Глава 4

НАСЛЕДОВАНИЕ И ПОЛИМОРФИЗМ

Наследование

Отношение между классами, при котором характеристики одного класса (суперкласса) передаются другому классу (подклассу) без их повторного описания, называется ***наследованием***.

Подкласс наследует переменные и методы суперкласса, используя ключевое слово **extends**. Класс может также реализовывать любое число интерфейсов, используя ключевое слово – **implements**. Подкласс имеет прямой доступ ко всем открытым переменным и методам родительского класса, как будто они находятся в подклассе. Исключение составляют члены класса, помеченные **private** (во всех случаях) и «по умолчанию» для подкласса в другом пакете. В любом случае (даже если ключевое слово **extends** отсутствует) класс автоматически наследует свойства суперкласса всех классов – класса **Object**.

Множественное наследование классов запрещено, хотя его аналог предоставляет реализация интерфейсов, которые не являются классами и содержат описание набора методов, позволяющих задать поведение объекта класса, реализующего эти интерфейсы. Наличие общих методов, которые должны быть реализованы в разных классах, обеспечивают им сходную функциональность.

Подкласс дополняет члены базового класса своими переменными и методами, имена которых могут частично совпадать с именами членов суперкласса. Если имена методов совпадают, а параметры различаются, то такое явление называется перегрузкой методов.

В подклассе можно объявить (переопределить) метод с тем же именем, списком параметров и возвращаемым значением, что и у метода суперкласса.

Способность ссылки динамически определять версию переопределенного метода в зависимости от переданного ссылке в сообщении типа объекта называется ***полиморфизмом***.

Полиморфизм является основой для реализации механизма динамического или «позднего связывания».

В следующем примере переопределяемый метод **typeEmployee()** находится в двух классах **Employee** и **Manager**. В соответствии с принципом полиморфизма вызывается метод, наиболее близкий к текущему объекту.

```
/* пример # 1 : наследование класса и переопределение метода:
Employee.java: Manager.java:  Runner.java */
package chapt04;

public class Employee {// рядовой сотрудник
      private int id;
```

```
	public Employee(int idc) {
		super();/* по умолчанию, необязательный явный вызов
				конструктора суперкласса */
		id = idc;
	}
	public int getId() {
		return id;
	}
	public void typeEmployee() {
		//…
		System.out.println("Работник");
	}
}

package chapt04;
// сотрудник с проектом, за который он отвечает

public class Manager extends Employee {
	private int idProject;

	public Manager(int idc, int idp) {
		super(idc); /* вызов конструктора суперкласса
				с параметром */
		idProject = idp;
	}
	public int getIdProject() {
		return idProject;
	}
	public void typeEmployee() {
		//…
		System.out.println("Менеджер");
	}
}
package chapt04;

public class Runner {
	public static void main(String[] args) {
		Employee b1 = new Employee(7110);
		Employee b2 = new Manager(9251, 31);
		b1.typeEmployee();// вызов версии из класса Employee
		b2.typeEmployee();// вызов версии из класса Manager
		// b2.getIdProject();// ошибка компиляции!!!
		((Manager) b2).getIdProject();
		Manager b3 = new Manager(9711, 35);
		System.out.println(b3.getIdProject());// 35
		System.out.println(b3.getId());// 9711
	}
}
```

Объект **b1** создается при помощи вызова конструктора класса **Employee**, и, соответственно, при вызове метода **typeEmployee()** вызывается версия метода из класса **Employee**. При создании объекта **b2** ссылка типа **Employee** инициализируется объектом типа **Manager**. При таком способе инициализации ссылка на суперкласс получает доступ к методам, переопределенным в подклассе.

При объявлении совпадающих по сигнатуре (имя, тип, область видимости) полей в суперклассе и подклассах их значения не переопределяются и никак не пересекаются, то есть существуют в одном объекте независимо друг от друга. В этом случае задача извлечения требуемого значения определенного поля, принадлежащего классу в цепочке наследования, ложится на программиста. Для доступа к полям текущего объекта можно использовать указатель **this**, для доступа к полям суперкласса – указатель **super**. Другие возможности рассмотрены в следующем примере:

```
/* пример # 2 : создание объекта подкласса и доступ к полям с одинаковыми именами: Course.java: BaseCourse.java: Logic.java */
package chapt04;

public class Course {
      public int id = 71;

      public Course() {
       System.out.println("конструктор класса Course");
       id = getId();//!!!
       System.out.println(" id=" + id);
      }
      public int getId() {
            System.out.println("getId() класса Course");
            return id;
      }
}
package chapt04;

public class BaseCourse extends Course {
      public int id = 90;// так делать не следует!

      public BaseCourse() {
       System.out.println("конструктор класса BaseCourse");
       System.out.println(" id=" + getId());
      }
      public int getId() {
          System.out.println("getId() класса BaseCourse");
          return id;
      }
}
package chapt04;

public class Logic {
```

```
      public static void main(String[] args) {
            Course objA = new BaseCourse();
            BaseCourse objB = new BaseCourse();
            System.out.println("objA: id=" + objA.id);
            System.out.println("objB: id=" + objB.id);
            Course objC = new Course();
      }
}
```

В результате выполнения данного кода последовательно будет выведено:

```
конструктор класса Course
getId() класса BaseCourse
 id=0
конструктор класса BaseCourse
getId() класса BaseCourse
 id=90
конструктор класса Course
getId() класса BaseCourse
 id=0
конструктор класса BaseCourse
getId() класса BaseCourse
 id=90
objA: id=0
objB: id=90
конструктор класса Course
getId() класса Course
 id=71
```

Метод **getId()** содержится как в классе **Course**, так и в классе **BaseCourse** и является переопределенным. При создании объекта класса **BaseCourse** одним из способов:

```
Course objA = new BaseCourse();
BaseCourse objB = new BaseCourse();
```

в любом случае перед вызовом конструктора **BaseCourse()** вызывается конструктор класса **Course**. Но так как в обоих случаях создается объект класса **BaseCourse**, то вызывается метод **getId()**, объявленный в классе **BaseCourse**, который в свою очередь оперирует полем **id**, еще не проинициализированным для класса **BaseCourse**. В результате **id** получит значение по умолчанию, т.е. нуль.

Воспользовавшись преобразованием типов вида **((BaseCourse)objA).id** или **((Course)objB).id**, легко можно получить доступ к полю **id** из соответствующего класса.

Использование final

Нельзя создать подкласс для класса, объявленного со спецификатором **final**:

```
// класс ConstCourse не может быть суперклассом
final class ConstCourse {/*код*/}
```

```
// следующий класс невозможен
class BaseCourse extends ConstCourse {/*код*/}
```

Использование super и this

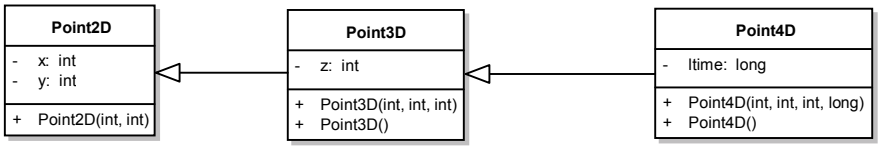
Ключевое слово **super** используется для вызова конструктора суперкласса и для доступа к члену суперкласса. Например:

```
super(список_параметров);/* вызов конструктора суперкласса
                            с передачей параметров или без нее*/
super.id = 71; /* обращение к атрибуту суперкласса */
super.getId(); // вызов метода суперкласса
```

Вторая форма **super** используется для доступа из подкласса к переменной **id** суперкласса. Третья форма специфична для Java и обеспечивает вызов из подкласса переопределенного метода суперкласса, причем если в суперклассе этот метод не определен, то будет осуществляться поиск по цепочке наследования до тех пор, пока метод не будет найден.

Каждый экземпляр класса имеет неявную ссылку **this** на себя, которая передается также и методам. После этого метод «знает», какой объект его вызвал. Вместо обращения к атрибуту **id** в методах можно писать **this.id**, хотя и не обязательно, так как записи **id** и **this.id** равносильны.

Следующий код показывает, как, используя **this**, можно строить одни конструкторы на основе других.




```
// пример # 3 : this в конструкторе: Point2D.java, Point3D.java, Point4D.java
package chapt04;

public class Point2D {
      private int x, y;

      public Point2D(int x, int y) {
            this.x = x;//this используется для присваивания полям класса
            this.y = y;//x, y, значений параметров конструктора x, y, z
      }
}
package chapt04;

public class Point3D extends Point2D {
      private int z;

      public Point3D(int x, int y, int z) {
            super(x, y);
            this.z = z;
      }
```

```
        public Point3D() {
            this(-1,-1,-1);// вызов конструктора Point3D с параметрами
        }
}
package chapt04;

public class Point4D extends Point3D{
        private long time;

        public Point4D(int x, int y, int z, long time) {
                super(x, y, z);
                this.time = time;
        }
        public Point4D() {
                // по умолчанию super();
        }
}
```

В классе **Point3D** второй конструктор для завершения инициализации объекта обращается к первому конструктору. Такая конструкция применяется в случае, когда в класс требуется добавить конструктор по умолчанию с обязательным использованием уже существующего конструктора.

Ссылка **this** используется в методе для уточнения того, о каких именно переменных **x**, **y** и **z** идет речь в методе, а конкретно для доступа к переменным класса из метода, если в методе есть локальные переменные с тем же именем, что и у класса. Инструкция **this()** должна быть единственной в вызывающем конструкторе и быть первой по счету выполняемой операцией.

Переопределение методов и полиморфизм

Способность Java делать выбор метода, исходя из типа объекта во время выполнения, называется ***поздним связыванием***. При вызове метода его поиск происходит сначала в данном классе, затем в суперклассе, пока метод не будет найден или не достигнут **Object** – суперкласс для всех классов.

Если два метода с одинаковыми именами и возвращаемыми значениями находятся в одном классе, то списки их параметров должны отличаться. То же относится к методам, наследуемым из суперкласса. Такие методы являются перегружаемыми (overloading). При обращении вызывается тот метод, список параметров которого совпадает со списком параметров вызова. Если объявление метода подкласса полностью, включая параметры, совпадает с объявлением метода суперкласса (порождающего класса), то метод подкласса переопределяет (overriding) метод суперкласса. Переопределение методов является основой концепции динамического связывания, реализующей полиморфизм. Когда переопределенный метод вызывается через ссылку суперкласса, Java определяет, какую версию метода вызвать, основываясь на типе объекта, на который имеется ссылка. Таким образом, тип объекта определяет версию метода на этапе выполнения. В следующем примере рассматривается реализация полиморфизма на основе динамического связывания. Так как суперкласс содержит методы, переопределенные подклассами, то объект суперкласса будет вызывать методы раз-

личных подклассов, в зависимости от того, на объект какого подкласса у него имеется ссылка.

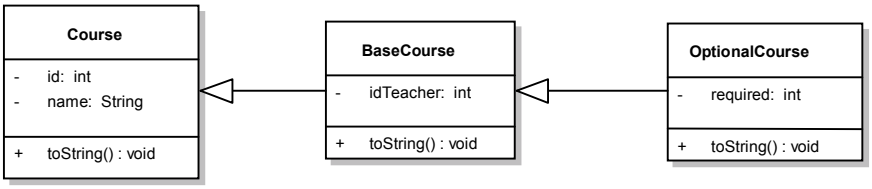


Рис. 4.1. Пример реализации полиморфизма

```
/* пример # 4 : динамическое связывание методов:  Course.java: BaseCourse.java:
OptionalCourse.java: DynDispatcher.java */
package chapt04;

public class Course {
      private int id;
      private String name;

      public Course(int i, String n) {
            id = i;
            name = n;
      }
      public String toString() {
            return "Название: " + name + "(" + id + ")";
      }
}
package chapt04;

public class BaseCourse extends Course {
      private int idTeacher;

      public BaseCourse(int i, String n, int it) {
            super(i, n);
            idTeacher = it;
      }
      public String toString() {
            /* просто toString() нельзя!!!
               метод будет вызывать сам себя, что
               приведет к ошибке во время выполнения  */
   return
      super.toString() + " препод.(" + idTeacher + ")";
      }
}
package chapt04;

public class OptionalCourse extends BaseCourse {
      private boolean required;
```

```
      public OptionalCourse(int i, String n, int it,
                                          boolean r) {
            super(i, n, it);
            required = r;
      }
      public String toString() {
        return super.toString() + " required->" + required;
      }
}
package chapt04;

public class DynDispatcher{
      public void infoCourse(Course c) {
            System.out.println(c.toString());
            //System.out.println(c);//идентично
      }
}
package chapt04;

public class Runner {
      public static void main(String[] args) {
            DynDispatcher d = new DynDispatcher();
            Course cc = new Course(7, "МА");
            d.infoCourse(cc);
            BaseCourse bc = new BaseCourse(71, "МП", 2531);
            d.infoCourse(bc);
            OptionalCourse oc =
                  new OptionalCourse(35, "ФА", 4128, true);
            d.infoCourse(oc);
      }
}
```

Результат:

```
Название: МА(7)
Название: МП(71) препод.(2531)
Название: ФА(35) препод.(4128) required->true
```

Следует помнить, что при вызове **toString()** обращение **super** всегда происходит к ближайшему суперклассу. Аналогично при вызове **super()** в конструкторе обращение происходит к соответствующему конструктору непосредственного суперкласса.

Основной вывод: выбор версии переопределенного метода производится на этапе выполнения кода.

Все методы Java являются виртуальными (ключевое слово virtual, как в C++, не используется).

Статические методы могут быть переопределены в подклассе, но не могут быть полиморфными, так как их вызов не затрагивает объекты. Их следует вызывать только с использованием имени класса.

Методы подставки

С пятой версии языка появилась возможность при переопределении методов указывать другой тип возвращаемого значения, в качестве которого можно использовать только типы, находящиеся ниже в иерархии наследования, чем исходный тип.

```
/* пример # 5 : методы-подставки:  CourseHelper.java:
BaseCourseHelper.java: RunnerCourse.java*/
package chapt04;

public class CourseHelper {
      public Course getCourse(){
             System.out.println("Course");
            return new Course();
      }
}
package chapt04;

public class BaseCourseHelper extends CourseHelper {
       public BaseCourse getCourse(){
              System.out.println("BaseCourse");
            return new BaseCourse();
      }
}
package chapt04;

public class RunnerCourse {
      public static void main(String[] args) {
            CourseHelper bch = new BaseCourseHelper();
            Course course = bch.getCourse();
            //BaseCourse course = bch.getCourse();//ошибка компиляции
            System.out.println(bch.getCourse().id);
      }
}
```

В данной ситуации при компиляции в подклассе **BaseCourseHelper** создаются два метода. При обращении к методу **getCourse()** версия метода определяется «ранним связыванием» без использования полиморфизма, но при выполнении вызывается метод-подставка. Обращение к полю производится по типу ссылки, возвращаемой методом **getCourse()**, то есть к полю класса **Course**.

Полиморфизм и расширяемость

В объектно-ориентированном программировании применение наследования предоставляет возможность расширения и дополнения программного обеспечения, имеющего сложную структуру с большим количеством классов и методов. В задачи базового класса в этом случае входит определение интерфейса (как способа взаимодействия) для всех наследников.

В следующем примере приведение к базовому типу происходит в выражении:

```
Transport s1 = new Bus();
Transport s2 = new Tram();
```

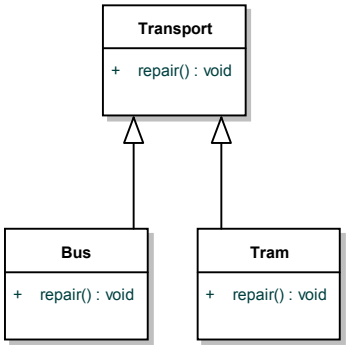


Рис. 4.2. Пример реализации полиморфизма

Базовый класс **Transport** предоставляет общий интерфейс для своих подклассов. Порожденные классы **Bus** и **Tram** перекрывают эти определения для обеспечения уникального поведения.

```
/* пример # 5 : полиморфизм: Transport.java: Bus.java: Tram.java:
RepairingCenter.java: Runner.java*/
package chapt04;
import java.util.Random;

class Transport {
      public void repair() {/* пустая реализация */
      }
}
class Bus extends Transport {
      public void repair() {
            System.out.println("отремонтирован АВТОБУС");
      }
}
class Tram extends Transport {
      public void repair() {
            System.out.println("отремонтирован ТРАМВАЙ");
      }
}
class RepairingFactory {//шаблон Factory
      public Transport getClassFromFactory(int numMode) {
            switch (new Random().nextInt(numMode)) {
             case 0:
                  return new Bus();
             case 1:
                  return new Tram();
             default:
```

```
                    throw new IllegalArgumentException();
                    // assert false;
                    // return null;
                    /*
              * if((int)(Math.random() * numMode)==0) return new Bus(); else
              * return new Tram(); как альтернативный и не очень удачный
              * вариант. Почему?
                     */
              }
        }
}
public class Runner {
      public static void main(String[] args) {
            RepairingFactory rc = new RepairingFactory();
            Transport[] box = new Transport[15];

      for (int i = 0; i < box.length; i++)
            /* заполнение массива единицами проверямого транспорта */
        box[i] = rc.getClassFromFactory(2);// 2 вида транспорта
            for (Transport s : box)
                  s.repair();// вызов полиморфного метода
      }
}
```

В процессе выполнения приложения будет случайным образом сформирован массив из автобусов и трамваев и информация об их ремонте будет выведена на консоль.

Класс **RepairingFactory** содержит метод **getClassFromFactory(int numMode)**, который возвращает ссылку на случайно выбранный объект подкласса класса **Transport** каждый раз, когда он вызывается. Приведение к базовому типу производится оператором **return**, который возвращает ссылку на **Bus** или **Tram**. Метод **main()** содержит массив из ссылок **Transport**, заполненный с помощью вызова **getClassFromFactory()**. На этом этапе известно, что имеется некоторое множество ссылок на объекты базового типа и ничего больше (не больше, чем знает компилятор). Когда происходит перемещение по этому массиву, метод **repair()** вызывается для каждого случайным образом выбранного объекта.

Если понадобится в дальнейшем добавить в систему, например, класс **TrolleyBus**, то это потребует только переопределения метода **repair()** и добавления одной строки в код метода **getClassFromFactory()**, что делает систему легко расширяемой.

Статические методы и полиморфизм

Переопределение статических методов класса не имеет практического смысла, так как обращение к статическому атрибуту или методу осуществляется посредством задания имени класса, которому они принадлежат. К статическим методам принципы «позднего связывания» неприменимы. При использовании ссылки для доступа к статическому члену компилятор при выборе метода или поля учитывает тип ссылки, а не тип объекта, ей присвоенного.

```
/* 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-

ние этих методов необходимо, если логика приложения предусматривает использование элементов в коллекциях. Метод **equals()** при сравнении двух объектов возвращает истину, если содержимое объектов эквивалентно, и ложь – в противном случае. При переопределении метода **equals()** должны выполняться соглашения, предусмотренные спецификацией языка Java, а именно:

- рефлексивность – объект равен самому себе;
- симметричность – если **x.equals(y)** возвращает значение **true**, то и **y.equals(x)** всегда возвращает значение **true**;
- транзитивность – если метод **equals()** возвращает значение **true** при сравнении объектов **x** и **y**, а также **y** и **z**, то и при сравнении **x** и **z** будет возвращено значение **true**;
- непротиворечивость – при многократном вызове метода для двух не подвергшихся изменению за это время объектов возвращаемое значение всегда должно быть одинаковым;
- ненулевая ссылка при сравнении с литералом **null** всегда возвращает значение **false**.

При создании информационных классов также рекомендуется переопределять методы **hashCode()** и **toString()**, чтобы адаптировать их действия для создаваемого типа.

Метод **hashCode()** переопределен, как правило, в каждом классе и возвращает число, являющееся уникальным идентификатором объекта, зависящим в большинстве случаев только от значения объекта. Его следует переопределять всегда, когда переопределен метод **equals()**. Метод **hashCode()** возвращает хэш-код объекта, вычисление которого управляется следующими соглашениями:

- во время работы приложения значение хэш-кода объекта не изменяется, если объект не был изменен;
- все одинаковые по содержанию объекты одного типа **должны** иметь одинаковые хэш-коды;
- различные по содержанию объекты одного типа **могут** иметь различные хэш-коды.

Один из способов создания правильного метода **hashCode()**, гарантирующий выполнение соглашений, приведен ниже, в примере # 10.

Метод **toString()** следует переопределять таким образом, чтобы кроме стандартной информации о пакете (опционально), в котором находится класс, и самого имени класса (опционально), он возвращал значения полей объекта, вызвавшего этот метод (то есть всю полезную информацию объекта), вместо хэш-кода, как это делается в классе **Object**. Метод **toString()** класса **Object** возвращает строку с описанием объекта в виде:

```
getClass().getName() + '@' +
Integer.toHexString(hashCode())
```

Метод вызывается автоматически, когда объект выводится методами **println()**, **print()** и некоторыми другими.

```
/* пример # 10 : переопределение методов equals(), hashCode, toString():
Student.java */
package chapt04;
```

```
public class Student {
      private int id;
      private String name;
      private int age;

      public Student(int id, String name, int age){
            this.id = id;
            this.name = name;
            this.age = age;
      }
      public int getId() {
            return id;
      }
      public String getName() {
            return name;
      }
      public int getAge() {
            return age;
      }
      public boolean equals(Object obj) {
            if (this == obj)
                  return true;
            if (obj == null)
                  return false;
            if (obj instanceof Student){ //warning
                  Student temp = (Student) obj;
                  return this.id == temp.id &&
                         name.equals(temp.name) &&
                         this.age == temp.age;
            } else
                  return false;
      }
      public int hashCode() {
            return (int)(31 * id + age
            + ((name == null) ? 0 : name.hashCode()));
      }
      public String toString() {
            return getClass().getName() + "@name" + name
                        + " id:" + id + " age:" + age;
      }
}
```

Выражение **`31 * id + age`** гарантирует различные результаты вычислений при перемене местами значений полей, а именно если **`id=1`** и **`age=2`**, то в результате будет получено **`33`**, если значения поменять местами, то **`63`**. Такой подход применяется при наличии у классов полей базовых типов.

Метод **`equals()`** переопределяется для класса **`Student`** таким образом, чтобы убедиться в том, что полученный объект является объектом типа **`Student`** или одним из его наследников, а также сравнить содержимое полей **`id`**, **`name`** и

age соответственно у вызывающего метод объекта и объекта, передаваемого в качестве параметра.

```
/*пример # 11 : класс студента факультета: SubStudent.java */
package chapt04;

public class SubStudent extends Student {
     private int idFaculty;

     public SubStudent (int id, String n, int a, int idf){
            super(id, n, a);
            this.idFaculty = idf;
       }
}
```

```
/*пример # 12 : демонстрация работы метода equals() при наследовании:
 StudentEq.java */
package chapt04;

public class StudentEq {
     public static void main(String[] args) {
            Student p1 = new Student(71, "Петров", 19);
            Student p2 = new Student(71, "Петров", 19);
            SubStudent p3 =
                 new SubStudent(71, "Петров", 19, 5);
            System.out.println(p1.equals(p2));
            System.out.println(p1.equals(p3));
            System.out.println(p3.equals(p1));
     }
}
```

В результате выполнения данного кода будет выведено следующее:

```
true
true
true
```

Переопределенный таким образом метод **equals()** позволяет сравнивать объекты суперкласса с объектами подклассов, но только по тем полям, которые являются общими. При наследовании с добавлением новых полей в подкласс использование метода сравнения из суперкласса приводит к некорректным результатам.

Эту проблему можно легко разрешить, если вместо строки с пометкой *//warning* в метод **equals()** класса **Student** подставить непосредственную проверку на соответствие типов сравниваемых объектов с использованием объекта класса **Class** в виде:

```
if (getClass() == obj.getClass())
```

то в результате будет выведено:

```
true
false
false
```

В то же время такая реализация метода **equals()** будет возвращать истину при сравнении объектов класса **SubStudent** с одинаковыми значениями полей, унаследованных от класса **Student**.

Клонирование объектов

Объекты в методы передаются по ссылке, в результате чего в метод передается ссылка на объект, находящийся вне метода. Поэтому если в методе изменить значение поля объекта, то это изменение коснется исходного объекта. Во избежание такой ситуации для защиты внешнего объекта следует создать клон (копию) объекта в методе. Класс **Object** содержит **protected**-метод **clone()**, осуществляющий побитовое копирование объекта производного класса. Однако сначала необходимо переопределить метод **clone()** как **public** для обеспечения возможности вызова из другого пакета. В переопределенном методе следует вызвать базовую версию метода **super.clone()**, которая и выполняет собственно клонирование. Чтобы окончательно сделать объект клонируемым, класс должен реализовать интерфейс **Cloneable**. Интерфейс **Cloneable** не содержит методов относится к помеченным (tagged) интерфейсам, а его реализация гарантирует, что метод **clone()** класса **Object** возвратит точную копию вызвавшего его объекта с воспроизведением значений всех его полей. В противном случае метод генерирует исключение **CloneNotSupportedException**. Следует отметить, что при использовании этого механизма объект создается без вызова конструктора. В языке C++ аналогичный механизм реализован с помощью конструктора копирования.

```
/* пример # 13 : класс, поддерживающий клонирование: Student.java */
package chapt04;

public class Student implements Cloneable {/*включение
                                                интерфейса */
      private int id = 71;

      public int getId() {
            return id;
      }
      public void setId(int value) {
            id = value;
      }
      public Object clone() {//переопределение метода
            try {
                  return super.clone();//вызов базового метода
            } catch (CloneNotSupportedException e) {
                  throw new AssertionError("невозможно!");
            }
      }
}
```

```
/* пример # 14 : безопасная передача по ссылке: DemoSimpleClone.java */
package chapt04;
```

```
public class DemoSimpleClone {
      private static void changeId(Student p) {
            p = (Student) p.clone();//клонирование
            p.setId(1000);
            System.out.println("->id = " + p.getId());
            }
      public static void main(String[] args) {
            Student ob = new Student();
            System.out.println("id = " + ob.getId());
            changeId(ob);
            System.out.println("id = " + ob.getId());
      }
}
```

В результате будет выведено:

```
id = 71
->id = 1000
id = 71
```

Если закомментировать вызов метода **clone()**, то выведено будет следующее:

```
id = 71
->id = 1000
id = 1000
```

Такое решение эффективно только в случае, если поля клонируемого объекта представляют собой значения базовых типов и их оболочек или неизменяемых (immutable) объектных типов. Если же поле клонируемого типа является изменяемым объектным типом, то для корректного клонирования требуется другой подход. Причина заключается в том, что при создании копии поля оригинал и копия представляют собой ссылку на один и тот же объект. В этой ситуации следует также клонировать и объект поля класса.

```
/* пример # 15 : глубокое клонирование: Student.java */
package chapt04;
import java.util.ArrayList;

public class Student implements Cloneable {
      private int id = 71;
      private ArrayList<Mark> lm = new ArrayList<Mark>();

      public int getId() {
            return id;
      }
      public void setId(int id) {
            this.id = id;
      }
      public ArrayList<Mark> getMark() {
            return lm;
      }
```

```
    public void setMark(ArrayList<Mark> lm) {
         this.lm = lm;
    }
    public Object clone() {
          try {
                Student copy =(Student)super.clone();
                copy.lm = (ArrayList<Mark>)lm.clone();
                return copy;
          } catch (CloneNotSupportedException e) {
                throw new AssertionError(
                            "отсутствует Cloneable!");
          }
    }
}
```

Такое клонирование возможно только в случае, если тип атрибута класса также реализует интерфейс **Cloneable** и переопределяет метод **clone()**. В противном случае вызов метода невозможен, так как он просто недоступен. Следовательно, если класс имеет суперкласс, то для реализации механизма клонирования текущего класса необходимо наличие корректной реализации такого механизма в суперклассе. При этом следует отказаться от использования объявлений **final** для полей объектных типов по причине невозможности изменения их значений при реализации клонирования.

"Сборка мусора" и освобождение ресурсов

Так как объекты создаются динамически с помощью операции **new**, а уничтожаются автоматически, то желательно знать механизм ликвидации объектов и способ освобождения памяти. Автоматическое освобождение памяти, занимаемой объектом, выполняется с помощью механизма "сборки мусора". Когда никаких ссылок на объект не существует, то есть все ссылки на него вышли из области видимости программы, предполагается, что объект больше не нужен, и память, занятая объектом, может быть освобождена. "Сборка мусора" происходит нерегулярно во время выполнения программы. Форсировать "сборку мусора" невозможно, можно лишь "рекомендовать" ее выполнить вызовом метода **System.gc()** или **Runtime.getRuntime().gc()**, но виртуальная машина выполнит очистку памяти тогда, когда сама посчитает это удобным. Вызов метода **System.runFinalization()** приведет к запуску метода **finalize()** для объектов утративших все ссылки.

Иногда объекту нужно выполнять некоторые действия перед освобождением памяти. Например, освободить внешние ресурсы. Для обработки таких ситуаций могут применяться два способа: конструкция **try-finally** и механизм finalization. Конструкция **try-finally** является предпочтительной, абсолютно надежной и будет рассмотрена в девятой главе. Запуск механизма finalization определяется алгоритмом сборки мусора и до его непосредственного исполнения может пройти сколь угодно много времени. Из-за всего этого поведение метода **finalize()** может повлиять на корректную работу программы, особенно при смене JVM. Если существует возможность освободить ресурсы или выполнить другие подобные действия без привлечения этого механизма, то лучше без него

обойтись. Виртуальная машина вызывает этот метод всегда, когда она собирается уничтожить объект данного класса. Внутри метода **finalize()**, вызываемого непосредственно перед освобождением памяти, следует определить действия, которые должны быть выполнены до уничтожения объекта.

Метод **finalize()** имеет следующую сигнатуру:

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

Ключевое слово **protected** запрещает доступ к **finalize()** коду, определенному вне этого класса. Метод **finalize()** вызывается только перед самой "сборкой мусора", а не тогда, когда объект выходит из области видимости, то есть заранее невозможно определить, когда **finalize()** будет выполнен, и недоступный объект может занимать память довольно долго. В принципе этот метод может быть вообще не выполнен! Недопустимо в приложении доверять такому методу критические по времени действия по освобождению ресурсов.

```
/* пример # 16 : класс Manager с поддержкой finalization : Manager.java */
package chapt04;

class Manager {
      private int id;

      public Manager(int value) {
            id = value;
      }
      protected void finalize() throws Throwable {
          try {
          //освобождение ресурсов
        System.out.println("объект будет удален, id=" + id);
          } finally {
              super.finalize();
          }
    }
}
package chapt04;

public class FinalizeDemo {
     public static void main(String[] args) {
          Manager d1 = new Manager(1);
          d1 = null;
          Manager d2 = new Manager(2);
          Object d3 = d2;                    //1
          //Object d3 = new Manager (3);     //2
          d2 = d1;
          System.gc();// просьба выполнить "сборку мусора"
     }
}
```

В результате выполнения этого кода перед вызовом метода **`System.gc()`** без ссылки останется только один объект.

```
объект будет удален, id=1
```

Если закомментировать строку 1 и снять комментарий со строки 2, то перед выполнением **`gc()`** ссылку потеряют уже два объекта.

```
объект будет удален, id=1
объект будет удален, id=2
```

Если не вызвать явно метод **`finalize()`** суперкласса, то он не будет вызван автоматически. Еще одна опасность заключается в том, что если при выполнении данного метода возникнет исключительная ситуация, то она будет проигнорирована и приложение будет продолжать выполняться, что также представляет опасность для его корректной работы.

Задания к главе 4

Вариант А

Создать приложение, удовлетворяющее требованиям, приведенным в задании. Аргументировать принадлежность классу каждого создаваемого метода и корректно переопределить для каждого класса методы **`equals()`**, **`hashCode()`**, **`toString()`**.

1. Создать объект класса **Текст**, используя класс **Абзац**. Методы: дополнить текст, вывести на консоль текст, заголовок текста.
2. Создать объект класса **Автомобиль**, используя класс **Колесо**. Методы: ехать, заправляться, менять колесо, вывести на консоль марку автомобиля.
3. Создать объект класса **Самолет**, используя класс **Крыло**. Методы: летать, задавать маршрут, вывести на консоль маршрут.
4. Создать объект класса **Беларусь**, используя класс **Область**. Методы: вывести на консоль столицу, количество областей, площадь, областные центры.
5. Создать объект класса **Планета**, используя класс **Материк**. Методы: вывести на консоль название материка, планеты, количество материков.
6. Создать объект класса **Звездная система**, используя классы **Планета**, **Звезда**, **Луна**. Методы: вывести на консоль количество планет в звездной системе, название звезды, добавление планеты в систему.
7. Создать объект класса **Компьютер**, используя классы **Винчестер**, **Дисковод**, **ОЗУ**. Методы: включить, выключить, проверить на вирусы, вывести на консоль размер винчестера.
8. Создать объект класса **Квадрат**, используя классы **Точка, Отрезок.** Методы: задание размеров, растяжение, сжатие, поворот, изменение цвета.
9. Создать объект класса **Круг**, используя классы **Точка, Окружность.** Методы: задание размеров, изменение радиуса, определение принадлежности точки данному кругу.
10. Создать объект класса **Котёнок**, используя классы **Животное, Кошка.** Методы: вывести на консоль имя, подать голос, рожать потомство (создавать себе подобных).

11. Создать объект класса **Наседка**, используя классы **Птица, Кукушка.** Методы: летать, петь, нести яйца, высиживать птенцов.
12. Создать объект класса **Текстовый файл**, используя класс **Файл.** Методы: создать, переименовать, вывести на консоль содержимое, дополнить, удалить.
13. Создать объект класса **Одномерный массив**, используя класс **Массив.** Методы: создать, вывести на консоль, выполнить операции (сложить, вычесть, перемножить).
14. Создать объект класса **Простая дробь**, используя класс **Число.** Методы: вывод на экран, сложение, вычитание, умножение, деление.
15. Создать объект класса **Дом**, используя классы **Окно, Дверь.** Методы: закрыть на ключ, вывести на консоль количество окон, дверей.
16. Создать объект класса **Роза**, используя классы **Лепесток, Бутон.** Методы: расцвести, завять, вывести на консоль цвет бутона.
17. Создать объект класса **Дерево**, используя классы **Лист.** Методы: зацвести, опасть листьям, покрыться инеем, пожелтеть листьям.
18. Создать объект класса **Пианино**, используя класс **Клавиша.** Методы: настроить, играть на пианино, нажимать клавишу.
19. Создать объект класса **Фотоальбом**, используя класс **Фотография.** Методы: задать название фотографии, дополнить фотоальбом фотографией, вывести на консоль количество фотографий.
20. Создать объект класса **Год**, используя классы **Месяц, День.** Методы: задать дату, вывести на консоль день недели по заданной дате, рассчитать количество дней, месяцев в заданном временном промежутке.
21. Создать объект класса **Сутки**, используя классы **Час, Минута.** Методы: вывести на консоль текущее время, рассчитать время суток (утро, день, вечер, ночь).
22. Создать объект класса **Птица**, используя класс **Крылья.** Методы: летать, питаться.
23. Создать объект класса **Тигр**, используя класс **Когти.** Методы: рычать, бежать, добывать пищу.
24. Создать объект класса **Гитара**, используя класс **Струна.** Методы: играть, натягивать струну.

Вариант В

Построить модель программной системы.

1. Система **Факультатив**. **Преподаватель** объявляет запись на **Курс**. **Студент** записывается на **Курс**, обучается и по окончании **Преподаватель** выставляет **Оценку**, которая сохраняется в **Архиве**. **Студентов**, **Преподавателей** и **Курсов** при обучении может быть несколько.
2. Система **Платежи**. **Клиент** имеет **Счет** в банке и **Кредитную Карту (КК)**. **Клиент** может оплатить **Заказ**, сделать платеж на другой **Счет**, заблокировать **КК** и аннулировать **Счет**. **Администратор** может заблокировать **КК** за превышение кредита.
3. Система **Больница**. **Пациенту** назначается лечащий **Врач**. **Врач** может сделать назначение **Пациенту** (процедуры, лекарства, операции). **Медсестра** или другой **Врач** выполняют назначение. **Пациент** может быть выписан из **Больницы** по окончании лечения, при нарушении режима или при иных обстоятельствах.

4. Система **Вступительные экзамены**. **Абитуриент** регистрируется на **Факультет**, сдает **Экзамены**. **Преподаватель** выставляет **Оценку**. Система подсчитывает средний балл и определяет **Абитуриентов**, зачисленных в учебное заведение.
5. Система **Библиотека**. **Читатель** оформляет **Заказ** на **Книгу**. Система осуществляет поиск в **Каталоге**. **Библиотекарь** выдает **Читателю Книгу** на абонемент или в читальный зал. При невозвращении **Книги Читателем** он может быть занесен **Администратором** в «черный список».
6. Система **Конструкторское бюро**. **Заказчик** представляет **Техническое Задание (ТЗ)** на проектирование многоэтажного **Дома**. **Конструктор** регистрирует **ТЗ**, определяет стоимость проектирования и строительства, выставляет **Заказчику Счет** за проектирование и создает **Бригаду Конструкторов** для выполнения Проекта.
7. Система **Телефонная станция**. **Абонент** оплачивает **Счет** за разговоры и **Услуги**, может попросить **Администратора** сменить номер и отказаться от услуг. **Администратор** изменяет номер, **Услуги** и временно отключает **Абонента** за неуплату.
8. Система **Автобаза**. **Диспетчер** распределяет заявки на **Рейсы** между **Водителями** и назначает для этого **Автомобиль**. **Водитель** может сделать заявку на ремонт. **Диспетчер** может отстранить **Водителя** от работы. **Водитель** делает отметку о выполнении **Рейса** и состоянии **Автомобиля**.
9. Система **Интернет-магазин**. **Администратор** добавляет информацию о **Товаре**. **Клиент** делает и оплачивает **Заказ** на **Товары**. **Администратор** регистрирует **Продажу** и может занести неплательщиков в «черный список».
10. Система **Железнодорожная касса**. **Пассажир** делает **Заявку** на станцию назначения, время и дату поездки. Система регистрирует **Заявку** и осуществляет поиск подходящего **Поезда**. **Пассажир** делает выбор **Поезда** и получает **Счет** на оплату. **Администратор** вводит номера **Поездов**, промежуточные и конечные станции, цены.
11. Система **Городской транспорт**. На **Маршрут** назначаются **Автобус**, **Троллейбус** или **Трамвай**. Транспортные средства должны двигаться с определенным для каждого **Маршрута** интервалом. При поломке на **Маршрут** должен выходить резервный транспорт или увеличиваться интервал движения.
12. Система **Аэрофлот**. **Администратор** формирует летную **Бригаду** (пилоты, штурман, радист, стюардессы) на **Рейс**. Каждый **Рейс** выполняется **Самолетом** с определенной вместимостью и дальностью полета. **Рейс** может быть отменен из-за погодных условий в **Аэропорту** отлета или назначения. **Аэропорт** назначения может быть изменен в полете из-за технических неисправностей, о которых сообщил командир.
13. **Система Периодические издания**. **Читатель** может сделать **Заявку**, предварительно выбрав периодические **Издания** из списка. Система подсчитывает сумму для оплаты. **Читатель** оплачивает **заявку**. **Администратор** добавляет **Заявку** в «черный список», если **Клиент** не оплачивает её в определённый срок.

14. **Система Заказ гостиницы**. **Клиент** оставляет **Заявку** на **Номер**, указав количество мест в номере, класс апартаментов и время пребывания. **Администратор** рассматривает **Заявку**, подтверждает или отклоняет её. Результат просматривает **Клиент**. В случае подтверждения **Заявки Клиент** оплачивает услуги.
15. **Система Жилищно-коммунальные услуги**. **Квартиросъемщик** отправляет **Заявку**, в которой указывает род работ, масштаб и желаемое время выполнения. **Диспетчер** формирует соответствующую **Бригаду** и регистрирует её в **Плане работ**. **Диспетчер** может отклонить **Заявку** в случае занятости всех **Бригад**.
16. **Система Прокат автомобилей**. **Клиент** выбирает **Автомобиль** из списка доступных, заполняет форму **Заказа**, указывая паспортные данные, срок аренды. **Администратор** может отклонить **Заявку**, указав причины отказа. При подтверждении **Заявки Клиент** оплачивает **Заказ**. **Система** выписывает сумму. В случае повреждения **Автомобиля Клиентом Администратор** вносит соответствующие пометки.

Тестовые задания к главе 4

Вопрос 4.1.

Дан код:

```
class Base {}
class A extends Base {}
 public class Quest{
    public static void main(String[] args){
      Base b = new Base();
      A ob = (A) b;
    } }
```

Результатом компиляции и запуска будет:

1) компиляция и выполнение без ошибок;
2) ошибка во время компиляции;
3) ошибка во время выполнения.

Вопрос 4.2.

Классы **A** и **Quest2** находятся в одном файле. Что необходимо изменить в объявлении класса **Quest2**, чтобы оно было корректным?

```
public class A{}
class Quest2 extends A, Object {}
```

1) необходимо убрать спецификатор **public** перед **A**;
2) необходимо добавить спецификатор **public** к **Quest2**;
3) убрать после **extends** один из классов;
4) класс **Object** нельзя указывать явно.

Вопрос 4.3.

Дан код:

```
class A {A(int i) {}}  // 1
class B extends A {}   // 2
```

Какие из следующих утверждений верны? (выберите два)

1) компилятор пытается создать по умолчанию конструктор для класса **A**;
2) компилятор пытается создать по умолчанию конструктор для класса **B**;
3) ошибка во время компиляции в строке 1;
4) ошибка во время компиляции в строке 2.

Вопрос 4.4.

Дан код, находящийся в файле **Quest.java**:

```
public class Base{
    Base(){
      int i = 1;
      System.out.print(i);
    } }
public class Quest4 extends Base{
    static int i;
    public static void main(String [] args){
      Quest4 ob = new Quest4();
      System.out.print(i);
    } }
```

В результате компиляции и запуска будет выведено:

1) ошибка компиляции;
2) 0;
3) 10;
4) 1;
5) ошибка выполнения.

Вопрос 4.5.

Что будет результатом компиляции и выполнения следующего кода?

```
class Q {
    private void show(int i){
      System.out.println("1");
    } }
class  Quest5 extends Q{
    public void show(int i){
      System.out.println("2");
    }
    public static void main(String[] args){
      Q ob = new  Quest5();
      int i = '1'; //1
      ob.show(i);
    } }
```

1) ошибка компиляции: метод **show()** недоступен;
2) ошибка времени выполнения: метод **show()** недоступен;
3) ошибка компиляции: несовпадение типов в строке 1;
4) 2;
5) 1.

Вопрос 4.6.

Что будет результатом компиляции и выполнения следующего кода?

```
class Q {
    void mQ(int i) {
        System.out.print("mQ" + i);
    } }
class Quest6 extends Q {
    public void mQ(int i) {
        System.out.print("mQuest" + i);
    }
        public void mP(int i) {
        System.out.println("mP" + i);
    }
    public static void main(String args[]) {
       Q ob = new Quest6(); //1
        ob.mQ(1); //2
        ob.mP(1); //3
    } }
```

1) mQ1 mP1;
2) mQuest1 mP1;
3) ошибка компиляции в строке //1;
4) ошибка компиляции в строке //2;
5) ошибка компиляции в строке //3.

Вопрос 4.7.

Как следует вызвать конструктор класса **A**, чтобы в результате выполнения кода была выведена на консоль строка в " Конструктор A ".

```
class A{
  A(int i){ System.out.print("Конструктор A");     }
}
public class Quest extends A{
    public static void main(String[] args){
      Quest s= new Quest();
      //1
    }
      public Quest(){
      //2
    }
   public void show()     {
      //3
    } }
```

1) вместо //1 написать **A(1);**
2) вместо //1 написать **super(1);**
3) вместо //2 написать **super(1);**
4) вместо //2 написать **A(1);**
5) вместо //3 написать **super(1).**