Programming in Java – Part 02
OOP in Java

Paolo Vercesi
ESTECO SpA

Agenda

- OOP
- Extension and inheritance
- Access control and encapsulation
- Interfaces and polymorphism
- The Three Musketeers of OOP

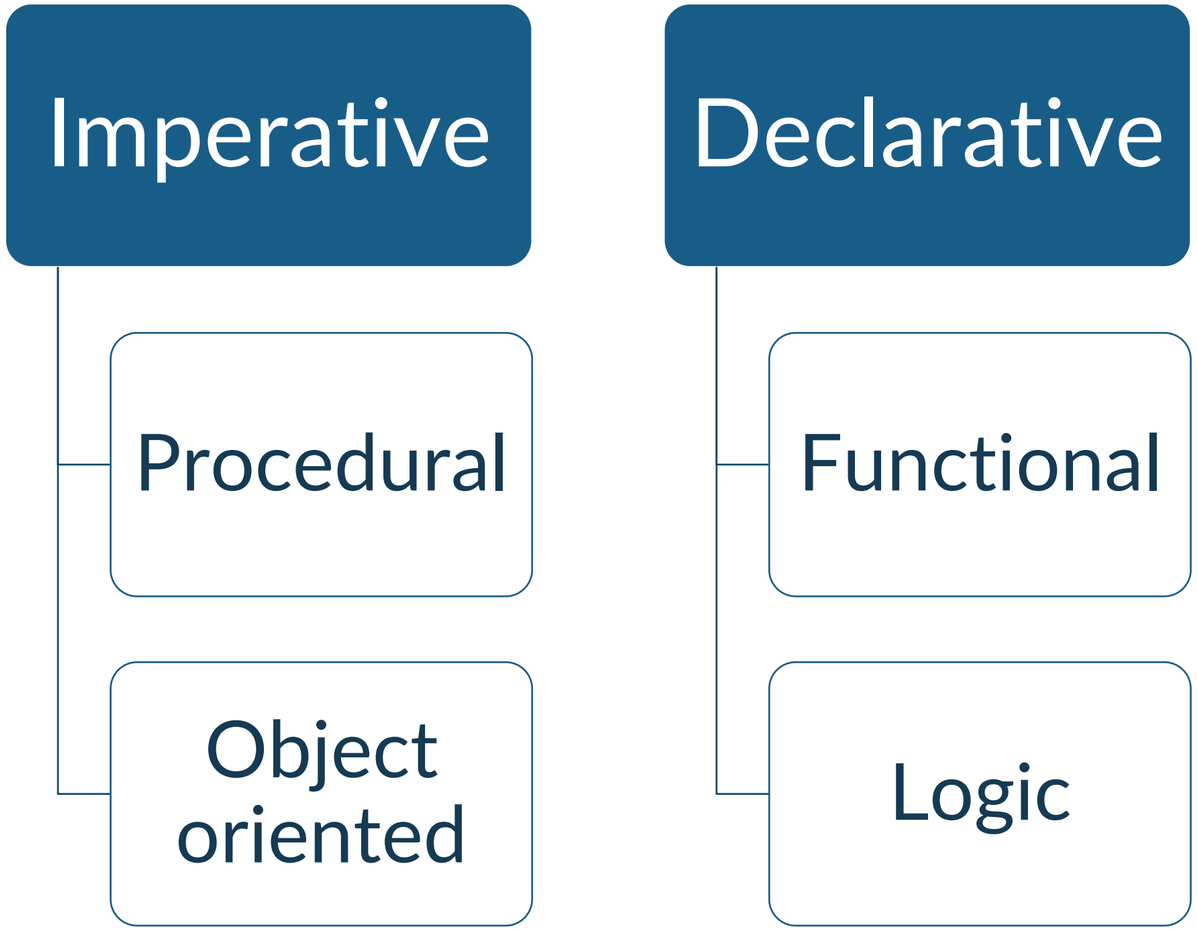
OOP

Object-oriented programming

Object-oriented programming is a programming paradigm

“A programming paradigm is a way of conceptualizing what it means to perform computation and how tasks to be carried out on a computer should be structured and organized”

Programming paradigms



Procedural paradigm

Emphasizes the use of procedures, functions, and modules to structure and organize code

A program is divided into a series of reusable procedures, each of which performs a specific task

Procedures are typically called in a sequence to accomplish a larger task and they can pass data between each other

Supported by computer processors that provide a stack register and instructions for calling procedures and returning from them

The C programming language is a procedural language, but the procedural paradigm can be also used with other languages like Java or Python

Is a popular paradigm and many people starts to program by using it

Object-oriented paradigm

Code is organized into reusable and self-contained interacting units known as objects

Each object is an instance of a class, and a class defines the state (variables) and behavior (methods) of the objects

Each object implements a responsibility and provides methods that can be used by other objects

OOP encourages modeling real-world entities and their interactions within a program, making it easier to understand, design, and maintain complex software systems

Some programming languages that support OOP are Java, C++, C#, Python, and Typescript

The shift from procedural programming to object-oriented programming can be difficult

Functional paradigm

Focus on immutability, pure functions, and a declarative style of programming

A pure function always produce the same output for the same input, regardless of the program's state

A pure function has no side effects, they don't modify variables outside their scope or perform any observable actions other than returning a value

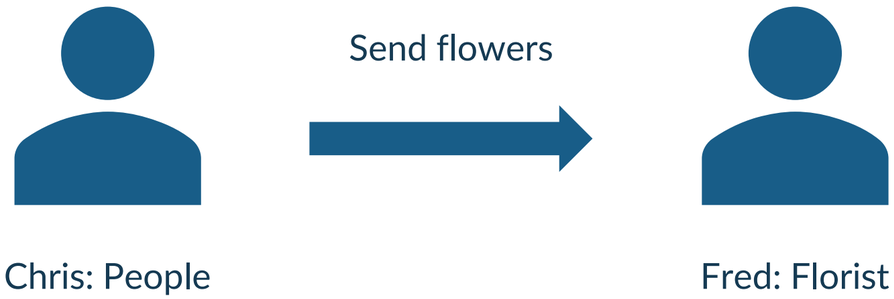
Data is typically immutable, once a data structure is created it cannot be changed, new data structures are created to represent changes, simplifying reasoning about code

Functions are first class citizens, they can be passed as arguments to other functions, returned as values from other functions, and assigned to variables

Functional programming encourages the composition of smaller functions to build more complex functions, to promote code reusability and modularity

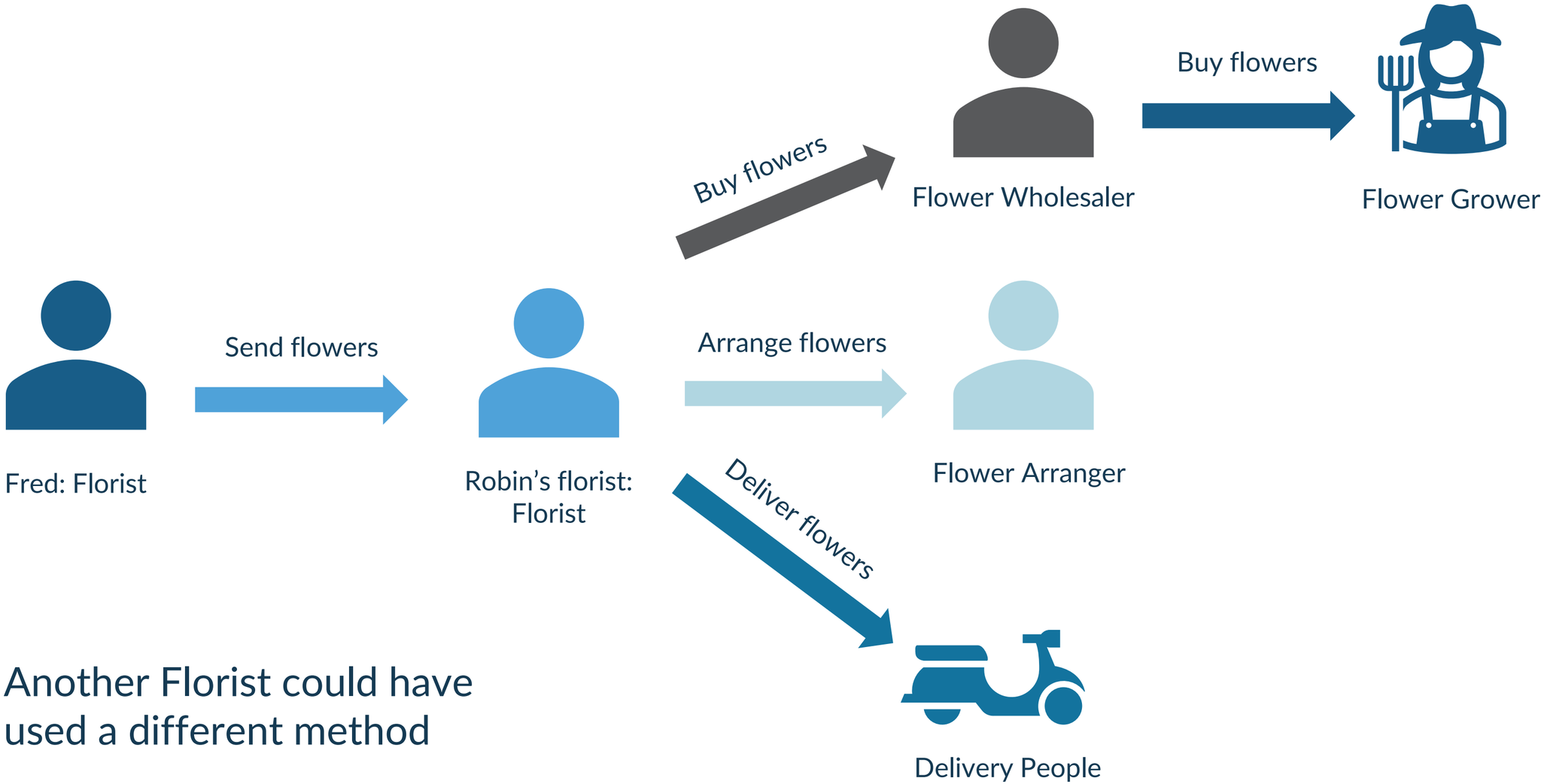
Real-world problem

- Suppose an individual named Chris wishes to send flowers to a friend named Robin, who lives in another city
- Because of the distance, Chris cannot simply pick the flowers and take them to Robin in person
- Chris simply walks to a nearby flower shop, run by a florist named Fred
- Chris will tell Fred the kinds of flowers to send to Robin and the address to which they should be delivered
- Chris can then be assured that the flowers will be delivered expediently and automatically



- Fred has the **responsibility** to send flowers
- Is Chris interested in how it is happening? It doesn't need!

A possible method



Another Florist could have used a different method

Messages and methods in OOP

You initiate an action by the transmitting a message to a receiver object responsible for the action. In other words, you ask the object to do something by invoking a method

The message encodes the request for an action and is accompanied by additional information (method arguments) needed to carry out the request

In response to a message, the receiver will perform some procedure to satisfy the request

Based on the information hiding principle (encapsulation), the client sending the message does not need to know how the receiver is implementing the action, nor it does not need to know anything about other private members

Public methods declare what are the responsibilities of an object

it.units.sdm.oop.florist.messagesandmethods.Person

```
public class Person {

    public static void main(String[] args) {
        Person robin = new Person();
        Florist fred = new Florist(new Wholesaler[]{new Wholesaler(10, 3), new Wholesaler(12, 2)});

        fred.sendFlowersTo(robin);
    }
}
```

it.units.sdm.oop.florist.messagesandmethods.Florist

```
public class Florist {

    private final Wholesaler[] wholesalers;

    public Florist(Wholesaler[] wholesalers) {
        this.wholesalers = wholesalers;
    }

    public void sendFlowersTo(Person person) {
        Flowers flowers = buyFlowers();
        flowers.arrange();
        deliverFlowers(flowers, person);
    }

    private Flowers buyFlowers() {
        return selectWholesaler().buyFlowers();
    }

    private Wholesaler selectWholesaler() {
        Wholesaler cheapest = wholesalers[0];
        for (Wholesaler wholesaler : wholesalers) {
            if (wholesaler.getPrice() < cheaper.getPrice()) {
                cheapest = wholesaler;
            }
        }
        return cheapest;
    }

    […]
}
```

Messages vs procedure calls

How does a message differ from a procedure call?

In both cases, the invoked method initiates a set of well-defined, but we can find two important distinctions

1. in a message there is a designated receiver; in a procedure call there is no designated receiver, we are just invoking a procedure in some module or library. The specific receiver will not be known until run time. Thus, there is late binding between the message (function or procedure name) and the code fragment (method) used to respond to the message. On the contrary, a procedure call is bind/linked much earlier at compile-time or link-time

2. the interpretation of the message, that is, the method used to respond to the message, is determined by the receiver and can vary with different receivers, this point is at the base of polymorphism

it.units.sdm.oop.florist.procedural.Person

```
public class Person {

    private static void sendFlowersTo(Florist florist, Person person) {
        Flowers flowers = buyFlowers(florist);
        arrange(flowers);
        deliverFlowers(flowers, person);
    }

    private static Flowers buyFlowers(Florist florist) {
        Wholesaler wholesaler = selectWholesaler(florist.getWholesalers());
        return wholesaler.buyFlowers();
    }

    private static Wholesaler selectWholesaler(Wholesaler[] wholesalers) {
        Wholesaler cheapest = wholesalers[0];
        for (Wholesaler wholesaler : wholesalers) {
            if (wholesaler.getPrice() < cheaper.getPrice()) {
                cheapest = wholesaler;
            }
        }
        return cheapest;
    }

    […]

    public static void main(String[] args) {
        Person robin = new Person();
        Florist fred = new Florist(new Wholesaler[]{new Wholesaler(10, 3), new Wholesaler(12, 2)});
        sendFlowersTo(fred, robin);
    }
}
```

it.units.sdm.oop.florist.procedural.Florist

```
public class Florist {

    private final Wholesaler[] wholesalers;

    public Florist(Wholesaler[] wholesalers) {
        this.wholesalers = wholesalers;
    }

    public Wholesaler[] getWholesalers() {
        return wholesalers;
    }
}
```

it.units.sdm.oop.florist.polymorphic.Florist

```
public interface Florist {

    void sendFlowersTo(Person people);

}
```

it.units.sdm.oop.florist.polymorphic.CheapFlorist

publ

it.units.sdm.oop.florist.polymorphic.FastFlorist

```
public class FastFlorist implements Florist {

    private final Wholesaler[] wholesalers;

    public CheapFlorist(Wholesaler[] wholesalers) {
        this.wholesalers = wholesalers;
    }

    public void sendFlowersTo(Person person) {
        Flowers flowers = buyFlowers();
        flowers.arrange();
        deliverFlowers(flowers, person);
    }

    private Flowers buyFlowers() {
        return selectWholesaler().buyFlowers();
    }

    private Wholesaler selectWholesaler() {
        Wholesaler fastest = wholesalers[0];
        for (Wholesaler wholesaler : wholesalers) {
            if (wholesaler.getDeliveryDays() < fastest.getDeliveryDays()) {
                cheapest = wholesaler;
            }
        }
        return fastest;
    }

}
```

}

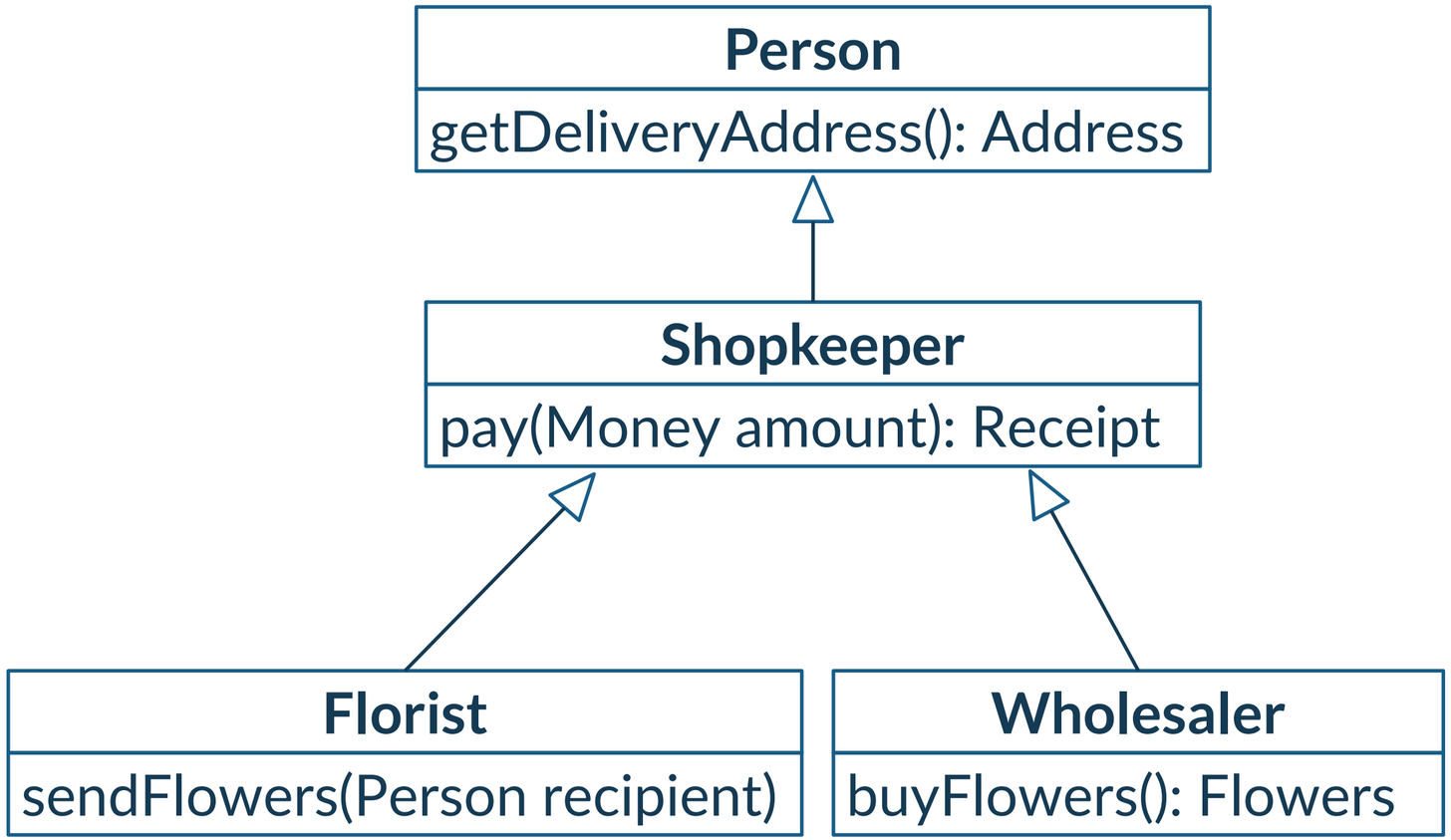
it.units.sdm.oop.florist.polymorphic.Person

```
public class Person {

    public static void main(String[] args) {
        Person robin = new Person();
        Florist fred = new CheapFlorist(new Wholesaler[]{new Wholesaler(10, 3), new Wholesaler(12, 2)});

        fred.sendFlowersTo(robin);
    }
}
```

it.units.sdm.oop.florist.polymorphic.Person

```
public class Person {

    public static void main(String[] args) {
        Person robin = new Person();
        Florist fred = new FastFlorist(new Wholesaler[]{new Wholesaler(10, 3), new Wholesaler(12, 2)});

        fred.sendFlowersTo(robin);
    }
}
```

Class hierarchies and inheritance

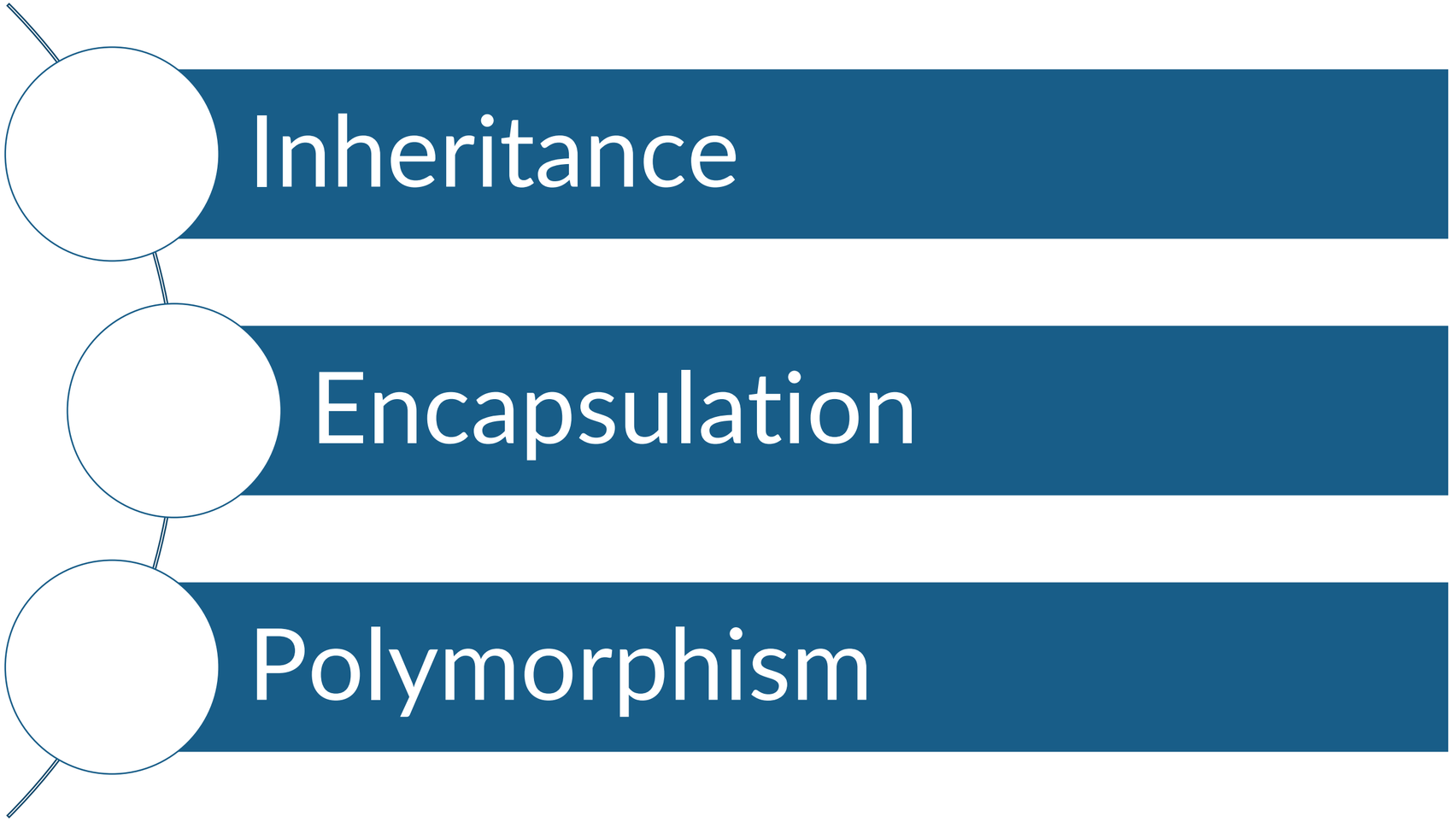


Chris has more knowledge about Fred, who is not only a Florist, but a Shopkeeper, and a Person too

The principle that knowledge of a more general category is also applicable to a more specific category is called inheritance

We say that the class Florist will inherit members of the class Shopkeeper.

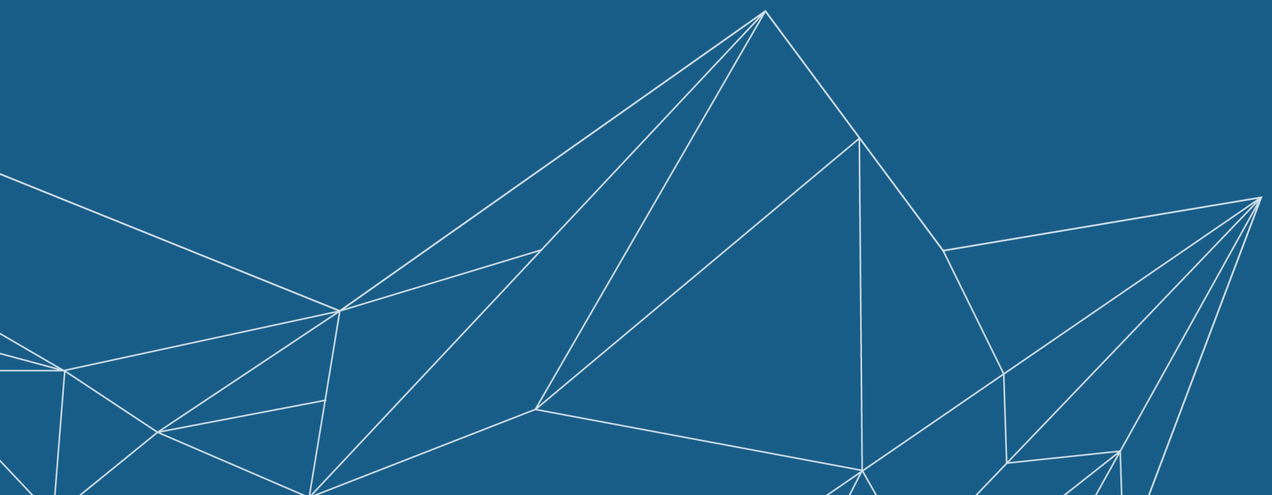
The three pillars of OOP



References

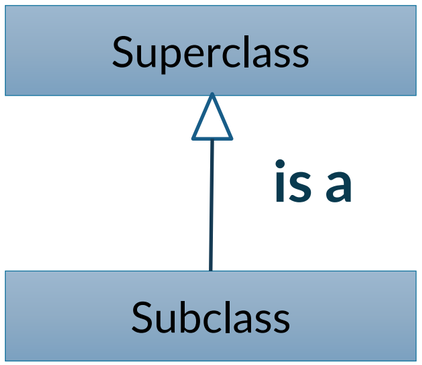
Timothy A. Budd, 2001, **An Introduction to Object-Oriented Programming (3rd. ed.)**, Addison-Wesley Longman Publishing Co., Inc., USA.

Extension and inheritance



Inheritance

Inheritance allow the definition of a new class by specifying that the new class is like another class



The is-a relationship is very strong. Its usage is not always appropriate, and it is very easy to misuse it

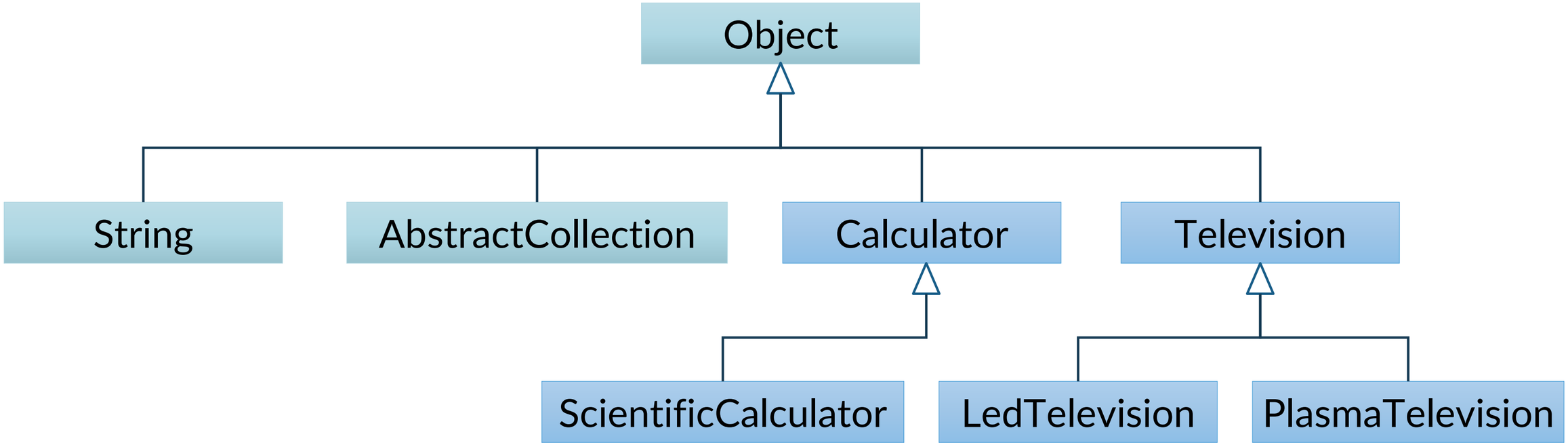
```
package it.units.sdm;

public class PlasmaTelevision extends Television {

    double usageHours;

}
```

Inheritance allows to define new classes by reusing other classes, specifying just the differences

Java uses the extends keyword to indicate the extended superclass

extends implies you are extending the Television class in something that is bigger

Java classes hierarchy



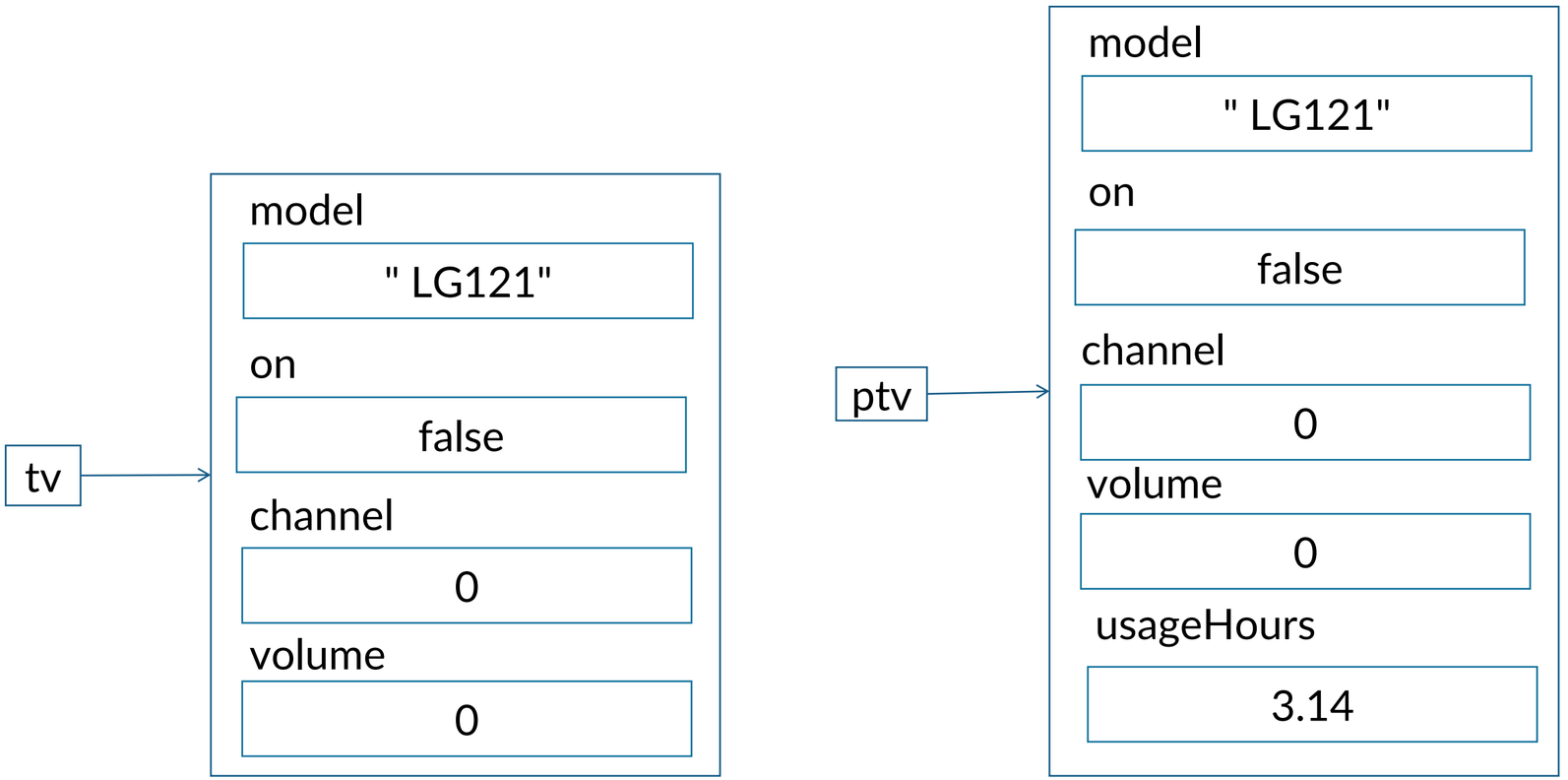
Construction of subclasses

```
package it.units.sdm;

public class PlasmaTelevision extends Television {

    double usageHours;

    public PlasmaTelevision(String model, double usageHours) {
        super(model);
        this.usageHours = usageHours;
    }
}
```

If the superclass defines at least one constructor, the subclass must define a constructor and it must invoke a constructor of the superclass by using super

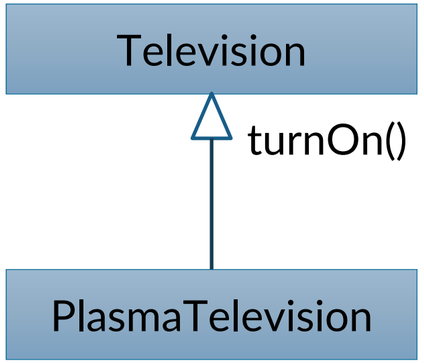
Constructors

```
PlasmaTelevision ptv = new PlasmaTelevision("LG121", 3.14);
Television tv = new Television("LG121");
```



Inherited methods

Inherited method can be used directly
on the instances of the subclass

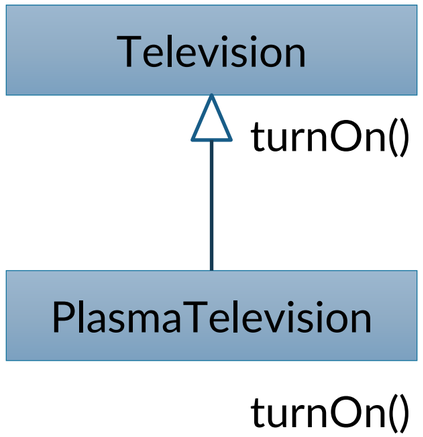



```
PlasmaTelevision ptv = new PlasmaTelevision("LG121", 0.0);
ptv.turnOn();
```

Overridden methods 1/2

```
package it.units.sdm;

public class PlasmaTelevision extends Television {

    double usageHours;
    long startTime;

    public PlasmaTelevision(String model) {
        super(model);
    }

    @Override
    void turnOn() {
        super.turnOn();
        startTime = System.currentTimeMillis();
    }

}
```

Methods in the subclass can override the methods in the superclass



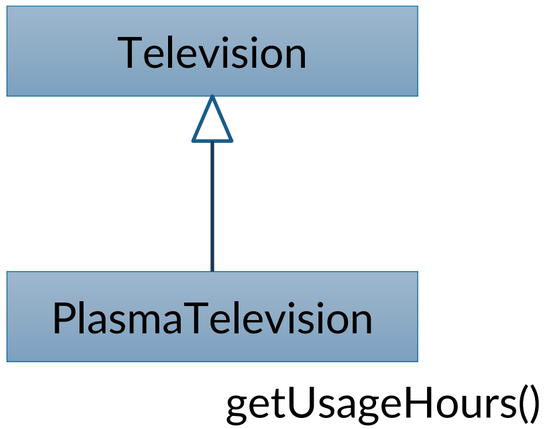
Overridden methods 2/2

```
public class PlasmaTelevision extends Television {

    private double usageHours;
    private long startTime;

    public PlasmaTelevision(String model) {
        super(model);
    }

    void turnOn() {
        super.turnOn();
        startTime = System.currentTimeMillis();
    }

    void turnOff() {
        super.turnOff();
        var endTime = System.currentTimeMillis();
        usageHours += (endTime-startTime) / (1000.0 * 60 * 60);
    }

    public double getUsageHours() {
        return usageHours;
    }
}
```

New methods definition

```
public class PlasmaTelevision extends Television {

    double usageHours;
    long startTime;

    public PlasmaTelevision(String model) {
        super(model);
    }

    public double getUsageHours() {
        return usageHours;
    }
}
```

New methods can also be defined



toString()

public String toString() is an instance method defined in the Object class that returns a human readable string representation of an object

```
public static void main(String[] args) {
    Television tv = new Television("LG120");
    PlasmaTelevision ptv = new PlasmaTelevision("LG121");

    System.out.println("tv: " + tv);
    System.out.println(ptv);
}
```

The toString() method is automatically used by Java when converting an object to a String

In general, the toString() method returns a string that "textually represents" this object. The result should be a concise but informative representation that is easy for a person to read. It is recommended that all subclasses override this method. The string output is not necessarily stable over time or across JVM invocations.

getClass()

```
public static void main(String[] args) {
    Television tv = new Television("LG120");
    PlasmaTelevision ptv = new PlasmaTelevision("LG121");
    Television ptv2 = new PlasmaTelevision("LG121");
    Object ptv3 = new PlasmaTelevision("LG121");

    System.out.println("tv.getClass() " + tv.getClass().getName());
    System.out.println("ptv.getClass() " + ptv.getClass().getName());
    System.out.println("ptv2.getClass() " + ptv2.getClass().getName());
    System.out.println("ptv3.getClass() " + ptv3.getClass().getName());
}
```

```
tv.getClass() it.units.sdm.Television
ptv.getClass() it.units.sdm.PlasmaTelevision
ptv2.getClass() it.units.sdm.PlasmaTelevision
ptv3.getClass() it.units.sdm.PlasmaTelevision
```

getClass() is an instance method defined in the Object class that returns the class of an object

instanceOf

instanceof is an operator that determines if an object is an instance of a specified class

```
public static void main(String[] args) {
    Television tv = new Television("LG120");
    Television ptv = new PlasmaTelevision("LG121");

    System.out.println("is tv a Television? " + (tv instanceof Television));
    System.out.println("is tv a PlasmaTelevision? " + (tv instanceof PlasmaTelevision));
    System.out.println("is ptv a Television? " + (ptv instanceof Television));
    System.out.println("is ptv a PlasmaTelevision? " + (ptv instanceof PlasmaTelevision));
}
```

```
is tv a Television? true
is tv a PlasmaTelevision? false
is ptv a Television? true
is ptv a PlasmaTelevision? true
```

Class casting

```
PlasmaTelevision ptv0 = new PlasmaTelevision("FullHD");
Television tv0 = ptv0;
Object obj0 = ptv0;

Television tv = new PlasmaTelevision("LG121");
PlasmaTelevision ptv = tv; //illegal assignment

PlasmaTelevision ptv2 = (PlasmaTelevision) tv;

Object obj = new PlasmaTelevision("LG121");
PlasmaTelevision ptv3 = (PlasmaTelevision) obj;

Calculator calculator = new Calculator();
PlasmaTelevision ptv4 = (PlasmaTelevision) calculator;
```

It is always possible to assign a variable referring a subclass to a variable of a superclass

To assign a variable of a superclass to a subclass we must use the cast operator

Assignments between variables of different hierarchies are not allowed

Late binding

com.esteco.sdm.Televsion

```
class Television {

    private String model;
    private boolean on;

    Television(String model) {
        this.model = model;
    }

    void turnOn() {
        on = true;
    }

    void turnOff() {
        on = false;
    }
}
```

com.esteco.sdm.PlasmaTelevsion

```
class PlasmaTelevision extends Television {

    double usageHours;
    long startTime;

    PlasmaTelevision(String model) {
        super(model);
    }

    @Override
    void turnOn() {
        super.turnOn();
        startTime = System.currentTimeMillis();
    }
}
```

Which turnOn() is invoked?

```
Television tv = new PlasmaTelevision("LG121");
tv.turnOn();
```

Inheritance with methods

- ✓ New methods can be defined in the subclass to specify the behavior of the objects of the subclass
- ✓ When a method is invoked on an object, the method is searched in the class of the receptor object
- ✓ If it is not found, then it is searched higher up in the hierarchy

Access control and encapsulation

Access control in Java

Manage the visibility and encapsulation of classes and their members by specifying which parts of a class can be accessed, modified, or inherited by other classes

A class may be declared with the modifier public, in which case that class is visible to all classes everywhere. If a class has no modifier (the default, also known as package-private), it is visible only within its own package.

A member with access modifier private can only be accessed from its own class

A member without an access modifier can only be accessed from within its own package (package-private)

A member with access modifier protected can be accessed from within its own package and from the subclasses of its class in another package

A member with access modifier public is visible from all the classes

Modifier	Class	Package	Subclass	World
public	Y	Y	Y	Y
protected	Y	Y	Y	N
no modifier	Y	Y	N	N
private	Y	N	N	N

Who can access this class? 1/3

com.esteco.sdm.Television

```
package com.esteco.sdm;

public class Television {

    private String model;

    public Television(String model) {
        this.model = model;
    }

    public String getModel() {
        return model;
    }
}
```

Anyone from any package!

Who can access this class? 2/3

com.esteco.sdm.Television

```
package com.esteco.sdm;

class Television {

    private String model;

    public Television(String model) {
        this.model = model;
    }

    public String getModel() {
        return model;
    }
}
```

Anyone from the com.esteco.sdm package!

Who can access this class? 3/3

com.esteco.sdm.Television

```
package com.esteco.sdm;

private class Television {

    private String model;

    public Television(String model) {
        this.model = model;
    }

    public String getModel() {
        return model;
    }
}
```

No one, this declaration is illegal!

Who can instantiate this class? 1/3

com.esteco.sdm.Television

```
package com.esteco.sdm;

public class Television {

    private String model;

    public Television(String model) {
        this.model = model;
    }

    public String getModel() {
        return model;
    }
}
```

Anyone from any package!

Who can instantiate this class? 2/3

com.esteco.sdm.Television

```
package com.esteco.sdm;

public class Television {

    private String model;

    private Television(String model) {
        this.model = model;
    }

    public String getModel() {
        return model;
    }
}
```

Only someone within this class!

Who can instantiate this class? 3/3

com.esteco.sdm.Television

```
package com.esteco.sdm;

class Television {

    private String model;

?   public Television(String model) {
        this.model = model;
    }

    public String getModel() {
        return model;
    }
}
```

Only someone within the same package!

Who can access the model variable? 1/3

com.esteco.sdm.Television

```
package com.esteco.sdm;

public class Television {

    private String model;

    public Television(String model) {
        this.model = model;
    }

    public String getModel() {
        return model;
    }
}
```

Only someone within the same class!

Who can access the model variable? 2/3

com.esteco.sdm.Television

```
package com.esteco.sdm;

public class Television {

    public String model;

    public Television(String model) {
        this.model = model;
    }

    public String getModel() {
        return model;
    }
}
```

Everyone can read and write the model variable.

In general, very dangerous.

Who can access the model variable? 3/3

com.esteco.sdm.Television

```
package com.esteco.sdm;

public class Television {

    public final String model;

    public Television(String model) {
        this.model = model;
    }

    public String getModel() {
        return model;
    }
}
```

Everyone can read the model variable.

But only the constructor can assign it!

Who can invoke the getModel() method? 1/4

com.esteco.sdm.Television

```
package com.esteco.sdm;

public class Television {

    private final String model;

    public Television(String model) {
        this.model = model;
    }

    public String getModel() {
        return model;
    }
}
```

Everyone can invoke the getModel() method

Who can invoke the getModel() method? 2/4

com.esteco.sdm.Television

```
package com.esteco.sdm;

public class Television {

    private final String model;

    public Television(String model) {
        this.model = model;
    }

    private String getModel() {
        return model;
    }
}
```

No one outside the Television class.

Who can invoke the getModel() method? 3/4

com.esteco.sdm.Television

```
package com.esteco.sdm;

public class Television {

    private final String model;

    public Television(String model) {
        this.model = model;
    }

    protected String getModel() {
        return model;
    }
}
```

Anyone from the same package,
or from any subclass.

Who can invoke the getModel() method? 4/4

com.esteco.sdm.Television

```
package com.esteco.sdm;

class Television {

    private final String model;

    public Television(String model) {
        this.model = model;
    }

    public String getModel() {
        return model;
    }
}
```

Everyone with a reference to a Television object.

In more details, everyone with a reference to a Television object or an object of a subclass of Television.

You cannot reduce the access level with extension

com.esteco.sdm.Television

```
public class Television {

    private String model;
    private boolean on;

    public Television(String model) {
        this.model = model;
    }

    public void turnOn() {
        on = true;
    }

    public void turnOff() {
        on = false;
    }
}
```

com.esteco.sdm.PlasmaTelevision

```
class PlasmaTelevision extends Television {

    double usageHours;
    long startTime;

    public PlasmaTelevision(String model) {
        super(model);
    }

    @Override
    protected void turnOn() {
        super.turnOn();
        startTime = System.currentTimeMillis();
    }

}
```

Tips on choosing an access level

If other programmers use your class, you want to ensure that errors from misuse cannot happen. Access levels can help you do this.

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. Public fields tend to link you to a particular implementation and limit your flexibility in changing your code.

https://docs.oracle.com/javase/tutorial/java/javaOO/accesscontrol.html

Final classes

com.esteco.sdm.Television

```
package com.esteco.sdm;

public final class Television {

    private String model;

    public Television(String model) {
        this.model = model;
    }

    public String getModel() {
        return model;
    }
}
```

Final classes cannot be extended.

Final methods

com.esteco.sdm.Television

```
public class Television {

    private String model;
    private boolean on;

    public Television(String model) {
        this.model = model;
    }

    final void turnOn() {
        on = true;
    }

    final void turnOff() {
        on = false;
    }
}
```

com.esteco.sdm.PlasmaTelevision

```
class PlasmaTelevision extends Television {

    double usageHours;
    long startTime;

    public PlasmaTelevision(String model) {
        super(model);
    }

    @Override
    void turnOn() {
        super.turnOn();
        startTime = System.currentTimeMillis();
    }

}
```

Final methods cannot be overridden.

Writing for others

If other programmers use your class, you want to ensure that errors from misuse cannot happen

Please consider yourself as another programmer too!

What's wrong with extending Television and overriding turnOn()?

What if forget to invoke super.turnOn()

com.esteco.sdm.PlasmaTelevision

```
class PlasmaTelevision extends Television {

    private double usageHours;
    private long startTime;

    public PlasmaTelevision(String model) {
        super(model);
    }

    @Override
    public void turnOn() {
        super.turnOn();
        startTime = System.currentTimeMillis();
    }
}
```

Abstract methods 1/2

If the turnOn() method is final, how do we allow other programmers to do something when a Television is turned on?

turnedOn() is the entry point for subclasses to perform operations after a Television is turned on.

turnOn() is defining a protocol. How the Television class works and how it should be extended.

com.esteco.sdm.Television

```
public class Television {

    private String model;
    private boolean on;

    public Television(String model) {
        this.model = model;
    }

    final void turnOn() {
        on = true;
        turnedOn();
    }

    protected void turnedOn() {
    }
}
```

Abstract methods 2/2

What if we want all subclasses to be forced to define the turnedOn() method?

We define turnedOn() as an abstract method. As a consequence, the Television class becomes abstract too.

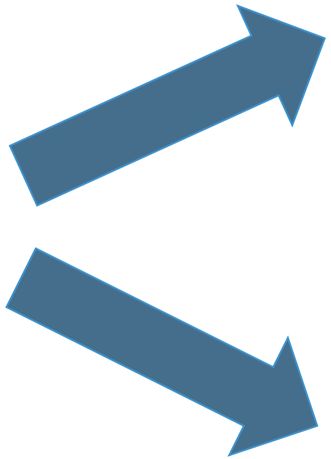
Abstract classes cannot be instantiated. But they can be extended by subclasses.

com.esteco.sdm.Televsion

```
public abstract class Television {

    private String model;
    private boolean on;

    protected Television(String model) {
        this.model = model;
    }

    public final void turnOn() {
        on = true;
        turnedOn();
    }

    protected abstract void turnedOn();
}
```

Final and abstract

The modifiers final and abstract can be applied to both classes and methods, and they are mutually exclusive



A final class cannot be extended

A final method cannot be overridden in a subclass

An abstract class must be extended

An abstract method must be overridden in a subclass

Interfaces and polymorphism

Interfaces

it.units.sdm.Display

```
public interface Display {

    void display(String text);

}
```

it.units.sdm.Calculator

```
public class Calculator {

    final Display display;
    //...

    Calculator(Display display) {
        this.display = display;
    }

     void onePressed() {
        string += "1";
        display.display(string);
    }
}
```

Interfaces are used to abstract what a class must do from how it does it

Interfaces are similar to classes, but

1. they don't have instance variables
2. all methods are abstract (with the exception of methods with a default implementation)
3. all methods are implicitly public

An interface definition doesn't say anything about how the methods are implemented

Interface implementation 1/2

it.units.sdm.Display

```
public interface Display {

    void display(String text);

}
```

```
class ConsoleDisplay implements Display {

    @Override
    public void display(String text) {
        System.out.println(text);
    }
}

class PopupDisplay implements Display {

    @Override
    public void display(String text) {
        JOptionPane.showMessageDialog(null, text);
    }
}
```

An interface can be implemented by any number of classes

A class can implement any number of interfaces

Interfaces are not inherited, they are implemented, so the single inheritance does not apply. There is no inheritance of instance members

The methods that implement an interface must be declared public so there no way to restrict the access

Interface implementation 2/2

it.units.sdm.Display

```
public interface Display {

    void display(double d);

}
```

```
class ConsoleDisplay implements Display {

    @Override
    public void display(double d) {
        System.out.println(format(d));
    }

    public String format(double d) {
        return String.valueOf(d);
    }
}
```

A class that implement interfaces can have its own instance variables and it can define its own constructors and methods

Partial interface implementation

it.units.sdm.Display

```
public interface Display {

    void display(String text);
    void display(double d);

}
```

A class implementing an interface must implement all the methods. Otherwise, it must be declared abstract

```
abstract class ConsoleDisplay implements Display {

    @Override
    public void display(String text) {
        System.out.println(text);
    }
}
```

Implementation of multiple interfaces 1/4

```
interface AutonomousCar {

    void driveTo(String address);

    void stop();
}

interface KeylessCar {

    void start();

    void stop();
}
```

If a class implements two interfaces that declare the same method, then the same method will be used by clients of either interface.

```
class FiatTopolino implements KeylessCar, AutonomousCar {

    @Override
    public void driveTo(String address) {
        //Do something
    }

    @Override
    public void start() {
        //Do something
    }

    @Override
    public void stop() {
        //Should I stop as a KeylessCar
        //or as an AutonomousCar?
    }
}
```

If a class implements more than one interface, the interfaces are separated with a comma.

Implementation of multiple interfaces 2/4

```
interface AutonomousCar {

    void driveTo(String address);

    void stop();
}

interface KeylessCar {

    void start();

    boolean stop();
}
```

```
class FiatTopolino implements KeylessCar, AutonomousCar {

    @Override
    public void driveTo(String address) {
        //Do something
    }

    @Override
    public void start() {
        //Do something
    }

    @Override
    public void/boolean stop() {
        //FiatTopolino cannot implement both interfaces
    }
}
```

A class cannot implement methods with the same name, the same parameters, but a different return type.

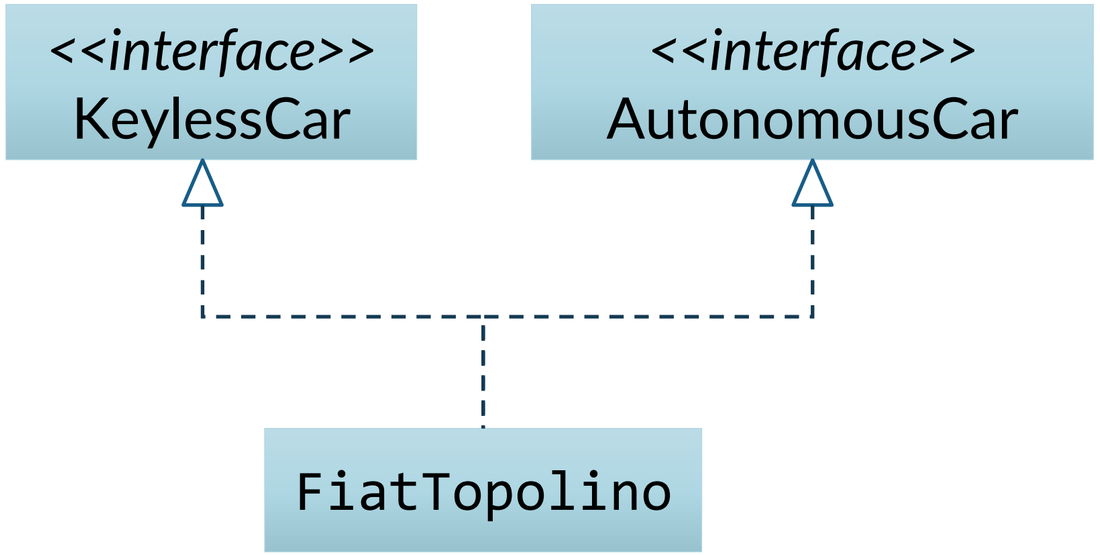
Implementation of multiple interfaces 3/4

```
FiatTopolino c = new FiatTopolino();
AutonomousCar ac = c;
KeylessCar kc = c;

ac = kc;
```

Is this a legal assignment?

Can I cast kc to an Autonomous car?

```
ac = (AutonomousCar) kc
```



Implementation of multiple interfaces 4/4

```
FiatTopolino c = new FiatTopolino();
AutonomousCar ac = c;
KeylessCar kc = c;

Display d1 = (Display) ac;
Display d2 = (Display) kc;
```

This code compiles, I will get a ClassCastException error at runtime.

```
FiatTopolino c = new FiatTopolino();
AutonomousCar ac = c;
KeylessCar kc = c;

Calculator c1 = (Calculator) c;
```

This code doesn't compile, cannot cast a FiatTopolino into a Calculator.

For the compiler, it is always legal to cast to an interface.
But I cannot cast to a class if the object belongs to a different hierarchy.

Interface extension

```
interface Collection {

    int getSize();

    boolean isEmpty();
}

interface MutableCollection {

    void clear();
}

interface List extends Collection, MutableCollection {

    void addElement(Object obj);
}
```

An interface can extend multiple interfaces

What’s wrong with this interface?

```
interface AutonomousCar {

    void driveTo(String address);

    void toString();
}
```

The return type of the toString() method clashes with Object.toString()

Anonymous classes 1/3

it.units.sdm.Display

```
public interface Display {

    void display(String text);

}
```

```
Display display = new Display() {
    @Override
    public void display(String text) {
        System.out.println();
    }
};
```

Interfaces can be implemented by anonymous classes too

An anonymous class is a class without a name

The new operator creates an object of a class that has no name

Anonymous classes 2/3

```
public class Calculator {

    public static void main(String[] args) {
        Display display = new Display() {
            @Override
            public void display(String text) {
                System.out.println(text);
            }
        };
        var calculator = new Calculator(display);
        calculator.onePressed();
        calculator.plusPressed();
        calculator.twoPressed();
        calculator.plusPressed();
        calculator.twoPressed();
        calculator.equalPressed();
    }
    //…
}
```

By compiling this code, two classes are created:

`Calculator.class`
`Calculator$1.class`

`Calculator$1.class` represents the anonymous class. That is anonymous in the source code, but it is not anonymous for the virtual machine.

Anonymous classes 3/3

```
Object a = new Object() {

    int a;

    {
        //there are no constructors but
        //we can use initializer blocks
    }

    public int getA() {
        return a;
    }

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

Anonymous classes are not used to implement interfaces only, but they can be used to extends objects, of any type

Can we invoke the `getA()` public method?

Java is a statically linked language

Is there any “type” defining that `getA()` method?

Anonymous classes “don’t define” a new type

You can invoke `getA()` from the same class

Interface static fields 1/2

```
interface AutonomousCar {

    String DEFAULT_ADDRESS = "3500 Deer Creek Road, Palo Alto, California";

    void driveTo(String address);

    void stop();
}
```

Variables can be declared inside interface declarations. They are implicitly public, final, and static, meaning they cannot be changed by the implementing class and that they must be initialized

They can be used as constants shared among the implementing classes

Interface static fields 2/2

```
class FiatTopolino implements KeylessCar, AutonomousCar {

    @Override
    public void driveTo(String address) {
        if (address == null) {
            address = DEFAULT_ADDRESS;
        }
        //drive to address
    }

    @Override
    public void start() {
        //Do something
    }

    @Override
    public void stop() {
        //Just stop!
    }
}
```

Sample usage of DEFAULT_ADDRESS

Static method in interfaces 1/2

Static methods in interfaces are exactly like static methods in classes

All static methods in interfaces are implicitly public

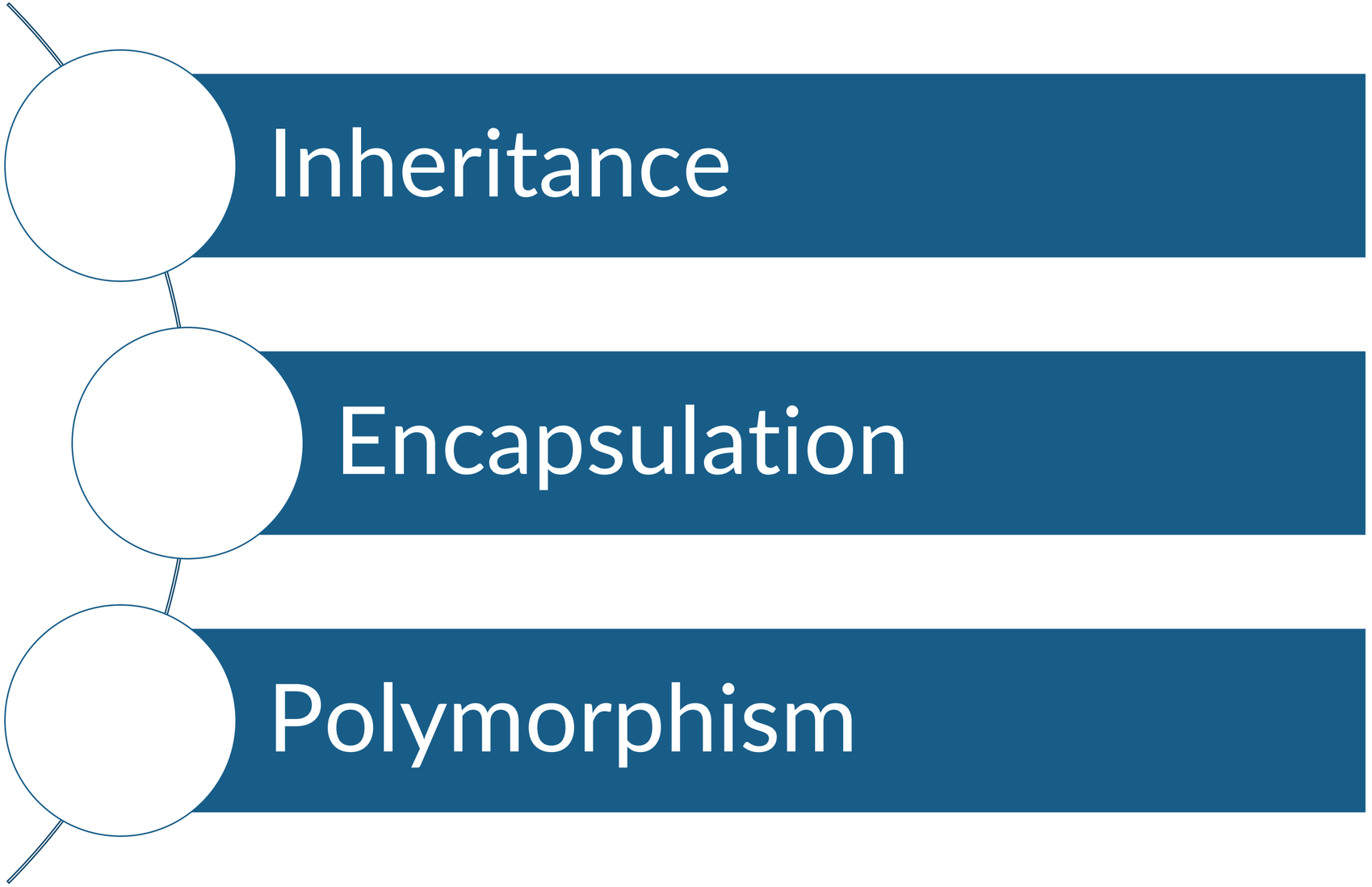
```
interface AutonomousCar {

    static String getDefaultAddress() {
        return "3500 Deer Creek Road, Palo Alto, California";
    }

    void driveTo(String address);

    void stop();
}
```

Static method in interfaces 2/2

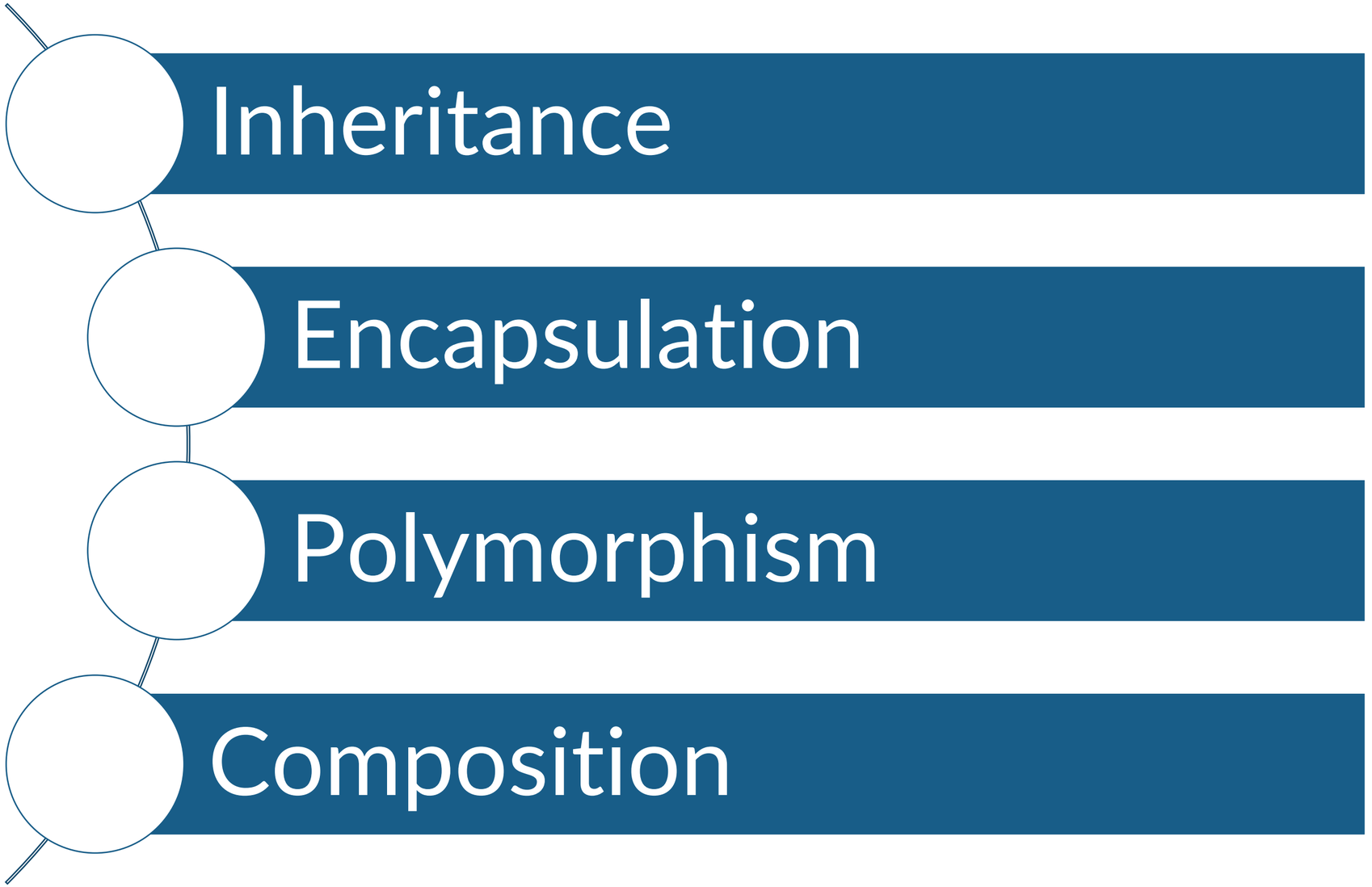
Static methods in interfaces can be used as factory methods

```
interface Display {

    void display(String text);


    static Display createDefaultDisplay() {
        return new Display() {
            @Override
            public void display(String text) {
                System.out.println();
            }
        };
    }

}
```

The Three Musketeers of OOP



What about the fourth Musketeer?



Inheritance vs Composition

```
package com.esteco;

public class PlasmaTelevision extends Television {
    …
}
```

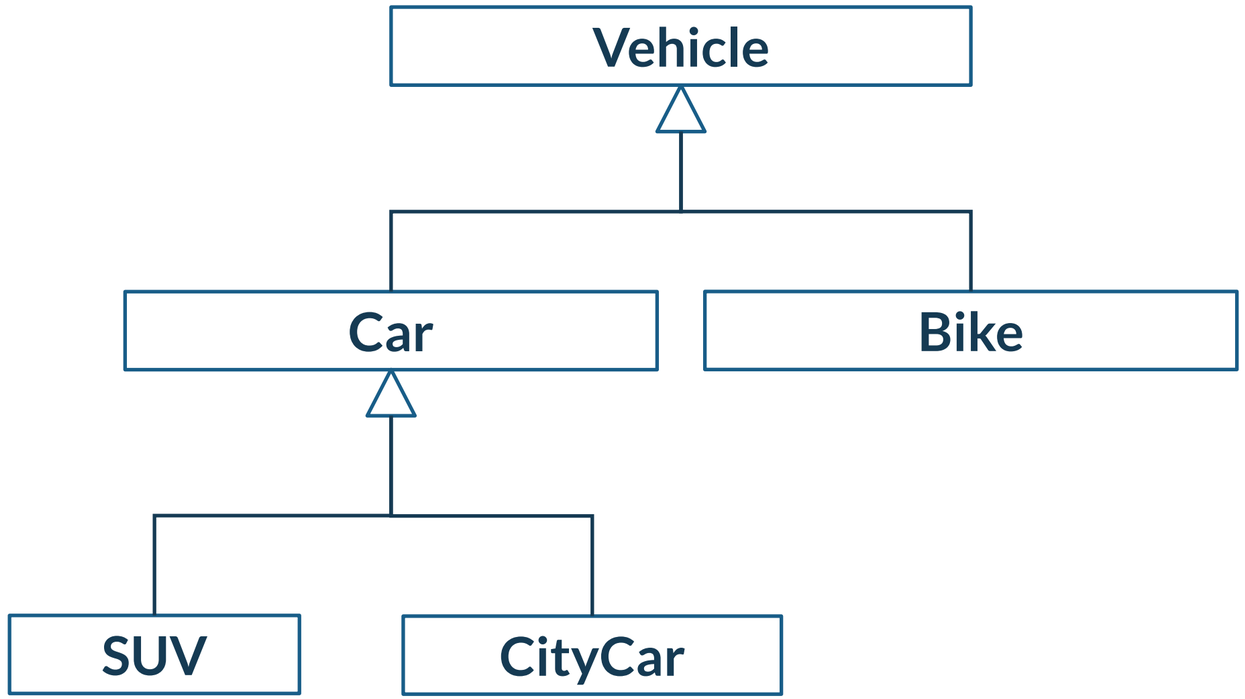
```
package com.esteco;

public class Calculator {

    final Display display;
}
```

5 forms of inheritance

1. Specialization/Generalization
2. Extension
3. Specification
4. Construction
5. Limitation

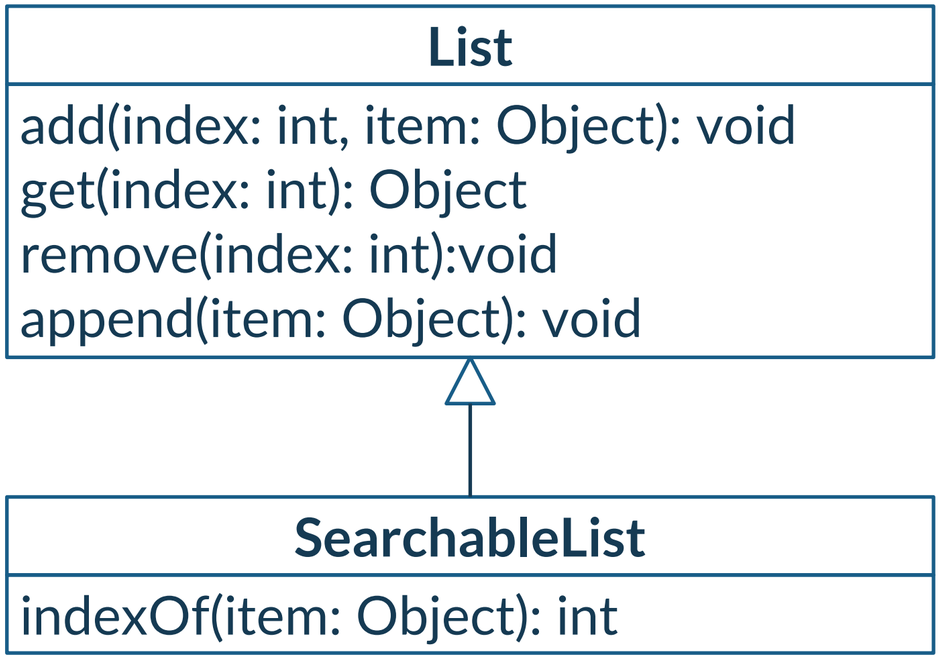
Specialization/Generalization

- The subclass is a specialization of the general parent class (1-to-1)
- The superclass is a generalization of the subclasses (1-to-many)
- Often used in domain modelling



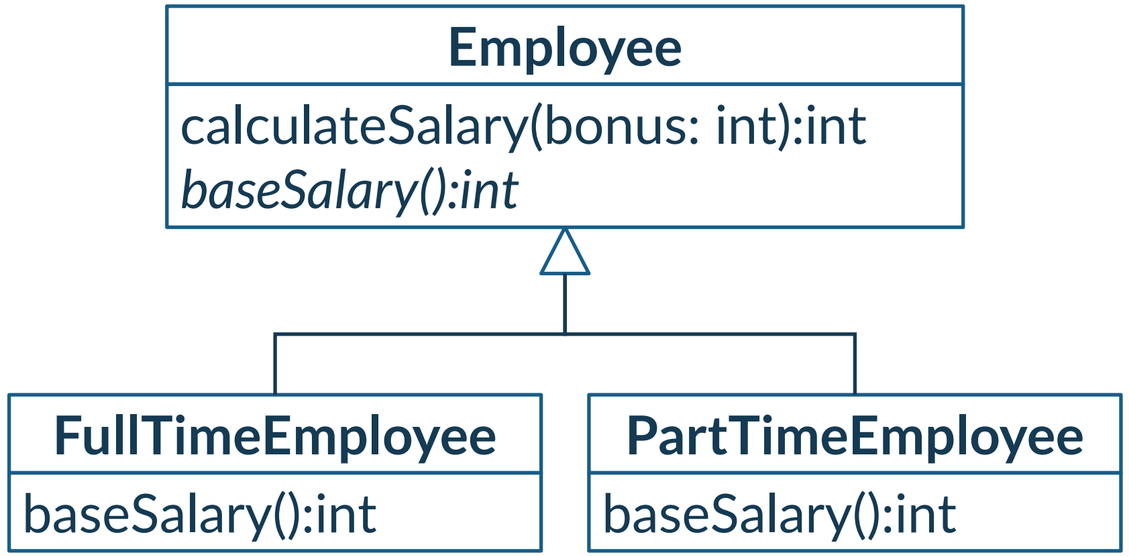
Extension

- The subclass adds new functionalities to the parent class
- It adds new methods to those of the parent
- In general, you will use the subclass as substitute of the superclass



Specification

- The subclass implements abstract methods of the superclass
- Used when the superclass defines a “protocol”, and it defers the implementation of some specialized methods to subclasses

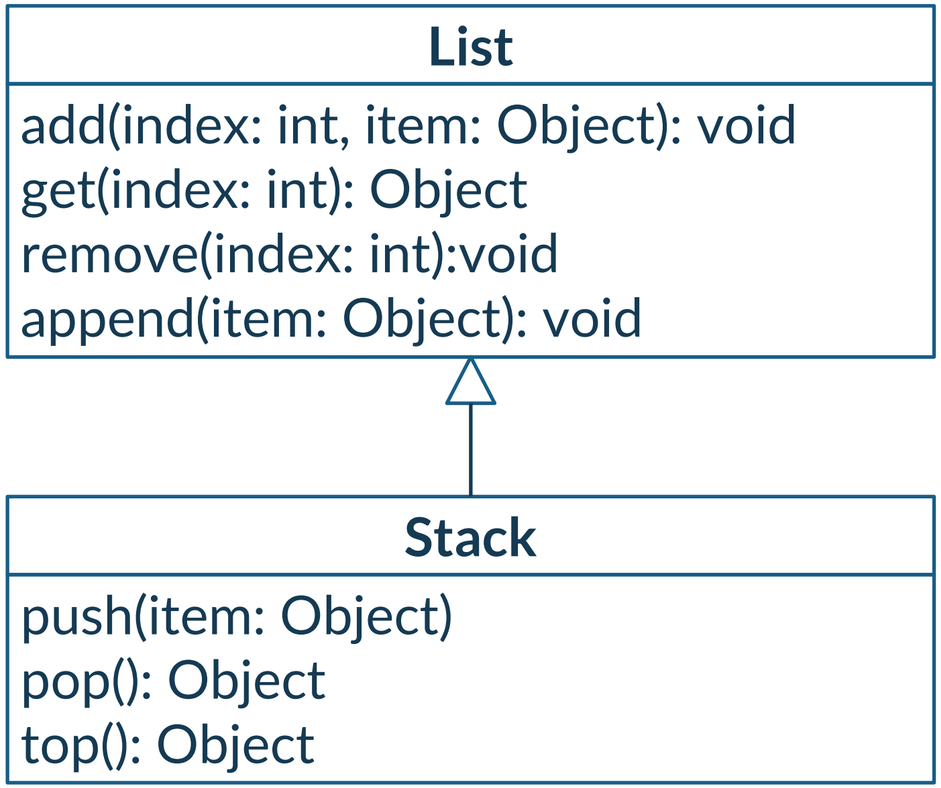


it.units.sdm.Employee

```
public abstract class Employee {

    protected abstract int baseSalary();

    public int totalSalary(int bonus) {
        return baseSalary() + bonus;
    }
}
```

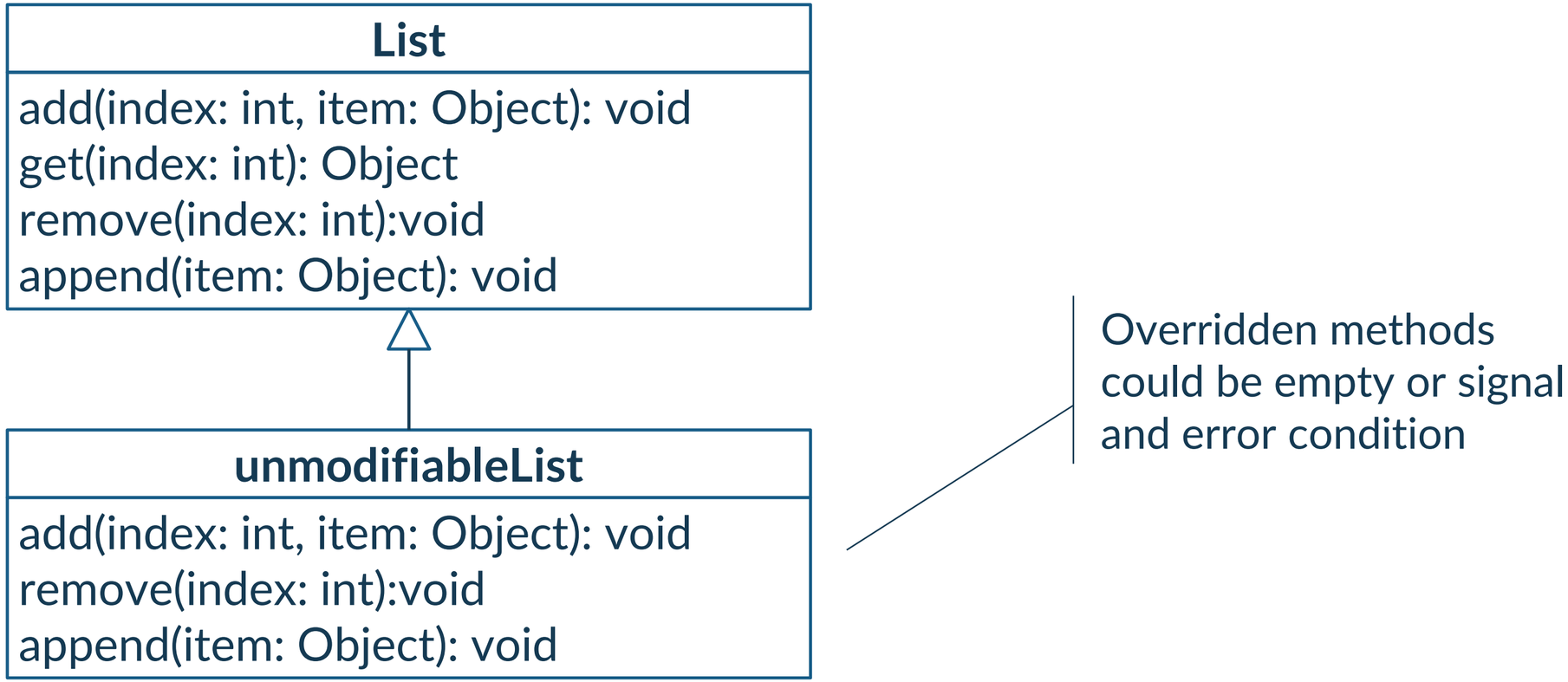
Construction

- The subclass and superclass share some behavior
- In general, you will not use the subclass as substitute of the superclass
- Composition could play a better role here!



Limitation

- The subclass restrict some properties of the superclass by overriding some methods
- In general, you cannot use the subclass as substitute of the superclass



When to use inheritance

Both classes are in the same logical domain

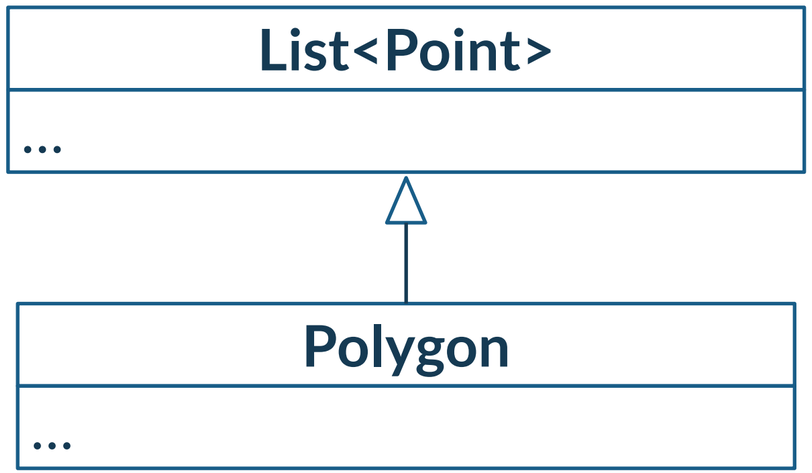
The subclass is a proper subtype of the superclass

The implementation of the superclass is necessary or appropriate for the subclass

The enhancements made by the subclass are primarily additive

https://www.thoughtworks.com/insights/blog/composition-vs-inheritance-how-choose

Both classes are in the same logical domain

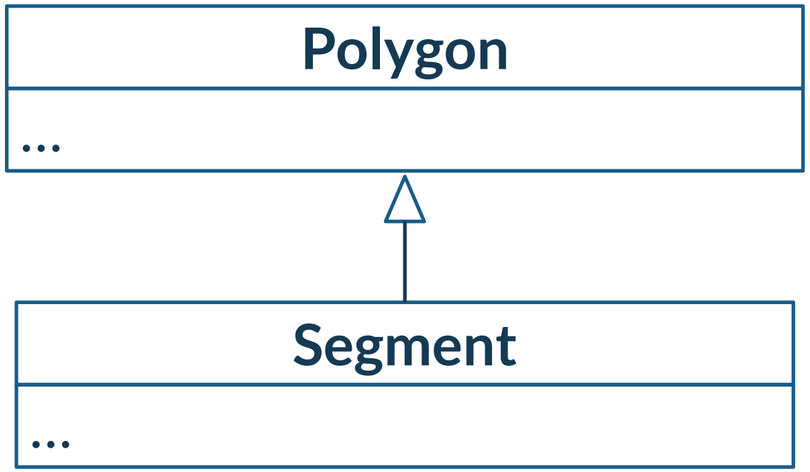


Implementation class

Domain class

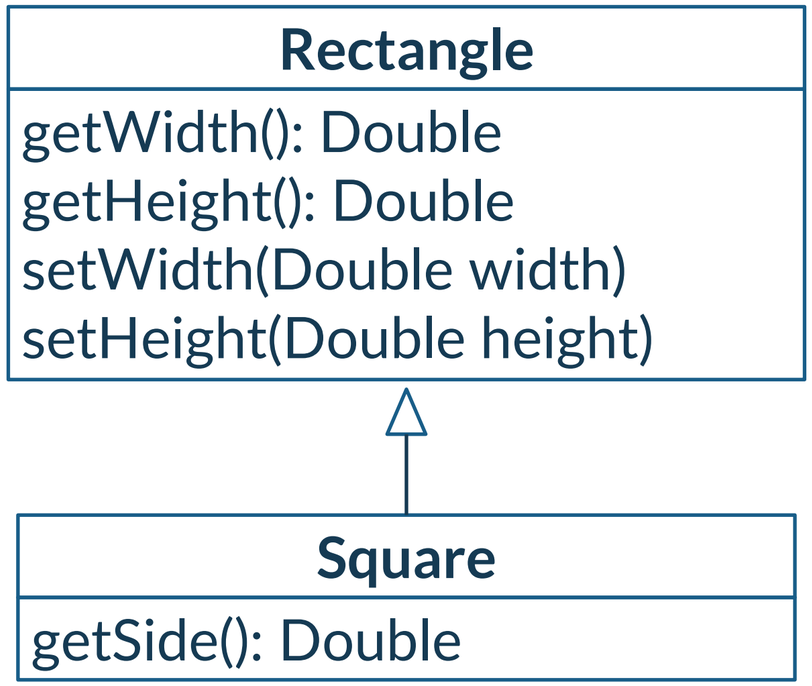
Maybe a polygon is composed by a list
of vertexes, and some other attributes

The subclass is a proper subtype of the superclass



There is something wrong at the semantic level

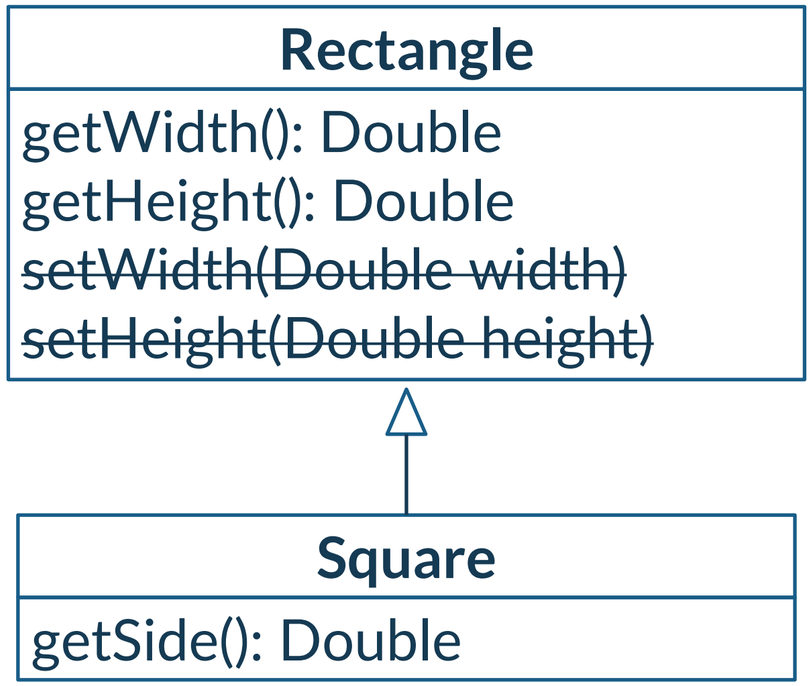
The implementation of the superclass is necessary for the subclass



Is a square a rectangle?

It could depend on the domain of the problem!

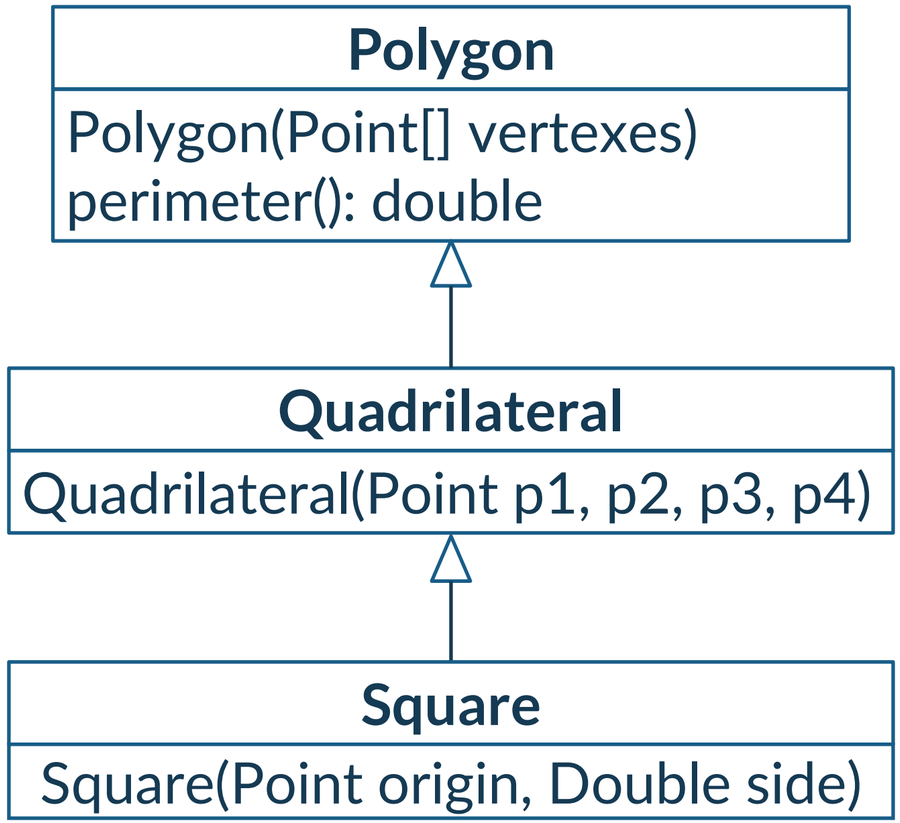
The implementation of the superclass is necessary for the subclass



Is a square a rectangle?

It could depend on the domain of the problem!

The enhancements made by the subclass are primarily additive



Do you really need this hierarchy?

Maybe not, you are adding constraints on a creation of an object

A good opportunity to use a Builder

Assignment

Assignment

```
public interface Collection {

    boolean isEmpty();

    int getSize();

    boolean contains(String string);

    String[] getValues();
}

public interface Stack extends Collection {

    void push(String string);

    String pop();

    String top();
}
```

```
public interface List extends Collection {

    void add(String string);

    String get(int index);

    void insertAt(int index, String string);

    void remove(int index);

    int indexOf(String string);

}
```

Implement the Stack and List interfaces by using only the topics we have seen so far, try to minimize code duplication

Hint: consider the usage of both inheritance and composition

Thank you!

esteco.com