```
public class Balance {
    String name;
    double bal;

    public Balance(String n, double b) {
        name = n;
        bal = b;
    }

    public void show() {
        if(bal<0)
        System.out.print("--> ");
        System.out.println(name + ": $" + bal);
    }
}
```

Как видите, класс `Balance` теперь объявлен как `public`. Его конструктор и метод `show()` также объявлены как `public`. Это означает, что они доступны для любого кода за пределами пакета `MyPack`. Например, класс `TestBalance` импортирует пакет `MyPack`, и поэтому в нем может быть использован класс `Balance`.

```
import MyPack.*;

class TestBalance {
    public static void main(String args[]) {
        /* Класс Balance объявлен как public, поэтому им можно
           воспользоваться и вызвать его конструктор. */
        Balance test = new Balance("J. J. Jaspers", 99.88);

        test.show(); // можно также вызвать метод show()
    }
}
```

В качестве эксперимента удалите модификатор `public` из объявления класса `Balance`, а затем попытайтесь скомпилировать класс `TestBalance`. Как упоминалось выше, это приведет к появлению ошибок.

Интерфейсы

С помощью ключевого слова `interface` можно полностью абстрагировать интерфейс класса от его реализации. Это означает, что с помощью ключевого слова `interface` можно указать, *что* именно должен выполнять класс, но не *как* это делать. Синтаксически интерфейсы аналогичны классам, но не содержат переменные экземпляра, а объявления их методов, как правило, не содержат тело метода. На практике это означает, что можно объявлять интерфейсы, которые не делают никаких допущений относительно их реализации. Как только интерфейс определен, его может реализовать любое количество классов. Кроме того, один класс может реализовать любое количество интерфейсов.

Чтобы реализовать интерфейс, в классе должен быть создан полный набор методов, определенных в этом интерфейсе. Но в каждом классе могут быть определены особенности собственной реализации данного интерфейса. Ключевое слово `interface` позволяет в полной мере использовать принцип полиморфизма "один интерфейс, несколько методов".

Интерфейсы предназначены для поддержки динамического разрешения вызовов методов во время выполнения. Как правило, для нормального выполнения вызова метода из одного класса в другом оба класса должны присутствовать во время компиляции, чтобы компилятор Java мог проверить совместимость сигнатур методов. Само по себе это требование создает статическую и нерасширяемую среду распределения классов. В такой системе функциональные возможности неизбежно передаются вверх по иерархии классов, в результате чего механизмы будут становиться доступными все большему количеству подклассов. Интерфейсы предназначены для предотвращения этой проблемы. Они изолируют определение метода или набора методов от иерархии наследования. А поскольку иерархия интерфейсов не совпадает с иерархией классов, то классы, никак не связанные между собой иерархически, могут реализовать один и тот же интерфейс. Именно в этом возможности интерфейсов проявляются наиболее полно.

Объявление интерфейса

Во многом определение интерфейса подобно определению класса. Упрощенная общая форма интерфейса имеет следующий вид:

```
доступ interface имя {
    возвращаемый_тип имя_метода1(список_параметров);
    возвращаемый_тип имя_метода2(список_параметров);
    тип имя_завершенной_переменной1 = значение;
    тип имя_завершенной_переменной2 = значение;
// ...
    возвращаемый_тип имя_методаN(список_параметров);
    тип имя_завершенной_переменнойN = значение;
}
```

Если определение не содержит никакого модификатора доступа, используется доступ по умолчанию, а интерфейс доступен только другим членам того пакета, в котором он объявлен. Если интерфейс объявлен как `public`, он может быть использован в любом другом коде. В этом случае интерфейс должен быть единственным открытым интерфейсом, объявленным в файле, а имя этого файла должно совпадать с именем интерфейса. В приведенной выше общей форме указанное *имя* обозначает конкретное имя интерфейса, которым может быть любой допустимый идентификатор. Обратите внимание на то, что объявляемые методы не содержат тел. Их объявления завершаются списком параметров, за которым следует точка с запятой. По существу, они являются абстрактными методами. Каждый класс, который включает в себя интерфейс, должен реализовать все его методы.

Прежде чем продолжить дальше, важно отметить, что в версии JDK 8 ключевое слово `interface` дополнено средством, значительно изменяющим его возможности. До версии JDK 8 в интерфейсе вообще нельзя было ничего реализовать. В интерфейсе, упрощенная форма объявления которого приведена выше, наличие тела в объявляемых методах не предполагалось. Следовательно, до версии JDK 8 в интерфейсе можно было определить только то, *что* именно следует сделать, но не *как* это сделать. Это положение изменилось в версии JDK 8. Теперь метод можно объявлять в интерфейсе с *реализацией по умолчанию*, т.е. указать его поведение. Но методы по умолчанию, по существу, служат специальной цели, сохраняя в то же

время исходное назначение интерфейса. Следовательно, интерфейсы можно по-прежнему создавать и использовать и без методов по умолчанию. Именно по этой причине обсуждение интерфейсов будет начато с их традиционной формы. А методы по умолчанию описываются в конце этой главы.

Как следует из приведенной выше общей формы, в объявлениях интерфейсов могут быть объявлены переменные. Они неявно объявляются как `final` и `static`, т.е. их нельзя изменить в классе, реализующем интерфейс. Кроме того, они должны быть инициализированы. Все методы и переменные неявно объявляются в интерфейсе как `public`.

Ниже приведен пример объявления интерфейса. В нем объявляется простой интерфейс, который содержит один метод `callback()`, принимающий единственный целочисленный параметр.

```
interface Callback {
    void callback(int param);
}
```

Реализация интерфейсов

Как только интерфейс определен, он может быть реализован в одном или нескольких классах. Чтобы реализовать интерфейс, в определение класса требуется включить выражение `implements`, а затем создать методы, определенные в интерфейсе. Общая форма класса, который содержит выражение `implements`, имеет следующий вид:

```
доступ class имя_класса [extends суперкласс]
        [implements интерфейс [, интерфейс...]] {
    // тело класса
}
```

Если в классе реализуется больше одного интерфейса, имена интерфейсов разделяются запятыми. Так, если в классе реализуются два интерфейса, в которых объявляется один и тот же метод, то этот же метод будет использоваться клиентами любого из двух интерфейсов. Методы, реализующие элементы интерфейса, должны быть объявлены как `public`. Кроме того, сигнатура типа реализующего метода должна в точности совпадать с сигнатурой типа, указанной в определении `interface`.

Рассмотрим небольшой пример класса, где реализуется приведенный ранее интерфейс `Callback`. Обратите внимание на то, что метод `callback()` объявлен с модификатором доступа `public`.

```
class Client implements Callback {
    // реализовать интерфейс Callback
    public void callback(int p) {

        System.out.println(
               "Метод callback(), вызываемый со значением " + p);
    }
}
```

Помните! Когда реализуется метод из интерфейса, он должен быть объявлен как `public`.

Вполне допустима и достаточно распространена ситуация, когда в классах, реализующих интерфейсы, определяются также собственные члены. Например, в следующей версии класса `Client` реализуется метод `callback()` и добавляется метод `nonIfaceMeth()`:

```
class Client implements Callback {
    // реализовать интерфейс Callback
    public void callback(int p) {
        System.out.println(
                "Метод callback(), вызываемый со значением " + p);
    }

    void nonIfaceMeth() {
        System.out.println("В классах, реализующих интерфейсы," +
                           "могут определяться и другие члены.");
    }
}
```

Доступ к реализациям через ссылки на интерфейсы

Переменные можно объявлять как ссылки на объекты, в которых используется тип интерфейса, а не тип класса. С помощью такой переменной можно ссылаться на любой экземпляр какого угодно класса, реализующего объявленный интерфейс. При вызове метода по одной из таких ссылок нужный вариант будет выбираться в зависимости от конкретного экземпляра интерфейса, на который делается ссылка. И это одна из главных особенностей интерфейсов. Поиск исполняемого метода осуществляется динамически во время выполнения, что позволяет создавать классы позднее, чем код, из которого вызываются методы этих классов. Вызывающий код может выполнять диспетчеризацию методов с помощью интерфейса, даже не имея никаких сведений о вызываемом коде. Этот процесс аналогичен использованию ссылки на суперкласс для доступа к объекту подкласса (см. главу 8).

Внимание! Поскольку в Java динамический поиск методов во время выполнения сопряжен со значительными издержками по сравнению с обычным вызовом методов, в прикладном коде, критичном к производительности, интерфейсы следует использовать только тогда, когда это действительно необходимо.

В следующем примере программы метод `callback()` вызывается через переменную ссылки на интерфейс:

```
class TestIface {
    public static void main(String args[]) {
        Callback c = new Client();
        c.callback(42);
    }
}
```

Эта программа выводит следующий результат:

```
Метод callback(), вызываемый со значением 42
```

Обратите внимание на то, переменной `c` присвоен экземпляр класса `Client`, несмотря на то, что она объявлена с типом интерфейса `Callback`. Переменную `c` можно использовать для доступа к методу `callback()`, она не предоставляет до-

ступа к каким-нибудь другим членам класса `Client`. Переменная ссылки на интерфейс располагает только сведениями о методах, объявленных в том интерфейсе, на который она ссылается. Таким образом, переменной `c` нельзя пользоваться для доступа к методу `nonIfaceMeth()`, поскольку этот метод объявлен в классе `Client`, а не в интерфейсе `Callback`.

Приведенный выше пример формально показывает, каким образом переменная ссылки на интерфейс может получать доступ к объекту его реализации, тем не менее, он не демонстрирует полиморфные возможности такой ссылки. Чтобы продемонстрировать такие возможности, создадим сначала вторую реализацию интерфейса `Callback`, как показано ниже.

```
// Еще одна реализация интерфейса Callback
class AnotherClient implements Callback {
    // реализовать интерфейс Callback
    public void callback(int p) {
        System.out.println("Еще один вариант метода callback()");
        System.out.println("p в квадрате равно " + (p*p));
    }
}
```

Теперь попробуем создать и опробовать следующий класс:

```
class TestIface2 {
    public static void main(String args[]) {
        Callback c = new Client();
        AnotherClient ob = new AnotherClient();

        c.callback(42);

        c = ob; // теперь переменная c ссылается на
                // объект типа AnotherClient
        c.callback(42);
    }
}
```

Эта программа выводит следующий результат:

```
Метод callback(), вызываемый со значением 42
Еще один вариант метода callback()
p в квадрате равно 1764
```

Как видите, вызываемый вариант метода `callback()` выбирается в зависимости от типа объекта, на который переменная `c` ссылается во время выполнения. Приведенный выше пример довольно прост, поэтому далее будет рассмотрен еще один, более практический пример.

Частичные реализации

Если класс включает в себя интерфейс, но не полностью реализует определенные в нем методы, он должен быть объявлен как `abstract`:

```
abstract class Incomplete implements Callback {
    int a, b;

    void show() {
        System.out.println(a + " " + b);
    }
```

```
    // ...
}
```

В данном примере кода класс `Incomplete` не реализует метод `callback()`, поэтому он должен быть объявлен как абстрактный. Любой класс, наследующий от класса `Incomplete`, должен реализовать метод `callback()` или быть также объявленным как `abstract`.

Вложенные интерфейсы

Интерфейс может быть объявлен членом класса или другого интерфейса. Такой интерфейс называется *интерфейсом-членом* или *вложенным интерфейсом*. Вложенный интерфейс может быть объявлен как `public`, `private` или `protected`. Этим он отличается от интерфейса верхнего уровня, который должен быть объявлен как `public` или использовать уровень доступа по умолчанию, как отмечалось ранее. Когда вложенный интерфейс используется за пределами объемлющей его области действия, его имя должно быть дополнительно уточнено именем класса или интерфейса, членом которого он является. Это означает, что за пределами класса или интерфейса, в котором объявлен вложенный интерфейс, его имя должно быть уточнено полностью.

В следующем примере демонстрируется применение вложенного интерфейса:

```
// Пример вложенного интерфейса

// Этот класс содержит интерфейс как свой член
class A {
    // это вложенный интерфейс
    public interface NestedIF {
        boolean isNotNegative(int x);
    }
}

// Класс B реализует вложенный интерфейс
class B implements A.NestedIF {
    public boolean isNotNegative(int x) {
        return x < 0 ? false : true;
    }
}

class NestedIFDemo {
    public static void main(String args[]) {

    // использовать ссылку на вложенный интерфейс
    A.NestedIF nif = new B();

    if(nif.isNotNegative(10))
        System.out.println("Число 10 неотрицательное");
    if(nif.isNotNegative(-12))
        System.out.println("Это не будет выведено");
    }
}
```

Обратите внимание на то, что в классе `A` определяется вложенный интерфейс `NestedIF`, объявленный как `public`. Затем вложенный интерфейс реализуется в классе `B` следующим образом:

```
implements A.NestedIF
```

Обратите также внимание на то, что имя интерфейса полностью уточнено и содержит имя класса. В теле метода `main()` создается переменная `nif` ссылки на интерфейс `A.NestedIF`, которой присваивается ссылка на объект класса `B`. И это вполне допустимо, поскольку класс `B` реализует интерфейс `A.NestedIF`.

Применение интерфейсов

Чтобы стали понятнее возможности интерфейсов, рассмотрим более практический пример. В предыдущих главах был разработан класс `Stack`, реализующий простой стек фиксированного размера. Но стек можно реализовать самыми разными способами. Например, стек может иметь фиксированный или "наращиваемый" размер. Стек может также храниться в массиве, связанном списке, двоичном дереве и т.п. Независимо от реализации стека, его интерфейс остается неизменным. Это означает, что методы `push()` и `pop()` определяют интерфейс стека независимо от особенностей его реализации. А поскольку интерфейс стека отделен от его реализации, то такой интерфейс можно определить без особого труда, оставив уточнение конкретных деталей в его реализации. Рассмотрим два примера применения интерфейса стека.

Создадим сначала интерфейс, определяющий целочисленный стек, разместив его в файле `IntStack.java`. Этот интерфейс будет использоваться в обеих реализациях стека.

```
// Определить интерфейс для целочисленного стека
interface IntStack {
    void push(int item); // сохранить элемент в стеке
    int pop();           // извлечь элемент из стека
}
```

В приведенной ниже программе создается класс `FixedStack`, реализующий версию целочисленного стека фиксированной длины.

```
// Реализация интерфейса IntStack для стека фиксированного размера
class FixedStack implements IntStack {
    private int stck[];
    private int tos;
    // выделить память и инициализировать стек
    FixedStack(int size) {
        stck = new int[size];
        tos = -1;
    }

    // разместить элемент в стеке
    public void push(int item) {
        if(tos==stck.length-1) // использовать поле длины стека
            System.out.println("Стек заполнен.");
        else
        stck[++tos] = item;
    }

    // извлечь элемент из стека
    public int pop() {
        if(tos < 0) {
            System.out.println("Стек не загружен.");
            return 0;
```

```
        }
        else
            return stck[tos--];
    }
}

class IFTest {
public static void main(String args[]) {
    FixedStack mystack1 = new FixedStack(5);
    FixedStack mystack2 = new FixedStack(8);

    // разместить числа в стеке
    for(int i=0; i<5; i++) mystack1.push(i);
    for(int i=0; i<8; i++) mystack2.push(i);

    // извлечь эти числа 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());
    }
}
```

Ниже приведена еще одна реализация интерфейса `IntStack`, в которой с помощью того же самого определения `interface` создается динамический стек. В этой реализации каждый стек создается с первоначальной длиной. При превышении этой начальной длины размер стека увеличивается. Каждый раз, когда возникает потребность в дополнительном свободном месте, размер стека удваивается.

```
// Реализация "наращиваемого" стека
class DynStack implements IntStack {
    private int stck[];
    private int tos;

    // выделить память и инициализировать стек
    DynStack(int size) {
        stck = new int[size];
        tos = -1;
    }

    // разместить элемент в стеке
    public void push(int item) {
        // если стек заполнен, выделить память под стек большего размера
        if(tos==stck.length-1) {
            int temp[] = new int[stck.length * 2]; // удвоить размер стека
            for(int i=0; i<stck.length; i++) temp[i] = stck[i];
            stck = temp;
            stck[++tos] = item;
        }
        else
            stck[++tos] = item;
    }

    // извлечь элемент из стека
    public int pop() {
        if(tos < 0) {
            System.out.println("Стек не загружен.");
```

```
            return 0;
        }
        else
            return stck[tos--];
    }
}

class IFTest2 {
    public static void main(String args[]) {
        DynStack mystack1 = new DynStack(5);
        DynStack mystack2 = new DynStack(8);

        // В этих циклах увеличиваются размеры каждого стека
        for(int i=0; i<12; i++) mystack1.push(i);
        for(int i=0; i<20; i++) mystack2.push(i);

        System.out.println("Стек в mystack1:");
        for(int i=0; i<12; i++)
            System.out.println(mystack1.pop());

        System.out.println("Стек в mystack2:");
        for(int i=0; i<20; i++)
            System.out.println(mystack2.pop());
    }
}
```

В приведенном ниже примере программы создается класс, в котором используются обе реализации данного интерфейса в классах `FixedStack` и `DynStack`. Для этого применяется ссылка на интерфейс. Это означает, что поиск вариантов при вызове методов `push()` и `pop()` осуществляется во время выполнения, а не во время компиляции.

```
/* Создать переменную интерфейса и
   обратиться к стекам через нее.
*/
class IFTest3 {
    public static void main(String args[]) {
        IntStack mystack; // создать переменную ссылки на интерфейс
        DynStack ds = new DynStack(5);
        FixedStack fs = new FixedStack(8);

        mystack = ds;     // загрузить динамический стек
        // разместить числа в стеке
        for(int i=0; i<12; i++) mystack.push(i);

        mystack = fs;     // загрузить фиксированный стек
        for(int i=0; i<8; i++) mystack.push(i);

        mystack = ds;
        System.out.println("Значения в динамическом стеке:");
        for(int i=0; i<12; i++)
            System.out.println(mystack.pop());

        mystack = fs;
        System.out.println("Значения в фиксированном стеке:");
        for(int i=0; i<8; i++)
            System.out.println(mystack.pop());
    }
}
```

В этой программе переменная mystack содержит ссылку на интерфейс IntStack. Следовательно, когда она ссылается на переменную ds, выбираются варианты методов push() и pop(), определенные при реализации данного интерфейса в классе DynStack. Когда же она ссылается на переменную fs, выбираются варианты методов push() и pop(), определенные при реализации данного интерфейса в классе FixedStack. Как отмечалось ранее, все эти решения принимаются во время выполнения. Обращение к нескольким реализациям интерфейса через ссылочную переменную интерфейса является наиболее эффективным средством в Java для поддержки полиморфизма во время выполнения.

Переменные в интерфейсах

Интерфейсы можно применять для импорта совместно используемых констант в несколько классов путем простого объявления интерфейса, который содержит переменные, инициализированные нужными значениями. Когда интерфейс включается в класс (т.е. реализуется в нем), имена всех этих переменных оказываются в области действия констант. (Это аналогично использованию в программе на C/C++ заголовочного файла для создания большого количества констант с помощью директив #define или объявлений const.) Если интерфейс не содержит никаких методов, любой класс, включающий в себя такой интерфейс, на самом деле ничего не реализует. Это все равно, как если бы класс импортировал постоянные поля в пространство имен класса в качестве завершенных переменных. В следующем примере программы эта методика применяется для реализации автоматизированной системы "принятия решений".

```
import java.util.Random;

interface SharedConstants {
    int NO = 0;
    int YES = 1;
    int MAYBE = 2;
    int LATER = 3;
    int SOON = 4;
    int NEVER = 5;
}

class Question implements SharedConstants {
    Random rand = new Random();
    int ask() {
        int prob = (int) (100 * rand.nextDouble());
        if (prob < 30)
            return NO;       // 30%
        else if (prob < 60)
            return YES;      // 30%
        else if (prob < 75)
            return LATER;    // 15%
        else if (prob < 98)
            return SOON;     // 13%
        else
            return NEVER;    // 2%
    }
}
```

```
class AskMe implements SharedConstants {
    static void answer(int result) {
        switch(result) {
            case NO:
                System.out.println("Нет");
                break;
            case YES:
                System.out.println("Да");
                break;
            case MAYBE:
                System.out.println("Возможно");
                break;
            case LATER:
                System.out.println("Позднее");
                break;
            case SOON:
                System.out.println("Вскоре");
                break;
            case NEVER:
                System.out.println("Никогда");
                break;
        }
    }

    public static void main(String args[]) {
        Question q = new Question();

        answer(q.ask());
        answer(q.ask());
        answer(q.ask());
        answer(q.ask());
    }
}
```

Обратите внимание на то, что в этой программе используется класс `Random` из стандартной библиотеки Java. В этом классе создаются псевдослучайные числа. Он содержит несколько методов, которые позволяют получать случайные числа в требуемой для программы форме. В данном примере применяется метод `nextDouble()`, возвращающий случайные числа в пределах от **0,0** до **1,0**.

В рассматриваемом здесь примере программы два класса, `Question` и `AskMe`, реализуют интерфейс `SharedConstants`, в котором определены константы `NO` (Нет), `YES` (Да), `MAYBE` (Возможно), `SOON` (Вскоре), `LATER` (Позднее) и `NEVER` (Никогда). Код из каждого класса ссылается на эти константы так, как если бы они определялись и наследовались непосредственно в каждом классе. Ниже приведен результат, выводимый при выполнении данной программы. Обратите внимание на то, что при каждом запуске программы результаты ее выполнения оказываются разными.

```
Позднее
Вскоре
Нет
Да
```

На заметку! Упомянутая выше методика применения интерфейса для определения общих констант весьма противоречива и представлена ради полноты изложения материала.