

Working with Abstraction

Architecture, Refactoring and Enumerations



SoftUni Team
Technical Trainers



SoftUni



Software University

<https://softuni.bg>

sli.do

#java-advanced

1. **Project** Architecture

- Methods
- Classes
- Projects

2. **Code** Refactoring

3. **Enumerations**

4. **Static** Keyword

5. **Java** Packages





Project Architecture

Splitting Code into Methods

- We use **methods** to split code into functional blocks
 - Improves code **readability**
 - Allows for easier **debugging**

```
for (char move : moves){  
    for (int r = 0; r < room.length; r++)  
        for (int c = 0; c < room[r].length; c++)  
            if (room[move][c] == 'b')  
                ...  
}
```



```
for (char move : moves) {  
    moveEnemies();  
    killerCheck();  
    movePlayer(move);  
}
```

Splitting Code into Methods


- **Methods** let us easily **reuse** code
- We change the **method** once to affect **all calls**

```
BankAccount bankAcc = new BankAccount();  
bankAcc.setId(1);  
bankAcc.deposit(20);  
System.out.printf("Account %d, balance %d",  
                  bankAcc.getId(), bankAcc.getBalance());  
  
bankAcc.withdraw(10);  
...  
System.out.println(bankAcc.toString());
```


Override **.toString()** to
set a global printing format

Splitting Code into Methods

- A **single** method should complete a **single task**



```
void doMagic ( ... )  
void depositOrWithdraw ( ... )  
BigDecimal depositAndGetBalance ( ... )  
String parseDataAndReturnResult ( ... )
```



```
void withdraw ( ... )  
void deposit ( ... )  
BigDecimal getBalance ( ... )  
string toString ( ... )
```

Problem: Rhombus of Stars

- Draw on the console a rhombus of stars with size **n**

n = 3



```
*  
* *  
* * *  
* *  
*
```

n = 2



```
*  
* *  
*
```

n = 1



```
*
```



Check your solution here: <https://alpha.judge.softuni.org/contests/1575>

Solution: Rhombus of Stars

```
int size = Integer.parseInt(sc.nextLine());
for (int starCount = 1; starCount <= size; starCount++) {
    printRow(size, starCount);
}
for (int starCount = size - 1; starCount >= 1; starCount--) {
    printRow(size, starCount);
}
```

Reusing code

Solution: Rhombus of Stars

```
static void printRow(int figureSize, int starCount) {  
    for (int i = 0; i < figureSize - starCount; i++)  
        System.out.print(" ");  
    for (int col = 1; col < starCount; col++) {  
        System.out.print("* ");  
    }  
    System.out.println("*");  
}
```

Splitting Code into Classes

- Just like methods, **classes** should **not** know or do too much

```
GodMode master = new GodMode();  
int[] numbers = master.parseAny(input);  
...  
int[] numbers2 = master.copyAny(numbers);  
master.printToConsole(master.getDate());  
master.printToConsole(numbers);
```



Splitting Code into Classes

- We can also break our code up logically into **classes**
 - Hiding implementation
 - Allow us to change the output destination
 - Helps us to avoid repeating code

Splitting Code into Classes

```
List<Integer> input = Arrays.stream(
    sc.nextLine().split(" "))
    .map(Integer::parseInt)
    .collect(Collectors.toList());

...
String result = input.stream()
    .map(String::valueOf)
    .collect(Collectors.joining(", "));
System.out.println(result);
```



```
ArrayParser parser = new ArrayParser();
OuputWriter printer = new OuputWriter();
int[] numbers = parser.integersParse(args);
int[] coordinates = parser.integerParse(args1);
printer.printToConsole(numbers);
```

Problem: Point in Rectangle

- Create a Point class holding the horizontal and vertical coordinates
- Create a **Rectangle class**
 - Holds 2 **points**
 - **Bottom left** and **top right**
- Add **Contains** method
 - Takes a **Point** as an argument
 - **Returns** it if it's inside the current object of the **Rectangle class**

Solution: Point in Rectangle

```
public class Point {  
    private int x;  
    private int y;  
    //TODO: Add getters and setters  
}  
  
public class Rectangle {  
    private Point bottomLeft;  
    private Point topRight;  
    //TODO: getters and setters  
    public boolean contains(Point point) {  
        //TODO: Implement  
    }  
}
```

Solution: Point in Rectangle

```
public boolean contains(Point point)
{
    boolean isInHorizontal =
        this.bottomLeft.getX() <= point.getX() &&
        this.topRight.getX() >= point.getX();

    boolean isInVertical =
        this.bottomLeft.getY() <= point.getY() &&
        this.topRight.getY() >= point.getY();

    boolean isInRectangle = isInHorizontal &&
                             isInVertical;

    return isInRectangle;
}
```




Refactoring

Refactoring

- **Restructures** code without changing the behaviour
- **Improves** code readability
- **Reduces** complexity



```
class ProblemSolver { public static void doMagic() { ... } }
```



```
class CommandParser {  
    public static <T> Function<T, T> parseCommand() { ... } }  
class DataModifier { public static <T> T execute() { ... } }  
class OutputFormatter { public static void print() { ... } }
```

- **Breaking code** into reusable units
- **Extracting parts of methods** and **classes** into **new** ones

```
depositOrWithdraw()
```



```
deposit()  
withdraw()
```

- **Improving names** of variables, methods, classes, etc.

```
String str;
```



```
String name;
```

- **Moving methods** or **fields** to more appropriate classes

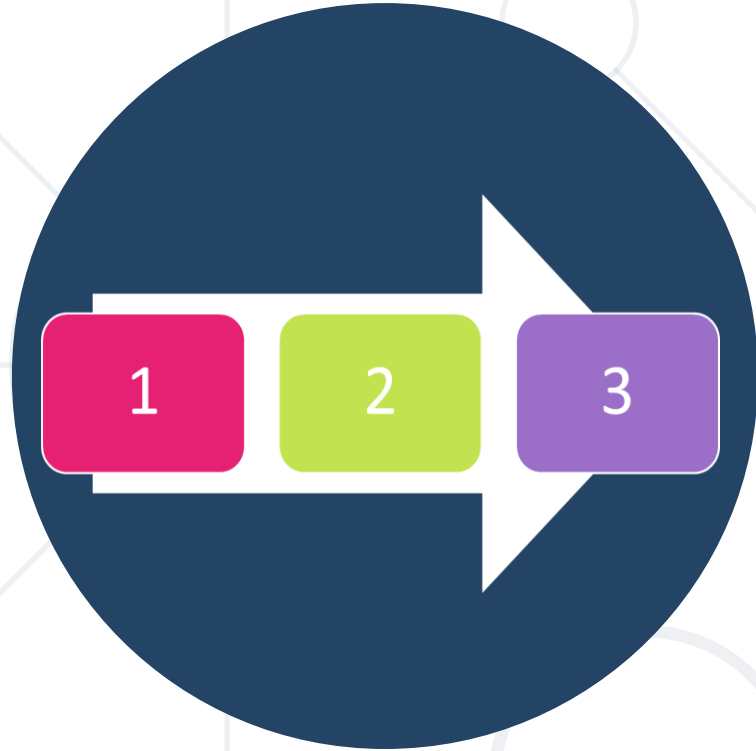
```
Car.open()
```



```
Door.open()
```

Problem: Student System

- You are given a **working** Student System project to refactor
- **Break it up** into smaller functional units and make sure it works
- It supports the following **commands**:
 - **"Create {studentName} {studentAge} {studentGrade}"**
 - creates a new student
 - **"Show {studentName}"**
 - prints information about a student
 - **"Exit"**
 - closes the program



Enumerations

Enumerations

- Represent a numeric value from a fixed set as a text
- We can use them to pass **arguments** to **methods** without making code confusing

```
enum Day {MON, TUE, WED, THU, FRI, SAT, SUN}
```

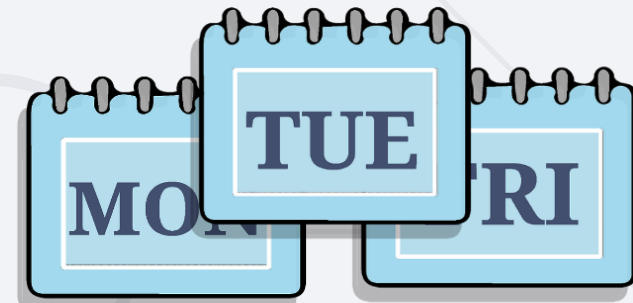
```
getDailySchedule(0) → getDailySchedule(Day.MON)
```

- By default, **enums** start at 0
- Every next value is incremented by 1



- We can **customize** enum **values**

```
enum Day {  
    MON(1),TUE(2),WED(3),THU(4),FRI(5),SAT(6),SUN(7);  
  
    private int value;  
  
    Day(int value) {  
        this.value = value;  
    }  
}  
  
System.out.println(Day.Sat); // SAT
```



- We can **customize** enum **values**

```
enum CoffeeSize {  
    SMALL(100), NORMAL(150), DOUBLE(300);  
    private int size;  
    CoffeeSize(int size) {  
        this.size = size;  
    }  
    public int getValue() { return this.size; }  
}  
  
System.out.println(CoffeeSize.SMALL.getValue()); // 100
```


Problem: Hotel Reservation

- Create a class PriceCalculator that calculates the total price of a holiday, by given **price per day**, **number of days**, **the season** and a **discount type**
- The discount type and season should be **enums**
- The price multipliers will be:
 - 1x for Autumn, 2x for Spring, etc.
- The discount types will be:
 - None – 0%
 - SecondVisit – 10%
 - VIP – 20%



Solution: Hotel Reservation

```
public enum Season {  
    SPRING(2), SUMMER(4), AUTUMN(1), WINTER(3);  
    private int value;  
    Season(int value) {  
        this.value = value;  
    }  
    public int getValue() {  
        return this.value;  
    }  
}
```

Solution: Hotel Reservation

```
public enum Discount {  
    NONE(0), SECOND_VISIT(10), VIP(20);  
    private int value;  
    Discount(int value) {  
        this.value = value;  
    }  
    public int getValue() {  
        return this.value;  
    }  
}
```

Solution: Hotel Reservation

```
public class PriceCalculator {  
    public static double calculatePrice(double pricePerDay,  
        int numberOfDays, Season season, Discount discount) {  
        int multiplier = season.getValue();  
        double discountMultiplier = discount.getValue() / 100.0;  
        double priceBeforeDiscount = numberOfDays * pricePerDay * multiplier;  
        double discountedAmount = priceBeforeDiscount * discountMultiplier;  
        return priceBeforeDiscount - discountedAmount;  
    }  
}
```



Static Keyword in Java


Static Keyword

- Used for **memory management** mainly
- Can apply with:
 - Nested class
 - Variables
 - Methods
 - Blocks
- Belongs to the class than an instance of the class

```
static int count;  
static void increaseCount() {  
    count++;  
}
```



Static Class

- 
- A **top level** class is a class that is not a nested class
 - A **nested** class is any class whose declaration occurs within the body of another class or interface
 - Only nested classes can be **static**

```
class TopClass {  
    static class NestedStaticClass {  
  
    }  
}
```

Static Variable

- Can be used to refer to the **common** variable of all objects
- Example
 - The company name of employees
 - College name of students
 - The name of the college is common for all students
- Allocate memory only once in the class area at the time of class loading



Example: Static Variable

```
class Counter {  
    int count = 0;    static int staticCount = 0;  
    public Counter() {  
        count++;      // incrementing value  
        staticCount++; // incrementing value  
    }  
    public void printCounters() {  
        System.out.printf("Count: %d\n", count);  
        System.out.printf("Static Count: %d\n", staticCount);  
    }  
}
```

Example: Static Variable

// Inside the Main Class

```
public static void main(String[] args) {  
    Counter c1 = new Counter();  
    c1.printCounters();  
    Counter c2 = new Counter();  
    c2.printCounters();  
    Counter c3 = new Counter();  
    c3.printCounters();  
    int counter = Counter.staticCount; // 3  
}
```



```
Count: 1  
Static Count: 1  
Count: 1  
Static Count: 2  
Count: 1  
Static Count: 3
```

Static Method

- Belongs to the class rather than the object of a class
- Can be **invoked** without the need for creating an instance of a class
- Can **access** static data member and can **change** the value of it
- Can **not use non-static** data member or call a **non-static method** directly
- **this** and **super** cannot be used in a static context



Example: Static Method

```
class Calculate {  
    static int cube(int x) { return x * x * x; }  
    public static void main(String args[]) {  
        int result = Calculate.cube(5);  
        System.out.println(result);           // 125  
        System.out.println(Math.pow(2, 3));  // 8.0  
    }  
}
```

Static Block



- A set of **statements**, which will be executed by the JVM before execution of the **main** method
- Executing **static block** is at the time of class loading
- A class can take any number of the static block but all blocks will be executed **from top to bottom**

Example: Static Block

```
class Main {  
    static int n;  
    public static void main(String[] args) {  
        System.out.println("From main");  
        System.out.println(n);  
    }  
    static {  
        System.out.println("From static block");  
        n = 10;  
    }  
}
```



From static block
From main
10




Packages

Packages in Java

- Used to group related classes
 - Like a folder in a file directory
- Use packages to avoid name conflicts and to write a better maintainable code
- Packages are divided into two categories:
 - **Built-in Packages** (packages from the **Java API**)
 - User-defined Packages (create own packages)



Build-In Packages

- 
- The library is divided into packages and classes
 - Import a single class or a whole package that contain all the classes
 - To use a class or a package, use the import keyword
 - The complete list can be found at Oracles website:

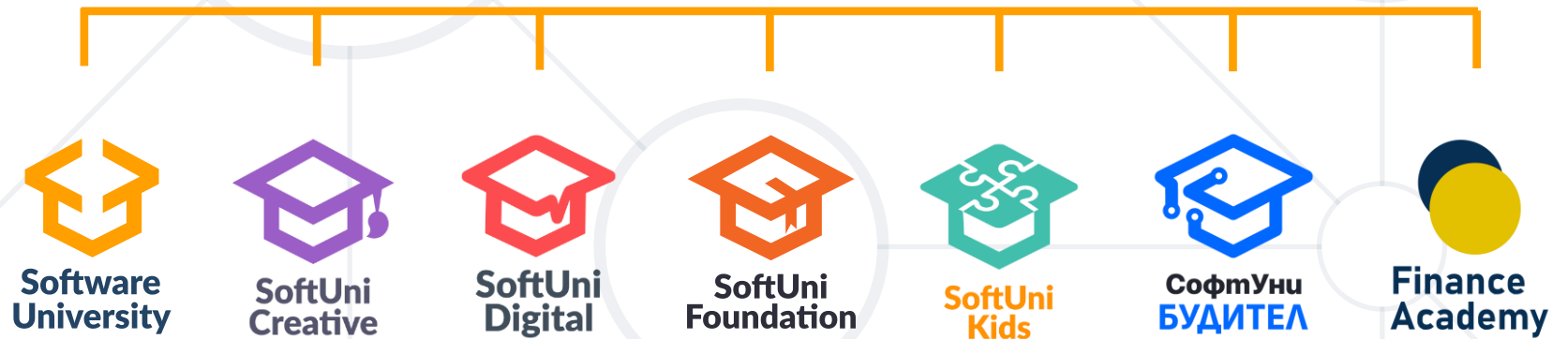
<https://docs.oracle.com/en/java/javase/>

```
import package.name.Class; // Import a single class
import package.name.*;    // Import the whole package
```

- Well organized code is easier to work with
- We can reduce complexity using **Methods, Classes** and **Projects**
- We can refactor existing code by **breaking code down**
- **Enumerations** define a fixed **set of constants**
 - Represent **numeric values**
 - We can easily **cast enums** to **numeric types**
- **Static** members and **Packages**



Questions?



SoftUni Diamond Partners



- Software University – High-Quality Education, Profession and Job for Software Developers

- softuni.bg

- Software University Foundation

- softuni.foundation

- Software University @ Facebook

- facebook.com/SoftwareUniversity



- This course (slides, examples, demos, exercises, homework, documents, videos and other assets) is **copyrighted content**
- Unauthorized copy, reproduction or use is illegal
- © SoftUni – <https://about.softuni.bg>
- © Software University – <https://softuni.bg>

