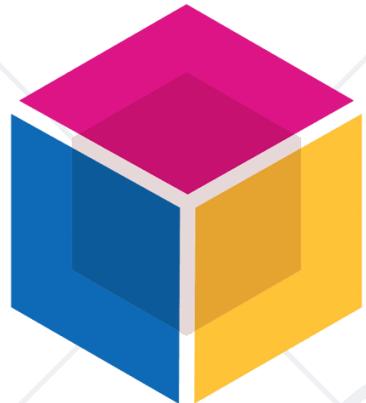


Working with Abstraction

Architecture, Refactoring and Enumerations



SoftUni Team

Technical Trainers



SoftUni



Software University
<https://softuni.bg>

Have a Question?



sli.do

#java-advanced

Table of Contents

1. Project Architecture

- Methods
- Classes
- Projects

2. Code Refactoring

3. Enumerations

4. Static Keyword

5. Java Packages



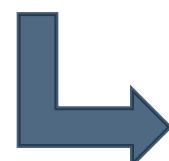


Project Architecture

Splitting Code into Methods (1)

- 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[row][col] == 'b')  
                ...  
}
```



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

Splitting Code into Methods (2)

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

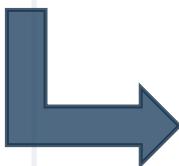
```
BankAccount bankAcc = new BankAccount();  
  
bankAcc.setId(1);  
  
bankAcc.deposit(20);  
  
System.out.printf("Account %d, balance %d",  
                  bankAcc.getId(), bankAcc.getBalance());  
  
bankAcc.withdraw(10);  
...  
  
System.out.println(bankAcc.toString());
```

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

Splitting Code into Methods (3)

- 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

$n = 3$



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

$n = 2$

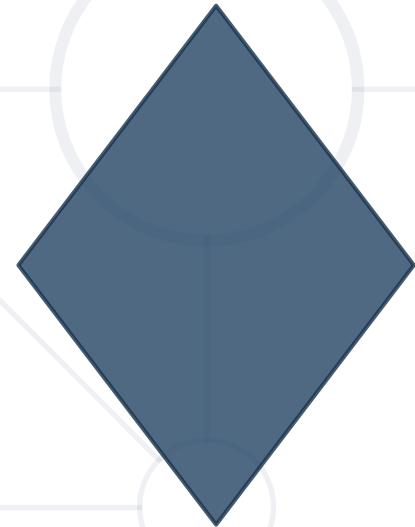


```
*
* *
* *
* 
```

$n = 1$



```
*
```



Solution: Rhombus of Stars (1)

```
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 (2)

```
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 (1)

- 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 (2)

- 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 (3)

```
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();  
OutputWriter printer = new OutputWriter();  
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 (2)

```
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() { ... } }
```



Refactoring Techniques

- **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;`

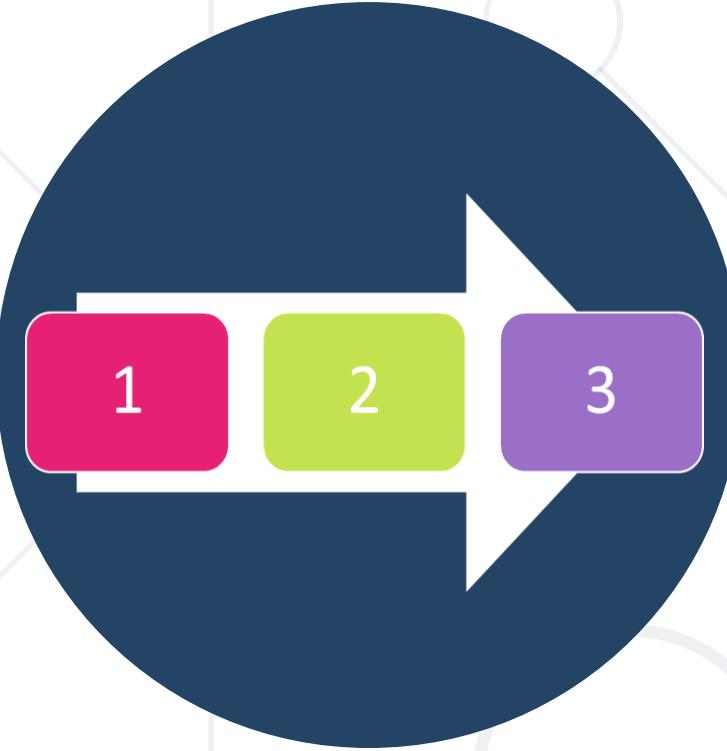
- **Moving methods or fields** to more appropriate classes

`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)
```

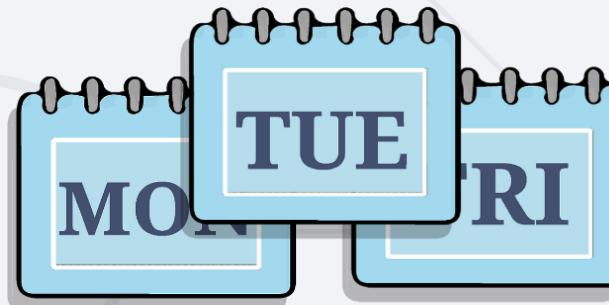
- By default, **enums** start at 0
- Every next value is incremented by 1



Enumerations (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
```



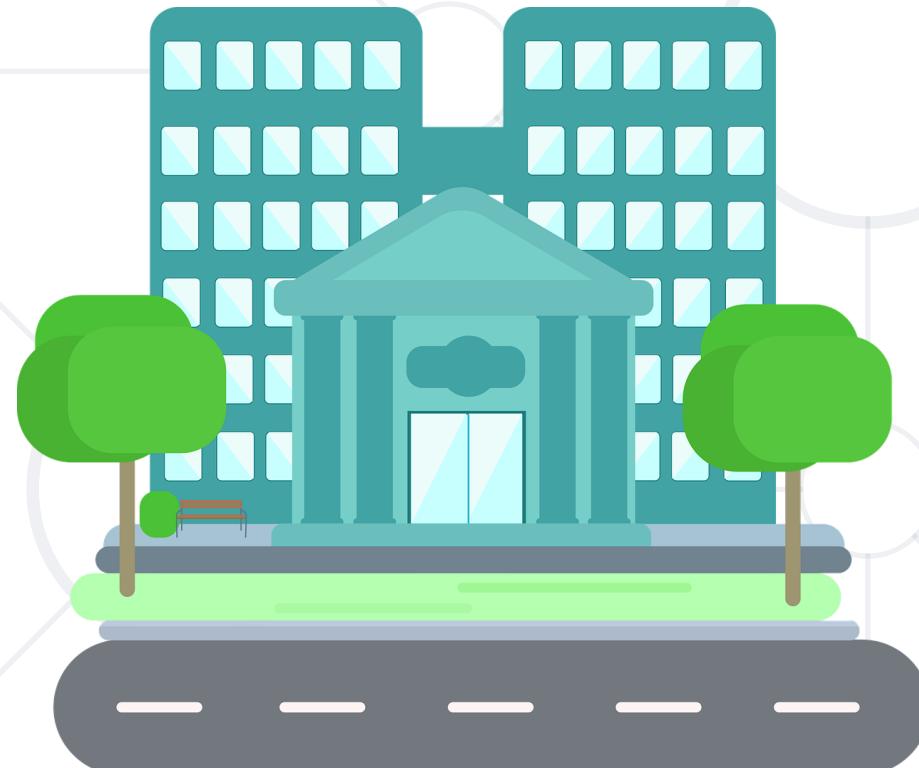
Enumerations (2)

- 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 (1)

```
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 (2)

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

Solution: Hotel Reservation (3)

```
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

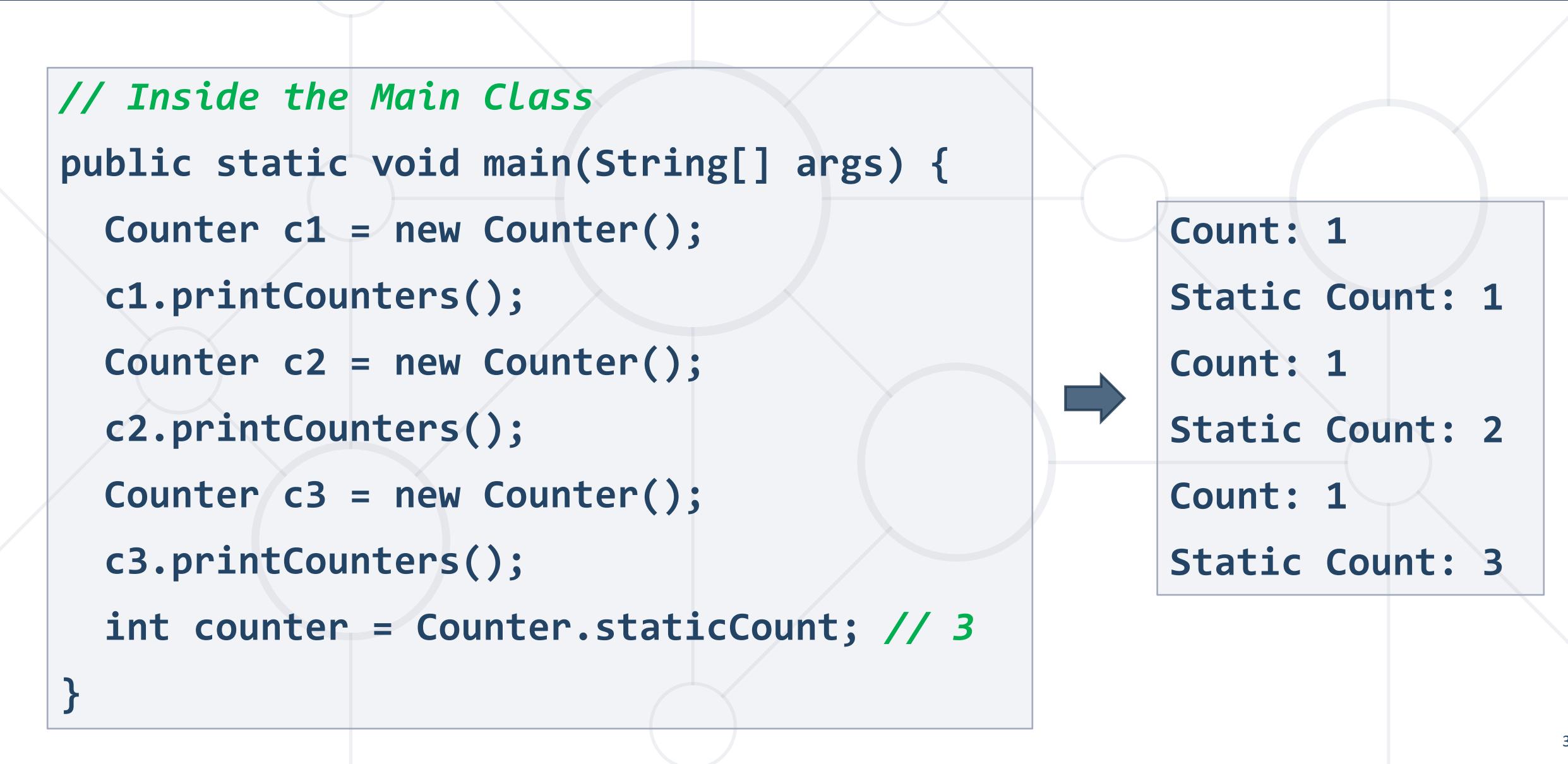


Example: Static Variable (1)

```
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 (2)

```
// 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
```

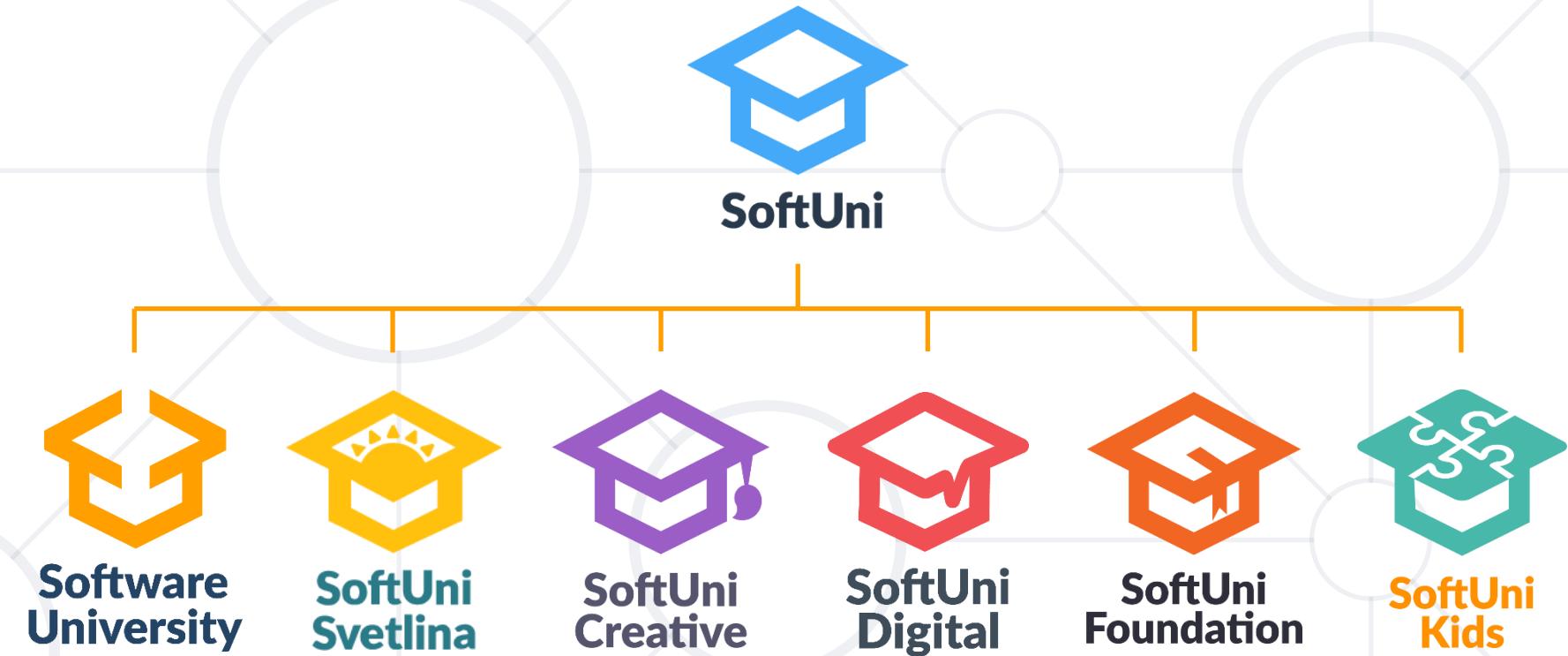


Summary

- 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



SCHWARZ



Coca-Cola HBC
Bulgaria



Postbank

Решения за твоето упре



Bosch.IO



SmartIT



POKERSTARS



CAREERS



AMBITIONED

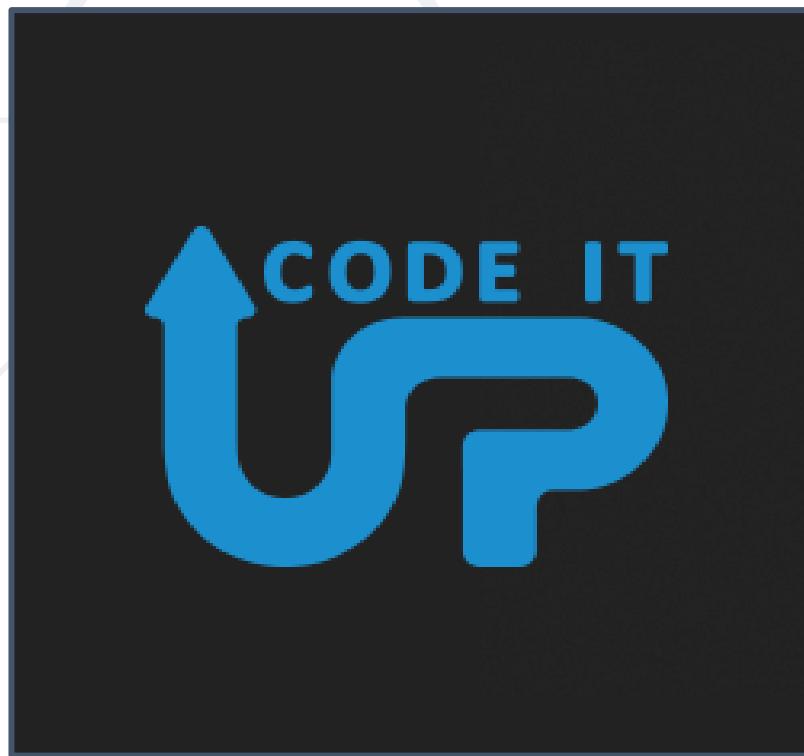


INDEAVR
Serving the high achievers



**SUPER
HOSTING
.BG**

Educational Partners



Trainings @ Software University (SoftUni)



- Software University – High-Quality Education, Profession and Job for Software Developers
 - softuni.bg
- Software University Foundation
 - softuni.foundation
- Software University @ Facebook
 - facebook.com/SoftwareUniversity
- Software University Forums
 - forum.softuni.bg



Software
University

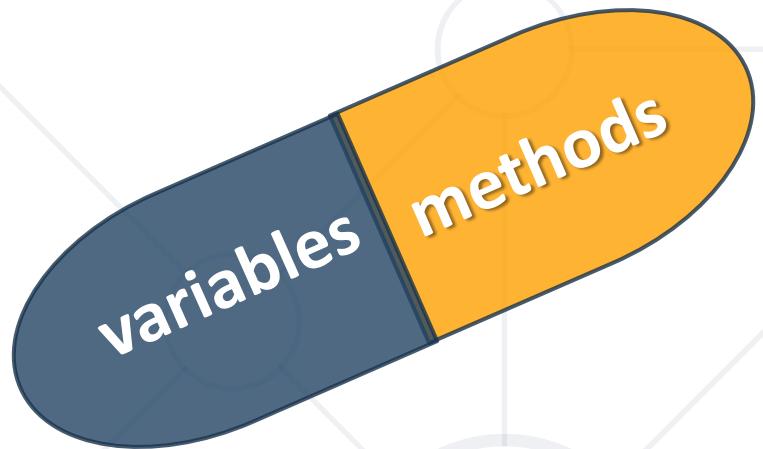


- This course (slides, examples, demos, exercises, homework, documents, videos and other assets) is **copyrighted content**
- Unauthorized copy, reproduction or use is illegal
- © SoftUni – <https://about.softuni.bg>
- © Software University – <https://softuni.bg>



Encapsulation

Benefits of Encapsulation



SoftUni Team
Technical Trainers



SoftUni



Software University
<https://softuni.bg>

Table of Contents

1. What is Encapsulation?
 - Keyword **this**
2. Access Modifiers
3. Validation
4. Mutable and Immutable Objects
5. Keyword **final**



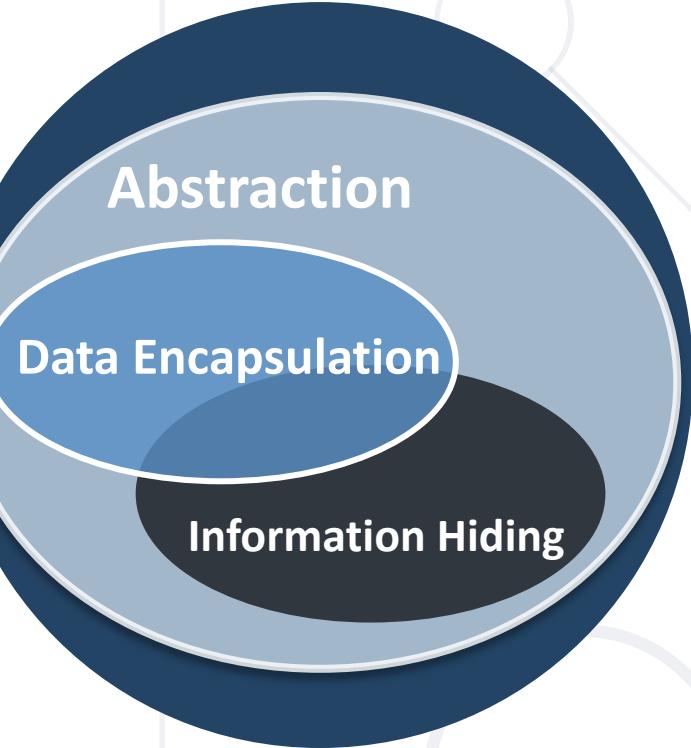
Have a Question?



sli.do

#java-advanced

Hiding Implementation



Encapsulation

- Process of wrapping code and data together into a single unit
- Flexibility and extensibility of the code
- Reduces **complexity**
- Structural changes remain **local**
- Allows **validation** and **data binding**



Encapsulation

- Objects fields **must be private**

```
class Person {  
    private int age;  
}
```



- Use **getters** and **setters** for data access

```
class Person {  
    public int getAge();  
    public void setAge(int age)  
}
```



Encapsulation – Example

- Fields should be **private**



Person

-name: string
-age: int

- == private

+Person(String name, int age)
+getName(): String
+setName(String name): void
+getAge(): int
+setAge(int age): void

+ == public

- Accessors and Mutators should be **public**

Keyword This (1)

- **this** is a reference to the **current object**
- **this** can refer to current class instance

```
public Person(String name) {  
    this.name = name;  
}
```

- **this** can invoke the current class method

```
public String fullName() {  
    return this.getFirstName() + " " + this.getLastName();  
}
```

Keyword This (2)

- **this** can invoke a current class constructor

```
public Person(String name) {  
    this.firstName = name;  
}
```

```
public Person (String name, Integer age) {  
    this(name);  
    this.age = age;  
}
```



Access Modifiers

Private Access Modifier

- Object hides data from the outside world

```
class Person {  
    private String name;  
    Person (String name) {  
        this.name = name;  
    }  
}
```

- Classes and interfaces **cannot** be private
- Data can be **accessed only within the declared class itself**



Protected Access Modifier

- Grants **access to subclasses**

```
class Team {  
    protected String getName () {...}  
    protected void setName (String name) {...}  
}
```

- The **protected** modifier cannot be applied to classes and interfaces
- Prevents a **nonrelated** class from trying to use it



Default Access Modifier

- Do not explicitly declare an access modifier

```
class Team {  
    String getName() {...}  
    void setName(String name) {...}  
}
```

- Available to any other class in the same package

```
Team real = new Team("Real");  
real.setName("Real Madrid");  
System.out.println(real.getName());  
                                // Real Madrid
```



Public Access Modifier

- Grants access to **any class** belonging to the **Java Universe**

```
public class Team {  
    public String getName() {...}  
    public void setName(String name) {...}  
}
```

- Import a package if you need to use a class
- The **main()** method of an application must be **public**



Problem: Sort by Name and Age

- Create a class **Person**

Person

```
-firstName: String  
-lastName: String  
-age: int  
  
+getFirstName(): String  
+getLastName(): String  
+getAge(): int  
+toString(): String
```

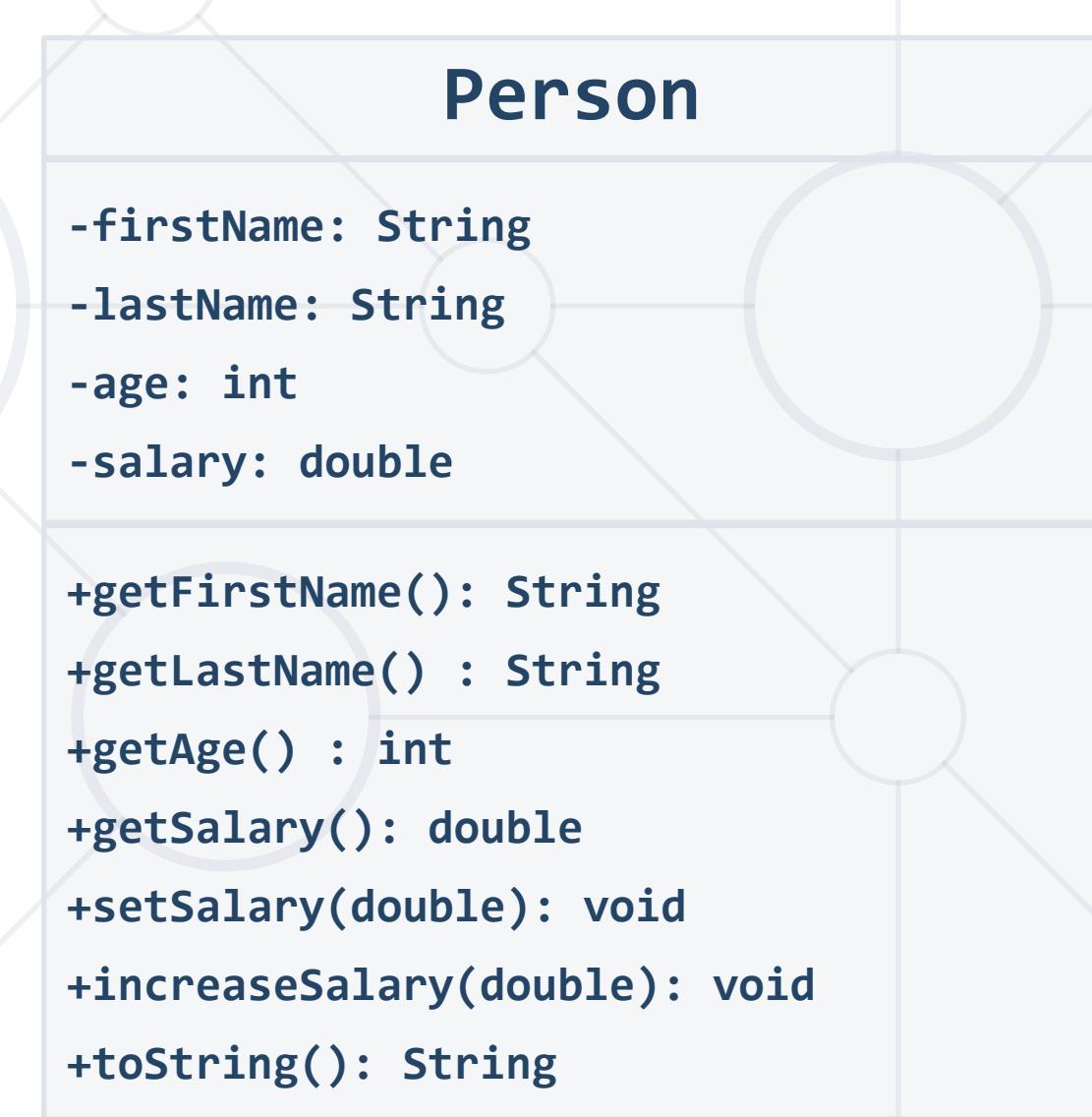
```
Collections.sort(persons, (firstPerson, secondPerson) -> {  
    int sComp = firstPerson  
        .getFirstName()  
        .compareTo(secondPerson.getFirstName());  
  
    if (sComp != 0) {  
        return sComp;  
    } else {  
        return firstPerson  
            .getAge()  
            .compareTo(secondPerson.getAge());  
    }  
});
```

Solution: Sort by Name and Age

```
public class Person {  
    private String firstName;  
    private String lastName; private int age;  
    // TODO: Implement Constructor  
  
    public String getFirstName() { /* TODO */ }  
    public String getLastName() { /* TODO */ }  
    public int getAge() { return age; }  
  
    @Override  
    public String toString() { /* TODO */ }  
}
```

Problem: Salary Increase

- Implement Salary
- Add:
 - getter for salary
 - increaseSalary by percentage
- Persons younger than 30 get
only half of the increase



Solution: Salary Increase (1)

- Expand Person from previous task

```
public class Person {  
    private double salary;  
    // Edit Constructor  
    public double getSalary() {  
        return this.salary;  
    }  
    public void setSalary(double salary) {  
        this.salary = salary;  
    }  
    // Next Slide...  
    // TODO: Edit toString() method  
}
```

Solution: Salary Increase (2)

- Expand Person from previous task

```
public void increaseSalary(double percentage) {  
    if (this.getAge() < 30) {  
        this.setSalary(this.getSalary() +  
                      (this.getSalary() * percentage / 200));  
    } else {  
        this.setSalary(this.getSalary() +  
                      (this.getSalary() * percentage / 100));  
    }  
}
```



Validation

Validation (1)

- Data validation happens in setters

```
private void setSalary(double salary) {  
    if (salary < 460) {  
        throw new IllegalArgumentException("Message");  
    }  
    this.salary = salary;  
}
```

It is better to throw exceptions,
rather than printing to the Console

- Printing with System.out couples your class
- The Client can handle class exceptions

Validation (2)

- Constructors use **private setters** with validation logic

```
public Person(String firstName, String lastName,  
             int age, double salary) {  
    setFirstName(firstName);  
    setLastName(lastName);  
    setAge(age);  
    setSalary(salary);  
}
```

Validation happens
inside the setter

- Guarantees **valid state** of an object in its creation
- Guarantees **valid state** for public setters

Problem: Validation Data

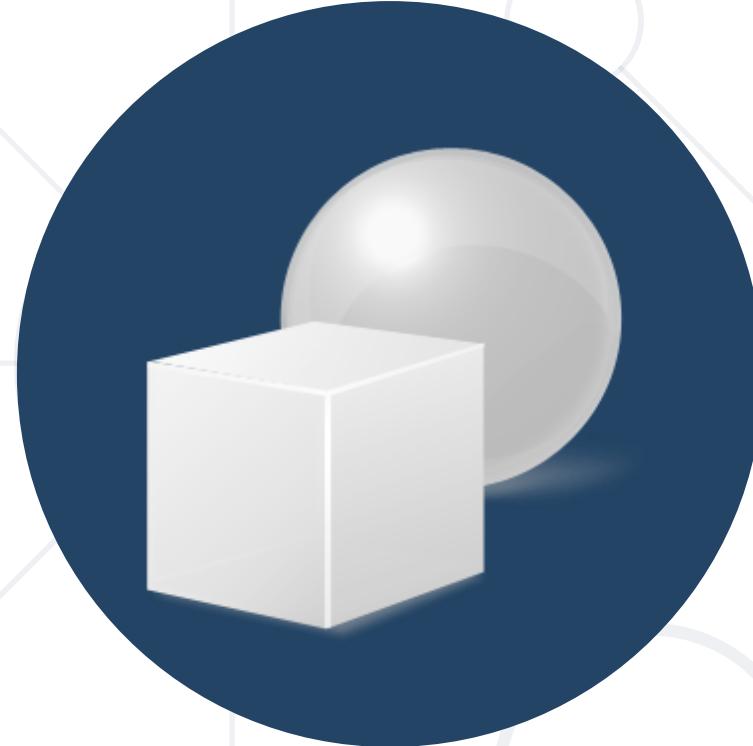
- Expand Person with **validation for every field**
- Names should be **at least 3 symbols**
- Age **cannot be zero or negative**
- Salary **cannot be less than 460**

```
classDiagram
    class Person {
        -firstName : String
        -lastName : String
        -age : int
        -salary : double

        +Person()
        +setFirstName(String fName)
        +setLastName(String lName)
        +setAge(int age)
        +setSalary(double salary)
    }
```

Solution: Validation Data

```
// TODO: Add validation for firstName  
// TODO: Add validation for lastName  
public void setAge(int age) {  
    if (age < 1) {  
        throw new IllegalArgumentException(  
            "Age cannot be zero or negative integer");  
    }  
    this.age = age;  
}  
// TODO: Add validation for salary
```



Mutable and Immutable Objects

Mutable vs Immutable Objects

■ Mutable Objects

- The contents of that instance **can** be altered

```
Point myPoint = new Point(0, 0);  
myPoint.setLocation(1.0, 0.0);  
System.out.println(myPoint);
```



java.awt.Point[1.0, 0.0]

■ Immutable Objects

- The contents of the instance **can't** be altered

```
String str = new String("old String");  
System.out.println(str);  
str.replaceAll("old", "new");  
System.out.println(str);
```



old String
old String

Mutable Fields

- **private** mutable fields are not fully encapsulated



```
class Team {  
    private String name;  
    private List<Person> players;  
  
    public List<Person> getPlayers() {  
        return this.players;  
    }  
}
```



- In this case, the **getter** is like a **setter** too

Mutable Fields – Example

```
Team team = new Team();
Person person = new Person("David", "Adams", 22);
team.getPlayers().add(person);
System.out.println(team.getPlayers().size()); // 1
team.getPlayers().clear();
System.out.println(team.getPlayers().size()); // 0
```

Immutable Fields

- For securing our collection we can return
Collections.unmodifiableList()



```
class Team {  
    private List<Person> players;  
  
    public void addPlayer(Person person) {  
        this.players.add(person);  
    }  
    public List<Person> getPlayers() {  
        return Collections.unmodifiableList(players);  
    }  
}
```

Add new methods for functionality over list

Returns a safe collections

Problem: First and Reserve Team

- Expand your project with class **Team**
- Team have two squads
first team and **reserve team**
- Read persons from console and
add them to team
- If they are **younger** than **40**,
they go to **first squad**
- **Print** both squad **sizes**

```
class Team {  
    -name: String  
    -firstTeam: List<Person>  
    -reserveTeam: List<Person>  
  
    +Team(String name)  
    +getName()  
    -setName(String name)  
    +getFirstTeam()  
    +getReserveTeam()  
    +addPlayer(Person person)
```

Solution: First and Reserve Team

```
private List<Person> firstTeam;
private List<Person> reserveTeam;

public void addPlayer(Person person) {
    if (person.getAge() < 40)
        this.firstTeam.add(person);
    else
        this.reserveTeam.add(person);
}
public List<Person> getFirstTeam() {
    return Collections.unmodifiableList(firstTeam);
}
// TODO: add getter for reserve team
```



final

Keyword Final

Keyword Final

- A **final class** can't be extended

```
public class Animal {}  
public final class Mammal extends Animal {}  
public class Cat extends Mammal {}
```



- A **final method** can't be overridden

```
public final void move(Point point) {}  
public class Mammal extends Animal {  
    @Override  
    public void move() {}  
}
```



Keyword Final

- The **final variable** value can't be changed once it is set

```
private final String name;  
private final List<Person> firstTeam;  
  
public Team (String name) {  
    this.name = name;  
    this.firstTeam = new ArrayList<Person> ();  
}  
  
public void doSomething(Person person) {  
    this.name = "";  
    this.firstTeam = new ArrayList<>();  
    this.firstTeam.add(person);  
}
```

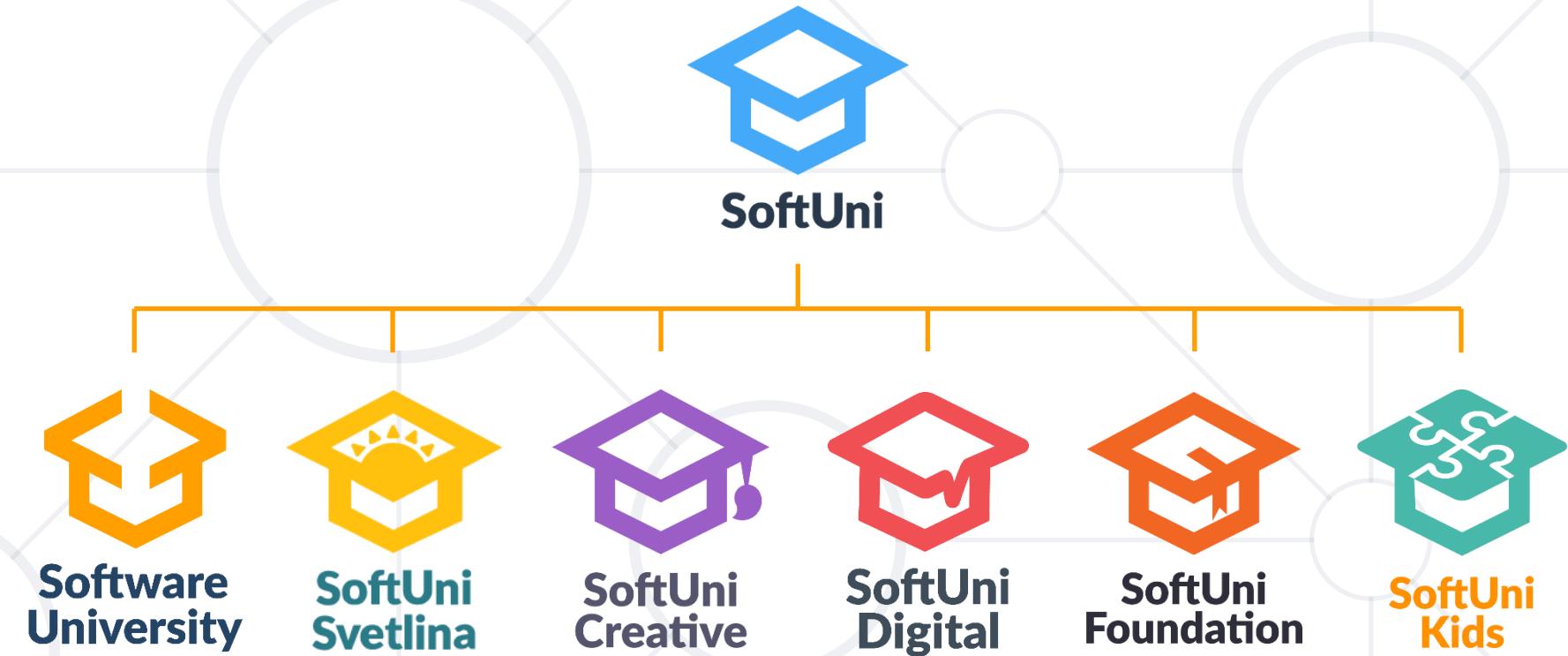
Compile time error

Summary

- Encapsulation:
 - Hides **implementation**
 - Reduces **complexity**
 - Ensures that structural changes remain local
- **Mutable** and **Immutable** objects
- Keyword **final**



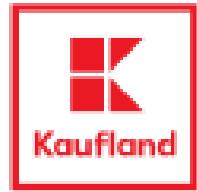
Questions?



SoftUni Diamond Partners



SCHWARZ



MOTION SOFTWARE

Educational Partners



VIRTUAL RACING SCHOOL



Trainings @ Software University (SoftUni)



- Software University – High-Quality Education, Profession and Job for Software Developers
 - softuni.bg
- Software University Foundation
 - softuni.foundation
- Software University @ Facebook
 - facebook.com/SoftwareUniversity
- Software University Forums
 - forum.softuni.bg



Software
University



- 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>



Inheritance

Extending Classes



SoftUni Team

Technical Trainers



SoftUni



Software University

<https://softuni.bg>

Table of Contents

1. Inheritance
2. Class Hierarchies
3. Inheritance in Java
4. Accessing Members of the Base Class
5. Types of Class Reuse
 - Extension, Composition, Delegation
6. When to Use Inheritance

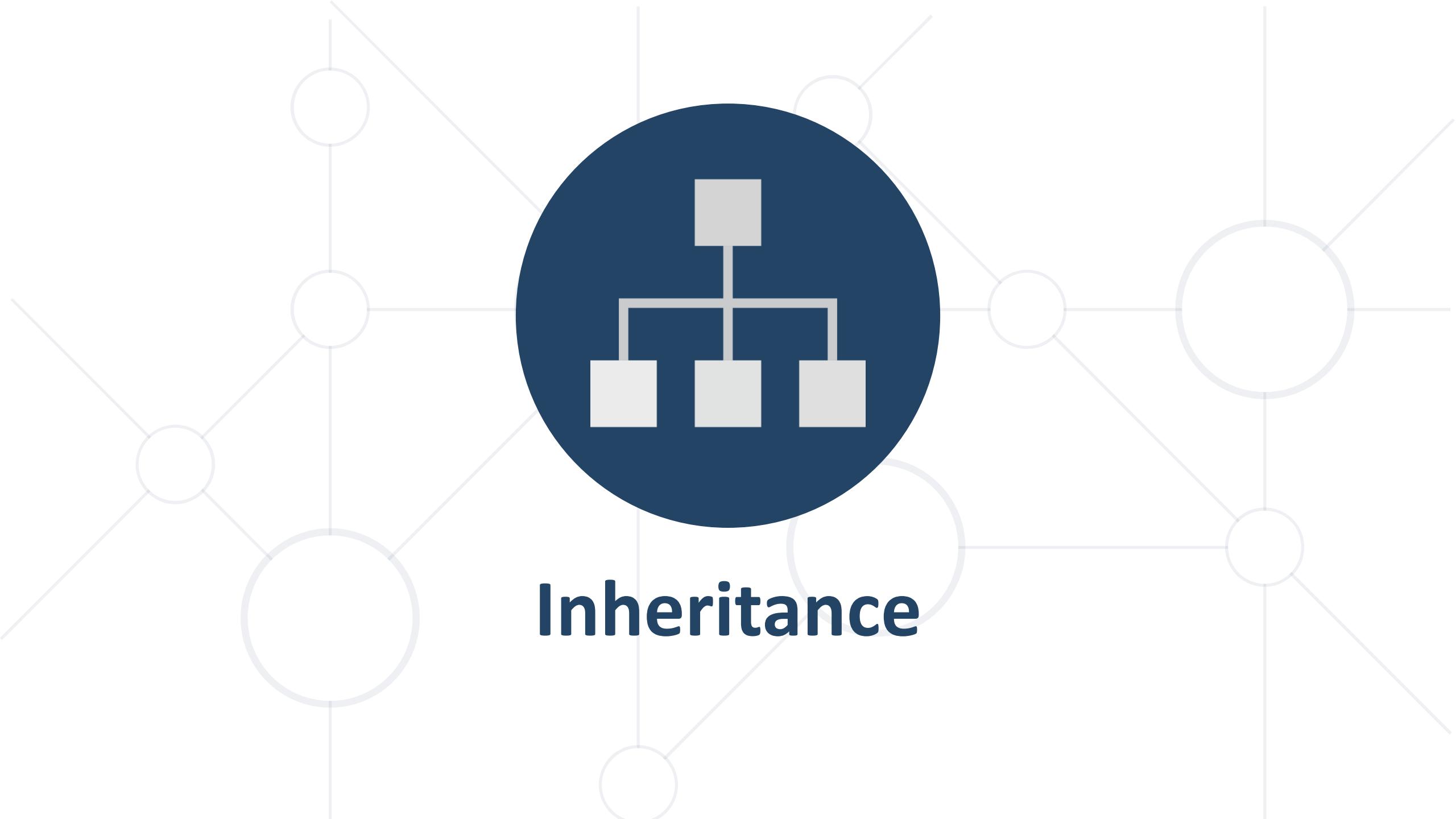


Have a Question?



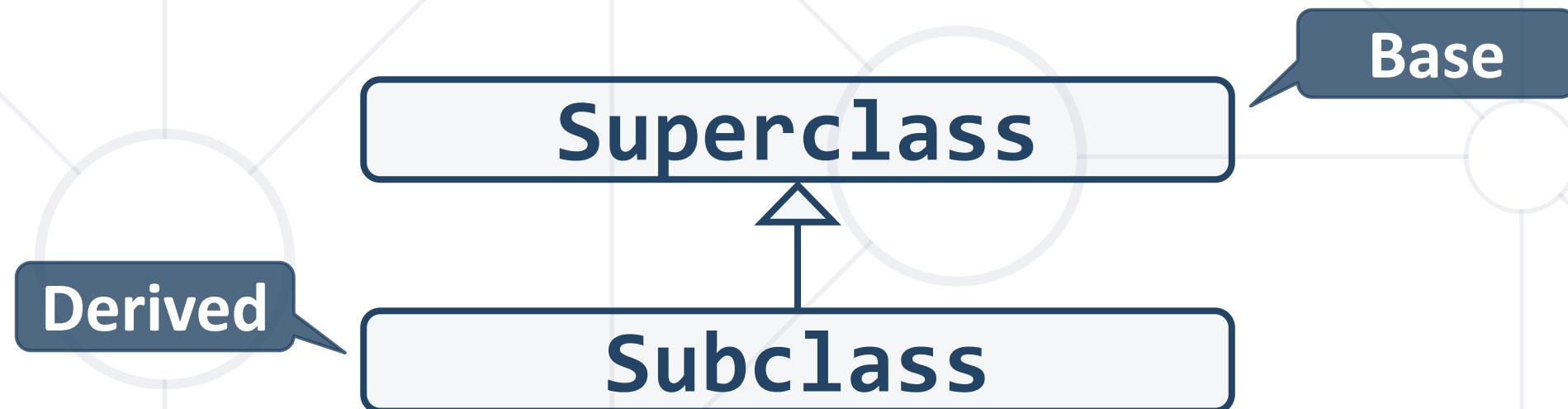
sli.do

#java-advanced

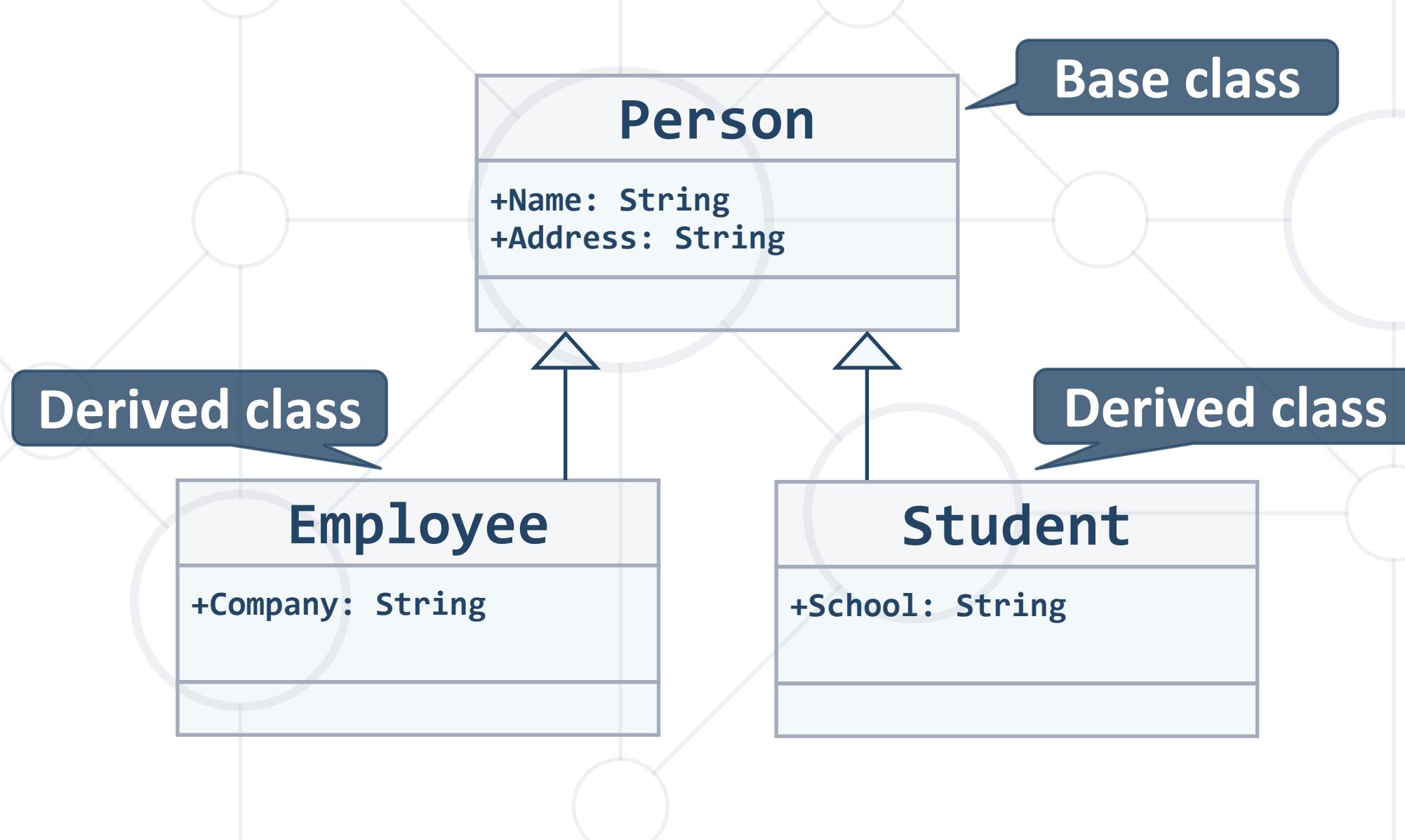


Inheritance

- **Superclass** - Parent class, Base Class
 - The class gives its members to its child class
- **Subclass** - Child class, Derived Class
 - The class taking members from its base class

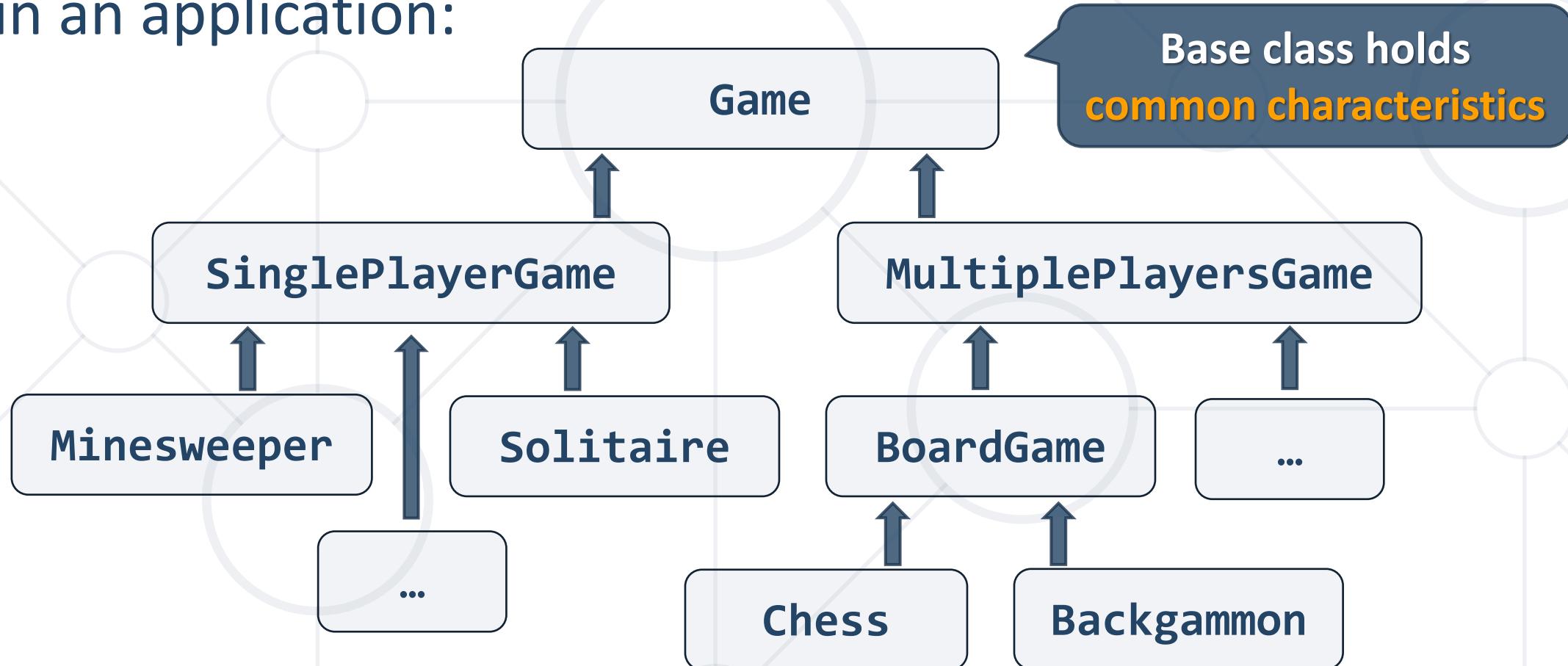


Inheritance – Example

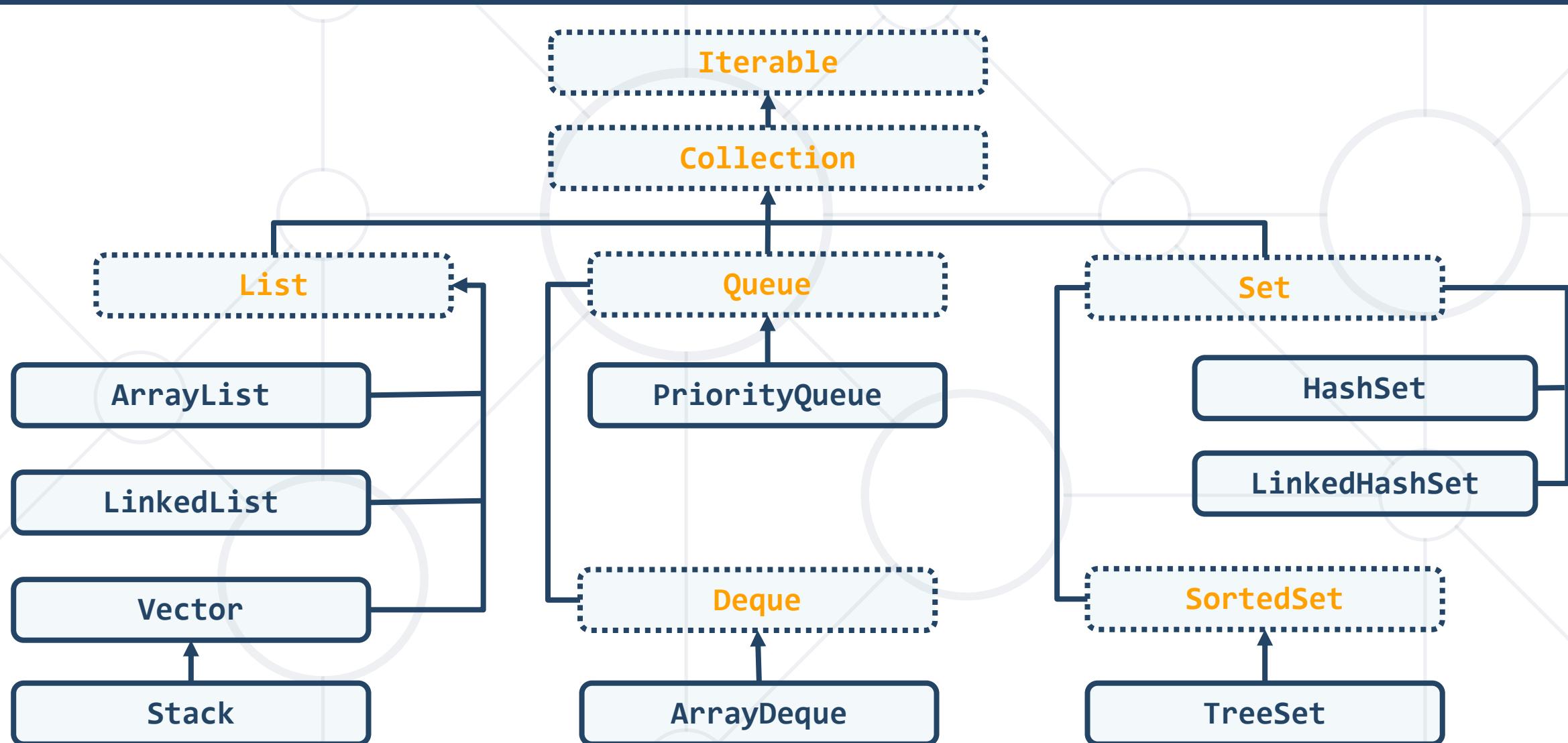


Class Hierarchies

- An **Inheritance** leads to **hierarchies** of classes and/or interfaces in an application:

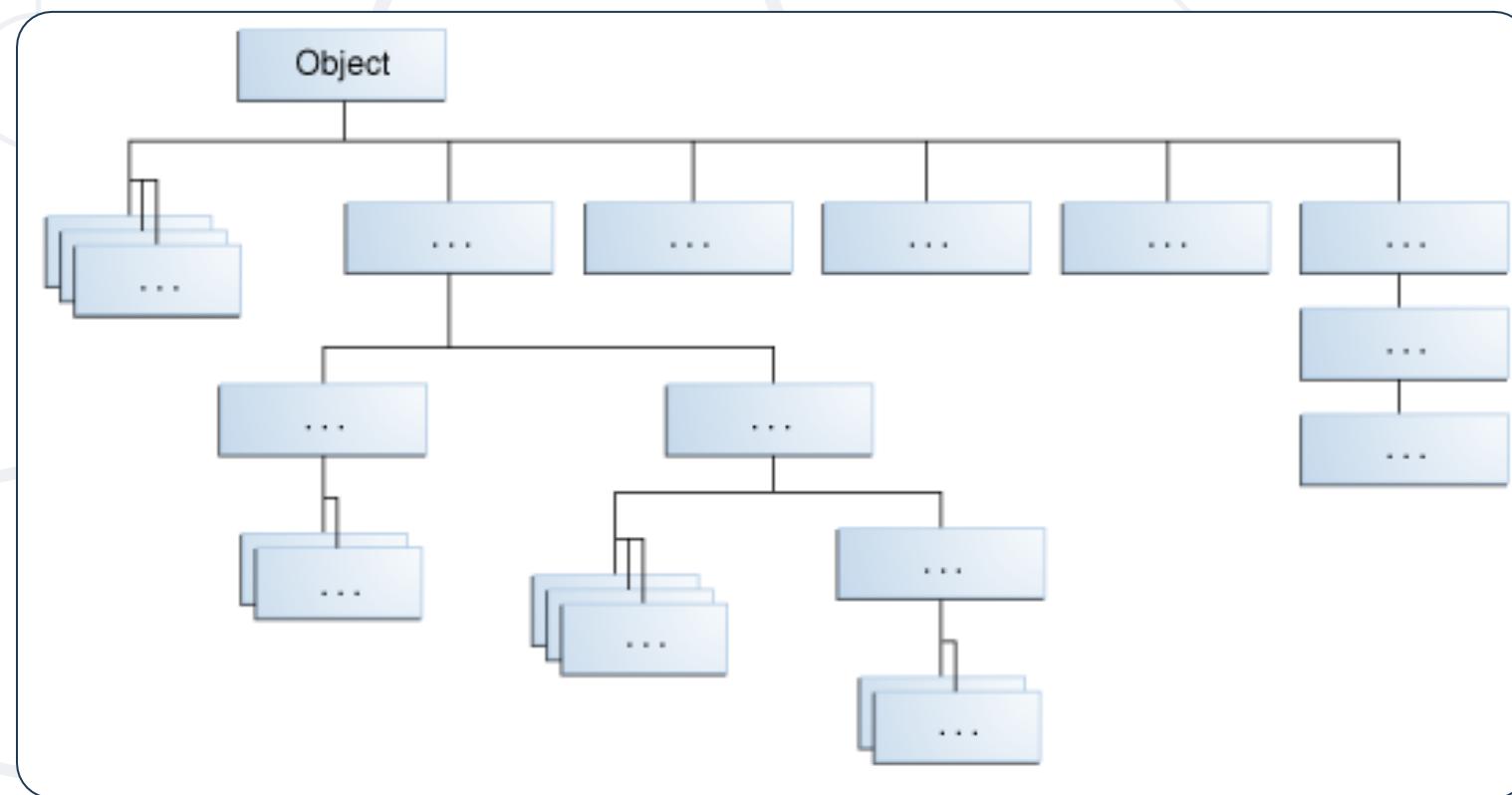


Class Hierarchies – Java Collection



Java Platform Class Hierarchy

- The **Object** is at the root of Java Class Hierarchy



Inheritance in Java

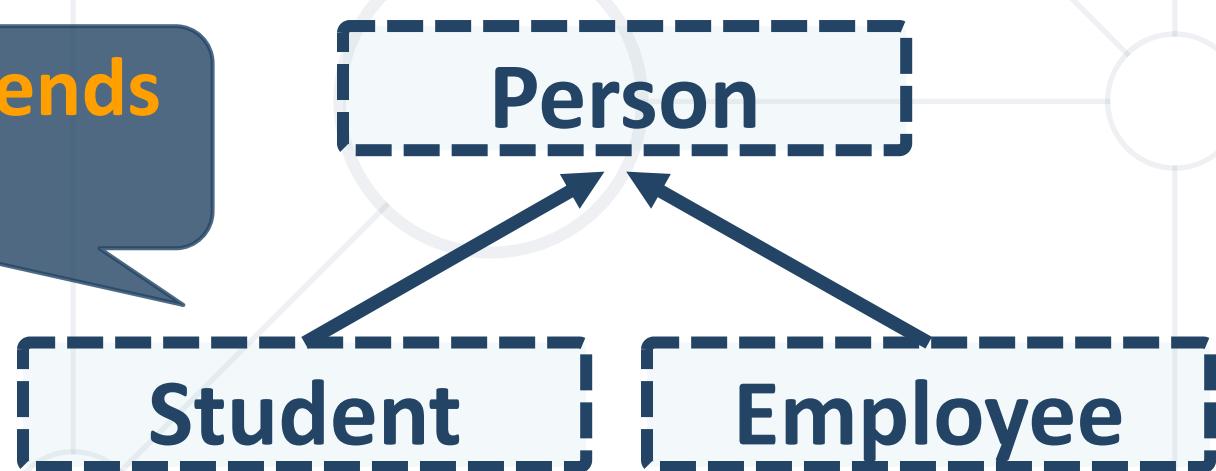
- Java supports inheritance through **extends** keyword

```
class Person { ... }
```

```
class Student extends Person { ... }
```

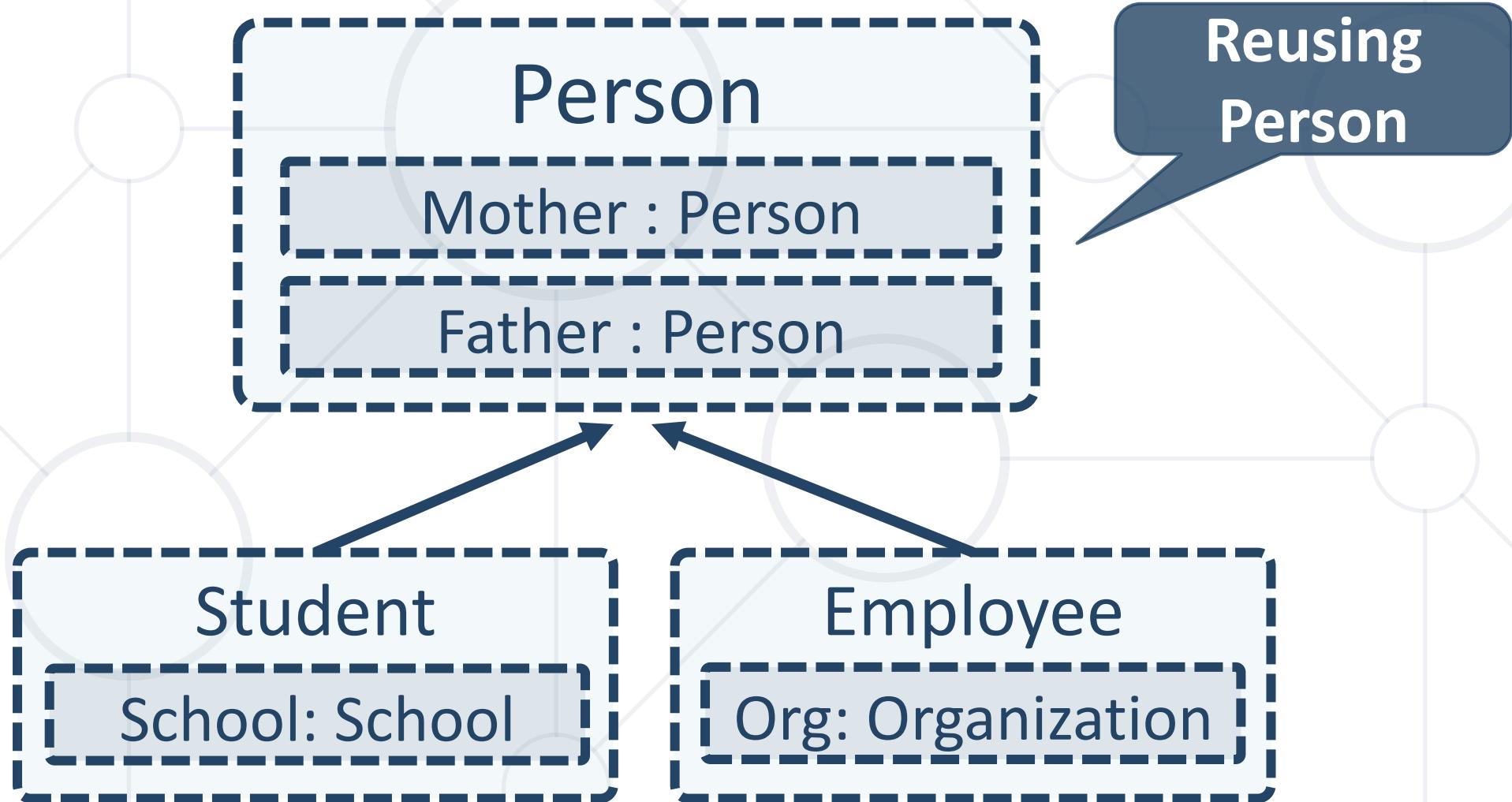
```
class Employee extends Person { ... }
```

Student **extends**
Person



Inheritance – Derived Class

- Class **taking all members** from another class



Using Inherited Members

- You can access inherited members

```
class Person { public void sleep() { ... } }
class Student extends Person { ... }
class Employee extends Person { ... }
```

```
Student student = new Student();
student.sleep();
Employee employee = new Employee();
employee.sleep();
```

Reusing Constructors

- Constructors are **not inherited**
- Constructors **can be reused** by the child classes

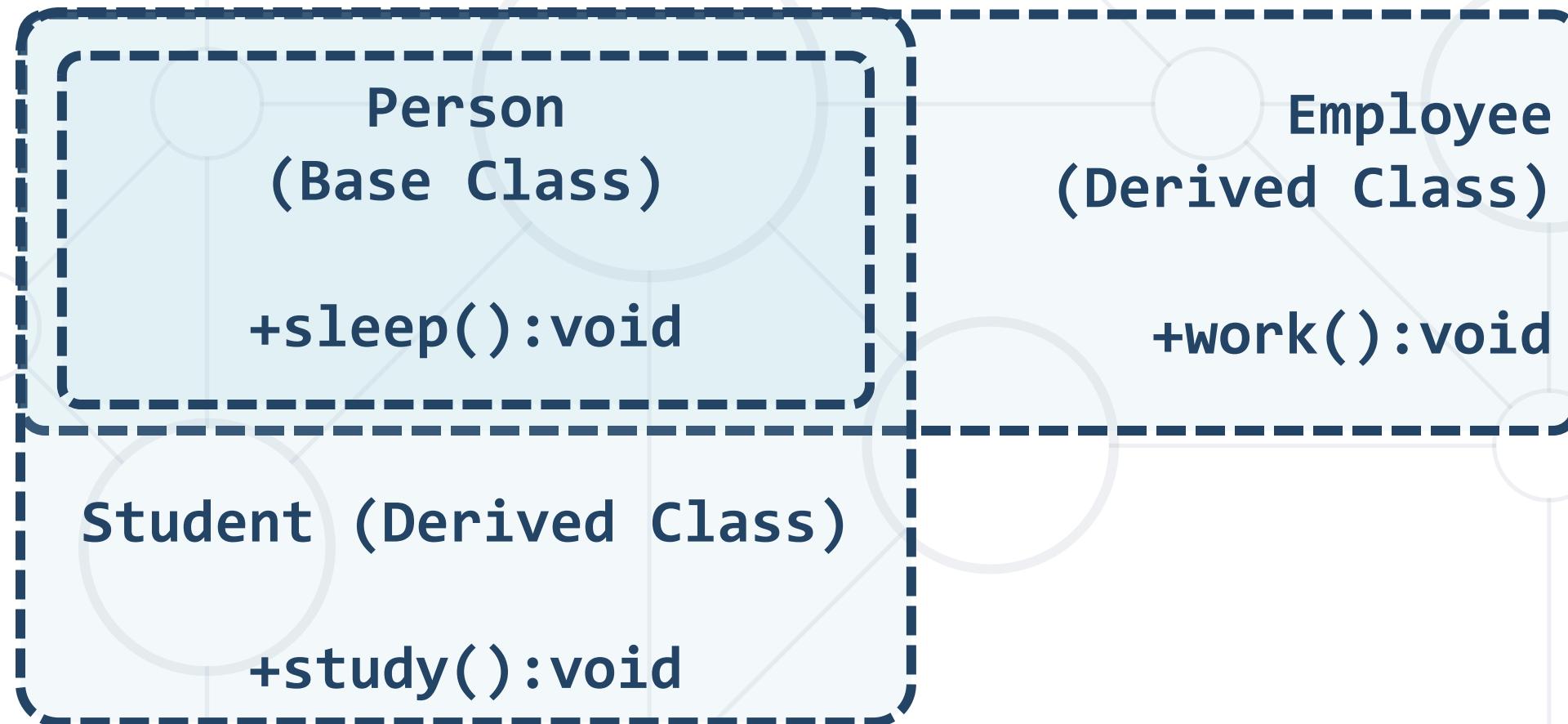
```
class Student extends Person {  
    private School school;  
    public Student(String name, School school) {  
        super(name);  
        this.school = school;  
    }  
}
```



Constructor call
should be first

Thinking about Inheritance – Extends

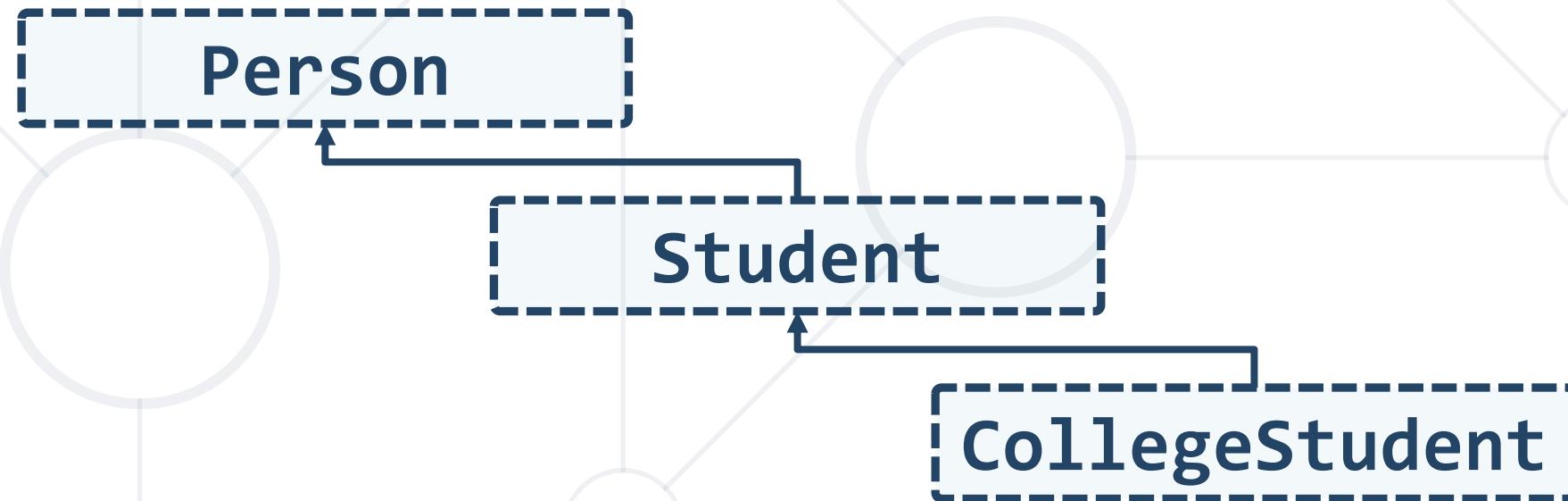
- A derived class instance **contains** an instance of its base class



Inheritance

- Inheritance has a **transitive relation**

```
class Person { ... }  
class Student extends Person { ... }  
class CollegeStudent extends Student { ... }
```



Multiple Inheritance

- In Java, there are no **multiple** inheritances
- Only **multiple interfaces can be implemented**



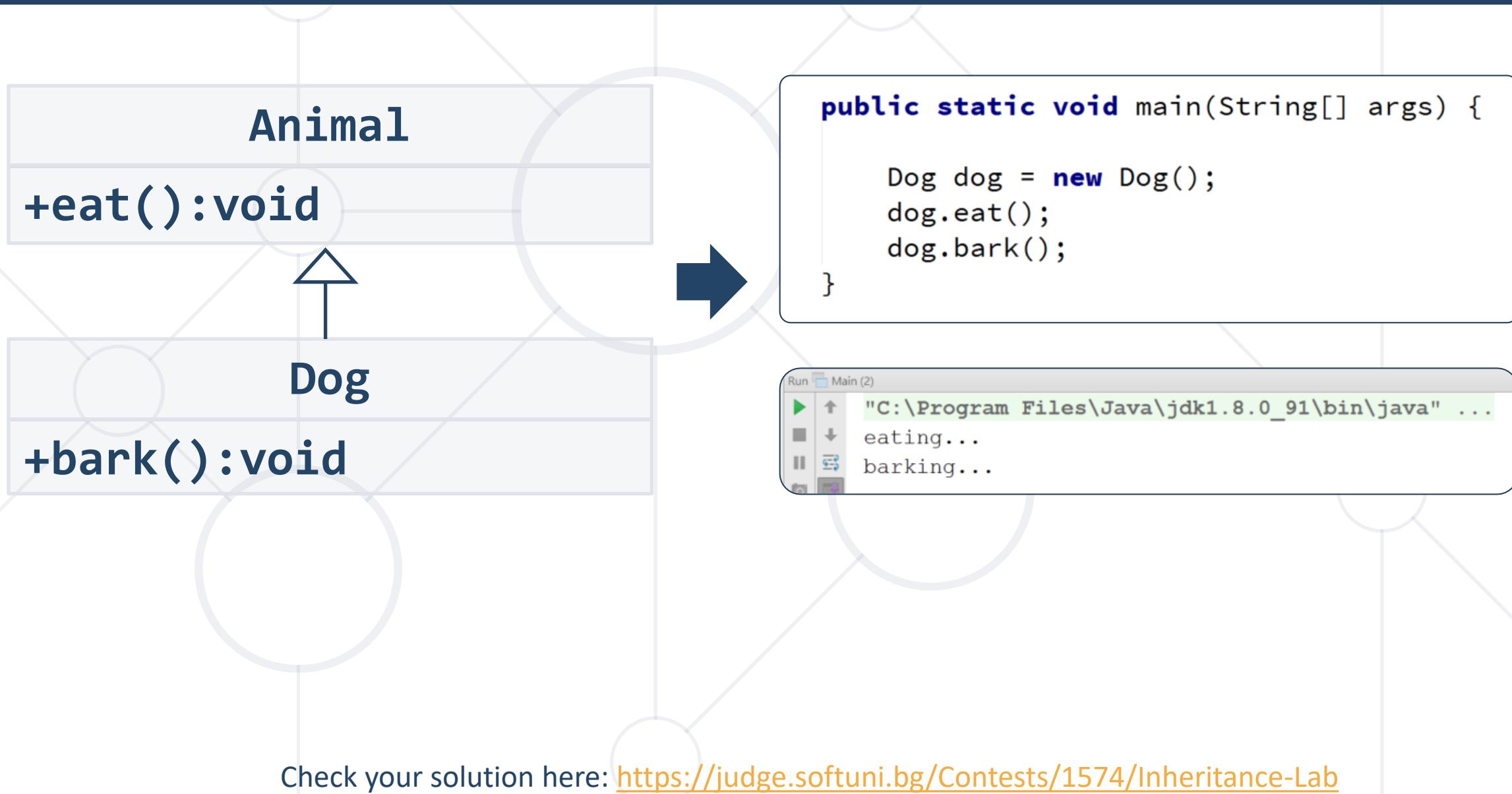
Access to Base Class Members

- Use the **super** keyword

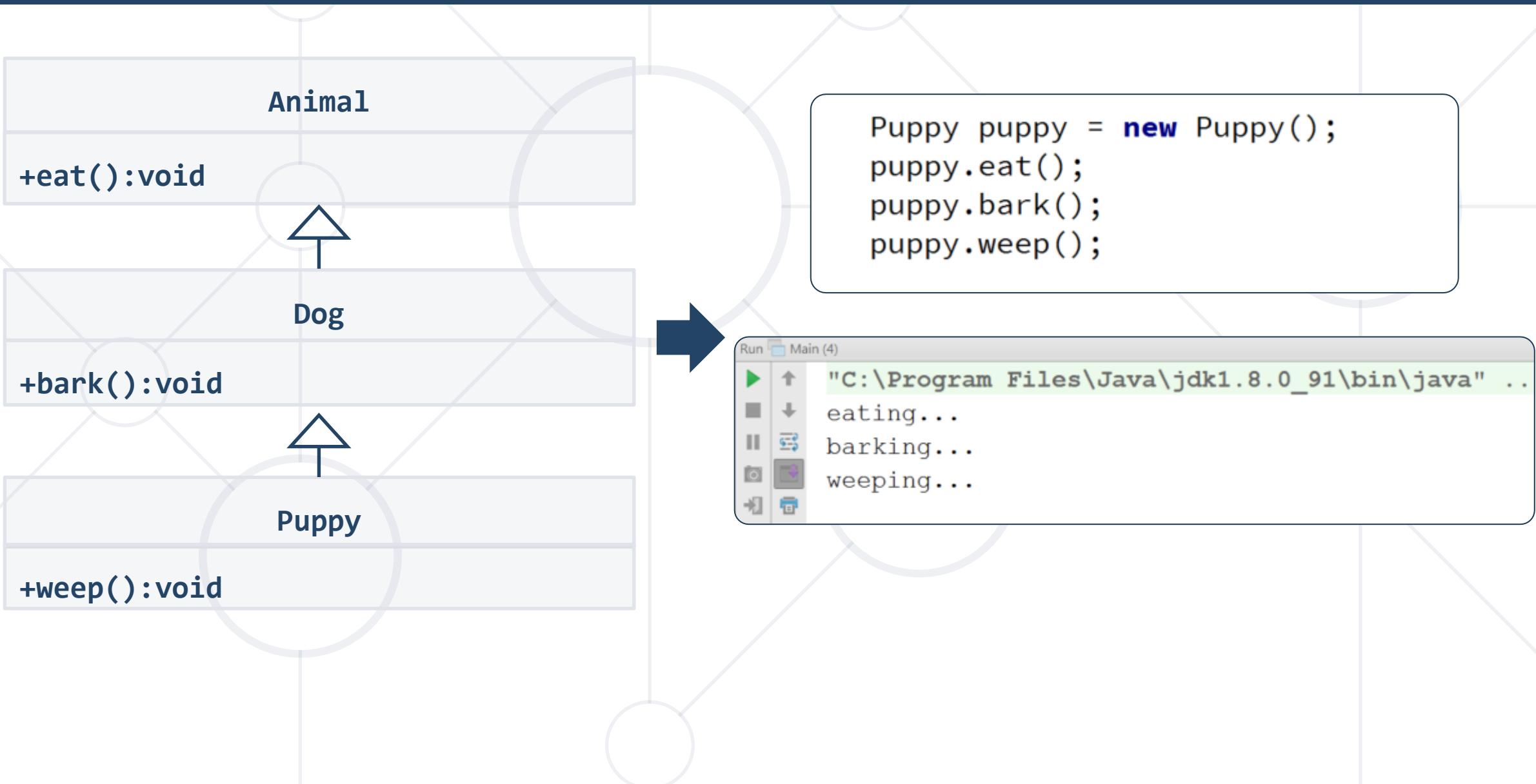
```
class Person { ... }
```

```
class Employee extends Person {
    public void fire(String reasons) {
        System.out.println(
            super.name +
            " got fired because " + reasons);
    }
}
```

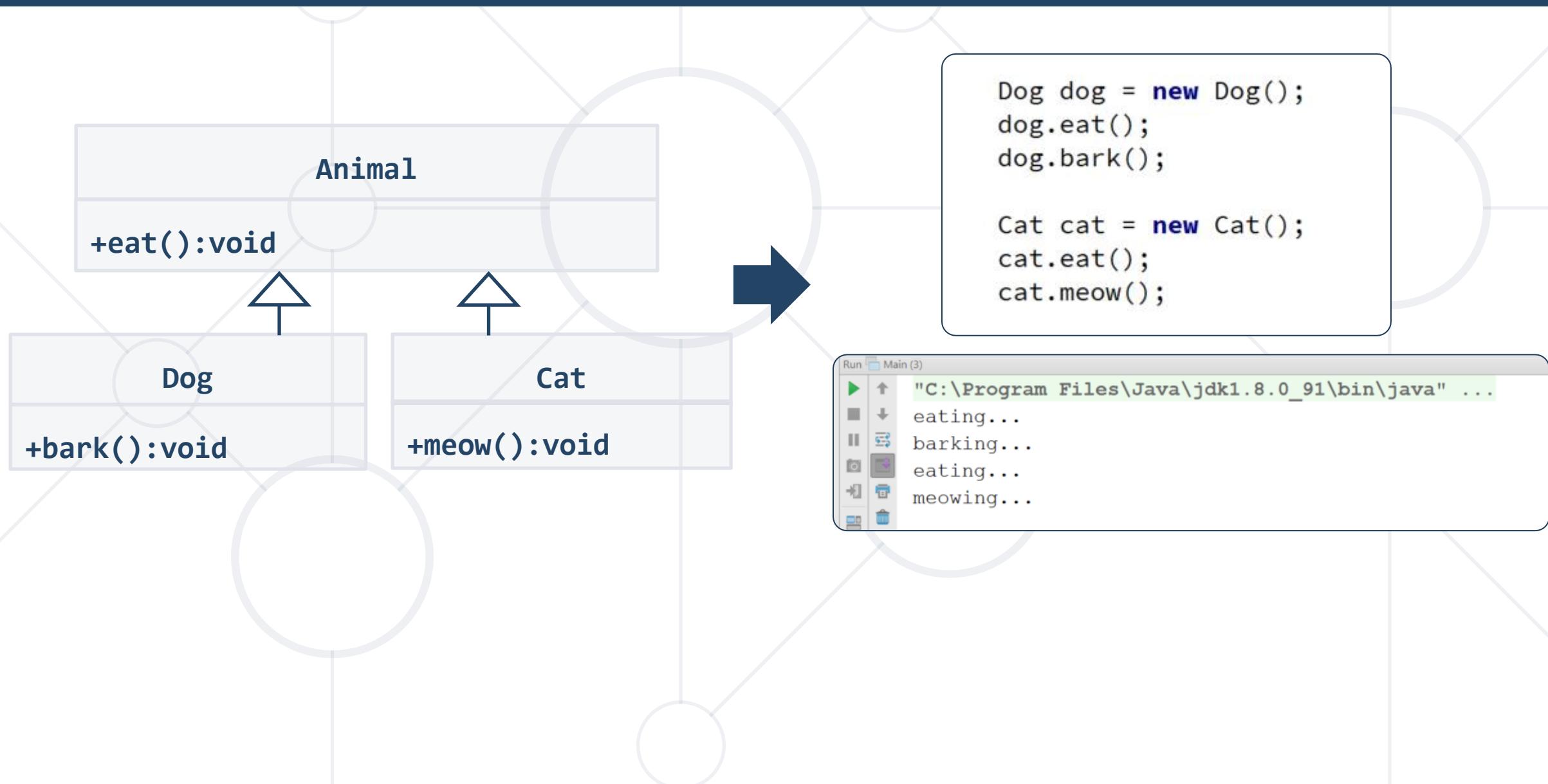
Problem: Single Inheritance



Problem: Multiple Inheritance



Problem: Hierarchical Inheritance



```
Dog dog = new Dog();
dog.eat();
dog.bark();
```

```
Cat cat = new Cat();
cat.eat();
cat.meow();
```

The diagram illustrates a UML class hierarchy. At the top is an abstract base class named **Animal**, indicated by a hollow circle. Below it are two concrete subclasses: **Dog** and **Cat**, represented by solid circles. Arrows point from the subclass names to their respective class boxes. Each class box contains a method definition. The **Dog** class has the method `+bark():void`. The **Cat** class has the method `+meow():void`. The **Animal** class has the method `+eat():void`.

```
graph TD; Animal((Animal)) --> Dog((Dog)); Animal --> Cat((Cat)); Dog --> DogBox[+bark():void]; Cat --> CatBox[+meow():void]; Animal --> AnimalBox[+eat():void]
```

```
Run Main (3)
▶ "C:\Program Files\Java\jdk1.8.0_91\bin\java" ...
█ eating...
█ barking...
█ eating...
█ meowing...
```



Reusing Classes

Inheritance and Access Modifiers

- Derived classes **can access all public and protected members**
- Derived classes can access **default** members **if in same package**
- **Private** fields **aren't inherited** in subclasses (can't be accessed)

```
class Person {  
    protected String address;  
    public void sleep();  
    String name;  
    private String id;  
}
```

Can be accessed
through other methods

Shadowing Variables

- Derived classes **can hide** superclass variables

```
class Person { protected int weight; }
```

```
class Patient extends Person {  
    protected float weight;  
    public void method() {  
        double weight = 0.5d;  
    }  
}
```

hides **int weight**

hides both

Shadowing Variables – Access

- Use **super** and **this** to specify member access

```
class Person { protected int weight; }
```

```
class Patient extends Person {  
    protected float weight;  
    public void method() {  
        double weight = 0.5d;  
        this.weight = 0.6f;  
        super.weight = 1;  
    }  
}
```

Local variable

Instance member

Base class member

Overriding Derived Methods

- A **child class** can redefine existing methods

```
public class Person {  
    public void sleep() {  
        System.out.println("Person sleeping"); }  
}
```

Method in base class **must not be final**

```
public class Student extends Person {  
    @Override  
    public void sleep(){  
        System.out.println("Student sleeping"); }  
}
```

Signature and return type **should match**

Final Methods

- **final** – defines a method that **can't be overridden**

```
public class Animal {  
    public final void eat() { ... }  
}
```

```
public class Dog extends Animal {  
  
    @Override  
    public void eat() {} // Error...  
}
```

Final Classes

- Inheriting from final classes is forbidden

```
public final class Animal {  
    ...  
}
```

```
public class Dog extends Animal {}           // Error...  
public class MyString extends String {}      // Error...  
public class MyMath extends Math {}          // Error...
```

Inheritance Benefits – Abstraction

- One approach for providing an abstraction

Focus on common properties

```
Person person = new Person();  
Student student = new Student();
```

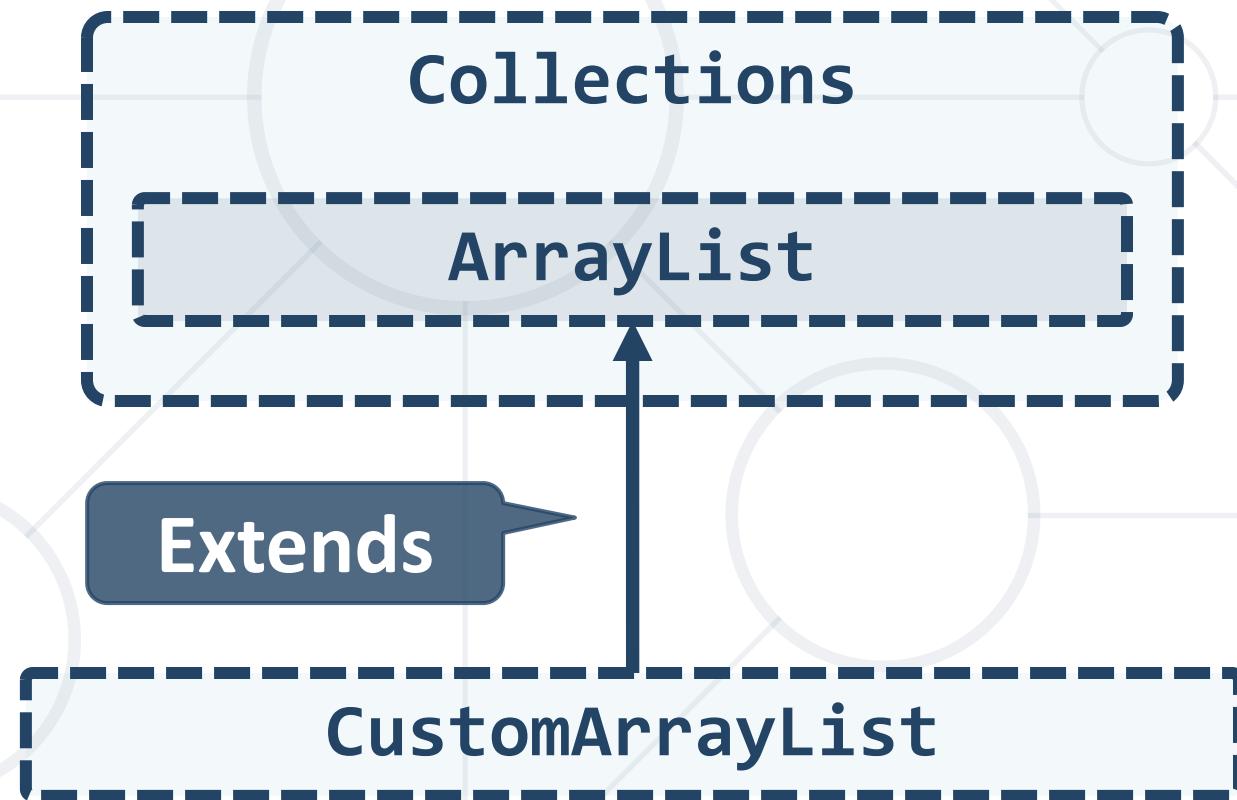
```
List<Person> people = new ArrayList();
```

```
people.add(person);  
people.add(student);
```



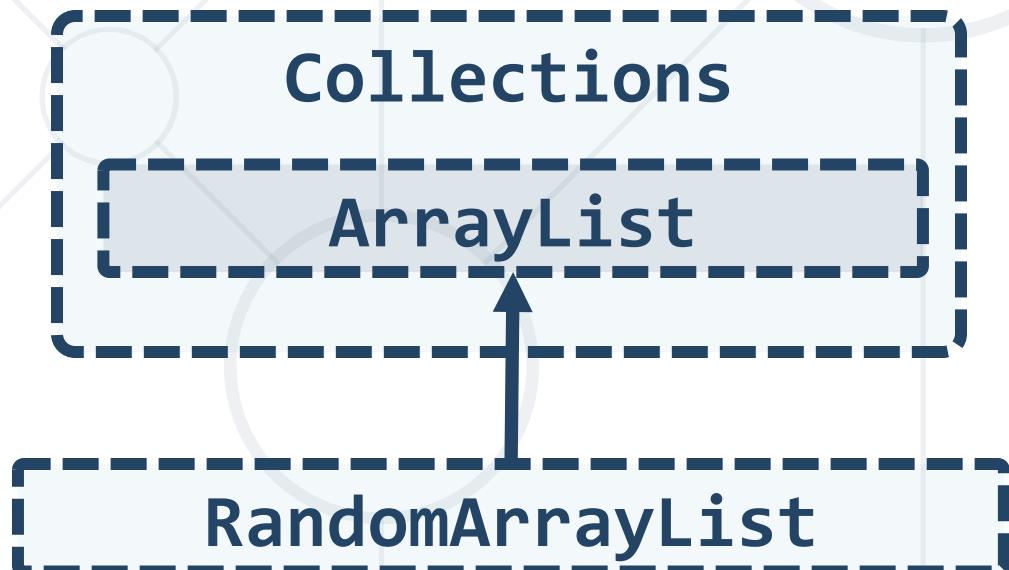
Inheritance Benefits – Extension

- We can **extend a class** that we **can't otherwise change**



Problem: Random Array List

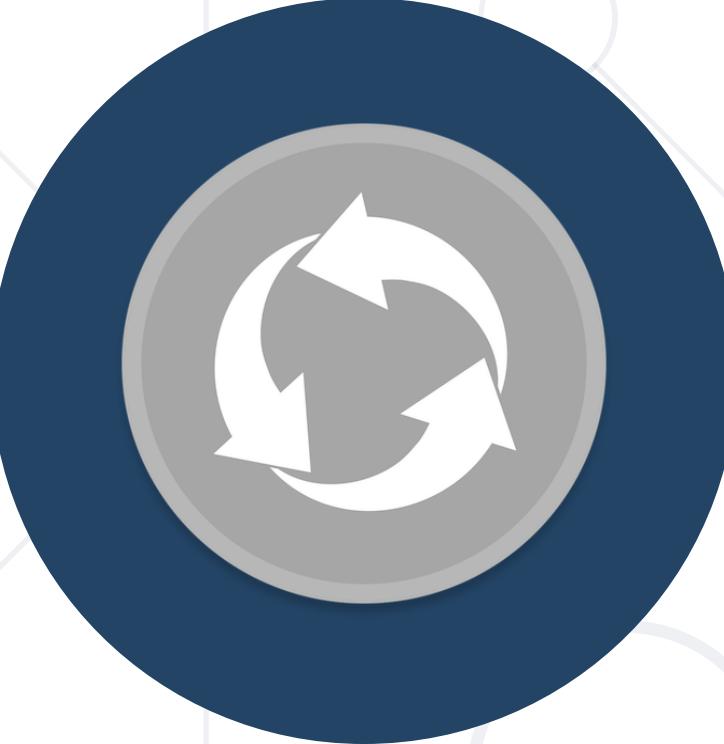
- Create an array list that has
 - All functionality of an ArrayList
 - Function that returns and removes a random element



+getRandomElement():Object

Solution: Random Array List

```
public class RandomArrayList extends ArrayList {  
    private Random rnd; // Initialize this...  
  
    public Object getRandomElement() {  
        int index = this.rnd.nextInt(super.size());  
        Object element = super.get(index);  
        super.remove(index);  
        return element;  
    }  
}
```



Types of Class Reuse

- **Duplicate code** is error prone
- **Reuse classes** through the **extension**
- Sometimes the only way



Composition

- Using classes to **define classes**

```
class Laptop {  
    Monitor monitor;  
    Touchpad touchpad;  
    Keyboard keyboard;  
    ...  
}
```

Reusing classes

Laptop

Monitor

Touchpad

Keyboard

Delegation

```
class Laptop {  
    Monitor monitor;  
    void incrBrightness() {  
        monitor.brighten();  
    }  
  
    void decrBrightness() {  
        monitor.dim();  
    }  
}
```

Laptop

Monitor

increaseBrightness()
decreaseBrightness()

Problem: Stack of Strings

- Create a simple Stack class which can store only strings

StackOfStrings

```
-data: List<String>  
+push(String) :void  
+pop(): String  
+peek(): String  
+isEmpty(): boolean
```

StackOfStrings

ArrayList

```
StackOfStrings sos = new StackOfStrings();  
sos.push("one");  
System.out.println(sos.pop());  
System.out.println(sos.isEmpty());  
System.out.println(sos.peek());
```

Solution: Stack of Strings

```
public class StackOfStrings {  
    private List<String> container;  
    // TODO: Create a constructor  
    public void push(String item) { this.container.add(item); }  
    public String pop() {  
        // TODO: Validate if list is not empty  
        return this.container.remove(this.container.size() - 1);  
    }  
}
```

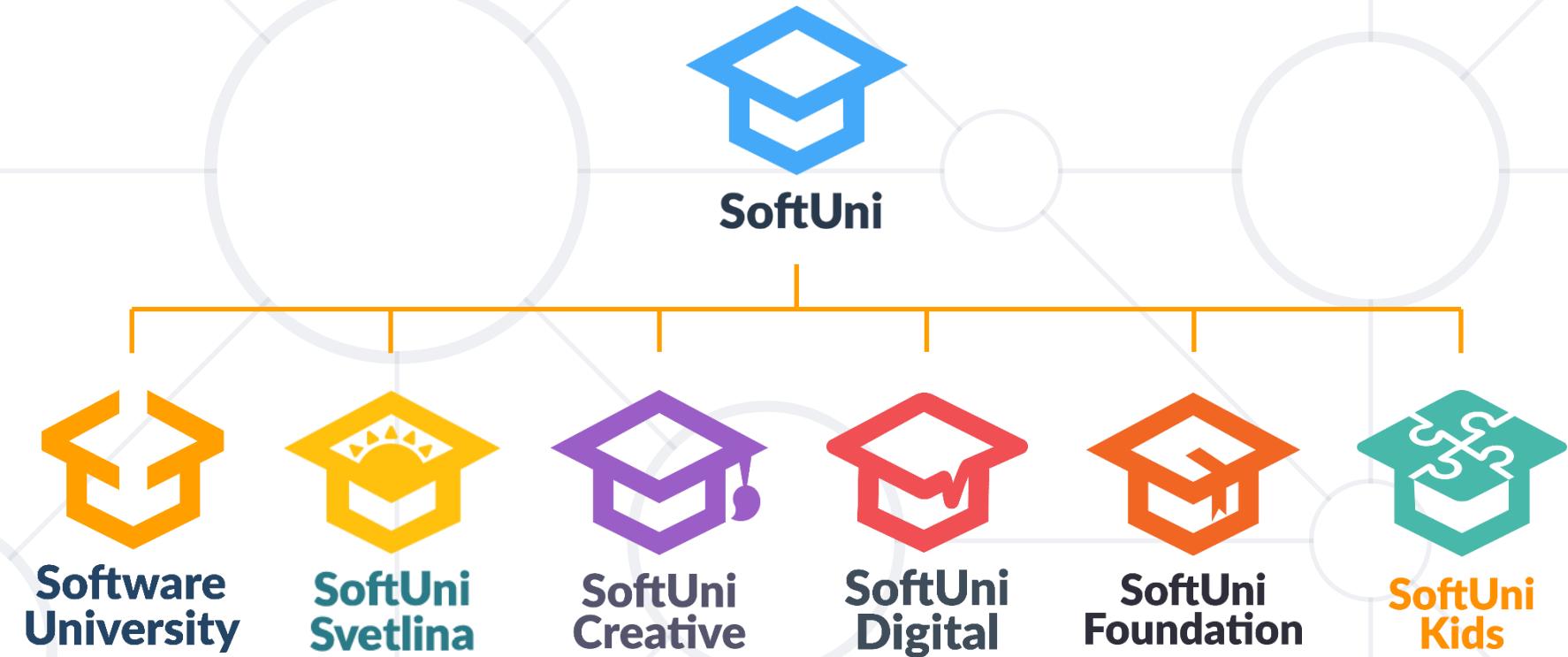
When to Use Inheritance

- Classes share **IS-A** relationship
 - Derived class **IS-A-SUBSTITUTE** for the base class
 - Share the **same role**
 - The derived class is the **same as the base class** but adds a **little bit more functionality**
- Too simplistic

- Inheritance is a powerful tool for **code reuse**
- **Subclass inherits** members from **Superclass**
- Subclass can **override** methods
- Look for classes with the **same role**
- Look for **IS-A** and **IS-A-SUBSTITUTE** for relationship
- Consider **Composition** and **Delegation** instead



Questions?



SoftUni Diamond Partners



SCHWARZ



MOTION SOFTWARE

Educational Partners



VIRTUAL RACING SCHOOL



Trainings @ Software University (SoftUni)



- Software University – High-Quality Education, Profession and Job for Software Developers
 - softuni.bg
- Software University Foundation
 - softuni.foundation
- Software University @ Facebook
 - facebook.com/SoftwareUniversity
- Software University Forums
 - forum.softuni.bg



Software
University

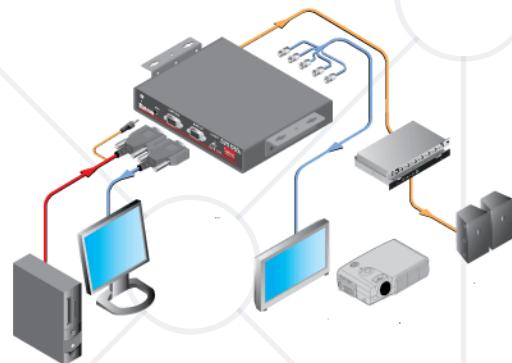


- This course (slides, examples, demos, exercises, homework, documents, videos and other assets) is **copyrighted content**
- Unauthorized copy, reproduction or use is illegal
- © SoftUni – <https://about.softuni.bg>
- © Software University – <https://softuni.bg>



Interfaces and Abstraction

Interfaces vs Abstract Classes
Abstraction vs Encapsulation



SoftUni Team

Technical Trainers

 Software University



SoftUni



Software University
<https://softuni.bg>

Table of Contents

1. Abstraction

- Abstraction vs Encapsulation

2. Interfaces

- Default Methods
- Static Methods

3. Abstract Classes

4. Interfaces vs Abstract Classes

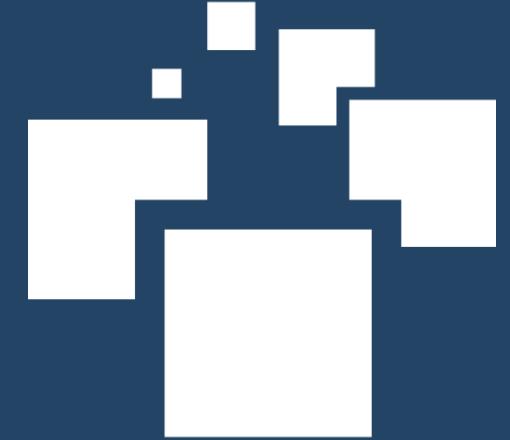


Have a Question?



sli.do

#java-advanced



Abstraction

What is Abstraction?

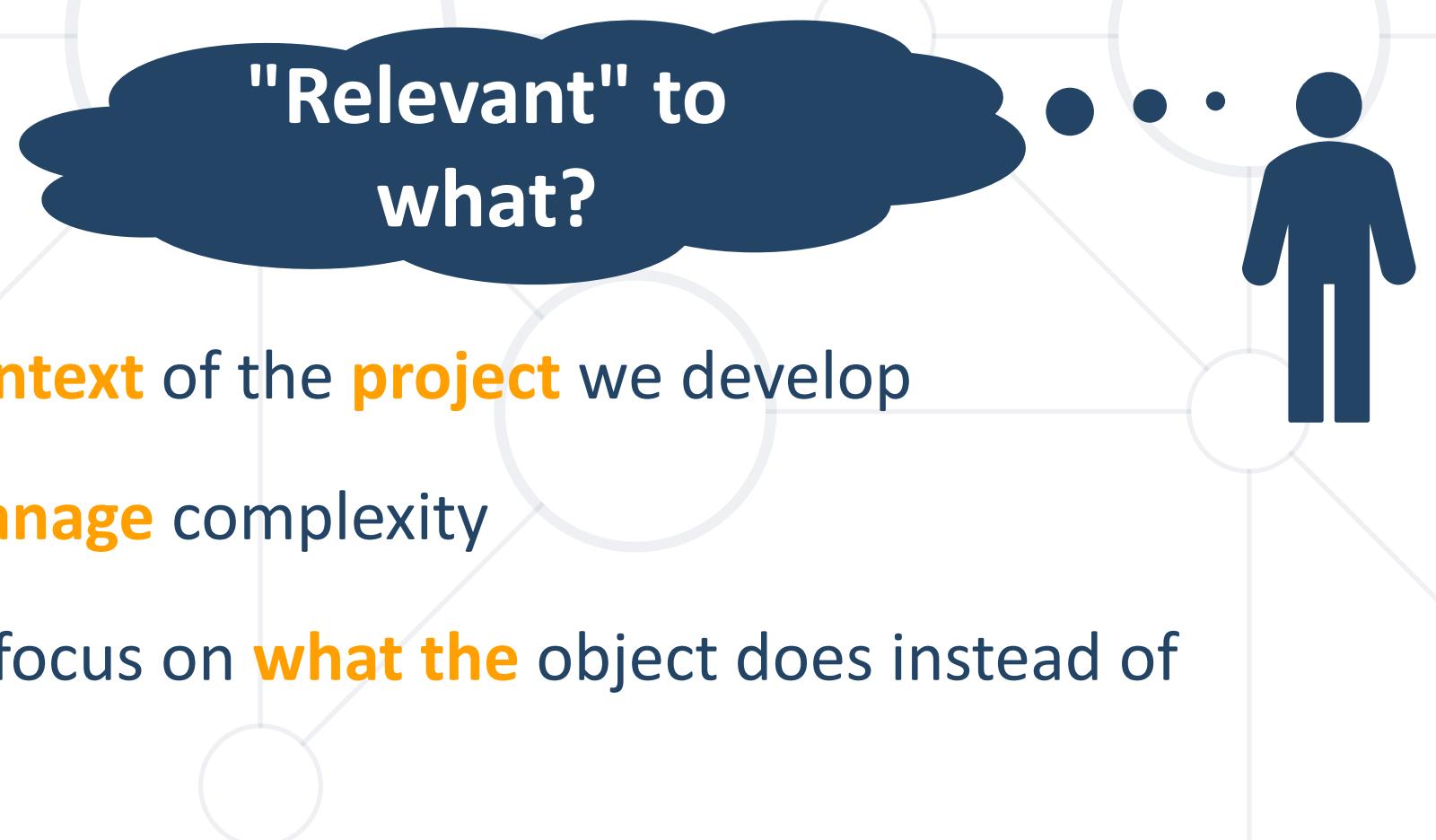
- Latin origin



- Preserving information that is **relevant** in a context
- Forgetting information that is **irrelevant** in that context



Abstraction in OOP

- Abstraction means ignoring irrelevant features, properties, or functions and emphasizing the relevant ones ...


"Relevant" to what?
- ... relevant to the context of the project we develop
- Abstraction helps manage complexity
- Abstraction lets you focus on what the object does instead of how it does it

Achieving Abstraction

- There are 2 ways to achieve abstraction in Java
 - Interfaces (**100% abstraction**)
 - Abstract class (**0% - 100% abstraction**)

```
public interface Animal {}
```

```
public abstract class Mammal {}
```

```
public class Person extends Mammal implements Animal {}
```

Abstraction vs. Encapsulation

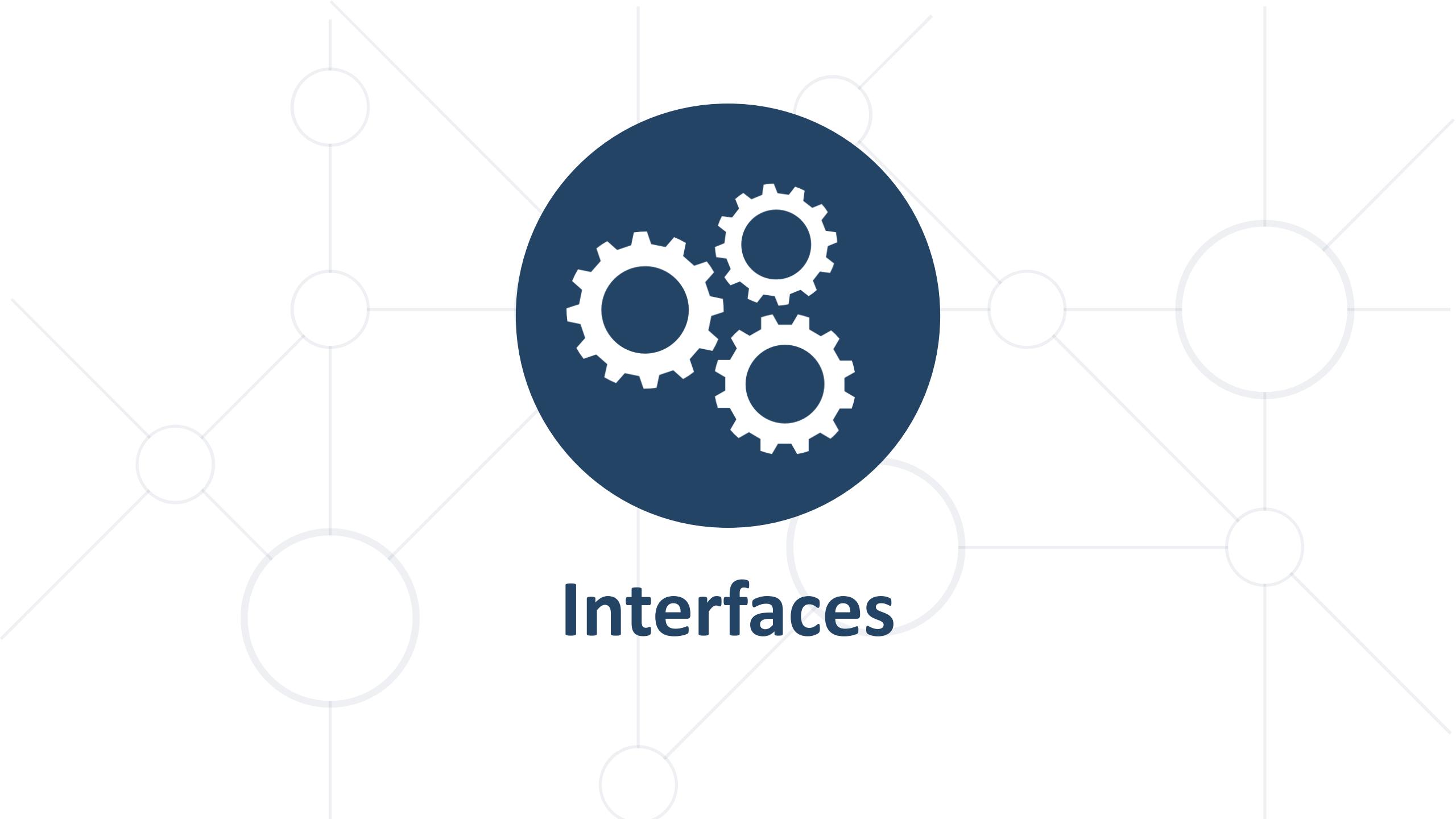
■ Abstraction

- Process of **hiding** the **implementation details** and showing only functionality to the user
- Achieved with **interfaces** and **abstract classes**

■ Encapsulation

- Used to **hide** the **code** and **data** inside a **single unit** to **protect** the data from the outside world
- Achieved with **access modifiers** (private, protected, public)





Interfaces

Interface

- Internal addition by a compiler

Public or default modifier

```
public interface Printable {  
    int MIN = 5;  
    void print();  
}
```

Keyword

Name

compiler

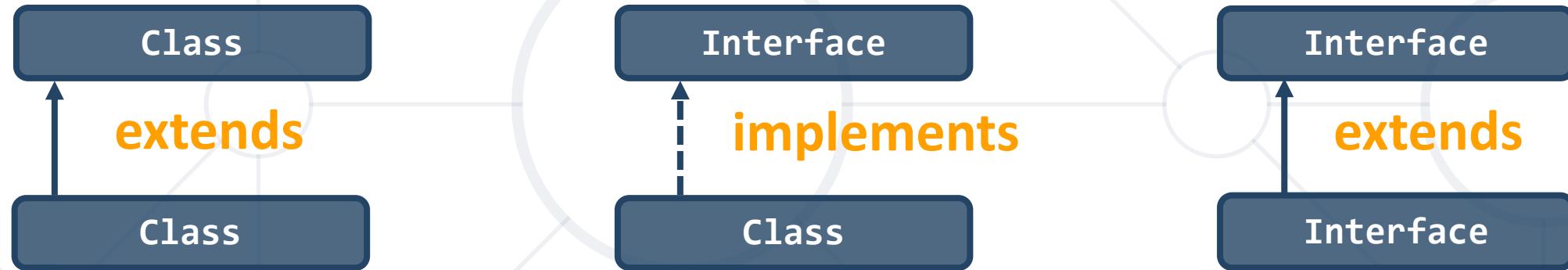
"public abstract" before methods

```
interface Printable {  
    public static final int MIN = 5;  
    public abstract void print();  
}
```

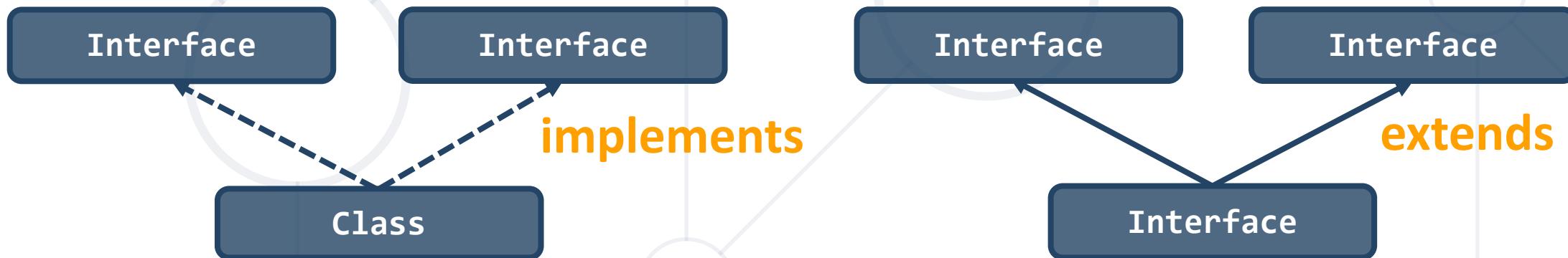
"public static final" before fields

Implements vs Extends

- Relationship between classes and interfaces



- Multiple inheritances



Interface Example

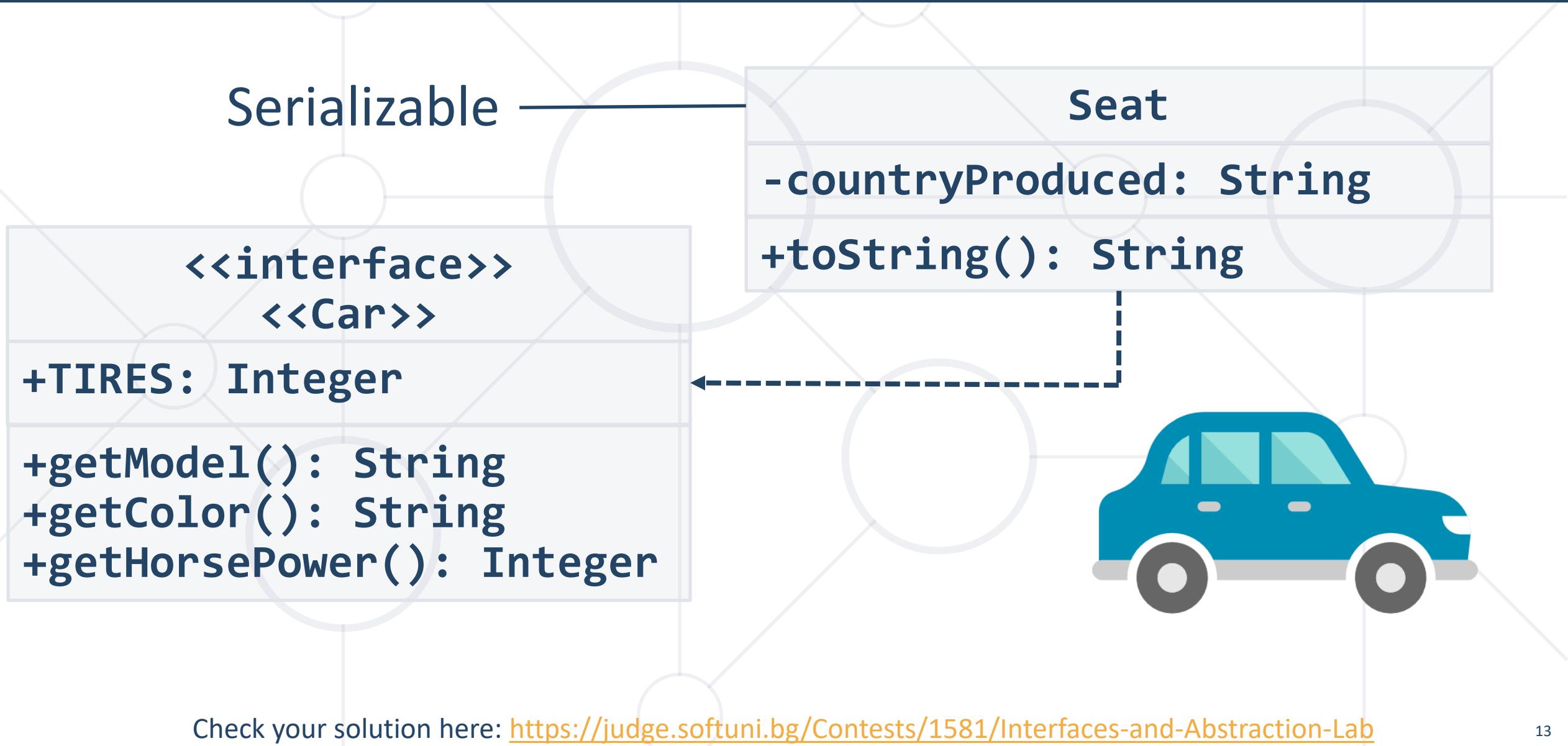
- Implementation of **print()** is provided in class **Document**

```
public interface Printable {  
    void print();  
}
```

```
class Document implements Printable {  
    public void print() { System.out.println("Hello"); }  
    public static void main(String args[]) {  
        Printable doc = new Document();  
        doc.print(); // Hello  
    }  
}
```

Polymorphism

Problem: Car Shop



Solution: Car Shop (1)

```
public interface Car {  
    int TIRES = 4;  
    String getModel();  
    String getColor();  
    Integer getHorsePower();  
    String countryProduced();  
}
```

Solution: Car Shop (2)

```
public class Seat implements Car, Serializable {  
    // TODO: Add fields, constructor and private methods  
    @Override  
    public String getModel() { return this.model; }  
    @Override  
    public String getColor() { return this.color; }  
    @Override  
    public Integer getHorsePower() { return this.horsePower; }  
}
```

Extend Interface

- The interface can **extend another interface**

```
public interface Showable {  
    void show();  
}
```



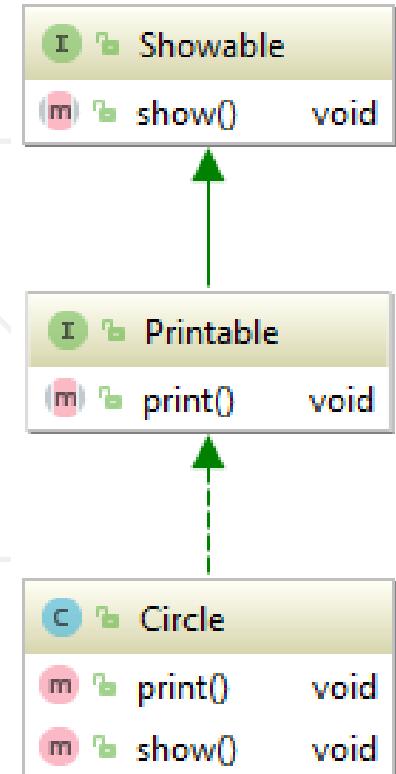
```
public interface Printable extends Showable {  
    void print();  
}
```

Extend Interface

- The class which implements **child** interface **must** provide an implementation for **parent** interface too

```
class Circle implements Printable
public void print() {
    System.out.println("Hello");
}

public void show() {
    System.out.println("Welcome");
}
```



Problem: Car Shop Extended

- Refactor your first problem code
 - Add for rentable cars
 - Add class **Cinterface** for sellable cars
 - Add **interface arlImpl**
 - Add class Audi, which **extends CarImpl** and **implements** rentable
 - Refactor class Seat to **extends CarImpl** and **implements** rentable

Solution: Car Shop Extended (1)

```
public interface Sellable extends Car {  
    Double getPrice();  
}
```

```
public interface Rentable extends Car {  
    Integer getMinRentDay();  
    Double getPricePerDay();  
}
```

Solution: Car Shop Extended (2)

```
public class Audi extends CarImpl implements Rentable {  
    public Integer getMinRentDay() {  
        return this.minDaysForRent; }  
    public Double getPricePerDay() {  
        return this.pricePerDay; }  
    // TODO: Add fields, toString() and Constructor  
}
```

Default Method

- Since Java 8 we can have a **method body** in the **interface**

```
public interface Drawable {  
    void draw();  
    default void msg() {  
        System.out.println("default method:");  
    }  
}
```

- If you need to **override** the default method think about your **design**

Default Method

- Implementation is **not needed** for **default methods**

```
class TestInterfaceDefault {  
    public static void main(String args[]) {  
        Drawable d = new Rectangle();  
        d.draw(); // drawing rectangle  
        d.msg(); // default method  
    }  
}
```

Static Method

- Since Java 11, we can have a **static method** in the **interface**

```
public interface Drawable {  
    void draw();  
    static int cube(int x) { return x*x*x; }  
}
```

```
public static void main(String args[]) {  
    Drawable d = new Rectangle();  
    d.draw();  
    System.out.println(Drawable.cube(3)); } // 27
```

Problem: Say Hello

- Design a project, which has
 - Interface for Person
 - 3 implementations for different nationalities
 - Override where needed

`<<Person>>`
Bulgarian

`-name: String`
`+sayHello(): String`

`<<Person>>`
European

`-name: String`

`<<Person>>`
Chinese

`-name: String`
`+sayHello(): String`

`<<interface>>`
`<<Person>>`
`+getName(): String`
`+sayHello():String`

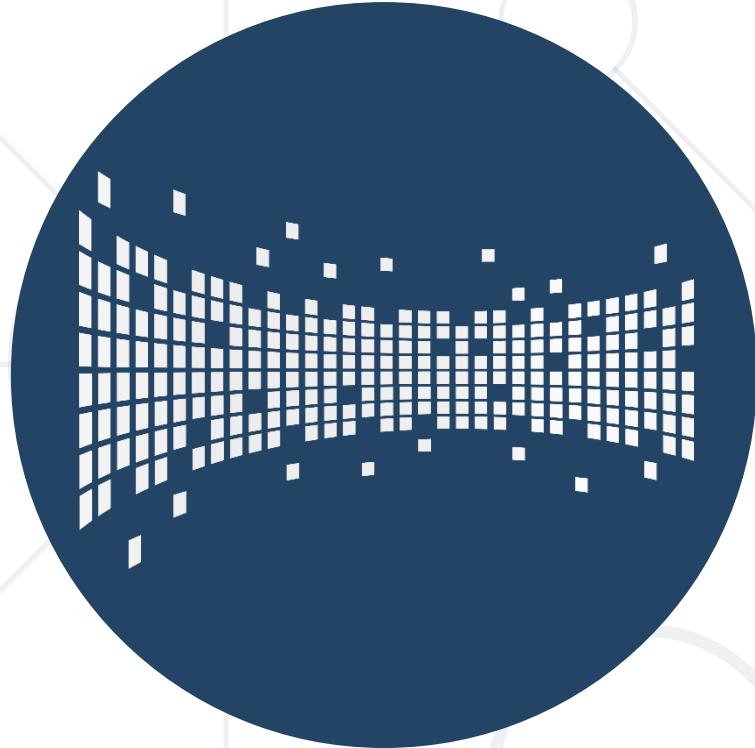
Solution: Say Hello (1)

```
public interface Person {  
    String getName();  
    default String sayHello() { return "Hello"; }  
}
```

```
public class European implements Person {  
    private String name;  
    public European(String name) { this.name = name; }  
    public String getName() { return this.name; }  
}
```

Solution: Say Hello (2)

```
public class Bulgarian implements Person {  
    private String name;  
    public Bulgarian(String name) {  
        this.name = name;  
    }  
    public String getName() { return this.name; }  
    public String sayHello() { return "Здравей"; }  
}  
// TODO: implement class Chinese
```



Abstract Classes

Abstract Class

- Cannot be instantiated
- May contain **abstract methods**
- Must provide an **implementation** for all **inherited interface members**
- Implementing an interface might map the interface methods onto **abstract** methods

```
public abstract class Animal {  
}
```



Abstract Methods

- Declarations are only permitted in **abstract classes**
- Bodies must be **empty** (no curly braces)
- An abstract method declaration provides **no** actual implementation:



```
public abstract void build();
```



Interfaces vs Abstract Classes

Interface vs Abstract Class (1)

- Interface
 - A class may **implement several interfaces**
 - **Cannot have access modifiers**, everything is assumed as public
- Abstract Class (AC)
 - May **inherit only one abstract** class
 - **Provides implementation** and/or just the **signature** that has to be overridden
 - Can **contain access modifiers** for the fields, functions, properties



Interface vs Abstract Class (2)

- Interface
 - If we add a **new method** we must **track down all the implementations** of the interface and **define implementation** for the new method
- Abstract Class
 - Fields and constants **can be defined**
 - If we add a **new method** we have the option of providing a **default implementation**



Problem: Say Hello Extended

- Refactor the code from the last problem
- Add **BasePerson abstract class**
 - In which move all **code duplication** from European, Bulgarian, Chinese

BasePerson

-name: String

#BasePerson(name)

-setName(): void

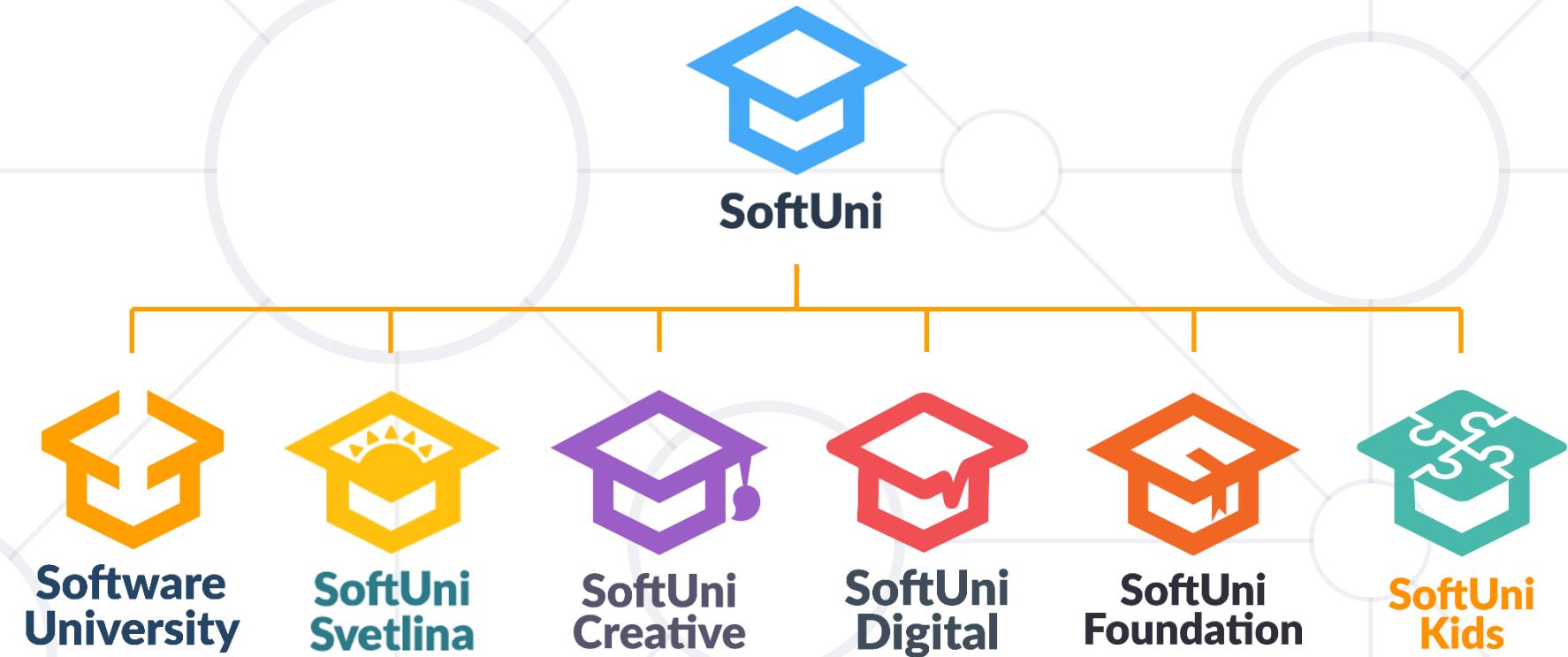
Solution: Say Hello Extended

```
public abstract class BasePerson implements Person {  
    private String name;  
    protected BasePerson(String name) {  
        this.setName(name);  
    }  
    private void setName(String name) { this.name = name; }  
    @Override  
    public String getName() {  
        return this.name;  
    }  
}
```

- Abstraction – **hiding** implementation and **showing** functionality
- Interfaces
 - **implements** vs **extends**
 - Default and Static methods
- Abstract classes
- Interfaces vs Abstract Classes



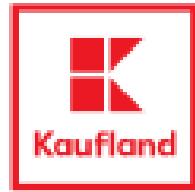
Questions?



SoftUni Diamond Partners



SCHWARZ



MOTION SOFTWARE

Educational Partners



VIRTUAL RACING SCHOOL



Trainings @ Software University (SoftUni)



- Software University – High-Quality Education, Profession and Job for Software Developers
 - softuni.bg
- Software University Foundation
 - softuni.foundation
- Software University @ Facebook
 - facebook.com/SoftwareUniversity
- Software University Forums
 - forum.softuni.bg



Software
University



- 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>



Polymorphism

Abstract Classes, Abstract Methods, Override Methods



SoftUni Team
Technical Trainers



SoftUni



Software University
<https://softuni.bg>

Table of Contents

1. Polymorphism

- What is Polymorphism?
- Types of Polymorphism
- Override Methods
- Overload Methods

2. Abstract Classes

- Abstract Methods

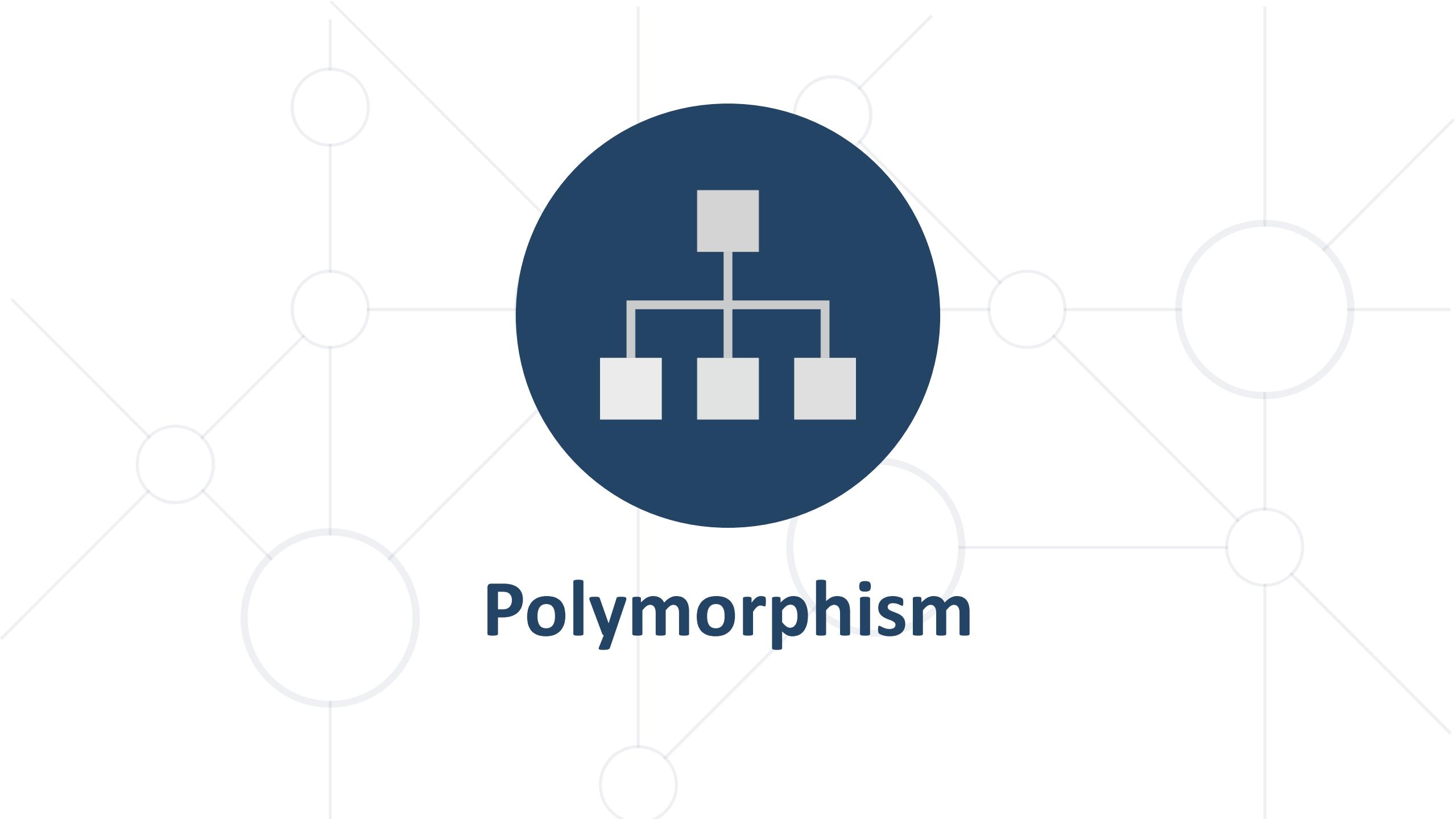


Have a Question?



sli.do

#java-advanced



Polymorphism

What is Polymorphism?

- From the Greek

Polys
(many)

Morphe
(shape/forms)

Polymorphos

- Such as a word having several different meanings based on the context
- Often referred to as the third pillar of OOP, after encapsulation and inheritance



Polymorphism in OOP

- The ability of an **object** to take on **many forms**

```
public interface Animal {}  
public abstract class Mammal {}  
public class Person extends Mammal implements Animal {}
```

Person IS-A Person

Person IS-A Mammal

Person IS-AN Animal

Person IS-AN Object

Reference Type and Object Type

```
public class Person extends Mammal implements Animal {}  
Animal person = new Person();  
Mammal personOne = new Person();  
Person personTwo = new Person();
```

Reference Type

Object Type

- **Variables** are saved in a **reference type**
- You can use only **reference methods**
- If you need an **object method** you need to **cast it or override it**

Keyword – instanceof

- Check if an **object** is an **instance** of a specific **class**

```
Mammal george = new Person();
```

```
Person peter = new Person();
```

Check object type of person

```
if (george instanceof Person) {  
    ((Person) george).getSalary();  
}
```

```
if (peter.getClass() == Person.class) {  
    ((Person) peter).getSalary();  
}
```

Cast to object type and use its methods

Types of Polymorphism

- **Runtime** polymorphism

```
public class Shape {}  
public class Circle extends Shape {}  
public static void main(String[] args) {  
    Shape shape = new Circle();  
}
```

Method
overriding

- **Compile time** polymorphism

```
int sum(int a, int b, int c){}  
double sum(Double a, Double b){}
```

Method
overloading

Compile Time Polymorphism

- Also known as **Static Polymorphism**

```
static int myMethod(int a, int b) {}  
static Double myMethod(Double a, Double b) {}
```

- Argument lists could **differ** in
 - Number of parameters
 - The data type of parameters
 - The sequence of Data type parameters

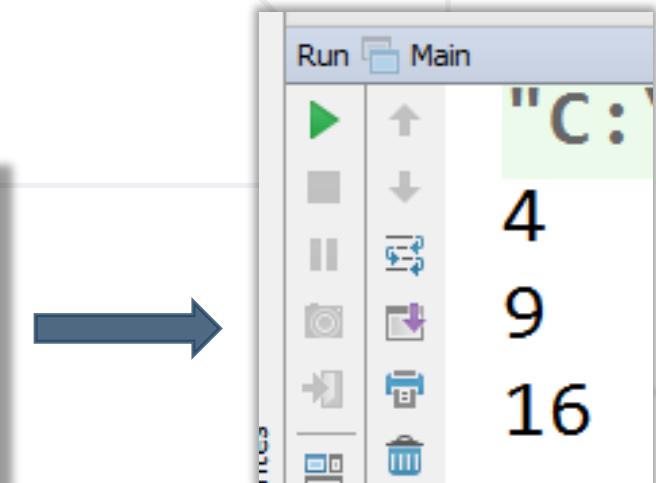
Method overloading

Problem: MathOperation

- Create a class **MathOperation**, which should have method **add()**
- Must be invoked with **two, three or four integers**

```
MathOperation  
+add(int a, int b): int  
+add(int a, int b, int c): int  
+add(int a, int b, int c, int d): int
```

```
MathOperation mathOperation = new MathOperation();  
  
System.out.println(mathOperation.add( a: 2, b: 2));  
System.out.println(mathOperation.add( a: 3, b: 3, c: 3));  
System.out.println(mathOperation.add( a: 4, b: 4, c: 4, d: 4));
```



Solution: MathOperation

```
public class MathOperation {  
    public int add(int a, int b) {  
        return a + b;  
    }  
    public int add(int a, int b, int c) {  
        return a + b + c;  
    }  
    public int add(int a, int b, int c, int d) {  
        return a + b + c + d;  
    }  
}
```

Rules for Overloading Method

- Overloading can take place in the **same class** or its **subclass**
- Constructors in Java can be **overloaded**
- Overloaded methods must have a **different argument list**
- The overloaded method should always be part of the same class (can also take place in a subclass), with the **same name** but **different parameters**
- They may have the **same** or **different return types**

Runtime Polymorphism

- Using of **override** method

```
public static void main(String[] args) {  
    Rectangle rect = new Rectangle(3.0, 4.0);  
    Rectangle square = new Square(4.0);  
  
    System.out.println(rect.area());  
    System.out.println(square.area());  
}
```

Method
overriding

Runtime Polymorphism

- Also known as **Dynamic Polymorphism**

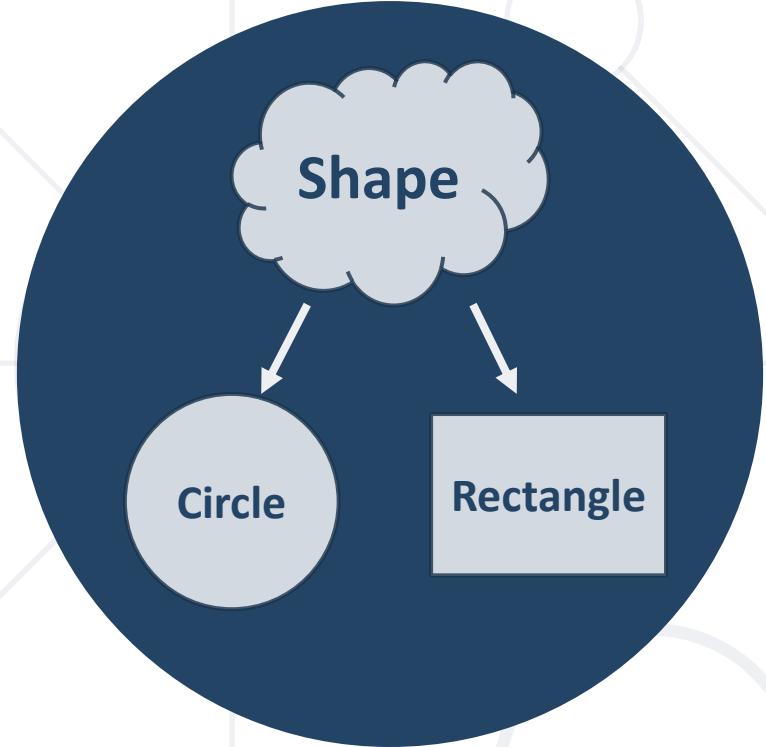
```
public class Rectangle {  
    public Double area() {  
        return this.a * this.b;  
    }  
}
```

```
public class Square extends Rectangle {  
    @Override  
    public Double area() {  
        return this.a * this.a;  
    }  
}
```

Method overriding

Rules for Overriding Method

- **Overriding** can take place in **sub-class**
- The **argument list** must be the **same** as that of the **parent method**
- The overriding method must have the **same return type**
- **Access modifier** cannot be more **restrictive**
- **Private**, **static**, and **final** methods can **NOT** be overridden
- The overriding method **must not** throw new or broader **checked exceptions**



Abstract Classes

Abstract Classes

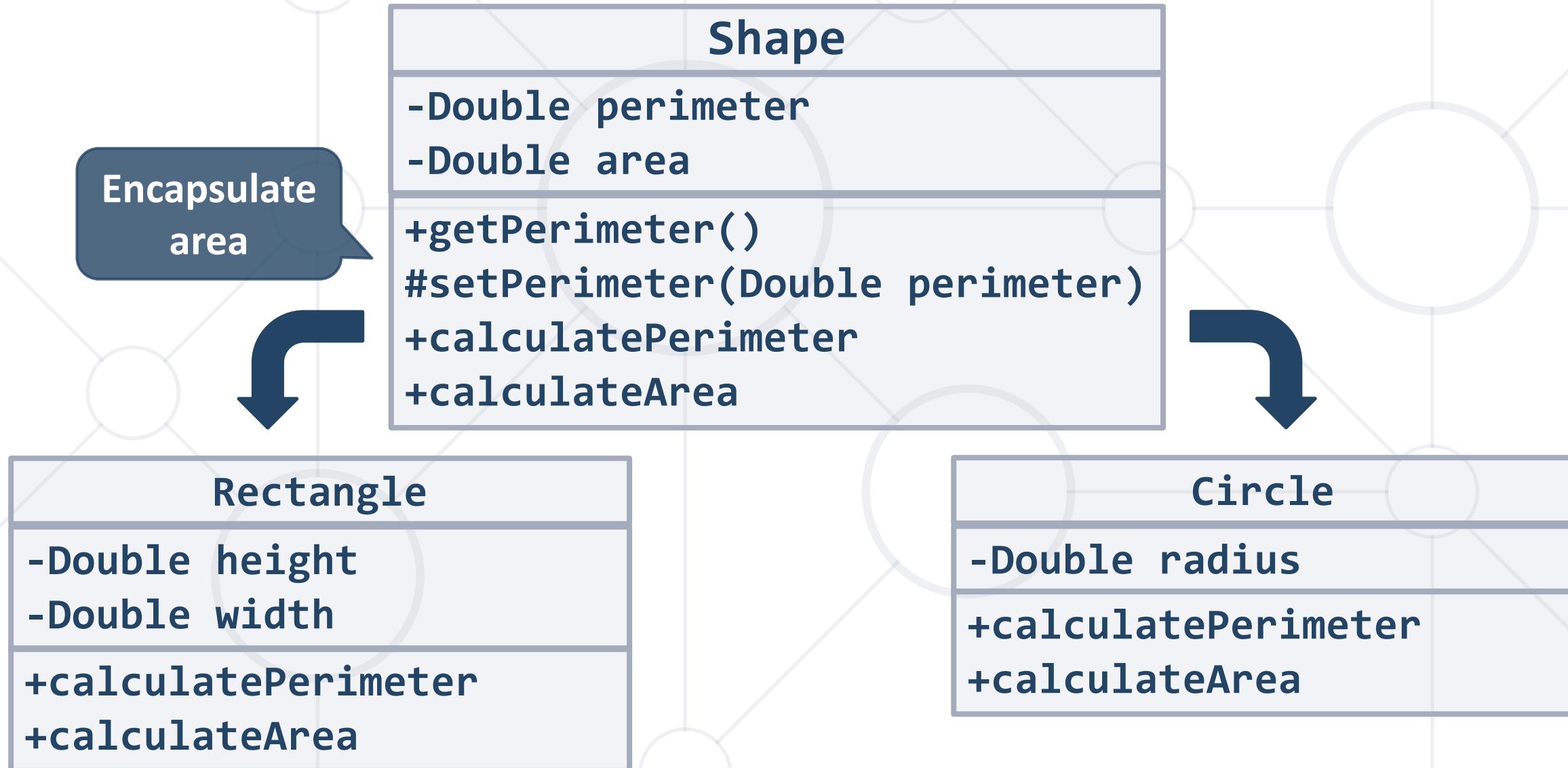
- An abstract class **can NOT** be instantiated

```
public abstract class Shape {}  
public class Circle extends Shape {}  
Shape shape = new Shape(); // Compile time error  
Shape circle = new Circle(); // polymorphism
```



- An **abstract** class may or may not include **abstract methods**
- If it has at least one abstract method, it must be declared **abstract**
- To use an **abstract class**, you need to **inherit** it

Problem: Shapes



Solution: Shapes (1)

```
public abstract class Shape {  
    private Double perimeter;  
    private Double area;  
    protected void setPerimeter(Double perimeter) {  
        this.perimeter = perimeter;  
    }  
    public Double getPerimeter() { return this.perimeter; }  
    protected void setArea(Double area) {this.area = area; }  
    public Double getArea() { return this.area; }  
    protected abstract void calculatePerimeter();  
    protected abstract void calculateArea();  
}
```

Solution: Shapes (2)

```
public class Rectangle extends Shape {  
    //TODO: Add fields  
    public Rectangle(Double height, Double width) {  
        this.setHeight(height); this.setWidth(width);  
        this.calculatePerimeter(); this.calculateArea(); }  
    //TODO: Add getters and setters  
    @Override  
    protected void calculatePerimeter() {  
        setPerimeter(this.height * 2 + this.width * 2); }  
    @Override  
    protected void calculateArea() {  
        setArea(this.height * this.width); } }
```

Solution: Shapes (3)

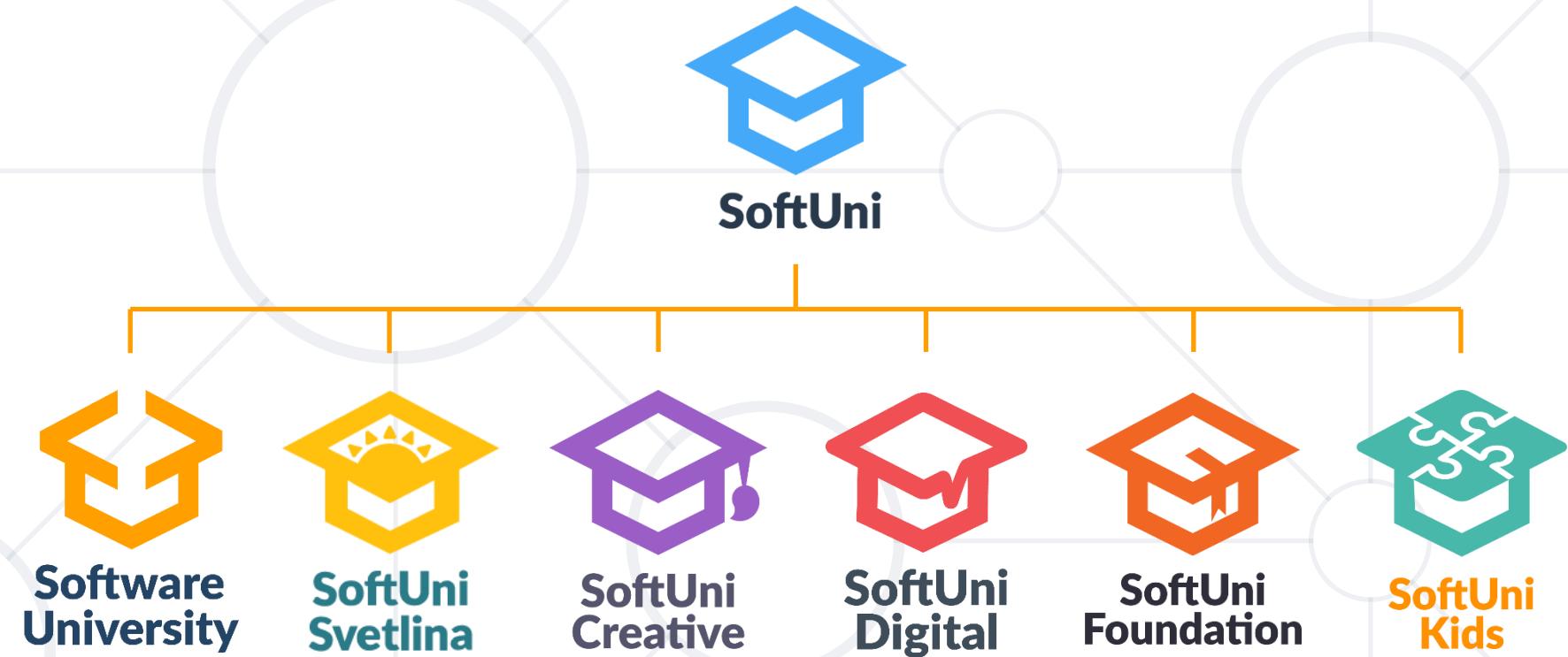
```
public class Circle extends Shape {  
    private Double radius;  
    public Circle (Double radius) {  
        this.setRadius(radius);  
        this.calculatePerimeter();  
        this.calculateArea();  
    }  
    public final Double getRadius() {  
        return radius;  
    }  
    //TODO: Finish encapsulation  
    //TODO: Override calculate Area and Perimeter  
}
```

Summary

- Polymorphism - **Definition and Types**
- Override Methods
- Overload Methods
- Abstraction
 - **Classes**
 - **Methods**



Questions?



SoftUni Diamond Partners



SCHWARZ



MOTION SOFTWARE

Educational Partners



VIRTUAL RACING SCHOOL



Trainings @ Software University (SoftUni)



- Software University – High-Quality Education, Profession and Job for Software Developers
 - softuni.bg
- Software University Foundation
 - softuni.foundation
- Software University @ Facebook
 - facebook.com/SoftwareUniversity
- Software University Forums
 - forum.softuni.bg



Software
University



- 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>



S.O.L.I.D.

The Benefits and Potential of Using SOLID Principles

- S → Single Responsibility
- O → Open/Closed
- L → Liskov substitution
- I → Interface Segregation
- D → Dependency Inversion



SoftUni Team
Technical Trainers



Table of Contents

1. S.O.L.I.D. Principles
2. Single Responsibility
3. Open / Closed
4. Liskov Substitution
5. Interface Segregation
6. Dependency Inversion

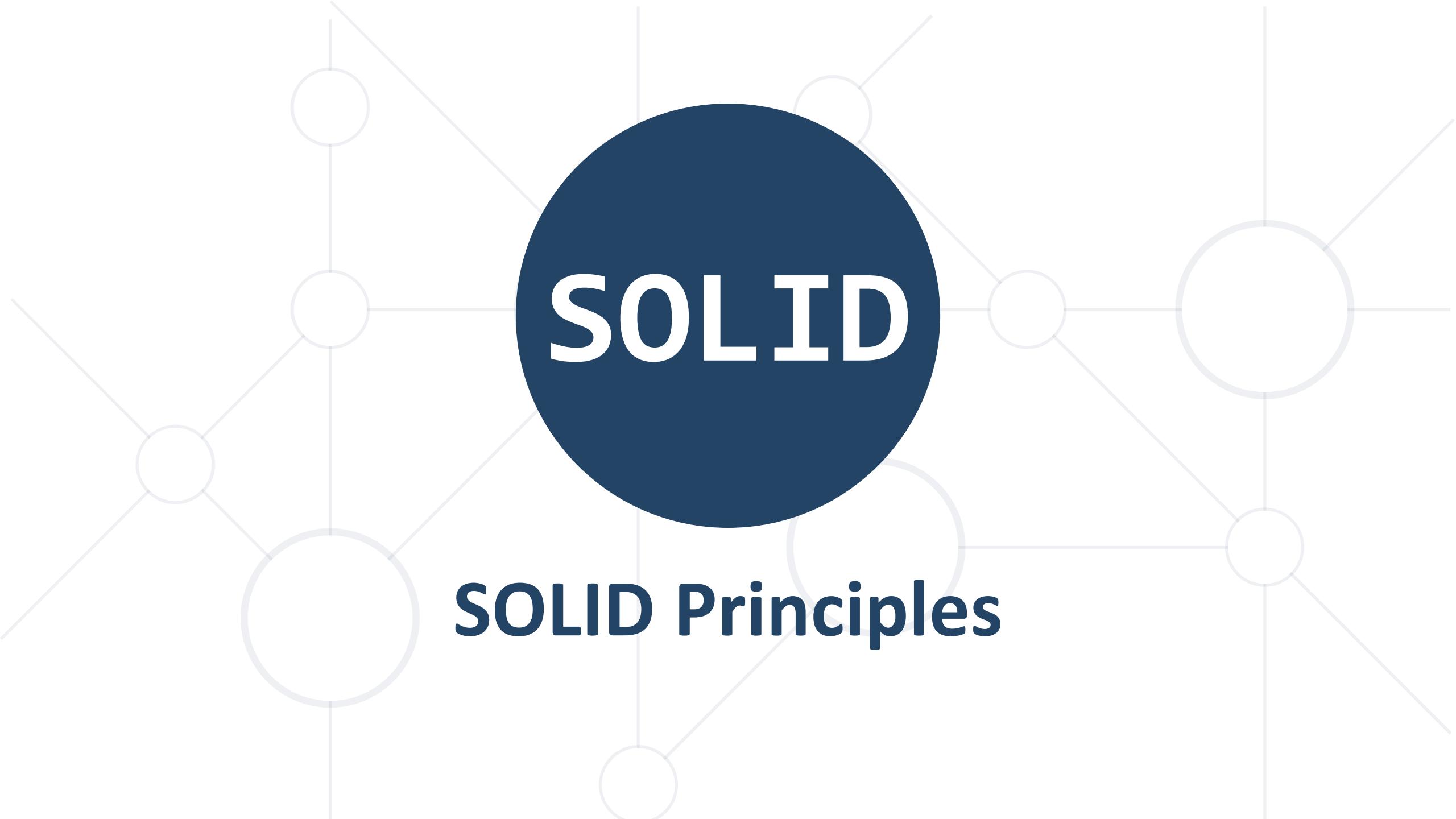


Have a Question?



sli.do

#java-advanced



SOLID

SOLID Principles

- **S – Single responsibility principle** – class should only have one responsibility
- **O – Open–closed principle** – open for extension, but closed for modification
- **L – Liskov substitution principle** – objects should be replaceable with instances of their subtypes without altering the correctness of that program

- I – Interface segregation principle – many specific interfaces are better than one general interface
- D – Dependency inversion principle – one should depend upon abstractions, not concretions



Single Responsibility

Single Responsibility Principle

- A class should **have only one responsibility**
 - Reduces **dependency** complexity
 - Each additional responsibility is an **axis to change the class**



```
public class HeroSettings {  
    public static void changeName(Hero hero) {  
        // Grant option to change  
    }  
}
```

Single Responsibility Principle

- Still, classes **can have multiple methods**
 - Each method should have a **single functionality** part of the class responsibility



```
public class HeroSettings {  
    public static void changeName(Hero hero) {  
        // Grant option to change name  
    }  
    public static void selectRole(Hero hero) {  
        // Grant option to select role  
    }  
}
```



Open / Closed

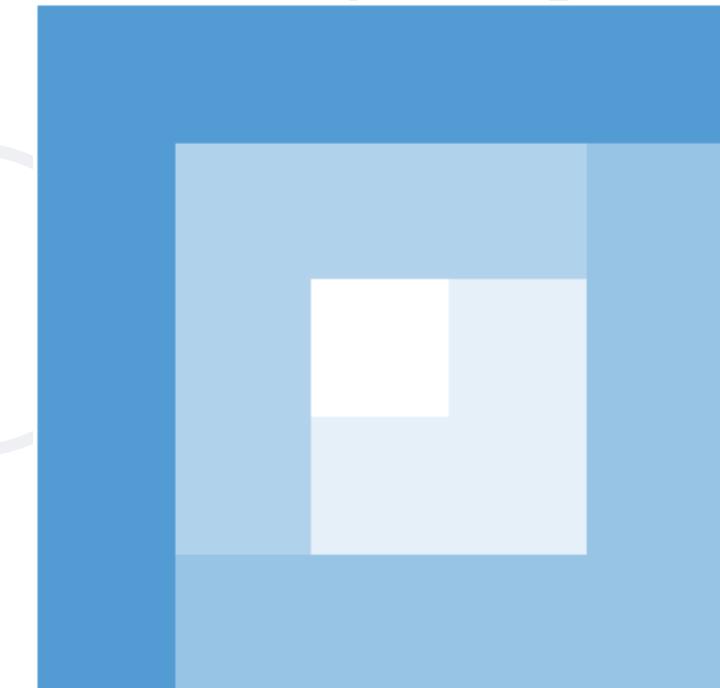
What is Open/Closed?

- Software entities (classes, modules, functions, etc.) should be
 - **open for extension**
 - **closed for modification**
- **Design** the code in a way that **new** functionality can be added with **minimum changes** in the **existing** code



Extensibility

- Implementation takes **future growth** into **consideration**
- New or **modified functionality** affects little or not at all the internal structure and data flow of the system



Reusability

- Software reusability refers to **design features** of a software **element** that enhance its **suitability** for **reuse**
- Modularity
- Low coupling
- High cohesion
- Coupling and Cohesion



- Cascading changes through modules
- Each change requires re-testing
- Logic depends on conditional statements



- Inheritance / Abstraction
- Inheritance / Template Method pattern
- Composition / Strategy patterns





Liskov Substitution

What is Liskov Substitution?

- Derived types must be **completely substitutable** for their base types
- Reference to the base class can be replaced with a derived class without affecting the functionality of the program module
- Derived classes extend without replacing the functionality of old classes



LSP Relationship

- OOP Inheritance

Student IS-A Person

- Plus LSP

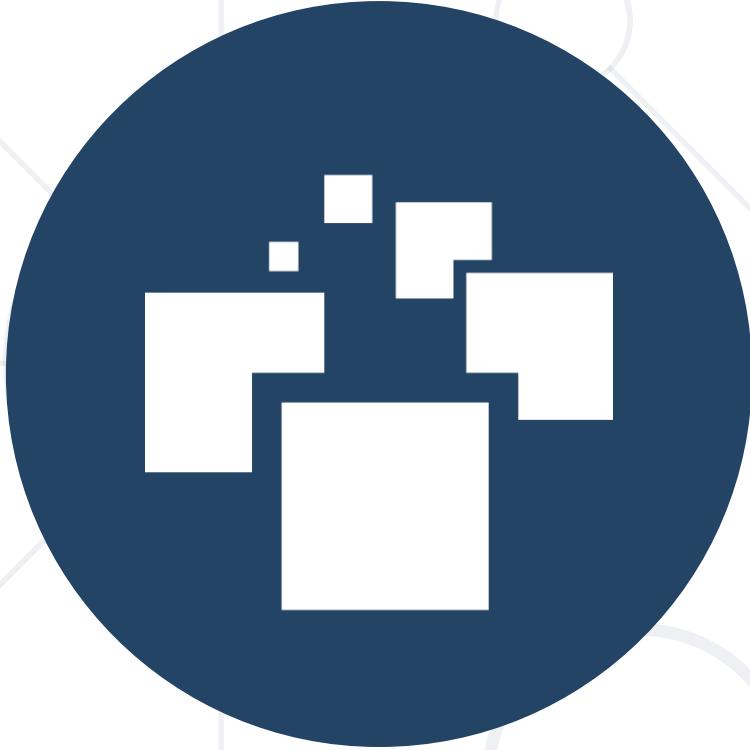
Student IS-SUBSTITUTED-FOR Person

OCP vs LSP

- Liskov Substitution Principle is just an **extension** of the Open-Closed Principle
- We must make sure that new derived classes are extending the base classes **without changing** their **behavior**



- Violations
 - Type Checking
 - Overridden methods say "I am not implemented"
 - Base class depends on its subtypes
- Solutions
 - Refactoring in the **base class**



Interface Segregation

ISP – Interface Segregation Principle

- Clients should **not** be forced to depend on methods they do **not** use
- Segregate interfaces
 - Prefer **small, cohesive** interfaces
 - Divide "**fat**" interfaces into "**role**" interfaces

Fat Interfaces

- Classes whose interfaces are not cohesive have "fat" interfaces

```
public interface Worker {  
    void work();  
    void sleep();  
}
```

Class Employee is
OK

```
public class Robot implements Worker {  
    public void work() {}  
    public void sleep() {  
        throw new UnsupportedOperationException();  
    }  
}
```

"Fat" Interfaces

- Having "**fat**" interfaces:
 - Classes have methods they do not use
 - Increased **coupling**
 - Reduced flexibility
 - Reduced maintainability

How to ISP?

- Solutions to broken ISP
 - Small interfaces
 - Cohesive interfaces
 - Let the client **define** interfaces – "role" interfaces

Cohesive Interfaces

- Small and Cohesive "Role" Interfaces

```
public interface Worker {  
    void work();  
}
```

```
public interface Sleeper {  
    void sleep();  
}
```

```
public class Robot implements Worker {  
    void work() {  
        // Do some work...  
    }  
}
```



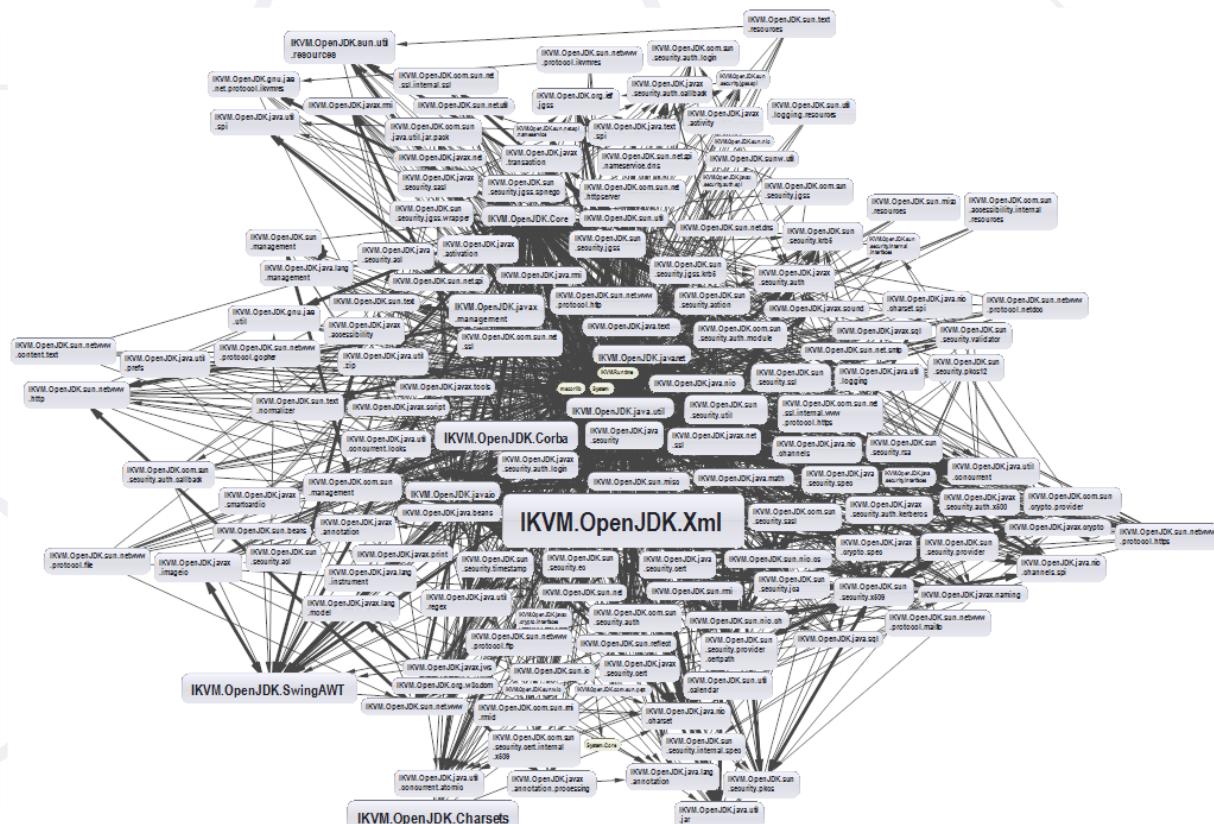
Dependency Inversion

Dependency Inversion Principle (DIP)

- High-level modules should not depend on low-level modules
 - Both should **depend on abstractions**
- Abstractions should not depend on details
- Details should depend on abstractions
- Goal: **decoupling between modules through abstractions**

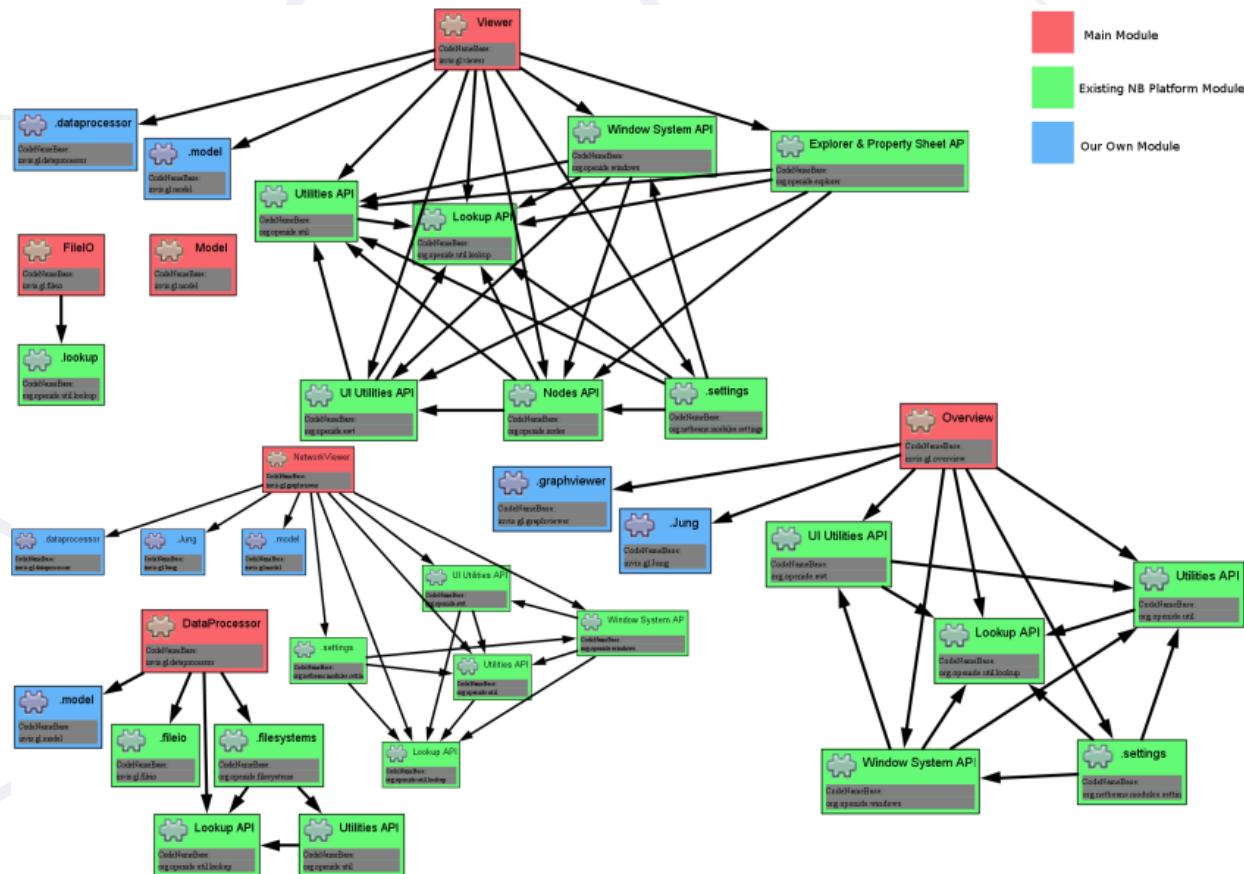
Dependencies and Coupling (1)

- What happens when modules **depend directly** on **other modules**



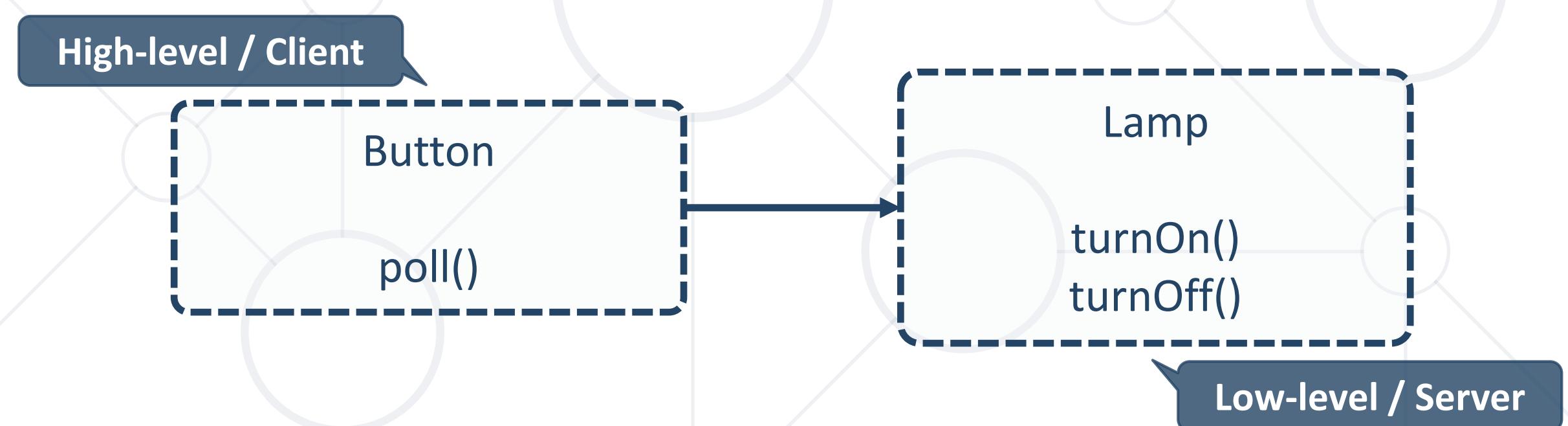
Dependencies and Coupling (2)

- The goal is to depend on abstractions



The Problem

- Button → Lamp Example – **Robert Martin**
- Button **depends on** Lamp



Dependency Inversion Solution

- Find the abstraction independent of details



Dependency Examples

- A **dependency** is an external component / system:
 - Framework
 - Third party library
 - Database
 - File system
 - Email
 - Web service
 - System resource (e.g. clock)
 - Configuration
 - The **new** keyword
 - Static method
 - Global function
 - Random generator
 - `System.in / System.out`

How to DIP? (1)

- **Constructor injection** - dependencies are passed through **constructors**
 - **Pros**
 - Classes **self-documenting** requirements
 - Works well without a container
 - Always **valid state**
 - **Cons**
 - Many parameters
 - Some methods may not need everything



Constructor Injection – Example

```
public class Copy {  
    private Reader reader;  
    private Writer writer;  
    public Copy(Reader reader, Writer writer) {  
        this.reader = reader;  
        this.writer = writer;  
    }  
    public void copyAll() {}  
}
```

How to DIP? (2)

- **Setter Injection** - dependencies are passed through **setters**
 - Pros
 - Can be changed anytime
 - Very **flexible**
 - Cons
 - Possible **invalid state** of the object
 - Less intuitive

Setter Injection – Example

```
public class Copy {  
    private Reader reader;  
    private Writer writer;  
    public void setReader(Reader reader) {}  
    public void setWriter(Writer writer) {}  
    public void copyAll() {}  
}
```

How to DIP? (3)

- **Parameter injection** - dependencies are passed through **method parameters**

- **Pros**

- No change in rest of the class
 - Very flexible

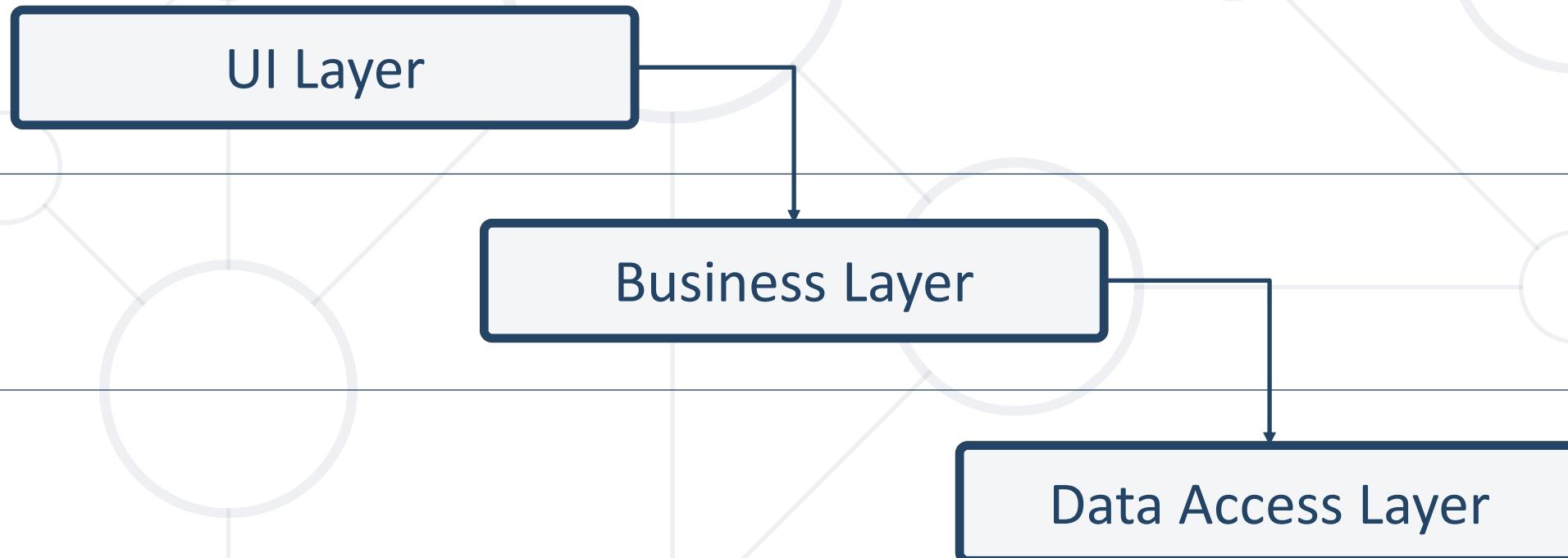
- **Cons**

- Many parameters
 - Breaks the method signature

```
public class Copy {  
    public void copyAll(Reader reader, Writer writer) {}  
}
```

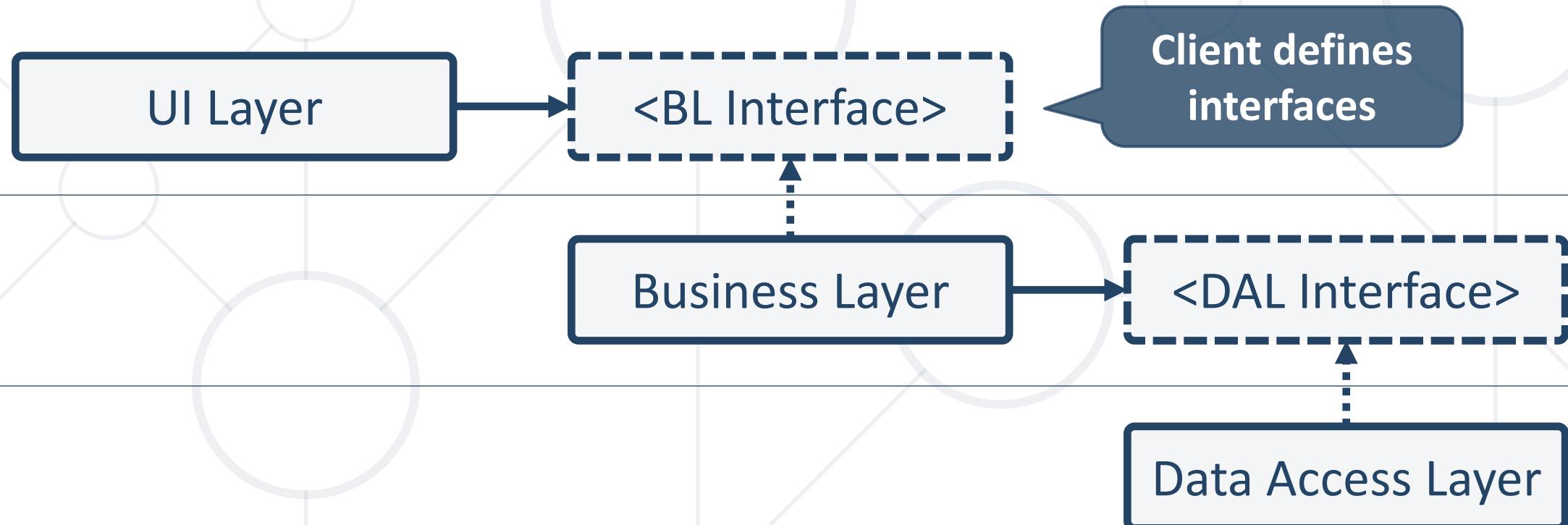
Layering (1)

- Traditional programming
 - **High-level** modules use **low-level** modules



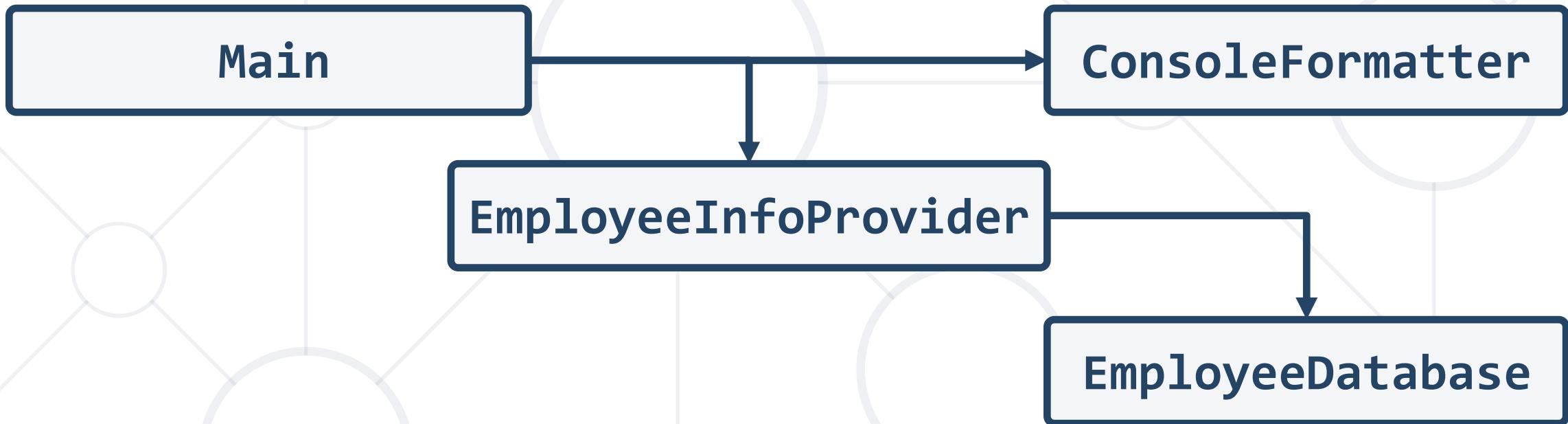
Layering (2)

- Dependency Inversion Layering
 - **High and low-level modules depend on abstractions**



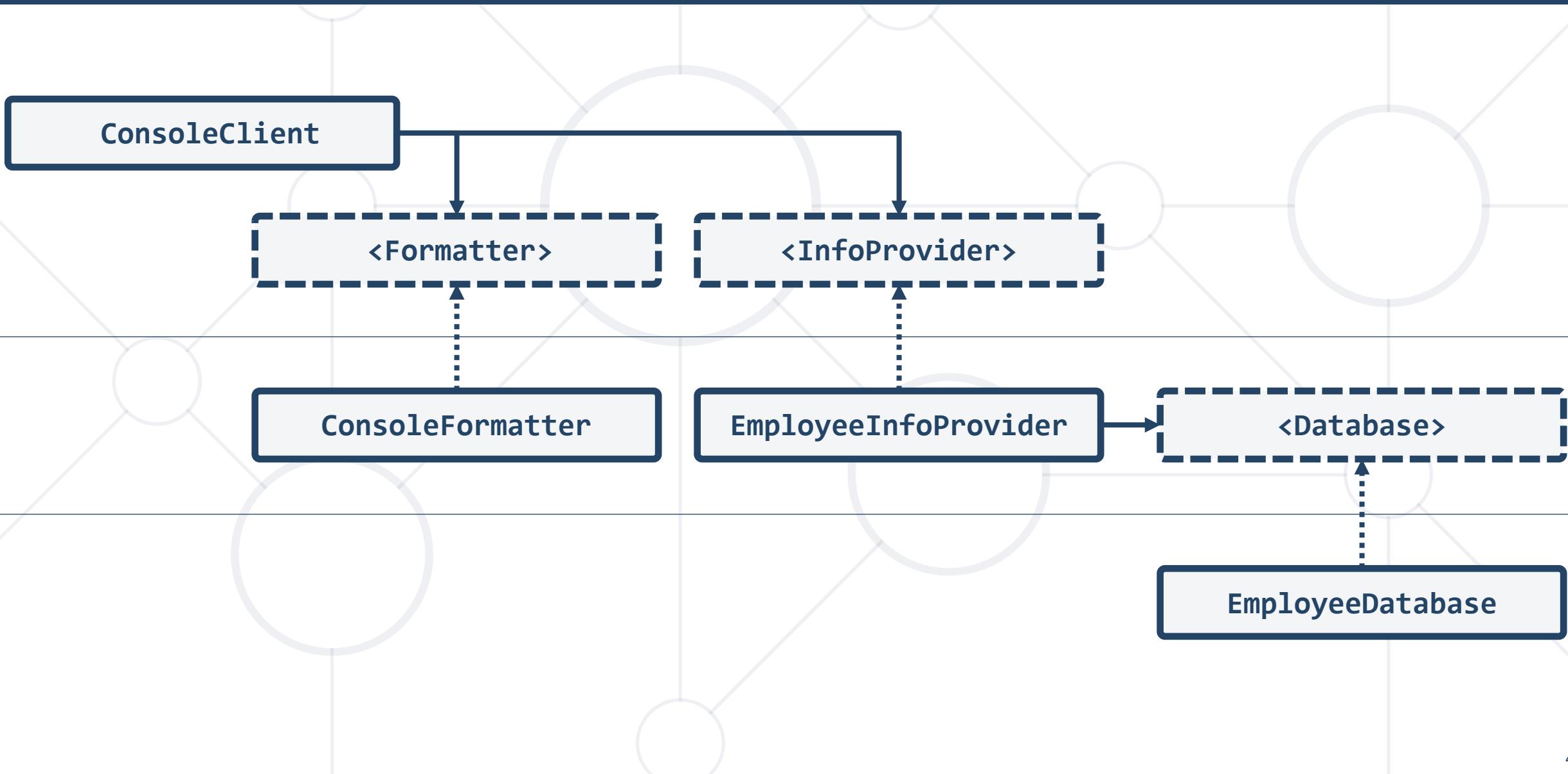
Problem: Employee Info

- You are given some classes



- Refactor the code so that it conforms to DIP

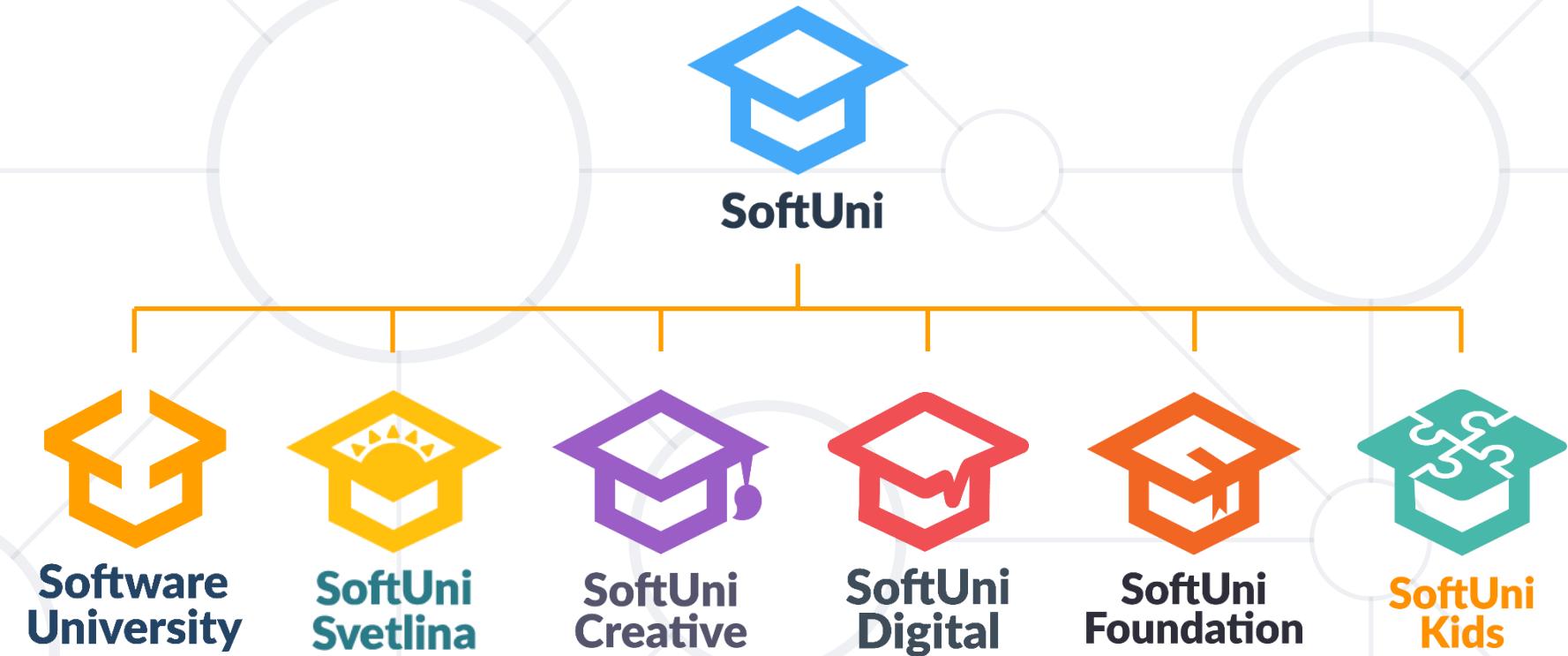
Solution: Employee Info



- **SOLID** principles make the software:
 - Understandable
 - Flexible
 - Maintainable



Questions?



SoftUni Diamond Partners



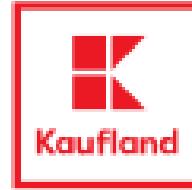
Coca-Cola HBC
Bulgaria



SUPER
HOSTING
.BG



SCHWARZ



MOTION SOFTWARE

Educational Partners



VIRTUAL RACING SCHOOL



Trainings @ Software University (SoftUni)



- Software University – High-Quality Education, Profession and Job for Software Developers
 - softuni.bg
- Software University Foundation
 - softuni.foundation
- Software University @ Facebook
 - facebook.com/SoftwareUniversity
- Software University Forums
 - forum.softuni.bg



Software
University



- 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>



Reflection and Annotations



SoftUni Team

Technical Trainers



SoftUni



Software University

<https://softuni.bg>

Table of Contents

1. Reflection - What? Why? Where?

2. Reflection API

- Reflecting Classes
- Reflecting Constructors
- Reflecting Fields
- Reflecting Methods
- Access Modifiers
- Reflecting Annotations



Have a Question?



sli.do

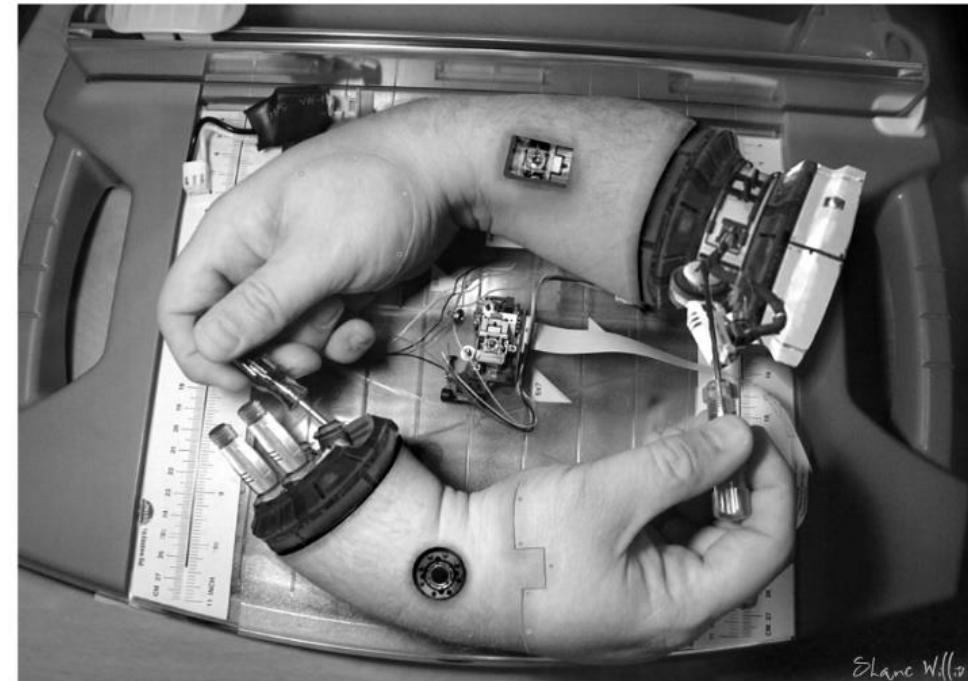
#java-advanced



Reflection

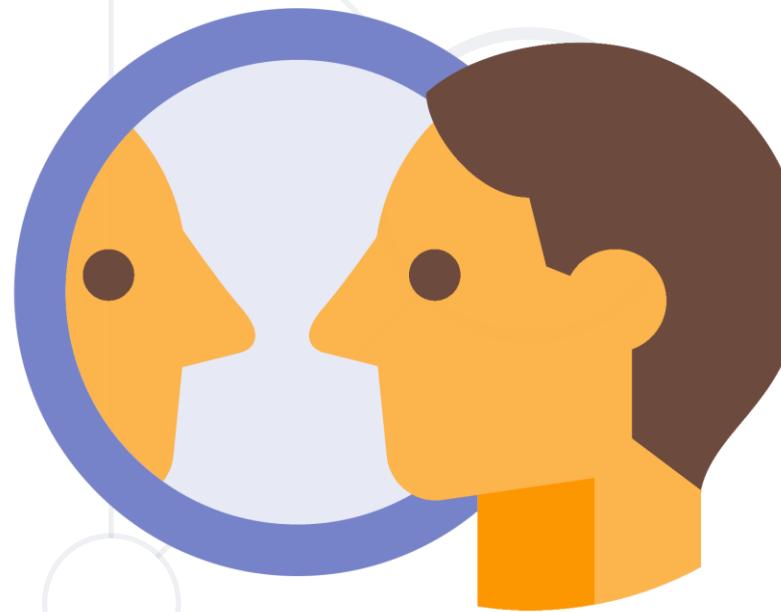
What is Metaprogramming?

- **Programming technique** in which computer programs have the ability to treat **programs as their data**
- The program can be designed to:
 - **Read**
 - **Generate**
 - **Analyze**
 - **Transform**
- **Modify itself while running**



What is Reflection?

- The ability of a programming language to be its **own metalinguage**
- Programs can examine information about **themselves**



When to Use Reflection?

- Whenever we want:
 - Code to become more **extendible**
 - To **reduce code length significantly**
 - Easier **maintenance**
 - Easier **testing**



When Not to Use Reflection?

- If it is **possible** to **perform** an operation **without** using **reflection**, then it is preferable to **avoid using it**
- Cons from using Reflection
 - **Performance** overhead
 - **Security** restrictions
 - Exposure of **internal logic**





Reflection API

The Class Object

- Obtain its **java.lang.Class** object

- If you **know the name**

```
Class myObjectClass = MyObject.class;
```

- If you **don't** know the name at **compile time**

```
Class myClass = Class.forName(className);
```

You need fully qualified
class name as String

Class Name

- Obtain **Class** name

- Fully qualified class name

```
String className = aClass.getName();
```

- Class name without the package name

```
String simpleClassName = aClass.getSimpleName();
```

- Obtain **parent class**

```
Class className = aClass.getSuperclass();
```

- Obtain **interfaces**

```
Class[] interfaces = aClass.getInterfaces();
```

- **Interfaces** are also **represented** by **Class** objects in Java Reflection
- Only the interfaces **specifically declared** implemented by a given class are **returned**

Problem: Reflection

- Import ReflectionClass to your **src** folder in your project
- Using **reflection** you should print:
 - This class type
 - Super class type
 - All Interfaces
 - Instantiate object using reflection and print it
- Don't change anything in class

Solution: Reflection

```
Class<Reflection> aClass = Reflection.class;  
System.out.println(aClass);  
System.out.println(aClass.getSuperclass());  
Class[] interfaces = aClass.getInterfaces();  
for (Class anInterface : interfaces)  
    System.out.println(anInterface);  
//Reflection ref = aClass.newInstance(); //Deprecated since Java 9  
Reflection ref = aClass.getDeclaredConstructor().newInstance();  
System.out.println(ref);
```

Create new object



Constructors, Fields and Methods

Constructors (1)

- Obtain **only public constructors**

```
Constructor[] ctors = aClass.getConstructors();
```

- Obtain **all constructors**

```
Constructor[] ctors =  
    aClass.getDeclaredConstructors();
```

- Get constructor by **parameters**

```
Constructor ctor =  
    aClass.getConstructor(String.class);
```

Constructors (2)

- Get **parameter types**

```
Class[] parameterTypes =  
    ctor.getParameterTypes();
```

- Instantiating objects using constructor

```
Constructor constructor =  
MyObject.class.getConstructor(String.class);  
  
MyObject myObject = (MyObject)constructor  
    .newInstance("arg1");
```

Fields Name and Type

- Obtain **public** fields

```
Field field = aClass.getField("somefield");
```

```
Field[] fields = aClass.getFields();
```

- Obtain **all** fields

```
Field[] fields = aClass.getDeclaredFields();
```

- Get field **name and type**

```
String fieldName = field.getName();
```

```
Object fieldType = field.getType();
```

Fields Set and Get

- Setting value for a field

```
Class aClass = MyObject.class;  
Field field = aClass.getDeclaredField("someField");  
MyObject objectInstance = new MyObject();  
field.setAccessible(true);  
Object value = field.get(objectInstance);  
field.set(objectInstance, value);
```

Change the behavior of the
AccessibleObject

The `objectInstance` parameter passed to
the `get` and `set` method should be an
instance of the class that owns the field

Methods

- Obtain **public** methods

```
Method[] methods = aClass.getMethods();
```

```
Method method =
```

```
aClass.getMethod("doSomething",String.class);
```

- Get methods without **parameters**

```
Method method =
```

```
aClass.getMethod("doSomething", null);
```

Method Invoke

- Obtain method **parameters** and **return type**

```
Class[] paramTypes = method.getParameterTypes();  
Class returnType = method.getReturnType();
```

- Get methods with **parameters**

```
Method method = MyObject.class  
    .getMethod("doSomething", String.class);  
Object returnValue = method.invoke(null, "arg1");
```

null is for static methods

Problem: Getters and Setters

- Using **reflection** get all methods and print:
- **Sort** getters and setters **alphabetically**
- **Getters:**
 - A getter method have its name start with "get", take 0 parameters, and returns a value
- **Setters:**
 - A setter method have its name start with "set", and takes 1 parameter

Solution: Getters

```
Method[] methods = Reflection.class.getDeclaredMethods();
Method[] getters = Arrays.stream(methods)
    .filter(m -> m.getName().startsWith("get") &&
        m.getParameterCount() == 0)
    .sorted(Comparator.comparing(Method::getName))
    .toArray(Method[]::new);
Arrays.stream(getters).forEach(m ->
    System.out.printf("%s will return class %s%n",
        m.getName(), m.getReturnType().getName()));
```



Access Modifiers

Access Modifiers

- Obtain the **class modifiers** like this

```
int modifiers = aClass.getModifiers();
```

- Each modifier is a **flag bit** that is either set or cleared
- You can check the modifiers

```
Modifier.isPrivate(modifiers);
```

```
Modifier.isProtected(modifiers);
```

```
Modifier.isPublic(modifiers);
```

```
Modifier.isStatic(modifiers);
```

getModifiers() can be called on constructors, fields, methods

- Creating arrays via Java Reflection

```
int[] intArray = (int[]) Array.newInstance(int.class, 3);
```

- Obtain parameter annotations

```
Array.set(intArray, 0, 123);
```

```
Array.set(intArray, 1, 456);
```

- Obtain fields and methods annotations

```
Class stringArrayComponentType =
```

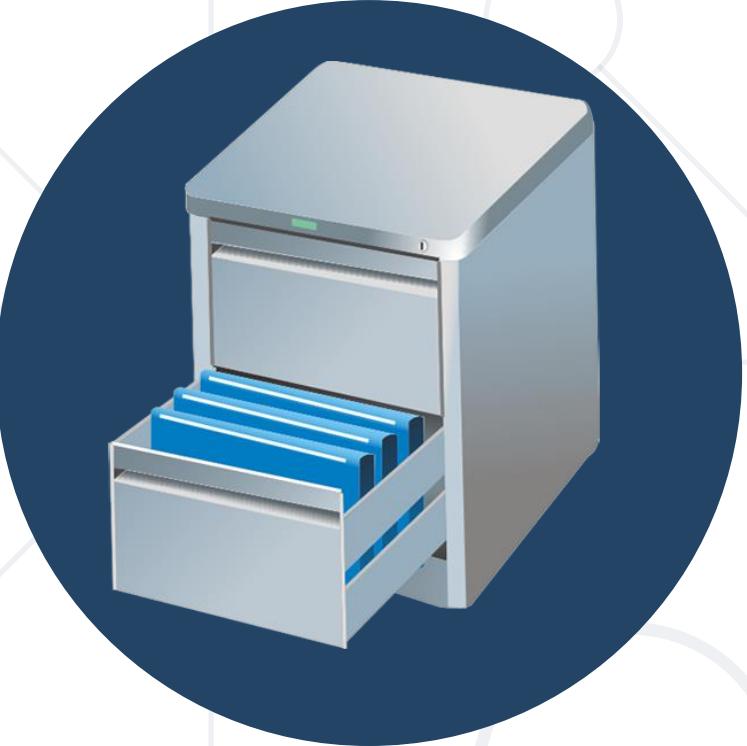
```
stringArrayClass.getComponentType();
```

Problem: High Quality Mistakes

- You perfectly know how to write High Quality Code
- Check **Reflection class** and print all mistakes in **access modifiers** which you can find
- Get all fields, getters and setters and sort each category by name
- First print mistakes in **fields**
- Then print mistakes in **getters**
- Then print mistakes in **setters**

Solution: High Quality Mistakes

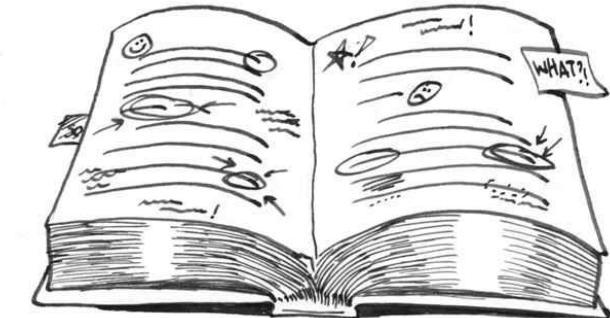
```
Field[] fields = Reflection.class.getDeclaredFields();
Arrays.stream(fields)
    .filter(f -> !Modifier.isPrivate(f.getModifiers()))
    .sorted(Comparator.comparing(Field::getName))
    .forEach(f -> System.out
        .printf("%s must be private!%n", f.getName()));
// TODO: Do the same for getters and setters
```



Annotations

Annotation

- **Data holding** class
- **Describe** parts of your code
- Applied to: **Classes, Fields, Methods**, etc.



```
@Deprecated
```

```
public void deprecatedMethod() {  
    System.out.println("Deprecated!");  
}
```

Annotation Usage

- To generate **compiler messages** or **errors**

```
@SuppressWarnings("unchecked")
```

```
@Deprecated
```

- As tools

- **Code generation** tools
 - **Documentation generation** tools
 - **Testing** Frameworks

- At runtime – **ORM**, **Serialization**, etc.

Built-In Annotations (1)

- **@Override** – generates **compile time error** if the method does not override a method in a parent class

```
@Override  
public String toString() {  
    return "new toString() method";  
}
```

Built-In Annotations (2)

- **@SuppressWarning** – turns off **compiler warnings**

```
@SuppressWarnings(value = "unchecked")
public <T> void warning(int size) {
    T[] unchecked = (T[]) new Object[size];
}
```

Annotation
with value

Generates compiler
warning

Built-In Annotations (3)

- **@Deprecated** – generates a **compiler warning** if the element is used

@Deprecated

Generates compiler
warning

```
public void deprecatedMethod() {  
    System.out.println("Deprecated!");  
}
```

Creating Annotations

- **@interface** – the keyword for annotations

```
public @interface MyAnnotation {  
    String myValue() default "default";  
}
```

Annotation element

```
@MyAnnotation(myValue = "value")  
public void annotatedMethod() {  
    System.out.println("I am annotated");  
}
```

Skip name if you have only
one value named "value"

Annotation Elements

- Allowed types for annotation elements:
 - Primitive types (**int**, **long**, **boolean**, etc.)
 - **String**
 - **Class**
 - **Enum**
 - **Annotation**
 - **Arrays** of any of the above

Meta Annotations – @Target

- Meta annotations annotate annotations
- **@Target** – specifies where the annotation is applicable

```
@Target(ElementType.FIELD)
```

Used to annotate
fields only

```
public @interface FieldAnnotation {  
}
```

- Available element types – **CONSTRUCTOR**, **FIELD**, **LOCAL_VARIABLE**, **METHOD**, **PACKAGE**, **PARAMETER**, **TYPE**

Meta Annotations – @Retention

- **@Retention** – specifies where the annotation is available

```
@Retention(RetentionPolicy.RUNTIME)  
public @interface RuntimeAnnotation {  
    // ...  
}
```

You can get info
at runtime

- Available retention policies – **SOURCE**, **CLASS**, **RUNTIME**

Problem: Create Annotation

- Create annotation **Subject** with a **String[]** element "categories"
 - Should be **available at runtime**
 - Can be **placed only on types**

```
 @Subject(categories = {"Test", "Annotations"})
public class TestClass {

}
```

Solution: Create Annotation

```
@Target(ElementType.TYPE)
@Retention(RetentionPolicy.RUNTIME)
public @interface Subject {
    String[] categories();
}
```

Annotations

- Obtain class annotations

```
Annotation[] annotations = aClass.getAnnotations();
```

```
Annotation annotation = aClass.getAnnotation(MyAnno.class);
```

- Obtain parameter annotations

```
Annotation[][] parameterAnnotations =
```

```
    method.getParameterAnnotations();
```

- Obtain fields and methods annotations

```
Annotation[] fieldAnots = field.getDeclaredAnnotations();
```

```
Annotation[] methodAnot = method.getDeclaredAnnotations();
```

Accessing Annotation (1)

- Some annotations can be accessed **at runtime**

```
@Author(name = "Gosho")  
  
public class AuthoredClass {  
    public static void main(String[] args) {  
        Class cl = AuthoredClass.class;  
        Author author = (Author) cl.getAnnotation(Author.class);  
        System.out.println(author.name());  
    }  
}
```

Accessing Annotation (2)

- Some annotations can be accessed **at runtime**

```
Class cl = AuthoredClass.class;  
  
Annotation[] annotations = cl.getAnnotations();  
  
for (Annotation annotation : annotations) {  
    if (annotation.annotationType().equals(Author.class)) {  
        Author author = (Author) annotation;  
        System.out.println(author.name());  
    }  
}
```

Problem: Coding Tracker

- Create annotation **Author** with a String element "name"
 - Should be **available at runtime**
 - Should be **placed only on methods**
- Create a class **Tracker** with a method:
 - **public static void printMethodsByAuthor()**

```
@Author(name = "George")
public static void main(String[] args) {
    Tracker.printMethodsByAuthor(Tracker.class);
}

@Author(name = "Peter")
public static void printMethodsByAuthor(Class<?> cl) { ... }
```



```
George: main()
Peter: printMethodsByAuthor()
```

Solution: Coding Tracker (1)

```
public class Tracker {  
    public static void printMethodsByAuthor(Class<?> cl) {  
        Map<String, List<String>> methodsByAuthor = new HashMap<>();  
        Method[] methods = cl.getDeclaredMethods();  
  
        for (Method method : methods) {  
            Author annotation = method.getAnnotation(Author.class);  
            // Continues on next slide
```

Solution: Coding Tracker (2)

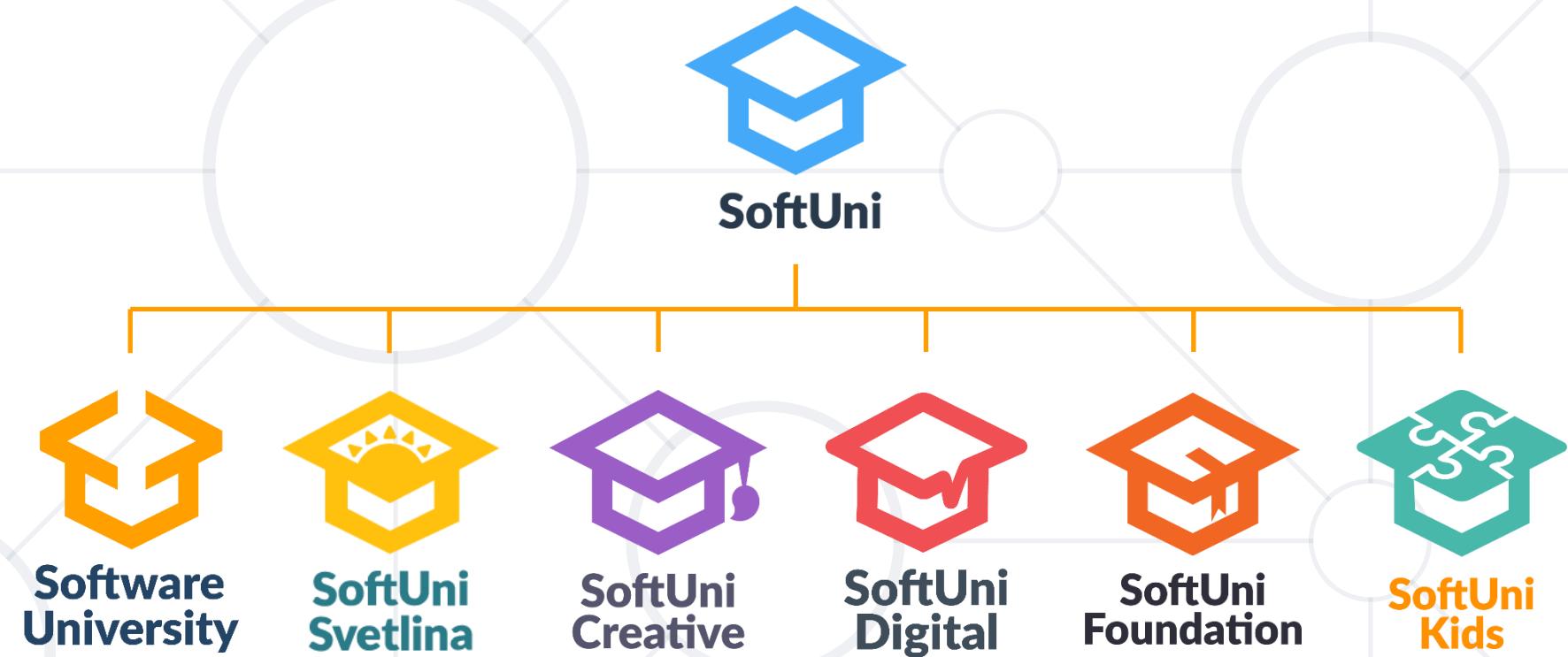
```
if (annotation != null) {  
    methodsByAuthor  
        .putIfAbsent(annotation.name(), new ArrayList<>());  
    methodsByAuthor  
        .get(annotation.name()).add(method.getName() + "()");  
}  
}  
}  
}  
}  
  
// TODO: print the results
```

Summary

- What is **Reflection**
- Reflection **API**
 - Reflecting Classes, Constructors, Fields, Methods
 - Access Modifiers
- **Annotations**
 - Used to describe our code
 - Provide the possibility to work with non-existing classes
 - Can be accessed through **reflection**



Questions?



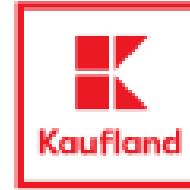
SoftUni Diamond Partners



Coca-Cola HBC
Bulgaria



SCHWARZ



MOTION SOFTWARE

Educational Partners



VIRTUAL RACING SCHOOL



Trainings @ Software University (SoftUni)



- Software University – High-Quality Education, Profession and Job for Software Developers
 - softuni.bg
- Software University Foundation
 - softuni.foundation
- Software University @ Facebook
 - facebook.com/SoftwareUniversity
- Software University Forums
 - forum.softuni.bg



Software
University

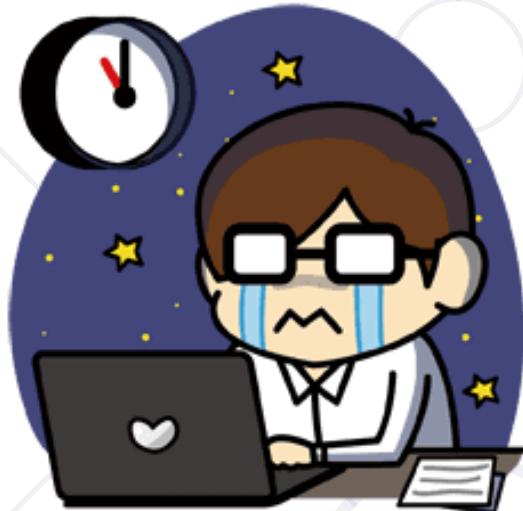


- This course (slides, examples, demos, exercises, homework, documents, videos and other assets) is **copyrighted content**
- Unauthorized copy, reproduction or use is illegal
- © SoftUni – <https://about.softuni.bg>
- © Software University – <https://softuni.bg>



Exception Handling

Handling Errors During the Program Execution



SoftUni Team

Technical Trainers



SoftUni



Software University

<https://softuni.bg>

Have a Question?



sli.do

#java-advanced

Table of Contents

1. What Are Exceptions in Java?

- The **Throwable** and **Exception** Classes
- Types of Exceptions and Their Hierarchy

2. Handling Exceptions: **try-catch-finally**

3. Raising (Throwing) Exceptions: **throw**

4. Best Practices in Exception Handling

5. Defining Custom Exceptions Classes





What Are Exceptions?

Notifications about Failed Operations

What Are Exceptions?

- **Exceptions** handle **errors** and **problems** at runtime
- **Throw** an exception to **signal** about a problem

```
if (size < 0)
    throw new Exception("Size cannot be negative!");
```

- **Catch** an exception to **handle** the problem

```
try {
    size = Integer.parseInt(text);
} catch (Exception ex) {
    System.out.println("Invalid size!");
}
```



More About Exceptions

- **Exceptions** occur when the normal flow of the program is interrupted due to a problem (or error)
 - When an operation **fails to execute** at runtime
 - **Example:** trying to read a non-existing file
- **Exceptions** allow problematic situations to be **handled** at multiple levels
 - Simplify code construction and maintenance
- **Exception objects** hold detailed information about the error: **error message, stack trace, etc.**



Unhandled Exception with Stack Trace

```
int x = Integer.parseInt("invalid number");
```

Error message

```
"C:\Program Files\Java\jdk-17.0.1\bin\java.exe" "-javaagent:C:\Program Files\JetBrains\IntelliJ IDEA Communi  
Exception in thread "main" java.lang.NumberFormatException Create breakpoint : For input string: "invalid number"  
at java.base/java.lang.NumberFormatException.forInputString(NumberFormatException.java:67)  
at java.base/java.lang.Integer.parseInt(Integer.java:668)  
at java.base/java.lang.Integer.parseInt(Integer.java:786)  
at ExceptionExample.main(ExceptionExample.java:5)
```

Stack trace

The Throwable Class

- Exceptions in Java are **objects**
- The **Throwable** class is a base for all Java exceptions
 - Contains information for the cause of the problem
 - **Message** – a text description of the exception
 - **StackTrace** – the snapshot of the "call stack" at the moment when the exception is thrown



Types of Exceptions in Java

- All Java exceptions inherit from **java.lang.Throwable**
- Direct descendants of **Throwable**:
 - **Error** – not expected to be caught from the program under normal circumstances
 - Examples: **StackOverflowError**, **OutOfMemoryError**
 - **Exception**
 - Used for exceptional conditions that user programs could catch
 - Examples: **ArithmeticException**, **IOException**

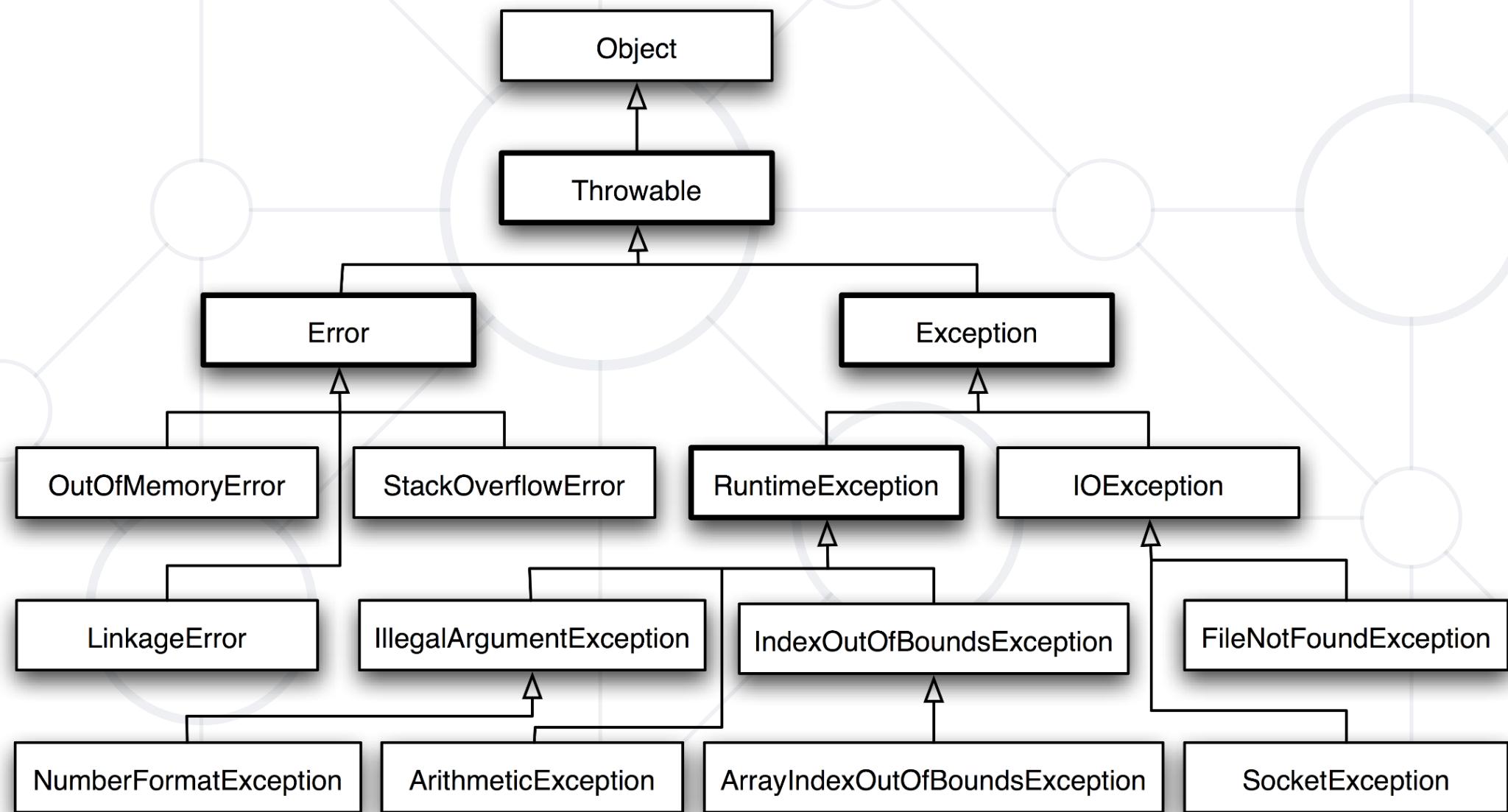
Exceptions

- **Exceptions** are two types:
 - **Checked** – an exceptions that should be obligatory handled → checked by the compiler during the compilation
 - Also called **compile-time** exceptions
- **Unchecked** – exceptions that occur at the time of execution
 - Also called **runtime exceptions**, not obligatory handled

```
public static void main(String args[]) {  
    File file = new File("non-existing-file.txt");  
    FileReader fr = new FileReader(file);  
}
```

FileNotFoundException

Exception Hierarchy in Java

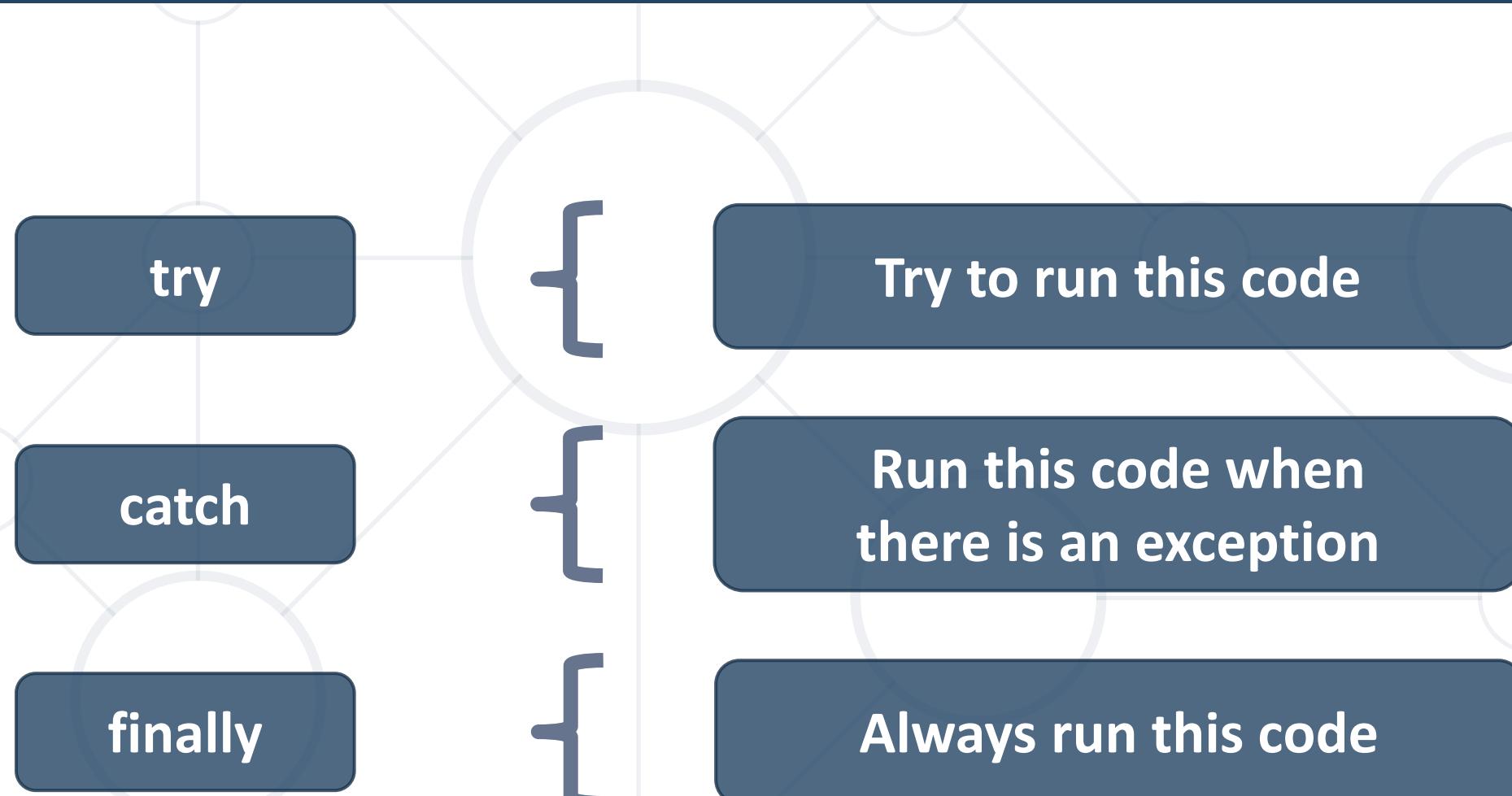




Handling Exceptions

Using **try-catch-finally**

How Do Exceptions Work?



Handling Exceptions

- In Java exceptions can be handled by the **try-catch** construction

```
try {  
    // Do some work that can raise an exception  
} catch (SomeException) {  
    // Handle the caught exception  
}
```



- **catch** blocks can be used multiple times to process different exception types

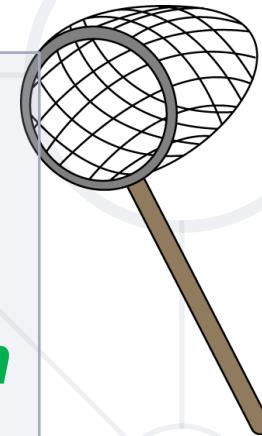
Using try-catch – Example

```
Scanner scanner = new Scanner(System.in);
String s = scanner.nextLine();
try {
    Integer.parseInt(s);
    System.out.printf(
        "You entered a valid integer number: %s", s);
} catch (NumberFormatException ex) {
    System.out.println("Invalid integer number!");
}
```

Handling Exceptions

- When **catching** an exception of a particular class, all its descendants (child exceptions) are caught too, e.g.

```
try {  
    // Do some work that can cause an exception  
} catch (IndexOutOfBoundsException iobEx) {  
    // Handle the caught out-of-bounds exception  
}
```



- Handles **IndexOutOfBoundsException** and its descendants **ArrayIndexOutOfBoundsException** and **StringIndexOutOfBoundsException**

Find the Mistake!

```
String str = scanner.nextLine();

try {
    Integer.parseInt(str);
} catch (Exception ex) {Should be last
    System.out.println("Cannot parse the number!");
} catch (NumberFormatException ex) {Unreachable code
    System.out.println("Invalid integer number!");
}
```

Problem: Number in Range

- Write a program to **enter an integer in a certain range**, e. g. 10-20
 - Read a **range** (two integers `start <= end`) and print the range
 - When an invalid number is entered or the number is out of range, print "**Invalid number: {num}**" and enter a number again
 - When the entered number is valid, print "**Valid number: {num}**"

10 20
5
xx
20



Range: [10...20]
Invalid number: 5
Invalid number: xx
Valid number: 20

-5 50
hi
-6
-1



Range: [-5...50]
Invalid number: hi
Invalid number: -6
Valid number: -1

Solution: Number in Range

```
private static int readNumberInRange(
    Scanner scanner, int start, int end) {
    while (true) {
        String line = scanner.nextLine();
        try {
            int num = Integer.parseInt(line);
            if (num >= start && num <= end)
                return num; // Valid number (in range)
        } catch (Exception ex) {
            // Parse failed --> invalid number
        }
        System.out.println("Invalid number: " + line);
    }
}
```

Solution: Number in Range (2)

```
public static void main(String[] args) {  
    Scanner scanner = new Scanner(System.in);  
  
    String[] range = scanner.nextLine().split(" ");  
    int start = Integer.parseInt(range[0]);  
    int end = Integer.parseInt(range[1]);  
    System.out.printf("Range: [%d...%d]\n", start, end);  
  
    int num = readNumberInRange(scanner, start, end);  
    System.out.println("Valid number: " + num);  
}
```



Try-Finally

Executing a Cleanup Code in All Cases

The try-finally Statement

- The statement:

```
try {  
    // Do some work that can cause an exception  
} finally {  
    // This block will always execute  
}
```

- Ensures execution of a given block in all cases
 - When an **exception** is raised or **not** in the **try** block
- Used for execution of **cleaning-up code**, e.g. releasing resources

Try-finally – Example

```
static void tryFinallyExample() {  
    System.out.println("Code executed before try-finally.");  
    try {  
        String str = scanner.nextLine();  
        Integer.parseInt(str);  
        System.out.println("Parsing was successful.");  
        return; // Exit from the current method → executes the "finally" block  
    } catch (NumberFormatException ex) {  
        System.out.println("Parsing failed!");  
    } finally {  
        System.out.println("This cleanup code is always executed.");  
    }  
    System.out.println("This code is after the try-finally block.");  
}
```



Throwing Exceptions

Using the "throw" Keyword

Using Throw Keyword

- **Throwing** an exception with an error message:

```
throw new IllegalArgumentException("Invalid amount!");
```

- Exceptions can accept **message** and **cause** (nested exception):

```
try {  
    ...  
} catch (SQLException sqlEx) {  
    throw new IllegalStateException("Cannot save invoice", sqlEx);  
}
```

- **Note:** if the original exception is not passed, the initial cause of the exception is lost

Throwing Exceptions

- Exceptions are **thrown** (raised) by the **throw** keyword
- Used to **notify the calling code** in case of an error or unusual situation
- When an exception is thrown:
 - The program execution **stops immediately**
 - The exception **travels over the stack**
 - Until a matching **catch** block is reached to handle it
- **Unhandled exceptions** display an error message

Re-Throwing Exceptions

- Caught exceptions can be **re-thrown** again:

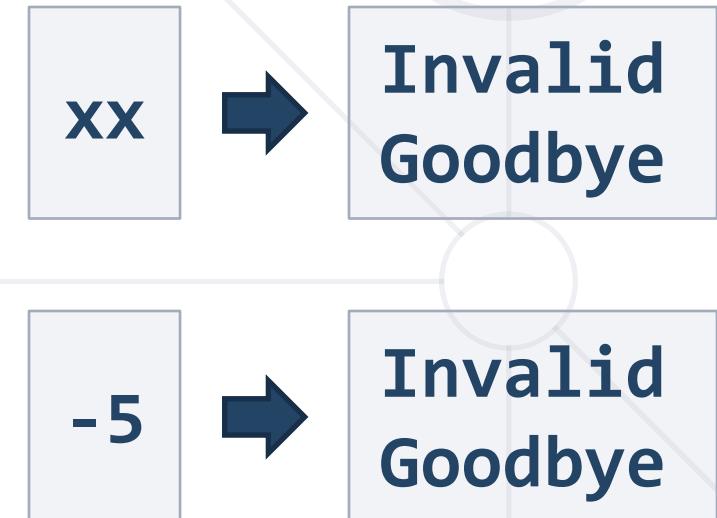
```
try {  
    Integer.parseInt(str);  
} catch (NumberFormatException ex) {  
    System.out.println("Parse failed!");  
    throw ex; // Re-throw the caught exception  
}
```

Throwing Exceptions – Example

```
public static double calcSqrt(double value) {  
    if (value < 0)  
        throw new ArithmeticException(  
            "Sqrt for negative numbers is undefined!");  
    return Math.sqrt(value);  
}  
  
public static void main(String[] args) {  
    try {  
        calcSqrt(-1);  
    } catch (ArithmeticException ex) {  
        System.err.println("Error: " + ex.getMessage());  
        ex.printStackTrace();  
    }  
}
```

Problem: Square Root

- Write a program that **reads an integer** number and calculates and prints its **square root** (with 2 digits after the decimal point)
 - If the number is **invalid** or **negative**, print "**Invalid**"
- In all cases finally print "**Goodbye**"
- Use **try-catch-finally**



Solution: Square Root

```
Scanner scanner = new Scanner(System.in);
try {
    int num = Integer.parseInt(scanner.nextLine());
    double sqrt = calcSqrt(num);
    System.out.printf("%.2f\n", sqrt);
} catch (Exception ex) {
    System.out.println("Invalid");
} finally {
    System.out.println("Goodbye");
}
```



The "throws" in Method Declarations

Forcing Invokers to Handle Certain Exceptions

Using "throws" in Method Declaration

```
static String readTextFile(String fName) throws IOException {  
    BufferedReader reader =  
        new BufferedReader(new FileReader(fName));  
    StringBuilder result = new StringBuilder();  
    try {  
        String line;  
        while ((line = reader.readLine()) != null)  
            result.append(line + System.lineSeparator());  
    } finally {  
        reader.close();  
    }  
    return result.toString();  
}
```

Invoking Method Declared with "throws"

```
public static void main(String[] args) {  
    String fileName = "./src/TextFileReader.java";  
    try {  
        String sourceCode = readTextFile(fileName);  
        System.out.println(sourceCode);  
    } catch (IOException ioex) {  
        System.err.println("Cannot read file: " + fileName);  
        ioex.printStackTrace();  
    }  
}
```

Catching **IOException** is obligatory!

Throwing from the Main Method

- The `main()` method can declare as "**throws**" all exception classes, which it refuses to handle

```
public static void main(String[] args)
    throws IOException {
    FileWriter file = new FileWriter("example.txt");
    file.write("Some text in the file");
    file.close();
}
```



Custom Exceptions

Declaring Your Own Exception Class

Creating Custom Exceptions

- Custom exceptions inherit an exception class (commonly – **Exception**)

```
public class FileParseException extends Exception {  
    private int lineNumber;  
  
    public FileParseException(String msg, int lineNumber) {  
        super(msg + " (at line " + lineNumber + ")");  
        this.lineNumber = lineNumber;  
    }  
  
    public int getLineNumber() { return lineNumber; }  
}
```

Using Custom Exceptions

- Throw your exceptions like any other:

```
throw new FileParseException(  
    "Cannot read setting", 75);
```

```
"C:\Program Files\Java\jdk-17.0.1\bin\java.exe" "-javaagent:C:\Program Files\JetBrains  
Exception in thread "main" FileParseException: Cannot read setting (at line 75)  
    at CustomExceptionsExample.main(CustomExceptionsExample.java:3)
```

- If your exception derives from **Exception** → handle it obligatory
- If it derives from **RuntimeException** → handle it optionally



Best Practices

Working with Exceptions the Right Way

Using The Catch Block

- The **catch** blocks should:
 - Begin with the exceptions **lowest** in the hierarchy
 - Continue with the more general exceptions
 - Otherwise, a **compilation error** will occur
- Each **catch** block should handle only these exceptions, which it expects
 - If a method is not competent to handle an exception, it should leave it unhandled
 - Handling all exceptions disregarding their type is a popular **bad practice** (anti-pattern)!

Common Exception Types in Java (1)

- When an application attempts to use **null** in a case where an object is required: **NullPointerException**
- A method has been passed an illegal or inappropriate argument: **IllegalArgumentException**
- An array has been accessed with an illegal index: **ArrayIndexOutOfBoundsException**
- An index is either negative or greater than the size of the string: **StringIndexOutOfBoundsException**

Common Exception Types in Java (2)

- Attempt to convert an inappropriate string to a numeric type:
NumberFormatException
- When an exceptional arithmetic condition has occurred:
ArithmaticException
- Attempt to cast an object to a subclass of which it is not an instance:
ClassCastException
- When a file or network or other input / output operation has failed:
IOException

Exceptions – Best Practices (1)

- When throwing an exception, always pass to the constructor a **good explanation message**
 - The **error message** should make obvious what the problem is
 - The exception message should explain **what causes the problem** (and give directions **how to solve it**)
 - **Good:** "Size should be integer in range [1...15]"
 - **Good:** "Invalid state. First call Initialize()"
 - **Bad:** "Unexpected error"
 - **Bad:** "Invalid argument"



Exceptions – Best Practices (2)

- Exceptions can decrease the application **performance**
 - Throw exceptions only in situations which are really exceptional and should be handled
 - Do not throw exceptions in the normal program control flow
 - The JVM could throw exceptions at any time with no way to predict them
 - E. g. **StackOverflowError** or **OutOfMemoryError**

- **Exceptions** provide a **flexible** error handling mechanism
- **Try-catch** allows exceptions to be handled
- **Unhandled exceptions** crash with an error message
- **Try-finally** ensures a given code block is always executed



Questions?



SoftUni



Software
University



SoftUni
Creative



SoftUni
Digital



SoftUni
Foundation



SoftUni
Kids



Finance
Academy

SoftUni Diamond Partners



**SUPER
HOSTING
.BG**

INDEAVR
Serving the high achievers

 **SOFTWARE
GROUP**

 **BOSCH**



**Coca-Cola HBC
Bulgaria**

 **AMBITIONED**

createX

 **DXC
TECHNOLOGY**

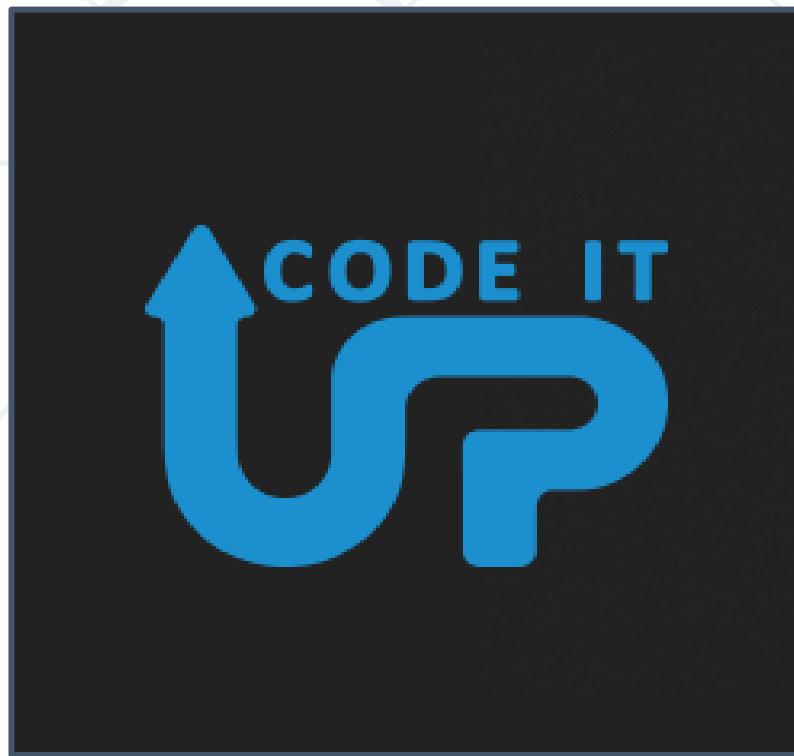
 **POKERSTARS**
POKER | CASINO | SPORTS
a Flutter International brand

 **DRAFT
KINGS**

 **Postbank**
Решения за твоето утре

 **SmartIT**

Educational Partners



Trainings @ Software University (SoftUni)



- Software University – High-Quality Education, Profession and Job for Software Developers
 - softuni.bg
- Software University Foundation
 - softuni.foundation
- Software University @ Facebook
 - facebook.com/SoftwareUniversity
- Software University Forums
 - forum.softuni.bg



Software
University

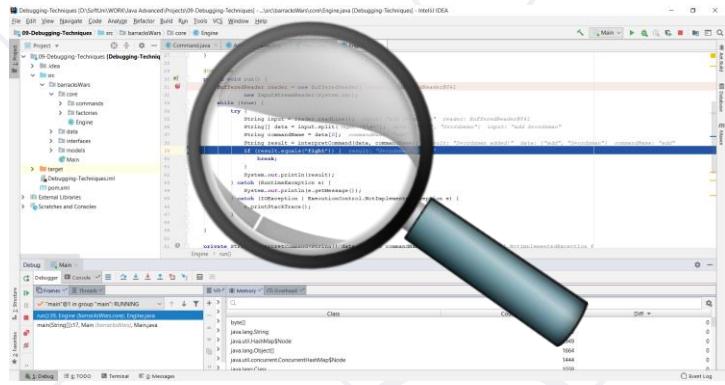


- This course (slides, examples, demos, exercises, homework, documents, videos and other assets) is **copyrighted content**
- Unauthorized copy, reproduction or use is illegal
- © SoftUni – <https://about.softuni.bg>
- © Software University – <https://softuni.bg>



Debugging

Building Rock-Solid Software



SoftUni Team
Technical Trainers



SoftUni



Software University
<https://softuni.bg>

Have a Question?



sli.do

#java-advanced

Table of Contents

1. Introduction to Debugging
2. IntelliJ IDEA Debugger
3. Breakpoints
4. Data Inspection
5. Finding a Defect





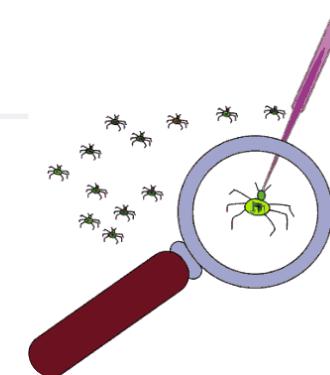
Introduction to Debugging

What is Debugging?

- The process of locating and fixing or bypassing **bugs** (errors) in computer program code
- To **debug** a program:
 - Start with a **problem**
 - Isolate the **source** of the problem
 - **Fix** it
- **Debugging tools** (called **debuggers**) help identify coding errors at various development stages

Debugging vs. Testing

- **Testing**
 - A means of initial detection of errors
- **Debugging**
 - A means of diagnosing and correcting the root causes of errors that have already been detected



Importance of Debugging

- \$60 Billion per year in economic **losses** due to software **defects**
 - E.g. the Cluster spacecraft failure was caused by a bug
- Perfect code is an illusion
 - There are factors that are out of our control
- Legacy code
 - You should be able to debug code that is written years ago
- A deeper understanding of the system as a whole

Debugging Philosophy

- Debugging can be viewed as one big **decision tree**
 - Individual nodes represent **theories**
 - Leaf nodes represent possible **root causes**
 - Traversal of tree boils down to process state **inspection**
 - Minimizing time to resolution is **key**
 - Careful traversal of the decision tree
 - Pattern recognition
 - Visualization and ease of use helps minimize time to resolution

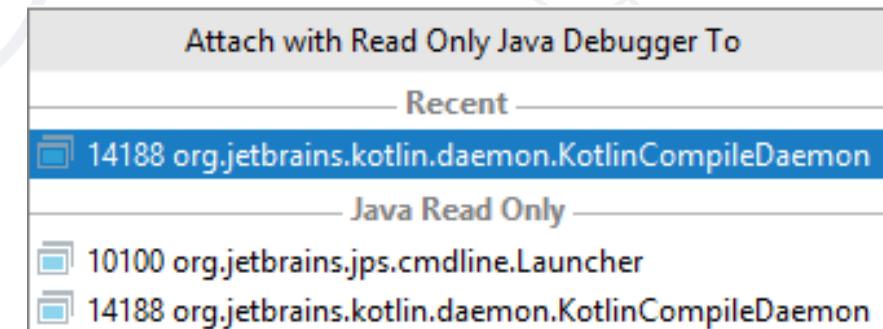


IntelliJ IDEA Debugger

- IntelliJ IDE gives us a lot of **tools** to **debug** your application
 - Adding **breakpoints**
 - Visualize the **program flow**
 - Control the **flow of execution**
 - **Data tips**
 - **Watch variables**
 - Debugging **multithreaded programs**
 - And many more...

How to Debug a Process

- Starting a process under the IntelliJ debugger
- Attaching to an already running process
 - Without a solution loaded you can still debug
 - Useful when a solution isn't readily available
 - **Ctrl + Alt + F5**



Debugging a Project

- Right click in **main** method, Debug '{class}.main()'
 - **Shift + F9** is a shortcut
- Easier access to the source code and symbols since its loaded in the solution
- Certain differences exist in comparison to debugging an already running process

Debug Windows

- Debug Windows are the means to introspect on the state of a process
- Opens a new window with the selected information in it
- Window categories
 - Frames / Threads
 - Variables
 - Watches
- Accessible from Debug window

Debugging Toolbar

- A convenient shortcut to common debugging tasks
 - **Step over** – F8
 - **Force Step Into** – through the method calls - Alt + Shift + F7
 - **Step Out** – Shift + F8
 - **Step into** – F7
 - **Continue**
 - **Break**
 - **Breakpoints**

Controlling Execution

- By default, an app will run uninterrupted (and stop on exception or breakpoint)
- Debugging is all about looking at the **state of the process**
- Controlling execution allows:
 - **Pausing** execution
 - **Resuming** execution

Options and Settings

- IntelliJ offers quite a few knobs and tweaks in the debugging experience
- Options and settings is available via **Settings/Preferences -> Build, Execution and Deployment (Ctrl + Alt + S)**:
 - Debugger -> Data Views -> **Java**
 - Compiler -> **Java Compiler**
- **Project Structure (Ctrl + Shift + Alt + S)**



Breakpoints

Breakpoints

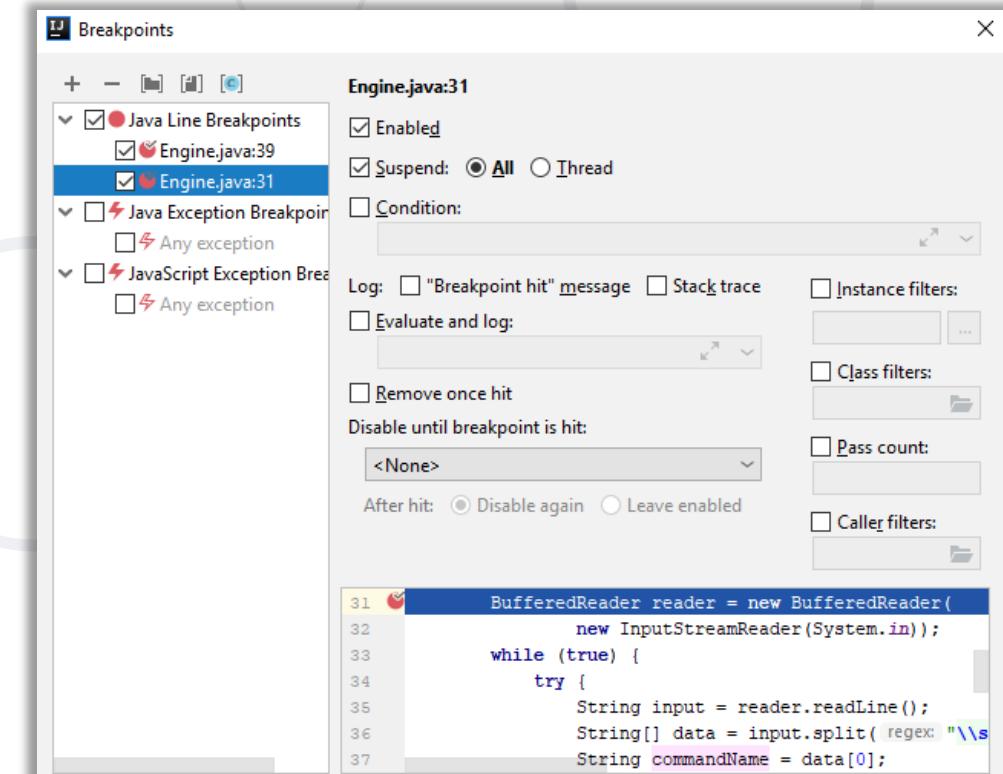
- The ability to stop execution based on certain criteria is key when debugging
 - When a function is hit
 - When data changes
 - When a specific thread hits a function
 - Much more...
- IntelliJ's debugger has a huge feature set when it comes to breakpoints

IntelliJ IDEA Breakpoints

- Stops execution at a specific instruction (line of code)
 - Can be set using:
 - **Ctrl + F8** shortcut
 - Clicking on the left most side of the source code window
 - By default, the breakpoint will hit every time execution reaches the line of the code
 - Additional capabilities: condition, hit count, value changed, when hit, filters

Managing Breakpoints

- Managed in the breakpoint window
- Adding breakpoints
- Removing or **disabling** breakpoints
- Open Breakpoints window
 - Ctrl + Shift + F8





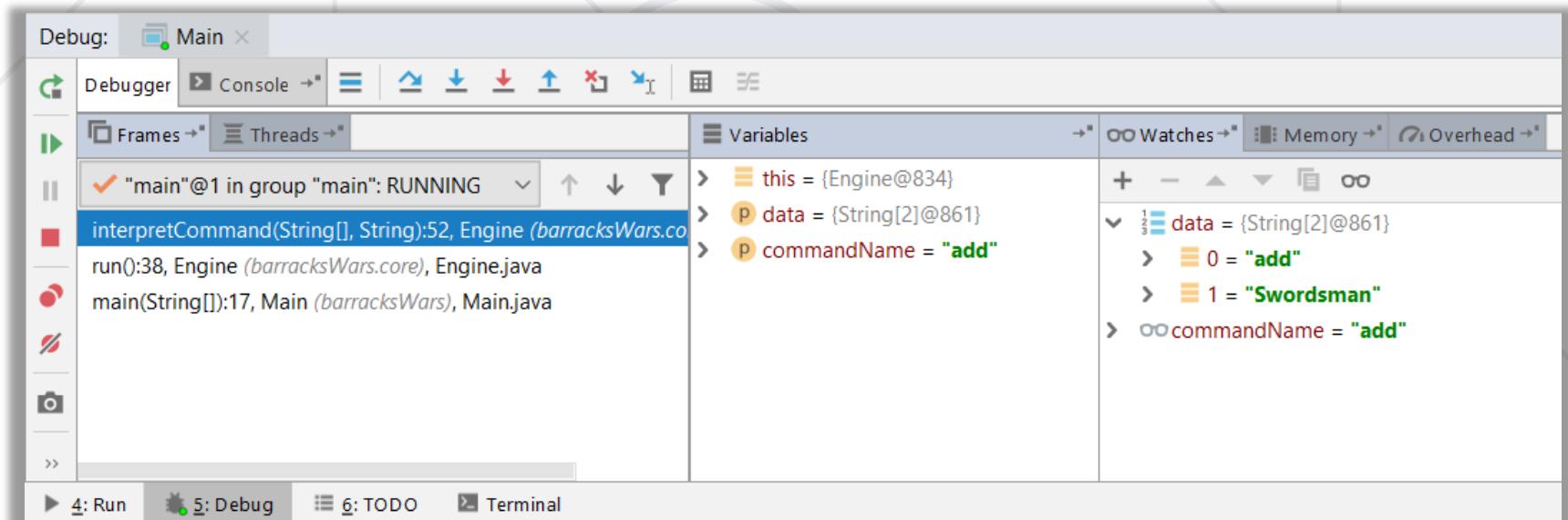
Data Inspection

Data Inspection

- Debugging is all about data inspection
 - What are the **local variables**?
 - What is in **memory**?
 - What is the **code flow**?
 - In general - What is the state of the process right now and how did it get there?
- As such, the ease of data inspection is key to the **quick resolution of problems**

IntelliJ Data Inspection

- IntelliJ offers great data inspection features
 - **Variables**
 - **Watches**
 - **Memory**
 - **Overhead**

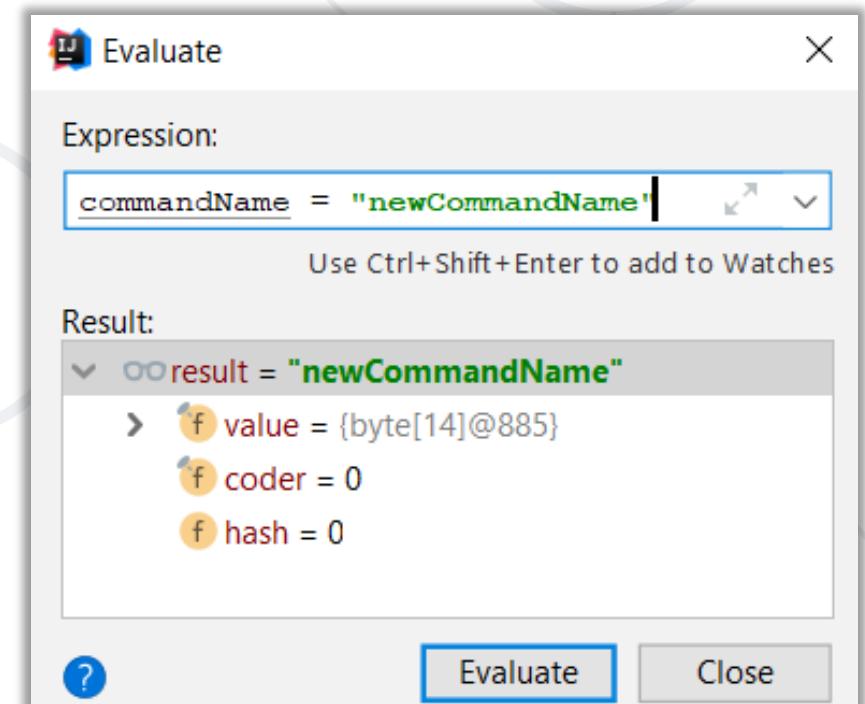


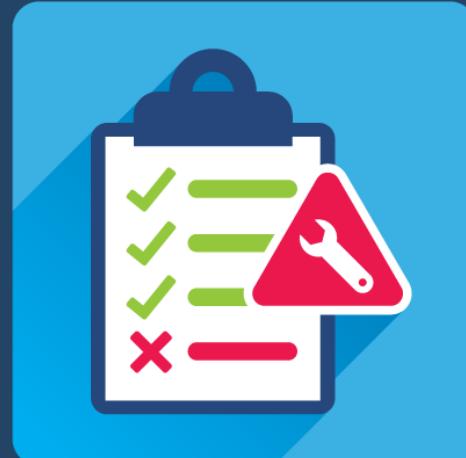
Variables and Watches Windows

- Allows you to inspect various states of your application
- Several different kinds of "predefined" watches window
- "Custom" watches windows are also possible
 - Contains only variables that you choose to add
 - Right click on the variable and select "Add to Watches"
 - Write the variable name in the Watches window

Evaluate Expression Window

- Enables to evaluate expressions and code fragments in the context of a stack frame
- Also evaluate operator expressions, lambda expressions, and anonymous classes
- Shortcut – Alt + F8





Finding a Defect

Finding a Defect

- Stabilize the error
- Locate the source of the error
 - Gather the data
 - Analyze the data and form a hypothesis
 - Determine how to prove or disprove the hypothesis
- Fix the defect
- Test the fix
- Look for similar errors



Tips for Finding Defects (1)

- Use all available data
- Refine the test cases
- Check unit tests
- Use available tools
- Reproduce the error in several different ways
- Generate more data to generate more hypotheses
- Use the results of negative tests
- Brainstorm for possible hypotheses



Tips for Finding Defects (2)

- Narrow the suspicious region of the code
- Be suspicious of classes and routines that have had defects before
- Check code that's changed recently
- Expand the suspicious region of the code
- Integrate incrementally
- Check for common defects
- Talk to someone else about the problem
- Take a break from the problem

Fixing a Defect

- Understand the problem before you fix it
- Understand the program, not just the problem
- Confirm the defect diagnosis
- Relax
- Save the original source code
- Fix the problem, not the symptom
- Make one change at a time
- Add a unit test that exposes the defect
- Look for similar defects



Psychological Considerations

- Your ego tells you that your code is good and doesn't have a defect even when you've seen that it has
- How "psychological set" contributes to debugging blindness
 - People expect a new phenomenon to resemble similar phenomena they've seen before
 - Do not expect anything to work "by default"
 - Do not be too devoted to your code – establish psychological distance

- Introduction to **Debugging**
- IntelliJ IDEA Debugger
- **Breakpoints**
- Data Inspection
 - **Variables, Watches, Frames**
- Finding a **Defect**



Questions?



SoftUni



Software
University



SoftUni
Creative



SoftUni
Digital



SoftUni
Foundation



SoftUni
Kids



Finance
Academy

SoftUni Diamond Partners



**SUPER
HOSTING
.BG**

INDEAVR
Serving the high achievers

 **SOFTWARE
GROUP**

 **BOSCH**



**Coca-Cola HBC
Bulgaria**

 **AMBITIONED**

createX

 **DXC
TECHNOLOGY**

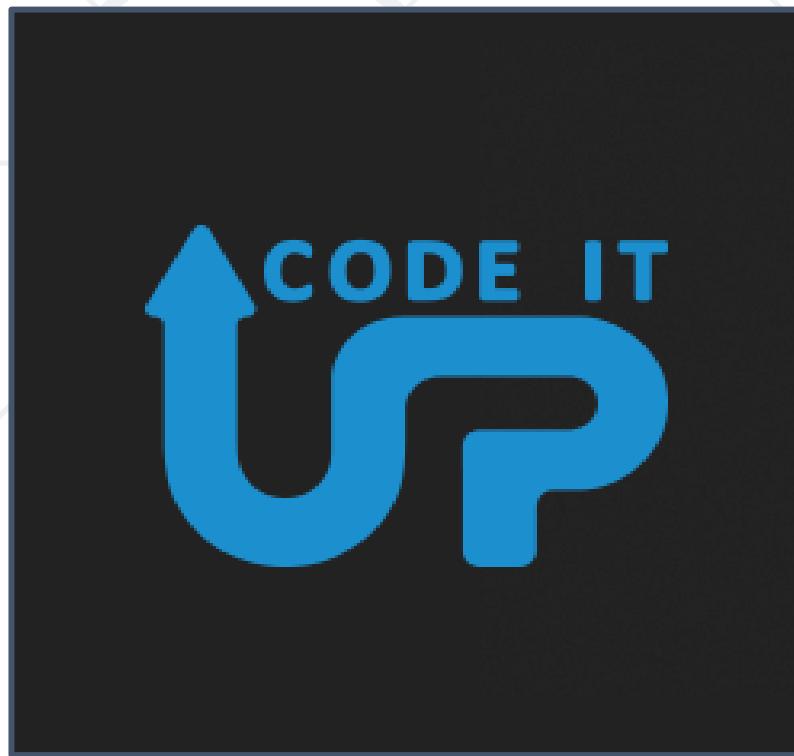
 **POKERSTARS**
POKER | CASINO | SPORTS
a Flutter International brand

 **DRAFT
KINGS**

 **Postbank**
Решения за твоето утре

 **SmartIT**

Educational Partners



Trainings @ Software University (SoftUni)



- Software University – High-Quality Education, Profession and Job for Software Developers
 - softuni.bg
- Software University Foundation
 - softuni.foundation
- Software University @ Facebook
 - facebook.com/SoftwareUniversity
- Software University Forums
 - forum.softuni.bg



Software
University

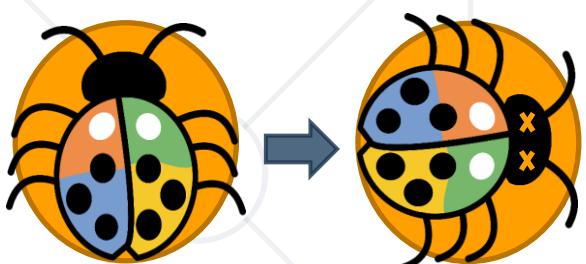


- This course (slides, examples, demos, exercises, homework, documents, videos and other assets) is **copyrighted content**
- Unauthorized copy, reproduction or use is illegal
- © SoftUni – <https://about.softuni.bg>
- © Software University – <https://softuni.bg>



Unit Testing

Building Rock-Solid Software



SoftUni Team
Technical Trainers



Software University
<https://softuni.bg>

Table of Contents

1. Seven Testing Principles
2. What Is Unit Testing?
 - Unit Testing Frameworks - JUnit
 - 3A Pattern
3. Best Practices
4. Dependency Injection
5. Mocking and Mock Objects



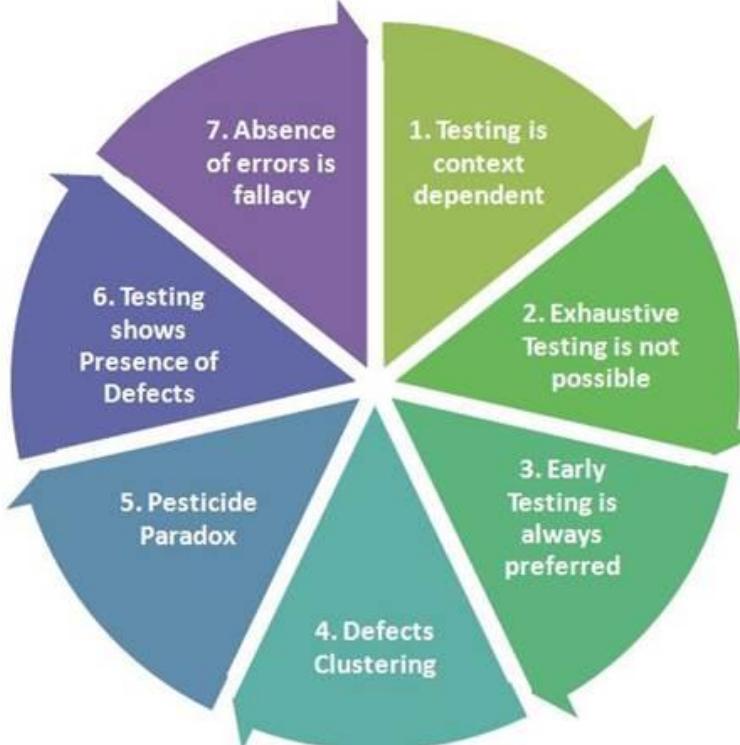
Have a Question?



sli.do

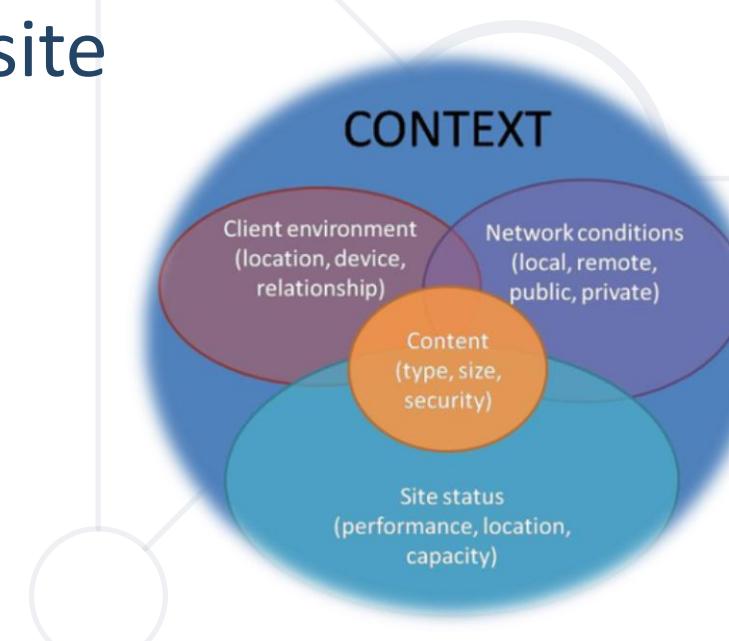
#java-advanced

Seven Testing Principles



Seven Testing Principles (1)

- Testing is context dependent
 - Testing is done differently in **different contexts**
- Example:
 - Safety-critical software is tested **differently** from an e-commerce site



Seven Testing Principles (2)

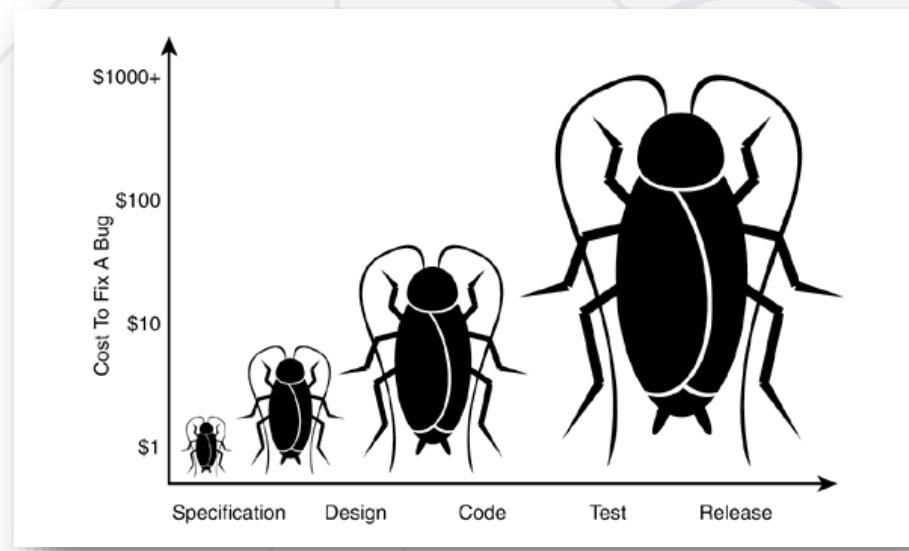
- Exhaustive testing is **impossible**
 - All combinations of inputs and preconditions are usually almost **infinite number**
 - Testing everything is not feasible
 - Except for trivial cases
 - Risk analysis and priorities should be used to focus testing efforts

Seven Testing Principles (3)

- Defect clustering
 - Testing effort shall be focused **proportionally**
 - To the expected and later observed defect density of modules
 - A **small number** of modules usually contains **most of the defects** discovered
 - Responsible for most of the operational failures

Seven Testing Principles (4)

- Early testing is **always preferred**
 - Testing activities shall be started as early as possible
 - And shall be focused on defined objectives
 - The later a bug is found – the more it costs!



Seven Testing Principles (5)

- Pesticide paradox
 - Same tests repeated **over and over again** tend to **lose their effectiveness**
 - Previously **undetected** defects remain **undiscovered**
 - New and modified test cases should be developed

Seven Testing Principles (6)

- Testing shows presence of defects
 - Testing can **show that defects are present**
 - Cannot prove that there are no defects
 - Appropriate testing **reduces** the probability for defects

Seven Testing Principles (7)

- Absence-of-errors fallacy
 - **Finding** and **fixing** defects itself does not help in these cases:
 - The system built is unusable
 - Does not fulfill the users needs and expectations



What is Unit Testing

Manual Testing (1)

- Not **structured**
- Not **repeatable**
- Can't **cover** all of the code
- Not as **easy** as it should be



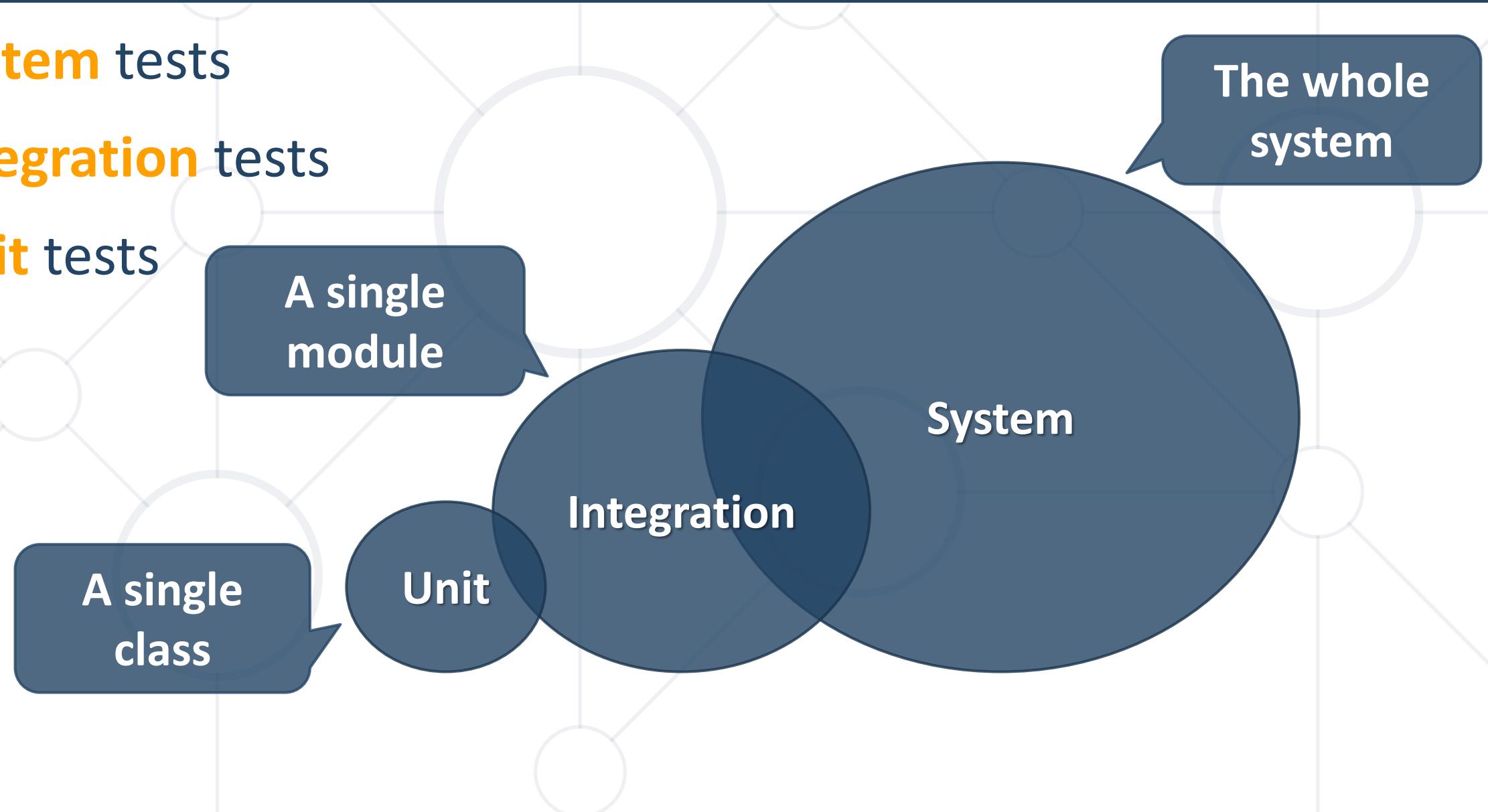
```
void testSum() {  
    if (this.sum(1, 2) != 3) {  
        throw new Exception("1 + 2 != 3");  
    }  
}
```

Manual Testing (2)

- We need a **structured approach** that:
 - Allows **refactoring**
 - Reduces the **cost of change**
 - **Decreases** the number of **defects** in the code
- Bonus:
 - Improves **design**

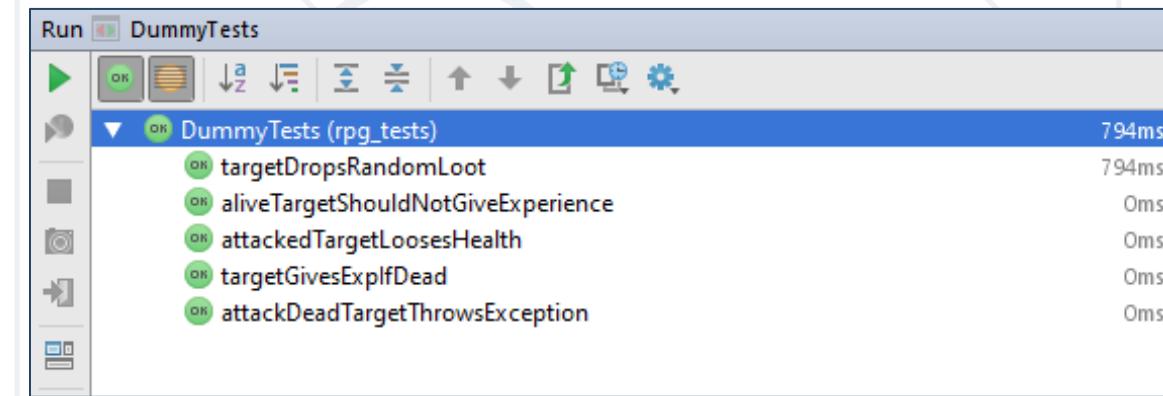
Automated Testing

- **System** tests
- **Integration** tests
- **Unit** tests



Junit (1)

- The first popular unit testing **framework**
- Most popular for Java development
- Based on Java, written by Kent Beck & Co.



Junit (2)

- Maven Repository – Junit 4.12
- Copy JUnit repository and paste in **pom.xml**

```
<project ...>
...
<dependencies>
    <dependency>
        <groupId>junit</groupId>
        <artifactId>junit</artifactId>
        <version>4.12</version>
        <scope>test</scope>
    </dependency>
</dependencies>
</project>
```

Junit – Writing Tests

- Create new package (e.g. **tests**)
- Create a class for test methods (e.g. **BankAccountTests**)
- Create a **public void** method annotated with **@Test**

```
@Test  
public void depositShouldAddMoney() {  
    /* magic */  
}
```

3A Pattern

- **Arrange** - Preconditions
- **Act** - Test a **single behavior**
- **Assert** - Postconditions

```
@Test  
public void depositShouldAddMoney() {  
    BankAccount account = new BankAccount();  
    account.deposit(50);  
    Assert.assertTrue(account.getBalance() == 50)  
}
```

Each test should test
a **single behavior!**

Exceptions

- Sometimes **throwing** an exception is the **expected behavior**

```
@Test(expected = IllegalArgumentException.class)  
public void depositNegativeShouldNotAddMoney() {  
    BankAccount account = new BankAccount();  
    account.deposit(-50);  
}
```

Act

Arrange

Assert

Problem: Test Axe

- Create a **Maven** project
- Add provided classes (**Axe**, **Dummy**, **Hero**) to project
- In **test/java** folder, create a package **rpg_tests**
- Create a class **AxeTests**
- Create the following tests:
 - Test if weapon **loses durability** after attack
 - Test attacking with a **broken weapon**



Solution: Test Axe (1)

```
@Test  
public void weaponLosesDurabilityAfterAttack() {  
    // Arrange  
    Axe axe = new Axe(10, 10);  
    Dummy dummy = new Dummy(10, 10);  
    // Act  
    axe.attack(dummy);  
    // Assert  
    Assert.assertTrue(axe.getDurabilityPoints() == 9);  
}
```

Solution: Test Axe (2)

```
@Test(expected = IllegalStateException.class) // Assert  
public void brokenWeaponCantAttack() {  
    // Arrange  
    Axe axe = new Axe(10, 1);  
    Dummy dummy = new Dummy(10, 10);  
    // Act  
    axe.attack(dummy);  
    axe.attack(dummy);  
}
```

Problem: Test Dummy

- Create a class **DummyTests**
- Create the following tests
 - Dummy **loses health** if attacked
 - Dead Dummy **throws exception** if attacked
 - Dead Dummy **can give XP**
 - Alive Dummy **can't give XP**



Solution: Test Dummy

```
@Test  
public void attackedTargetLoosesHealth() {  
    // Arrange  
    Dummy dummy = new Dummy(10, 10);  
    // Act  
    dummy.takeAttack(5);  
    // Assert  
    Assert.assertTrue(dummy.getHealth() == 5);  
}  
// TODO: Write the rest of the tests
```

There is a better
solution...



Unit Testing Best Practices

Assertions

- **assertTrue() vs assertEquals()**

- **assertTrue()**

```
Assert.assertTrue(account.getBalance() == 50);
```

```
+java.lang.AssertionError <3 internal calls>
```

- **assertEquals(expected, actual)**

```
Assert.assertEquals(50, account.getBalance());
```

Better description when
expecting value

```
java.lang.AssertionError:  
Expected :50  
Actual   :35  
<Click to see difference>
```

Assertion Messages

- Assertions can **show messages**
 - Helps with **diagnostics**
- **Hamcrest** is useful tool for test diagnostics

`Assert.assertEquals(`

"Wrong balance", 50, account.getBalance()));

Helps finding
the problem

```
java.lang.AssertionError: Wrong balance
Expected :50
Actual    :35
<Click to see difference>
```

Magic Numbers

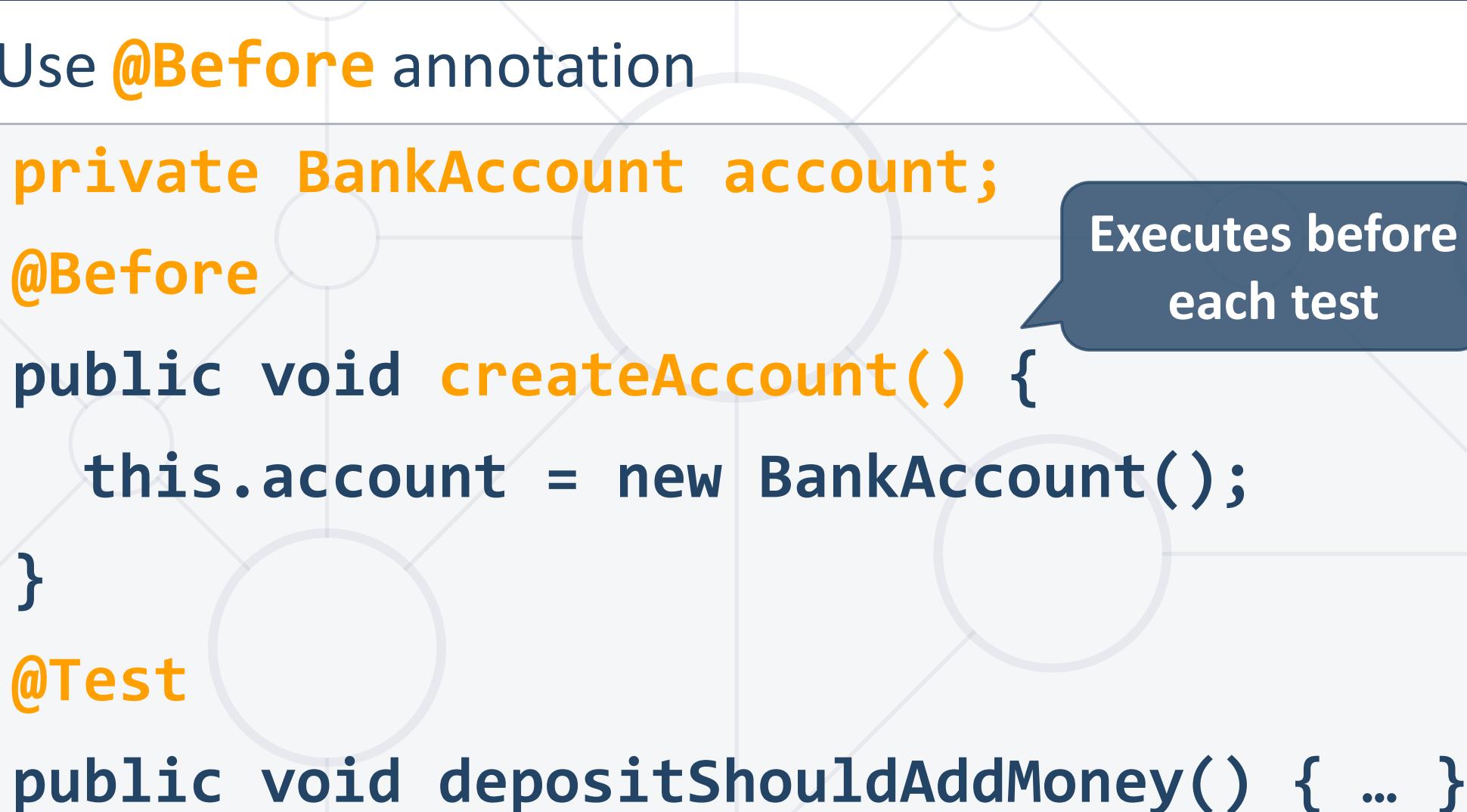
- Avoid using magic numbers (use **constants** instead)

```
private static final int AMOUNT = 50;  
  
@Test  
  
public void depositShouldAddMoney() {  
    BankAccount account = new BankAccount();  
    account.deposit(AMOUNT);  
    Assert.assertEquals("Wrong balance",  
                      AMOUNT, account.getBalance());  
}
```

@Before

- Use **@Before** annotation

```
private BankAccount account;  
@Before  
public void createAccount() {  
    this.account = new BankAccount();  
}  
@Test  
public void depositShouldAddMoney() { ... }
```



The diagram consists of several light gray circles connected by thin gray lines, forming a network-like structure. A blue speech bubble is positioned to the right of the code snippet. It contains the text "Executes before each test". Arrows point from the word "@Before" in the code to the first circle, and from the first circle to the start of the "createAccount" method. Another arrow points from the word "@Test" in the code to the start of the "depositShouldAddMoney" method.

Naming Test Methods

- Test names
 - Should use **business domain terminology**
 - Should be **descriptive** and **readable**

```
incrementNumber() {}
```

```
test1() {}
```

```
testTransfer() {}
```



```
depositAddsMoneyToBalance() {}
```

```
depositNegativeShouldNotAddMoney() {}
```

```
transferSubtractsFromSourceAddsToDestAccount() {}
```



Problem: Refactor Tests

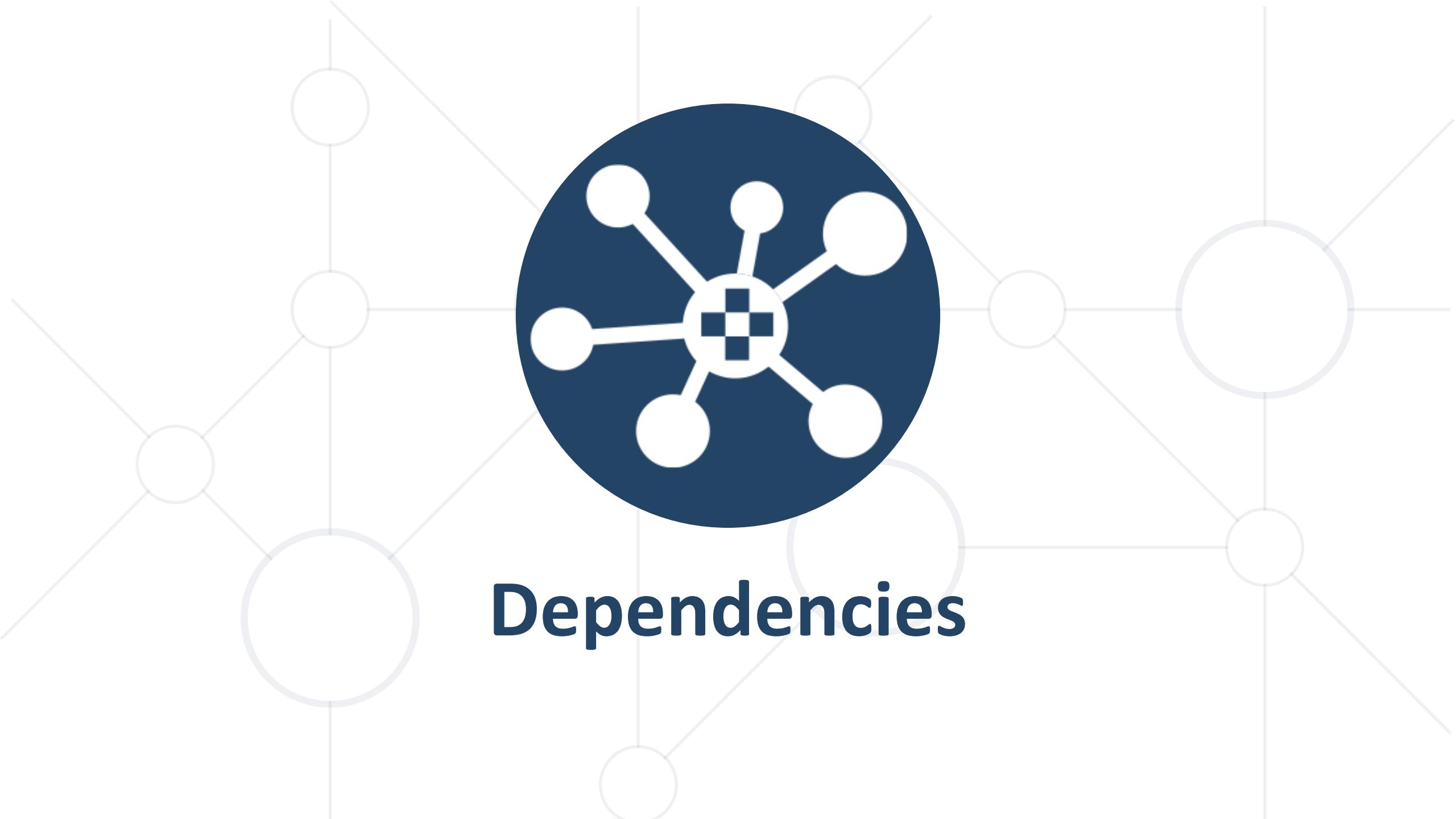
- Refactor the tests for **Axe** and **Dummy** classes
- Make sure that:
 - **Names** of test methods are **descriptive**
 - You use **appropriate assertions** (assert equals vs assert true)
 - You use **assertion messages**
 - There are **no magic numbers**
 - There is no **code duplication** (Don't Repeat Yourself)

Solution: Refactor Tests (1)

```
private static final int AXE_ATTACK = 10;  
private static final int AXE_DURABILITY = 1;  
private static final int DUMMY_HEALTH = 20;  
private static final int DUMMY_XP = 10;  
  
private Axe axe;  
private Dummy dummy;  
  
@Before  
public void initializeTestObjects() {  
    this.axe = new Axe(AXE_ATTACK, AXE_DURABILITY);  
    this.dummy = new Dummy(DUMMY_HEALTH, DUMMY_XP); }
```

Solution: Refactor Tests (2)

```
@Test  
public void weaponLosesDurabilityAfterAttack() {  
    this.axe.attack(this.dummy);  
    Assert.assertEquals("Wrong durability",  
        AXE_DURABILITY - 1,  
        axe.getDurabilityPoints()); }  
  
@Test(expected = IllegalStateException.class)  
public void brokenWeaponCantAttack() {  
    this.axe.attack(this.dummy);  
    this.axe.attack(this.dummy); }
```



Dependencies

Coupling and Testing (1)

- Consider testing the following code:

- We want to test a **single behavior**

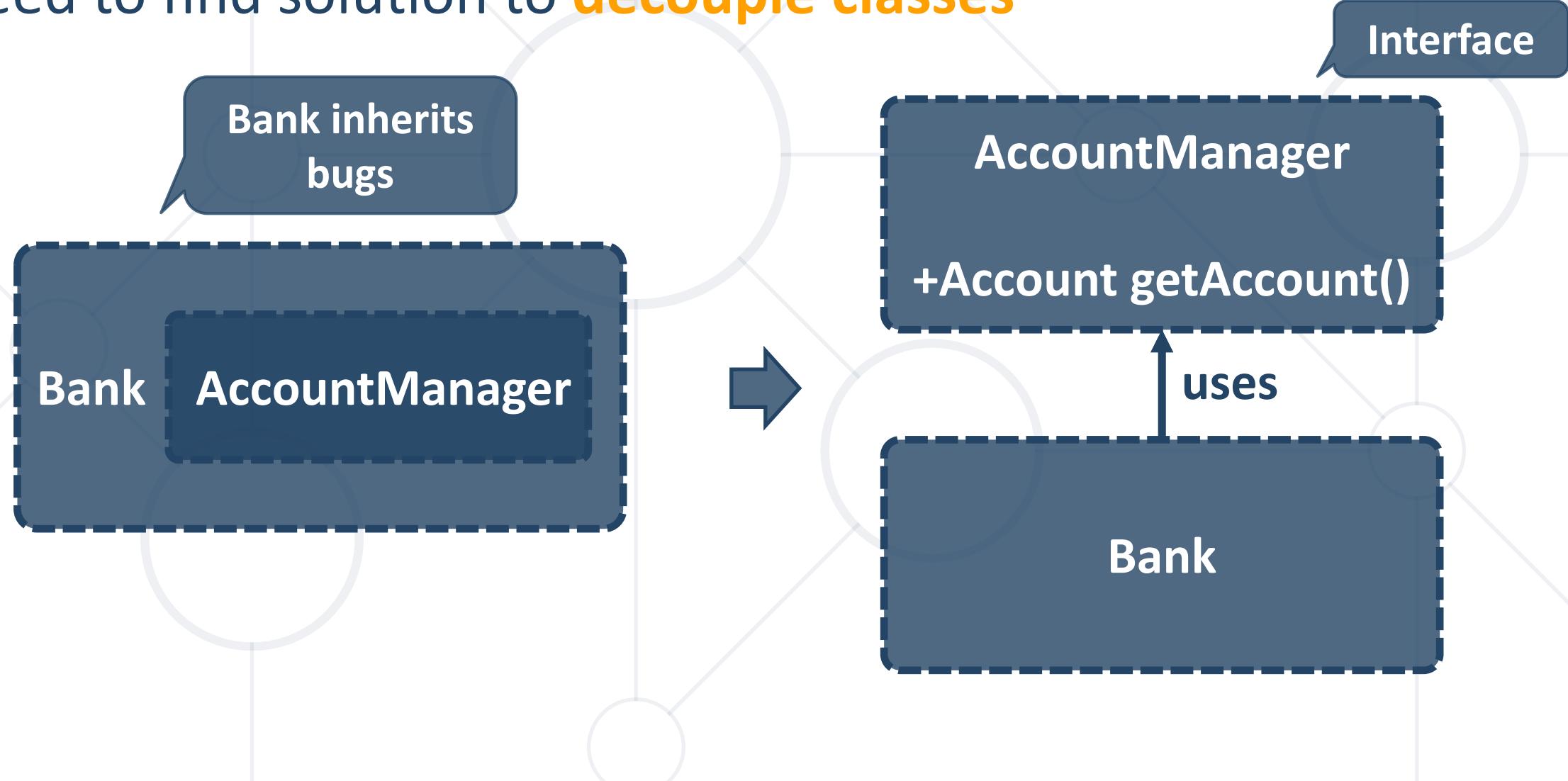
```
public class Bank {  
    private AccountManager accountManager;  
    public Bank() {  
        this.accountManager = new AccountManager();  
    }  
    public AccountInfo getInfo(String id) { ... }  
}
```

Concrete
Implementation

Bank depends on
AccountManager

Coupling and Testing (2)

- Need to find solution to **decouple classes**



Dependency Injection

- Decouples classes and **makes code testable**



```
interface AccountManager {  
    Account getAccount();  
}  
  
public class Bank {  
    private AccountManager accountManager;  
  
    public Bank(AccountManager accountManager) {  
        this.accountManager = accountManager;  
    }  
}
```

Using Interface

Independent from Implementation

Injecting dependencies

Goal: Isolating Test Behavior

- In other words, to **fixate** all **moving parts**

```
@Test
```

```
public void testGetInfoById() {  
    // Arrange  
    AccountManager manager = new AccountManager() {  
        public Account getAccount(String id) { ... }  
    };  
    Bank bank = new Bank(manager);  
    AccountInfo info = bank.getInfo(ID);  
    // Assert... }
```

Anonymous class

Fake interface implementation
with fixed behavior

Problem: Fake Axe and Dummy

- Test if hero **gains XP** when **target dies**
- To do this, first:
 - Make **Hero** class **testable** (use **Dependency Injection**)
 - Introduce **Interfaces** for Axe and Dummy
 - Interface Weapon
 - Interface Target
 - Create test using a **fake Weapon** and **fake Dummy**

Solution: Fake Axe and Dummy (1)

```
public interface Target {  
    void takeAttack(int attackPoints);  
    int getHealth();  
    int giveExperience();  
    boolean isDead();  
}
```

```
public interface Weapon {  
    void attack(Target target);  
    int getAttackPoints();  
    int getDurabilityPoints(); }
```

Solution: Fake Axe and Dummy (2)

```
// Hero: Dependency Injection through constructor  
  
public Hero(String name, Weapon weapon) {  
    this.name = name;          /* Hero: Dependency Injection */  
    this.experience = 0;        /* through constructor */  
    this.weapon = weapon; }
```

```
public class Axe implements Weapon {  
    public void attack(Target target) { ... }  
}
```

```
// Dummy: implement Target interface  
public class Dummy implements Target { }
```

Solution: Fake Axe and Dummy (3)

```
@Test  
public void heroGainsExperienceAfterAttackIfTargetDies() {  
    Target fakeTarget = new Target() {  
        public void takeAttack(int attackPoints) { }  
        public int getHealth() { return 0; }  
        public int giveExperience() { return TARGET_XP; }  
        public boolean isDead() { return true; }  
    };  
    // Continues on next slide...
```

Solution: Fake Axe and Dummy (4)

```
// ...  
  
Weapon fakeWeapon = new Weapon() {  
    public void attack(Target target) {}  
    public int getAttackPoints() { return WEAPON_ATTACK; }  
    public int getDurabilityPoints() { return 0; }  
};  
  
Hero hero = new Hero(HERO_NAME, fakeWeapon);  
hero.attack(fakeTarget);  
// Assert...  
}
```

Fake Implementations

- Not **readable**, cumbersome and boilerplate

```
@Test  
public void testRequiresFakeImplementationOfBigInterface() {  
    // Arrange  
    Database db = new BankDatabase() {  
        // Too many methods...  
    };  
    AccountManager manager = new AccountManager(db);  
    // Act & Assert...  
}
```

Not suitable for
big interfaces

Mocking

- Mock objects **simulate behavior** of real objects
 - **Supplies data** exclusively for the test - e.g. **network** data, **random** data, **big** data (database), etc.

```
@Test  
public void testAlarmClockShouldRingInTheMorning() {  
    Time time = new Time();  
  
    AlarmClock clock = new AlarmClock(time);  
  
    if (time.isMorning()) { Test will pass only in the morning!  
        Assert.assertTrue(clock.isRing());  
    } }
```



- [Mockito Web Site](#) - [Mockito 3.0.0 dependency](#)
- Copy dependency in **pom.xml**

```
<dependency>
    <groupId>org.mockito</groupId>
    <artifactId>mockito-core</artifactId>
    <version>3.0.0</version>
    <scope>test</scope>
</dependency>
```

- Framework for mocking objects

```
@Test  
public void testAlarmClockShouldRingInTheMorning() {  
    Time mockedTime = Mockito.mock(Time.class);  
    Mockito.when(mockedTime.isMorning()).thenReturn(true);  
    AlarmClock clock = new AlarmClock(mockedTime);  
    if (mockedTime.isMorning()) {  
        Assert.assertTrue(clock.isRinging());  
    }  
}
```

Always true



Problem: Mocking

- Include **Mockito** in the project dependencies
- Mock fakes from previous problem
- Implement Hero **Inventory**, holding unequipped weapons
 - method - **Iterable<Weapon> getInventory()**
- Implement Target giving random weapon upon death
 - field - **private List<Weapon> possibleLoot**
- Test Hero killing a target getting loot in his inventory
- Test Target drops random loot

Solution: Mocking (1)

```
@Test  
public void attackGainsExperienceIfTargetIsDead() {  
    Weapon weaponMock = Mockito.mock(Weapon.class);  
    Target targetMock = Mockito.mock(Target.class);  
    Mockito.when(targetMock.isDead()).thenReturn(true);  
    Mockito.when(targetMock.giveExperience()).thenReturn(TARGET_XP);  
    Hero hero = new Hero(HERO_NAME, weaponMock);  
    hero.attack(targetMock);  
  
    Assert.assertEquals("Wrong experience", TARGET_XP,  
        hero.getExperience());  
}
```

Solution: Mocking (2)

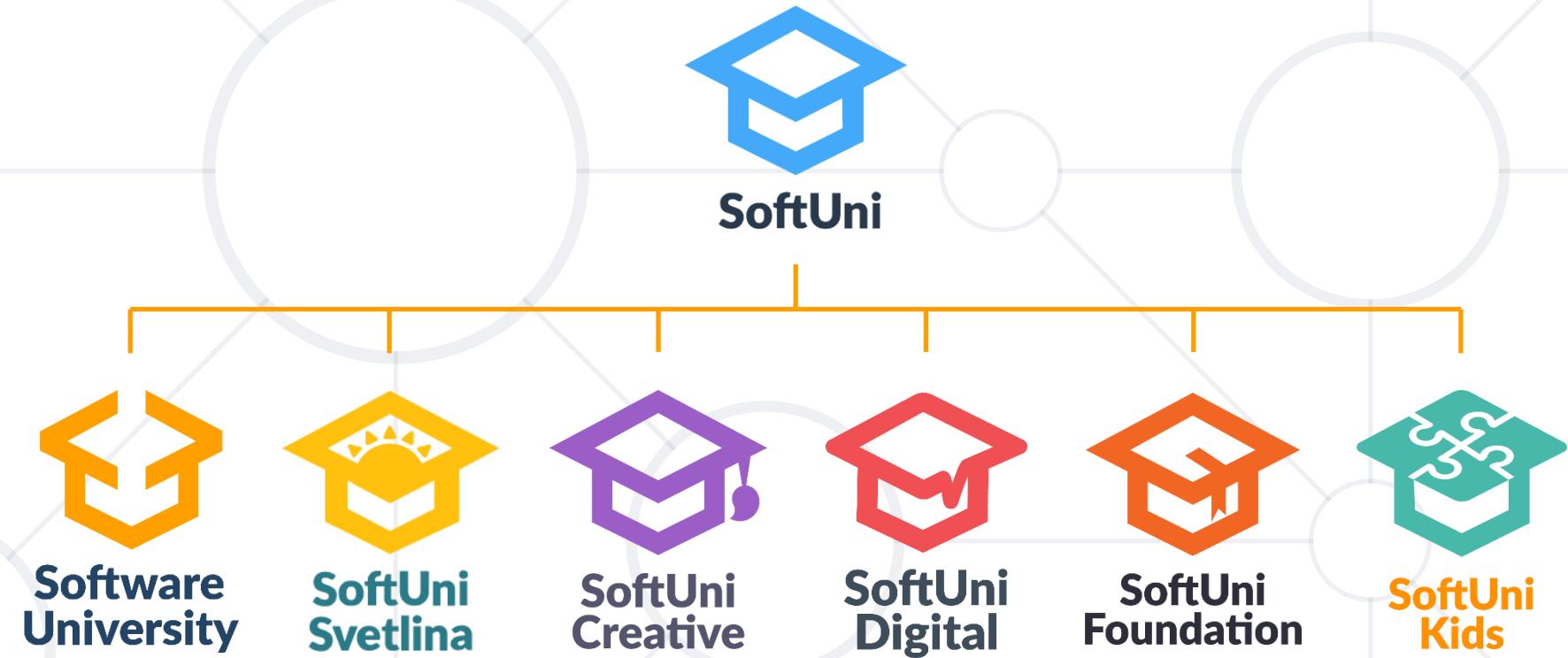
- Create **RandomProvider** Interface
- Hero method
 - **attack(Target target, RandomProvider rnd)**
- Target method
 - **dropLoot(RandomProvider rnd)**
- Mock weapon, target and random provider for test

Summary

- **Unit Testing** helps us build **solid code**
- **Structure** your unit tests – **3A Pattern**
- Use **descriptive names** for your tests
- Use different **assertions** depending on the situation
- **Dependency Injection**
 - makes your classes **testable**
 - **Looses coupling** and **improves design**
- **Mock objects** to **isolate tested behavior**



Questions?



SoftUni Diamond Partners



SCHWARZ



MOTION SOFTWARE

Educational Partners



VIRTUAL RACING SCHOOL



Trainings @ Software University (SoftUni)



- Software University – High-Quality Education, Profession and Job for Software Developers
 - softuni.bg
- Software University Foundation
 - softuni.foundation
- Software University @ Facebook
 - facebook.com/SoftwareUniversity
- Software University Forums
 - forum.softuni.bg



Software
University



- 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>



Test-Driven Development

Learn the "Test First" Approach to Coding



SoftUni Team

Technical Trainers



SoftUni



Software University
<https://softuni.bg>

Table of Contents

1. Code and Test
2. Test-Driven Development
3. Reasons to Use Test-Driven Development



Have a Question?



sli.do

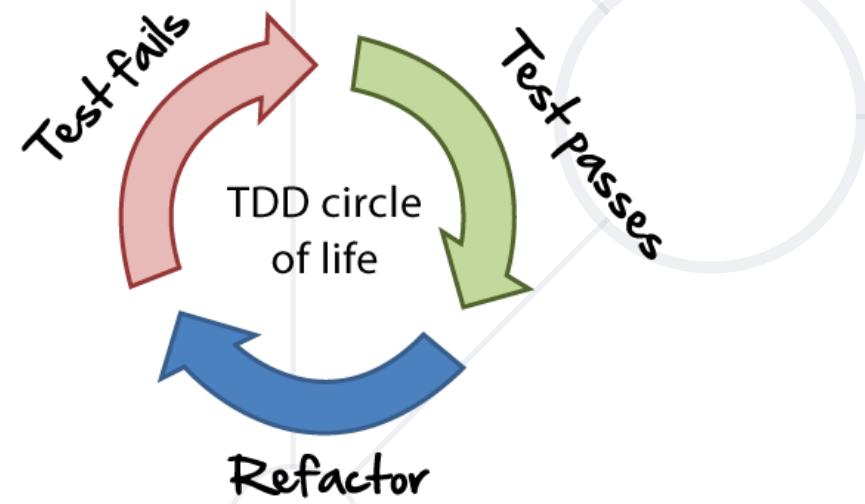
#java-advanced



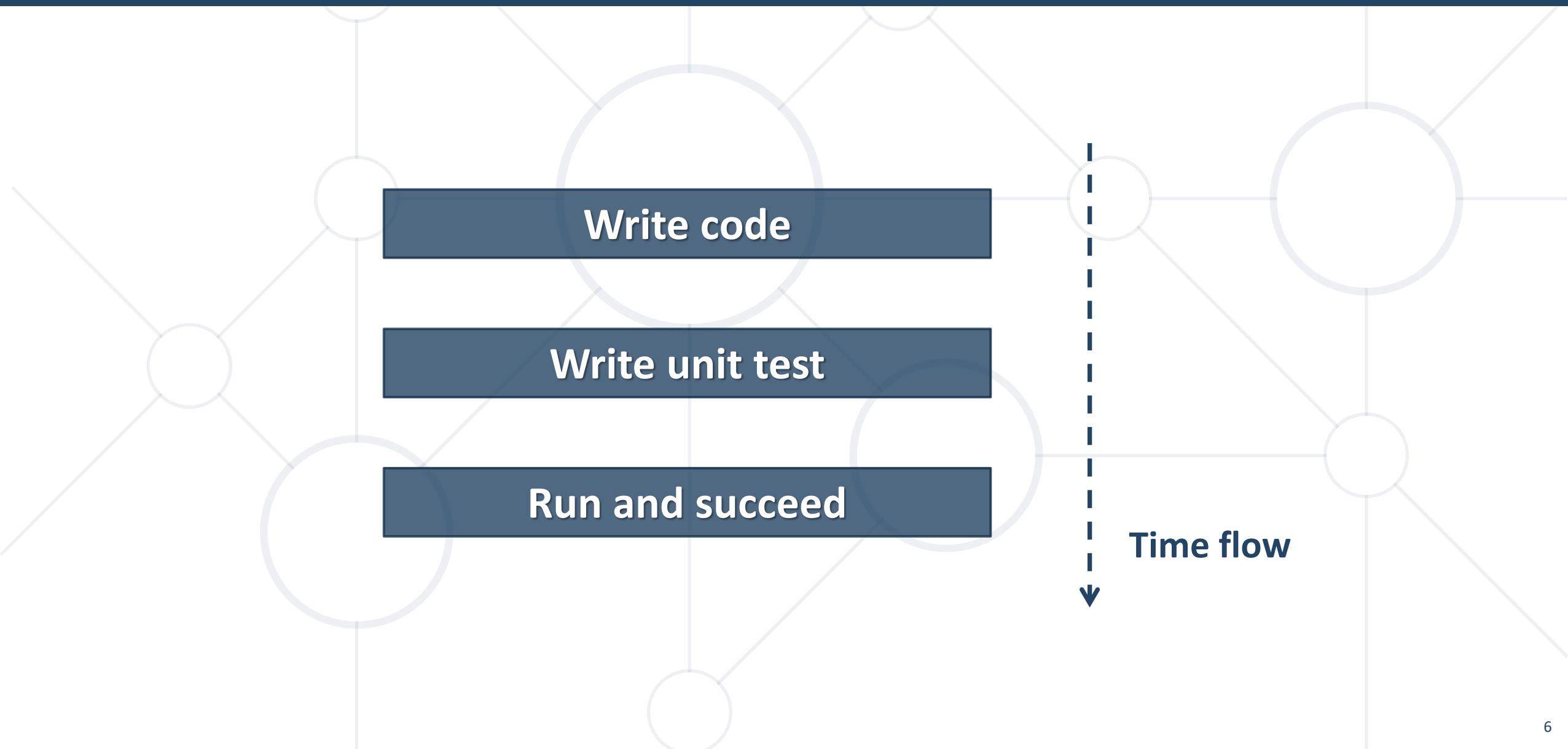
Test-Driven Development

Unit Testing Approaches

- "Code First" (code and test) approach
 - Classical approach
- "Test First" approach
 - Test-driven development (TDD)



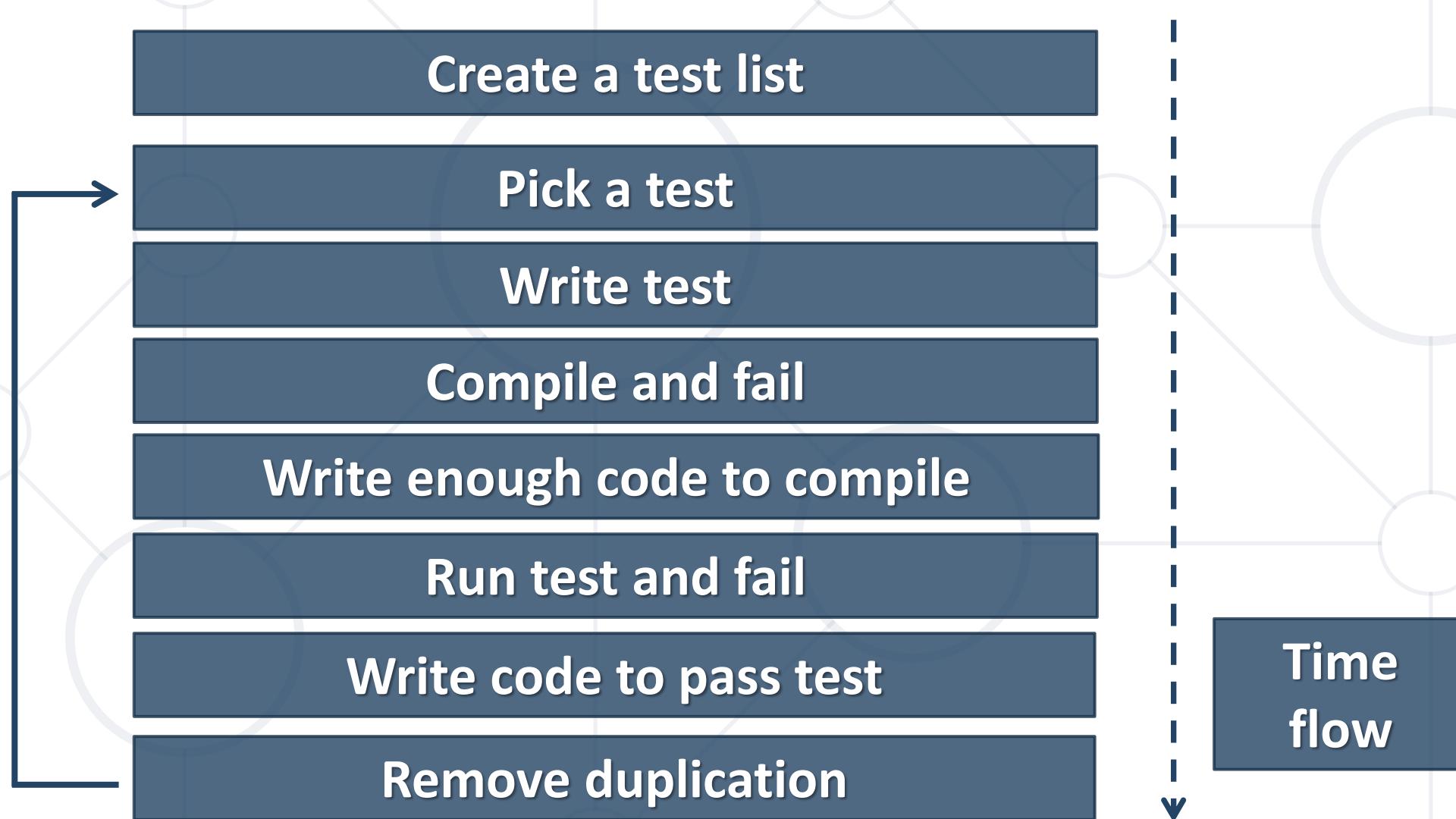
The Code and Test Approach



The Test-Driven Development Approach



Test-Driven Development (TDD)



Why TDD?

- TDD helps find design issues early
 - Avoids reworking
 - Writing code to satisfy a test is a focused activity
 - Less chance of error
 - Tests will be more comprehensive than if they are written after the code

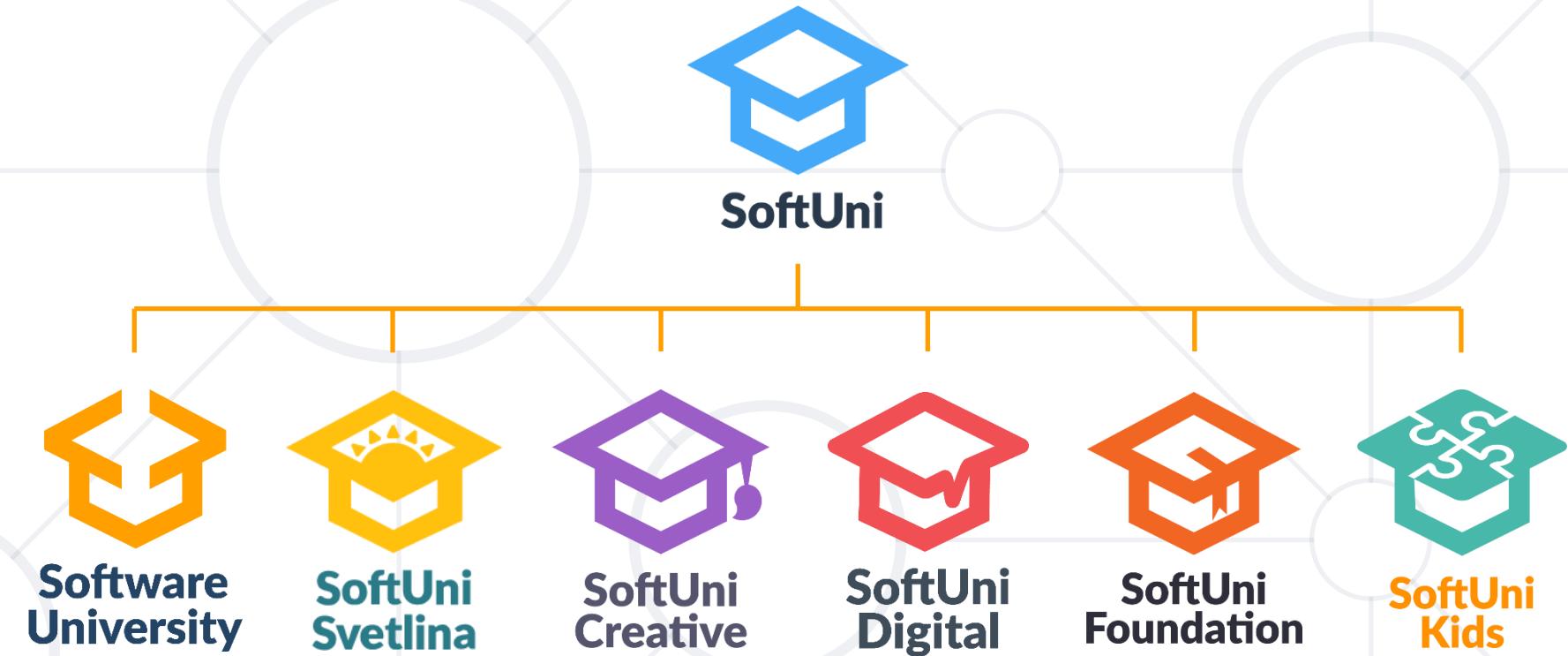


Summary

- Code and Test
 - Write code, then test it
- Test-Driven Development
 - Write tests first
- Reasons to use TDD
 - Prevent some application design flaws
 - Manage complexity more easily



Questions?



SoftUni Diamond Partners



SCHWARZ



MOTION SOFTWARE

Educational Partners



VIRTUAL RACING SCHOOL



Trainings @ Software University (SoftUni)



- Software University – High-Quality Education, Profession and Job for Software Developers
 - softuni.bg
- Software University Foundation
 - softuni.foundation
- Software University @ Facebook
 - facebook.com/SoftwareUniversity
- Software University Forums
 - forum.softuni.bg



Software
University



- 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>



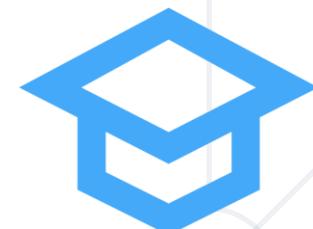
Design Patterns



SoftUni Team

Technical Trainers

 Software
University



SoftUni



Software University

<https://softuni.bg>

Have a Question?



sli.do

#java-advanced

Table of Contents

1. Definition of Design Patterns
2. Benefits and Drawbacks
3. Types of Design Patterns
 - Creational
 - Structural
 - Behavioral





Design Patterns

What Are Design Patterns?

- General and reusable solutions to common problems in software design
- A template for solving given problems
- Add additional layers of abstraction in order to reach flexibility



What Do Design Patterns Solve?

- Patterns solve **software structural problems** like:
 - Abstraction
 - Encapsulation
 - Separation of concerns
 - Coupling and cohesion
 - Separation of interface and implementation
 - Divide and conquer



Elements of a Design Pattern

- Pattern name - Increases **vocabulary** of designers
- Problem - **Intent**, context, and when to apply
- Solution - **Abstract** code
- Consequences - **Results** and trade-offs





Why Design Patterns?

Benefits

- Names form a common vocabulary
- Enable large-scale **reuse** of software architectures
- Help improve developer **communication**
- Help ease the **transition** to Object-Oriented technology
- Can **speed-up** the development



Drawbacks

- Do not lead to a direct code reuse
- Deceptively simple
- Developers may suffer from **pattern overload** and **overdesign**
- Validated by **experience** and discussion, not by automated testing
- Should be used only of **understood well**





Types of Design Patterns

Main Types

- Creational patterns
 - Deal with **initialization and configuration** of classes and objects
- Structural patterns
 - Describe ways to **assemble** objects to implement **new functionality**
 - **Composition** of classes and objects
- Behavioral patterns
 - Deal with dynamic **interactions** among societies of classes
 - Distribute **responsibility**



Creational Patterns

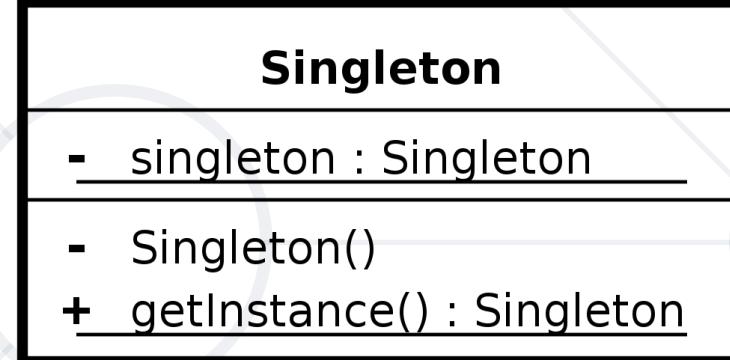
Purposes

- Deal with **object creation** mechanisms
- Trying to create objects in a **manner suitable to the situation**
- Two main ideas
 - **Encapsulating** knowledge about which classes the system uses
 - **Hiding** how instances of these classes are created



Singleton Pattern

- The most often used creational design pattern
- A Singleton class is supposed to have **only one instance**
- It is **not a global variable**
- Possible problems
 - Lazy loading
 - Thread-safe



Double-Check Singleton Example

```
public static Singleton getInstanceDC() {  
    if(instance == null) { // Single checked  
        synchronized (Singleton.class) {  
            if (instance == null) { // Double checked  
                instance = new Singleton();  
            }  
        }  
    }  
    return instance;  
}
```

Builder Pattern

- **Separates** the construction of a complex object from its representation
 - The same construction process can create different representations
- Provides control over steps of the construction process

Example: Computer Class

```
public class Computer {  
    private String RAM;  
    private boolean isGraphicsCardEnabled;  
  
    public String getRAM() { return RAM; }  
    public boolean isGraphicsCardEnabled() {  
        return isGraphicsCardEnabled;  
    }  
    public Computer(String ram, boolean isGraphicsCardEnabled) {  
        this.RAM = ram;  
        this.isGraphicsCardEnabled = isGraphicsCardEnabled;  
    }  
}
```

Example: ComputerBuilder Class

```
public class ComputerBuilder {  
    private String RAM;  
    private boolean isGraphicsCardEnabled;  
  
    public ComputerBuilder(String ram){ this.RAM = ram; }  
    public ComputerBuilder setGraphicsCardEnabled(  
        boolean isGraphicsCardEnabled) {  
        this.isGraphicsCardEnabled = isGraphicsCardEnabled;  
        return this; }  
  
    public Computer build(){  
        return new Computer(this.RAM, this.isGraphicsCardEnabled);  
    }  
}
```



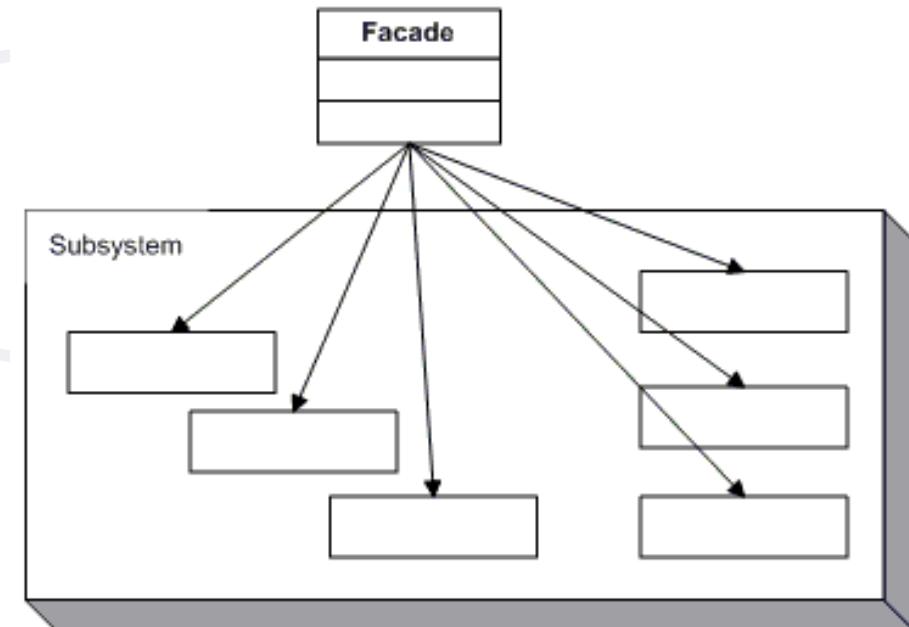
Structural Patterns

Purposes

- 
- Describe ways to assemble **objects** to implement a **new functionality**
 - Ease the design by identifying a simple way to realize the **relationship** between entities
 - All about Class and Object composition
 - **Inheritance** to compose interfaces
 - Ways to compose objects to obtain **new functionality**

Façade Pattern

- Provides a **unified interface** to a set of interfaces in a subsystem
- Defines a **higher-level interface** that makes the subsystem easier to use



Façade Example (1)

```
public interface Shape {  
    void draw();  
}  
  
public class Rectance implements Shape {  
    @Override  
    public void draw() {  
        System.out.println("Rectangle::draw()");  
    }  
}
```

Façade Example (2)

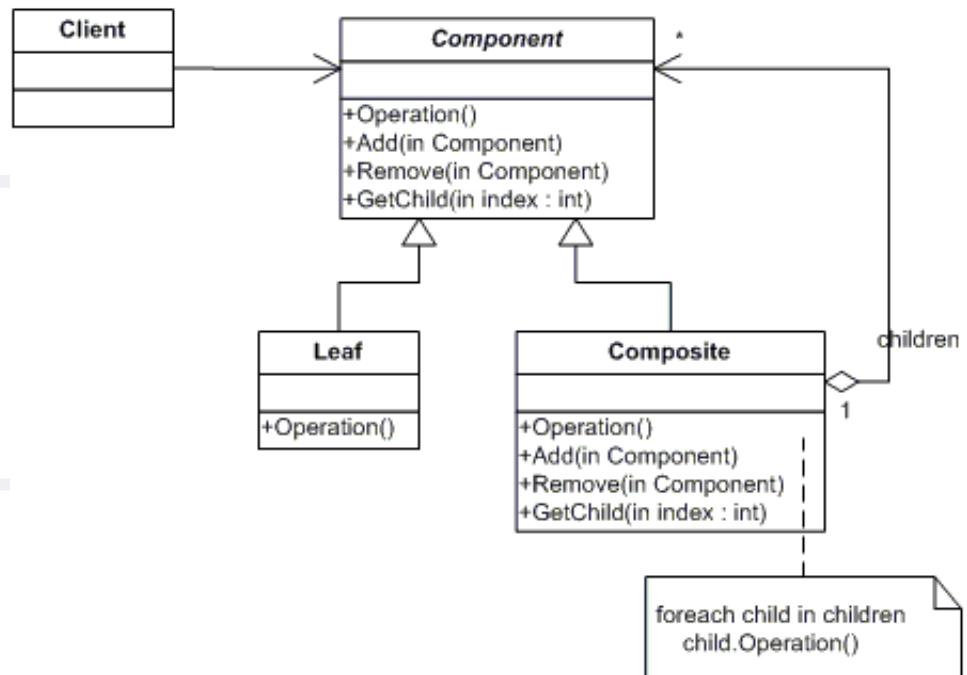
```
public class Square implements Shape {  
    @Override  
    public void draw() {  
        System.out.println("Square::draw()");  
    }  
}
```

Façade Example (3)

```
public class ShapeMaker {  
    private Shape rectangle;  
    private Shape square;  
    public ShapeMaker() {  
        rectangle = new Rectangle();  
        square = new Square(); }  
  
    public void drawRectangle(){  
        rectangle.draw(); }  
    public void drawSquare(){  
        square.draw(); }  
}
```

Composite Pattern

- Allows to **combine** different types of objects in tree structures
- Gives the possibility to treat the **same object(s)**
- Used when
 - You have different objects that you want to **treat the same way**
 - You want to present a **hierarchy** of objects



The Component Abstract Class

```
abstract class Component {  
    protected String name;  
  
    public Component(String name) {  
        this.name = name; }  
  
    public abstract void add(Component c);  
    public abstract void remove(Component c);  
    public abstract void display(int depth);  
}
```

The Composite Class (1)

```
class Composite extends Component {  
    private List<Component> children = new ArrayList<Component>();  
    public Composite(String name) { super(name); }  
    @Override  
    public void add(Component component) {  
        children.add(component); }  
    @Override  
    public void remove(Component component) {  
        children.Remove(component); }
```

The Composite Class (2)

```
@Override  
public void display(int depth) {  
    System.out.println(printNameInDepth(depth, name));  
    foreach (Component component : children) {  
        component.display(depth + 2);  
    }  
}  
  
public void printNameInDepth(int depth, String name) {  
    for(int i = 0; i < depth; i++)  
        System.out.print("-");  
    System.out.print(name);  
}
```

The Leaf Class

```
class Leaf extends Component {  
    public Leaf(String name) { super(name); }  
  
    @Override  
    public void add(Component c) {  
        System.out.println("Cannot add to a leaf"); }  
    @Override  
    public void Remove(Component c) {  
        System.out.println("Cannot remove from a leaf"); }  
    @Override  
    public void Display(int depth) {  
        System.out.println(printNameInDepth(depth, name)); }  
}
```



Behavioral Patterns

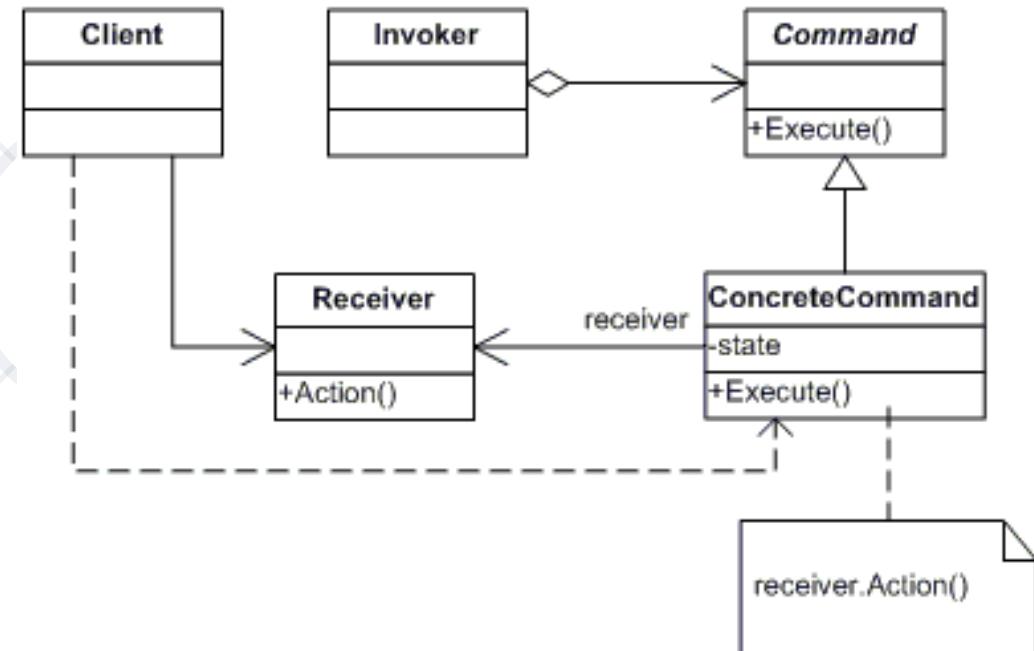
Purposes

- Concerned with the **interaction** between objects
 - Either with the **assignment of responsibilities** between objects
 - Or **encapsulating behavior** in an object and delegating requests to it
- Increases **flexibility** in carrying out cross-classes communication



Command Pattern

- An object **encapsulates** all the information needed to call a method at a later time
 - Lets you **parameterize** clients with different requests, queue or log requests, and support undoable operations



The Command Abstract Class

```
abstract class Command {  
    protected Receiver receiver;  
  
    public Command(Receiver receiver) {  
        this.receiver = receiver; }  
  
    public abstract void execute();  
}
```

Concrete Command Class

```
class ConcreteCommand extends Command {  
    public ConcreteCommand(Receiver receiver) {  
        super(receiver); }  
  
    @Override  
    public void execute() {  
        receiver.action(); }  
}
```

The Receiver Class

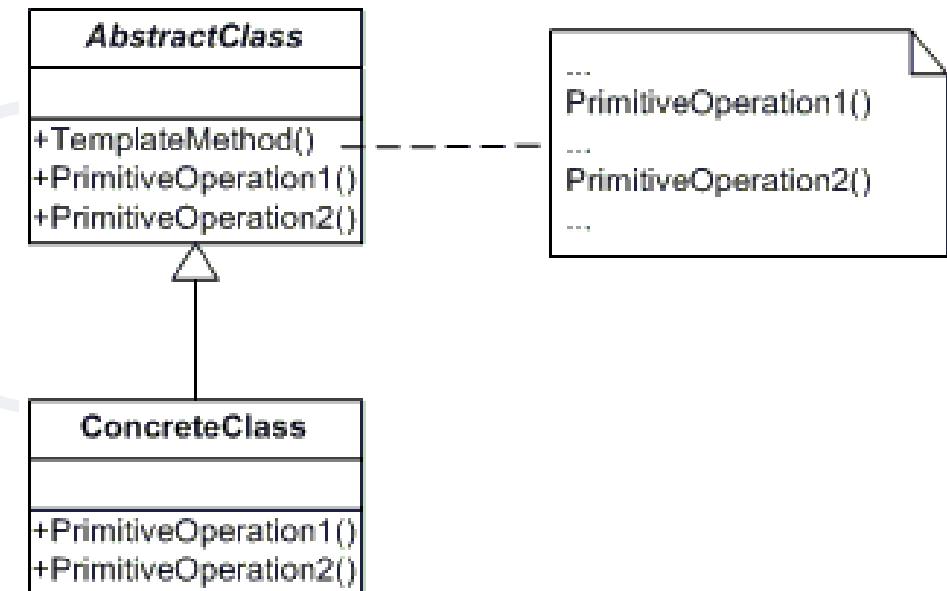
```
class Receiver {  
    public void action() {  
        System.out.println("Called Receiver.action()");  
    }  
}
```

The Invoker Class

```
class Invoker {  
    private Command command;  
  
    public void setCommand(Command command) {  
        this.command = command; }  
  
    public void ExecuteCommand() {  
        command.execute(); }  
}
```

Template Pattern

- Define the **skeleton** of an algorithm in a method, leaving some implementation to its subclasses
- Allows the subclasses to **redefine** the implementation of some of the **parts** of the algorithm, but not its structure



The Abstract Class

```
abstract class AbstractClass {  
    public abstract void primitiveOperation1();  
    public abstract void primitiveOperation2();  
  
    public void templateMethod() {  
        primitiveOperation1();  
        primitiveOperation2();  
        System.out.println(""); }  
}
```

A Concrete Class

```
class ConcreteClassA extends AbstractClass {  
    @Override  
    public void primitiveOperation1() {  
        System.out.println("ConcreteClassA.primitiveOperation1()"); }  
    @Override  
    public void primitiveOperation2() {  
        System.out.println("ConcreteClassA.primitiveOperation2()");  
    }  
}
```

- Design Patterns
 - Provide **solution** to common problems
 - Add additional layers of **abstraction**
- Three main types of Design Patterns
 - **Creational**
 - **Structural**
 - **Behavioral**



Questions?



SoftUni



Software
University



SoftUni
Creative



SoftUni
Digital



SoftUni
Foundation



SoftUni
Kids



Finance
Academy

SoftUni Diamond Partners



**SUPER
HOSTING
.BG**

INDEAVR
Serving the high achievers

 **SOFTWARE
GROUP**

 **BOSCH**



**Coca-Cola HBC
Bulgaria**

 **AMBITIONED**

createX

**DXC
TECHNOLOGY**

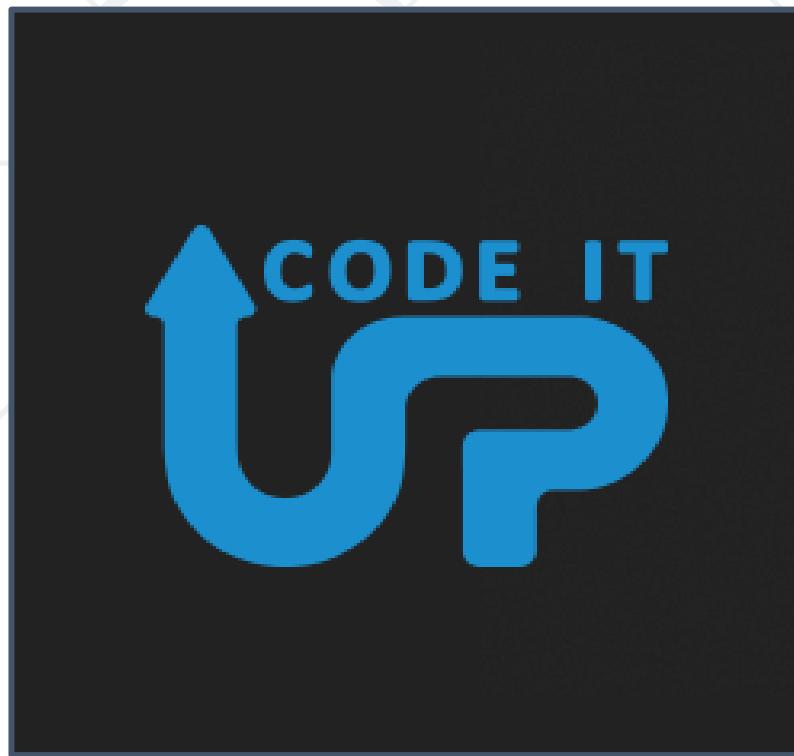
 **POKERSTARS**
POKER | CASINO | SPORTS
a Flutter International brand

 **DRAFT
KINGS**

 **Postbank**
Решения за твоето утре

 **SmartIT**

Educational Partners



Trainings @ Software University (SoftUni)



- Software University – High-Quality Education, Profession and Job for Software Developers
 - softuni.bg
- Software University Foundation
 - softuni.foundation
- Software University @ Facebook
 - facebook.com/SoftwareUniversity
- Software University Forums
 - forum.softuni.bg



Software
University



- 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>

