Глава 4

НАСЛЕДОВАНИЕ И ПОЛИМОРФИЗМ

Если делегированию полномочий уделять внимание, ответственность накопится внизу, подобно осадку.

Закон делегирования Раска

Наследование

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

Подкласс наследует поля и методы суперкласса, используя ключевое слово **extends**. Класс может также реализовать 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, аналог предоставляет реализация интерфейсов, которые не являются классами и содержат описание набора методов, задающих поведение объекта класса, реализующего эти интерфейсы. Наличие общих методов, которые должны быть реализованы в разных классах, обеспечивают им сходную функциональность.

Подкласс дополняет члены суперкласса своими полями и\или методами и\или переопределяет методы суперкласса. Если имена методов совпадают, а параметры различаются, то такое явление называется перегрузкой методов (статическим полиморфизмом).

Если же совпадают имена и параметры методов, то этот механизм называется динамическим полиморфизмом. То есть 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 по ссылке типа объекта называется ***полиморфизмом***.

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

В следующем классе переопределяемый метод **doPayment()** находится в суперклассе **CardAction** и его подклассе **CreditCardAction**. В соответствии с принципом полиморфизма по ссылке вызывается метод наиболее близкий к текущему объекту.

```
/* # 1 # наследование класса и переопределение метода # CardAction.java
 # CreditCardAction.java # CardRunner.java */
```

```
package by.epam.learn.inheritance;
public class CardAction {
  public void doPayment(double amountPayment) {
    System.out.println("complete from debt card");
  }
}
```

```
package by.epam.learn.inheritance;
public class CreditCardAction extends CardAction {
  @Override // is used when override a method in sub class
  public void doPayment(double amountPayment) { // override method
    System.out.println("complete from credit card");
  }
  public boolean checkCreditLimit() { // own method
    return true;
  }
}
```

```
package by.epam.learn.inheritance;
public class CardRunner {
  public static void main(String[] args) {
    CardAction action1 = new CardAction();
    CardAction action2 = new CreditCardAction();
    CreditCardAction cc = new CreditCardAction();
    // CreditCardAction cca = new CardAction(); // compile error: class cast
    action1.doPayment(15.5); // method of CardAction
    action2.doPayment(21.2); // polymorphic method: CreditCardAction
    // dc2.checkCreditLimit(); // compile error: non-polymorphic method
    ((CreditCardAction) action2).checkCreditLimit(); // ок
    cc.doPayment(7.0); // polymorphic method: CreditCardAction
    cc.checkCreditLimit(); // non-polymorphic method CreditCardAction
    ((CreditCardAction) action1).checkCreditLimit(); // runtime error: class cast
  }
}
```

Объект по ссылке **action1** создается при помощи вызова конструктора класса **CardAction** и, соответственно, при вызове метода **doPayment()** вызывается версия метода из класса **CardAction**. При создании объекта **action2** ссылка типа **CardAction** инициализируется объектом типа **CreditCardAction**. 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, т.е. существуют в одном объекте независимо друг от друга. Такое решение является плохим примером кода, который не используется в практическом программировании. Не следует использовать вызов методов, которые можно переопределить, в конструкторе. Это действие может привести к некорректной работе конструктора при инициализации полей объекта и в целом некачественному созданию объекта. Для доступа к полям текущего объекта можно использовать указатель **this**, для доступа к полям суперкласса — указатель **super**.

К чему может привести вызов полиморфных методов в конструкторе и объявление идентичных полей иерархии наследования рассмотрено ниже:

/ # 2 # вызов полиморфного метода из конструктора # Dumb.java # Dumber.java */*

```
package by.epam.learn.inheritance;
class Dumb {
  {
    this.id = 6;
  }
  int id;
  Dumb() {
    System.out.println("constructor Dumb ");
    id = getId(); // ~ this.getId(); // ~ Dumb.this.getId();
    System.out.println(" id=" + id);
  }
  int getId() { // 1
    System.out.println("getId() of Dumb ");
    return id;
  }
}
class Dumber extends Dumb {
  int id = 9;
  Dumber() {
    System.out.println("constructor Dumber");
    id = this.getId();
    System.out.println(" id=" + id);
  }
  @Override
  int getId() { // 2
    System.out.println("getId() of Dumber");
    return id;
  }
}
```

В результате создания экземпляра **Dumb dumb = new Dumber()** последовательно будет выведено:

constructor Dumb
getId() of Dumber
id=0
constructor Dumber
getId() of Dumber
id=9

Метод **getId()** объявлен в классе **Dumb** и переопределен в его подклассе **Dumber**. Перед вызовом конструктора **Dumber()** вызывается конструктор класса **Dumb**. Но так как создается объект класса **Dumber**, то вызывается ближайший к создаваемому объекту метод **getId()**, объявленный в классе **Dumber**, который, в свою очередь, оперирует полем **id**, еще не проинициализированным для класса **Dumber**. В результате **id** получит значение по умолчанию, т.е. ноль.

Разработчик класса **Dumb** предполагал, что объект будет создаваться по его правилам, и метод будет работать так всегда. Но если метод переопределен в подклассе, то, соответственно, и в конструкторе суперкласса будет вызвана переопределенная версия, которая может выполнять совершенно другие действия, и создание объекта пойдет уже по другому пути.

Поля с одинаковыми именами в подклассе не замещаются. Объект подкласса будет иметь в итоге два поля с одинаковыми именами. У разработчика появится проблема их различить. Воспользовавшись преобразованием типов вида **((Dumber) dumb).id**, можно получить доступ к полю **id** из подкласса, но это в случае открытого доступа. Если поле приватное, то эта простая задача становится проблемой.

Практического смысла в одинаковых именах полей в иерархии наследования не просматривается. Это просто ошибка проектирования, которой следует избегать.

Методы, объявленные как **private**, не переопределяются.

/ # 3 # попытка наследования приватного метода # Dumb.java # Dumber.java */*

```
class Dumb {
  private void action() {
    System.out.println("Dumb");
  }
}
class Dumber extends Dumb {
  @Override // compile error
  void action() {
    System.out.println("Dumber");
  }
}
```

Аннотация **@Override** будет выдавать ошибку компиляции из-за того, что она просто не видит переопределяемый метод.

Без этой аннотации код будет компилироваться, только цепочка переопределения будет начинаться с версии метода в подклассе и никак не будет связана с версией метода суперкласса.

Классы и методы final

Запрещено переопределять метод в порожденном классе, если в суперклассе он объявлен со спецификатором **final**:

/ # 4 # попытка переопределения final-метода # MasterCardAction.java # VisaCardAction.java */*

```
package by.epam.learn.inheritance;
public class MasterCardAction extends CreditCardAction{
  @Override// doPayment() cannot be polymorphic
  public final void doPayment(double amountPayment) {
    System.out.println("complete from MasterCard");
  }
}
package by.epam.learn.inheritance;
public class VisaCardAction extends MasterCardAction {
  //@Override
  //public void doPayment(double amountPayment) {// impossible method
  //}
}
```

Если разработчик объявляет метод как **final**, следовательно, он считает, что его версия в этой ветви наследования метода окончательна и переопределению\совершенствованию не подлежит.

Что же общего у **final**-метода со статическим? Версия статического метода жестко определяется на этапе компиляции. И если **final**-метод вызван на объекте класса **MasterCardAction**, в котором он определен, или на объекте любого его подкласса, например, **VisaCardAction**, то также в точке вызова будет зафиксирован вызов именно этой версии метода.

/ # 5 # вызов полиморфного метода # CardService.java */*

```
public class CardService {
  public void doService(CardAction action, double amount){
    action.doPayment(amount);
  }
}
```

При передаче в метод **doService()** объектов классов **CardAction** или **CreditCardAction** будут вызваны их собственные версии методов **doPayment()**, определенные в каждом из классов, что представляет собой еще одну иллюстрацию полиморфизма.

При передаче же в метод **doService()** объектов классов **MasterCardAction** или **VisaCardAction** будет вызвана версия метода **doPayment()**, определенная в классе **MasterCardAction**, так как в классе **VisaCardAction** метод не определен, то будет вызвана версия метода ближайшая по восходящей цепочке наследования.

Применение **final**-методов также показательно при разработке конструктора класса. Процесс инициализации экземпляра должен быть строго определен и не подвергаться изменениям. Исключить подмену реализации метода, вызываемого в конструкторе, следует объявлением метода как **final**, т.е. при этом метод не может быть переопределен в подклассе. Подобное объявление гарантирует обращение именно к этой реализации. Корректное определение вызова метода класса из конструктора представлено ниже:

/ # 6 # вызов нестатического final-метода из конструктора # AutenticationService.java */*

```
package by.epam.learn.service;
public class AutenticationService {
  public AutenticationService() {
    authenticate();
  }
  public final void authenticate() {
    //appeal to the database
  }
}
```

Если убрать **final** в объявлении метода суперкласса, то становится возможным переопределить метод в подклассе.

/ # 7 # переопределение не final-метода # NewAutenticationService.java */*

```
package by.epam.learn.service;
public class NewAutenticationService extends AutenticationService {
  @Override
  public void authenticate() {
    //empty method
  }
}
```

Тогда при создании объекта подкласса конструктор суперкласса вызовет версию метода из подкласса как самую близкую по типу создаваемого объекта, и проверка данных в БД не будет выполнена.

Рекомендуется при разработке классов из конструкторов вызывать методы, на которые не распространяются принципы полиморфизма. Метод может быть еще объявлен как **private** или **static** с таким же результатом.

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

```
public final class String { /* code */ }
class MegaString extends String { /* code */ } //impossible: compile error
```

Если необходимо создать собственную реализацию с возможностями **final**-класса, то создается класс-оболочка, где в качестве поля представлен **final**-класс. В свою очередь, необходимые или даже все методы делегируются из **final**-класса, и им придается необходимая разработчику функциональность.

Такой подход гарантирует невозможность прямого использования класса-оболочки вместо оборачиваемого класса и наоборот.

// # 8 # класс-оболочка для класса String # WrapperString.java

```
package by.epam.learn.string;
public class WrapperString {
  private String str;
  public WrapperString() {
    str = new String();
  }
  public WrapperString(String str) {
    this.str = str;
  }
  public int length() {// delegate method
    return str.length();
  }
  public boolean isEmpty() {// delegate method
    return str.isEmpty();
  }
  public int indexOf(int arg) { // delegate method
    int value = // new realization
    return value;
  }
  // other methods
}
```

Класс **WrapperString** не является наследником класса **String**, и его объект не может быть использован для передачи по ссылке на класс **String**. Класс **WrapperString**, в свою очередь, уже может быть суперклассом, поэтому его поведение можно изменять.

Использование super и this

Ключевое слово **super** применяется для обращения к конструктору суперкласса и для доступа к полю или методу суперкласса. Например:

```
super(parameters); // call to superclass constructor
super.id = 42; // superclass attribute reference
super.getId(); // superclass method call
```

Первая форма **super** применяется только в конструкторах для обращения к конструктору суперкласса только в качестве первой строки кода конструктора и только один раз.

Вторая форма **super** используется для доступа из подкласса к переменной **id** суперкласса. Третья форма специфична для Java и обеспечивает вызов из подкласса метода суперкласса, что позволяет избежать рекурсивного вызова в случае, если вызываемый с помощью **super** метод переопределен в данном подклассе. Причем, если в суперклассе этот метод не определен, то будет осуществляться поиск по цепочке наследования до тех пор, пока он не будет найден.

Во всех случаях с использованием **super** можно обратиться только к ближайшему суперклассу, т.е. «перескочить» через суперкласс, чтобы обратиться к его суперклассу, невозможно.

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

// # 9 # super и this в конструкторе # Point1D.java # Point2D.java # Point3D.java

```
package by.epam.learn.point;
public class Point1D {
  private int x;
  public Point1D(int x) {
    this.x = x;
  }
}
package by.epam.learn.point;
public class Point2D extends Point1D {
  private int y;
  public Point2D(int x, int y) {
    super(x);
    this.y = y;
  }
}
package by.epam.learn.point;
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); // call Point3D constructor with parameters
  }
}
```

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, если в методе объявлены локальные переменные с тем же именем, что и переменные экземпляра класса. Локальная переменная имеет

преимущество перед полем класса и закрывает к нему доступ. Чтобы получить доступ к полю класса, требуется воспользоваться явной ссылкой **this** перед именем поля, так как поле класса является частью объекта, а локальная переменная нет.

Инструкция **this()** должна быть единственной в вызывающем конструкторе и быть первой по счету выполняемой операцией, иначе возникает возможность вызова нескольких конструкторов суперкласса или ветвления при обращении к конструктору суперкласса. Компилятор выполнять подобные действия запрещает.

Возможна ситуация с зацикливанием обращений конструкторов друг к другу, что также запрещено:

// # 10 # ошибка с зацикливанием вызова конструктора # Point1D.java

```
public class Point1D {
  private int x;

  public Point1D(int x) { // recursive constructor invocation
    this();
    this.x = x;
  }
  public Point1D() { // recursive constructor invocation
    this(42);
  }
}
```

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

```
/* # 11 # динамическое связывание методов # Point1D.java # Point2D.java
 # Point3D.java # PointReport.java # PointMain.java */
package by.epam.learn.point;
public class Point1D {
  private int x;
  public Point1D(int x) {
    this.x = x;
  }
  public double length() {
    return Math.abs(x);
  }
  @Override
  public String toString() {
    return " x=" + x;
  }
}
package by.epam.learn.point;
public class Point2D extends Point1D {
  private int y;
  public Point2D(int x, int y) {
    super(x);
    this.y = y;
  }
  @Override
  public double length() {
    return Math.hypot(super.length(), y);
    /* just length() is impossible, because the method will call itself, which will
       lead to infinite recursion and an error at runtime */
  }
  @Override
  public String toString() {
    return super.toString() + " y=" + y;
  }
}
package by.epam.learn.point;
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);
  }
```

```
  @Override
  public double length() {
    return Math.hypot(super.length(), z);
  }
  @Override
  public String toString() {
    return super.toString() + " z=" + z;
  }
}
package by.epam.learn.point;
public class PointReport {
  public void printReport(Point1D point) {
    System.out.printf("length=%.2f %s%n", point.length(), point);
    // out.print(point.toString()) is identical to out.print(point)
  }
}
package by.epam.learn.point;
public class PointMain {
  public static void main(String[] args) {
    PointReport report = new PointReport();
    Point1D point1 = new Point1D(-7);
    report.printReport(point1);
    Point2D point2 = new Point2D(3, 4);
    report.printReport(point2);
    Point3D point3 = new Point3D(3, 4, 5);
    report.printReport(point3);
  }
}
```

Результат:

length=7.00 x=-7 length=5.00 x=3 y=4 length=7.07 x=3 y=4 z=5

Аннотация **@Override** позволяет выделить в коде переопределенный метод и сгенерирует ошибку компиляции в случае, если программист изменит имя метода, типы его параметров или их количество в описании сигнатуры полиморфного метода.

Следует помнить, что при вызове метода обращение **super** всегда производится к ближайшему суперклассу. Переадресовать вызов, минуя суперкласс, невозможно! 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 нестатические методы Java являются виртуальными (ключевое слово virtual, как в C++, не используется).

Статические методы можно перегружать и «переопределять» в подклассах, но их доступность всегда зависит от типа ссылки и атрибута доступа, и никогда — от типа самого объекта.

Методы подставки

После выхода 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 тип возвращаемого значения метода суперкласса.

/ # 12 # методы-подставки # Point1DCreator.java # Point2DCreator.java # PointMain.java*/*

```
package by.epam.learn.inheritance.service;
import by.epam.learn.point.Point1D;
import java.util.Random;
public class Point1DCreator {
  public Point1D create() {
    System.out.println("log in Point1DCreator");
    return new Point1D(new Random().nextInt(100));
  }
}
package by.epam.learn.inheritance.service;
import by.epam.learn.point.Point2D;
import java.util.Random;
public class Point2DCreator extends Point1DCreator {
  @Override
  public Point2D create() {
    System.out.println("log in Point2DCreator");
    Random random = new Random();
    return new Point2D(random.nextInt(10), random.nextInt(10));
  }
}
```

```
package by.epam.learn.inheritance;
import by.epam.learn.inheritance.service.Point1DCreator;
import by.epam.learn.inheritance.service.Point2DCreator;
import by.epam.learn.point.Point1D;
import by.epam.learn.point.Point2D;
public class PointMain {
  public static void main(String[] args) {
    Point1DCreator creator1 = new Point2DCreator();
//  Point2D point = creator1.create(); // compile error
    Point1D pointA = creator1.create(); /* when compiling - overload,
                                          when running – overriding */
    System.out.println(pointA);
    Point2DCreator creator2 = new Point2DCreator();
    Point2D pointB = creator2.create();
    System.out.println(pointB);
  }
}
```

В обоих случаях создания объекта будут созданы объекты класса **Point2D**:

```
log in Point2DCreator
 x=5 y=4
log in Point2DCreator
 x=2 y=7
```

В данной ситуации при компиляции в подклассе **Point2DCreator** создаются два метода **create()**. Один имеет возвращаемое значение **Point2D**, другой (явно невидимый) — **Point1D**. При обращении к методу **create()** версия метода определяется «ранним связыванием» без использования полиморфизма, но при выполнении срабатывает полиморфизм и вызывается метод с возвращаемым значением **Point2D**.

Получается, что на этапе компиляции метод-подставка ведет себя как обычный метод, видимый по типу ссылки, а на этапе выполнения включается механизм переопределения и срабатывает метод из подкласса.

На практике методы подставки могут использоваться для расширения возможностей класса по прямому извлечению (без преобразования) объектов подклассов, инициализированных в ссылке на суперкласс.

«Переопределение» статических методов

Для статических методов принципы «позднего связывания» не работают. Динамический полиморфизм к статическим методам класса неприменим, так как обращение к статическому атрибуту или методу осуществляется по типу ссылки, а не по типу объекта, через который производится обращение. Версия вызываемого статического метода всегда определяется на этапе компиляции. При использовании ссылки для доступа к статическому члену компилятор при выборе метода учитывает тип ссылки, а не тип объекта, ей присвоенного.

/ # 13 # поведение статического метода при «переопределении» # StaticMain.java */*

```
class StaticDumb {
  public static void go() {
    System.out.println("go() from StaticDumb ");
  }
}
class StaticDumber extends StaticDumb {
  //@Override - compile error
  public static void go() { // similar to dynamic polymorphism
    System.out.println("go() from StaticDumber ");
  }
}
class StaticMain {
  public static void main(String[ ] args) {
    StaticDumb dumb = new StaticDumber();
```

```
    dumb.go(); // warning: static member accessed via instance reference
    StaticDumber dumber = null;
    dumber.go(); // will not NullPointerException !
  }
}
```

В результате выполнения данного кода будет выведено:

go() from StaticDumb
go() from StaticDumber

При таком способе инициализации объекта **dumb** метод **go()** будет вызван из класса **StaticDumb**. Если же спецификатор **static** убрать из объявления методов, то вызов методов будет осуществляться в соответствии с принципами полиморфизма.

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

```
StaticDumb.go();
StaticDumber.go();
```

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

Абстракция

Множество моделей предметов реального мира обладают 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. Абстрактный класс может и не содержать вовсе абстрактных методов. Предназначение такого класса — быть вершиной иерархии его различных реализаций.

Объекты таких классов нельзя создать с помощью оператора **new**, но можно создать объекты подклассов, которые реализуют все 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. Объект класса **Number** нельзя создать явно при помощи его собственного конструктора.

/ # 14 # абстрактный класс и метод # AbstractCardAction.java */*

```
package by.epam.learn.inheritance;
public abstract class AbstractCardAction {
  private long actionId;
  public AbstractCardAction() {
  }
  public void check() {
  }
  public abstract void doPayment(double amountPayment);
}
```

/ # 15 # подкласс абстрактного класса # CreditCardAction.java # */*

```
package by.epam.learn.inheritance;
public class CreditCardAction extends AbstractCardAction {
  @Override
  public void doPayment(double amountPayment) {
    // code
  }
}
```

Ссылка **action** на абстрактный суперкласс инициализируется объектом подкласса, в котором реализованы все абстрактные методы суперкласса:

```
AbstractCardAction action;
// action = new AbstractCardAction(); //compile error: cannot create object!
action = new CreditCardAction();
action.doPayment(7);
```

С помощью этой ссылки могут вызываться также и неабстрактные методы абстрактного класса, если они не переопределены в подклассе:

```
action.check();
```

Полиморфизм и расширение функциональности

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

В следующем примере приведение к базовому типу происходит в выражении:

```
AbstractQuest quest1 = new DragnDropQuest();
AbstractQuest quest2 = new SingleChoiceQuest();
```

Суперкласс **AbstractQuest** предоставляет общий интерфейс (методы) для своих подклассов. Подклассы **DragnDropQuest** и **SingleChoiceQuest** переопределяют эти определения для обеспечения уникального поведения.

Небольшое приложение демонстрирует возможности полиморфизма.

/ # 16 # общий пример на полиморфизм # AbstractQuest.java # DragnDropQuest.java # SingleChoiceQuest.java # Answer.java # QuestFactory.java # QuizMain.java */*

```
package by.epam.learn.inheritance.quiz;
public abstract class AbstractQuest {
  private long questId;
  private String content;
  // constructors, methods
  public abstract boolean check(Answer answer);
}
package by.epam.learn.inheritance.quiz;
import java.util.Random;
public class DragnDropQuest extends AbstractQuest {
  // constructors, methods
  @Override
  public boolean check(Answer answer) {
    return new Random().nextBoolean(); // demo
  }
}
package by.epam.learn.inheritance.quiz;
import java.util.Random;
public class SingleChoiceQuest extends AbstractQuest {
  // constructors, methods
  @Override
  public boolean check(Answer answer) {
    return new Random().nextBoolean();
  }
}
package by.epam.learn.inheritance.quiz;
```

```
public class Answer {
  // fields, constructors, methods
}
package by.epam.learn.inheritance.quiz;
public class QuestFactory { // pattern Factory Method (simplest)
  public static AbstractQuest getQuestFromFactory(int mode) {
    switch (mode) {
      case 0:
        return new DragnDropQuest();
      case 1:
        return new SingleChoiceQuest();
      default:
        throw new IllegalArgumentException("illegal mode");
        // assert false; // bad
        // return null; // ugly
    }
  }
}
package by.epam.learn.inheritance.quiz;
import java.util.Random;
public class TestGenerator {
  public AbstractQuest[] generateTest(final int NUMBER_QUESTS, int maxMode) {
    AbstractQuest[] test = new AbstractQuest[NUMBER_QUESTS];
    for (int i = 0; i < test.length; i++) {
      int mode = new Random().nextInt(maxMode); // stub
      test[i] = QuestFactory.getQuestFromFactory(mode);
    }
    return test;
  }
}
package by.epam.learn.inheritance.quiz;
public class TestAction {
  public int checkTest(AbstractQuest[] test) {
    int counter = 0;
    for (AbstractQuest s : test) {
      // вызов полиморфного метода
      counter = s.check(new Answer()) ? ++counter : counter;
    }
    return counter;
  }
}
package by.epam.learn.inheritance.quiz;
public class QuizMain {
  public static void main(String[] args) {
    TestGenerator generator = new TestGenerator();
    AbstractQuest[] test = generator.generateTest(60, 2); // 60 questions of 2 types
    // here should be the code of the test process …
    TestAction action = new TestAction();
    int result = action.checkTest(test);
    System.out.println(result + " correct answers, " + (60 - result) + " incorrect");
  }
}
```

В процессе выполнения приложения будет случайным образом сформирован массив-тест из вопросов разного типа, будет выполнена проверка теста, а общая информация об ответах на них будет выведена на консоль:

27 correct answers, 33 incorrect

Класс **QuestFactory** содержит метод **getQuestFromFactory(int numMode)**, который возвращает ссылку на случайно выбранный объект подкласса класса **AbstractQuest** каждый раз, когда он вызывается. Приведение к базовому типу производится оператором **return**, который возвращает ссылку на **DragnDropQuest** или **SingleChoiceQuest**. Метод **main()** содержит массив из ссылок **AbstractQuest**, заполненный с помощью вызова **getQuestFromFactory()**. 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 **check()** вызывается для каждого случайным образом выбранного объекта.

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

Невозможно приравнивать ссылки на классы, находящиеся в разных ветвях наследования, так как не существует никакого способа привести один такой тип к другому.

Класс Object

На вершине иерархии классов находится класс **Object**, суперкласс для всех классов. Изучение класса **Object** и его методов необходимо, т.к. его свойствами обладают все классы Java. Ссылочная переменная типа **Object** может указывать на объект любого другого класса, на любой массив, так как массив реализован как класс-наследник **Object**.

В классе **Object** определен набор методов, который наследуется всеми классами:

- **protected Object clone()** — создает и возвращает копию вызывающего объекта;
- **public boolean equals(Object ob)** — предназначен для использования и переопределения в подклассах с выполнением общих соглашений о сравнении содержимого двух объектов одного и того же типа;
- **public Class<? extends Object> getClass()** — возвращает экземпляр типа **Class**;
- **protected void finalize()** — (deprecated) автоматически вызывается сборщиком мусора (garbage collection) перед уничтожением объекта;
- **public int hashCode()** — вычисляет и возвращает хэш-код объекта (число, в общем случае вычисляемое на основе значений полей объекта);
- **public String toString()** — возвращает представление объекта в виде строки.

Статический импорт

При вызове статических методов и обращении к статическим константам приходится использовать в качестве префикса имя класса, что утяжеляет код и снижает скорость его восприятия.

// # 24 # обращение к статическому методу и константе # ImportMain.java

```
package by.epam.learn.demo;
public class ImportMain {
  public static void main(String[ ] args) {
    System.out.println(2 * Math.PI * 3);
    System.out.println(Math.floor(Math.cos(Math.PI / 3)));
  }
}
```

Статические константы и статические методы класса можно использовать без указания принадлежности к классу, если применить статический импорт,

```
import static java.lang.Math.*;
```

как это показано ниже.

// # 25 # статический импорт методов и констант # ImportLuxMain.java

```
package by.epam.learn.demo;
import static java.lang.Math.*;
public class ImportLuxMain {
  public static void main(String[ ] args) {
    System.out.println(2 * PI * 3);
    System.out.println(floor(cos(PI / 3)));
  }
}
```

Если необходимо получить доступ только к одной статической константе или методу, то импорт производится в следующем виде:

```
import static java.lang.Math.E; // for one constant
import static java.lang.Math.cos; // for one method
```

Рекомендации при проектировании иерархии

При построении иерархии необходимо помнить, что отношение между классами можно выразить как «is-a», или «является». *Студент* «является» *Человеком*. Поля класса находятся с классом в отношении «has-a», или «содержит». *Студент* «содержит» *номер зачетной книжки*.

Наследование и переопределение методов используются для реализации отличий поведения. Если наследование можно заменить агрегацией, то следует

так и поступить. Нет смысла создавать подкласс *Студент-заочник*, если можно в подкласс *Студент* добавить поле *Форма обучения*.

При наследовании в новые классы добавляются новые возможности в виде полей и методов или переопределения методов. Если новых возможностей не обнаруживается, то использование наследования, как правило, не имеет для этого оснований.

Базовая функциональность должна определяться в вершине иерархии проектируемых классов. Если в подклассе добавляются новые методы, характеризующие поведение иерархии в целом, следует заняться перепроектированием. Но нельзя учесть все возможные изменения иерархии в процессе разработки. Избыточный функционал придется поддерживать. Лучше на поздних стадиях добавить методы в суперкласс, чем пытаться понять, зачем нужен метод, который никто еще не использовал.

Не использовать значения переменных для характерного изменения поведения. Для этих целей следует создать подкласс и переопределить метод.

Для различных семантических сущностей не создавать общую иерархию, даже если действия для них идентичны по форме. Блокировка *Теста* и блокировка *Студента* — действия суть похожие, но отличные по своему функционалу и последствиям.

Вопросы к главе 4

1. Принципы ООП.
2. Правила переопределения метода boolean **equals(Object o)**.
3. Зачем переопределять методы **hashCode()** и **equals()** одновременно?
4. Написать метод **equals()** для класса, содержащего одно поле типа **String**.
5. Правила переопределения метода **int hashCode()**. Можно ли в качестве результата возвращать константу?
6. Правила переопределения метода **clone()**.
7. Чем отличаются **finally** и *finalize*? Для чего используется ключевое слово **final**?
8. JavaBeans: основные требования к классам Bean-компонентов, соглашения об именах.
9. Как работает *Garbage Collector*. Какие самые распространенные алгоритмы? Можно ли самому указать сборщику мусора, какой объект удалить из памяти.
10. В каких областях памяти хранятся значения и объекты, массивы?
11. Чем является класс **Object**? Перечислить известные методы класса **Object**, указать их назначение.
12. Что такое хэш-значение? Объяснить, почему два разных объекта могут сгенерировать одинаковые хэш-коды?
13. Для чего используется наследование классов в java-программе? Привести пример наследования. Поля и методы, помеченные модификатором доступа **private**, наследуются?

14. Как вызываются конструкторы при создании объекта производного класса? Что в конструкторе класса делает оператор **super()**?
15. Возможно ли в одном конструкторе использовать операторы **super()** и **this()**?
16. Объяснить утверждения: «ссылка базового класса может ссылаться на объекты своих производных типов» и «объект производного класса может быть использован везде, где ожидается объект его базового типа». Верно ли обратное и почему?
17. Что такое переопределение методов? Зачем оно нужно? Можно ли менять возвращаемый тип при переопределении методов? Можно ли менять атрибуты доступа при переопределении методов? Можно ли переопределить методы в рамках одного класса?
18. Определить правило вызова переопределенных методов. Можно ли статические методы переопределить нестатическими и наоборот?
19. Какие свойства имеют финальные методы и финальные классы? Зачем их использовать?
20. Какие применяются правила приведения типов при наследовании. Записать примеры явного и неявного преобразования ссылочных типов. Объяснить, какие ошибки могут возникать при явном преобразовании ссылочных типов.
21. Что такое объект класса **Class**? Чем использование метода **getClass()** и последующего сравнения возвращенного значения с **Type.class** отличается от использования оператора **instanceof**?
22. Что такое абстрактные классы и методы? Зачем они нужны? Бывают ли случаи, когда абстрактные методы содержат тело? Можно ли в абстрактных классах определять конструкторы? Могут ли абстрактные классы содержать неабстрактные методы? Можно ли от абстрактных классов создавать объекты и почему?
23. Для чего служит интерфейс **Cloneable**? Как правильно переопределить метод **clone()** класса **Object**, для того чтобы объект мог создавать свои адекватные копии?
24. Что такое перечисления в Java. Как объявить перечисление? Чем являются элементы перечислений? Кто и когда создает экземпляры перечислений?
25. Могут ли перечисления реализовывать интерфейсы или содержать абстрактные методы? Могут ли перечисления содержать статические методы?
26. Можно ли самостоятельно создать экземпляр перечисления? А ссылку типа перечисления? Как сравнить, что в двух переменных содержится один и тот же элемент перечисления и почему именно так?
27. Что такое параметризованные классы? Для чего они необходимы? Привести пример параметризованного класса и пример создания объекта параметризованного класса.
28. Ссылки какого типа могут ссылаться на объекты параметризованных классов? Можно ли создать объект, параметризовав его примитивным типом данных?

29. Какие ограничения на вызов методов существуют у параметризованных полей? Как эти ограничения снимает использование при параметризации ключевого слова **extends**?
30. Как параметризуются статические методы, как определяется конкретный тип параметризованного метода? Можно ли методы экземпляра класса параметризовать отдельно от параметра класса, и если «да», то как тогда определять тип параметра?
31. Что такое *wildcard*? Привести пример его использования?
32. Для чего используется параметризация <**? extends** *Type*>, <**? super** *Type*>?

Задания к главе 4

Вариант А

Создать приложение, удовлетворяющее требованиям, приведенным в задании. Наследование применять только в тех заданиях, в которых это логически обосновано. Аргументировать принадлежность классу каждого создаваемого метода и корректно переопределить для каждого класса методы **equals()**, **hashCode()**, **toString()**.

1. Создать объект класса **Текст**, используя классы **Предложение**, **Слово**. Методы: дополнить текст, вывести на консоль текст, заголовок текста.
2. Создать объект класса **Автомобиль**, используя классы **Колесо**, **Двигатель**. Методы: ехать, заправляться, менять колесо, вывести на консоль марку автомобиля.
3. Создать объект класса **Самолет**, используя классы **Крыло**, **Шасси**, **Двигатель**. Методы: летать, задавать маршрут, вывести на консоль маршрут.
4. Создать объект класса **Государство**, используя классы **Область, Район**, **Город**. Методы: вывести на консоль столицу, количество областей, площадь, областные центры.
5. Создать объект класса **Планета**, используя классы **Материк**, **Океан**, **Остров**. 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**, **Оперативная память, Процессор**. 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 **Щенок**, используя классы **Животное**, **Собака**. Методы: вывести на консоль имя, подать голос, прыгать, бегать, кусать.
11. Создать объект класса **Наседка**, используя классы **Птица**, **Воробей**. Методы: летать, петь, нести яйца, высиживать птенцов.
12. Создать объект класса **Текстовый файл**, используя классы **Файл**, **Директория**. Методы: создать, переименовать, вывести на консоль содержимое, дополнить, удалить.
13. Создать объект класса **Одномерный массив**, используя классы **Массив**, **Элемент**. 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 **Цветок**, используя классы **Лепесток**, **Бутон**. Методы: расцвести, завять, вывести на консоль цвет бутона.
17. Создать объект класса **Дерево**, используя классы **Лист**, **Ветка**. Методы: зацвести, опасть листьям, покрыться инеем, пожелтеть листьям.
18. Создать объект класса **Пианино**, используя классы **Клавиша**, **Педаль**. Методы: настроить, играть на пианино, нажимать клавишу.
19. Создать объект класса **Фотоальбом**, используя классы **Фотография**, **Страница**. 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 классы **Крылья**, **Клюв**. Методы: летать, садиться, питаться, атаковать.
23. Создать объект класса **Хищник**, используя классы **Когти**, **Зубы**. Методы: рычать, бежать, спать, добывать пищу.
24. Создать объект класса **Гитара**, используя класс **Струна**, **Скворечник**. Методы: играть, настраивать, заменять струну.

Вариант В

Создать консольное приложение, удовлетворяющее следующим требованиям:

- Использовать возможности ООП: классы, наследование, полиморфизм, инкапсуляция.
- Каждый класс должен иметь отражающее смысл название и информативный состав.