Глава 6

ИНТЕРФЕЙСЫ И АННОТАЦИИ

Решение сложной задачи поручайте ленивому сотруднику — он найдет более легкий путь.

Закон Хлейда

Интерфейсы

Назначение интерфейса — описание или спецификация функциональности, которую должен реализовывать каждый класс, его имплементирующий. Класс, реализующий интерфейс, предоставляет к использованию объявленный интерфейс в виде набора **public**-методов в полном объеме.

Интерфейсы подобны абстрактным классам, хотя и не являются классами. В языке Java существует три вида интерфейсов: интерфейсы, определяющие функциональность для классов посредством описания методов, но не их реализации; функциональные интерфейсы, специфицирующие в одном абстрактном методе свое применение; интерфейсы, реализация которых автоматически придает классу определенные свойства. К последним относятся, например, интерфейсы **Cloneable**, **Serializable**, **Externalizable**, отвечающие за клонирование и сохранение объекта (сериализацию) в информационном потоке соответственно.

Общее определение интерфейса имеет вид:

```
[public] interface Name [extends NameOtherInterface,…, NameN] {
  // constants, methods
}
```

Все объявленные в интерфейсе абстрактные методы автоматически трактуются как **public abstract**, а все поля — как **public static final**, даже если они так не объявлены. Интерфейсы могут объявлять статические методы. В интерфейсах могут объявляться методы с реализацией с ключевым словом **default**. Эти методы могут быть **public** или **private**.

Класс может реализовывать любое число интерфейсов, указываемых через запятую после ключевого слова **implements**, дополняющего определение класса. После этого класс обязан реализовать все абстрактные методы, полученные им от интерфейсов, или объявить себя абстрактным классом.

Если необходимо определить набор функциональности для какого-либо рода деятельности, например, для управления счетом в банке, то следует использовать интерфейс вида:

/ # 1 # объявление интерфейса управления банковским счетом # AccountAction.java */*

```
package by.epam.learn.advanced;
public interface AccountAction{
  boolean openAccount();
  boolean closeAccount();
  void blocking();
  default void unBlocking() {}
  double depositInCash(int accountId, int amount);
  boolean withdraw(int accountId, int amount);
  boolean convert(double amount);
  boolean transfer(int accountId, double amount);
}
```

В интерфейсе обозначены, но не реализованы, действия, которые может производить клиент со своим счетом. Реализация не представлена из-за возможности различного способа выполнения действия в конкретной ситуации. А именно: счет может блокироваться автоматически, по требованию клиента или администратором банковской системы. В каждом из трех указанных случаев реализация метода **blocking()** будет уникальной и никакого базового решения предложить невозможно. С другой стороны, наличие в интерфейсе метода заставляет класс, его имплементирующий, предоставить реализацию методу. Программист получает повод задуматься о способе реализации функциональности, так как наличие метода в интерфейсе говорит о необходимости той или иной функциональности всем классам, реализующим данный интерфейс.

Метод **unblocking()** предоставляет дефолтную реализацию метода, которая может быть при необходимости переопределена в подклассе.

Интерфейс следует проектировать ориентированным на выполнение близких по смыслу задач, например, отделить действия по созданию, закрытию и блокировке счета от действий по снятию и пополнению средств. Такое разделение разумно в ситуации, когда клиенту посредством Интернет-системы не предоставляется возможность открытия, закрытия и блокировки счета. Тогда вместо одного общего интерфейса можно записать два специализированных: один для администратора, второй — для клиента.

/ # 2 # общее управление банковским счетом # AccountBaseAction.java */*

```
package by.epam.learn.advanced;
public interface AccountBaseAction {
  boolean openAccount();
  boolean closeAccount();
  void blocking();
  default void unBlocking() {}
}
```

/ # 3 # операционное управление банковским счетом # AccountManager.java */*

```
package by.epam.learn.advanced;
public interface AccountManager {
  double depositInCash(int accountId, int amount);
  boolean withdraw(int accountId, int amount);
  boolean convert(double amount);
  boolean transfer(int accountId, double amount);
}
```

Реализация интерфейса при реализации всех методов выглядит следующим образом:

/ # 4 # реализация общего управления банковским счетом # AccountBaseActionImpl.java */*

```
package by.epam.learn.advanced.impl;
import by.epam.learn.advanced.AccountBaseAction;
public class AccountBaseActionImpl implements AccountBaseAction {
  public boolean openAccount() {
    // more code
  }
  public boolean closeAccount() {
    // more code
  }
  public void blocking() {
    // more code
  }
  public void unBlocking() {
    // more code
  }
}
```

Если по каким-либо причинам для данного класса не предусмотрена реализация метода или его реализация нежелательна, рекомендуется генерация непроверяемого исключения в теле метода, а именно:

```
public boolean blocking() {
  throw new UnsupportedOperationException();
}
```

Менее хорошим примером будет реализация в виде:

```
public boolean blocking() {
  return false;
}
```

так как пользователь метода может считать реализацию корректной.

В интерфейсе не могут быть объявлены поля без инициализации, интерфейс в качестве полей содержит только **final static** константы.

Множественное наследование между интерфейсами допустимо. Классы, в свою очередь, интерфейсы только реализуют. Класс может наследовать один суперкласс и реализовывать произвольное число интерфейсов. В Java интерфейсы обеспечивают большую часть той функциональности, которая в C++ представляется с помощью механизма множественного наследования.

При именовании интерфейса в конец его названия может добавляться **able** в случае, если в названии присутствует действие (глагол). Не рекомендуется в имени интерфейса использовать слово **Interface** или любое его сокращение. В конец имени класса, реализующего интерфейс, для указания на источник действий можно добавлять слово **Impl**. Реализации интерфейсов помещать следует в подпакет с именем **impl**.

Интерфейсы можно применить, например, в задаче, где на поверхности линии соединяются в фигуры. Описание функционала методов по обработке информации о линиях и фигурах:

/ # 5 # объявление интерфейсов # LineGroupAction.java */*

```
package by.epam.learn.advanced;
import by.epam.learn.entity.AbstractShape;
public interface LineGroupAction {
  double computePerimeter(AbstractShape shape);
}
```

/ # 6 # наследование интерфейсов # ShapeAction.java */*

```
package by.epam.learn.advanced;
import by.epam.learn.entity.AbstractShape;
public interface ShapeAction extends LineGroupAction {
  double computeSquare(AbstractShape shape);
}
```

Класс, который будет реализовывать интерфейс **ShapeAction**, должен будет определить все методы из цепочки наследования интерфейсов. В данном случае это методы **computePerimeter()** и **computeSquare()**.

/ # 7 # абстрактная фигура # AbstractShape.java */*

```
package by.epam.learn.entity;
public class AbstractShape {
  private long shapeId;
  public long getShapeId() {
    return shapeId;
  }
  public void setShapeId(long shapeId) {
    this.shapeId = shapeId;
  }
}
```

Интерфейсы обычно объявляются как **public**, потому что описание функциональности, предоставляемое ими, может быть использовано в других пакетах проекта. Интерфейсы с областью видимости в рамках пакета, атрибут доступа по умолчанию, могут использоваться только в этом пакете и нигде более.

Реализация интерфейсов классом может иметь вид:

```
[public] class NameClass implements Name1, Name2,…, NameN {/*  */}
```

Здесь *Name1, Name2,..., NameN* — перечень используемых интерфейсов. Класс, который реализует интерфейс, должен предоставить полную реализацию всех абстрактных методов, объявленных в интерфейсе. Кроме того, данный класс может объявлять свои собственные методы.

/ # 8 # реализация интерфейса # RectangleAction.java */*

```
package by.epam.learn.advanced.impl;
import by.epam.learn.entity.AbstractShape;
import by.epam.learn.entity.Rectangle;
import by.epam.learn.advanced.ShapeAction;
public class RectangleAction implements ShapeAction {
  @Override
  public double computeSquare(AbstractShape shape) {
    double square;
    if (shape instanceof Rectangle rectangle) {
      square = rectangle.getHeight() * rectangle.getWidth();
    } else {
      throw new IllegalArgumentException("Incompatible shape " + shape.getClass());
    }
    return square;
  }
  @Override
  public double computePerimeter(AbstractShape shape) {
    double perimeter;
    if (shape instanceof Rectangle rectangle) {
      perimeter = 2 * (rectangle.getWidth() + rectangle.getHeight());
    } else {
      throw new IllegalArgumentException("Incompatible shape " + shape.getClass());
    }
    return perimeter;
  }
}
```

/ # 9 # реализация интерфейса # TriangleAction.java */*

```
package by.epam.learn.advanced.impl;
import by.epam.learn.entity.AbstractShape;
import by.epam.learn.entity.RightTriangle;
import by.epam.learn.advanced.ShapeAction;
public class TriangleAction implements ShapeAction {
```

```
  @Override
  public double computeSquare(AbstractShape shape) {
    double square;
    if (shape instanceof RightTriangle triangle) {
      square = 1./2 * triangle.getSideA() * triangle.getSideB();
    } else {
      throw new IllegalArgumentException("Incompatible shape " + shape.getClass());
    }
    return square;
  }
  @Override
  public double computePerimeter(AbstractShape shape) {
    double perimeter = 0;
    if (shape instanceof RightTriangle triangle) {
      double a = triangle.getSideA();
      double b = triangle.getSideB();
      perimeter = a + b + Math.hypot(a, b);
    } else {
      throw new IllegalArgumentException("Incompatible shape " + shape.getClass());
    }
    return perimeter;
  }
}
```

При неполной реализации интерфейса класс должен быть объявлен как абстрактный, что говорит о необходимости реализации абстрактных методов уже в его подклассе.

/ # 10 # неполная реализация интерфейса # PentagonAction.java */*

```
package by.epam.learn.advanced.impl;
import by.epam.learn.entity.AbstractShape;
import by.epam.learn.advanced.ShapeAction;
public abstract class PentagonAction implements ShapeAction {
  @Override
  public double computePerimeter(AbstractShape shape) {
          // code
  }
}
```

Классы для определения фигур, используемые в интерфейсах:

/ # 11 # прямоугольник, треугольник # Rectangle.java # RightTriangle.java */*

```
package by.epam.learn.entity;
public class Rectangle extends AbstractShape {
  private double height;
  private double width;
  public Rectangle(double height, double width) {
    this.height = height;
```

```
    this.width = width;
  }
  public double getHeight() {
    return height;
  }
  public double getWidth() {
    return width;
  }
}
package by.epam.learn.entity;
import java.util.StringJoiner;
public class RightTriangle extends AbstractShape {
  private double sideA;
  private double sideB;
  public RightTriangle(double sideA, double sideB) {
    this.sideA = sideA;
    this.sideB = sideB;
  }
  public double getSideA() {
    return sideA;
  }
  public void setSideA(double sideA) {
    this.sideA = sideA;
  }
  public double getSideB() {
    return sideB;
  }
  public void setSideB(double sideB) {
    this.sideB = sideB;
  }
  @Override
  public String toString() {
    return new StringJoiner(", ", RightTriangle.class.getSimpleName() + "[", "]")
            .add("sideA=" + sideA).add("sideB=" + sideB).toString();
  }
}
```

/ # 12 # свойства ссылок на интерфейс # ActionMain.java */*

```
package by.epam.learn.advanced;
import by.epam.learn.advanced.impl.RectangleAction;
import by.epam.learn.advanced.impl.TriangleAction;
import by.epam.learn.entity.Rectangle;
import by.epam.learn.entity.RightTriangle;
public class ActionMain {
  public static void main(String[] args) {
    ShapeAction action;
    try {
      Rectangle rectShape = new Rectangle(2, 5);
      action = new RectangleAction();
```

```
      System.out.println("Square rectangle: "+ action.computeSquare(rectShape));
      System.out.println("Perimeter rectangle: "
                         + action.computePerimeter(rectShape));
      RightTriangle trShape = new RightTriangle(3, 4);
      action = new TriangleAction();
      System.out.println("Square triangle: " + action.computeSquare(trShape));
      System.out.println("Perimeter triangle: "
                         + action.computePerimeter(trShape));
      action.computePerimeter(rectShape); // runtime exception
    } catch (IllegalArgumentException e) {
      System.err.println(e.getMessage());
    }
  }
}
```

В результате будет выведено:

Square rectangle: 10.0
Perimeter rectangle: 14.0
Square triangle: 6.0
Perimeter triangle: 12.0
Incompatible shape class by.epam.learn.entity.Rectangle

Допустимо объявление ссылки на интерфейсный тип или использование ее в качестве параметра метода. Такая ссылка может указывать на экземпляр любого класса, который реализует объявленный интерфейс. При вызове метода через такую ссылку будет вызываться его реализованная версия, основанная на текущем экземпляре класса. Выполняемый метод разыскивается динамически во время выполнения, что позволяет создавать классы позже кода, который вызывает их методы.

Параметризация интерфейсов

Реализация для интерфейсов, параметры методов которых являются ссылками на иерархически организованные классы, в данном случае представлена достаточно тяжеловесно. Необходимость проверки принадлежности обрабатываемого объекта к допустимому типу снижает гибкость программы и увеличивает количество кода, в том числе и на генерацию, и обработку исключений.

Сделать реализацию интерфейса удобной, менее подверженной ошибкам и практически исключающей проверки на принадлежность типу можно достаточно легко, если при описании интерфейса добавить параметризацию в виде:

/ # 13 # параметризация интерфейса # ShapeGeneric.java */*

```
package by.epam.learn.advanced;
import by.epam.learn.entity.AbstractShape;
public interface ShapeGeneric<T extends AbstractShape> {
```

```
  double computeSquare(T shape);
  double computePerimeter(T shape);
}
```

Параметризованный тип **T extendsAbstractShape** указывает, что в качестве параметра методов может использоваться только подкласс **AbstractShape**, что мало чем отличается от случая, когда тип параметра метода указывался явно. Но когда дело доходит до реализации интерфейса, то указывается конкретный тип объектов, являющихся подклассами **AbstractShape**, которые будут обрабатываться методами данного класса, а в качестве параметра метода также прописывается тот же самый конкретный тип:

/ # 14 # реализация интерфейса с указанием типа параметра # RectangleGeneric.java # TriangleGeneric.java */*

```
package by.epam.learn.advanced.impl;
import by.epam.learn.advanced.ShapeGeneric;
import by.epam.learn.entity.Rectangle;
public class RectangleGeneric implements ShapeGeneric<Rectangle> {
  @Override
  public double computeSquare(Rectangle shape) {
    return shape.getHeight() * shape.getWidth();
  }
  @Override
  public double computePerimeter(Rectangle shape) {
    return 2 * (shape.getWidth() + shape.getHeight());
  }
}
package by.epam.learn.advanced.impl;
import by.epam.learn.advanced.ShapeGeneric;
import by.epam.learn.entity.RightTriangle;
public class TriangleGeneric implements ShapeGeneric<RightTriangle> {
  @Override
  public double computeSquare(RightTriangle shape) {
    return 0.5 * shape.getSideA() * shape.getSideB();
  }
  @Override
  public double computePerimeter(RightTriangle shape) {
    double a = shape.getSideA();
    double b = shape.getSideB();
    double perimeter = a + b + Math.hypot(a, b);
    return perimeter;
  }
}
```

На этапе компиляции исключается возможность передачи в метод объекта, который не может быть обработан, т.е. код **action.computePerimeter(rectShape)** спровоцирует ошибку компиляции, если **action** инициализирован объектом класса **TriangleAction**.

Применение параметризации при объявлении интерфейсов в данном случае позволяет избавиться от лишних проверок и преобразований типов при реализации непосредственно самого интерфейса и использовании созданных на их основе классов.

/ # 15 # использование параметризованных интерфейсов # GenericMain.java */*

```
package by.epam.learn.advanced;
import by.epam.learn.advanced.impl.RectangleGeneric;
import by.epam.learn.advanced.impl.TriangleGeneric;
import by.epam.learn.entity.Rectangle;
import by.epam.learn.entity.RightTriangle;
public class GenericMain {
  public static void main(String[] args) {
    Rectangle rectangle = new Rectangle(2, 5);
    ShapeGeneric<Rectangle> rectangleGeneric = new RectangleGeneric();
    RightTriangle triangle = new RightTriangle(3, 4);
    ShapeGeneric<RightTriangle> triangleGeneric = new TriangleGeneric();
    System.out.println("Square rectangle: "
                           + rectangleGeneric.computeSquare(rectangle));
    System.out.println("Perimeter rectangle: "
                           + rectangleGeneric.computePerimeter(rectangle));
    System.out.println("Square triangle: "
                           + triangleGeneric.computeSquare(triangle));
    System.out.println("Perimeter triangle: "
                           + triangleGeneric.computePerimeter(triangle));
    // triangleGeneric.computePerimeter(rectangle); // compile error
  }
}
```

Аннотации

Аннотации — это метатеги, которые добавляются к коду и применяются к объявлению пакетов, классов, конструкторов, методов, полей, параметров и локальных переменных. Аннотации всегда обладают некоторой информацией и связывают эти дополнительные данные и все перечисленные конструкции языка. Фактически аннотации представляют собой их дополнительные модификаторы, применение которых не влечет за собой изменений ранее созданного кода. Аннотации позволяют избежать создания шаблонного кода во многих ситуациях, активируя утилиты для его генерации из аннотаций в исходном коде.

В языке Java определено несколько встроенных аннотаций для разработки новых аннотаций — **@Retention**, **@Documented**, **@Target** и **@Inherited** — из пакета **java.lang.annotation**. Из других аннотаций выделяются — **@Override**, **@Deprecated** и **@SuppressWarnings** — из пакета **java.lang**. Широкое использование аннотаций в различных технологиях и фреймворках обуславливается возможностью сокращения кода и снижения его связанности.