

INTRODUCTION

# Programming Principles II

# Topics

OOP Principle: Polymorphism  
Method overriding (revisited)

# The power of polymorphism

---

```
public void printProperties(Shape shape ) {  
    System.out.println("Area: "+ shape.calcArea());  
    System.out.println("Perimeter: "+ shape.calcPerimeter());  
}
```

This method can be called with any subclass:

```
Circle circle = new Circle(10);  
Rectangle rect = new Rectangle(20,7);  
Square sqr = new Square(15);  
  
printProperties(circle);  
printProperties(rect);  
printProperties(sqr);
```

# Overriding as Polymorphism

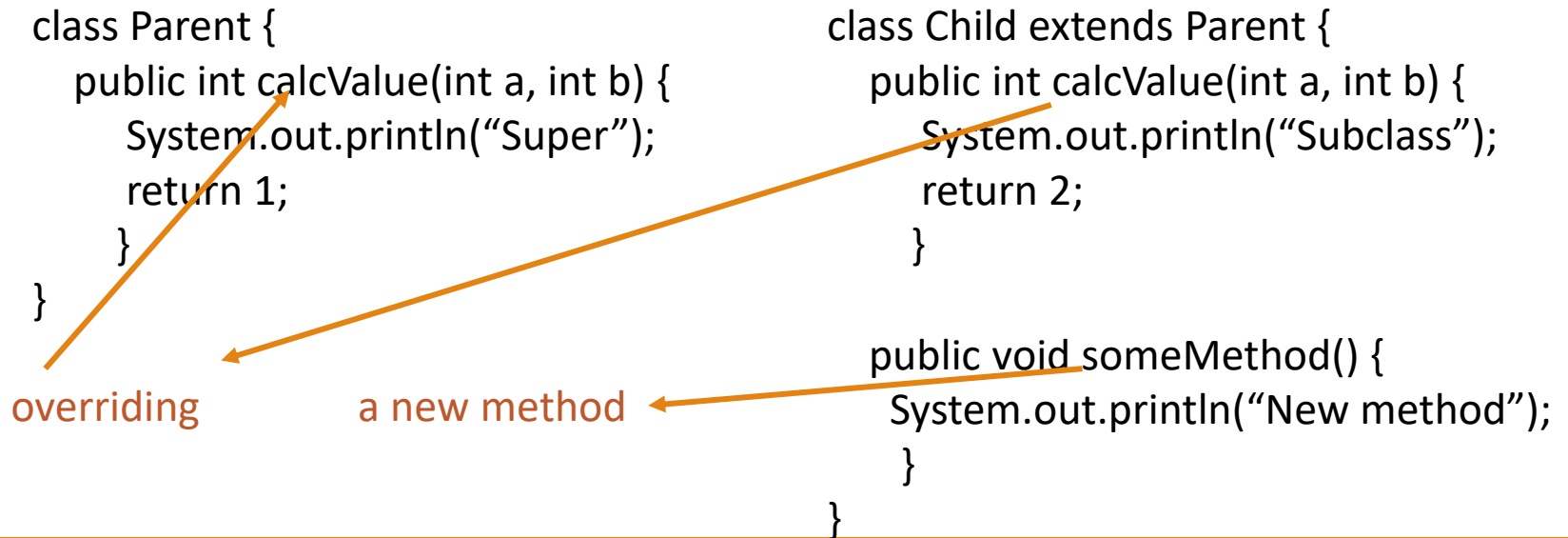
---

In Java, a reference of a super class can be assigned an instance of a child class :

Parent testObj = new Child();

Compiler allows it, since Child also has all accessible members/methods of the Parent (similar analogy to the automatic type conversion).

---



# Overriding as Polymorphism

---

For the previous inheritance example, the following code will work:

```
Parent obj = new Child();  
obj.calcValue(3,5);
```

and print “Subclass”.

Calling `obj.someMethod()` will cause an error – this method is only available in Child. To use this method the declaration shall be done as usual:

(1)  
`Child obj = new Child();`  
`obj.someMethod();`

(2)  
`Parent obj = new Child();`  
`((Child) obj).someMethod();`

# Overriding as Polymorphism

---

Assigning a parent class type reference, the subclass object allows to use the power of polymorphism: a single class can have different behaviour or forms (poly + morph) depending on the implementation:

---

```
class Animal {  
    public void makeSound() {  
        System.out.println("Any sound");  
    }  
}
```

```
Animal a1 = new Dog();  
Animal a2 = new Cat();  
a1.makeSound();  
a2.makeSound();
```

```
class Dog extends Animal {  
    public void makeSound() {  
        System.out.println("Haf-haf");  
    }  
}
```

```
class Cat extends Animal {  
    public void makeSound() {  
        System.out.println("Meow");  
    }  
}
```

# Overriding as Polymorphism

---

The declaration, initialization and method call syntax is checked during the compile of the Java classes.

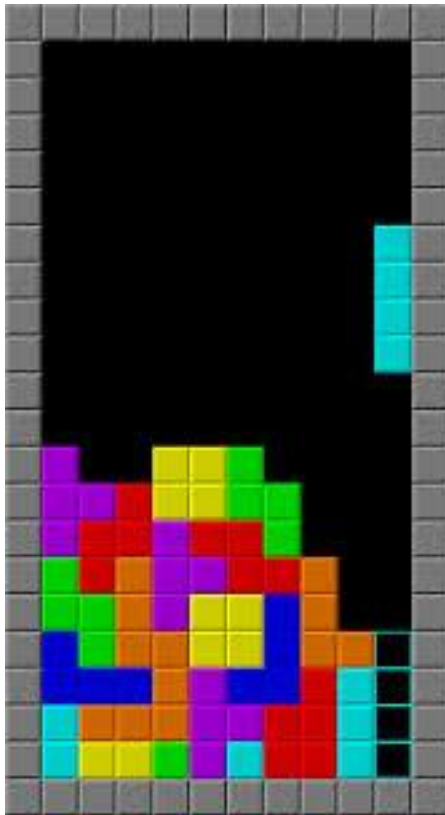
When *makeSound()* is called on a1 and a2, *Java runtime* dynamically resolves the method to be called (if it is Dog or Cat). This is called a *Dynamic Method Dispatch*, the run-time implementation of the polymorphism by Java.

This concept plays an important role in OOP: once the superclass is defined, the subclasses can be imported and used through the superclass methods that are already implemented. So, if someone implements a *Bear* class in the *animals* package, we just need to:

```
import animals.Bear;  
...  
Animal x = new Bear();  
x.makeSound();
```

# OOP in Game development

---



All shapes may inherit from one class that has the following methods:

- appear(int xpos)
- moveDown(int speed)
- stop()
- rotateLeft()
- rotateRight()
- setSpeed(int speed)

and members like:

- shapeMatrix
- color
- speed



There can be different characters that are based on one Character class with methods walk, turn, jump, swim and so on.

[OOP Design of Chess](#)

[Asteroid game from OOP perspective](#)



Topics

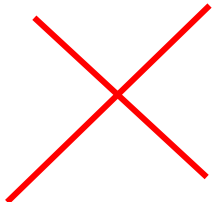
Overriding & modifiers

# Overriding & access modifiers

In Java, the access modifier of the overriding method in the subclass **cannot be more restrictive** than the overridden method of the superclass:


```
class A {  
    public void sayHi() {  
        System.out.println("Hello by A");  
    }  
}
```

```
class B extends A {  
    protected void sayHi() {  
        System.out.println("Hello by A");  
    }  
}
```



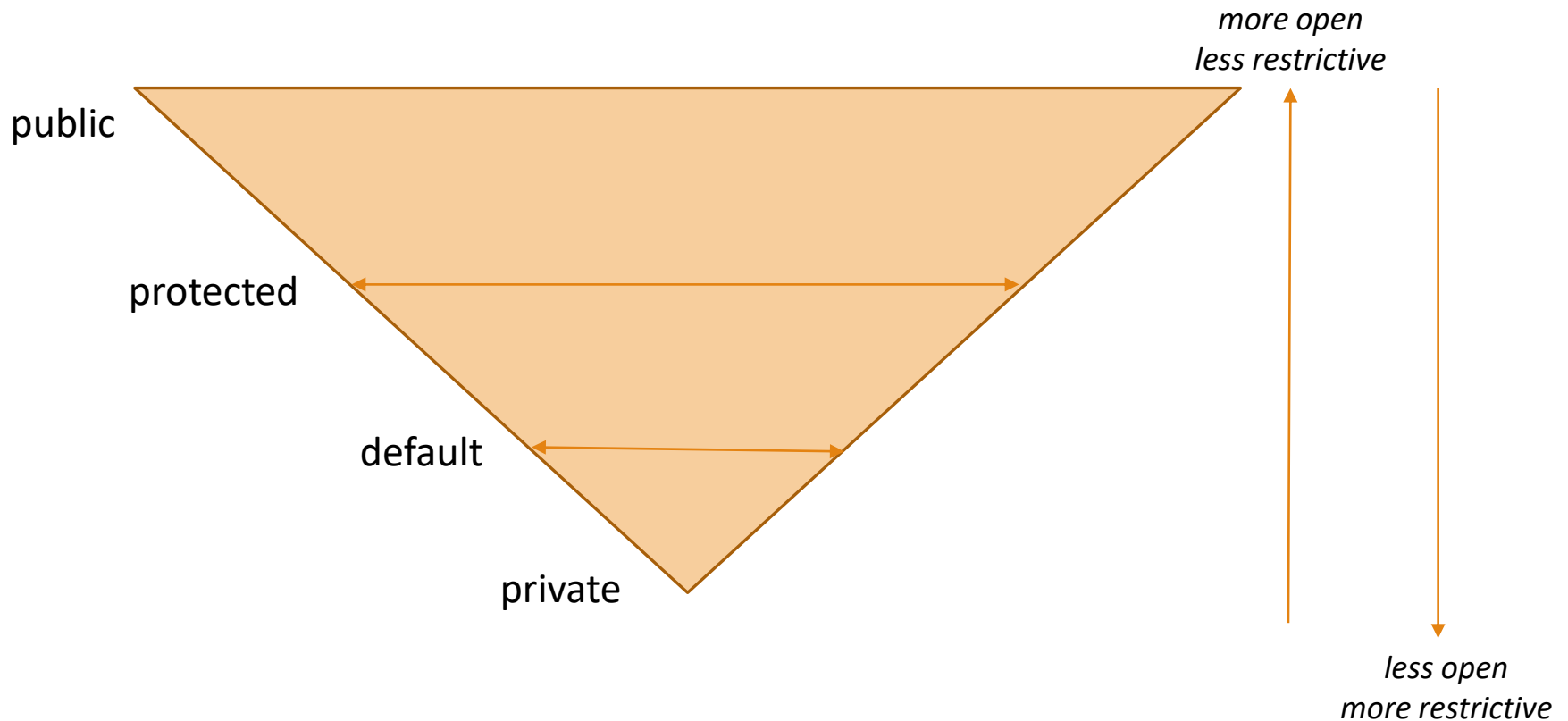
```
class A {  
    void sayHi() {  
        System.out.println("Hello by A");  
    }  
}
```

```
class B extends A {  
    {empty/protected/public} void sayHi() {  
        System.out.println("Hello by A");  
    }  
}
```



# Overriding & access modifiers

---



# Static & overriding

---

If a subclass defines a static method with the same signature as a static method in the superclass, then the method in the subclass **hides** the one in the superclass.

The distinction between hiding a static method and overriding an instance method has important implications:

- The version of the overridden instance method that gets invoked is the one in the subclass.
- The version of the hidden static method that gets invoked depends on whether it is invoked from the superclass or the subclass.

**Note:** *if in our previous example we declare the makeSound() method(s) as static, we would have a different result.*

# Static & overloading

---

In Java, a method cannot be overloaded by just having a static keyword as a difference:

```
class A {  
    public void printText() {  
        System.out.println("Non-static");  
    }  
  
    public static void printText() { // Error! – cannot redefine printText()  
        System.out.println("static");  
    }  
}
```

Topics

final keyword

# Final

---

In Java, the *final* keyword helps define objects and types as unmodifiable.

```
int x = 5;  
x = 10;
```

```
final int x = 5;  
x = 10; // error
```

# Final

---

Final objects and arrays (which are objects as well) can change their values through the methods and indices correspondingly but **cannot** have a new reference assigned to themselves:

```
final SomeObj obj = new SomeObj();
```

```
obj.setValue(25); // it is ok
```

```
obj = new SomeObj(); // error
```

```
final int arr[] = { 1,2,3};
```

```
arr[1] = 25; // it is ok
```

```
arr = new int[3]; // error
```



# Final - initialization

---

The final variable can be initialized in the declaration line as described in the previous slides or can be initialized in:

## Instance initializer

```
class MyClass {  
    final int a;  
  
    {  
        a = 152;  
    }  
  
    public MyClass(int x) {}  
}
```

## Constructor

```
class MyClass {  
    final int a;  
  
    public MyClass(int x) {  
        a = x;  
    }  
}
```

# Final - CONSTANT<sub>(member)</sub>S

---

Final keyword is usually used to define members as constants:

```
final int MAX_VALUE = 25;
```

As you remember from PP1 course, according to the naming conventions, the constant variables are defined with capital letters.

To make the constant available without a class instantiation ***public*** and ***static*** keywords shall be used:

```
public static final int MAX_VALUE = 25;
```

# Final - prevention of overriding

---

Using final keyword in method declaration **prevents** its overriding:

```
class Parent {  
    public final void someMethod() {  
        System.out.println("Parent");  
    }  
}
```

```
class Child extends Parent {  
    public void someMethod() { // will cause an error: cannot override  
        System.out.println("Child");  
    }  
}
```

# Final - prevention of inheritance

---

Declaring a class as final means the class **cannot** be extended by any other class:

```
final class Parent {  
    public void someMethod() {  
        System.out.println("Parent");  
    }  
}
```

```
class Child extends Parent {  
  
}
```

// will cause an error: Parent is final!

Topics

Object class

# Object class

---

There is a ***java.lang.Object*** class that is an implicit superclass of all Java classes. Any Java class instance can be assigned to an Object variable:

```
Object obj = new Dog();
```

The below methods implemented in Object can be called on any Java object as well:

Object clone()	Creates a copy of the object
boolean equals(Object obj)	Checks the equality of the provided object to the object itself
Class getClass()	Gets the class of an object at runtime
int hashCode()	Return the hash code associated with the object
String toString()	converts the object to String

# The clone() method

---

The *clone()* method of Object class allows to create the copy of the object:

```
Rectangle r1 = new Rectangle(10,5);    Rectangle r1 = new Rectangle(10,5);  
Rectangle r2 = r1;                      Rectangle r2 = (Rectangle)r1.clone();
```

```
System.out.println(r1.equals(r2));  
System.out.println(r1 == r2);
```

The clone() is a good idea of creating a copy of an array:

```
int [] a = {1, 2, 3, 4, 5, 6, 7};  
int [] b = (int []) a.clone();  
a[3] = 0; // b is not changed
```

# Homework – non graded

---

Create a Rectangle class and override the ***equals()*** method of the Object class such that, if the width and height of the provided rectangle are the same as the current object, then return true.

```
class Rectangle {  
    int width, height;  
  
    public Rectangle(int w, int h) {  
        width = w;    height = h;  
    }  
  
    public boolean equals(Object obj) {  
        Rectangle rect = (Rectangle) obj;  
        // your code here  
    }  
}
```

And test it with below code:

```
Rectangle r1 = new Rectangle(5,10);  
Rectangle r2 = new Rectangle(15,10);  
Rectangle r3 = new Rectangle(5,10);  
  
System.out.println(r1.equals(r2));  
System.out.println(r1.equals(r3));
```



# Homework – step 2

---

Now create a Square class that extends Rectangle and has a constructor with one argument – side. Test your code as given in the following example to see the power of OOP!

```
Object o1 = new Rectangle(5,10);  
Object o2 = new Rectangle(15,15);  
Object o3 = new Square(15);  
  
System.out.println("Objects are identical: "+ o1.equals(o2));  
System.out.println("Objects are identical: "+ o1.equals(o3));  
System.out.println("Objects are identical: "+ o2.equals(o3));
```

# Returning objects

---

In Java, methods may get objects as a parameter and return them:

```
public boolean testDog(Animal a) {  
    String str = a.makeSound();  
  
    return "Huf-huf".equals(str);  
}
```

```
public Animal createAnimal(String str) {  
    Animal a;  
  
    if ("dog".equals(str))  
        a = new Dog();  
    else if ("cat".equals(str))  
        a = new Cat();  
    else  
        a = new Bear();  
  
    return a;  
}
```

Topics

call-by-value / call-by-reference

# Recollecting PP1 stuff

---

What will be the output of below code?

```
class PassExample {  
    public void changeValue(int x) {  
        x = x + 2;  
    }  
  
    public static void main(String args[]) {  
        PassExample pe = new PassExample();  
        int a = 5;  
        System.out.println("In the start a is :" + a);  
        pe.changeValue(a);  
        System.out.println("Now a is :" + a);  
    }  
}
```

# Passing variables

---

How about this one?

```
class PassExample {  
    public void changeValue(int [] arr) {  
        arr[1] = -1;  
    }  
  
    public static void main(String args[]) {  
        PassExample pe = new PassExample();  
        int [] arr = {1,2,3};  
        System.out.println("In the start : {" + arr[0] + ","+arr[1]+","+arr[2]+"}");  
        pe.changeValue(arr);  
        System.out.println("Now : {" + arr[0] + ","+arr[1]+","+arr[2]+"}");  
    }  
}
```

# Passing variables

---

And this one?

```
class MyObj{
    int a = 5;

    public int getA() {
        return a;
    }

    public void setA(int a) {
        this.a = a;
    }
}
```

```
class PassExample {
    public void changeValue(MyObj obj) {
        obj.setA(22);
    }

    public static void main(String args[]) {
        PassExample pe = new PassExample();
        MyObj mo = new MyObj();
        System.out.println("In the start :" + mo.getA());
        pe.changeValue(mo);
        System.out.println("Now a is :" + mo.getA());
    }
}
```

# Passing variables

---

In Java, the primitive types are passed to the methods as values (a copy of their values) – the changes made to the primitive type parameter in the method does not change its value. This is called *call-by-value*.

The objects are passed as a reference – all changes made to the objects change the original object. This is called *call-by-reference*.

# Topics

OOP Principle: Abstraction

Abstract classes



# Abstract classes

---

In the *Animal* superclass, the *makeSound()* method does not need any implementation – there is no any general sound that characterizes all animals. In this case, we can leave this method ***unimplemented***.

These unimplemented methods are called ***abstract*** and have the following syntax:

```
abstract returnType methodName(parameters);
```

As seen from the syntax, there is no curly braces that define the body of the method. The method declaration ends with the semicolon.

Example:

```
abstract int calcArea(int length, int width);  
  
public abstract void makeSound();
```

# Abstract classes

---

Having a method as abstract says that it needs to be implemented by the subclasses.

A class with an abstract method shall be defined as abstract itself:

```
abstract class Animal {  
    public void someMethod () {  
        System.out.println("Some method with implementation");  
    }  
    abstract void makeSound();  
}
```

An abstract class **cannot be instantiated**:

```
Animal a = new Animal(); // will raise an error.
```

# Abstract classes

---

The subclass of the abstract class shall **implement the abstract methods of the superclass** or **define itself as abstract** as well:

```
abstract class Animal {  
    public abstract void makeSound();  
}
```

```
abstract class Dog extends Animal {  
    public boolean isTrainable() {  
        return true;  
    }  
}
```

*If not declared as abstract, will  
raise a compile error*



```
abstract class Animal {  
    public abstract void makeSound();  
}
```

```
class Dog extends Animal {  
    public boolean is Trainable() {  
        return true;  
    }  
}
```

```
    public void makeSound() {  
        System.out.println("Huf-huf");  
    }  
}
```

Topics

Interfaces

# Interfaces

---

Interfaces are similar to abstract classes with below exceptions:

- Interfaces shall not implement any code ( prior to JDK8 )
- Interfaces do not have instance variables
- Interfaces do not have constructors
- The default access modifier for interface methods is **public**
- A class can implement more than one interface (recall multiple inheritance)

*Generally speaking, interfaces tell you **what to do** but **NOT how to do it**.*

# Interfaces - syntax

---

```
access interface name {  
    return-type method-name1(parameter-list);  
    return-type method-name2(parameter-list);  
  
    type final-varname1 = value;  
    type final-varname2 = value;  
    //...  
    return-type method-nameN(parameter-list);  
    type final-varnameN = value;  
}
```

Interfaces can have **public or no access modifier**. If no access modifier is given, then the interface is accessible only to classes defined in the same package as the interface.

# Interfaces - implementation

---

*We use implements,  
not extends!*

```
interface MyInt {  
    double calcCircumference(double radius);  
    double calcArea(double radius);  
}
```

*The methods that  
implement an  
interface must be  
declared **public**.*

```
class Circle implements MyInt {  
    public double calcCircumference(double radius) {  
        return 2*Math.PI * radius;  
    }  
    public double calcArea(double radius) {  
        return Math.PI * Math.pow(radius,2);  
    }  
}
```

# Interfaces – incomplete implementations

---

```
interface MyInt {  
    double calcCircumference(double radius);  
    double calcArea(double radius);  
}
```

*If a class does not fully implement the methods required by the interface, then that class must be declared as **abstract**.*



```
abstract class Circle implements MyInt {  
  
    public double calcArea(double radius) {  
        return Math.PI * Math.pow(radius,2);  
    }  
}
```



# Interfaces

---

As mentioned previously, a class can implement more than one interface (recall multiple inheritance). Interfaces can be implemented by extending a superclass at the same time:

```
class SomeClass extends Superclass implements Int1, Int2, Int3 {}
```

An interface can extend another interface:

```
interface EllipseInt extends CircleInt {}
```

# Interfaces - variables

---

Variables defined in interfaces are implicitly made as *final* and *static*.

Variables need to have an initial value:

```
int a; // will raise an error
```

```
int a = 5; // OK. Will be created as final static int a = 5
```

# Interfaces - notes

---

- Starting JDK 8, interfaces
  - can define the implementation of the methods, through the default keyword
  - can have static methods
- Starting JDK 9, interfaces can include private methods.

# Interfaces – default method

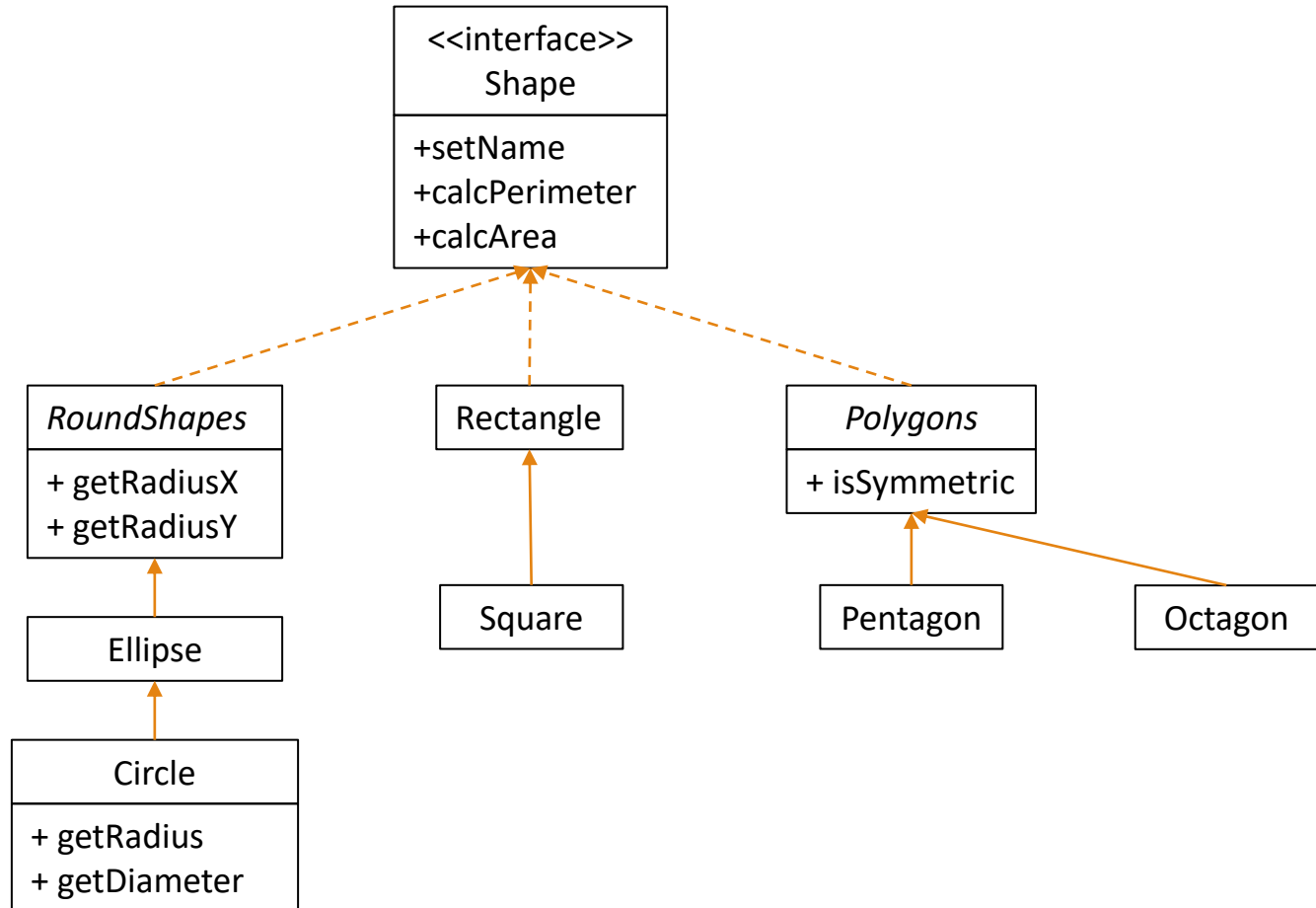
---

Starting JDK8, an interface method can have an implementation by having a **default** keyword:

```
interface SampleInterface {  
    void unimplementedMethod();  
  
    default int calcArea(int x, int y) {  
        return x * y;  
    }  
}
```

# Shapes implemented

---



# Homework (non-graded)

---

Create an interface called **Exam**. It shall have the following methods:

- void start()
- void finish()
- boolean isOver()
- void setGrade(double g)
- double getGrade()

# Homework (non-graded)

---

Create **MidtermExam** and **FinalExam** classes that implement the Exam interface.

When `start()` is called,

- MidtermExam shall print “Midterm started”
- FinalExam shall print “Final started”

When `finish()` is called,

- MidtermExam shall print “Midterm finished”
- FinalExam shall print “Final finished”

In both classes

- when `isOver()` called, it shall return true if the exam is finished. Otherwise, it shall return false (hint: you need to add a boolean field for it).
- declare a field called *grade* and implement `setGrade()` and `getGrade()` methods.

# Homework (non-graded)

---

The FinalExam shall have additional fields:

- double [] assignmentGrades;
- double [] quizGrades;
- MidtermExam midtermResult;

Write getter/setter for each of fields.

In FinalExam, write a new method called *calcTotal()*. It shall return the sum of all assignment, quiz, midterm and final grades, only if the final exam is over. Otherwise, it shall return -1;



# Homework (non-graded)

---

Create each code in separate files: Exam.java, MidtermExam.java and FinalExam.java.

If everything is implemented correctly, the below class shall compile, run and print 84:

```
public class HomeworkTest {  
    public static void main(String args[]) {  
        MidtermExam m = new MidtermExam();  
        m.setGrade(21);  
  
        double [] assg = { 7, 5, 8, 5};  
        double [] quiz = { 3, 2, 5, 2, 3};  
        FinalExam f = new FinalExam();  
        f.setAssignmentGrades(assg);  
        f.setQuizGrades(quiz);  
        f.setMidtermResult(m);  
        f.setGrade(23);  
        f.finish();  
        double total = f.calcTotal();  
        System.out.println("Your total grade is: "+total);  
    }  
}
```

Topics

Enumerations

# Enumerations

---

An ***enumeration*** is a list of named constants that define a new data type and its values. The syntax is similar to C language:

```
enum Color { Red, Green, Blue, White, Yellow, Purple, Orange}
```

In Java, an enumeration is an object, and it can hold only provided values.

```
Color myColor;  
myColor = Color.Red;  
  
if (myColor == Color.Red) {  
    ...  
}
```

Enums are implicitly final, which means you cannot extend or implement them.

Read more: <https://docs.oracle.com/javase/tutorial/java/javaOO/enum.html>

# Enumerations

---

The enumerations are usually used to define types that can take discrete and strict values.

In this case, the assignment and checking the value of the variable is quite easy:

```
enum Response { YES, NO, CANCEL };  
Response userResponse;  
....  
if (userResponse == Response.YES) { ...}
```

Enums are also widely used in error code definition:

```
enum ErrorCode { NO_DATA_FOUND, ILLEGAL_OPERATION, ... };  
...  
if (x == 0) { return ErrorCode.ILLEGAL_OPERATION}
```

# Enumerations

---

When used in a ***switch*** statement, there is no need to provide the enum type (i.e., no need to use Color.Red, Red is just right):

```
switch (myColor) {  
    case Red:  
        System.out.println("The color is red");  
        break;  
    case Green:  
        System.out.println("The color is green");  
        break;  
    default:  
        System.out.println("The color is different");  
        break;  
}
```

# Enumerations

---

The ***values()*** method returns an array that contains a list of the enumeration constants.

The ***valueOf(String str)*** method returns the enumeration constant whose value corresponds to the string passed in *str*.

```
Color colors[] = Color.values();  
for (Color c : colors)  
    System.out.println(c);
```

```
Color col = Color.valueOf("Orange");  
System.out.println("There is "+col+" in our list");
```

# Example

---

```
class MyEnums {
    enum Color { Red, Green, Blue, White, Black, Gray, Yellow};
}

public class EnumTest {
    public static void main(String[] args) {
        try {
            MyEnums.Color argCol = MyEnums.Color.valueOf(args[0]);
            System.out.println("Exists: " +argCol);
        }
        catch (java.lang.IllegalArgumentException iae) {
            System.out.println(args[0]+" does not exist. ");
            MyEnums.Color colors [] = MyEnums.Color.values();
            System.out.println("Choose one the following colors: ");
            for (MyEnums.Color c : colors)
                System.out.println(" - "+c);
        }
    }
}
```

# Enumerations as Classes

---

Since enumerations are Java classes(that inherit *java.lang.Enum*), they can have constructors and methods. The constructors are called each time when an enumeration constant is created:

```
enum Fruit {  
    Apple(2), Banana(2), Grapes(3), Plum(4);  
  
    private int price;  
  
    Fruit() { this.price = -1; }  
    Fruit(int p) { this.price = p; }  
  
    int getPrice() { return price; }  
}  
  
Fruit f = Fruit.Apple;
```



# Ordinal

---

The position of the enumeration constant can be obtained by calling the *ordinal()* method:

```
Fruit f = Fruit.Grapes;  
System.out.println(f);  
System.out.println(f.ordinal());
```

Unlike C++, we cannot set the ordinal number of an enum value. Instead, we can use the approach shown in previous slide.

# Enumerations: abstract methods

---

Enumerations might have abstract methods too. Then each enumeration constant must override it.

```
enum Response {  
    YES {  
        public String asAzerbaijani() { return "BƏLİ"; }  
    },  
    NO {  
        public String asAzerbaijani() { return "XEYR"; }  
    },  
    CANCEL {  
        public String asAzerbaijani() { return "LƏĞV"; }  
    };  
    public abstract String asAzerbaijani();  
}
```

.....

.....

```
Response myResponse = Response.YES;  
  
System.out.println(myResponse.toString());  
System.out.println(myResponse.asAzerbaijani());
```

Topics

Nested classes

# Nested classes

---

The Java programming language allows you to define a class within another class. Such a class is called a ***nested class*** :

```
class OuterClass {  
    ...  
    class NestedClass {  
        ...  
    }  
}
```

# Inner classes

---

Nested classes are divided into two categories: static and non-static.

Nested classes that are declared static are called ***static nested classes***.  
Non-static nested classes are called ***inner classes***.

```
class OuterClass {  
    ...  
    static class StaticNestedClass {  
        ...  
    }  
    class InnerClass {  
        ...  
    }  
}
```

# Nested classes

---

A nested class is a member of its enclosing class.

- Non-static nested classes (inner classes) **have access** to other members of the enclosing class, *even if they are declared private*.
- Static nested classes **do not have access** to other members of the enclosing class.
- Since an inner class is associated with an instance, it cannot define any static members itself.
- As a member of the OuterClass, a nested class can be declared *private*, *public*, *protected*, or *package private*. (Recall that outer classes can only be declared *public* or *package private*.)

# Compiled classes

---

When the following code is compiled (javac OuterDemo.java):

```
public class OuterDemo {  
    class InnerDemo {  
  
    }  
  
    static class StaticNestedDemo {  
  
    }  
}
```

The following files will be created:

**OuterDemo.class**

**OuterDemo\$InnerDemo.class**

**OuterDemo\$StaticNestedDemo.class**

# Static Nested class

---

A static nested is accessed with the following syntax:

```
OuterClass.StaticNestedClass
```

To create an object of the static nested class:

```
OuterClass.StaticNestedClass nestedObject =  
    new OuterClass.StaticNestedClass();
```

To instantiate an inner class, the outer class must be instantiated first:

```
OuterClass outerObject = new OuterClass();  
OuterClass.InnerClass innerObject = outerObject.new InnerClass();
```



# Initializers

---

```
public class NestedDemo {
    class InnerClass {
        {
            System.out.println("InnerClass - init");
        }
        // cannot have static initializers
    }

    static class StaticNestedClass {
        {
            System.out.println("StaticNestedClass - init");
        }
        static {
            System.out.println("StaticNestedClass - static init");
        }
    }
}
```

# Shadowing

---

If an inner class declares the member as its outer class, then the declaration *shadows* the declaration of the outer class's scope. To refer to the outer class's member, the *OuterClass.this.member* syntax shall be used.

```
public class ShadowTest {
    public int x = 0;

    class FirstLevel {
        public int x = 1;

        void methodInFirstLevel(int x) {
            System.out.println("x = " + x + "\nthis.x = " + this.x + "\nShadowTest.this.x = " + ShadowTest.this.x);
        }
    }

    public static void main(String [] args) {
        ShadowTest st = new ShadowTest();
        ShadowTest.FirstLevel fl = st.new FirstLevel();
        fl.methodInFirstLevel(23);
    }
}
```

Output:

```
x = 23
this.x = 1
ShadowTest.this.x = 0
```

# Topics

More on Inner classes

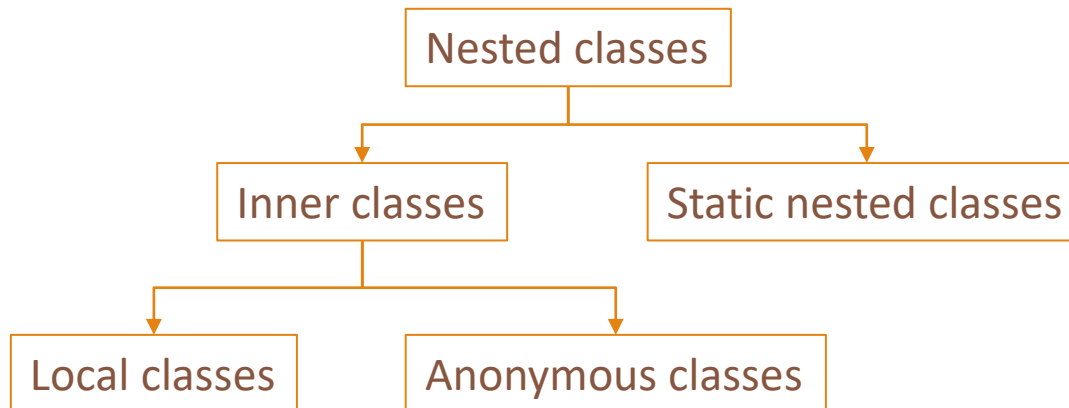
Anonymous classes

# Inner classes

---

There are two types of the Inner classes:

- **Local classes** that are defined within a code block (method, loop, etc.)
- **Anonymous classes** that enable to declare and instantiate a class at the same time



# Local classes

---

A local class can be defined inside any block - in a method body, a for loop, or an if clause.

```
public void operate(int val) {  
    class MathLocal {  
        int v;  
  
        MathLocal(int v) { this.v = v; }  
  
        int calcCube() {return v*v*v;}  
  
        int factorial() {  
            int f = 1;  
            for (int i = 1; i <= v; i++)  
                f *= i;  
            return f;  
        }  
    }  
  
    MathLocal ml = new MathLocal(val);  
    System.out.println("Cube is "+ml.calcCube());  
    System.out.println("Factorial is "+ml.factorial());  
}
```

# Local classes

---

Starting Java 8, local classes can access the method parameters:

```
public void operate(int val) {  
    class MathLocal {  
  
        int calcCube() {return val*val*val;}  
        int factorial() {  
            int f = 1;  
            for (int i = 1; i <= val; i++)  
                f *= i;  
            return f;  
        }  
    }  
  
    MathLocal ml = new MathLocal();  
    System.out.println("Cube is "+ml.calcCube());  
    System.out.println("Factorial is "+ml.factorial());  
}
```

*This might be a reason  
to use local classes –  
there is no need to have  
a constructor or setters  
to introduce variables.  
Just take the method  
argument and use it!*

# Local classes

---

Starting Java 8, local classes can access the variables of the outer scope, if those variables are ***final*** or ***effectively final*** :

```
public void operate(int val) {  
    final int a = 1;  
    int b = 2;    // It's effectively final, means the value will not be changed  
  
    class MathLocal {  
        int calcCube() {return val*val*val;}  
  
        int factorial() {  
            // b = 3;    if uncomment, will cause an error – b is not effectively final anymore  
            int f = 1;  
            for (int i = 1; i <= val; i++)  
                f *= i;  
            return f;  
        }  
  
        int test() { return a*b; }  
    }  
}
```

# Local classes

---

- Local classes are similar to inner classes - they **cannot** define or declare any static members.
- Local classes are **non-static** because they have access to instance members of the enclosing block.
- It is **not allowed** to declare an interface inside a block; interfaces are inherently static.
- It is **not allowed** to declare static initializers or member interfaces in a local class.
- A local class **can** have static members declared as **final** (constants)



# Anonymous classes

---

*Anonymous classes* are inner classes without a name that help declare and instantiate a class at the same time. Anonymous classes are useful when the local class is used for the single use (only once).

An anonymous class can be created by instantiation of an interface or abstract class. In the following code example, we create an object that implements the Runnable interface:

```
Runnable myRunnable = new Runnable() {  
  
    public void run() {  
        // Implementation is here  
    }  
  
};
```

← Pay attention to the semicolon at the end!

# Anonymous classes

---

By using anonymous classes, it's easy to implement an interface or extend a class on the go and use it. The following class creates a Runnable object that calculates the factorial and starts it right away:

```
class LocalClassTester {  
    public void operate(int val) {  
        Runnable factorialThread = new Runnable() {  
            public void run() {  
                int fact = 1;  
                for (int i = 1; i <= val; i++)  
                    fact *= i;  
                System.out.println("Result : "+fact);  
            }  
        };  
        Thread t = new Thread(factorialThread);  
        t.start();  
        // Continue to do the other stuff....  
    }  
  
    public static void main(String [] args) {  
        LocalClassTester lct = new LocalClassTester();  
        lct.operate(5);  
    }  
}
```

# Anonymous classes

---

Like local classes, anonymous classes can capture variables; they have the same access to local variables of the enclosing scope:

- An anonymous class has access to the members of its enclosing class.
- An anonymous class cannot access local variables in its enclosing scope that are not declared as final or effectively final.
- Like a nested class, a declaration of a type (such as a variable) in an anonymous class shadows any other declarations in the enclosing scope that have the same name.

Anonymous classes also have the same restrictions as local classes with respect to their members:

- You **cannot** declare static initializers or member interfaces in an anonymous class.
- An anonymous class **can** have static members provided that they are constant variables.

# Topics

Sealing classes

Encapsulation with Records

# Sealing classes

---

In Java classes can be extended by any class that has access to it or can be made final to prevent from being extended. What if one can let the class be extended by a limited number of specific subclasses?

A *sealed class* is a class which allows only a specific subclasses to directly extend it.

Note that indirectly sealed classes can be extended.

Sealed classes should be defined in the same package or named module as their direct subclasses.

The *permits* clause is optional if the sealed class and its direct subclasses are declared within the same file, or the subclasses are nested within the sealed class.

# Sealing classes

---

Sealed classes can be extended by only one of the followings:

- A Final class
- Another sealed class
- A non-sealed class

Sealed class keywords:

- sealed - Indicates that a class or interface may only be extended/implemented by named classes or interfaces
- permits - Used with the sealed keyword to list the classes and interfaces allowed
- non-sealed - Applied to a class or interface that extends a sealed class, indicating that it can be extended by unspecified classes

# Sealing classes

*sealed* class Bear *permits* Kodiak, Panda, Tiger {

```
final class Kodiak extends Bear {}
```

```
non-sealed class Panda extends Bear {}
```

```
sealed class Tiger extends Bear
    permits Siberian {}
final class Siberian extends Tiger {}
```

```
class Giant extends Panda{}  
class Red extends Panda{}
```

# Encapsulation with Records

---

According to encapsulation fields of a class is made private providing access to them by accessors (getters) and modifiers (setters).

This has certain benefits, but what if you have 20 fields of the class?

Java 14 introduced Records to reduce time spend on developing boilerplate code for each POJO (plain old java object).

```
public record Account(String customerName, double balance) { }
```



keyword

record  
name

List of fields  
surrounded by parantheses



# Encapsulation with Records

---

Record is a special type of data-oriented class within which the compiler inserts the boilerplate code including:

- All arguments constructor (data types and the order of the fields in the constructor are exactly the same as they are defined in the record)
- Getters (without a **get** or **is** prefix)
- Implementations of `toString()`, `equals()` and `hashCode()`

However, additional methods can be added to the records as well.

Records are implicitly final, i.e., they cannot be extended.

Records are immutable, that is why there are no setter methods.

The fields of the records are inherently final, i.e., no setter methods can be added.

# Records – defining constructors

---

If some validations need to be added in the constructor, constructor can be manually added to the records:

- Long constructor (a constructor that sets all the fields)
- Compact constructor (a constructor that only performs validation and necessary transformations)

# Records – long constructor

---

```
public Account(String customerName, double balance) {  
    this.customerName = customerName == null ||  
        customerName.trim().isEmpty()  
        ? "" : customerName;  
    this.balance = balance < 0 ? 0 : balance;  
}
```

The drawback of this constructor is if you have a long list of fields and only a few needs validation or transformation, you still need to set all the fields.

Long constructors can also be overloaded.

# Records – compact constructor

The constructor has neither parentheses nor input parameters

```
public Account {  
    customerName = customerName == null ||  
        customerName.trim().isEmpty()? "" : customerName;  
  
    customerName = customerName.substring(0,1).toUpperCase()  
        + customerName.substring(1).toLowerCase()  
  
    balance = balance < 0 ? 0 : balance;  
}
```

Long constructor is implicitly called in the end

One can define either long or compact constructor, not both.

# Topics

Monetary Operations

BigInteger & BigDecimal classes

# Monetary Operations

---

The problem with **float** and **double**:

```
System.out.println(2.35 - 1.95);
```

**The result is different than 0.4 (try it)**

or try this one:

```
System.out.println(1000000.0f + 1.2f - 1000000.0f);
```

# Monetary Operations

---

***BigDecimal*** class helps solve some problems with doing floating-point operations with *float* and *double*.

The value of the BigDecimal is initialized as String:

```
import java.math.BigDecimal;
```

```
BigDecimal amount = new BigDecimal("22.15");
```

# Monetary Operations

---

***java.math.BigDecimal*** class provides the methods for manipulating the numbers:

- `add(value)`
- `subtract(value)`
- `multiply()`
- `divide(value, scale, roundingMode)`



# Monetary Operations

---

To output the value of BigDecimal ***toString()*** (can output some scientific details) or ***toPlainString()*** (simple output) methods can be used.

***setScale(int newScale, RoundingMode roundingMode)*** method allows to define the number of digits after the decimal point:

```
BigDecimal bd = new BigDecimal("12.34567");  
bd = bd.setScale(2, RoundingMode.FLOOR);
```

***Note: BigDecimal is also immutable***

**Read more:** Joshua Bloch: Effective Java, ITEM 60: AVOID FLOAT AND DOUBLE IF EXACT ANSWERS ARE REQUIRED

**Read more:** <https://docs.oracle.com/en/java/javase/11/docs/api/java.base/java/math/RoundingMode.html>

# Monetary Operations

---

```
BigDecimal bd1 = new BigDecimal("2.35");  
BigDecimal bd2 = new BigDecimal("1.95");  
  
BigDecimal result = bd1.subtract(bd2);
```

# BigInteger & BigDecimal

---

Both classes can be used when it is required to store and process arbitrary large numbers (and precise in case of BigDecimal)

They provide operations analogous to those of primitive numbers in Java as well as relevant methods of `java.lang.Math` class.

Read more:

<https://docs.oracle.com/en/java/javase/11/docs/api/java.base/java/math/BigInteger.html>

<https://docs.oracle.com/en/java/javase/11/docs/api/java.base/java/math/BigDecimal.html>

# Questions

