```
/* пример # 6 : поведение статического метода при «переопределении»:
Runner.java */
package chapt04;

class Base {
      public static void assign() {
            System.out.println(
                  "метод assign() из Base");
      }
}
class Sub extends Base {
      public static void assign() {
            System.out.println(
                  "метод assign() из Sub");
      }
}
public class Runner {
      public static void main(String[] args) {
            Base ob1 = new Base();
            Base ob2 = new Sub();
            Sub ob3 = new Sub();
            ob1.assign();//некорректный вызов статичесого метода
            ob2.assign();//следует вызывать Base.assign();
            ob3.assign();
      }
}
```

В результате выполнения данного кода будет выведено:

```
метод assign() из Base
метод assign() из Base
метод assign() из Sub
```

При таком способе инициализации объектов **ob1** и **ob2**, метод **assign()** будет вызван из класса **Base**. Для объекта **ob3** будет вызван собственный метод **assign()**, что следует из способа объявления объекта. Если же спецификатор **static** убрать из объявления методов, то вызовы методов будут осуществляться в соответствии с принципами полиморфизма.

Статические методы всегда следует вызывать через имя класса, в котором они объявлены, а именно:

```
Base.assign();
Sub.assign();
```

Вызов статических методов через объект считается нетипичным и нарушающим смысл статического определения.

Абстракция и абстрактные классы

Множество предметов реального мира обладает некоторым набором общих характеристик и правил поведения. Абстрактное понятие «Геометрическая фигура» может содержать описание геометрических параметров и расположения центра тяжести в системе координат, а также возможности определения площади и

периметра фигуры. Однако в общем случае дать конкретную реализацию приведенных характеристик и функциональности невозможно ввиду слишком общего их определения. Для конкретного понятия, например «Квадрат», дать описание линейных размеров и определения площади и периметра не составляет труда. Абстрагирование понятия должно предоставлять абстрактные характеристики предмета реального мира, а не его ожидаемую реализацию. Грамотное выделение абстракций позволяет структурировать код программной системы в целом и повторно использовать абстрактные понятия для конкретных реализаций при определении новых возможностей абстрактной сущности.

Абстрактные классы объявляются с ключевым словом **abstract** и содержат объявления абстрактных методов, которые не реализованы в этих классах, а будут реализованы в подклассах. Объекты таких классов создать нельзя, но можно создать объекты подклассов, которые реализуют эти методы. При этом допустимо объявлять ссылку на абстрактный класс, но инициализировать ее можно только объектом производного от него класса. Абстрактные классы могут содержать и полностью реализованные методы, а также конструкторы и поля данных.

С помощью абстрактного класса объявляется контракт (требования к функциональности) для его подклассов. Примером может служить уже рассмотренный выше абстрактный класс **Number** и его подклассы **Byte**, **Float** и другие. Класс **Number** объявляет контракт на реализацию ряда методов по преобразованию данных к значению конкретного базового типа, например **floatValue()**. Можно предположить, что реализация метода будет различной для каждого из классов-оболочек. Хотя объект класса **Number** нельзя создать, он может получить численное значение любого базового типа. Однако у самого класса нет возможности преобразовать это значение к конкретному базовому типу.

```
/* пример # 7 : абстрактный класс и метод : AbstractManager.java */
package chapt04;

public abstract class AbstractManager {
      private int id;
      public AbstractManager(int id) {// конструктор
            this.id = id;
      }
      // абстрактный метод
      public abstract void assignGroupToCourse(
                  int groupId, String nameCourse);
}
```

```
/* пример # 8 : подкласс абстрактного класса : CourseManager.java */
package chapt04;


// assignGroupToCourse() должен быть реализован в подклассе
public class CourseManager extends AbstractManager {
      public void assignGroupToCourse(
                  int groupId, String nameCourse) {
            //...
```

```
            System.out.println("группа " + groupId
                  + " назначена на курс " + nameCourse);
      }
}
```

```
/* пример # 9 : объявление объектов и вызов методов : Runner.java */
package chapt04;

public class Runner {
      public static void main(String[] args) {
            AbstractManager mng; // можно объявить ссылку
          // mng = new AbstractManager();нельзя создать объект!
            mng = new CourseManager();
            mng.assignGroupToCourse(10, "Алгебра");
      }
}
```

В результате будет получено:

```
группа 10 назначена на курс Алгебра
```

Ссылка на абстрактный суперкласс **mng** инициализируется объектом подкласса, в котором реализованы все абстрактные методы суперкласса. С помощью этой ссылки могут вызываться реализованные методы абстрактного класса, если они не переопределены в подклассе.

Класс Object

На вершине иерархии классов находится класс **Object**, который является суперклассом для всех классов. Ссылочная переменная типа **Object** может указывать на объект любого другого класса, на любой массив, так как массивы реализуются как классы. В классе **Object** определен набор методов, который наследуется всеми классами:

protected Object clone() – создает и возвращает копию вызывающего объекта;

boolean equals(Object ob) – предназначен для переопределения в подклассах с выполнением общих соглашений о сравнении содержимого двух объектов;

Class<? extends Object> getClass() – возвращает объект типа **Class**;

protected void finalize() – вызывается перед уничтожением объекта автоматическим сборщиком мусора (garbage collection);

int hashCode() – возвращает хэш-код объекта;

String toString() – возвращает представление объекта в виде строки.

Методы **notify()**, **notifyAll()** и **wait()** будут рассмотрены в главе «Потоки выполнения».

Если при создании класса предполагается проверка логической эквивалентности объектов, которая не выполнена в суперклассе, следует переопределить два метода: **equals(Object ob)** и **hashCode()**. Кроме того, переопределе-