();
        System.out.println("Объем mycube равен " + vol);
        System.out.println("Вее mycube равен " + mycube.weight);
        System.out.println();
    }
}
```

Эта программа генерирует следующий вывод:

```
Объем mybox1 равен 3000.0
Вее mybox1 равен 34.3

Объем mybox2 равен 24.0
Вее mybox2 равен 0.076

Объем туьох3 равен -1.0
Вес туьох3 равен -1.0

Объем myclone равен 3000.0
Вее myclone равен 34.3

Объем mycube равен 27.0
Вее mycube равен 2.0
```

Обратите особое внимание на следующий конструктор в BoxWeight():

```
// построить клон объекта
BoxWeight(BoxWeight ob) { // передать объект конструктору
   super(ob);
   weight = ob.weight;
}
```

Заметьте, super() вызывается с объектом типа BoxWeight — не типа Box. Но вызывается при этом все же конструктор Box(Box ob). Как было упомянуто ранее, переменная суперкласса может использоваться для ссылки на любой производный объект этого класса. Таким образом, можно передать объект BoxWeight конструктору Box. Но, конечно, только Box "знает" свои собственные члены.

Сделаем обзор ключевых концепций super(). Когда подкласс вызывает super(), он вызывает конструктор своего непосредственного суперкласса. Таким образом, super() всегда обращается к непосредственному суперклассу вызывающего класса. Это справедливо даже в многоуровневой иерархии. Кроме того, super() всегда должен быть первым оператором, выполняемым внутри конструктора подкласса.

Использование второй формы *super*

Вторая форма super действует в чем-то подобно ссылке this, за исключением того, что она всегда обращается к суперклассу подкласса, в котором используется. Общий формат такого использования super имеет вид:

```
super. member
```

где member может быть либо методом, либо переменной экземпляра.

Вторая форма super применима в ситуациях, когда имена элементов (членов) подкласса скрывают элементы с тем же именем в суперклассе. Рассмотрим следующую простую иерархию классов:

Программа 28. Использование super для доступа к членам суперкласса

```
// Использование super для преодоления скрытия имен.
class A{
   int i;
}
// Создание подкласса B расширением класса A.
class B extends A {
   int i;                // этот i скрывает i в A
   B(int a,  int b){
      super.i = a;       // i из A
      i = b;             // i из B
   }
   void show()   {
      System.out.println("i из суперкласса:  " + super.i);
      System.out.println("i из подкласса:    " + i);
   }
}
```

```
class UseSuper {
   public static void main(String args[]){
      B subOb = new B(1,  2);
      subOb.show();

   }
}
```

Эта программа выполняет следующий вывод:

```
i из суперкласса:  1
i из подкласса:    2
```

Хотя экземплярная переменная i класса B скрывает i класса A, super позволяет получить доступ к i, определенной в суперклассе. Кроме того, super можно также использовать для вызова методов, скрытых подклассом.

1.6. Создание многоуровневой иерархии

До этого момента мы использовали простые иерархии классов, которые состоят только из суперкласса и подкласса. Однако можно строить иерархии, которые содержат несколько уровней наследования. Как уже говорилось, можно использовать подкласс в качестве суперкласса другого класса. Например, при наличии трех классов с именам A, B и C, класс C может быть подклассом B, который, в свою очередь, является подклассом A. В этом случае каждый подкласс наследует все черты всех своих суперклассов. В данном случае C наследует все аспекты B и A. Чтоб продемонстрировать пользу многоуровневой иерархии, рассмотрим следующую программу:

Программа 29. Многоуровневая иерархия

```
// Расширить BoxWeight для включения стоимости отгрузки.
// Начать с класса Box.
class Box {                  // См. прогр. 15
   private double width;
   private double height;
   private double depth;
// создать клон объекта
   Box(Box ob)   {       // Передать объект конструктору
      width  = ob.width;
      height = ob.height;
      depth  = ob.depth;
   }
// конструктор, использующий все размеры
   Box(double w, double h, double d)   {
      width = w;
      height = h;
      depth = d;
```

```
   }
//конструктор, не использующий размеров
   Box(){
      width  = -1; // использовать -1 для указания
      height = -1; // неинициализированного
      depth  = -1; // блока
   }
//конструктор для создания куба
   Box(double len)   {
      width = height = depth = len;
   }
//вычислить и возвратить объем
   double volume()   {
      return width * height * depth;
   }
}
// Добавить вес.
class BoxWeight extends Box {      // См. Прогр. 27
   double weight;  // вес блока
//создать клон объекта
   BoxWeight(BoxWeight ob) {       // передать объект конструктору
      super(ob);
      weight = ob.weight;
   }
//конструктор, использующий все специфицированные параметры
   BoxWeight(double w, double h, double d, double m){
      super(w, h, d); // вызвать конструктор суперкласса
      weight = m;
   }
//конструктор по умолчанию
   BoxWeight()  {
      super();
      weight = -1;
   }
//конструктор для создания куба
   BoxWeight(double len, double m){
      super(len);
      weight = m;
   }
}
// Добавить стоимость отгрузки
class Shipment extends BoxWeight {
   double cost;
//построить клон объекта
   Shipment(Shipment ob)  { // передать объект конструктору
      super(ob);
      cost = ob.cost;
   }
//конструктор для всех специфицированных параметров
   Shipment(double w, double h, double d, double m, double c)  {
      super(w, h, d, m); // вызвать конструктор суперкласса
      cost = c;
   }
//умалчиваемый конструктор
   Shipment() {
```

```
      super();     // Вызов конструктора суперкласса BoxWeight
      cost = -1;
   }
//конструктор для создания куба
   Shipment(double len, double m, double c){
      super(len, m);
      cost = c;
   }
}
class DemoShipment {
   public static void main(String args[]){
      Shipment shipment1 = new Shipment(10, 20, 15, 10, 3.41);
      Shipment shipment2 = new Shipment(2,  3,  4,  0.76,  1.28);
      double vol;
      vol = shipment1.volume();
      System.out. println ("Объем shipmentl равен " + vol);
      System.out.println("Вее shipmentl равен " + shipment1.weight);
      System.out.println("Стоимость отгрузки: $" + shipment1.cost);
      System.out.println();
      vol = shipment2.volume();
      System.out. println("Объем shipment2 равен " + vol);
      System.out.println("Вее shipment2 равен " + shipment2.weight);
      System.out.println ("Стоимость отгрузки: $" + shipment2. cost);
   }
}
```

Здесь подкласс BoxWeight используется как суперкласс для создания подкласса с именем Shipment. Класс Shipment наследует все члены классов BoxWeight и Box и добавляет поле с именем cost (стоимость), которое содержит стоимость отгрузки такого пакета.

Вывод этой программы:

```
Объем shipmentl равен 3000.0
Вее shipmentl равен 10.0
Стоимость отгрузки: $3.41

Объем shipment2 равен 24.0
Вее shipment2 равен 0.76
Стоимость отгрузки: $1.28
```

Из-за наследования Shipment может использовать предварительно определенные классы Box и BoxWeight, добавляя только дополнительную информацию, которая требуется для своего собственного специфического применения. Одно из преимуществ наследования — возможность повторного использования кода.

Этот пример иллюстрирует другой важный аспект: super() всегда обращается к конструктору в самом близком суперклассе. В конструкторе класса Shipment super() вызывает конструктор класса BoxWeight. Super() в BoxWeight вызывает конструктор класса Box. В иерархии классов, если конструктор суперкласса требует наличие параметров, то все подклассы должны передать эти параметры "вверх

по линии". И это не зависит от того, нуждается ли подкласс в своих собственных параметрах.

Когда вызываются конструкторы

Когда иерархия классов создана, в каком порядке вызываются конструкторы классов, образующих иерархию? Например, если имеется подкласс B и суперкласс A, A-конструктор вызывается перед B-конструкторами, или наоборот? Ответ заключается в том, что в иерархии классов конструкторы вызываются в порядке подчиненности классов — от суперкласса к подклассу. Далее, так как super() должен быть первым оператором, выполняемым в конструкторе подкласса, этот порядок всегда одинаков и не зависит от того, используется ли super(). Если super() не используется, будет выполнен конструктор по умолчанию (без параметров) каждого суперкласса. Следующая программе иллюстрирует, когда выполняются конструкторы:

Программа 30. Порядок вызова конструкторов

```
// Демонстрирует порядок вызова конструкторов.
// Создать суперкласс A.
class A {
   A()   {
      System.out.println("Внутри A-конструктора. ") ;
   }
}
// Создать подкласс B расширением A.
class B extends A {
   B() {
      System.out.println("Внутри B-конструктора.");
   }
}
// Создать другой класс  (C), расширяющий B.
class C extends B {
   C() {
      System.out.println("Внутри C-конструктора.");
   }
}
class CallingCons {
   public static void main(String args[]) {
      C c = new C();
   }
}
```

Вывод этой программы:

```
Внутри A-конструктора.
Внутри B-конструктора.
Внутри C-конструктора.
```

Нетрудно видеть, что конструкторы вызываются в порядке подчиненности классов.

Поскольку суперкласс не имеет никакого знания о каком-либо подклассе, любая инициализация, которую ему нужно выполнить, является отдельной и, по возможности, предшествующей любой инициализации, выполняемой подклассом. Поэтому-то она и должна выполняться первой.

1.7. Переопределение методов

В иерархии классов, если метод в подклассе имеет такое же имя и сигнатуру типов (type signature), как метод в его суперклассе, говорят, что метод в подклассе ***переопределяет*** (override) метод в суперклассе. Когда переопределенный метод вызывается в подклассе, он будет всегда обращаться к версии этого метода, определенной подклассом. Версия метода, определенная суперклассом, будет скрыта. Рассмотрим следующий фрагмент:

Программа 31. Переопределение методов

```
// Переопределение методов.
class A {
   int i, j;
   A(int a,  int b) {
      i = a; j = b;
   }
// Показать i и j на экране
   void show(){
      System.out.println("i и j:  " + i + " " + j);
   }
}
class B extends A {
   int k;
   B(int a, int b, int c){
      super(a, b);
      k = c;
   }
// Показать на экране k (этот show(i переопределяет show() из A)
   void show()  {
      System.out.println("k:  " + k);
   }
}
class Override {
   public static void main(String args[]){
      B subOb = new B(1,  2,  3);
      subOb.show();      // Здесь вызывается show() из B
   }
}
```

Вывод этой программы:

```
k:  3
```

Когда show() вызвается для объекта типа B, используется версия show(), определенная в классе B. То есть версия show() внутри B переопределяет (отменяет) версию, объявленную в A.

Если нужно обратиться к версии суперкласса переопределенного метода, то можно сделать это, используя super. Например, в следующей версии подкласс B вызывается show() версия суперкласса A. Это позволяет отобразить все экземплярные переменные.

Программа 32. Вызов переопределенных методов

```
// Класса A как в программе 31
class A {
   int i, j;
   A(int a,  int b) {
      i = a; j = b;
   }
// Показать i и j на экране
   void show(){
      System.out.println("i и j:  " + i + " " + j);
   }
}
// Класс B модифицирован вызовом через super метода show класса A
class B extends A {
   int k;
   B(int a, int b, int c){
      super(a, b);    // Вызов конструктора суперкласса A
      k = c;
   }
// Показать на экране k (этот show() переопределяет show() из A)
   void show(){
      super.show();  // Вызов show() суперкласса A
      System.out.println("k:  " + k);
   }
}

public class OverrideSuper {
   public static void main(String args[]){
      B subOb = new B(1,  2,  3);
      subOb.show();      // Здесь вызывается show() из B

   }
}
```

Программа выводит:

```
i и j:  1 2
k:  3
```

Здесь super.show() вызывает версию show() суперкласса.

Переопределение метода происходит только тогда, когда имена и сигнатуры типов этих двух методов идентичны. Если это не так, то оба

метода просто перегружены. Например, рассмотрим следующую модифицированную версию предыдущего примера:

Программа 33. Перегрузка методов

```
// Методы с различными сигнатурами типов перегружа-ются,
// а не переопределяются.
class A {
   int i, j;
   A(int a, int b){
      i = a;
      j = b;
   }
// Показать i и j
   void show()  {
      System.out.println("i и j: " + i + " " + j);
   }
}
// Создать подкласс B расширением класса A.
class B extends A {
   int k;
   B(int a, int b, int c){
      super(a, b);
      k= c;
   }
   // Перегруженный show()
   void show(String msg){
      System.out.println(msg + k);
   }
}
class Override {
   public static void main(String args[]){
      B subOb = new B(1,  2,  3);
      subOb.show("Это к:  ");  // вызов show()  класса B
      subOb.show();            // вызов show()  класса A
   }
}
```

Вывод этой программы:

```
Это к:  3
i и j: 1 2
```

Версия show() в классе B имеет строчный параметр. Это делает сигнатуру его типов отличной от класса A, который не имеет никаких параметров. Поэтому никакого переопределения (или скрытия имени) нет.

1.8. Динамическая диспетчеризация методов

Методика переопределения методов формирует основу для одной из наиболее мощных концепций Java — *динамической диспет**черизации***

методов. Динамическая диспетчеризация методов это механизм, с помощью которого решение на вызов переопределенной функции принимается во время выполнения, а не во время компиляции. Динамическая диспетчеризация методов важна потому, что таким способом Java реализует полиморфизм времени выполнения.

Ссылочная переменная суперкласса может ссылаться на на объект подкласса. Java использует этот факт, чтобы принимать решения о вызове переопределенных методов во время выполнения. Вот как это делается. Когда переопределенный метод вызывается через ссылку суперкласса, Java определяет, какую версию этого метода следует выполнять, основываясь на *типе объекта,* на который указывает ссылка в момент вызова. Еще раз подчеркнем, что это определение делается *во время выполнения.* Когда ссылка указывает на различные типы объектов, будут вызываться различные версии переопределенного метода. Другими словами, именно *тип объекта, на который сделана ссылка* (а не тип ссылочной переменной) определяет, какая версия переопределенного метода будет выполнена.

Вот пример, который иллюстрирует динамическую диспетчеризацию методов:

Программа 34. Динамическая диспетчеризация методов

```
// Динамическая диспетчеризация методов.
 class A {
   void callme()   {
      System.out.println("Внутри A метод callme");
   }
}
class B extends A {
// переопределить callme ()
   void callme()   {
      System.out.println("Внутри B метод callme");
   }
}
class C extends A {
// переопределить callme ()
   void callme()   {
      System.out.println("Внутри C метод callme");
   }
}
class Dispatch {
   public static void main(String args[]){
      A a = new A();  // объект типа A
      B b = new B();  // объект типа B
      C c = new C();  // объект типа C
      A r;            // определить ссылку типа A
      r = a;          // r указывает на A-объект
      r.callme ();    // вызывает A-версию callme
      r = b;          // r указывает на B-объект
```

```
      r.callme();     // вызывает B-версию callme
      r = c;          // r указывает на C-объект ,
      r.callme();     // вызывает C-версию callme
   }
}
```

Вывод этой программы:

```
Внутри A метод callme
Внутри B метод callme
Внутри C метод callme
```

Эта программа создает один суперкласс с именем A и два его подкласса B и C. Подклассы B и C переопределяют callme(), объявленный в A. Внутри метода main() объявлены объекты типа A, B и C. Объявлена также ссылка типа A с именем r. Затем программа назначает ссылку r на каждый объект и использует эту ссылку, чтобы вызвать соответствующий метод callme(). Как показывает вывод, версия выполняемого caiime() определяется типом объекта, на который указывает ссылка во время вызова. Если бы она была определена типом ссылочной переменной r, мы увидели бы три обращения к методу callme() из класса A.

Переопределенные методы в Java подобны виртуальным функциям в C++.

Зачем нужны переопределенные методы?

Переопределенные методы позволяют поддерживать полиморфизм времени выполнения. Полиморфизм позволяет базовому классу определять методы, которые будут общими для всех его производных классов, и, в то же время, разрешает подклассам определять специфические реализации некоторых или всех таких методов. Переопределяемые методы — это еще один способ реализации аспекта полиморфизма "один интерфейс, множественные методы".

Частичным залогом успешного применения полиморфизма является понимание того, что суперклассы и подклассы формируют иерархию, которая движется от меньшей к большей специализации. Суперкласс обеспечивает все элементы, которые подкласс может использовать непосредственно. Он также определяет те методы, которые производный класс должен реализовать на свой собственный манер. Это позволяет подклассу гибко определять свои методы и одновременно предписывает непротиворечивый интерфейс. Таким образом, комбинируя наследование с переопределением методов, суперкласс может определять общую форму методов, которая будет использоваться всеми его подклассами.

Применение переопределения методов

Рассмотрим более практический пример, который использует переопределение метода. Следующая программа создает суперкласс с именем Figure, который хранит размеры различных двумерных объектов. Он также определяет метод с именем area(), который вычисляет площадь объекта. Программа определяет также два подкласса Figure. Первый — Rectangle, а второй — Triangle. Каждый из этих подклассов переопределяет area() так, чтобы он возвращал площадь прямоугольника и треугольника, соответственно.

Программа 35. Класс плоских фигур

```
// Использование полиморфизма времени выполнения.
class Figure {
   double dim1;
   double dim2;
   Figure(double a, double b){
      Dim1 = a; dim2 = b;
   }
   double area()   {
      System.out.println("Площадь Figure не определена.");
      return 0;
   }
}
class Rectangle extends Figure{
   Rectangle(double a, double b){
      super(a, b);
   }
// переопределить area для прямоугольника
   double area()  {
      System.out.println("Внутри Area для Rectangle.");
      return diml * dim2;
   }
}
class Triangle extends Figure {
   Triangle(double a, double b){
      super(a, b);
   }
// переопределить area для прямоугольного треугольника
   double area(){
      System.out.println("Внутри Area для Triangle.");
      return diml * dim2 / 2;
   }
}
class FigureAreas {
   public static void main(String args[]){
      Figure f    = new Figure(10,  10);
      Rectangle r = new Rectangle(9,  5);
      Triangle t  = new Triangle(10,  8);
      Figure figref;
      figref = r;
```

```
        System.out.println("Площадь равна " + figref.area());
        figref = t;
        System.out.println("Площадь равна " + figref.area ());
        figref = f;
        System.out.println("Площадь равна " + figref.area());
    }
}
```

Вывод этой программы:

```
Внутри Area для Rectangle.
Площадь равна 45.0
Внутри Area для Triangle.
Площадь равна 40.0
Площадь Figure не определена.
Площадь равна 0.0
```

Через двойные механизмы наследования и полиморфизма времени выполнения можно определить один непротиворечивый интерфейс, который используется несколькими различными, но связанными типами объектов. В этом случае, если объект является производным от Figure-объекта, то его площадь может быть получена с помощью вызова метода area(). Интерфейс этой операции один и тот же, независимо от того, какой тип фигуры применяется.

1.9. Использование абстрактных классов

Существуют ситуации, когда нужно определить суперкласс, который объявляет структуру некоторой абстракции без законченной реализации каждого метода. В такой ситуации находится класс Figure из предыдущего примера. Определение area() можно рассматривать в качестве "хранителя места". Метод не вычисляет и не отображает площадь ни для какого объекта.

При создании собственных библиотек классов вовсе не обязательно иметь сколько-нибудь значимое определение методов в контексте их суперклассов. Управлять этой ситуацией можно двумя способами. Во-первых, как показано в предыдущем примере, можно просто выдать предупреждающее сообщение. Хотя этот подход в некоторых ситуациях, таких как отладка, может быть полезен, обычно он неприемлем. Во-вторых, существует возможность определять методы суперкласса, которые должны быть переопределены подклассом для того, чтобы приобрести для указанного подкласса определенное значение. Рассмотрим класс Triangle. В этом случае не имеет никакого значения, определен метод area() или нет. Вы просто хотите иметь некоторый способ, гарантирующий, что подкласс действительно переопределяет все необходимые методы. Решением этой проблемы является *абстрактный метод.*

Можно потребовать, чтобы некоторые методы были переопределены подклассами с помощью модификатора типа abstract. Эти методы иногда называют методами, "отданными на ответственность подклассу" (subclasser responsibility), потому что для них не определено никакой реализации в суперклассе. Таким образом, подкласс обязательно должен переопределить их — он не может просто использовать версию, определенную в суперклассе. Для объявления абстрактного метода используется следующая общая форма:

```
abstract type name(parameter-list) ;
```

Обратите внимание, что тело метода отсутствует.

Любой класс, который содержит один и более абстрактных методов, должен также быть объявлен абстрактным. Чтобы объявить абстрактный класс, используется ключевое слово abstract перед ключевым словом class в начале объявления класса. Нельзя создавать никакие объекты абстрактного класса. То есть для абстрактного класса нельзя прямо создать объект с помощью операции new. Такие объекты были бы бесполезны, потому что абстрактный класс определен не полностью. Вы не можете также объявлять абстрактные конструкторы или абстрактные статические методы. Любой подкласс абстрактного класса должен или реализовать все абстрактные методы суперкласса, или сам должен быть объявлен как abstract.

Далее показан простой пример класса с абстрактным методом, за которым следует класс, реализующий данный метод:

Программа 36. Абстрактный класс

```
// Простая демонстрация абстракций Java.
abstract class A {
   abstract void callme();
   // в абстрактных классах допустимы обычные методы
   void callmetoo(){
      System.out.println("Это конкретный метод.");
   }
}
class B extends A{
   void callme(){
      System.out.println("B – реализация callme.");
   }
}
class AbstractDemo {
   public static void main(String args[]){
      B b = new B();
      b.callme();
      b.callmetoo() ;
   }
}
```

Обратите внимание, что никакие объекты класса A не объявлены в программе. Как сказано выше, нельзя определять экземпляры (объекты) абстрактного класса. Кроме того, класс A реализует конкретный метод с именем callmetoo(). Это вполне допустимо.

Хотя абстрактные классы не могут использоваться для создания объектов, их можно использовать для создания объектных ссылок, потому что подход Java к полиморфизму времени выполнения реализован с помощью ссылок суперкласса. Таким образом, возможно создавать ссылку к абстрактному классу так, чтобы она могла использоваться для указания на объект подкласса. Вы увидите использование этого свойства в следующем примере.

Применяя абстрактный класс, можно улучшать класс Figure, показанный ранее. Так как нет никакого правила для вычисления площади произвольной двумерной фигуры, следующая версия программы объявляет метод area() как абстрактный внутри Figure. Это, конечно, означает, что все классы, производные от Figure, должны переопределить area().

Программа 37. Абстрактный класс Figure

```
// Использование абстрактных методов и классов.
abstract class Figure {
   double dim1;
   double dim2;
   Figure(double a, double b){
      dim1 = a;
      dim2 = b;
   }
   // area теперь абстрактный метод
   abstract double area();
}
class Rectangle extends Figure {
   Rectangle(double a, double b){
      super(a, b);
   }
// переопределить area для прямоугольника
   double area(){
      System.out.println("Внутри Area для Rectangle.");
      return dim1 * dim2;
   }
}
class Triangle extends Figure {
   Triangle(double a, double b){
      super(a, b);
   }
// переопределить area для прямоугольного треугольника
   double area(){
      System.out.println("Внутри Area для Triangle.");
      return dim1 * dim2 / 2;
```

```
    }
}
class AbstractAreas {
    public static void main(String args[]){
        // Figure f = new Figure(10,  10);    // теперь незаконно
        Rectangle r = new Rectangle(9, 5) ;
        Triangle t  = new Triangle(10,  8);
        Figure figref;                         // ОК, объект не создается
        figref = r;
        System.out.println("Площадь равна " + figref.area());
        figref = t;
        System.out.println("Площадь равна " + figref.area());
    }
}
```

Программа выводит:

```
Внутри Area для Rectangle.
Площадь равна 45.0
Внутри Area для Triangle.
Площадь равна 40.0
```

Как указывает комментарий внутри main(), нельзя объявлять объекты типа Figure, так как этот класс теперь абстрактный. Кроме того, все подклассы Figure должны переопределять area(). Без этого будет ошибка времени компиляции.

Хотя невозможно создать объект типа Figure, можно создать ссылочную переменную типа Figure. Переменная figref объявлена как ссылка на Figure, что означает, что она может использоваться для ссылки на объект любого класса, производного от Figure. Напомним, что именно через ссылочные переменные суперкласса выбираются переопределенные методы (во время выполнения).

Использование ключевого слова final с наследованием

Ключевое слово final имеет три применения. Первое — его можно использовать для создания эквивалента именованной константы. Такое использование было описано в предыдущей главе. Два других применения final связаны с наследованием. Оба рассмотрены ниже.

Использование final для отказа от переопределения

Хотя переопределение методов — одно из наиболее мощных свойств Java, может появиться потребность отказаться от него. Чтобы отменить переопределение метода, укажите модификатор final в начале его объявления. Методы, объявленные как final, не могут переопределяться. Следующий фрагмент иллюстрирует final в таком применении:

```
class A {
```

```
final void meth(){
   System.out.println("Это метод final.");
   }
}
class B extends A{
   void meth(){          // ОШИБКА! Нельзя переопределять.
      System, out.println("Ошибка!");
   }
}
```

Поскольку meth() объявлен как final, он не может быть переопределен в классе B. Если вы попытаетесь сделать это, то получите ошибку во время компиляции.

Методы, объявленные как final, могут иногда улучшать эффективность. Компилятору предоставлено право выполнять *встроенные* (inline) вызовы таких методов, потому что он "знает", что те не будут переопределяться подклассом. Когда речь идет о небольшой, например, final-функции, то компилятор Java часто может копировать (встраивать) байт-код подпрограммы прямо в точку вызова откомпилированного кода вызывающего метода, устраняя таким образом дополнительные затраты времени, связанные с обычным вызовом (невстроенного метода). Встраивание допустимо только для final-методов. Обычно Java организует вызовы методов динамически, во время выполнения. Это называется *поздним связыванием* (вызова с вызываемой функцией). Однако, т. к. final-методы не являются переопределяемыми, их вызов может быть организован во время компиляции. Это называется *ранним связыванием.*

Использование final для отмены наследования

Иногда нужно разорвать наследственную связь классов (отменить наследование одного класса другим). Чтобы сделать это, предварите объявление класса ключевым словом final, что позволит неявно объявить и все его методы. Заметим, что недопустимо объявлять класс одновременно как abstract и final, т. к. абстрактный класс неполон сам по себе и полагается на свои подклассы, чтобы обеспечить полную реализацию. Пример final-класса:

```
final class A {
// ...
}
// Следующий класс незаконный.
class B extends A { // ошибка! B не может быть подклассом A
// . . .
}
```

Комментарий здесь означает, что B не может наследовать A, т. к. A объявлен как final.

1.10. Класс Object

В Java определен один специальный класс — Object. Все другие классы являются его подклассами, Object — это суперкласс всех других классов. Это означает, что ссылочная переменная типа object может обращаться к объекту любого другого класса. Кроме того, т. к. массивы реализуются как классы, переменная типа object может также обращаться к любому массиву.

В классе Object определены методы (табл.8), которые доступны в каждом объекте.

Таблица 8. Методы класса Object

Метод	Цель
Object clone ()	Создает новый объект, который является таким же, как имитируемый объект
boolean equals (Object object)	Определяет, является ли один объект равным другому
void finalize ()	Вызывается прежде, чем неиспользованный объект будет переработан (сборщиком мусора)
Class getclass ()	Получает класс объекта во время выполнения
int hashCode()	Возвращает хэш-код, связанный с вызовом объекта
void notify()	Возобновляет выполнение потока, ожидающего на объекте вызова
void notifyAll()	Возобновляет выполнение всех потоков, ожидающих на объекте вызова
String toString()	Возвращает строку, которая описывает объект
void wait () void wait (long millisrconds) void wait(long millisrconds, int nanoseconds)	Ждет выполнения на другом потоке

Методы getClass(), notify(), notifyAll() и wait() объявлены как final.

Другие можно переопределять. Здесь отметим два метода: equals() и toString(). Метод equals() сравнивает содержимое двух объектов. Он возвращает true, если объекты эквивалентны, и false — в противном случае. Метод ToString() возвращает строку, содержащую описание объекта, на котором он вызывается. Кроме того, этот метод вызывается автоматически, когда объект выводится методом println(). Много классов переопределяют данный метод, что позволяет им приспосабливать описание специально для типов объектов, которые они создают