

4. НАСЛЕДОВАНИЕ РЕАЛИЗАЦИИ

Цель – изучить принципы механизма наследования реализации при разработке объектно-ориентированных приложений.

Задачи:

- изучить синтаксические принципы реализации наследования;
- изучить принципы реализации наследования при решении практических задач.

Формируемые компетенции: ПК-11; ПК-12; ПК-30.

Теоретическая часть

Механизм наследования реализации. *Наследование* – фундаментальное понятие объектно-ориентированного программирования. Идея, лежащая в основе наследования, заключается в том, что новые классы можно создавать из уже существующих. При наследовании реализации методы, и поля существующего класса используются повторно (наследуются) вновь создаваемым классом, причем для адаптации нового класса к новым ситуациям в него добавляют дополнительные поля и методы. Этот прием играет в языке Java важную роль.

Проектирование приложения с применением механизма наследования. Для демонстрации принципов наследования рассмотрим пример. В рамках проекта необходимо спроектировать классы для сущностей предметной области геоинформационная система (ГИС). Выделим следующие сущности: «Подвижный объект», «Неподвижный объект», «Человек», «Транспорт», «Здание», «Дорога». Очевидно, что для описания реально функционирующей ГИС, предметную область нужно детализировать. При проектировании данного учебного примера будем придерживаться определенных допущений. Превосходство объектно-ориентированного подхода в том, что на данном этапе проектирования разработчик может описывать сущности (и их свойства) предметной области в терминах естественного языка:

- для сущности «Подвижный объект» необходимо реализовать хранение данных о перемещении объекта;
- сущность «Неподвижный объект» содержит информацию о размерах, геометрии, координатах объекта;
- сущность «Человек» инкапсулирует информацию: фамилия, имя, отчество, выполняемое задание;
- сущность «Транспорт» инкапсулирует информацию: номер, тип, модель;
- сущность «Здание» предполагает, что все здания имеют форму прямоугольника;
- дороги в ГИС представляются отрезками; сущность «Дорога» инкапсулирует начальную и конечную координату представляемого пути.

Для ГИС характерно, что все представляемые на карте объекты должны иметь определенные координаты – то есть свойство наличия координат является общим для всех сущностей.

Следует также обратить внимание, что сущности «Дорога» и «Здание» обладает определенной семантической связью с сущностью «Неподвижный объект»: две случаи являются частными случаями третьей. Подобные связи можно установить и для прочих выявленных типов объектов проектируемой ГИС.

От словесного описания сущностей следует перейти к определению классов Java. Это удобно реализовать в виде графического представления – диаграммы классов. На рис. 4.1 представлена диаграмма классов.

Кроме перечисленных сущностей на диаграмме отображен класс Point, созданный для представления двумерной координаты в ГИС.

На диаграмме введен «главный» класс GISObject (любой объект, присутствующий в ГИС). Этот класс инкапсулирует информацию, которая необходима для любого объекта, присутствующего в нашей системе: координата и наименование. Эти поля реализованы как protected. В классе присутствует также открытый метод getName(), который возвращает имя объекта (рис. 4.2).

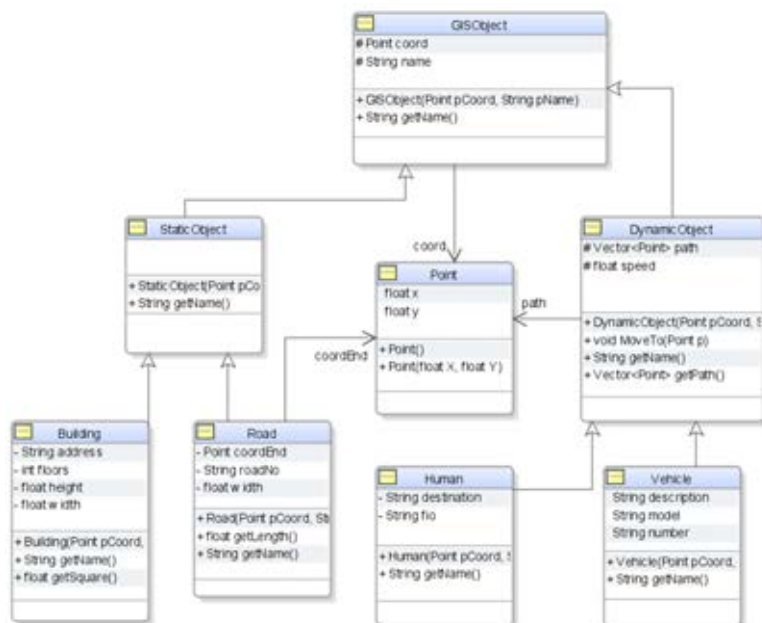


Рис. 4.1. Диаграмма классов приложения ГИС

```

1 public class GISObject {
2     protected Point coord;
3     protected String name;
4
5     public GISObject(Point pCoord, String pName) {
6         coord = pCoord;
7         name = pName;
8     }
9
10    public String getName() {
11        return this.name;
12    }
13 }
14
15

```

Рис. 4.2. Класс GISObject

Поля класса GISObject понадобятся в каждом классе, но на диаграмме эти поля не повторяются – вместо этого между классами установлены связи. Графически связь представлена как \rightarrow . Класс, к которому направлена стрелка, является базовым (класс-родитель, суперкласс); от которого – производным (класс-наследник, подкласс). Подобная связь указывает, что класс DynamicObject наследует реализацию GISObject. Под реализацией класса понимаются поля данных и сигнатуры и тела методов, то есть класс-наследник обладает такой же реализацией (функциональностью). Но, одновременно с реализацией, класс-наследник расширяет базовый класс, то есть добавляет что-то свое, специфическое – добавляет поля и методы, отсутствующие в базовом классе. Теперь трансформируем диаграмму в код Java. Рассмотрим трансформацию на примере классов DynamicObject и GISObject. Диаграмма была создана в IDE JDeveloper 12с, исходный код был сгенерирован автоматически

В технологии Java для механизма наследования реализации наиболее подходит термин «расширяет». Синтаксически наследование определяется с использованием зарезервированного слова `extends` (рис. 4.3). В реализации класса DynamicObject присутствует метод `getName()`, сигнатура которого совпадает с методом класса GISObject. Это демонстрация механизма переопределения методов. Метод один для обоих классов, но реализация его особенная для каждого класса.

Существуют поля (например, поле `name` класса GISObject), определение которых отсутствуют в производных классах, но они видны в классах-наследниках (на рис. 4.3 в строке 11 происходит обращение к полю базового класса). Это указывает на тот факт, что реализация базового класса была унаследована в производном классе.

На рис. 4.1 показан не только механизм наследования на примере двух классов (родитель – потомок), а спроектированы классы ГИС в виде иерархии. Совокупность всех классов, производных от одного суперкласса, называется иерархией наследования (например, рис. 4.1). Путь от конкретного класса к его потомкам в иерархии называется цепочкой наследования.

```

1 public class DynamicObject extends GISObject {
2     protected Vector<Point> path;
3     protected float speed;
4
5     public DynamicObject(Point pCoord, String pName) {
6         super(pCoord, pName);
7     }
8
9     @Override
10    public String getName() {
11        return "Мобильный объект: "
12            + this.name;
13    }
14
15    public Vector<Point> getPath() {
16        return path;
17    }
18
19    public void MoveTo(Point p) {
20        path.add(p);
21    }
22 }

```

Рис. 4.3. Класс DynamicObject

Базовый класс иерархии – GISObject. Одни поля этого класса (name, coord) используются всеми классами иерархии; другие (метод getName()) переопределяются, если это необходимо.

Дополнительно, необходимо обратить внимание на спецификаторы доступа полей name и coord класса GISObject. Они маркированы protected. Это гарантирует, что поля будут закрыты для доступа к ним через объектную переменную типа GISObject, но они будут видны в классах-потомках (в контекстах их методов).

Зарезервированные слова. При реализации наследования программисту доступны зарезервированные слова, которые указывают, какой именно метод необходимо вызывать: метод базового класса или метод производного. Если при вызове метода используется вызов метода с использованием синтаксиса:

myMeth();

то этот вызов, фактически, заменяется на

this.myMeth();

Но если метод `myMeth()` переопределен и необходимо явно указать, что нужно вызвать метод базового класса, то программист должен указать это явно: `super.myMeth()`;

Зарезервированное слово `super` отправляет компилятор к полям базового класса.

Реализация полиморфизма. Экономия при наборе кода, предотвращение дублирования блоков кода – не единственные преимущества наследования. Наследование как механизм открывает путь к полиморфному поведению объектов. Полиморфизм, наряду с наследованием, является фундаментальным принципом ООП.

Существует простое правило, позволяющее определить, стоит ли в конкретной ситуации применять наследование или нет. Если между объектами существует отношение «является» («is-a»), то каждый объект подкласса является объектом суперкласса.

Например, каждый объект типа `Vehicle` («Транспорт») является объектом типа `DynamicObject` («Подвижный объект»). Следовательно, имеет смысл сделать класс `Vehicle` подклассом класса `DynamicObject`. Естественно, обратное утверждение неверно – не каждый подвижный объект в ГИС является автотранспортом.

Другой способ – принцип подстановки. Этот принцип гласит, что объект подкласса можно использовать вместо любого объекта суперкласса.

Например, объект подкласса можно присвоить переменной суперкласса (рис. 4.4).

```

3 public class Starter {
4     public static void main(String[] args) {
5         DynamicObject e = new DynamicObject(new Point(), "Объект 0");
6         e = new Human(new Point(), "Новик Е.И.", "?");
7         e = new Vehicle(new Point(), "Hyundai Sol",
8                         "Y445KE", "?");
9         System.out.println(e.getName());
10    }
11 }

```

Рис. 4.4. Полиморфный объект

В языке Java объектные переменные являются полиморфными. Переменная типа `DynamicObject` может ссылаться как на объект `DynamicObject`, так и на объект любого подкласса класса `DynamicObject` (например, `Vehicle`, `Human`).

Рассмотрим проявление полиморфизма на примере. Для манипулирования объектами в ГИС создается единый массив объектов.

На рис. 4.3 представлено определение класса `DynamicObject`, на рис. 4.5 и 4.6 представлены определения классов `Vehicle` и `Human` соответственно.

```
1 public class Vehicle extends DynamicObject {  
2     String model;  
3     String number;  
4     String description;  
5  
6     public Vehicle(Point pCoord, String pModel,  
7         String pNumber, String pDescr) {  
8         super(pCoord, pNumber);  
9         model = pModel;  
10        description = pDescr;  
11    }  
12  
13    @Override  
14    public String getName() {  
15        return "Транспорт" +  
16            "\n Модель: " + model +  
17            "\n Номер: " + number +  
18            "\n Информация: " + description;  
19    }  
20 }  
21  
22 }
```

Рис. 4.5. Определение класс Vehicle

```
1 public class Human extends DynamicObject {  
2     private String fio;  
3     private String destination;  
4  
5     public Human(Point pCoord, String pFIO, String pDest) {  
6         super(pCoord, pFIO);  
7         fio = pFIO;  
8         destination = pDest;  
9     }  
10  
11    @Override  
12    public String getName() {  
13        return "ФИО: " + fio +  
14            "\n Задание: " + destination;  
15    }  
16 }  
17  
18 }
```

Рис. 4.6. Определение класс Human

Модуль, который будет использовать данные классы должен работать со всеми подвижными объектами единообразными методами, то есть не нужно писать два разных алгоритма, чтобы вывести описание сущностей «Человек» и «Транспорт». Поэтому необходимо хранить объекты классов Human и Vehicle в одном списке (массиве), работая с каждым элементом единообразно, вызывая методы базового класса DynamicObject (рис. 4.7).

```

6 public class Starter {
7     public static void main(String[] args) {
8
9         Vector<DynamicObject> allDynamics =
10             new Vector<DynamicObject>();
11         allDynamics.add(new Vehicle(
12             new Point(), "Hyundai Sol", "Y445KE", "?"));
13         allDynamics.add(new Human(
14             new Point(), "Новак Е.И.", "в центр"));
15         allDynamics.add(new Vehicle(
16             new Point(), "Kia Rio", "YlllKE", "новый"));
17         allDynamics.add(new Human(
18             new Point(), "Николаев Е.И.", "на юго-запад"));
19         allDynamics.add(new Vehicle(
20             new Point(), "Лада", "K433TE", "белый"));
21
22         // Единообразная работа со всеми элементами
23         for(int i = 0; i < allDynamics.size(); i++)
24             System.out.println(allDynamics.get(i).getName());
25     }
26 }

```

Рис. 4.7. Работа со списком DynamicObject

На рис. 4.8 показан фрагмент вывода программы, демонстрирующий, что метод getName(), вызываемый на каждом объекте коллекции, отображает различную информацию. Данный метод не только выводит разные строки – он обращается к различным полям внутри объектов-элементов коллекции.



Рис. 4.8. Вывод приложения ГИС

Предотвращение наследования. Абстрактные классы. Иногда наследование является нежелательным. Классы, которые нельзя расширить, называются *финальными*. Для того чтобы отметить этот факт, в определении класса используется ключевое слово `final`. Например:

```

1 public final class Human extends DynamicObject {
2     private String fio;
3     private String destination;
4 }

```

При таком определении класса `Human` он не может являться суперклассом ни для какого другого класса; на данном классе прерывается цепочка наследования.

Спецификатором `final` может быть помечен отдельный метод класса. В этом случае запрещено переопределять этот метод в подклассах.

Например, класс `String` библиотеки `Java` определен как `final`. Создавать подклассы этого класса запрещено. Поэтому если у вас есть

переменная типа `String`, то вы можете быть уверены, что она ссылается именно на строку, и ни на какой другой объект.

При разработке приложений программист определяет собственные классы, не являющиеся подклассами какого-либо иного класса. Например класс `GISObject` является корнем иерархии (рис. 4.1). В Java такие классы неявным образом расширяют класс `Object`.

Если суперкласс явно не указан, им считается класс `Object`. Поскольку в языке Java каждый класс расширяет `Object`, программисту необходимо знать, какими возможностями обладает сам класс `Object`.

Динамическое связывание. Важно понимать, что происходит при вызове метода, принадлежащего некоторому объекту. При этом выполняются следующие действия.

1. Компилятор проверяет объявленный тип объекта и имя метода. Предположим, происходит вызов метода `ob.MyMeth(args)`, причем неявный параметр `ob` объявлен как экземпляр класса `MyClass`. Заметим, что может существовать несколько методов с именем `MyMeth`, имеющих разные типы параметров (например, `MyMeth(int)` и метод `MyMeth(String)`). Компилятор нумерует все методы с именем `MyMeth` в классе `MyClass` и все общедоступные методы с именем `MyMeth` в суперклассах класса `MyClass`. Теперь компилятор знает всех возможных «кандидатов» при вызове метода.

2. Затем компилятор определяет типы параметров, указанных при вызове метода. Если среди всех методов с именем `MyMeth` есть только один метод, типы параметров которого совпадают с указанными, происходит его вызов. Этот процесс называется разрешением перегрузки. Например, при вызове `ob.MyMeth(«Привет»)` компилятор выберет метод `MyMeth(String)`, а не метод `MyMeth(int)`. Если компилятор не находит ни одного метода с подходящим набором параметров или в результате преобразования типов возникает несколько методов, соответствующих данному вызову, выдается сообщение об ошибке. Теперь компилятор знает имя и типы параметров метода, подлежащего вызову.

3. Если метод является приватным (`private`), статическим (`static`), финальным (`final`) или конструктором, компилятор точно знает, как его вызвать. Такой процесс называется *статическим связыванием*.

В противном случае метод, подлежащий вызову, определяется по фактическому типу неявного параметра и во время выполнения программы используется динамическое связывание. В нашем примере компилятор сгенерировал бы вызов метода `MyMeth(String)` с помощью динамического связывания.

4. Если при выполнении программы для вызова метода используется динамическое связывание, виртуальная машина должна вызвать версию метода, соответствующую фактическому типу объекта, на который ссылается переменная `ob`. Предположим, что объект имеет фактический тип `SuperClass`, являющийся суперклассом класса `MyClass`. Если в классе `SuperClass` определен метод `MyMeth(String)`, то вызывается именно он. Если нет, то поиск метода `MyMeth(String)`, подлежащего вызову, выполняется в суперклассе и т.д.

На поиск вызываемого метода уходит слишком много времени, поэтому виртуальная машина заранее создает для каждого класса таблицу методов, в которой перечисляются сигнатуры всех методов и фактические методы, подлежащие вызову. При вызове метода виртуальная машина просто просматривает таблицу методов. В нашем примере виртуальная машина проверяет таблицу методов класса `SuperClass` и обнаруживает метод `MyMeth(String)`, подлежащий вызову. Такими методами могут быть `SuperClass.MyMeth(String)` или `SomeClass.MyMeth(String)`, если `SomeClass` – некоторый суперкласс класса `SuperClass`.

Описанный поэтапный механизм называется динамическим связыванием. Динамическое связывание обладает одной важной особенностью: оно позволяет модифицировать программы без перекомпиляции их кодов. Это делает программы динамически расширяемыми.

Класс `Object`. Метод `equals()`. Класс `Object` является предком всех классов – каждый класс в языке Java расширяет класс `Object`. Указывать явно расширение класса `Object` с использованием зарезервированного слова `extends` не нужно. Если суперкласс явно не указан, им считается класс `Object`. В языке Java каждый класс расширяет `Object`, поэтому каждый класс обладает некоторой базовой функциональностью, которая предоставлена суперклассом `Object`. Рассмотрим основные возможности.

1. Переменную типа `Object` можно использовать в качестве ссылки на объект любого типа:

```
Object ob = new Human(new Point(100, 100), "Николаев Е.И.", "Юг");
```

При таком использовании объектов суперкласса, переменная `ob` будет использована только для хранения значений произвольного типа (вызывать методы класса `Human` нельзя). Для получения доступа к полям класса `Human` необходимо выполнить процедуру преобразования типа:

```
Human obH = (Human) ob;  
obH.MoveTo(new Point(0.0f, 0.0f));
```

В Java не являются объектами, производными от `Object`, переменные простых типов. Наследование от `Object` распространяется на массивы (как на переменные-массивы, так и на элементы массивов).

2. Класс `Object` предоставляет метод `equals()`. Данный метод проверяет, эквивалентны ли два объекта. Поскольку метод `equals()` реализован в классе `Object`, он определяет лишь, ссылаются ли переменные на один и тот же объект. В качестве проверки по умолчанию эти действия вполне оправданы: всякий объект эквивалентен самому себе. Для некоторых классов большего и не требуется. Однако в ряде случаев эквивалентными должны считаться объекты одного типа, имеющие одинаковые состояния. Например, на рис. 4.9 представлен класс `Order` (Заказ). В классе присутствуют поля, отражающие свойства: номер заказа, дата, товар, количество, цена, покупатель; в структуре класса предусмотрены методы: конструктор, получение общей стоимости заказа, представление заказа.

```

5 public class Order {
6     private int no; // номер заказа
7     private Date dt; // дата заказа
8     private String stuff; // товар
9     private float price; // цена
10    private int count; // количество
11    private String customer; // покупатель
12
13    public Order(int pNo, String pStuff, float pPrice,
14                int pCount, String pCustomer){
15        no = pNo; stuff = pStuff; price = pPrice;
16        count = pCount; customer = pCustomer;
17        dt = new Date();
18    }
19
20    public float getTotalPrice(){
21        return price * count;
22    }
23
24    public String getOrderName(){
25        return "Заказ №" + String.valueOf(no) +
26            " от " + String.valueOf(dt) +
27            "(" + getTotalPrice() + ")";
28    }
29 }

```

Рис. 4.9. Класс Order

На рис. 4.10 продемонстрировано использование объектов данного класса.

```

50 public static void main(String[] args) throws InterruptedException{
51     // Создание первого заказа
52     Order o1 = new Order(1, "Переходник HDMI-DVI",
53         780.00f, 12, "Николаев Е.И.");
54     // Создание заказа идентичного первому
55     Order o1_copy = new Order(1, "Переходник HDMI-DVI",
56         780.00f, 12, "Николаев Е.И.");
57     // Задержка
58     Thread.sleep(5000);
59     // Создание второго заказа
60     Order o2 = new Order(2, "Монитор Samsung L110",
61         7600.00f, 2, "Ирисова А.Ю.");
62     // Вывод информации
63     System.out.println(o1.getOrderName());
64     System.out.println(o1_copy.getOrderName());
65     System.out.println(o2.getOrderName());
66
67     System.out.println(o1.equals(o1_copy));
68     System.out.println(o1.equals(o2));
69 }

```

Рис. 4.10. Использование возможностей объектов типа Order

В результате выполнения данного кода получим результат, представленный на рис. 4.11.

```
Заказ №1 от Thu Feb 12 15:24:17 MSK 2015(9360.0)
Заказ №1 от Thu Feb 12 15:24:17 MSK 2015(9360.0)
Заказ №2 от Thu Feb 12 15:24:22 MSK 2015(15200.0)
false
false
```

Рис. 4.11. Результат сравнения объектов Order

Каждый вызов метода equals() вернул false, так как анализировались ссылки, хранящиеся в переменных o1, o1_copy, o2. Все эти переменные ссылаются на различные области памяти.

Необходимо доработать класс Order для реализации корректного сравнения объектов данного типа. Добавим переопределение метода equals():

```
30- @Override
31- public boolean equals(Object anotherOb){
32-     if(!(anotherOb instanceof Order))
33-         return false;
34-     Order ob = (Order)anotherOb;
35-     if((this.no == ob.no) && this.dt.equals(ob.dt) &&
36-        this.stuff.equals(ob.stuff) && (this.price == ob.price) &&
37-        (this.count == ob.count) && this.customer.equals(ob.customer))
38-         return true;
39-
40-     return false;
41- }
```

Рис. 4.12. Переопределение метода equals()
в подклассе Order суперкласса Object

Вывод программы представлен на рис. 4.13.

```
Заказ №1 от Thu Feb 12 15:43:43 MSK 2015(9360.0)
Заказ №1 от Thu Feb 12 15:43:43 MSK 2015(9360.0)
Заказ №2 от Thu Feb 12 15:43:48 MSK 2015(15200.0)
true
false
```

Рис. 4.13. Результат работы программы с использованием
переопределенного метода equals()

Таким образом, программист может самостоятельно определять бизнес-логику приложения и программировать правила сравнения объектов.

Спецификация языка Java требует, чтобы метод equals() обладал следующими характеристиками.

1. *Рефлексивность*. Для любой ненулевой ссылки ob вызов ob.equals(ob) должен возвращать значение true.

2. *Симметричность*. Для любых ссылок ob1 и ob2 вызов ob1.equals(ob2) должен возвращать значение true тогда и только тогда, когда ob2.equals(ob1) возвращает true.

3. *Транзитивность*. Для любых ссылок ob1, ob2 и ob3, если вызовы ob1.equals (ob2) и ob2.equals(ob3) возвращают значение true, вызов ob1.equals(ob3) возвращает значение true.

4. *Согласованность*. Если объекты, на которые ссылались переменные ob1 и ob2 не изменяются, то повторный вызов ob1.equals (ob2) должен возвращать то же значение.

5. Для любой null-ссылки ob1 вызов ob1.equals(null) должен возвращать значение false.

При реализации метода сравнения программисту важно придерживаться следующих *стадий сравнения объектов в методе equals()*.

1. Предположим, что явный параметр называется anotherOb. После приведения типа – ob (рис. 4.12).

2. Проверить, идентичны ли ссылки this и anotherOb

```
if(this == anotherOb)
    return true;
```

Это выражение используется для оптимизации проверки. Гораздо быстрее проверить идентичность ссылок, чем сравнивать поля объектов.

3. Выяснить, является ли ссылка anotherOb нулевой (null). Если да, вернуть значение false. Эту проверку нужно делать обязательно.

```
if(anotherOb == null)
    return false;
```

4. Сравнить классы this и anotherOb. Если семантика проверки может измениться в подклассе, использовать метод getClass ()

```

if(getClass() != anotherOb.getClass())
    return false;

if(!(anotherOb instanceof Order))
    return false;

```

Если принцип проверки остается справедливым для всех подклассов, использовать операцию instanceof:

```

if(getClass() != anotherOb.getClass())
    return false;

if(!(anotherOb instanceof Order))
    return false;

```

5. Преобразовать объект anotherOb в переменную требуемого класса.

```
Order ob = (Order)anotherOb;
```

6. Далее следует реализовать логику сравнения с использованием требований предметной области. Для полей простых типов используется операция ==, для объектных полей – метод equals().

```

if((this.no == ob.no) && this.dt.equals(ob.dt) &&
    this.stuff.equals(ob.stuff) && (this.price == ob.price) &&
    (this.count == ob.count) && this.customer.equals(ob.customer))
    return true;

```

Метод toString() класса Object. Еще одним важным методом класса Object является toString(), возвращающий значение объекта в виде строки. Например, для объекта класса Order, определенного на рис. 4.9, вызов метода toString() возвратит следующую строку: «lesson4.Order@70dea4e». То есть, если метод суперкласса Object не переопределен в подклассе, то он возвращает строку формата:

<имя_класса>@<адрес_объекта>

Создадим переопределение метода toString() для класса Order:


```
public String getOrderName(){
    return "Заказ №" + String.valueOf(no)
        " от " + String.valueOf(dt) +
        "(" + getTotalPrice() + ")";
}

public boolean equals(Object anotherOb){

@Override
public String toString(){
    return this.getOrderName();
}
```

Рис. 4.14. Переопределение метода toString() класса Order

Таким образом, метод toString() суперкласса, переопределенный в Order, выполняет роль пользовательского метода getOrderName(). Метод getOrderName() может быть удален.






При разработке программ рекомендуется всегда переопределять метод toString() для пользовательских классов: это не только является хорошим стилем программирования, но и позволяет персонифицировать поведение пользовательских классов.

Методика и порядок выполнения работы

1. Спроектируйте иерархию классов приложения в соответствии с предметной областью, описанной в индивидуальном задании. Приведенную иерархию классов в задании необходимо дополнить и расширить тремя дополнительными классами.
2. Реализуйте все спроектированные классы в приложении.
3. Запустите приложение, протестируйте его и исправьте ошибки.
4. Продемонстрируйте полиморфное поведение объектов.

Индивидуальное задание

Перед выполнением задания требуется самостоятельно определить закономерность изменения членов последовательности, чтобы применить цикл, условный оператор или, если потребуется, оператор выбора.

Вариант	Предметная область (иерархия сущностей)
1	 <pre> graph BT ГрузовойАвто --> Автомобиль ЛегковойАвто --> Автомобиль Автомобиль --> ТехСредство Вертолёт --> ТехСредство </pre>
2	 <pre> graph BT Сервер --> КомпТехника Ноутбук --> КомпТехника ПК --> КомпТехника </pre>
3	 <pre> graph BT Квадрат --> Четырехугольник Ромб --> Четырехугольник Четырехугольник --> Геометрия Треугольник --> Геометрия </pre>
4	 <pre> graph BT Книга --> Издание Журнал --> Издание ЭлРесурс --> Издание </pre>
5	 <pre> graph BT Растровое --> Изображение Векторное --> Изображение Изображение --> Мультимедиа Видео --> Мультимедиа </pre>

6	<pre> graph BT S[Стрелковое] --> V[Вооружение] T[Танк] --> S C[Сомолет] --> V </pre>
7	<pre> graph BT A[Авиасредство] --> M[Машины] G[Грузовик] --> M B[Вертолет] --> A I[Истребитель] --> A </pre>
8	<pre> graph BT T[Трава] --> R[Растение] D[Дерево] --> R K[Кустарник] --> R </pre>
9	<pre> graph BT O[ОС] --> P[ПО] Pr[Прикладное] --> P S[СвободныеОС] --> O Pl[ПлатныеОС] --> O </pre>
10	<pre> graph BT M[Мобильный] --> E[ЭВМ] St[Стационарный] --> E P[Планшет] --> M N[Ноутбук] --> M </pre>

11	<pre>graph BT; A[Электроника] --> B[СотТелефон]; A --> C[Фотоаппарат]; A --> D[Планшет];</pre>
12	<pre>graph BT; A[ТехСредство] --> B[Автомобиль]; A --> C[Вертолёт]; B --> D[ГрузовойАвто]; B --> E[ЛегковойАвто];</pre>
13	<pre>graph BT; A[БытТехника] --> B[Телевизор]; A --> C[Утюг]; A --> D[Микроволновка];</pre>
14	<pre>graph BT; A[ЭВМ] --> B[Мобильный]; A --> C[Стационарный]; B --> D[Планшет]; B --> E[Ноутбук];</pre>
15	<pre>graph BT; A[КанцТовары] --> B[Ручка]; A --> C[Органайзер]; A --> D[Скоросшиватель];</pre>

16	<pre> graph BT Вертолет --> Авиасредство Истребитель --> Авиасредство Авиасредство --> Машины Грузовик --> Машины </pre>
17	<pre> graph BT Квартира --> Недвижимость Дом --> Недвижимость Гараж --> Недвижимость </pre>
18	<pre> graph BT Трактор --> Тяжелое Комбайн --> Тяжелое Тяжелое --> Машиностроение Среднее --> Машиностроение </pre>
19	<pre> graph BT Фотон --> ЭлемЧастица Электрон --> ЭлемЧастица Протон --> ЭлемЧастица </pre>
20	<pre> graph BT ПрямоПараллелепипед --> Многоугольник Треугольник --> Многоугольник Многоугольник --> Геометрия Эллипс --> Геометрия </pre>

21	<pre> graph BT П[Программист] --> С[Специалист] В[Врач] --> С Л[Летчик] --> С </pre>
22	<pre> graph BT С[СвободныеОС] --> ОС[ОС] П[ПлатныеОС] --> ОС ОС --> ПО[ПО] ПО_П[Прикладное] --> ПО </pre>
23	<pre> graph BT Н[Накопительная] --> БК[БанкКарта] П[Платежная] --> БК З[Зарплатная] --> БК </pre>
24	<pre> graph BT П[Продажи] --> С[Сервис] Л[Лизинг] --> С К[Кредит] --> С </pre>
25	<pre> graph BT С[Стол] --> М[Мебель] ММ[МягкМебель] --> М Ш[Шкаф] --> М </pre>

Содержание отчета и его форма

Отчет по лабораторной работе должен содержать следующие данные.

1. Номер и название лабораторной работы; задачи.
3. Ответы на контрольные вопросы.
4. Диаграмма классов, экранные формы (консольный вывод) и листинг программного кода с комментариями, показывающие порядок выполнения лабораторной работы, и результаты, полученные в ходе её выполнения.

Отчет о выполнении лабораторной работы, подписанный студентом, сдается преподавателю.

Контрольные вопросы

1. Что такое *наследование реализации*?
2. Опишите графическое представление наследования на диаграммах классов. Опишите синтаксические конструкции для реализации наследования в приложениях Java.
3. Опишите сходства и различия в механизмах наследования в Java и C#.
4. Что такое полиморфизм? Опишите механизм полиморфного поведения объектов Java.
5. Поясните смысл терминов: «суперкласс», «подкласс», «наследование реализации», «класс-наследник», «иерархическая цепочка», «динамическое связывание».
6. Какие возможности предоставляет класс Object библиотеки Java. Опишите назначение его основных методов. Существует ли класс с подобным функционалом в C#?
7. На рис. 4.12 метод equals() класса Order помечен как @Override. Поясните, какая цель подобного аннотирования.

Литература: 2–4.