

Session 3

Session 3

- Homework 4 solution & review
- Abstract classes
- Abstract methods
- Composition vs inheritance
- Anonymous classes
- Mutability
- Exceptions
- Using 3rd party libraries: JUnit

Abstract Classes

- An abstract class is a class that is declared abstract.
- Abstract classes cannot be instantiated, but they can be subclassed.
- When an abstract class is subclassed, the subclass usually provides implementations for all of the abstract methods in its parent class. However, if it does not, then the subclass must also be declared abstract.

```
abstract class GraphicObject {  
    int x, y;  
    void moveTo(int newX, int newY) {  
        // ... implement...  
    }  
}  
  
class Circle extends GraphicObject {  
    void draw() {  
    }  
    void resize() {  
    }  
}
```

Abstract Methods

- An abstract method is a method that is declared without an implementation.
- If a class includes abstract methods, then the class itself must be declared abstract.

```
abstract class GraphicObject {  
    int x, y;  
    void moveTo(int newX, int newY) {  
        // ... implement...  
    }  
    abstract void draw();  
    abstract void resize();  
}  
  
class Circle extends GraphicObject {  
    void draw() {  
    }  
    void resize() {  
    }  
}
```

Anonymous Class

- A class can implement / extend a parent class and be instantiated without prior definition.
- Try: Extend and instantiate another instance, a3 and verify the class name.

```
public class Main {  
    public static void main(String[] args) {  
        A a1 = new A();  
        A a2 = new A() {  
            @Override  
            public String toString() {  
                return super.toString();  
            }  
        };  
        System.out.println(a1.getClass());  
        System.out.println(a2.getClass());  
    }  
}  
  
class A {  
}
```

Pass by value or reference?

```
class Circle {  
    private double radius;  
  
    public Circle(double r) {  
        radius = r;  
    }  
  
    public double getRadius() {  
        return radius;  
    }  
}
```

```
public void manipulate(Circle circle) {  
    circle = new Circle(3);  
}  
  
Circle c = new Circle(2);  
manipulate(c);  
System.out.println("Radius: " + c.getRadius());
```

Java is a “pass-by-value” language

Mutability

- If the state of an object can be changed after it is created, we call it a **mutable** object.
- Similarly, if we cannot change the state of an object, it is an **immutable** object.
- Note that you can always change the reference of an object, but it will just point to another object in heap.
- Changing the reference of an object does not change its internal state.

```
// String is immutable
String s = "This cannot change";

// You can change the reference
s = "Another string value";

// Mutable
Cat cat = new Cat();
cat.setAge(1);    // State changed
```


Mutating an object

```
class Circle {  
    private double radius;  
  
    public Circle(double r) {  
        radius = r;  
    }  
  
    public double getRadius() {  
        return radius;  
    }  
  
    public void setRadius(double r) {  
        radius = r;  
    }  
}
```

```
public void manipulate(Circle circle) {  
    circle.setRadius(circle.getRadius() + 1);  
}  
  
Circle c = new Circle(2);  
manipulate(c);  
System.out.println("Radius: " + c.getRadius());  
  
c = new Circle(5);  
manipulate(c);  
System.out.println("Radius: " + c.getRadius());
```

Immutable objects

- An object is considered **immutable** if its state cannot change after it is constructed (instantiated).
- Typically the parameters are allowed only via the constructor, internal state fields are private and final, and there are no public “setter” methods.
- Example: java.lang.String
- The keyword **final** in Java makes it impossible to reassign a reference after it is initialized.
- Defining the primitive state variables in a class as final, makes the class state immutable only if all variables are of primitive types.

```
final Circle c = new Circle(2);  
manipulate(c);  
  
c = new Circle(3); // error
```

```
final int x = 1;  
x = 2; // error
```

Validating the input

```
class Circle {  
    private double radius;  
  
    public Circle(double r) {  
        if (r > 0)  
            radius = r;  
        else  
            System.err.println("Wrong input");  
    }  
  
    public double getRadius() {  
        return radius;  
    }  
}
```

```
Circle c = new Circle(-1.0);
```

// What is the output?

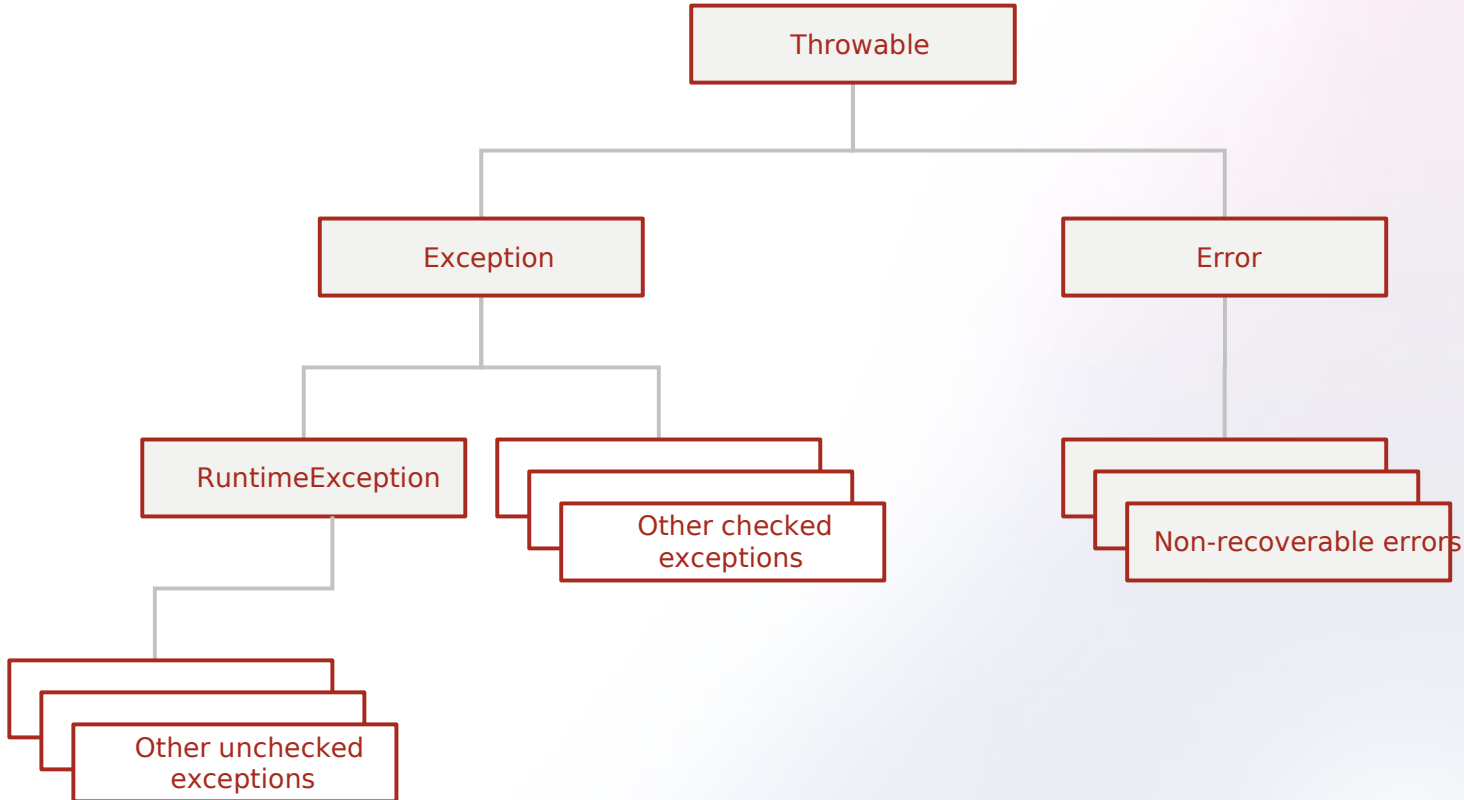
```
System.out.println("Radius: " + c.getRadius());
```

// Why this is not a good mechanism?

Validating using exceptions

- Implement a Circle object that validates the radius input in a “safer” way.
- By saying safe we mean that we should not get a Circle object if the provided radius was incorrect and if we receive an object we must be sure that it is in a “good” state.

Checked and runtime exceptions




JUnit

- JUnit is a unit testing framework for the Java programming language.
- How to test code?
 - Manual: Run the program, provide input, see if it gives you the expected output
 - Automated: Automatically “feed” your implementation classes with input, make them perform some actions, and check if the real output matches the expected output



JUnit setup

- Add JUnit to your project
 1. Search for “Maven JUnit 4” or go to <https://mvnrepository.com/artifact/junit/junit>
 2. Choose 4.13.2, then download the JAR file
 3. Create a folder “lib” in your project and move the downloaded JAR file there.
 4. In IntelliJ: File > Project Structure > Libraries > Add (+) > Java > Choose the file > OK

 JUnit » 4.13.2 JUnit is a unit testing framework to write and run repeatable automated tests on Java.	
License	EPL 1.0
Categories	Testing Frameworks
Tags	testing junit
Organization	JUnit
HomePage	http://junit.org
Date	Feb 13, 2021
Files	jar (375 KB) View All
Repositories	Central HeavenArk Minebench Lutece Paris Xceptance
Ranking	#1 in MvnRepository (See Top Artifacts) #1 in Testing Frameworks
Used By	119,073 artifacts

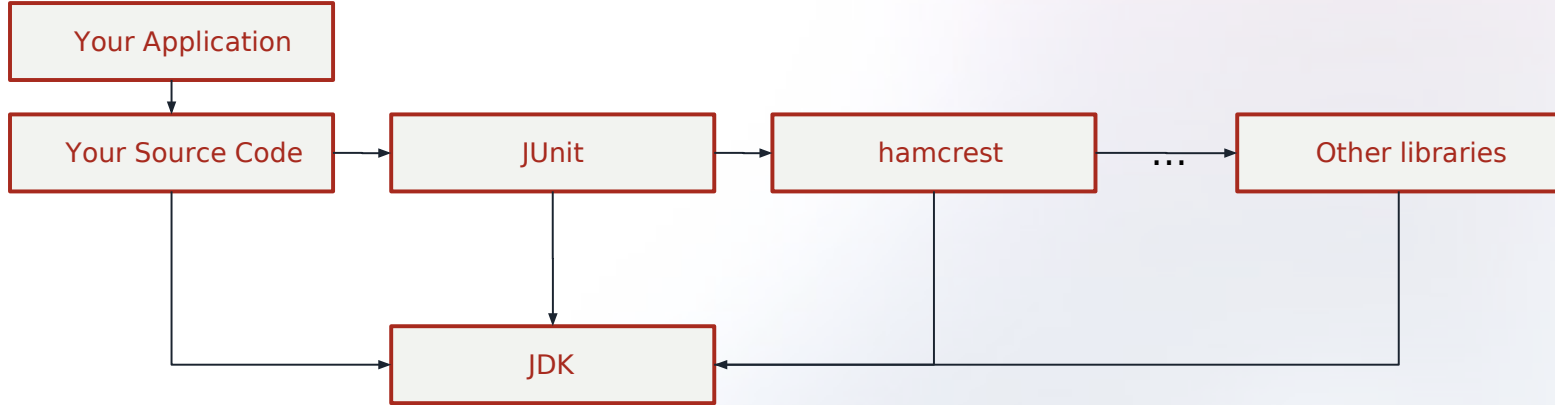
Your first JUnit test

- Create a **public class CircleTest**
- Add a method **public void testArea() { ... }**
- Annotate the method with **@Test**
- Run the method with IntelliJ (Green run icon)

```
1  import org.junit.Test;
2
3  public class CircleTest {
4      @Test
5      public void testArea() {
6          System.out.println("Circle area test");
7      }
8  }
```


Adding missing dependency

- JUnit depends on another library to run the tests
- Add it to your project in order to fix the problem.
- <https://mvnrepository.com/artifact/org.hamcrest/hamcrest-all/1.3>
- As noticed, managing external libraries manually can be difficult.
- Later, we will use Maven as a dependency & build tool to simplify this task.



Enums

- An enum type is a special data type that enables for a variable to be a set of predefined constants.
- The variable must be equal to one of the values that have been predefined for it.
- Direction: North, South, East, West
- Operations: Addition, Subtraction, Multiplication, Division
- Suitable to replace usage of predefined “constants”

```
enum Direction {  
    North,  
    South,  
    East,  
    West  
}
```

Homework 5



Using OOP techniques and Java concepts we covered so far, implement an object-oriented program that models a simple calculator.

Create a break-down of a calculator object which is made of at least a screen, buttons, and internal memory that holds the input and the calculation result.

It is enough to implement only 0-9 input buttons and basic four operations.