

## Раздел 6

# Перегрузка, переопределение, тип времени исполнения и объектно ориентированное программирование

### Цели

Этот раздел поможет Вам подготовиться к экзамену, раскрывая следующие темы:

Как определять внутренние классы и классы верхнего уровня, как правильно использовать все допустимые модификаторы (такие как `public`, `final`, `static`, `abstract` и так далее). Понимать значение каждого из этих модификаторов, как по отдельности так и в комбинации, а так же знать какой эффект это возымеет на элементы пакета и их связь.

Для данного класса, знать, как определить будет ли создан конструктор по умолчанию, и если да, то осознавать прототип этого конструктора.

Как распознать правильно сконструированное объявление класса (всех форм включая внутренние классы), объявление интерфейсов и их реализаций.

Понимать преимущества инкапсуляции в объектно-ориентированном подходе, и писать код, который будет воплощать плотно инкапсулированные классы и связи типа «Является ли?» и «Имеет ли?».

Знать, как пишется код вызывающий перегруженные или переопределенные методы и родительские или переопределенные конструкторы. А так же уметь описать эффект от вызова этих методов.

Знать, как писать код для создания объектов любого конкретного класса, включая обычные классы верхнего уровня, внутренние классы, статические внутренние и анонимные классы.

## **Обзор**

<b>Объектно-ориентированное программирование и Java</b>	<b>133</b>
<b>Декларирование классов</b>	<b>137</b>
<b>Конструкторы</b>	<b>138</b>
<b>Декларирование интерфейсов</b>	<b>141</b>
<b>Внутренние и анонимные классы</b>	<b>142</b>
<b>Перегрузка методов</b>	<b>149</b>
<b>Переопределение методов</b>	<b>149</b>

## **Стратегия обучения**

По мере изучения этого раздела Вы должны сконцентрироваться на следующих ключевых моментах:

- Как объявляются классы и интерфейсы
- Как объявляются внутренние и анонимные классы и как они используются
- Как объявляются конструкторы
- Когда создается конструктор по умолчанию
- Как один конструктор может вызывать другой из этого класса или его суперкласса
- Как переопределять и перегружать методы
- Какие ограничения накладываются на переопределенный метод
- Как классы обуславливают использование наследования типа «Является» и «Имеет»
- Как выглядит полностью инкапсулированный класс

## **Введение**

Так как Java является объектно-ориентированным языком программирования, сертификационный экзамен требует от Вас знания базовых концепций и технологий объектно-ориентированного программирования. В этой главе Вы познакомитесь с концепциями объектно-ориентированного программирования и узнаете, как создавать классы и интерфейсы, а так же как использовать внутренние и анонимные классы. Вы также узнаете, как объявлять и использовать конструкторы, перегружать и переопределять методы.

## **Объектно-ориентированное программирование и Java**

Одной из множества особенностей Java, является обширная поддержка объектно-ориентированного программирования. Java предоставляет все базовые элементы ООП: повторное использование объектов, иерархию классов, наследование, инкапсуляцию, полиморфизм, и динамическое связывание – в контексте программирования это полезно и эффективно. Следующие параграфы раскрывают эти возможности.

### **Структура объектов и повторное использование**

Тот факт, что объект может быть создан или сконструирован из другого, является сердцем объектно-ориентированного программирования. Это позволяет более сложным объектам быть построенными из базовых блоков. Например, предположим, что вы строите графический редактор. Ваше приложение будет состоять из таких объектов как: окна, меню, холсты для рисования, палитры цветов и инструментов и т.д. Некоторые из этих объектов доступны в библиотеке классов, а некоторые могут быть построены из примитивных компонентов. Вы разрабатываете Ваше приложение, собирая и строя его классы, а затем монтируете их в единое целое.

Объектная структура не только позволяет Вам упростить организацию в Ваших программах, но также позволяет повторно использовать их для дальнейших разработок. Например, Вы можете разработать классы как часть Вашего графического редактора, а затем использовать их в разработке настольной издательской системы. Вы можете также создать пакет из этих классов и передать

или продать их кому-нибудь, кто будет использовать их как базу для своих приложений. Повторное использование объектов предоставляет Вам возможность построить библиотеку классов (наподобие Java API) из которой Вы можете быстро и легко собирать Ваши программы. Без повторного использования классов, Вы будете вынуждены начинать с сырых набросков каждую, разрабатываемую Вами, программу.

## **Классификация и наследование**

Повторное использование не ограничивается композицией объекта. Оно также используется мощнейшей способностью объектно-ориентированного программирования известной как наследование. Наследование не только позволяет объектам быть использованными, как есть, но и предоставляет возможность создавать новые объекты, расширяя и подстраивая под себя уже существующие.

Классификация это способ, с помощью которого мы обычно организуем полученные знания. Мы используем классификацию независимо от того, занимаемся ли мы объектно-ориентированным программированием или нет. Когда мы встречаем новый объект в нашей повседневной жизни, мы пытаемся подогнать его под существующую классификационную схему. Если он подходит под существующую категорию – мы понимаем, с чем мы имеем дело. Если нет – мы создаем новую категорию.

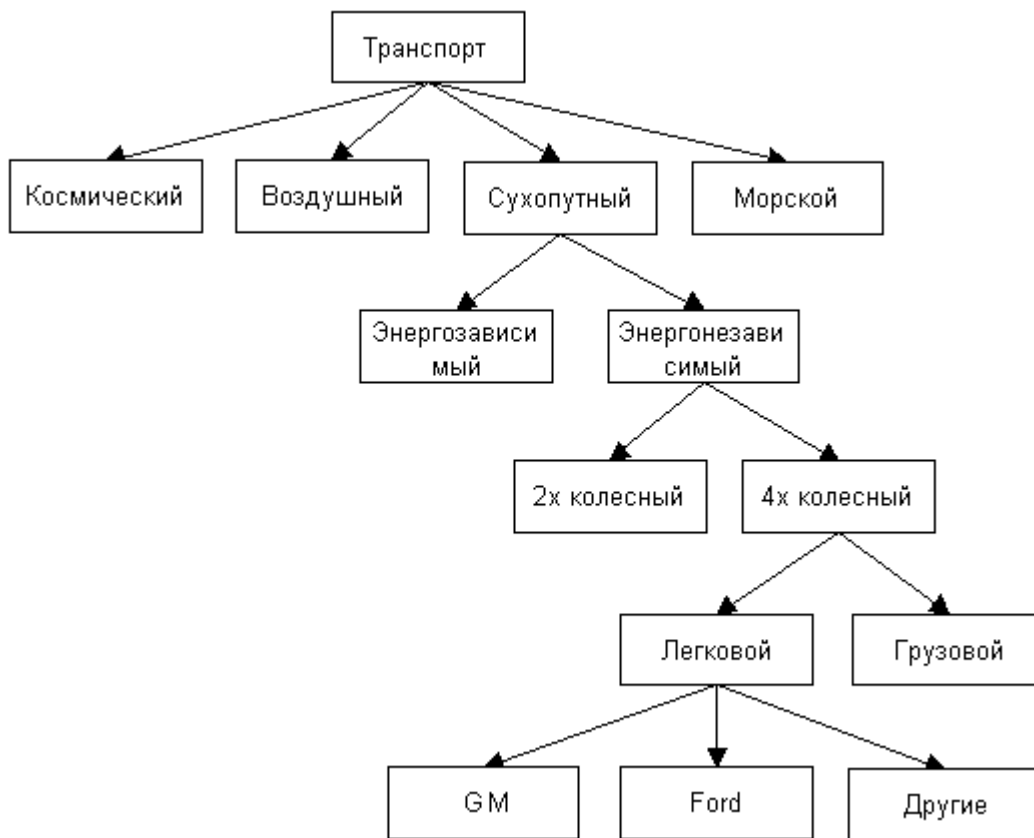
Рисунок 6.1 иллюстрирует, как мы используем классификацию для представления знаний. Когда мы классифицируем объекты в иерархической манере, класс на вершине дерева включает все классы под ним. Если класс присутствует в классификационном дереве, он обладает всеми свойствами всех категорий под ним в этом дереве.

Рисунок 6.2 представляет классификационное дерево различных типов транспортных средств. Все категории в дереве ниже машины, например, разделяют такие характеристики как, четыре колеса, энерго-автономность, и предназначение для пассажирских перевозок. Тот факт, что категории низшего уровня разделяют характеристики категорий над ними в классификационном дереве, известен как наследование. Говорят, что категории низшего уровня наследуют характеристики категорий расположенных в дереве над ними.

В Java, наследование применимо к связи между классом и подклассом. Когда класс X расширяет класс Y, он наследует все (не private) переменные и методы Y. Это мощная особенность повторного использования – не только позволяющая использовать уже существующий класс в его первоначальном виде, но и позволяющая легко расширять и настраивать его, добавляя данные и методы в последующие подклассы.



**Рисунок 6.1 Иерархическая классификация знаний**



**Рисунок 6.2 Классификационное дерево «Транспорта»**

**Примечание**

Когда один класс расширяет другой, он наследует данные и методы расширяемого класса. Этот процесс известен как одиночное наследование. В некоторых языках, но не в Java, допускается множественное наследование, когда класс расширяет несколько классов из разных ветвей в дереве иерархии. Т.к. в Java каждый класс может иметь только один родительский класс, Java не поддерживает множественное наследование.

## Инкапсуляция

Одной из разрекламированных особенностей объектно-ориентированного программирования, является *инкапсуляция*. Этот термин означает объект, заключенный в контейнер. Инкапсуляция представляет собой комбинацию данных и кода, манипулирующего этими данными, заключенную в единый компонент – объект. Инкапсуляции также приписывают функции контроля доступа к реализации компонента. Доступ к объекту ограничен, хорошо определенным интерфейсом доступа. Это позволяет объектам быть самодостаточными и защищенными от случайных инцидентов, что важно для надежного конструирования. Когда класс полностью инкапсулирован, появляется возможность модифицировать его реализацию без опасений нарушить связи с другими компонентами, взаимодействующими с ним. Полностью инкапсулированный класс объявляет все свои поля как `private`, и предоставляет методы для доступа к этим полям.

### Совет к экзамену

**Инкапсуляция.** Убедитесь, что Вы поняли концепцию инкапсуляции. Вы увидите, по крайней мере, один вопрос по этой теме во время экзамена.

## Полиморфизм

Полиморфизм является способностью принимать различные формы. В объектно-ориентированном программировании, это относится к способности объектов иметь несколько методов с одним и тем же именем, но с разными типами аргументов. Методы `print()` и `println()` класса `PrintStream`, являются отличным примерами полиморфизма. Эти методы поддерживают печать объектов разных типов.

Компилятор и виртуальная машина, поддерживают полиморфизм, определяя вызов каждого метода по его форме. Способность определить, какой метод использовать в сложных ситуациях, и является сутью полиморфизма.

## Позднее связывание

Иногда программа нуждается в соединении с объектами различных классов. Например, представьте программу, которая имеет возможность получить объект через линию связи. Программа может не знать какому классу принадлежит объект до тех пор, пока не получит его. Способность отложить решение о том, какому классу принадлежит объект до времени выполнения, известна как *позднее* или *динамическое связывание*. Позднее связывание важная особенность в объектно-ориентированном программировании, т.к. оно позволяет разрабатывать программы в более гибкой манере. Без позднего связывания, программа обязана знать заранее к каким классам объектов она будет обращаться к моменту компиляции. Позднее связывание – ключ к поддержке наследования, т.к. оно предоставляет алгоритм, позволяющий классу работать с любым из его подклассов. Система времени исполнения определяет текущий класс объекта и вызывает его любые переопределенные методы.

## Объявление классов

Теперь, после того как мы охватили концепции объектно-ориентированного программирования, мы можем перейти к его азам. В основном, все, что Вы будете делать, как Java программист, это объявлять классы и интерфейсы, а затем наполнять их плотью в виде полей переменных и методов. Классы объявляются, используя следующий синтаксис:

```
Modifiers class ClassName extendsClause implementsClause {  
    //Class body  
}
```

### Примечание

**Внутренние классы.** Внутренние классы – специальный тип классов, объявляемый внутри контекста класса, интерфейса или блока. Дальнейшее описание касается обычных классов верхнего уровня. Мы рассмотрим внутренние классы позднее в этой главе.

Модификаторы, `extends` и `implements` блоки – необязательные. Однако, если они предоставлены, используются следующим образом:

- ❖ Модификаторы (Modifiers) – Класс верхнего уровня может быть объявлен, как `public`, `final` или `abstract`. `Public` класс доступен вне его пакета. Если класс не объявлен как `public`, он доступен только классам и интерфейсам этого же пакета. `Final` класс – это класс на расширение которого наложен запрет. При попытке расширить класс объявленный как `final`, возникает ошибка компиляции. `Abstract` класс – это класс объявляющий один или более `abstract` методов. Т.к. `abstract` методы должны быть переопределены, объявление такого класса подразумевает, что он будет расширен классом, который предоставит наполнение `abstract` методов. Класс не может быть и `abstract` и `final` одновременно.
- ❖ `Extends` блок – состоит из ключевого слова `extends` с последующим именем класса, который расширяет данный. Класс, определенный в блоке `extends`, называется родительским или прямым суперклассом. Класс может иметь только одного родителя. Если блок `extends` опущен, класс расширяет `java.lang.Object` по умолчанию.
- ❖ `Implements` блок – определяет интерфейсы, воплощаемые классом. Он состоит из ключевого слова `implements` с последующим списком интерфейсов, разделенным запятой. Чтобы класс воплотил интерфейс, он должен предоставить методы с такими же сигнатурами, как и все методы определенные в интерфейсе или наследованные из интерфейса, который тот в свою очередь расширяет. `Abstract` класс может предоставлять `abstract` методы. `Non-abstract` класс должен предоставлять только `non-abstract` методы. Если класс воплощает интерфейс, он должен быть определен в блоке `implements` или в соответствующем блоке родителя. Если интерфейс не определен в блоке `implements`, класс не

воплощает его, даже если он и реализует все методы объявленные в интерфейсе.

Тело класса объявляет все члены (поля переменных и методы), конструкторы и инициализаторы. Членами класса могут быть также внутренние классы и (или) внутренние интерфейсы.

#### Совет к экзамену

**Явные подклассы.** Все классы рассматриваются как подклассами самих себя. Явным подклассом класса X является любой подкласс X за исключением самого себя.

### Конструирование классов, используя подход «Является» и «Имеет»

Вы, вероятно, столкнетесь с несколькими вопросами на сертификационном экзамене, которые проверят Ваши способности по конструированию класса, используя англоязычную спецификацию его характеристик. Это описание определяет связи, используя «является» (is a) и «имеет»(has a). «Является» описывает наследственную связь, а «имеет» определяет поля переменных. Следующий пример описывает эту ситуацию:

*A circle is a shape that has a center point and a radius.  
Круг является фигурой, имеющей центр и радиус.*

Следующий код определяет класс, описанный выше:

```
public class Circle extends Shape{
    Point center;
    double radius;
}
```

Вообще, где бы Вы ни увидели «является»(is a) , размышляйте в терминах расширения класса. Соответственно «имеет» (has a), должно наводить Вас на мысль о наличии полей переменных.

#### Совет к экзамену

**“Is A” и “Has A”.** Вы должны обратить особое внимание на семантику “Is A” и “Has A”. Несколько тестов в экзамене потребуют от Вас сконструировать класс на основе этих описаний.

### Конструкторы

Объекты это экземпляры классов, созданные с помощью конструкторов. Каждый класс требует наличие конструктора для создания экземпляра класса. В действительности, если класс не объявляет конструктор, компилятор автоматически создает его для класса. Тем не менее, в большинстве случаев, Вы не будете полагаться на компилятор и создадите его сами.

Синтаксис объявления конструктора:

```
modifiers ClassName (arguments) throwClause {
    //Constructor body
}
```

`Modifiers`, `arguments`, `throwsClause` – это произвольные блоки. Допустимыми модификаторами являются: `public`, `protected` и `private`. Отсутствие модификаторов означает, что конструктор доступен только в пакете, в котором определен сам класс. Действие модификаторов `public`, `protected` и `private` точно такое же, как и в случае с методами. `Arguments` и блок `throws` используется как в методах (см. Раздел 4 «Объявления и контроль доступа»).

Важным отличием конструкторов от методов, является то, что они не наследуются. Каждый класс должен объявлять свои собственные конструкторы. Конструктор по умолчанию не имеет аргументов и просто вызывает конструктор суперкласса для создания экземпляра конкретного класса.

### Использование `this()` и `super()`

Возможно, Вы захотите определить несколько конструкторов, получающих подмножество аргументов для создания объекта. Один конструктор получает все аргументы и поддерживает код, создающий объект; остальные конструкторы получают только часть аргументов, инициализируя недостающие значениями по умолчанию, и вызывают главный конструктор. В качестве примера рассмотрим Листинг 6.1.

#### Листинг 6.1

**Конструкторы класса `Box` используют `this()` для доступа к друг другу**

```
public class Box {
    double x, y, width, height;
    public Box(double x, double y, double width, double height)
    {
        this.x = x;
        this.y = y;
        this.width = width;
        this.height = height;
    }
    public Box(double x, double y) {
        this(x,y,10,10);
    }
    public Box() {
        this(1,1);
    }
}
```

Первый `Box()` конструктор получает аргументы `x`, `y`, `width`, `height` и связывает их с соответствующими полями. Второй получает только координаты в качестве параметров и вызывает первый, передавая 10 в качестве значения по умолчанию для `width` и `height`. Третий не имеет аргументов вообще и вызывает второй со значениями `x` и `y` равными 1.

Выражения `this(x, y, 10, 10)` и `this(1,1)` используемые во втором и третьем конструкторе, являются специальной нотацией Java, позволяющими одному конструктору вызывать другой из того же класса. Чтобы подобные вызовы стали возможными должен присутствовать конструктор с соответствующим списком аргументов.

#### Примечание

**Использование `this()`.** Если `this()` используется в конструкторе, это выражение должно быть первым выражением конструктора.

В следующем примере, вместо вызова конструктора того же класса, при конструировании объекта, вызовем конструктор суперкласса. Для этого используем вызов конструктора суперкласса `super()`. Чтобы посмотреть, как этот вызов работает, разберемся с классом `MyBox` из Листинга 6.2.

#### Листинг 6.2

**Конструкторы `MyBox` использует `super()` для доступа к конструктору `Box`**

```
import java.awt.Color;

public class MyBox extends Box {
    Color outerColor;
    Color innerColor;
    public MyBox(double x, double y, double width, double
height, Color outer, Color inner) {
        super(x,y,width,height);
        outerColor = outer;
        innerColor = inner;
    }
    public MyBox(Color outer, Color inner) {
        super(10,10,100,100);
        outerColor = outer;
        innerColor = inner;
    }
    public MyBox() {
        super();
        outerColor = Color.black;
        innerColor = Color.white;
    }
}
```

Класс `MyBox` расширяет класс `Box`, способностью устанавливать цвет периметра (`perimeter`) и цвет заливки (`inner`). Первый конструктор вызывает `super(x, y, width, height)`, чтобы конструктор суперкласса (`Box`) инициализировал положение и размеры объекта. Второй – вызывает конструктор суперкласса и передает ему в качестве параметров: `10, 10, 100, 100`; координаты – `10` и размеры – `100` соответственно. Третий просто вызывает из суперкласса конструктор по умолчанию.

#### **Примечание**

**Использование `super()`.** Если `super()` используется в конструкторе, это выражение должно быть первым выражением конструктора.

Если `this()` или `super()` используется в конструкторе, они должны быть первыми выражениями конструктора, что исключает их совместное использование. Если ни `this()`, ни `super()` не задействованы, происходит неявный вызов `super()` (без аргументов), поддерживаемый компилятором. Это приводит к тому, что часть класса, принадлежащая суперклассу, инициализируется до того как создается сам объект. Также, если класс не определяет доступный конструктор, он не может быть расширен.

## **Объявление интерфейсов**

Интерфейс определяет набор методов, воплощаемых в классе. Тем не менее, он может быть использован для объявления констант, внутренних классов и интерфейсов.

Синтаксис объявления интерфейса следующий:

```
Modifiers interface InterfaceName extendsClause {  
    //Interface body  
}
```

Интерфейс может использовать модификаторы `public` и `abstract`. Тем не менее, использование `abstract` излишне, т.к. все интерфейсы являются `abstract`, поскольку объявляют только абстрактные методы. Интерфейс объявленный, как `public`, доступен и вне пакета.

Блок `extends` состоит из ключевого слова `extends` и перечня расширяемых интерфейсов через запятую. В этом случае интерфейс наследует все константы и методы расширяемых интерфейсов.

Тело интерфейса состоит из объявлений констант, абстрактных методов, внутренних классов и интерфейсов. Внутренние классы и интерфейсы менее распространены и обсуждаются позднее в этом разделе. Константы объявляются как `public`, `static` и `final`. Если Вы опустите эти модификаторы, компилятор выполнит эти преобразования автоматически.

Абстрактные методы подробно обсуждались в Разделе 4.

## Внутренние и анонимные классы

Внутренние классы и интерфейсы представлены в JDK, начиная с версии 1.1. Внутренний класс (`inner`), также известный как вложенный (`nested`), является классом, объявленным в качестве члена другого класса или локально в блоке выражений. Внутренние интерфейсы менее распространены, они объявляются как члены другого класса или интерфейса.

### Совет к экзамену

<p><b>Внутренние и анонимные классы.</b> Убедитесь в том, что Вы понимаете, как внутренние и анонимные классы используются. Вы встретите несколько вопросов на эту тему в течение экзамена.</p>
-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------

## Внутренние классы

Когда внутренний класс определяется как член другого класса, он может иметь модификаторы доступа (`public`, `protect`, `private` или пакетный доступ). Он также может быть объявлен, как `abstract`, `final` или `static`. Модификаторы доступа определяют уровень доступа к классу извне класса и пакета. Они используются точно так же, как с переменными и методами.

Внутренние классы редко объявляются `abstract` или `final`. Наиболее важным моментом является отличие статических и не статических классов.

Следующий пример демонстрирует объявление простого внутреннего класса:

```
class Outer{
    class Inner{
        //Inner class body
    }
}
```

Класс Outer – обычный класс верхнего уровня. Inner – простой класс, объявленный внутренним по отношению к Outer.

## Нестатические внутренние классы

Нестатические внутренние классы могут использоваться для создания объектов, являющихся экземплярами класса. Каждый экземпляр нестатического внутреннего класса связан с экземпляром внешнего класса, в котором он объявлен. Фактически, экземпляр внутреннего класса не может быть создан без предваряющего его создания экземпляра внешнего класса. Следующая программа демонстрирует этот момент:

```
class Outer{
    public static void main(String[] args){
        Inner inner = new Inner();
    }
    class Inner{
        //Inner class body
    }
}
```

Если Вы попытаете откомпилировать эту программу, Вы получите следующее сообщение об ошибке (при условии использования того же компилятора):

```
Outer.java:4: No enclosing instance of class Outer is
in scope; an explicit one must be provided when creating
inner class Outer. Inner, as in "outer. new Inner()" or
"outer.super()".
```

```
    Inner inner = new Inner();
                  ^
```

```
1 error
```

Чтобы исправить эту ошибку, Вы должны создать экземпляр Outer и передать его в качестве окружающего объекта для экземпляра new Inner:

```
class Outer{
    public static void main(String[] args){
        Outer outer = new Outer();
        Inner inner = outer.new Inner();
    }
    class Inner{
        //Inner class body
    }
}
```

Нотация `outer.new Inner()`, указывает на то, что экземпляр класса `Outer`, представленный переменной `outer`, является окружающим объектом, связанным с новым экземпляром класса `Inner`.

Вы также можете создать экземпляр класса `Inner`, используя `this` ссылающееся на экземпляр, окружающего объекта. Следующая программа демонстрирует эту возможность:

```
class Outer{
    public static void main(String[] args){
        new Outer();
    }
    Outer(){
        new Inner();
    }
    class Inner{
        Inner(){
        }
    }
}
```

Поскольку конструктор `Inner()` вызывается из конструктора `Outer()`, `this` ссылается на созданный объект `Outer()`. *Прим. переводчика: В данном примере `this` существует неявно, но Вы можете изменить бю строку на: `this.new Inner()`;*

Для достижения той же цели есть можно использовать и более компактную запись:

```
class Outer{
    public static void main(String[] args){
        new Outer().new Inner();
    }
    class Inner{
    }
}
```

Переменные и методы, объявленные во внешнем классе, доступны из внутреннего. Однако статический внутренний класс может обращаться только к статическим переменным и методам внешнего класса.

### Примечание

<p><b>Многоуровневое вложение внутренних классов.</b> Многоуровневое вложение допустимо, однако редко имеет практический смысл.</p>
-------------------------------------------------------------------------------------------------------------------------------------

В некоторых случаях важно понимать разницу между экземпляром внутреннего класса и экземпляром внешнего. Ключевое слово `this` всегда указывает на текущий объект. Когда оно используется в пределах внутреннего класса – указывает на него, и наоборот, в пределах внешнего ссылается на внешний класс. Однако, бывают случаи когда необходимо обратиться к внешнему классу из внутреннего. Следующий пример демонстрирует эту возможность:

```

class Outer{
    String s = "Outer";
    public static void main(String[] args){
        new Outer().new Inner();
    }
    class Inner{
        String s = "Inner";
        Inner(){
            System.out.println(this.s);
            System.out.println(Outer.this.s);
        }
    }
}

```

Эта программа печатает строку Inner, а затем Outer. В конструкторе Inner(), this.s возвращает значение переменной s ссылающейся на класс Inner, а Outer.this.s на переменную внешнего класса.

### **Статические внешние классы**

Статические внутренние классы отличаются от своих нестатических собратьев, тем, что они не производятся как отдельные экземпляры классов. Более того, они не ассоциируются с их окружающими классами, хотя и могут обращаться к их статическим переменным и методам.

Статические внутренние классы попадают в статическую область и с ними можно обращаться как с классами верхнего уровня. На них даже ссылаются как на классы верхнего уровня в некоторых изданиях по Java. Это приводит к противоречию – внутренние классы верхнего уровня.

Пример простого статического внутреннего класса:

```

class Outer{
    public static void main(String[] args){
        new Inner();
    }
    static class Inner{
        Inner(){
            System.out.println("Some static");
        }
    }
}

```

Заметьте, что т.к. Inner является статическим классом, для его создания нет надобности ссылаться на экземпляр Outer.

## Локальные внутренние классы

Внутренние классы могут быть локальными по отношению к определенному блоку кода. Т.к. локальные внутренние классы не являются членами внешнего класса, они не привязаны к его экземпляру. В их объявлении могут участвовать также модификаторы: `private`, `public`, `protected` и `static`.

Так как, локальные классы объявляются внутри блока кода (как локальные переменные), они имеют доступ только к локальным переменным или параметрам метода того блока, в котором они объявлены. На локальные классы налагается дополнительное ограничение: они имеют доступ только к проинициализированным переменным и параметрам объявленным `static`. Следующий пример демонстрирует использование локального внутреннего класса.

```
class MyClass {
    public static void main(String[] args){
        final String s = "some local results";
        new Outer().new Inner();
        class Local{
            Local(){
                System.out.println(s);
            }
        }
        new Local();
    }
}
```

В этом примере внутренний класс `Local` определен в контексте метода `main()`. Обратите внимание на то, что переменная `s` должна быть объявлена как `static`, чтобы класс `Local` получил к ней доступ.

## Анонимные классы

В некоторых случаях удобно объявить и использовать класс непосредственно на месте. Хорошим примером такого класса, является класс, воплощающий интерфейс слушателя событий или расширяющий класс-адаптер.

Анонимные классы полностью подходят для использования их в одиночном выражении. Анонимные классы (как следует из названия), являются классами без имен. Они объявляются именами классов, которые они расширяют или интерфейсов которые воплощают. Следующие примеры описывают обе синтаксические формы:

```
Расширение класса
new SuperClassName(argument){
    //Class body
}
```

```

Воплощение интерфейса
new InterfaceName() {
    //Implement interface methods
}

```

В первой форме, суперкласс идентифицирует объявленный класс. Модификаторы и extends блок не допускаются. Аргументы (необязательные), поддерживаемые суперклассом, передаются в его конструктор (суперкласса) при создании объекта. Суперкласс должен иметь конструктор с соответствующей сигнатурой, в противном случае будет сгенерирована ошибка компиляции. Анонимные классы не имеют конструктора.

Во второй форме, анонимный класс объявлен в отношении воплощаемого интерфейса. В этом случае имя интерфейса определяет тип создаваемого объекта. Когда анонимный класс объявляется, таким образом, он автоматически становится дочерним классом по отношению к классу Object. Так же стандарт требует, чтобы анонимный класс воплощал все методы объявленные в интерфейсе. При объявлении класса как интерфейс – аргументы не поддерживаются.

Следующая программа (Листинг 6.3) иллюстрирует использование обеих форм анонимных классов.

### Листинг 6.3

#### Анонимные классы удобны для обработки событий

```

import java.awt.*;
import java.awt.event.*;

class Anonymous extends Frame {
    public static void main(String[] args) {
        new Anonymous();
    }
    Anonymous() {
        setTitle("Anonymous");
        setLayout(new FlowLayout());
        final Button button = new Button("Click here!");
        button.addActionListener(new ActionListener() {
            public void actionPerformed(ActionEvent e) {
                String label = button.getLabel();
                if(label.equals("Click here!"))
                    button.setLabel("Try again");
                else
                    button.setLabel("Click here!");
            }
        });
        add(button);
        addWindowListener(new WindowAdapter() {
            public void windowClosing(WindowEvent e) {
                System.exit(0);
            }
        });
    }
}

```

```

    pack();
    setSize(200,200);
    show();
}
}

```

Программа Anonymus использует внутренние классы в двух участках кода. Первый раз, создается интерфейс ActionListener, передающийся в качестве параметра в метод addActionListener(), объекта Button:

```

button.addActionListener(new ActionListener() {
    public void actionPerformed(ActionEvent e) {
        String label = button.getLabel();
        if(label.equals("Click here!"))
            button.setLabel("Try again");
        else
            button.setLabel("Click here!");
    }
});

```

Метод actionPerformed() интерфейса ActionListener, воплощается с целью реагирования на нажатие кнопки. Вы узнаете больше о событиях (event) в Разделе 13: «Обработка событий».

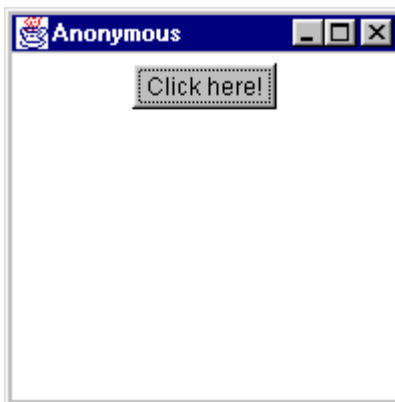
Эта программа также использует анонимный класс для создания объекта, передаваемого в качестве аргумента для метода addWindowListener(). Он расширяет класс WindowAdapter для отслеживания закрытия окна.

```

addWindowListener(new WindowAdapter() {
    public void windowClosing(WindowEvent e) {
        System.exit(0);
    }
});

```

Программа генерирует окно Рисунок 6.3, а когда Вы щелкаете на кнопку, она меняет свою надпись как показано на Рисунке 6.4.



**Рисунок 6.3** Окно программы Anonymus

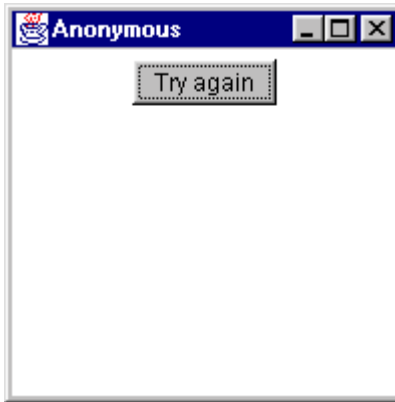


Рисунок 6.4 Надпись на кнопке меняется после щелчка

## Перегрузка методов

Ранее в этом разделе, полиморфизм был упомянут, как важная часть объектно-ориентированных языков. И хотя не только полиморфизм воплощает суть объектно-ориентированного подхода, он предоставляет огромные преимущества. Перегрузка методов – главный путь, в котором полиморфизм проявляется в Java. Перегрузка методов проявляется как набор методов с одним и тем же именем, но с разными наборами параметров и возвращаемыми типами в пределах одного класса. Существует всего одно ограничение – методы не могут иметь одно и тоже имя при одинаковых списках аргументов. Другими словами, хотя бы один аргумент должен отличаться. Списки аргументов считаются разными, если они отличаются последовательностью типов аргументов, имена аргументов не имеют значения.

Отличным примером, в этом отношении, являются методы `print()` и `println()`, перегружаемые для поддержки всех примитивных типов и объектов.

### Примечание

**Перегрузка операторов.** Некоторые языки, например C++, поддерживают перегрузку операторов. Перегрузка операторов позволяет одному и тому же оператору использоваться с различными типами операндов. Java не позволяет определять новые операторы. Однако, Java перегружает оператор `+`, который может быть использован как для арифметических операций так и для конкатенации строк.

**Ограничения по перегрузки методов.** Перегруженные методы могут иметь разные возвращаемые типы. Однако, недопустимо, чтобы существовало два метода с одним списком аргументов и разными возвращаемыми типами.

## Переопределение методов

В дополнение к перегрузке методов, Java поддерживает их переопределение. Переопределение похоже на перегрузку в смысле использования одного и того же имени метода. Тем не менее, переопределение отличается тем, что обращается к использованию методов с тем же именем, аргументами и возвращаемым типом между классом и его суперклассом. В то время как перегрузка допускает существование нескольких методов с одним именем, переопределение позволяет подклассу переопределить код методов, унаследованных им от суперкласса, и таким образом специализировать поведение подкласса.

Когда класс расширяет его суперкласс, он наследует все его не-private методы. В некоторых случаях Вам понадобится переопределить некоторые унаследованные методы. Если подкласс создает метод с таким же именем, как унаследованное, но с другим списком параметров – это перегрузка метода. Если же класс создает метод с сигнатурой, идентичной методу суперкласса - это переопределение метода. Следующие правила применимы к переопределению:

- ❖ Методы переопределяющего (подкласса) и переопределяемого (суперкласса) должны иметь один и тот же возвращаемый тип.
- ❖ Права доступа к переопределяющему методу не должны быть строже, чем у переопределяемого метода.
- ❖ Блок `throws` переопределяющего метода должен декларировать те же исключения, что и переопределенный метод.

Давайте посмотрим на каждое из ограничений. Первое следует из золотого правила, гласящего, что класс не может иметь два метода с одним и тем же именем, списком аргументов и отличными возвращаемыми типами. Т.к. если бы класс мог определить метод с таким же именем, списком аргументов и другим возвращаемым типом, чем у суперкласса – возник бы конфликт при вызове методов.

Второе ограничение немного сложнее. Его целью является предупредить снижение уровня доступа к методу ниже, чем у суперкласса. Однако, определить уровень доступа иногда тяжело. Вот следующие правила:

- ❖ Если переопределяемый метод `public`, то и переопределенный должен быть `public`.
- ❖ Если переопределяемый метод `protected`, то переопределенный должен быть `public` или `protected`.
- ❖ Если переопределяемый метод имеет пакетный уровень доступа (модификаторы отсутствуют), переопределенный может быть `public`, `protected` или использоваться без модификаторов.
- ❖ Если переопределяемый метод `private`, то он не наследуется у суперкласса, и переопределения не происходит.

Третье ограничение требует, чтобы любое исключение, посылаемое переопределяющим методом, посылалось так же и переопределенным методом. Это означает, что исключение должно быть подклассом исключения, посылаемого методом суперкласса.

## Итоги по разделу

В этом разделе, Вы познакомились с объектно-ориентированным программированием на Java. Вы научились, как объявлять классы верхнего уровня и интерфейсы, и как использовать внутренние и анонимные классы. А так же, как объявлять конструкторы, переопределять и перегружать методы. Следующие вопросы помогут протестировать Ваши знания по этим темам и помогут подготовиться к сертификационному экзамену.

## Ключевые термины

- Повторное использование объектов

- Классификация
- Одиночное наследование
- Множественное наследование
- Инкапсуляция
- Полиморфизм
- Позднее связывание
- Родительский класс
- Суперкласс
- Прямой суперкласс
- Подкласс
- Прямой подкласс
- Член класса
- Абстрактный класс
- Внутренний класс
- Локальный внутренний класс
- Анонимный класс
- Перегрузка метода
- Переопределение метода