



# Object-Oriented Programming In Mechatronic Systems

## Summer School

### Module 3

Aachen, Germany, August 8<sup>th</sup>, 2018

Cybernetics Lab IMA & IfU  
Faculty of Mechanical Engineering  
RWTH Aachen University



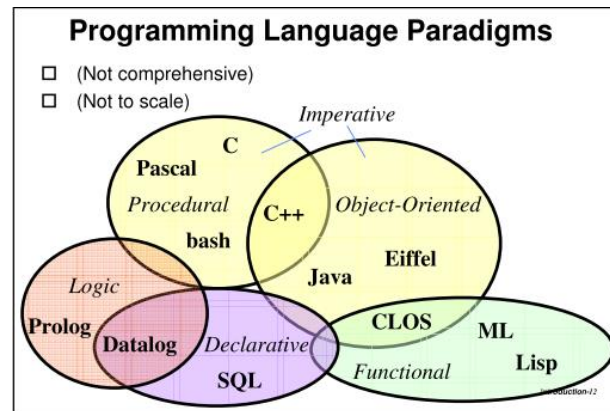
# What is Object-Oriented Programming?

# What is Object-Oriented Programming?



## First of all: It is a programming paradigm!

- A programming paradigm can be understood as a **style**
- There are **many languages** and **several paradigms** out there!
- For instance **procedural**, **declarative** or **functional**
- ... and of course: object-oriented programming (**OOP**), e.g. Java or C++
- Many languages support more then one paradigm (Java too!)

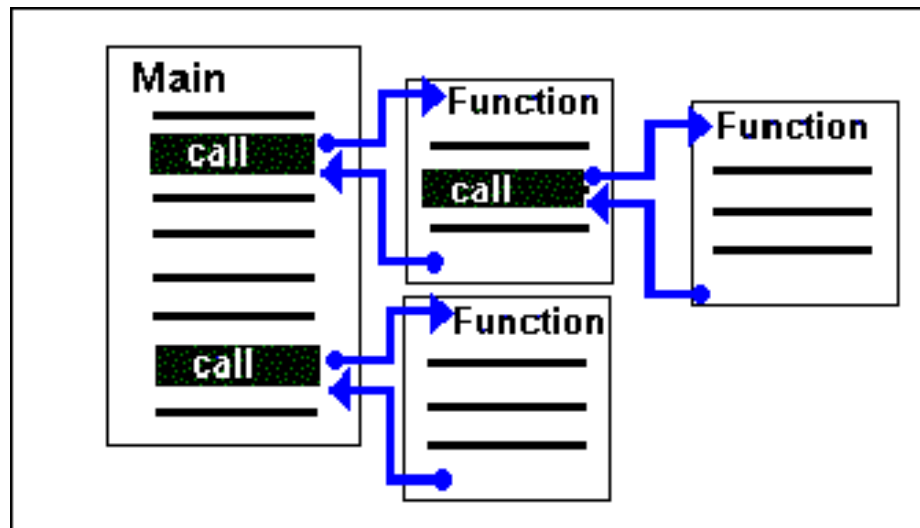


# What is Object-Oriented Programming?

---

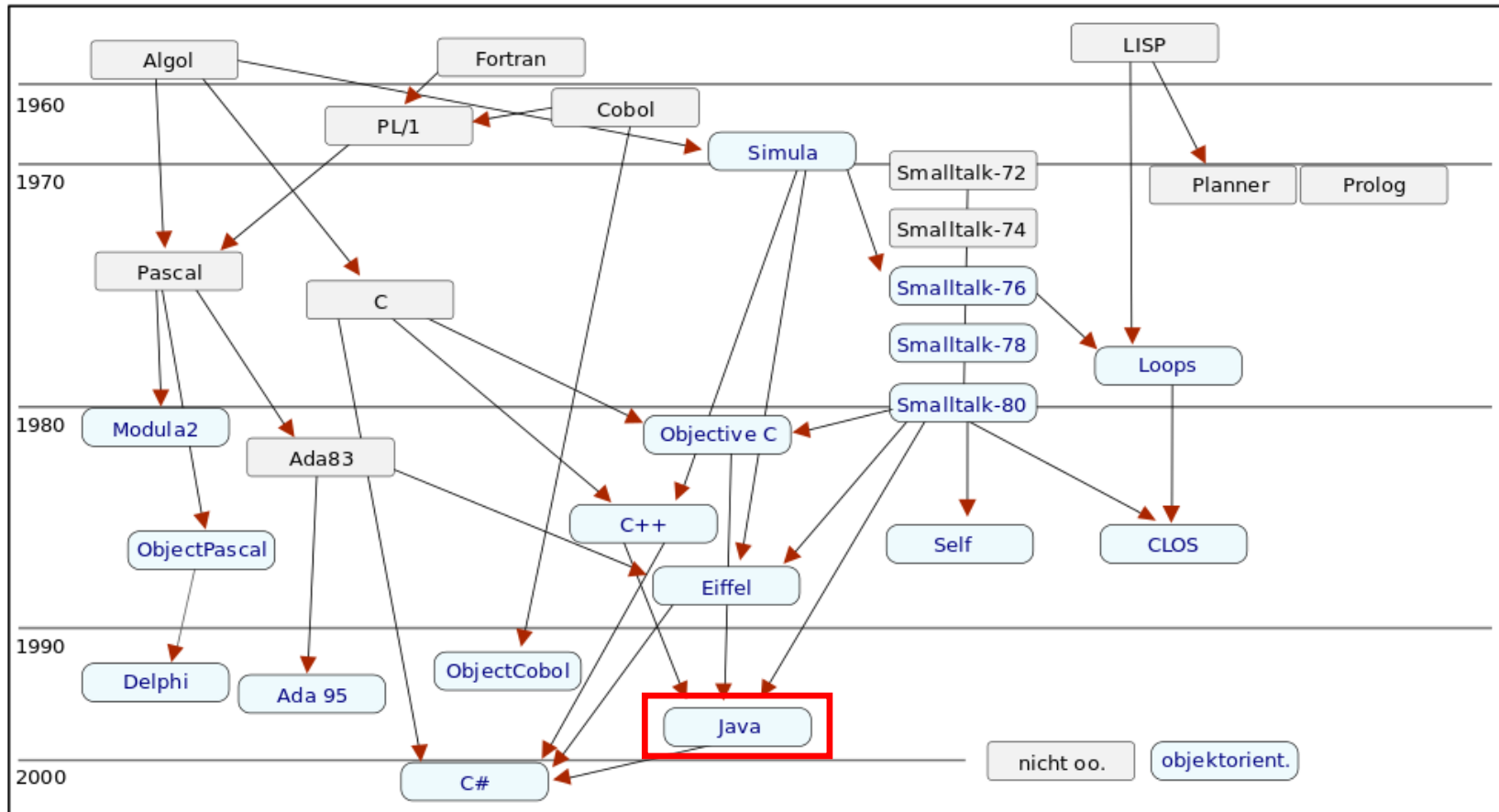
## Excuse: Procedural programming

- **Core concept: Procedure calls**
- Procedures: routines, subroutines or functions
- Contain a series of computational steps to be carried out
- Any procedure can be called during the program's execution
- Including other procedures or itself
- **Examples: C or Pascal**



# What is Object-Oriented Programming?

## Excuse: Brief history of OOP languages



# What is Object-Oriented Programming?

---

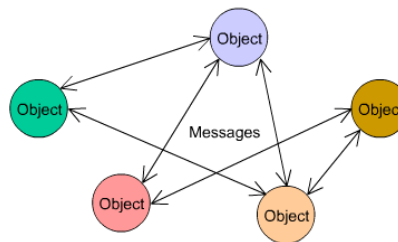
## The core concept of OOP is the object!

Real-world objects have **two characteristics**:

1. **State**, e.g. a bicycle could have several states (e.g. gear, speed ...)
2. **Behavior**, e.g. a bicycle could do things and behave differently (e.g. applying breaks)

## Motivation for using OOP

- Software objects are **conceptually similar to real-world objects**
- They store their **state** in **fields** and **expose** their **behavior** through **methods**
- Objects **communicate** with each other by passing “**messages**”



Interaction of objects via message passing

# What is Object-Oriented Programming?

---

## Motivation for using OOP

- **Modularity:** Source code can be written and maintained independently
- **Information-Hiding:** Internal implementation remains hidden from the outside
- **Code Re-use:** If an object already exists this object can be used by you

## OOP vs Procedural Programming

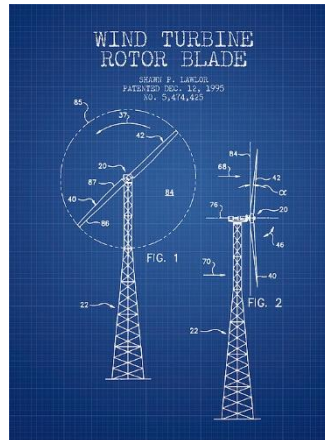
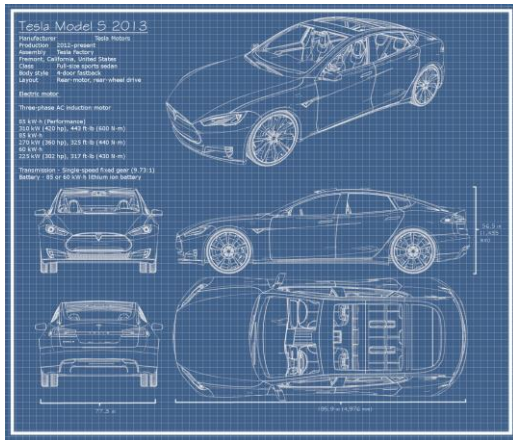
- Procedural programming uses **procedures** to operate on data structures
- ... while in OOP they are bundled together
- An **object** operates on its **own** data structures!
- You can also use the procedural paradigm in Java!



# What is Object-Oriented Programming?

## What are classes?

- A template, a blueprint - it's **not** a concrete realization of something!
- Think of a **concept**!
- And **you need a class (the blueprint) before you can create an object**





# What is Object-Oriented Programming?

---

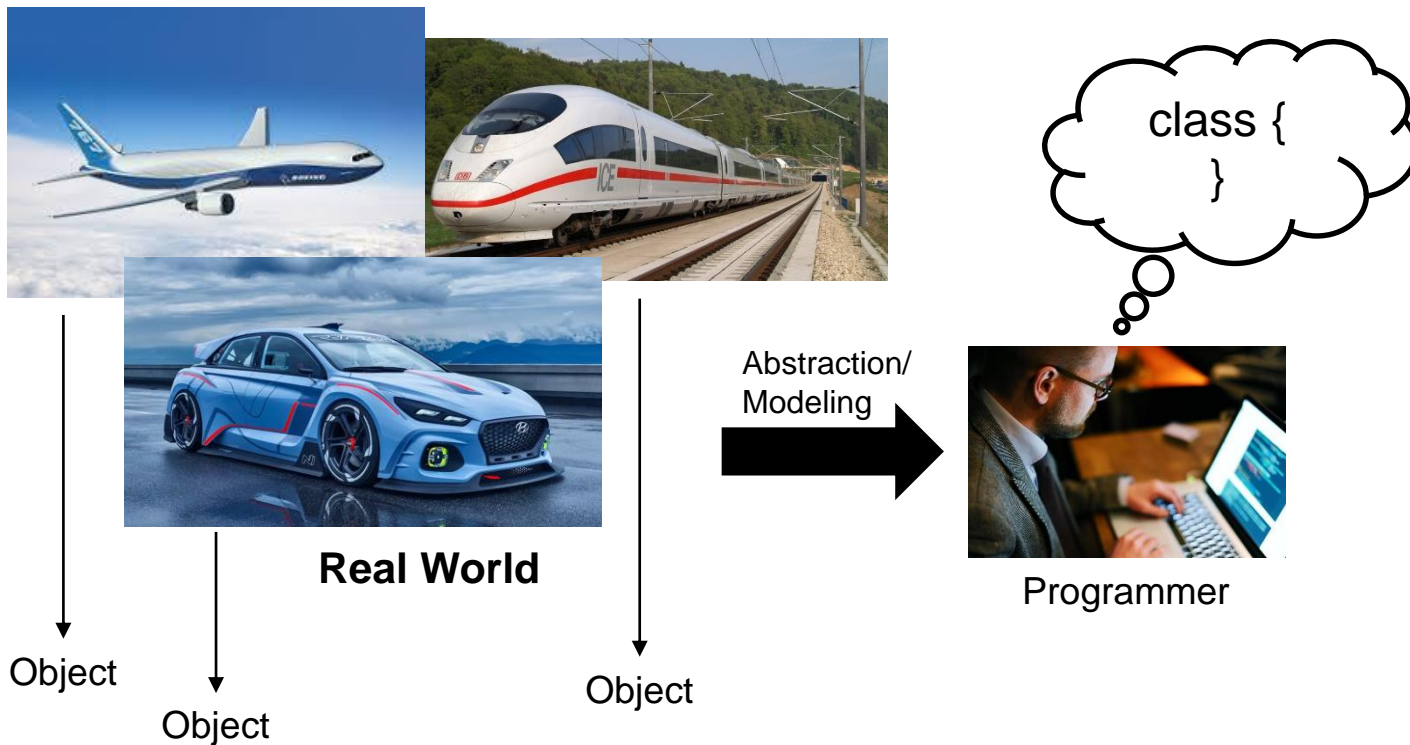
## Terminology

- **Class:** Defines what an object of this class knows (state) and does (behavior)
- **Object:** An instance of a class (a concrete thing create from the template / blueprint)
- **Instance Variables:** They represent what an object knows (**the state**)
- **Methods:** They represent what an object can do (**the behavior**)

## Difference between a class and object

- A **class** is **not an object**!
- A **class** represents a **blueprint for an object**
- It tells us (or the JVM) how to make objects of a particular class
- Each object made of a class has its own states

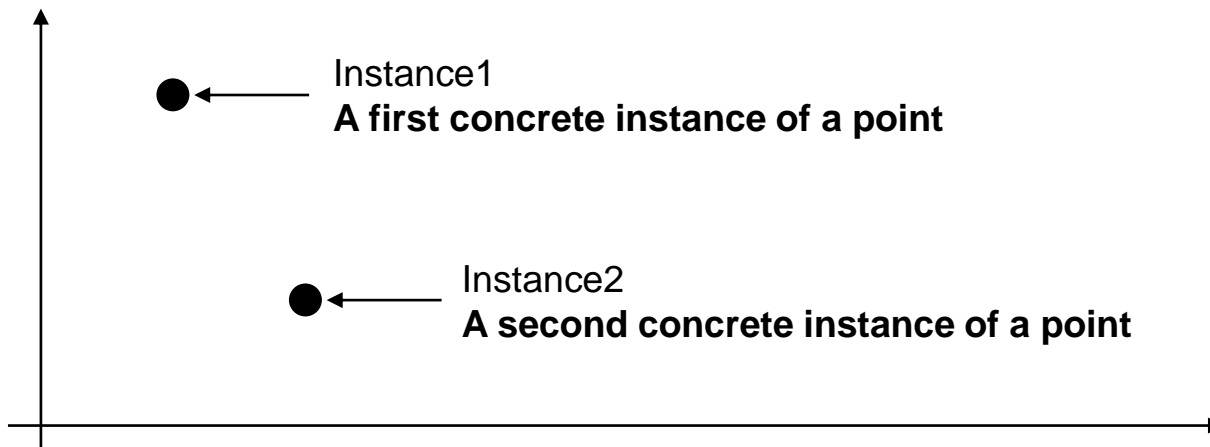
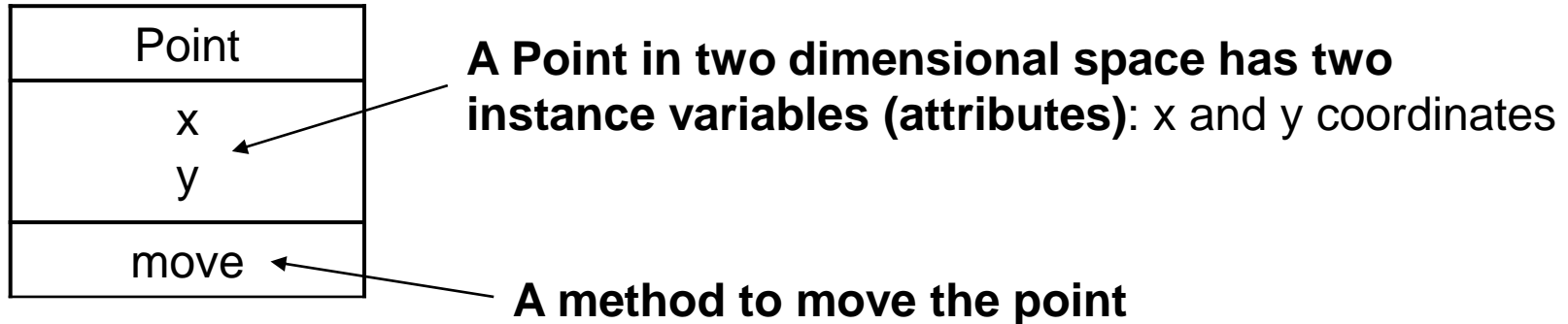
# What is Object-Oriented Programming?



If successful, this medium of expression (the object-oriented way) will be significantly easier, more flexible, and efficient than the alternatives as problems grow larger and more complex.

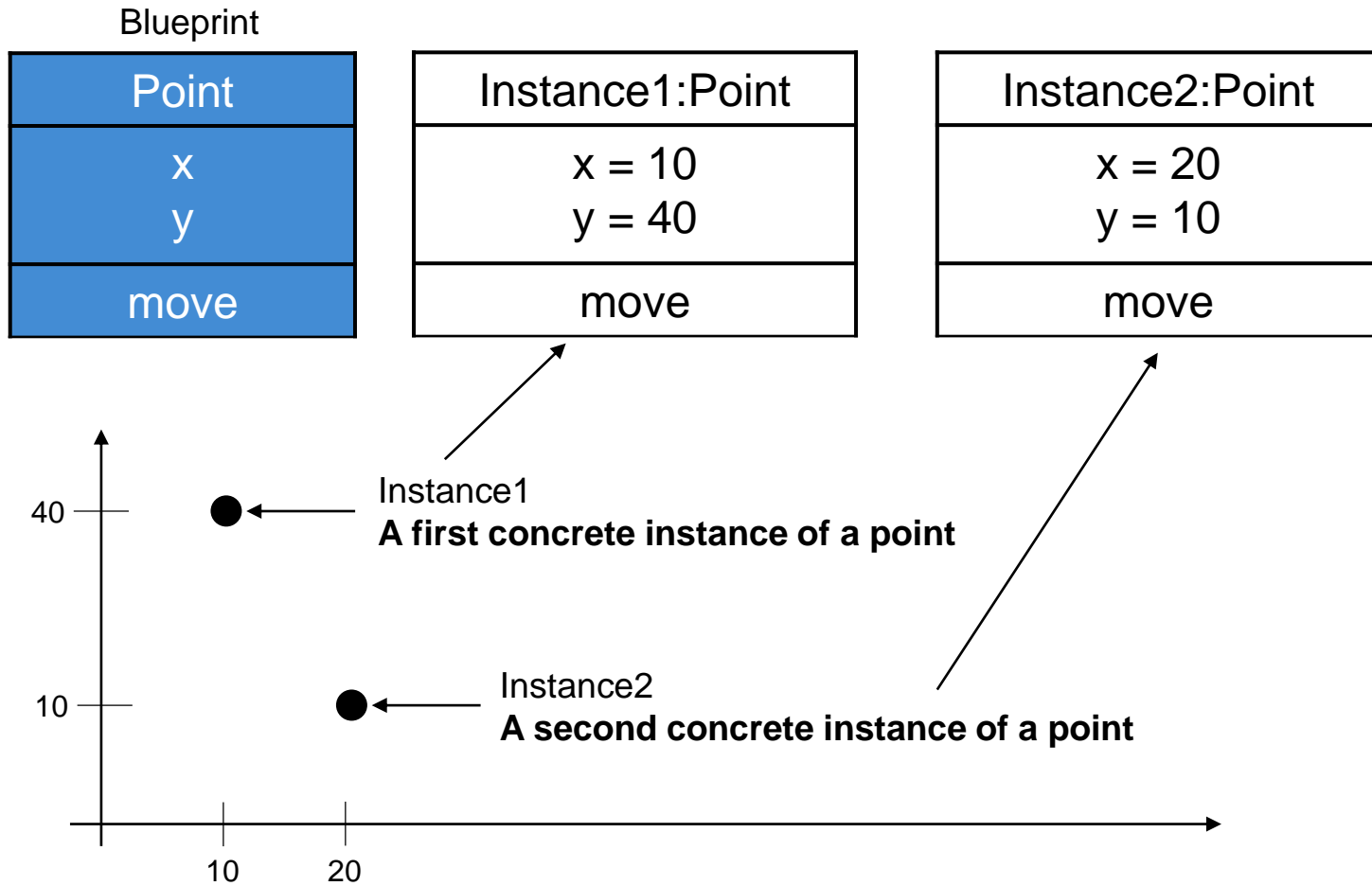
# What is Object-Oriented Programming?

## Example: Classes and Objects



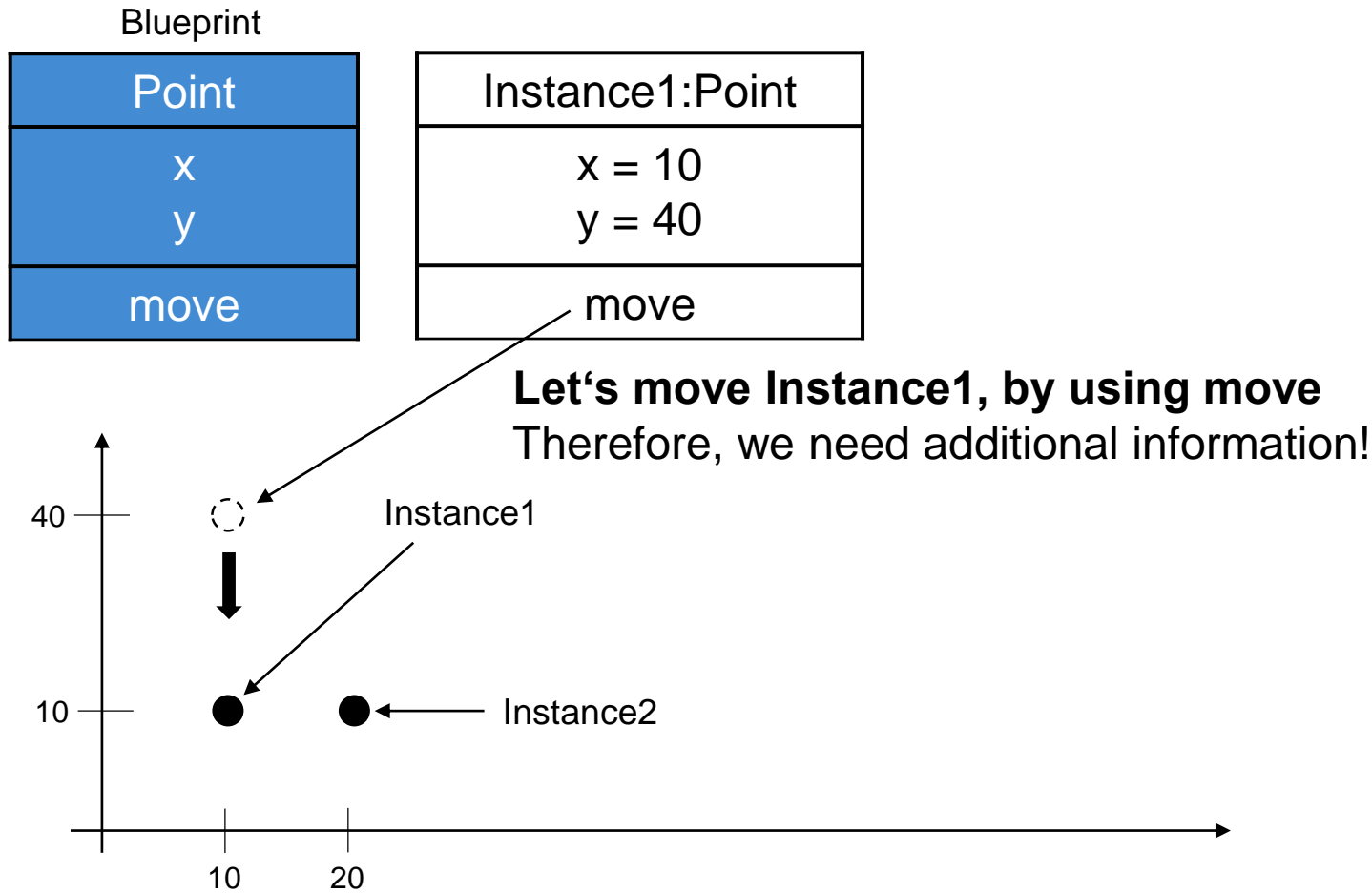
# What is Object-Oriented Programming?

## Example: Classes and Objects



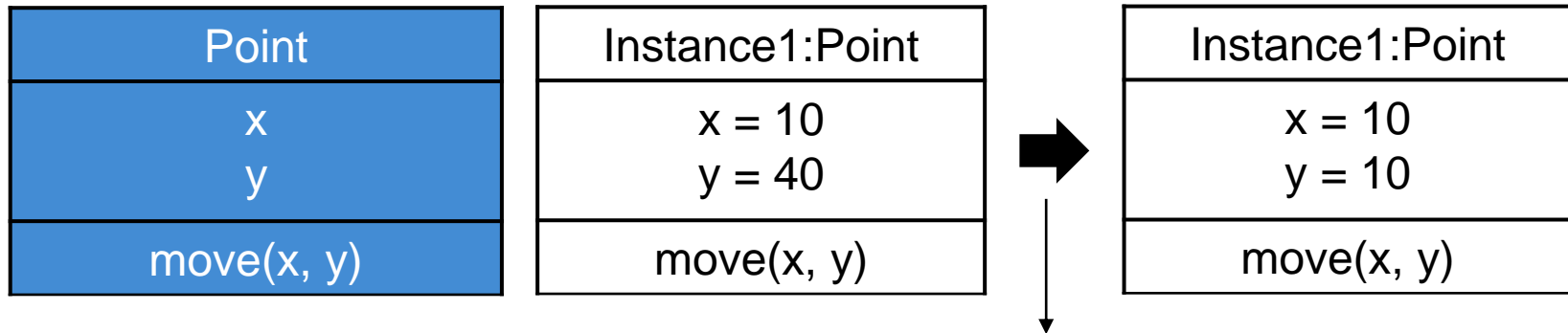
# What is Object-Oriented Programming?

## Example: Classes and Objects



# What is Object-Oriented Programming?

## Example: Classes and Objects



**Concrete call:** move(0, -30)



# What is Object-Oriented Programming?

---

## Example: Classes and Objects

Point
x y
move(x, y)

**We did not specify the type of the attributes and parameters, yet!**



Point
x:int y:int
move(x:int, y:int)

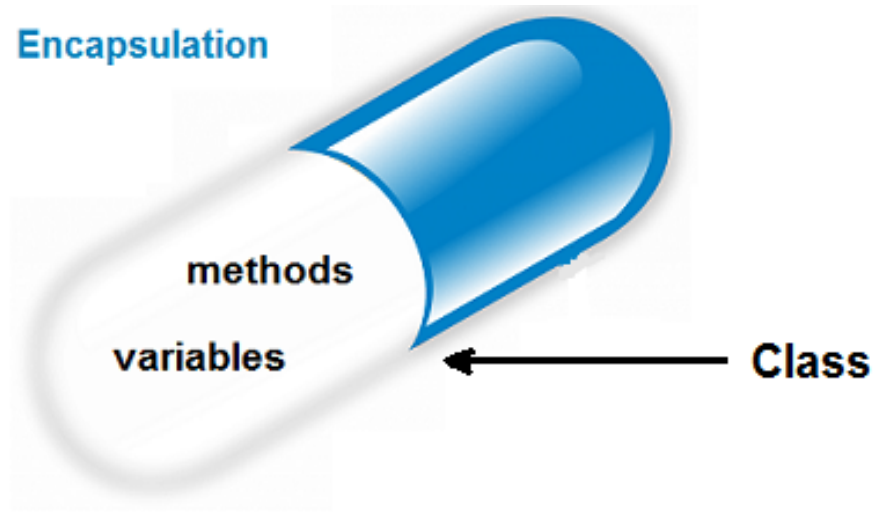
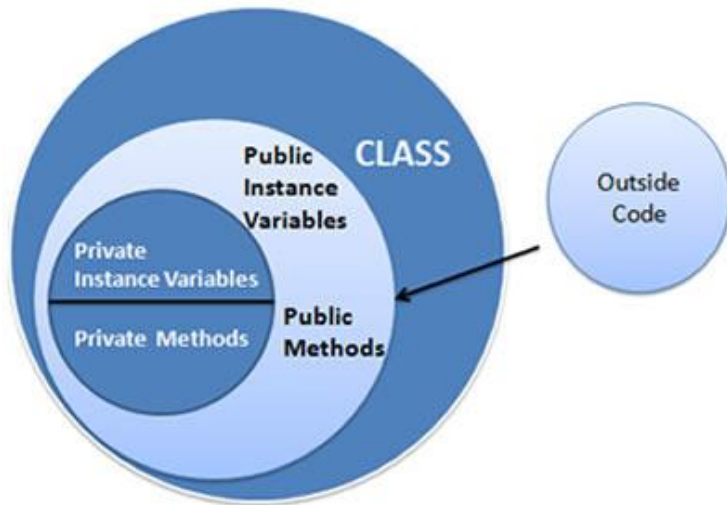
**Now we have a blueprint to create hundreds and thousand of points**



# What is Object-Oriented Programming?

## Core Concept I: Data encapsulation

- Encapsulation is the bundling of data with the methods that operate on them
- Remember that in procedural programming this is not the case!
- Also used for hiding the internals of the object from outside view
- Only the object's own methods can operate on it's data
- Protects an object's integrity!

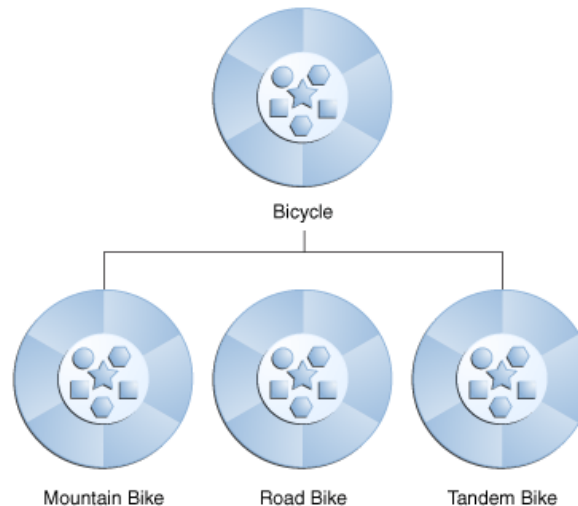


# What is Object-Oriented Programming?

---

## Core Concept II: Inheritance

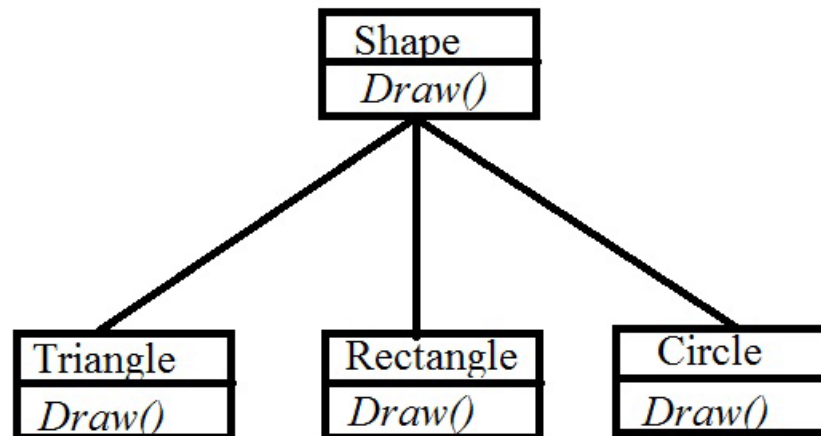
- Different kinds of objects have a certain amount in common with each other ...
- E.g. mountain bikes and tandems all share the characteristics of bicycles
- ... yet they each define additional features that make them different
- OOP allows classes to inherit commonly state and behavior from other classes:
- `Bicycle` can be a **superclass** of the **subclasses** `MountainBike` and `Tandem`



# What is Object-Oriented Programming?

## Core Concept III: Polymorphism

- It's a principle from biology
- An organism can have many different forms or stages
- Poly: many (e.g. polygon); Morph: form (e.g. morphology)
- In OOP it allows to provide a single interface to varying entities of the same type



# What is Object-Oriented Programming?

## Core Concept IV: Communication between objects

- Objects can communicate with each other
- ... by passing messages!
- One object can get another object to do something
- ... through method calls!
- Call a method and pass it some arguments (i.e. messages) or
- Get something from a method through its returned value.

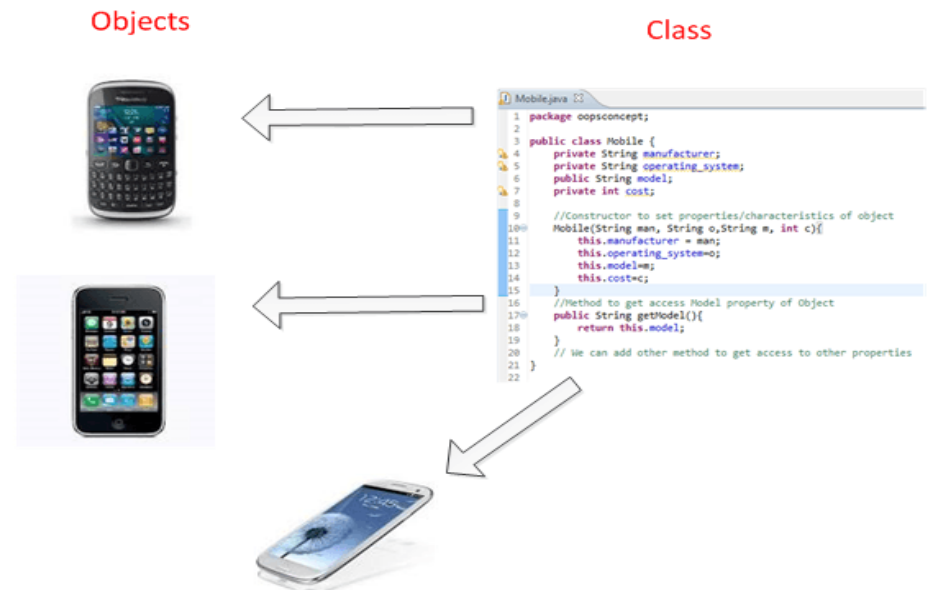
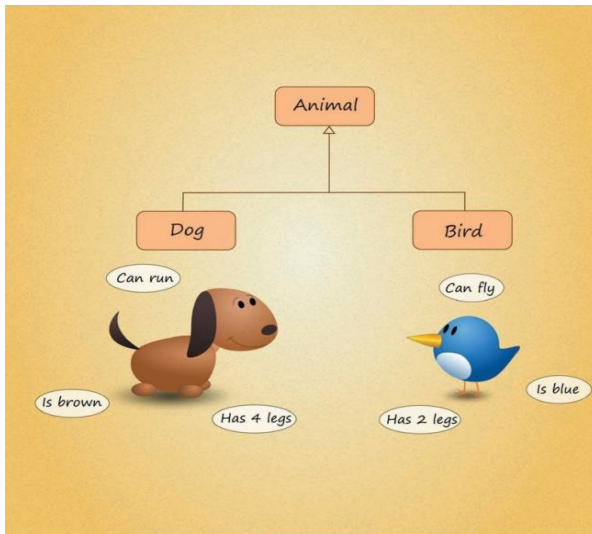


Methods will be covered in detail later this lecture!

# OOP in Java

## Introduction

- In lecture 1 we have already met a class definition
- How to create objects from it?
- Java supports **inheritance**, **encapsulation**, **polymorphism** and **message passing**



## Creating a class I: A simple example

```
public class Point {
```

```
    public int x;
```

```
    public int y;
```

Instance variables  
(or fields)

```
    public void move(int delta_x, int delta_y) {
```

```
        x = x + delta_x;
```

```
        y = y + delta_y;
```

```
    }
```

```
}
```

A method



## Creating a class II: A simple example

```
public class Application {  
  
    public static void main (String[] args) {  
  
        Point p1 = new Point();  
  
        p1.x = 10;  
        p1.y = 40;  
  
        p1.move(0, -30);  
    }  
}
```

Create a Point object, i.e. create an instance

Set the coordinates of p1

Call move-method of p1

## Creating a class III: A simple example

```
public class Application {  
  
    public static void main (String[] args) {  
  
        Point p1 = new Point();  
  
        p1.x = 10;  
        p1.y = 40;  
  
        p1.move(0, -30);  
    }  
}
```

Data type  
of p1

Create **new**  
object of the  
specified type



The `new` keyword is used for creating instances of classes

## Creating a class IV: A simple example

```
public class Application {  
  
    public static void main (String[] args) {  
  
        Point p1 = new Point();  
  
        p1.x = 10;  
        p1.y = 40;  
  
        p1.move(0, -30);  
    }  
}
```



Use the **dot operator** to access public instance variables or methods

## Inheritance I

- In Java classes can be derived from other classes ...
- Thereby inheriting fields and methods from those classes
- By using the keyword `extends`

## Terminology

- **Subclass:** A class derived from an other class
- Subclasses are also know as derived, extended or child classes
- **Superclass:** The class from which the subclass is derived
- Superclasses are also know as base or parent classes



Classes can be derived from classes that are derived from ...

## Inheritance II

- Every class has one and only one direct superclass
- Except `Object` which has no superclass (see next slide)
- If no other superclass is given every class is implicitly a subclass of `Object`

## Inheritance III

- Charming idea: Reuse existing classes with desired functionality
- ... by inheriting from them!
- A subclass inherits all members (fields, methods) from it's superclass
- But not its constructors!



For now think of special methods.  
We'll cover constructors shortly!

## Inheritance IV

- A subclass inherits all `public` and `protected` members of its parent
- These members can be used in the subclass or replaced or supplemented
- That is you can use fields and methods of the superclass
- You can declare new fields in the subclass (that are not in the superclass)
- Same goes for methods

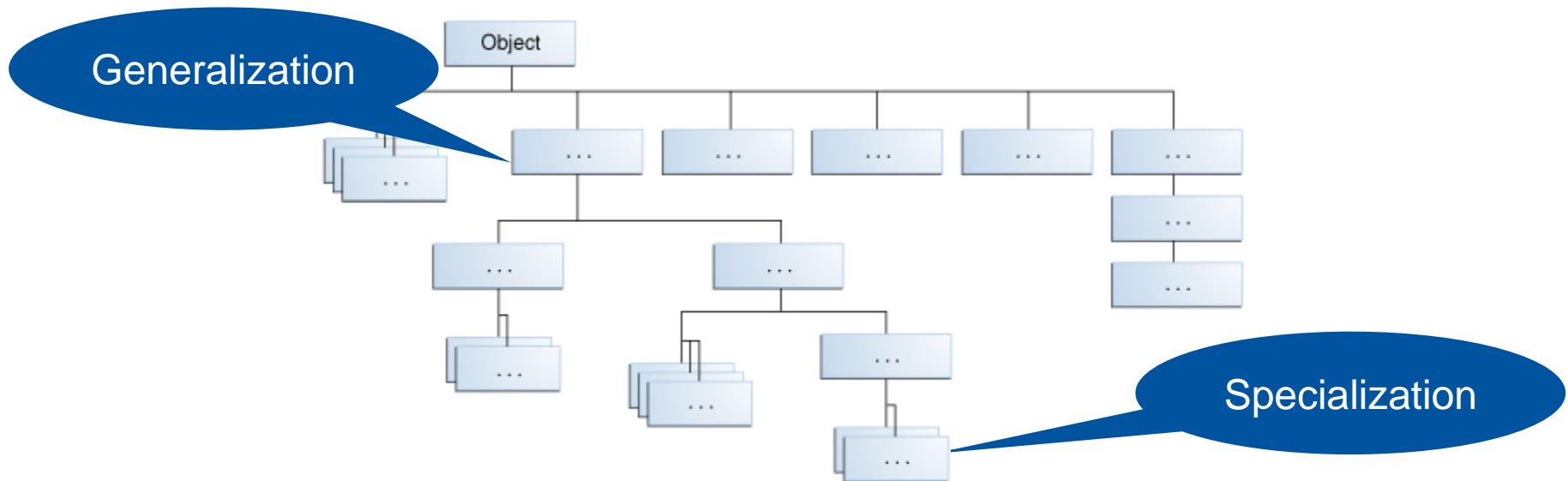


A subclass does not inherit `private` members of its parent

`public`, `protected` and `private` are  
topics of data encapsulation (which will be  
covered shortly)

## Inheritance V: The class object

- It's on the top of the Java class hierarchy
- It's the most general of all Java classes
- Defines and implements behavior common to all classes



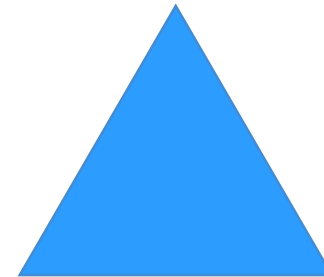


## Inheritance: Example

```
public class Shape {  
  
    public String color;  
  
    public boolean filled;  
  
}
```



This **Rectangle** is a **Shape**  
with color "red" and not filled



This **Triangle** is a **Shape**  
with color "blue" and filled

## Inheritance: Example

```
public class Rectangle extends Shape {  
  
    public int x1, y1, x2, y2;  
  
    public int calculateArea() {  
        return Math.abs(x2 - x1) * Math.abs(y2 - y1);  
    }  
  
}
```

Rectangle is a  
subclass of Shape

It extends its superclass  
by a method and four  
attributes

## Inheritance: Example

Triangle is  
another subclass  
of Shape

```
public class Triangle extends Shape {

    public int x1, y1, x2, y2, x3, y3;

    public double getSideA() {
        return Math.sqrt(Math.pow(x2 - x1, 2.0) + Math.pow(y2 - y1, 2.0));
    }

    // similar methods to calculate side B and side C
    ...

    public double calculateArea() {

        // calculate area by Heron's formula
        double s = 0.5 * (getSideA() + getSideB() + getSideC());
        double area = Math.sqrt(s * (s - getSideA()) * (s - getSideB()) * (s -
            getSideC()));
        return area;
    }
}
```

## Inheritance: Example

```
public class ShapeApplication {  
  
    public static void main(String[] args) {  
  
        Rectangle rectangle = new Rectangle();  
        Triangle triangle = new Triangle();  
  
        rectangle.color = "red";  
        triangle.color = "green";  
  
    }  
  
}
```



Each Triangle and  
Rectangle is a  
Shape

## Inheritance: Example

```
public class ShapeApplication {  
  
    public static void main(String[] args) {  
        Rectangle rectangle = new Rectangle();  
        rectangle.color = "red";  
        rectangle.x1 = 0;  
        rectangle.x2 = 10;  
        rectangle.y1 = 0;  
        rectangle.y2 = 5;  
        System.out.println("Area of rectangle: " +  
            rectangle.calculateArea());  
    }  
}
```

## Inheritance: Example

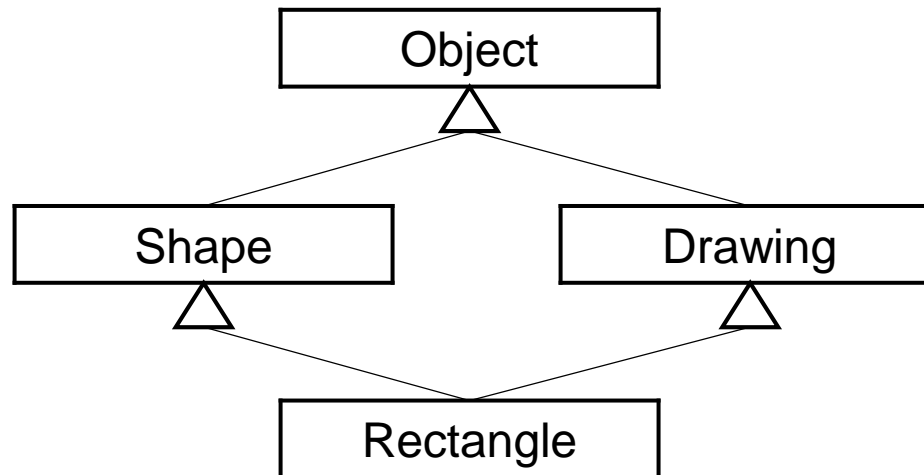
```
public class ShapeApplication {  
  
    public static void main(String[] args) {  
        Triangle triangle = new Triangle();  
        triangle.color = "green";  
        triangle.x1 = 0;  
        triangle.x2 = 0;  
        triangle.x3 = 10;  
        triangle.y1 = 0;  
        triangle.y2 = 5;  
        triangle.y3 = 0;  
        System.out.println("Area of triangle: " +  
            triangle.calculateArea());  
    }  
}
```

## Inheritance VI: Inheriting from multiple classes?

```
public class Rectangle extends Shape extends Drawing {
```



That's not allowed in Java! *“Deadly Diamond of Death”*





## Data encapsulation in Java

- Remember: Encapsulation wraps data and code together as a single unit and
- The variables of a class will be hidden from other classes and
- Can be accessed only through the methods of their current class
- In Java: Declare the variables of a class as `private`
- And provide `public` methods to modify and view the variable values
- **Again: Encapsulation protects an object's integrity!**

## Data encapsulation: Tips on choosing the access level

- Use the most restrictive access level that makes sense for a particular member
- Use `private` unless you have a good reason not to
- Avoid `public` fields except for constants

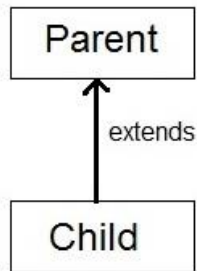
## Data encapsulation: Access modifiers

- Java has access modifiers for controlling access to members of a class
- There are two level of access control
- At the top/class level: `public`, or package-private (no explicit modifier)
- At the member level: `public`, `private`, `protected`, or package-private

Modifier	Class	Package	Subclass	World
<code>public</code>	Y	Y	Y	Y
<code>protected</code>	Y	Y	Y	N
<code>no modifier</code>	Y	Y	N	N
<code>private</code>	Y	N	N	N

## Polymorphism in Java

- Polymorphism: the reference type can be a superclass of the actual object type!
- **Anything** that extends the declared reference variable type can be assigned ...
- ... to the reference variable, but **not** the other way round (**Downcasting!**)
- You can have also have polymorphic arguments (and return types) for methods



Parent p = new Parent( );

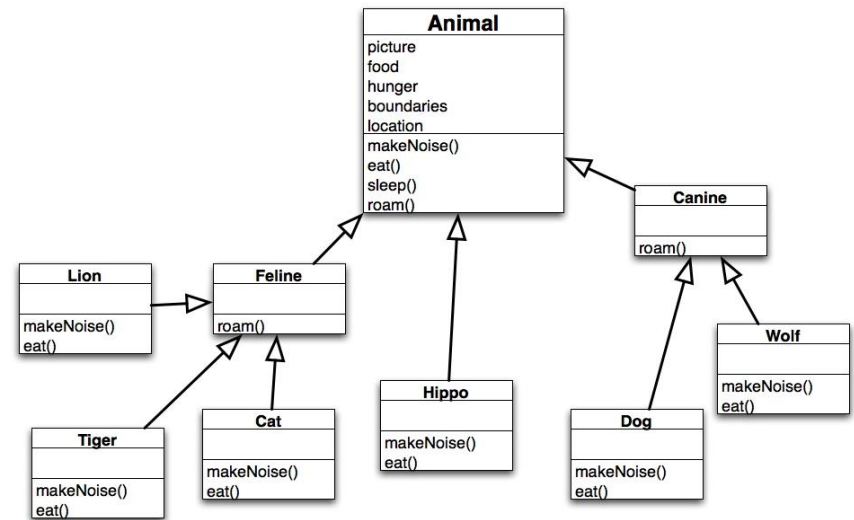
Child c = new Child( );

Parent p = new Child( );

Upcasting

~~Child c = new Parent( );~~

incompatible type



## Polymorphism: Example

```
public class ShapeApplication {  
  
    public static void main(String[] args) {  
  
        Shape shape1 = new Rectangle();  
        Shape shape2 = new Triangle();  
  
        shape1.color = "red";  
        shape2.color = "green";  
  
    }  
}
```

Each  
Rectangle is a  
Shape

Also each  
Triangle is a  
Shape

Hence we can store  
them in a variables of  
type Shape

## Polymorphism: Example

```
public class ShapeApplication {  
  
    public static void main(String[] args) {  
  
        Shape shape1 = new Rectangle();  
        Shape shape2 = new Triangle();  
  
        shape1.color = "red";  
        shape2.color = "green";  
  
        shape1.calculateArea();  
    }  
}
```

But we only see the  
instance variables and  
methods of a Shape

Hence, you cannot  
call the method  
calculateArea


## Polymorphism: Example

```
public class ShapeApplication {  
    public static void main(String[] args) {  
        Shape shape1 = new Rectangle();  
        Shape shape2 = new Triangle();  
        Rectangle rectangle = (Rectangle) shape1;  
        rectangle.x1 = 0;  
        rectangle.x2 = 10;  
        rectangle.y1 = 0;  
        rectangle.y2 = 5;  
        System.out.println("Area of rectangle: " +  
            rectangle.calculateArea());  
    }  
}
```

Since we know shape1 contains a Rectangle, we can use explicit casts


## Polymorphism: Example

```
public class ShapeApplication {  
  
    public static void main(String[] args) {  
        Shape shape1 = new Rectangle();  
        Shape shape2 = new Triangle();  
        Triangle triangle = (Triangle) shape2;  
        triangle.x1 = 0;  
        triangle.x2 = 0;  
        triangle.x3 = 10;  
        triangle.y1 = 0;  
        triangle.y2 = 5;  
        triangle.y3 = 0;  
        System.out.println("Area of triangle: " +  
            triangle.calculateArea());  
    }  
}
```



## Polymorphism: Arguments

```
public class ShapeDrawer{  
  
    public void draw(Shape s) {  
        // some fancy code to draw shapes  
    }  
  
}
```



Any subclass of  
Shape allowed



With polymorphism you can write very flexible code!

The above example will work with any new subclass of Shape, e.g. class Triangle!



# Methods: A Closer Look

## Overview

- **Methods:** They represent what an object does (**the behavior**)
- Methods use instance variables
- They can have **parameters**
- They must have an **return type** (which can be void)

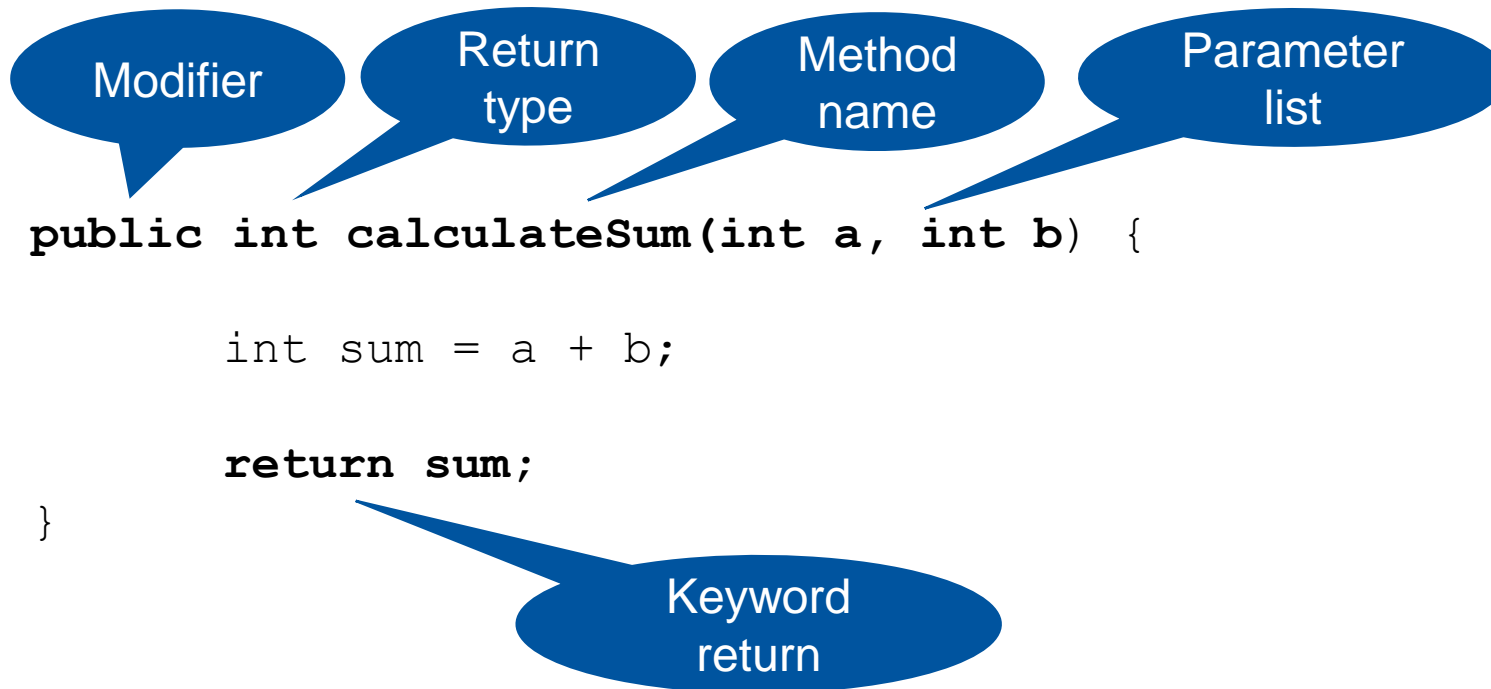
## A short example (for a void method with no parameters)

```
public void printHelloAachen() {  
  
    System.out.println("Hello Aachen");  
  
}
```

# Methods: A Closer Look

---

## A more complex example (and terminology)



## Method declarations have six components

- **Modifiers:** such as `public`, `private` and others
- **Return type:** data type of the value returned by the method (or `void` if no return)
- **Method names:** see below
- **Parameter list:** comma-delimited list of input parameters, preceded by data type
- **Exception list:** discussed in lecture 3
- **Method body:** methods code including local variables

## Conventions for naming methods

- Names should be a verb in lowercase...
- ... or a multi-word name that begins with a verb in lowercase
- Examples: `run`, `runFast`, `isEmpty`, `getFinalData`, `setEngineSpeed`



Java is always *pass-by-value*!

## Method signatures

- The method's name and the parameter types form the *signature*
- **Example:** `calculateSum(int, int)`
- The return type is not part of the signature

## Overloading methods

- Java can distinguish between methods with different signatures
- Methods within a class can have the same name
- But only if they have different parameter lists!
- They are differentiated by the number and type of the arguments passed to them
- You can not declare two methods with same signature but different return type!
- **Examples:** `draw(String s)`, `draw(int i)`, `draw(int i, double f)`

## Excuse: Getters and Setters

- Are ordinary methods, i.e. they take parameters and return a value
- They let you get and set things, mostly instance variables
- A Getter sends back the value of whatever is supposed to get
- A Setter takes an argument and uses it to set the value of an instance variable

## Example

```
public class Shape {  
    private String color;  
    public String getColor() {  
        return color;  
    }  
    public void setColor(String newColor) {  
        color = newColor;  
    }  
}
```

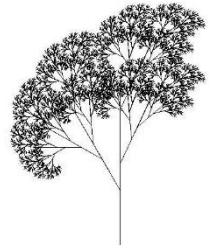


Remember  
encapsulation? 😊

## Recursive Calls

- It's a concept which allows a method to call itself!
- Remember the calculation of the factorial? Now we'll use recursion!

$$\text{fact}(n) = \begin{cases} 1 & \text{if } n = 0 \\ n \cdot \text{fact}(n - 1) & \text{if } n > 0 \end{cases}$$



## Example

```
public int factorial(int n)
{
    if (n == 0)
        return 1;
    else
        return n * factorial(n-1);
}
```

# Constructors



## Constructors I

- A class contains *constructors* that are invoked to create objects
- They are there to instantiate a class!
- Constructors look similar to methods ...
- ... except that they use the same name as the class and have no return type!

## Example

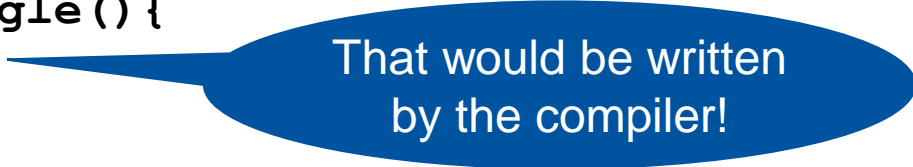
```
public class Rectangle extends Shape {  
  
    public Rectangle(int aX1, int aY1, int aX2, int aY2) {  
        x1 = aX1;  
        y1 = aY1;  
        ...  
    }  
}
```

## Constructors II

- Constructors run *before* the object can be assigned to a reference!
- It runs every time you invoke `new`
- If you don't write a constructor for your class the compiler writes one for you ...
- ... which is called the **default constructor**!

## Example

```
public class Rectangle extends Shape {  
  
    public Rectangle() {  
  
    }  
  
}
```



That would be written  
by the compiler!

## Constructors III

- **Constructors are not inherited by the subclass!**
- You can have multiple constructors in your class
- That's called **constructor overloading!**
- Each constructor must have a different parameter list!

But we'll learn  
super in lec. 3!

## Example

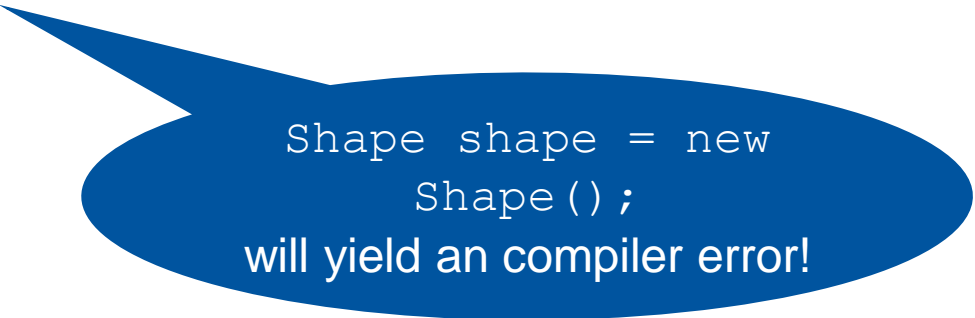
```
public class Rectangle {  
  
    public Rectangle() {}  
    public Rectangle(int x1, int y1, int x2, int y2) {...}  
    public Rectangle(int x1, int y1, int x2, int y2, String  
        color) {...}  
  
    ...  
}
```

## Abstract Classes

- Some classes should not be instantiated!
- Think of the Shape class ... it is just an abstract definition of shapes!
- You can prevent class from being instantiated by marking them `abstract`!
- The opposite to abstract classes are concrete classes!
- **Generally, abstract classes are used for polymorphism (or for inheritance)**

## Example

```
public abstract class Shape {  
    ...  
}
```



Shape shape = new  
 Shape();  
will yield an compiler error!

## Abstract Methods

- You can mark methods `abstract`, too!
- **If you declare a method abstract the class must be abstract as well!**
- An abstract method must be overridden in a concrete subclass!
- An abstract method has no body: just end the declaration with a semicolon!

## Example

```
public abstract class Shape {  
  
    public abstract double calculateArea();  
    public abstract double calculatePerimeter();  
  
}
```

## Interfaces

- Sometimes it's necessary for **programmers to agree on a contract** ...
- Generally speaking, **interfaces** are such **contracts**!
- In Java an interface is a reference type!
- **An interface defines only abstract methods!**
- An interface is created using the keyword `interface`!
- A class *implements* an interface using the keyword `implements`!
- A class can **implement multiple interfaces**!



## Interfaces: Example

```
public interface Drawable {  
  
    public abstract void draw();  
    public abstract void rotate();  
  
}
```

## Interfaces: Example

```
public class Rectangle implements Drawable {  
  
    private String color;  
  
    ...  
  
    // Implement this method! It's the contract!  
    public void draw() {...}  
  
    // Implement this method! It's the contract!  
    public void rotate() {...}  
  
}
```



## Class vs subclass vs abstract class vs interface

- New class (that doesn't extend anything): if there's nothing to meaningful extend
- Subclass: If a more specific version of an existing class is needed
- Abstract class: If nobody should make objects of the class (e.g. it's a template)
- Interface: For defining a contract that other classes must fulfill!



# Thank you very much!