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 полностью абстрактным классам, хотя и не являются классами. Ни один из объявленных методов не может быть реализован внутри интерфейса. В языке Java существует два вида интерфейсов: интерфейсы, определяющие функциональность для классов посредством описания методов, но не их реализации; и интерфейсы, реализация которых автоматически придает классу определенные свойства. К последним относятся, например, интерфейсы **Cloneable** и **Serializable**, отвечающие за клонирование и сохранение объекта в информационном потоке соответственно.

Общее определение интерфейса имеет вид:

```
[public] interface Имя [extends Имя1, Имя2,…, ИмяN] { /* реализация интерфейса */ }
```

Все объявленные в интерфейсе методы автоматически трактуются как **public** и **abstract**, а все поля — как **public**, **static** и **final**, даже если они так не объявлены. Класс может реализовывать любое число интерфейсов, указываемых через запятую после ключевого слова **implements**, дополняющего определение класса. После этого класс обязан реализовать все 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 # обьявление интерфейса управления банковским счетом # AccountAction.java */*

```
package by.bsu.proj.accountlogic;
public interface AccountAction {
        // по умолчанию все методы public abstract
```

```
        boolean openAccount(); // объявление сигнатуры метода
        boolean closeAccount();
        void blocking();
        void unBlocking();
        double depositInCash(int accountNumber, int amount);
        boolean withdraw(int accountNumber, int amount);
        boolean convert(double amount);
        boolean transfer(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.

Интерфейс можно сделать 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.bsu.proj.accountlogic;
public interface AccountBaseAction {
        boolean openAccount();
        boolean closeAccount();
        void blocking();
        void unBlocking();
}
```

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

```
package by.bsu.proj.accountlogic;
public interface AccountOperationManager {
        double depositInCash(int accountNumber, int amount);
        boolean withdraw(int accountNumber, int amount);
        boolean convert(double amount);
        boolean transfer(int accountNumber, double amount);
}
```

Реализация интерфейса при реализации всех методов выглядит следующим образом:

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

```
package by.bsu.proj.accountlogic;
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 метод для данного класса не имеет реализации или его реализация нежелательна, рекомендуется генерация исключения в теле метода, а именно:

```
public boolean blocking() {
          throw new UnsupportedOperationException(); // лучше собственное исключение
}
```

Менее хорошим примером будет реализация в виде:

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

так как пользователь метода будет считать реализацию корректной.

В интерфейсе не могут быть объявлены поля без инициализации и методы с реализацией, интерфейс в качестве полей содержит только константы, в качестве методов — только абстрактные методы.

```
interface WrongInterface { // нежелательно использование слова Interface при объявлении
          int id; // ошибка, инициализации "по умолчанию" не существует
          void method() {} // ошибка, так как абстрактный метод не может иметь тела!
}
```

Множественное наследование между интерфейсами допустимо. Классы, в свою очередь, интерфейсы только реализуют. Класс может наследовать один суперкласс и реализовывать произвольное число интерфейсов. В языке Java интерфейсы обеспечивают большую часть той функциональности,

которая в C++ представляется с помощью механизма множественного наследования.

Например:

/ # 5 # обьявление интерфейсов # ILineGroupAction.java # IShapeAction.java */*

```
package by.bsu.shapes.action;
public interface ILineGroupAction {
          double computePerimeter(AbstractShape shape);
}
```

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

```
package by.bsu.shapes.action;
public interface IShapeAction extends ILineGroupAction {
          double computeSquare(AbstractShape shape);
}
```

Для более простой идентификации интерфейсов в большом проекте в сообществе разработчиков действует негласное соглашение о добавлении к имени интерфейса символа **I**, в соответствии с которым вместо имени **ShapeAction** следует записать **IShapeAction**. В конец названия может добавляться **able** в случае, если в названии присутствует действие (глагол). В конец имени класса, реализующего интерфейс, для указания на источник действий часто добавляется слово **Impl**.

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

Интерфейсы обычно объявляются как **public**, потому что описание функциональности, предоставляемое ими, может быть использовано в нескольких 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:

```
[доступ] class ИмяКласса implements Имя1, Имя2,…, ИмяN {/*код класса*/}
```

Здесь *Имя1, Имя2,..., ИмяN* — перечень используемых интерфейсов. Класс, который реализует интерфейс, должен предоставить полную реализацию всех методов, объявленных в интерфейсе. Кроме того, данный класс может объявлять свои собственные методы. Если класс расширяет интерфейс, но полностью не реализует его методы, то этот класс должен быть объявлен как **abstract**.

/ # 7 # реализация интерфейса # RectangleAction.java */*

```
package by.bsu.shapes.action;
import by.bsu.shapes.entity.AbstractShape;
import by.bsu.shapes.entity.Rectangle;
public class RectangleAction implements IShapeAction {
```

```
@Override // реализация метода из интерфейса
public double computeSquare(AbstractShape shape)  { // площадь прямоугольника
          double square = 0;
          // необходимо проверить тип
          if (shape instanceof Rectangle) {
          Rectangle rectangle = (Rectangle) shape;
          square = rectangle.getA() * rectangle.getB();
          } else {
          throw new IllegalArgumentException("Incompatible shape:"
                                                + shape.getClass());
          }
          return square;
          }
@Override // реализация метода из интерфейса
public double computePerimeter(AbstractShape shape) { // периметр прямоугольника
          double perimeter = 0;
          if (shape instanceof Rectangle) {
          Rectangle rectangle = (Rectangle) shape;
          perimeter = 2 * (rectangle.getA() + rectangle.getB());
          } else {
          throw new IllegalArgumentException("Incompatible shape:"
                                           + shape.getClass());
          }
          return perimeter;
          }
}
```

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

```
package by.bsu.shapes.action;
import by.bsu.shapes.entity.AbstractShape;
import by.bsu.shapes.entity.Triangle;
public class TriangleAction implements IShapeAction {
          @Override
          public double computeSquare(AbstractShape shape) {
                    double square = 0;
                    if (shape instanceof Triangle) {
                              Triangle triangle = (Triangle) shape;
                              square = 1 / 2 * triangle.getA() * triangle.getB()
                                        * Math.sin(triangle.getAngle());
                    } else {
                              throw new IllegalArgumentException("Incompatible shape"
                                                                  + shape.getClass());
                    }
                    return square;
          }
          @Override
          public double computePerimeter(AbstractShape shape) {
                    double perimeter = 0;
                    if (shape instanceof Triangle) {
```

```
                Triangle triangle = (Triangle) shape;
                perimeter = triangle.getA() + triangle.getB() + triangle.getC();
        } 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.

/ # 9 # неполная реализация интерфейса # PentagonAction.java */*

```
package by.bsu.shapes.action;
import by.bsu.shapes.entity.AbstractShape;
/* метод computeSquare() в данном абстрактном классе не реализован */
public abstract class PentagonAction implements IShapeAction {
        // поля, конструкторы
        @Override
        public double computePerimeter(AbstractShape shape) {
                // проверка и вычисление периметра
        }
}
```

Классы для определения фигур, используемые в интерфейсах:

/ # 10 # абстрактная фигура, прямоугольник, треугольник # AbstractShape.java # Rectangle.java # Triangle.java */*

```
package by.bsu.shapes.entity;
public abstract class AbstractShape {
        private double a;
        public AbstractShape(double a) {
                this.a = a;
        }
        public double getA() {
                return a;
        }
}
package by.bsu.shapes.entity;
public class Rectangle extends AbstractShape {
        private double b;
        public Rectangle(double a, double b) {
                super(a);
                this.b = b;
        }
```

```
        public double getB() {
                return b;
        }
}
package by.bsu.shapes.entity;
public class Triangle extends AbstractShape {
        private double b;
        private double angle; // угол между сторонами в радианах
        public Triangle(double a, double b, double angle) {
                super(a);
                this.b = b;
                this.angle = angle;
        }
        public double getAngle() {
                return angle;
        }
        public double getB() {
                return b;
        }
        public double getC() {
                double c = // stub : вычисление по теореме косинусов
                return c;
        }
}
```

/ # 11 # свойства ссылки на интерфейс # ActionMain.java */*

```
package by.bsu.shapes;
import by.bsu.shapes.action.IShapeAction;
import by.bsu.shapes.action.RectangleActionImpl;
import by.bsu.shapes.action.TriangleActionImpl;
import by.bsu.shapes.entity.AbstractShape;
import by.bsu.shapes.entity.Rectangle;
import by.bsu.shapes.entity.Triangle;
import static java.lang.Math.PI;
public class ActionMain {
   public static void main(String[ ] args) {
      IShapeAction action;
      try {
          Rectangle rectShape = new Rectangle(2, 3);
          action = new RectangleAction();
          System.out.println("Square rectangle: " + action.computeSquare(rectShape));
          System.out.println("Perimeter rectangle: " + action.computePerimeter(rectShape));

          Triangle trShape = new Triangle(3, 4, PI/6);
          action = new TriangleAction ();
          System.out.println("Square triangle: " + action.computeSquare(trShape));
          System.out.println("Perimeter triangle: " + action.computePerimeter(trShape));
          action.computePerimeter(rectShape); // ошибка времени выполнения
```

```
        } catch (IllegalArgumentException ex) {
            System.err.println(ex.getMessage());
        }
    }
}
```

В результате будет выведено:

Square rectangle: 6.0
Perimeter rectangle: 10.0
Square triangle: 3.0
Perimeter triangle: 8.0
Incompatible shape class by.bsu.shapes.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 (предыдущий пример) 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:

/ # 12 # параметризация интерфейса # IShapeAction.java */*

```
package by.bsu.shapes.action;
public interface IShapeAction <T extends AbstractShape> {
          double computeSquare(T shape);
          double computePerimeter(T shape);
}
```

Параметризованный тип **T extends AbstractShape** указывает, что в качестве параметра методов может использоваться только подкласс **AbstractShape**, что, в общем, мало чем отличается от случая, когда тип параметра метода указывался явно. Но когда дело доходит до реализации интерфейса, то при реализации

интерфейса указывается конкретный тип объектов, являющийся подклассом **AbstractShape**, которые будут обрабатываться методами данного класса, а в качестве параметра метода также прописывается тот же самый конкретный тип:

/ # 13 # реализация интерфейса с указанием типа параметра # RectangleAction.java # TriangleAction.java */*

```
package by.bsu.shapes.action;
public class RectangleAction implements IShapeAction<Rectangle> {
          @Override
          public double computeSquare(Rectangle shape) {
                    return shape.getA() * shape.getB();
          }
          @Override
          public double computePerimeter(Rectangle shape) {
                    return 2 * (shape.getA() + shape.getB());
          }
}
package by.bsu.shapes.action;
public class TriangleAction implements IShapeAction<Triangle> {
          @Override
          public double computeSquare(Triangle shape) {
                    return 0.5 * shape.getA() * shape.getB() * Math.sin(shape.getAngle());
          }
          @Override
          public double computePerimeter(Triangle shape) {
                    return shape.getA() + shape.getB() + shape.getC();
          }
}
```

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.

/ # 14 # использование параметризованных интерфейсов # ShapeMain.java */*

```
package by.bsu.shapes.action;
public class ShapeMain {
    public static void main(String[ ] args) {
          Rectangle rectShape = new Rectangle(2, 3);
          IShapeAction<Rectangle> rectAction = new RectangleAction();
          Triangle trShape = new Triangle(3, 4, PI / 6);
          IShapeAction<Triangle> trAction = new TriangleAction();
```

```
        System.out.println("Square rectangle: " + rectAction.computeSquare(rectShape));
        System.out.println("Perimeter rectangle: " + rectAction.computePerimeter(rectShape));
        System.out.println("Square triangle: " + trAction.computeSquare(trShape));
        System.out.println("Perimeter triangle: " + trAction.computePerimeter(trShape));
        // trAction.computePerimeter(rectShape); // ошибка компиляции
        }
}
```

Аннотации

Аннотации — это своего рода 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 SE определено несколько встроенных аннотаций, четыре типа — **@Retention**, **@Documented**, **@Target** и **@Inherited** — из пакета **java.lang.annotation**. Из оставшиеся 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.

В следующем коде приведено объявление аннотации.

/ # 15 # многочленная аннотация класса # BaseAction.java */*

```
@Target(ElementType.TYPE)
public @interface BaseAction {
        int level();
        String sqlRequest();
}
```

Ключевому слову **interface** предшествует символ **@**. Такая запись сообщает компилятору об объявлении аннотации. В объявлении также есть два метода-члена: **int level(), String sqlRequest().**

После объявления аннотации ее можно использовать для аннотирования объявлений класса. Объявление практически любого типа может иметь аннотацию, связанную с ним. Даже к аннотации можно добавить аннотацию. Во всех случаях аннотация предшествует объявлению.

Применяя аннотацию, нужно задавать значения для ее методов-членов, если при объявлении аннотации не было задано значение по умолчанию. Далее приведен фрагмент, в котором аннотация **BaseAction** сопровождает объявление класса:

```
/* # 16 # примитивное использование аннотации класса # Base.java */
@BaseAction (
          level = 2,
          sqlRequest = "SELECT * FROM phonebook"
          )
public class Base {
          public void doAction() {
                              Class<Base> f = Base.class;
                              BaseAction a = (BaseAction)f.getAnnotation(BaseAction.class);
                              System.out.println(a.level());
                              System.out.println(a.sqlRequest());
          }
}
```

Данная аннотация помечает класс **Base**. За именем аннотации, начинающимся с символа **@**, следует заключенный в круглые скобки список инициализирующих значений для методов-членов. Для того, чтобы передать значение методу-члену, имени этого метода присваивается значение. Таким образом, в приведенном фрагменте строка **"SELECT * FROM phonebook"** присваивается методу **sqlRequest(),** члену аннотации типа **BaseAction**. При этом в операции присваивания после имени **sqlRequest** нет круглых скобок. Когда методу-члену передается инициализирующее значение, используется только имя метода. Следовательно, в данном контексте методы-члены выглядят как поля.

Все аннотации содержат только объявления методов, добавлять тела этим методам не нужно, так как их реализует сам язык. Кроме того, эти методы не могут содержать параметров секции **throws** и действуют скорее как поля. Допустимые типы возвращаемого значения: базовые типы, **String**, **Enum**, **Class** и массив любого из вышеперечисленных типов.

Все типы аннотаций автоматически расширяют интерфейс **Annotation** из пакета **java.lang.annotation**. В этом интерфейсе приведен метод **annotationType(),** который возвращает объект типа **Class**, представляющий вызывающую аннотацию.

Если необходим доступ к аннотации в процессе функционирования приложения, то перед объявлением аннотации задается правило сохранения **RUNTIME** в виде кода

```
import java.lang.annotation.Retention;
import java.lang.annotation.RetentionPolicy;
@Retention(RetentionPolicy.RUNTIME)
```

предоставляющее максимальную продолжительность существования аннотации.

С правилом **SOURCE** аннотация существует только в исходном тексте программы и отбрасывается во время компиляции. Аннотация с правилом сохранения **CLASS** помещается в процессе компиляции в файл *имя***.class**, но недоступна в JVM во время выполнения.

Основные типы аннотаций: аннотация-маркер, одночленная и многочленная.

Аннотация-маркер не содержит методов-членов. Цель — пометить объявление. В этом случае достаточно присутствия аннотации. Поскольку у интерфейса аннотации-маркера нет методов-членов, достаточно определить наличие аннотации:

```
public @interface MarkerAnnotation {}
```

Для проверки наличия аннотации используется метод **isAnnotationPresent().**

Одночленная аннотация содержит единственный метод-член. Для этого типа аннотации допускается краткая условная форма задания значения для метода-члена. Если есть только один метод-член, то просто указывается его значение при создании аннотации. Имя метода-члена указывать не нужно. Но для того, чтобы воспользоваться краткой формой, следует для метода-члена использовать имя **value().**

Многочленные аннотации содержат несколько методов-членов. Поэтому используется полный синтаксис (*имя_параметра = значение*) для каждого параметра.

Реализация обработки аннотации, приведенная в методе **doAction()** класса **Base**, крайне примитивна. Разработчику каждый раз при использовании аннотаций придется писать код по извлечению значений полей-членов и реакцию метода, класса и поля на значение аннотации. Необходимо привести реализацию аннотации таким образом, чтобы программисту достаточно было только аннотировать класс, метод или поле и передать нужное значение. Реакция системы на аннотацию должна быть автоматической. Ниже приведен пример работы с аннотацией, реализованной на основе ReflectionAPI. Примеры корректного использования аннотаций короткими не бывают.

Аннотация **BankingAnnotation** предназначена для задания уровня проверки безопасности при вызове метода.

/ # 17 # одночленная аннотация метода # BankingAnnotation.java */*

```
package by.bsu.proj.annotation;
import java.lang.annotation.ElementType;
import java.lang.annotation.Retention;
import java.lang.annotation.RetentionPolicy;
import java.lang.annotation.Target;
@Retention(RetentionPolicy.RUNTIME)
@Target(ElementType.METHOD)
public @interface BankingAnnotation {
          SecurityLevelEnum securityLevel() default SecurityLevelEnum.NORMAL;
}
```

/ # 18 # возможные значения уровней безопасности # SecurityLevelEnum.java */*

```
package by.bsu.proj.annotation;
public enum SecurityLevelEnum {
          LOW, NORMAL, HIGH
}
```

Методы класса логики системы выполняют действия с банковским счетом. В зависимости от различных причин конкретным реализациям интерфейса **AccountOperationManager** могут понадобиться дополнительные действия помимо основных.

/ # 19 # аннотирование методов # AccountOperationManagerImpl.java */*

```
package by.bsu.proj.accountlogic;
import by.bsu.proj.annotation.BankingAnnotation;
import by.bsu.proj.annotation.SecurityLevelEnum;
public class AccountOperationManagerImpl implements AccountOperationManager {
          @BankingAnnotation(securityLevel = SecurityLevelEnum.HIGH)
          public double depositInCash(int accountNumber, int amount) {
                    // зачисление на депозит
                    return 0; // stub
          }
          @BankingAnnotation(securityLevel = SecurityLevelEnum.HIGH)
          public boolean withdraw(int accountNumber, int amount) {
                    // снятие суммы, если не превышает остаток
                    return true; // stub
          }
          @BankingAnnotation(securityLevel = SecurityLevelEnum.LOW)
          public boolean convert(double amount) {
                    // конвертировать сумму
                    return true; // stub
          }
          @BankingAnnotation
          public boolean transfer(int accountNumber, double amount) {
                    // перевести сумму на счет
                    return true; // stub
          }
}
```

/ # 20 # конфигурирование и запуск # AnnoRunner.java */*

```
package by.bsu.proj.run;
import by.bsu.proj.accountlogic.AccountOperationManager;
import by.bsu.proj.accountlogic.AccountOperationManagerImpl;
import by.bsu.proj.annotation.logic.SecurityFactory;
public class AnnoRunner {
          public static void main(String[] args) {
                    AccountOperationManager account = new AccountOperationManagerImpl();
                    // "регистрация класса" для включения аннотаций в обработку.
                    AccountOperationManager securityAccount =
                                                  SecurityFactory.createSecurityObject(account);
                    securityAccount.depositInCash(10128336, 6);
                    securityAccount.withdraw(64092376, 2);
```

```
                securityAccount.convert(200);
                securityAccount.transfer(64092376, 300);
        }
}
```

Подход с вызовом некоторого промежуточного метода для включения в обработку аннотаций достаточно распространен и, в частности, используется в технологии JPA.

Вызов метода **createSecurityObject(),** регистрирующего методы класса для обработки аннотаций, можно разместить в конструкторе некоторого промежуточного абстрактного класса, реализующего интерфейс **AccountOperationManager**, или в статическом логическом блоке самого интерфейса. Тогда реагировать на аннотации будут все методы всех реализаций интерфейса **AccountOperationManager.**

Класс, определяющий действия приложения в зависимости от значения **SecurityLevelEnum**, передаваемого в аннотированный метод.

/ # 21 # логика обработки значения аннотации # SecurityLogic.java */*

```
package by.bsu.proj.annotation.logic;
import java.lang.reflect.Method;
import by.bsu.proj.annotation.SecurityLevelEnum;
public class SecurityLogic {
        public void onInvoke(SecurityLevelEnum level, Method method, Object[] args) {
                StringBuilder argsString = new StringBuilder();
                for (Object arg : args) {
                        argsString.append(arg.toString() + " :");
                }
                argsString.setLength(argsString.length() - 1);
                System.out.println(String.format(
                        "Method %S was invoked with parameters : %s", method.getName(),
                                argsString.toString()));
                switch (level) {
                  case LOW:
                       System.out.println("Низкий уровень проверки безопасности: " + level);
                          break;
                  case NORMAL:
                       System.out.println("Обычный уровень проверки безопасности: " + level);
                          break;
                  case HIGH:
                       System.out.println("Высокий уровень проверки безопасности: " + level);
                          break;
                }
        }
}
```

Статический метод **createSecurityObject(Object targetObject)** класса-фабрики создает для экземпляра класса **AccountOperationManagerImpl** экземпляр-заместитель, обладающий кроме всех свойств оригинала возможностью

неявного обращения к логике, выбор которой определяется выбором значения для поля аннотации.

/ # 22 # создание прокси-экземпляра, включающего функциональность SecurityLogic # SecurityFactory.java */*

```
package by.bsu.proj.annotation.logic;
import java.lang.reflect.InvocationHandler;
import java.lang.reflect.InvocationTargetException;
import java.lang.reflect.Method;
import java.lang.reflect.Proxy;
import by.bsu.proj.accountlogic.AccountOperationManager;
import by.bsu.proj.annotation.BankingAnnotation;
public class SecurityFactory {
    public static AccountOperationManager createSecurityObject(
                            AccountOperationManager targetObject) {
          return (AccountOperationManager)Proxy.newProxyInstance(
          targetObject.getClass().getClassLoader(),
          targetObject.getClass().getInterfaces(),
          new SecurityInvokationHandler(targetObject));
          }
          private static class SecurityInvokationHandler implements InvocationHandler {
          private Object targetObject = null;
          public SecurityInvokationHandler(Object targetObject) {
          this.targetObject = targetObject;
          }
                    public Object invoke(Object proxy, Method method, Object[ ] args)
                                        throws Throwable {
                    SecurityLogic logic = new SecurityLogic();
                    Method realMethod = targetObject.getClass().getMethod(
                                        method.getName(),
                                        (Class[]) method.getGenericParameterTypes());

                    BankingAnnotation annotation = realMethod
                                        .getAnnotation(BankingAnnotation.class);
                    if (annotation != null) {
                     // доступны и аннотация и параметры метода
                    logic.onInvoke(annotation.securityLevel(), realMethod, args);
                    try {
                    return method.invoke(targetObject, args);
                    } catch (InvocationTargetException e) {
                    System.out.println(annotation.securityLevel());
                    throw e.getCause();
                    }
                    } else {
                    /* в случае если аннотирование метода обязательно следует
                    генерировать исключение на факт ее отсутствия */
                    /* throw new InvocationTargetException(null, "method "
                    + realMethod + " should be annotated "); */
```

```
                    // в случае если допустимо отсутствие аннотации у метода
                    return null;
                    }
            }
    }
}
```

Использование аннотации может быть обязательным или опциональным. Если метод **invoke()** при отсутствии аннотации у метода генерирует исключение, то это говорит об обязательности явного использования аннотации всеми без исключения методами класса. Если метод возвращает в этой ситуации значение **null**, то аннотированными могут быть только те методы, которые необходимы программисту. Последний вариант использования предпочтителен, так как не требует обязательного использования аннотации для методов общего назначения, например: **equals()** или **toString().**

Задания к главе 6

Вариант А

Создать и реализовать интерфейсы, также использовать наследование и полиморфизм для следующих предметных областей:

1. interface Издание ← abstract class Книга ← class Справочник и Энциклопедия.
2. interface Абитуриент ← abstract class Студент ← class Студент-Заочник.
3. interface Сотрудник ← class Инженер ← class Руководитель.
4. interface Здание ← abstract class Общественное Здание ← class Театр.
5. interface Mobile ← abstract class Siemens Mobile ← class Model.
6. interface Корабль ← abstract class Военный Корабль ← class Авианосец.
7. interface Врач ← class Хирург ← class Нейрохирург.
8. interface Корабль ← class Грузовой Корабль ← class Танкер.
9. interface Мебель ← abstract class Шкаф ← class Книжный Шкаф.
10. interface Фильм ← class Отечественный Фильм ← class Комедия.
11. interface Ткань ← abstract class Одежда ← class Костюм.
12. interface Техника ← abstract class Плеер ← class Видеоплеер.
13. interface Транспортное Средство ← abstract class Общественный Транспорт ← class Трамвай.
14. interface Устройство Печати ← class Принтер ← class Лазерный Принтер.
15. interface Бумага ← abstract class Тетрадь ← class Тетрадь Для Рисования.
16. interface Источник Света ← class Лампа ← class Настольная Лампа.