

Programming in Java – Part 01
OOP in Java



Paolo Vercesi
ESTECO SpA

© 2019 ESTECO SpA

Agenda



Object-oriented programming

The three musketeers of OOP

Equivalence and identity

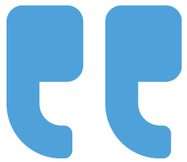
© 2019 ESTECO SpA



Object-oriented programming



© 2019 ESTECO SpA

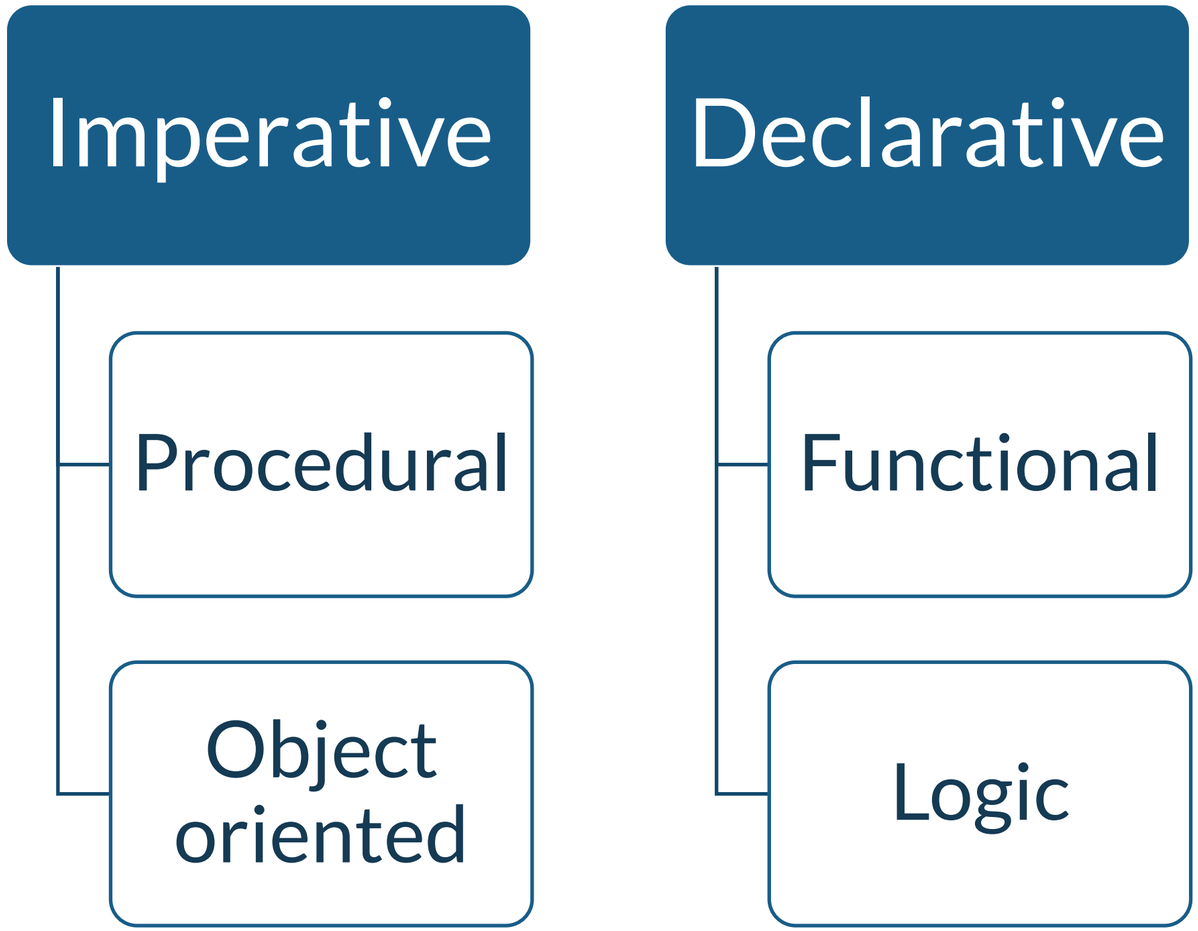


A programming paradigm is a way of conceptualizing what it means to perform computation and how tasks to be carried out on a computer should be structured and organized



© 2019 ESTECO SpA

Popular programming paradigms



© 2019 ESTECO SpA

Procedural paradigm

A program is divided into a series of reusable procedures, each of which performs a specific task. Procedures are typically invoked in a sequence, and they can pass data between each other, to accomplish a larger task.

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

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

The C programming language is one of the most used procedural languages, 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



© 2019 ESTECO SpA

Object-oriented paradigm

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

Objects define both state (variables) and behavior (methods)

In Java, each object is an instance of a class, and the class defines the state and behavior 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; not all of them are class-based

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



© 2019 ESTECO SpA

Functional paradigm

A program consist of combination of function call

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



© 2019 ESTECO SpA



An object-oriented program is structured as a community of interacting units, called objects

Each object has a role to play

Each object provides a service, or performs an action, that is used by other members of the community



© 2019 ESTECO SpA

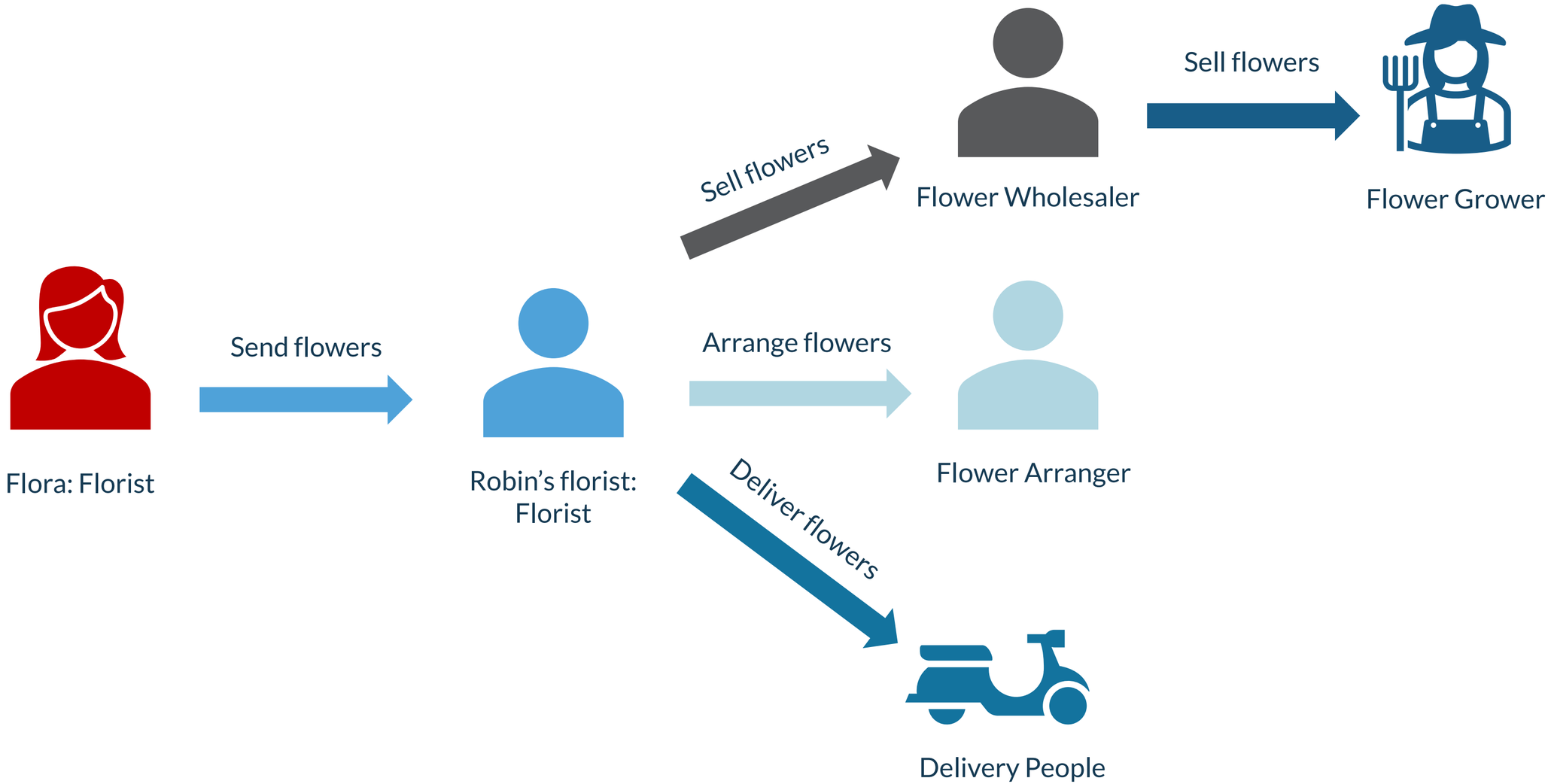
Sending flowers to a friend

- Suppose an individual named Chris wants to send flowers to a friend named Robin
- Robin lives in another city and Chris cannot pick the flowers and take them to Robin in person
- Chris walks to a nearby flower shop, run by Flora, a florist
- Chris tells Flora
  - the kinds of flowers to send to Robin
  - the address to which they should be delivered



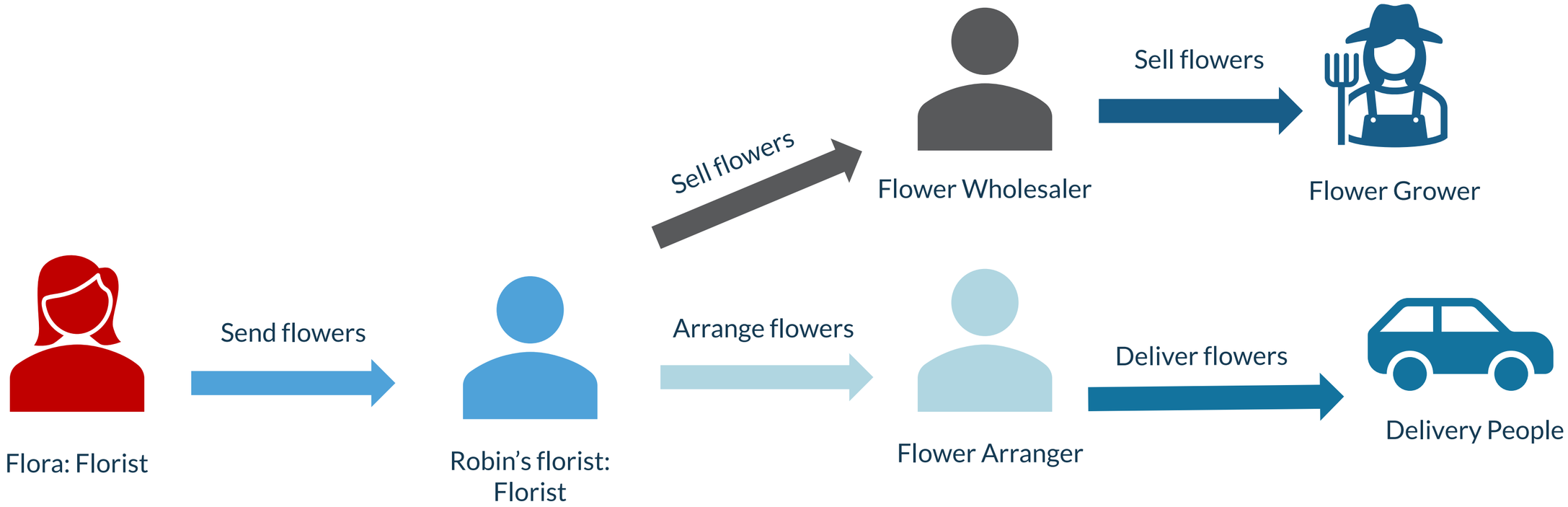
© 2019 ESTECO SpA

How the flowers are sent to Chris 1/3



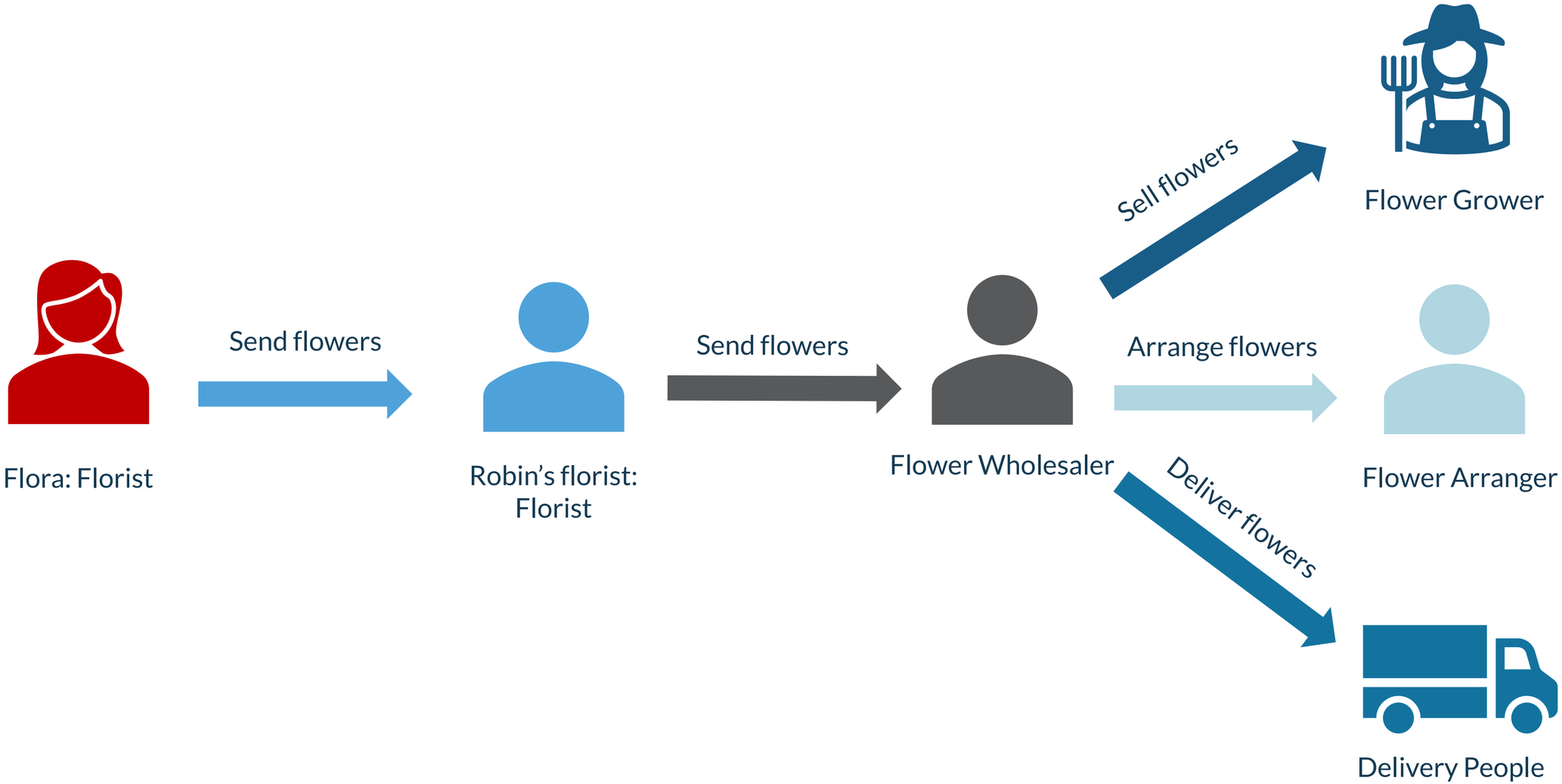
© 2019 ESTECO SpA

How the flowers are sent to Chris 2/3



© 2019 ESTECO SpA

How the flowers are sent to Chris 3/3



© 2019 ESTECO SpA

Abstraction & responsibility



Abstraction is the *purposeful suppression or hiding, of some details of* a complex system by showing only the essential information

Responsibility define what an object does, Flora is a Florist, and she has the responsibility to deliver the flowers

In general, by solving a problem in terms of responsibilities you increase the level of abstraction

© 2019 ESTECO SpA

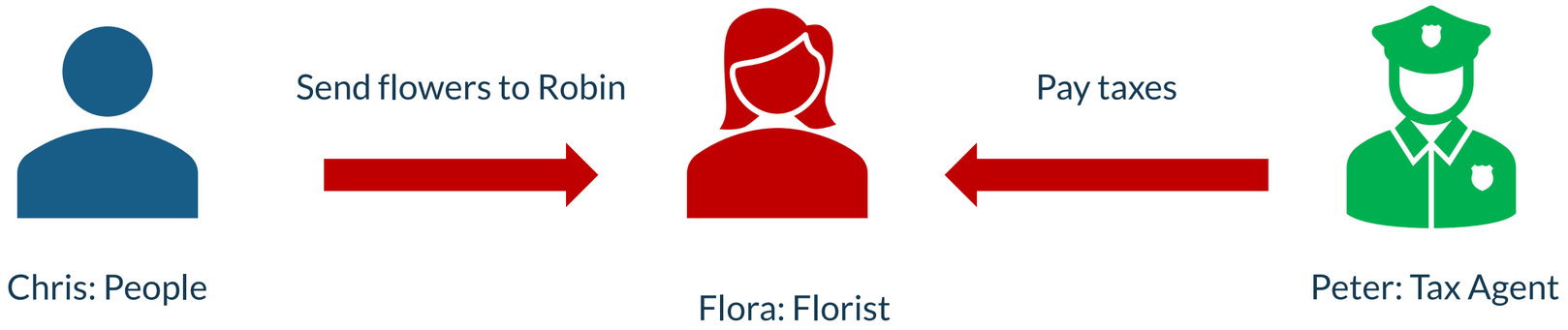
Abstraction & context

The abstraction depends on the context or on the problem we want to solve

Flora is a Florist and Chris wants to ask Flora to send flowers to his friends

On the other hand, the Income Office wants to send Flora a Tax Payment Notice, thus Flora is not only a Florist but also a TaxPayer

The same real-world entity has different abstractions, depending on the scenarios you want to support



© 2019 ESTECO SpA

Procedural programming vs OOP

Procedural programming

- you write procedures that operate on data passed as argument
- the strategy for solving the problem is hard-coded into the procedure

Example
- sendFlowersThroughWholesaler(…)
- buyArrangeAndDeliver(…)

The procedure name itself might reflect the chosen strategy
- to change the strategy, you must change which procedure is called

The procedures can be defined in a library, and the actual procedure to be executed is determined at compile time (or link time in C) — this is called early binding or static dispatch

Object-oriented programming

- you write methods that operates on object members
- the strategy for solving the problem depends on the object

Example
- flora. sendFlowers()
- wholesaler.sendFlowers()

The method name stays the same
- the actual strategy is chosen at runtime, and it depends on the invoked object

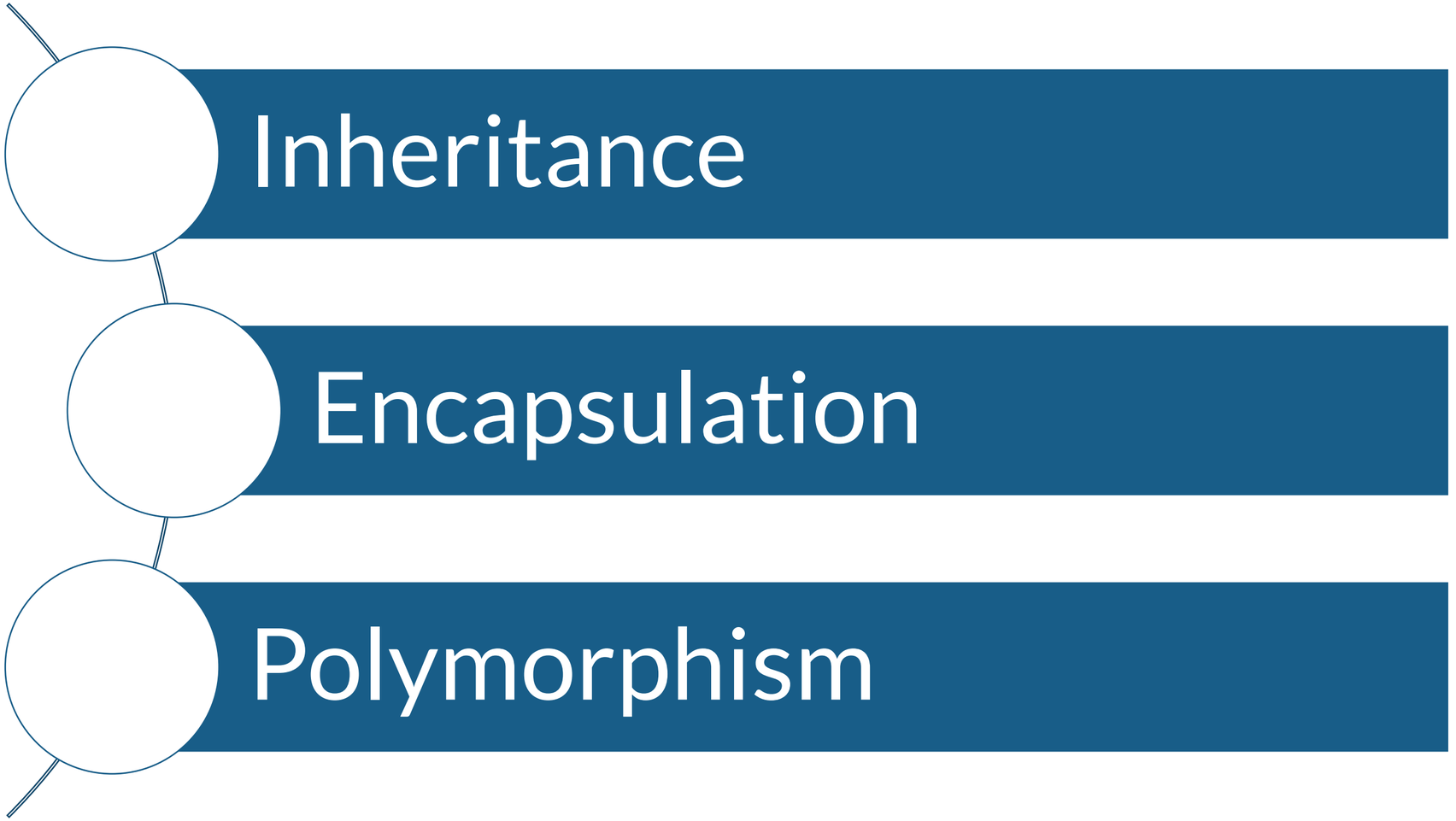
You don’t need to recompile the program to change the strategy –thanks to late binding or dynamic dispatch

You can even use strategies that weren’t known when the program was originally compiled



© 2019 ESTECO SpA

Other characteristics of OOP languages



© 2019 ESTECO SpA

Inheritance

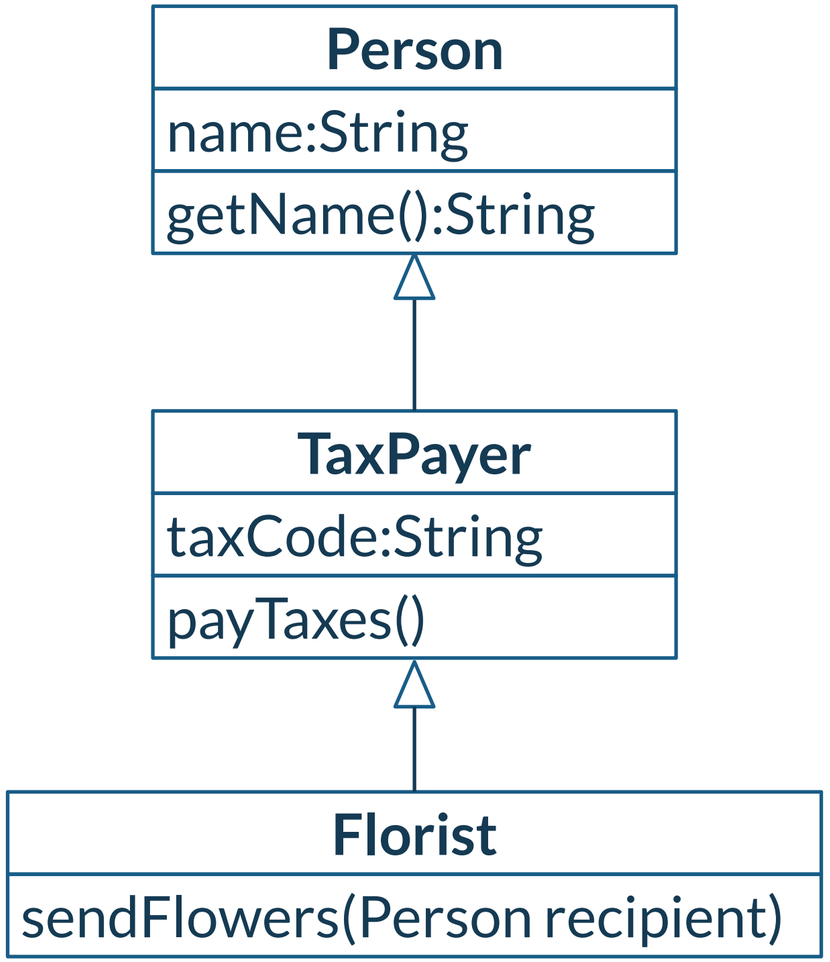
You use inheritance *primarily* for code reuse – not for taxonomy or classification

In prototype-based inheritance a child object inherits behavior and data (methods and fields) of an existing parent object called prototype

In class-based inheritance a child class (a derived or subclass) inherits behavior and data (methods and fields) of an existing class (a parent or super-class)

Composition is another way to support code reuse

Can you see any flaw in this hierarchy?



© 2019 ESTECO SpA

Encapsulation

Encapsulation serves two purposes

1. separation of interface from implementation – abstraction focuses on the observable behavior of an object, whereas encapsulation focuses on the implementation of that behavior
2. data protection – internal details are kept private, they are exposed only through a public interface



Encapsulation is often based on information hiding, all the secrets of an object that do not contribute to its interface must be hidden

By encapsulating the data and methods, you create a black box that allows you to change the internal implementation of an object without affecting any of the code that uses it, as long as the interface remains the same

Do you need to know what is under the hood, to drive a car?

Do you need to tamper with the engine, to drive a car? (Better if you don't)

© 2019 ESTECO SpA

Polymorphism

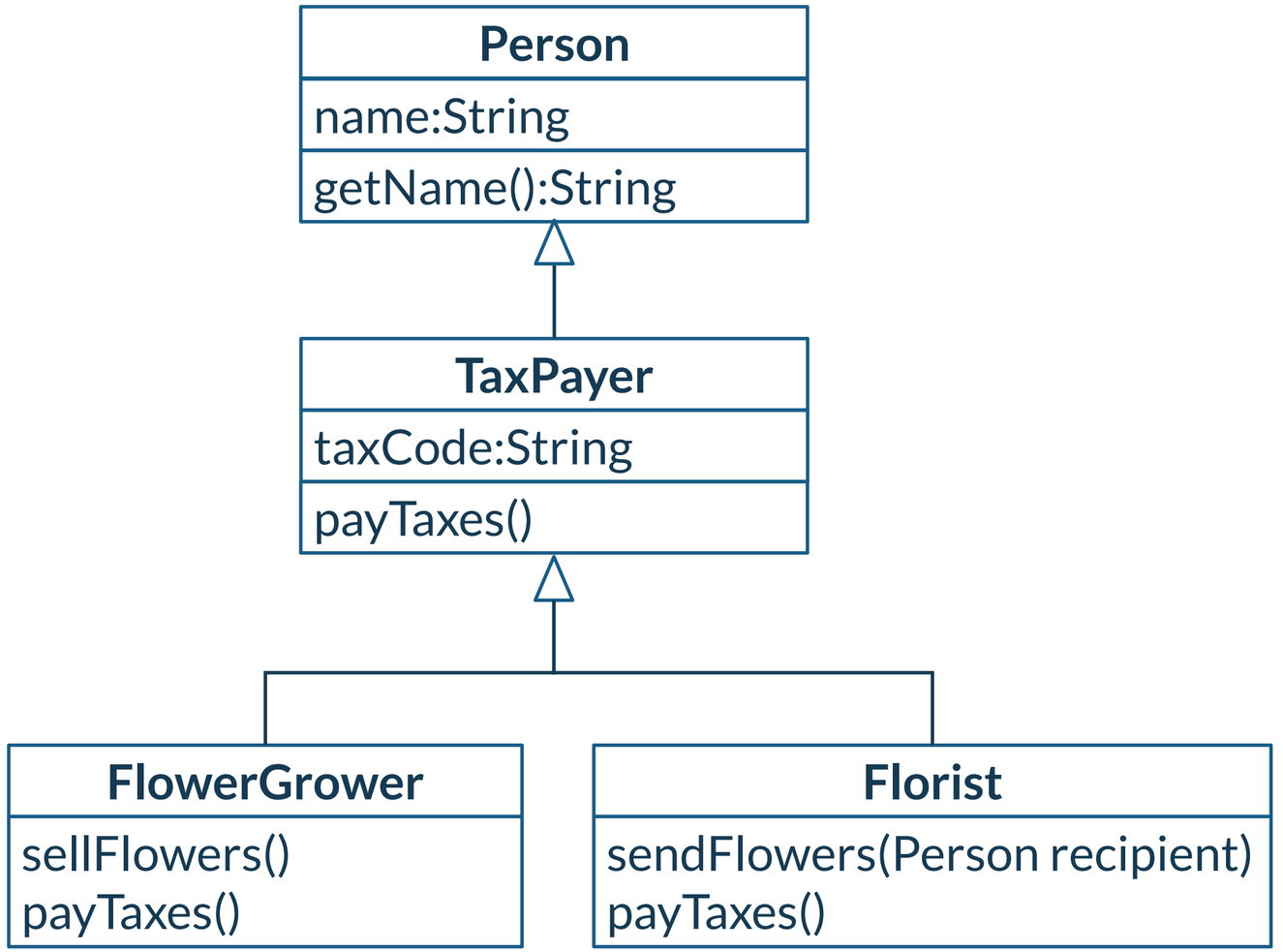
Polymorphism means many forms

It's a property of object-oriented languages that allows objects of different classes to be treated as objects of a common superclass

For example, a TaxPayer can take the form of FlowerGrower or Florist, you can ask them to pay taxes, they can pay taxes in very different ways, and you don't need to worry about that

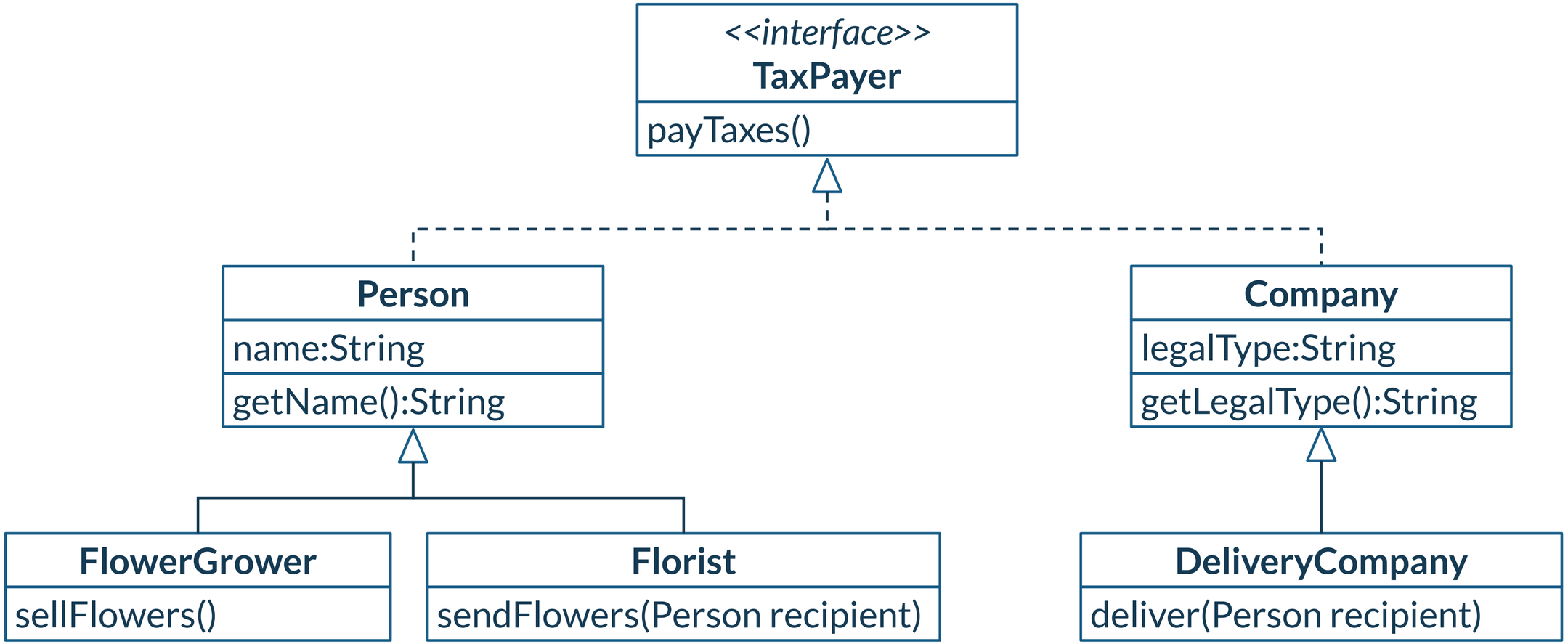
For simplicity, inheritance was used to create a class hierarchy and not for code reuse, here

In the next part you will see a different way to implement polymorphism in Java



© 2019 ESTECO SpA

Polymorphism in Java



Finally, everyone pays taxes even if they are outside the Person hierarchy

© 2019 ESTECO SpA

Additional content - Duck typing

⚠️ Python code

```
def processTaxPayer(tax_payer):
    tax_payer.payTaxes()

class Florist:
    def payTaxes(self):
        pass

class FlowerGrower:
    def payTaxes(self):
        pass

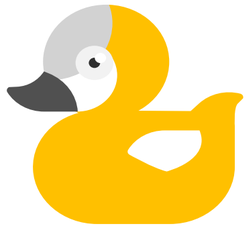
processTaxPayer(Florist())
processTaxPayer(FlowerGrower())
```

In dynamically typed languages, like Python, you declare variables without specifying a type, so `tax_payer` can reference objects of any type

`tax_payer` is a reference to an object of the right type, only if you can invoke `payTaxes()` on that object, otherwise you get a runtime error

Duck typing is based on the duck test

> "If it walks like a duck and it quacks like a duck, then it must be a duck"



© 2019 ESTECO SpA

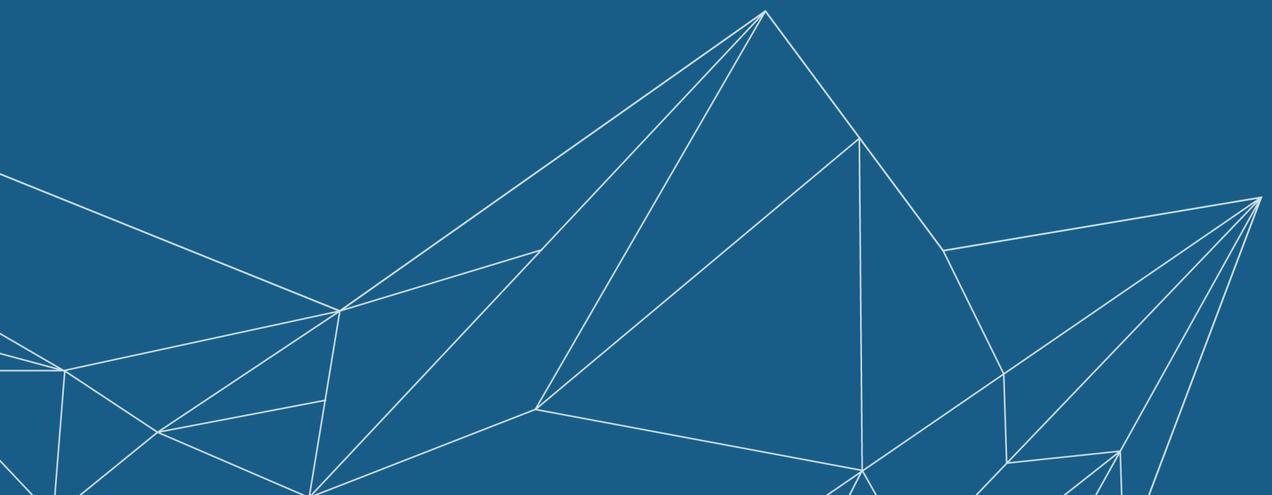
References

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

© 2019 ESTECO SpA



Extension and inheritance



© 2019 ESTECO SpA

Classes

Classes are used to define new types, once a class is defined, you can create new objects of that class, these objects are instances of that class, and that class is the type of these objects

As the words class and type can be used interchangeably, so object and instance can be used interchangeably too

A class is composed of

- (instance) methods, that define the operations that can be performed on its instances
- (instance) variables, that define the data that is associated to an object

Methods and instance variables are collectively known as members

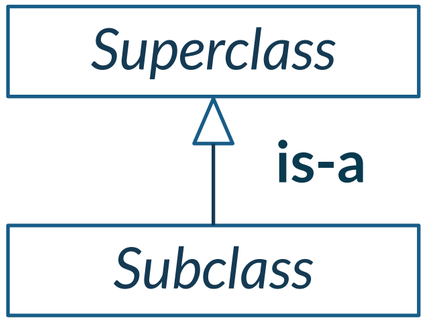
(Instance) methods define how other entities can interact with the objects of that class

(Instance) variables differentiate the behavior of different instances of the same class

© 2019 ESTECO SpA

Inheritance

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



In Java, that's class-based and statically-typed, inheritance define an is-a relationship

The is-a relationship is a "strong" relationship that creates hierarchies

```
package it.units.sdm;

public class Florist extends Shop {

    int inventory;

    void deliverFlower(String address) {
        …
        inventory--;
    }

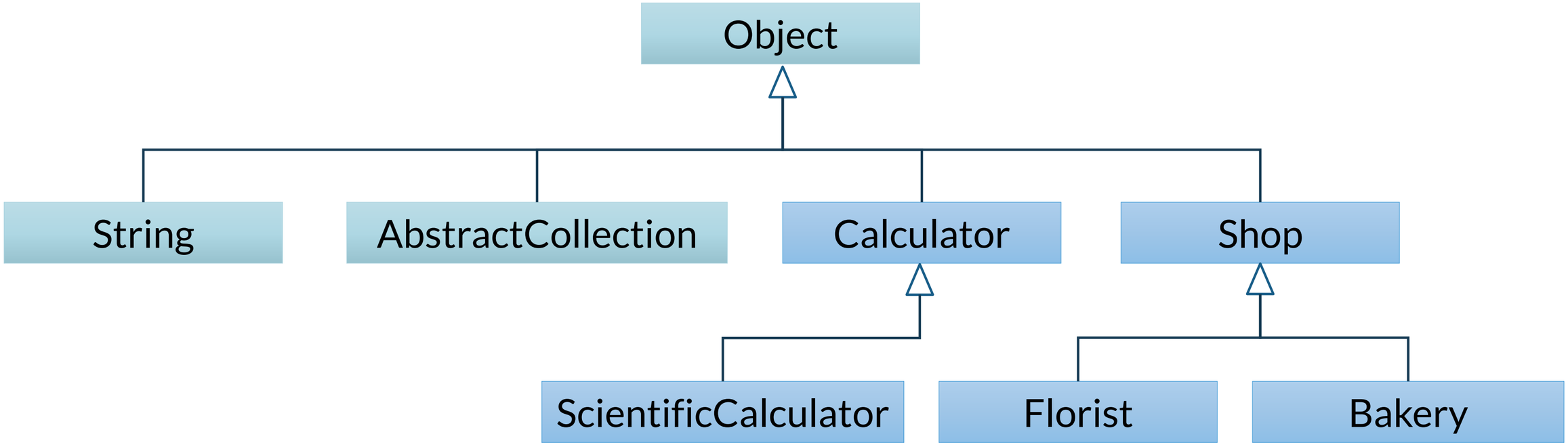
    void sellFlower() {
        inventory--;
    }
}
```

Java uses the extends keyword to indicate the extended superclass

extends implies you are extending the Shop class by adding some behavior

© 2019 ESTECO SpA

Java classes hierarchy

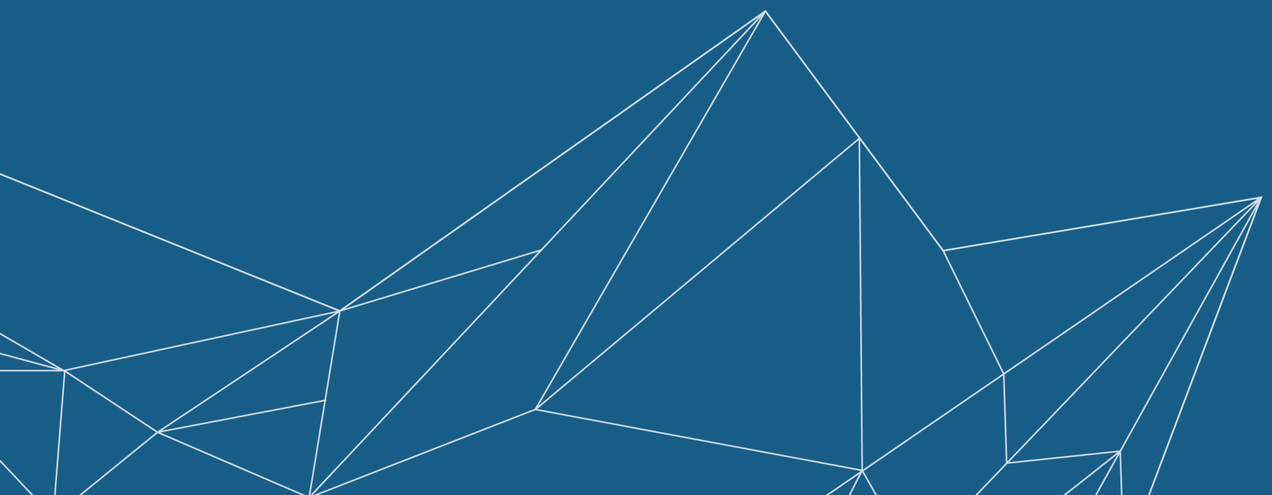


In Java, everything, but primitive values, is-an `Object`

© 2019 ESTECO SpA



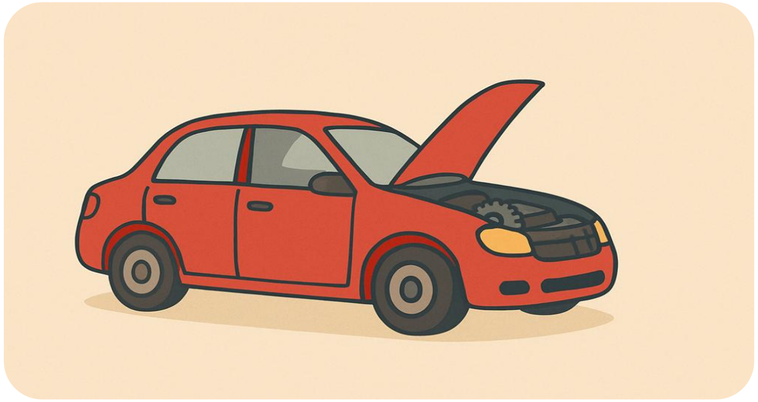
Access control and encapsulation



© 2019 ESTECO SpA

Encapsulation serves two purposes

1. separation of interface from implementation – abstraction focuses on the observable behavior of an object, whereas encapsulation focuses on the implementation of that behavior
2. data protection – internal details are kept private, they are exposed only through a public interface



© 2019 ESTECO SpA

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



© 2019 ESTECO SpA

Who can access this class? 1/3

com.esteco.sdm.Shop

```
package com.esteco.sdm;

public class Shop {

    private String name;

    public Shop(String name) {
        this.name = name;
    }

    public String getName() {
        return name;
    }
}
```

Anyone from any package!



© 2019 ESTECO SpA

Who can access this class? 2/3

com.esteco.sdm.Shop

```
package com.esteco.sdm;

class Shop {

    private String name;

    public Shop(String name) {
        this.name = name;
    }

    public String getName() {
        return name;
    }
}
```

Anyone from the com.esteco.sdm package!



© 2019 ESTECO SpA

Who can access this class? 3/3

com.esteco.sdm.Shop

```
package com.esteco.sdm;

private class Shop {

    private String name;

    public Shop(String name) {
        this.name = name;
    }

    public String getName() {
        return name;
    }
}
```

No one, this declaration is illegal!



© 2019 ESTECO SpA

Who can instantiate this class? 1/3

com.esteco.sdm.Shop

```
package com.esteco.sdm;

public class Shop {

    private String name;

    public Shop(String name) {
        this.name = name;
    }

    public String getName() {
        return name;
    }
}
```

Anyone from any package!



© 2019 ESTECO SpA

Who can instantiate this class? 2/3

com.esteco.sdm.Shop

```
package com.esteco.sdm;

public class Shop {

    private String name;

    private Shop(String name) {
        this.name = name;
    }

    public String getName() {
        return name;
    }
}
```

Only someone within this class!



© 2019 ESTECO SpA

Who can instantiate this class? 3/3

com.esteco.sdm.Shop

```
package com.esteco.sdm;

class Shop {

    private String name;

?   public Shop(String name) {
        this.name = name;
    }

    public String getName() {
        return name;
    }
}
```

Only someone within the same package!



© 2019 ESTECO SpA

Who can access the name variable? 1/3

com.esteco.sdm.Shop

```
package com.esteco.sdm;

public class Shop {

    private String name;

    public Shop(String name) {
        this.name = name;
    }

    public String getName() {
        return name;
    }
}
```

Only someone within the same class!



© 2019 ESTECO SpA

Who can access the name variable? 2/3

com.esteco.sdm.Shop

```
package com.esteco.sdm;

public class Shop {

    public String name;

    public Shop(String name) {
        this.name = name;
    }

    public String getName() {
        return name;
    }
}
```

Everyone can read and write the name variable

In general, very dangerous, a serious violation of data hiding and encapsulation



© 2019 ESTECO SpA

Who can access the name variable? 3/3

com.esteco.sdm.Shop

```
package com.esteco.sdm;

public class Shop {

    public final String name;

    public Shop(String name) {
        this.name = name;
    }

    public String getName() {
        return name;
    }
}
```

Everyone can read the name variable

But only the constructor can assign it!



© 2019 ESTECO SpA

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

com.esteco.sdm.Shop

```
package com.esteco.sdm;

public class Shop {

    private final String name;

    public Shop(String name) {
        this.name = name;
    }

    public String getName() {
        return name;
    }
}
```

Everyone can invoke the getName() method



© 2019 ESTECO SpA

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

com.esteco.sdm.Shop

```
package com.esteco.sdm;

public class Shop {

    private final String name;

    public Shop(String name) {
        this.name = name;
    }

    private String getName() {
        return name;
    }
}
```

No one outside the Shop class



© 2019 ESTECO SpA

Who can invoke the `getName()` method? 3/4

com.esteco.sdm.Shop

```
package com.esteco.sdm;

public class Shop {

    private final String name;

    public Shop(String name) {
        this.name = name;
    }

    protected String getName() {
        return name;
    }
}
```

Anyone from the same package,
or from any subclass.



© 2019 ESTECO SpA

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

com.esteco.sdm.Shop

```
package com.esteco.sdm;

class Shop {

    private final String name;

    public Shop(String name) {
        this.name = name;
    }

    public String getName() {
        return name;
    }
}
```

Everyone that can get a reference to a Shop object

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



© 2019 ESTECO SpA

You cannot reduce the access level of overridden methods

com.esteco.sdm.Shop

```
public class Shop {

    private String name;
    private boolean open;

    public Shop(String name) {
        this.name = name;
    }

    public void open() {
        open = true;
    }

    public void close() {
        open = false;
    }
}
```

com.esteco.sdm.Florist

```
class Florist extends Shop {

    private int inventory;

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

    @Override
    protected void open() {
        super.open();
        turnOnSign();
    }
    …

}
```



© 2019 ESTECO SpA

Tips on choosing an access level (to implement encapsulation)

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

Finally, objects want their own privacy

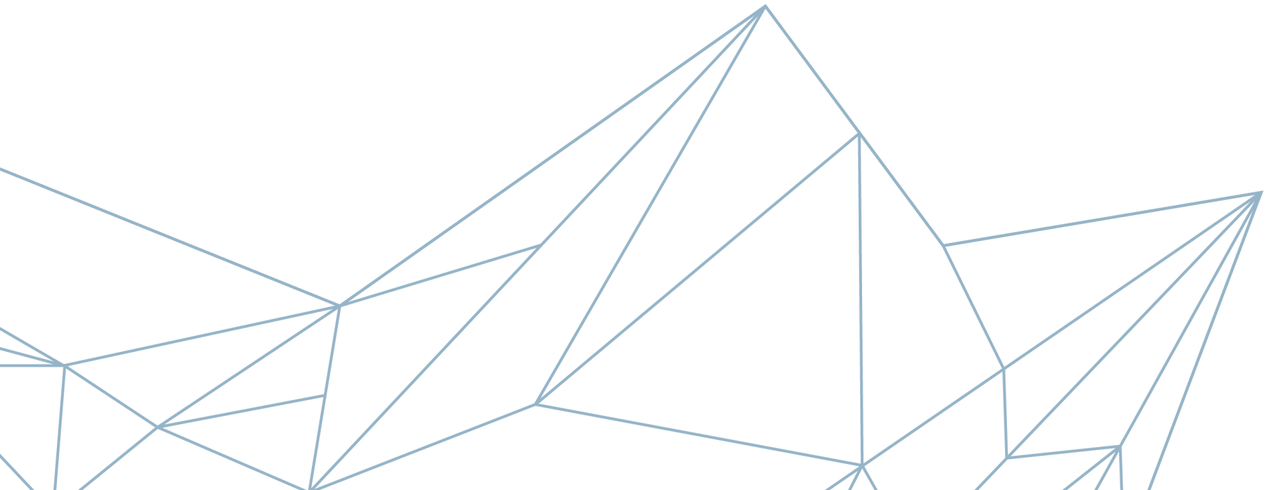
© 2019 ESTECO SpA





The `final` and `abstract` modifiers

A little detour from encapsulation



© 2019 ESTECO SpA

Final classes

com.esteco.sdm.Shop

```
package com.esteco.sdm;

public final class Shop {

    private String name;

    public Shop(String name) {
        this.name = name;
    }

    public String getName() {
        return name;
    }
}
```

Final classes cannot be extended

String is an example of final class



© 2019 ESTECO SpA

Final methods

com.esteco.sdm.Shop

```
public class Shop {

    private String name;
    private boolean open;

    public Shop(String name) {
        this.name = name;
    }

    final void open() {
        open = true;
    }

    final void close() {
        open = false;
    }
}
```

com.esteco.sdm.Florist

```
class Florist extends Shop {

    int inventory;

    public Florist(String name) {
        super(name);
    }

    @Override
    void open() {
        super.open();
        turnOnSign();
    }
    …
}
```

Final methods cannot be overridden



© 2019 ESTECO SpA

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 Shop and overriding open()?

What if

- you are given the Shop class by a third-party,
- you extend Shop,
- you override the open() method, and
- you forget to invoke super.open()

com.esteco.sdm.Florist

```
class Florist extends Shop {

    private int inventory;

    public Florist(String name) {
        super(name);
    }

    @Override
    public void open() {
        super.open();
        turnOnSign();
    }
    …
}
```



© 2019 ESTECO SpA

Abstract methods 1/2

If the open() method is final in Shop, how do we allow subclasses to do something when a shop is opened?

opened() is the entry point for subclasses to perform operations after a shop is opened

open() is defining a protocol, i.e., how the Shop class works and how it should be extended

com.esteco.sdm.Shop

```
public class Shop {

    private String name;
    private boolean open;

    public Shop(String name) {
        this.model = name;
    }

    final void open() {
        open = true;
        turnedOn();
    }

    protected void turnedOn() {
    }
}
```



© 2019 ESTECO SpA

Abstract methods 2/2

What if you want all the subclasses to be forced to define the opened() method?

You define opened() as an abstract method,
abstract methods have no implementation, therefore, the Shop class cannot be instantiated, and it becomes abstract too

Abstract classes cannot be instantiated, but they can be extended by subclasses

Subclasses that don't implement all the abstract methods are abstract too

com.esteco.sdm.Shop

```
public abstract class Shop {

    private String name;
    private boolean open;

    protected Shop(String name) {
        this.name = name;
    }

    public final void open() {
        open = true;
        opened();
    }

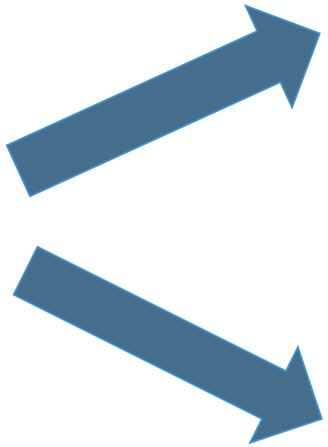
    protected abstract void opened();
}
```



© 2019 ESTECO SpA

Final and abstract

You can apply the final and abstract modifiers to both classes and methods, but 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

© 2019 ESTECO SpA



Interfaces and polymorphism



© 2019 ESTECO SpA

Interfaces

it.units.sdm.Courier

```
public interface Courier {

  void deliver(String from, String to);
}
```

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

Java interfaces are a crucial component of abstraction and together with inheritance they allow the implementation of polymorphism

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

it.units.sdm.Florist

```
public class Florist extends Shop {

  private final Courier courier;
  private int inventory;
  …

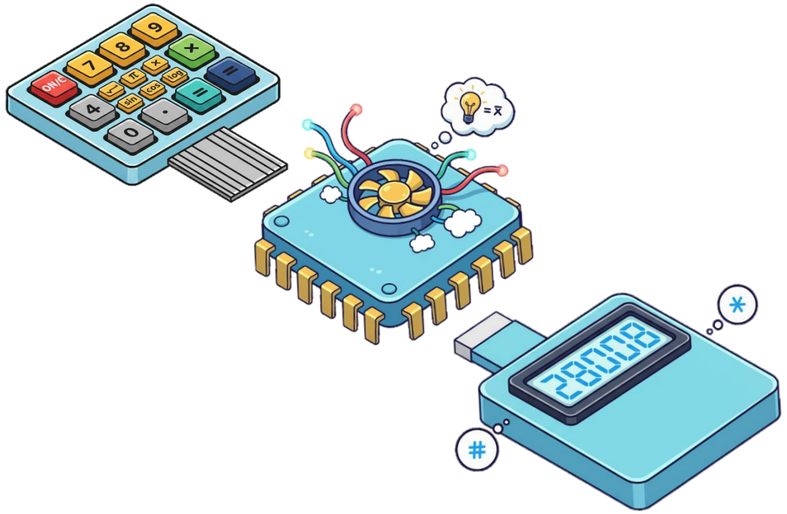
  Florist(String name, String address, Courier courier) {
      super(name, address);
      this.courier = courier;
  }

  void deliverFlowers(String address) {
    courier.deliverFromTo(getAddress(), address);
    inventory--;
  }
}
```



© 2019 ESTECO SpA

A less abstract (not final) example



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);
    }


     void twoPressed() {
        string += "2";
        display.display(string);
    }
    …
}
```

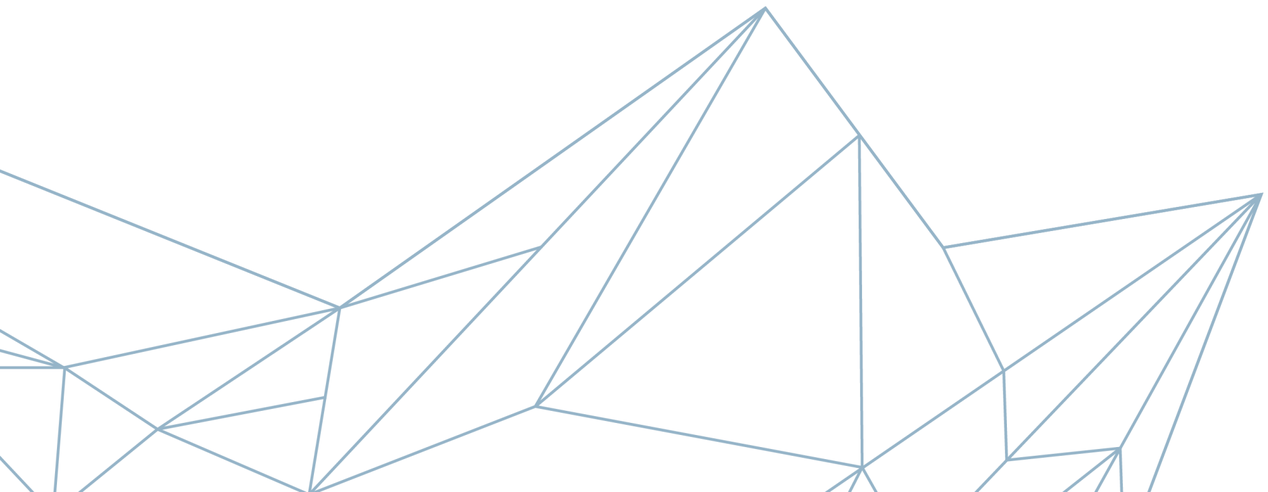
© 2019 ESTECO SpA





Why ducks don’t care about interfaces

Type definition in dynamically typed languages



© 2019 ESTECO SpA

Duck typing

⚠️ Python code ahead

```
class Zoo:
    def make_it_speak(self, animal):
        print(animal.speak())

class Dog:
    def speak(self): return "woof"

class Cat:
    def speak(self): return "meow"


zoo = Zoo()
zoo.make_it_speak(Dog())  # woof
zoo.make_it_speak(Cat())  # meow
```

The Zoo class defines a convention: to speak in a Zoo you need to implement a speak() method

Dog and Cat classes depend on the convention defined by Zoo so, in the end, they depends on the Zoo class

To add a new pet, e.g., a Duck, you just need to follow the same convention, you don't need to change Zoo, imagine Zoo is in another module, or in another package provided by a third party



© 2019 ESTECO SpA

A Java implementation

```
class Dog {
    public String speak() {return "woof";}
}

class Cat {
    public String speak() {return "meow";}
}

public class Zoo {

    void makeItSpeak(Dog d) {
        IO.println(d.speak());
    }

    void makeItSpeak(Cat c) {
        IO.println(c.speak());
    }

}
void main() {
    makeItSpeak(new Dog()); //woof
    makeItSpeak(new Cat()); //meow
}
```

The literal translation to Java, that seems to mimic the previous one, does not define any convention

The Zoo class is now coupled with both Cat and Dog classes, any change on one of these two classes might require some changes in the Zoo class, furthermore you cannot add any new pet, e.g., Duck, without modifications to the Zoo class

Finally, this Java implementation is not equivalent to the Python example



© 2019 ESTECO SpA

Another Java implementation

```
public class Zoo {

    abstract static class Animal {
        public abstract String speak();
    }

    void makeItSpeak(Animal a) {
        IO.println(a.speak());
    }

    void main() {
        makeItSpeak(new Dog()); //woof
        makeItSpeak(new Cat()); //meow
    }
}

class Dog extends Zoo.Animal {
    public String speak() {return "woof";}
}

class Cat extends Zoo.Animal {
    public String speak() {return "meow";}
}
```

Now the abstract class Animal is defining the convention, that must be followed by the subclasses

We inverted the dependency; Dog and Cat depends on an abstraction, the abstract Animal class

Why should Animal belong to Zoo?

Because Zoo is the user, is the client of Animal, so it has the right to specify how Animal is defined

“If you want to speak at the Zoo, you must be an Animal!”

This implementation is not yet equivalent to the Python example, i. e., what if we want to bring to the Zoo a RobotDog that cannot extend Animal because it belongs to another inheritance hierarchy?



© 2019 ESTECO SpA

A final Java implementation

```
public class Zoo {

    interface Pet {
        String speak();
    }

    void makeItSpeak(Pet p) {
        IO.println(p.speak());
    }

    void main() {
        makeItSpeak(new Dog()); //woof
        makeItSpeak(new RobotDog()); //wauf
    }
}

class Dog extends Animal implements Zoo.Pet {
    public String speak() {return "woof";}
}

class RobotDog extends Robot implements Zoo.Pet {
    public String speak() {return "wauf";}
}
```

By defining an interface, you allow any Pet to speak at the Zoo

In dynamically typed languages, e.g., Python, there are no interfaces because you don't need them



© 2019 ESTECO SpA

Conclusion

```java
public class Zoo {

    interface Pet {
        String speak();
    }

    void makeItSpeak(Pe
        IO.println(p.sp
    }

    void main() {
        makeItSpeak(new
        makeItSpeak(new
    }
}

class Dog extends Animal implements Zoo.Pet {
    public String speak() {return "woof";}
}

class RobotDog extends Robot implements Zoo.Pet {
    public String speak() {return "wauf";}
}
```

```python
class Dog:
    def speak(self): return "woof"

class Cat:
    def speak(self): return "meow"

class Zoo:
    def make_it_speak(self, animal):
        print(animal.speak())

zoo = Zoo()
zoo.make_it_speak(Dog())   # woof
zoo.make_it_speak(Cat())   # meow
```

In dynamically typed languages, polymorphism emerge as behavioral compatibility, not type hierarchy — the type system is just *flexible enough to let it happen* — if it quacks like a duck, it works

"In dynamically typed languages, behavior defines type"

What you implement is what you are ↔ an interface is used to specify what a class must implement

In statically typed languages like Java, polymorphism must be made explicit through interfaces (or method overriding), which define contracts the compiler can verify



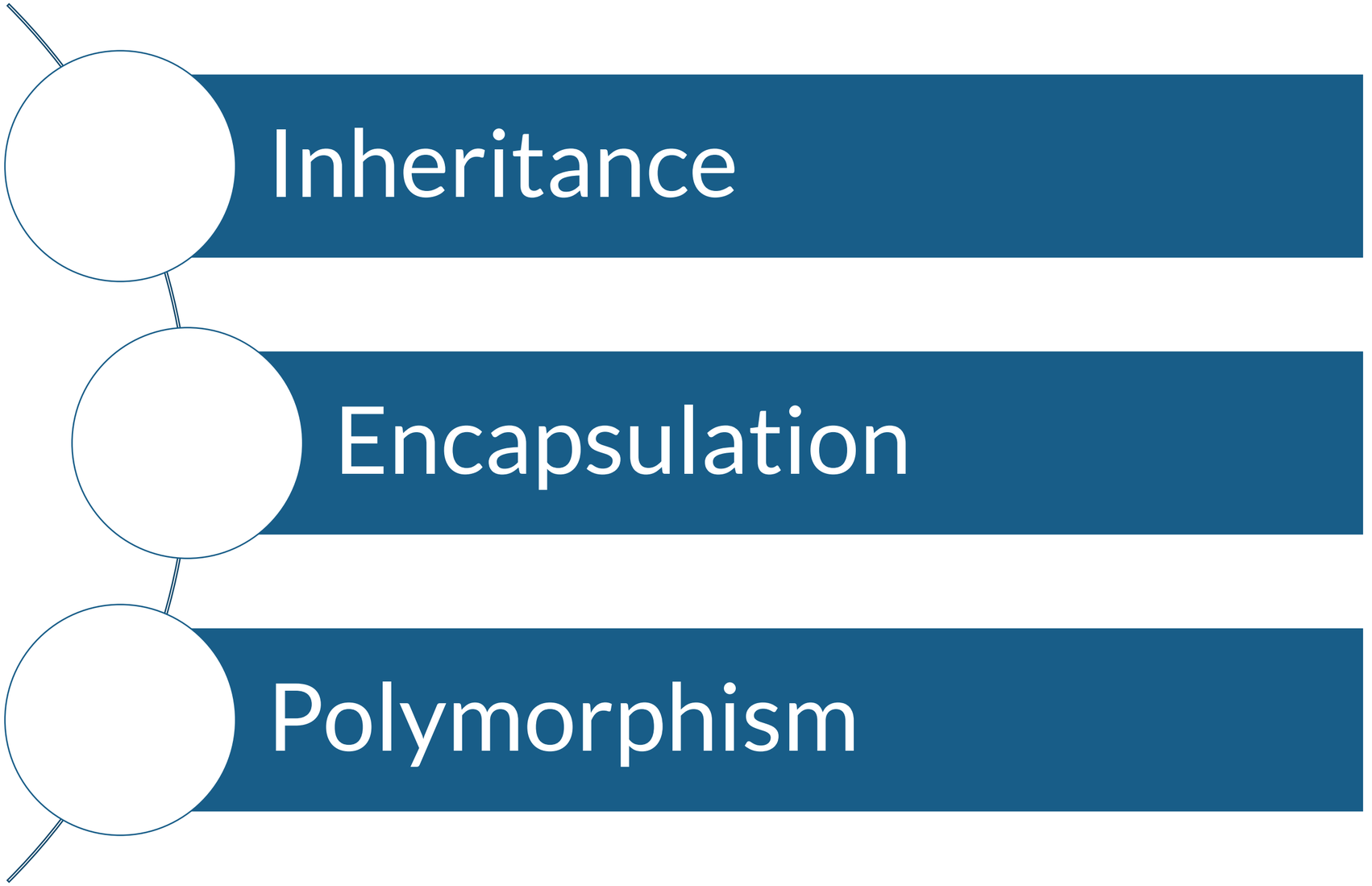
© 2019 ESTECO SpA



The Three Musketeers of OOP



© 2019 ESTECO SpA

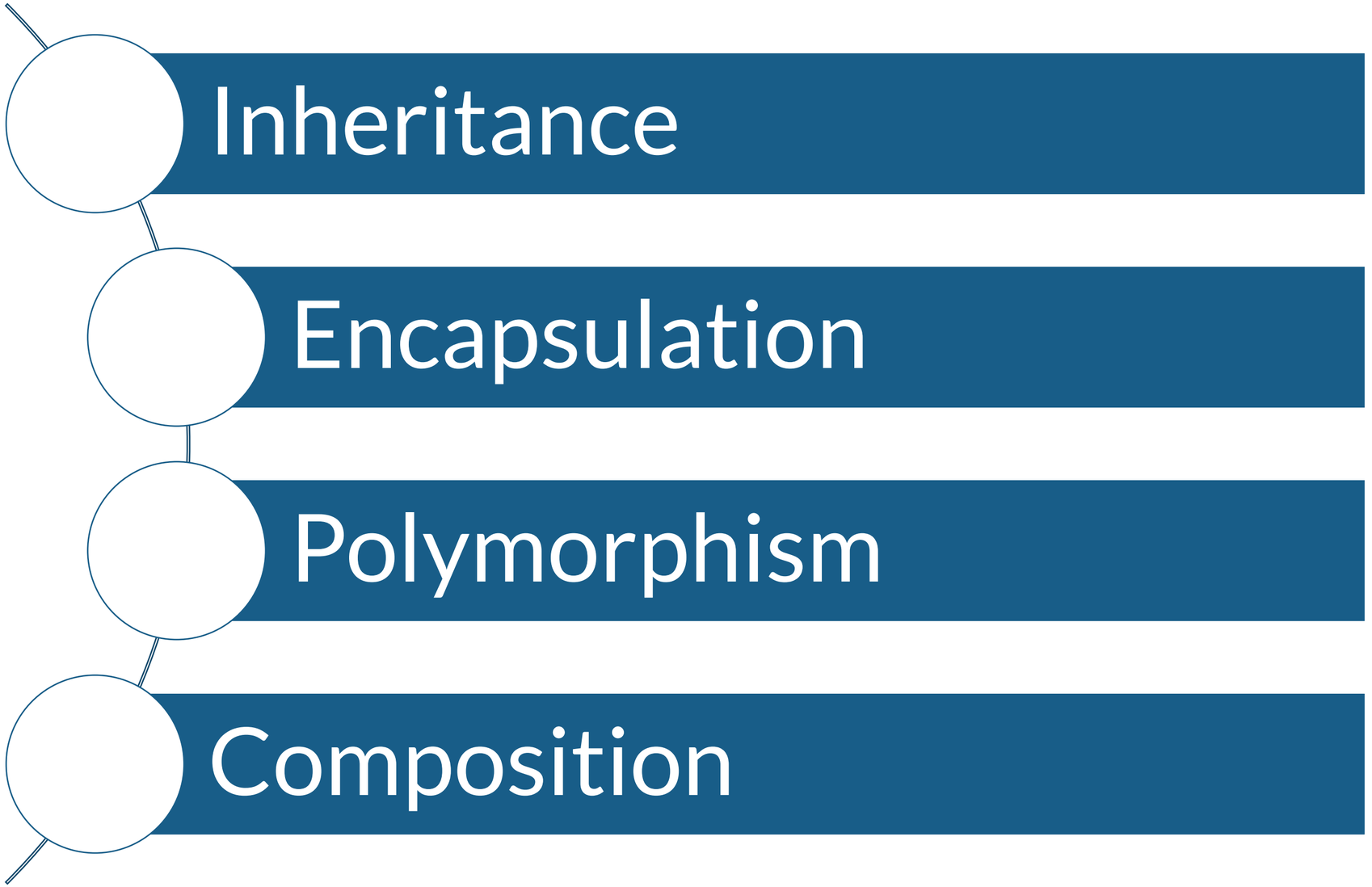


© 2019 ESTECO SpA

What about the fourth Musketeer?



© 2019 ESTECO SpA



© 2019 ESTECO SpA

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

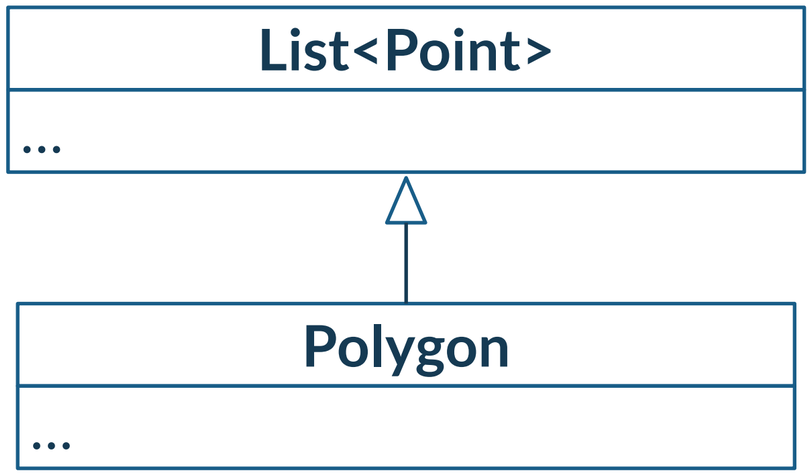
The enhancements made by the subclass are primarily additive

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



© 2019 ESTECO SpA

Both classes are in the same logical domain



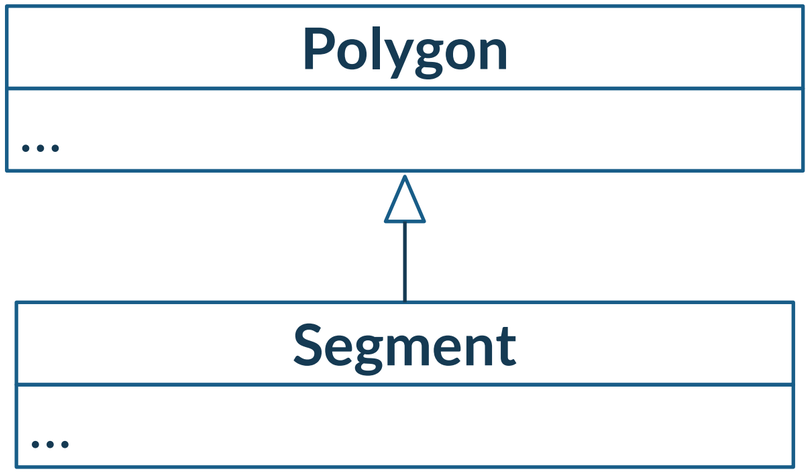
Implementation class

Domain class

Maybe a polygon is composed by a list of vertexes, and some other attributes, rather than being an extension of a list

© 2019 ESTECO SpA

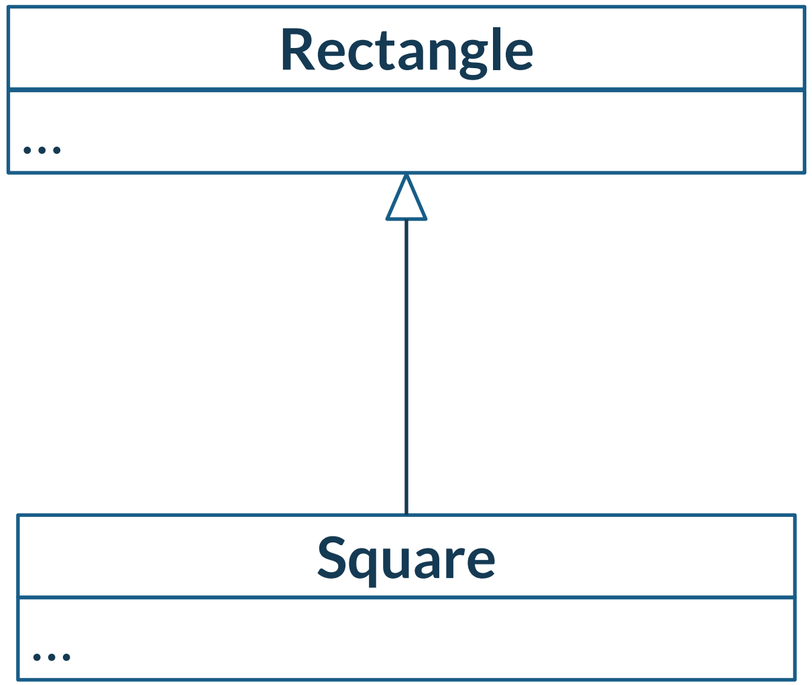
The subclass is a proper subtype of the superclass



There is something wrong at the semantic level

© 2019 ESTECO SpA

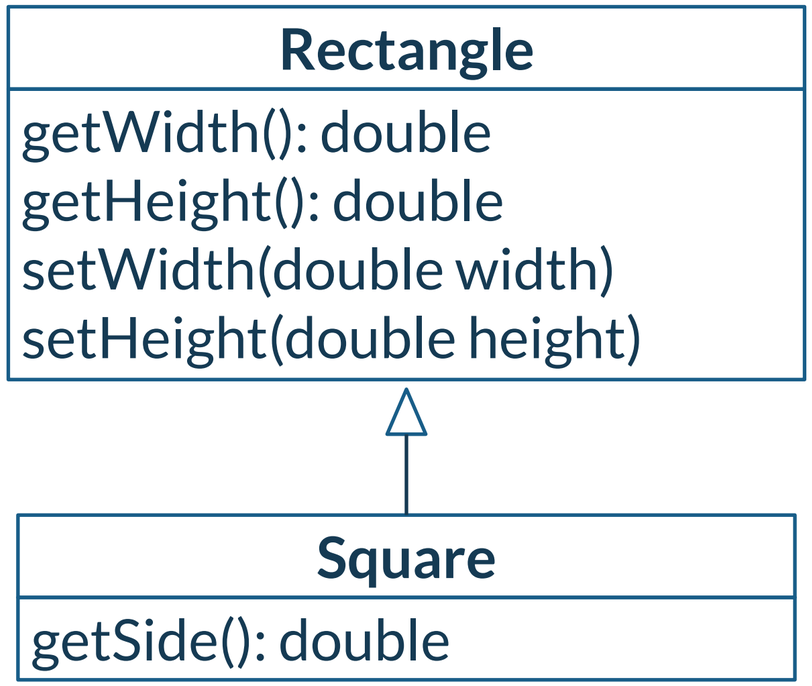
The implementation of the superclass is necessary for the subclass



Is a square a rectangle?

© 2019 ESTECO SpA

The implementation of the superclass is necessary for the subclass

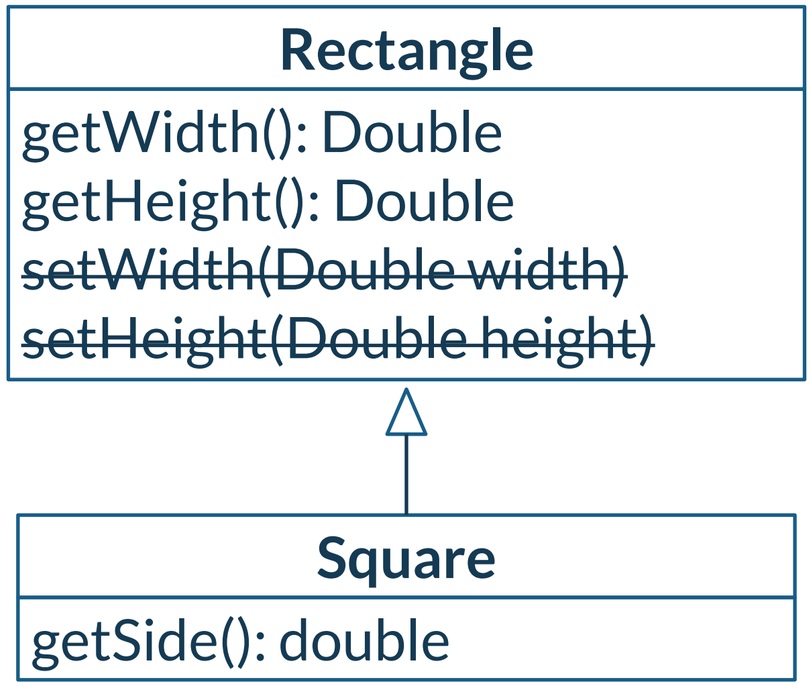


Is a square a rectangle?

It could depend on the domain of the problem!

© 2019 ESTECO SpA

The implementation of the superclass is necessary for the subclass

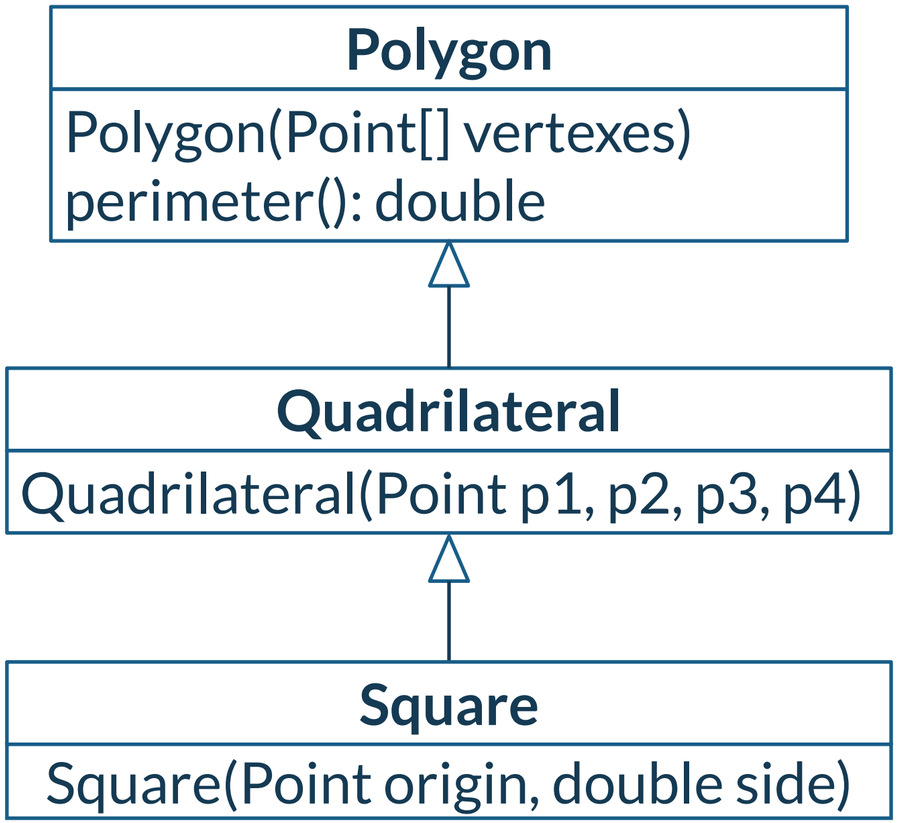


Is a square a rectangle?

Depends on how we define the Rectangle, this case is violating the Liskov substitution principle

© 2019 ESTECO SpA

The enhancements made by the subclass are primarily additive



Do you really need this hierarchy?

Maybe not, you arejust adding constraints on the creation of an object

A good opportunity to use a builder method

© 2019 ESTECO SpA

The enhancements made by the subclass are primarily additive

Polygon
Polygon(Point[] vertexes) perimeter(): double quadrilateral(Point p1, p2, p3, p4): Polygon square(Point origin, double side): Polygon



© 2019 ESTECO SpA

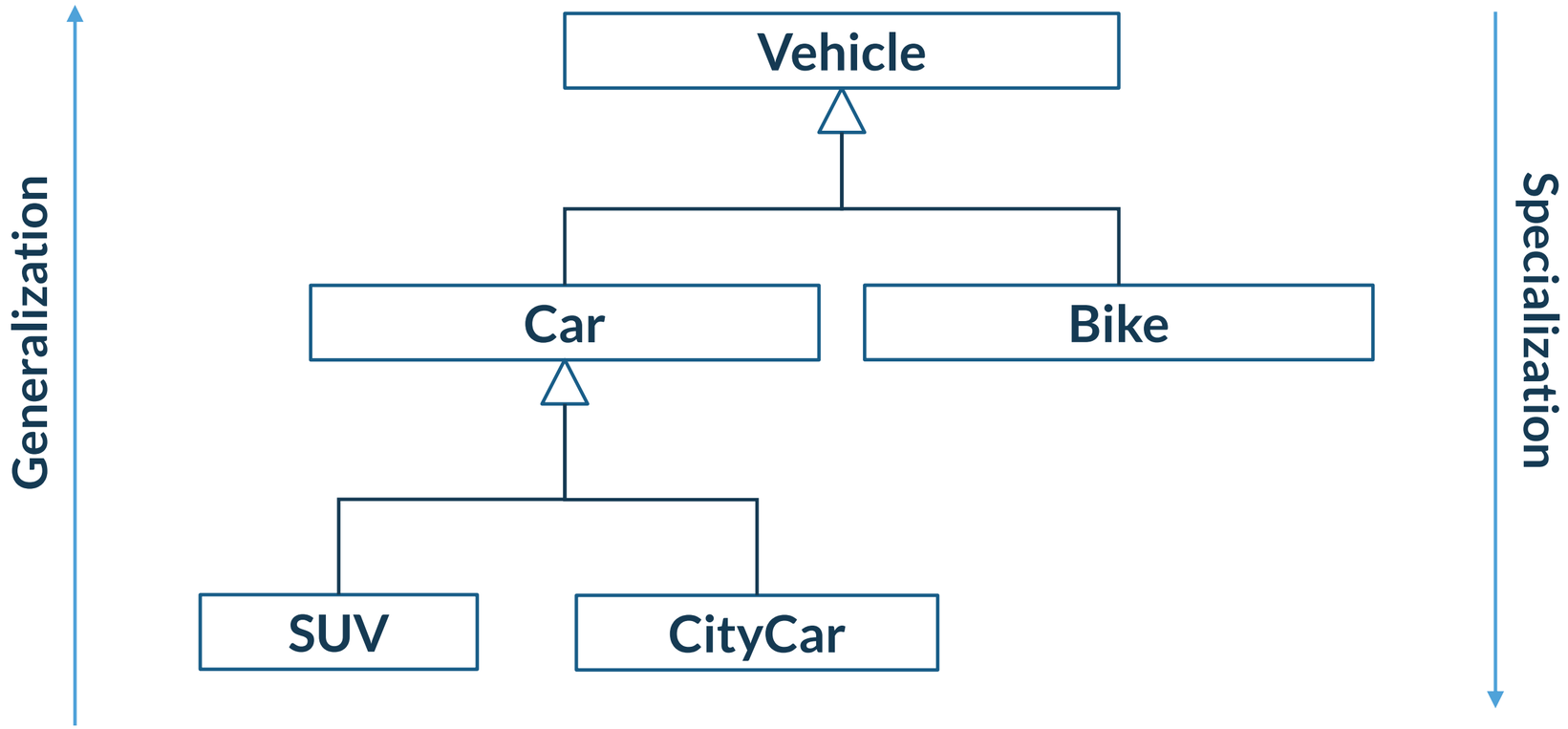
Additional content – 5 forms of inheritance

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

© 2019 ESTECO SpA

Additional content – Specialization/Generalization

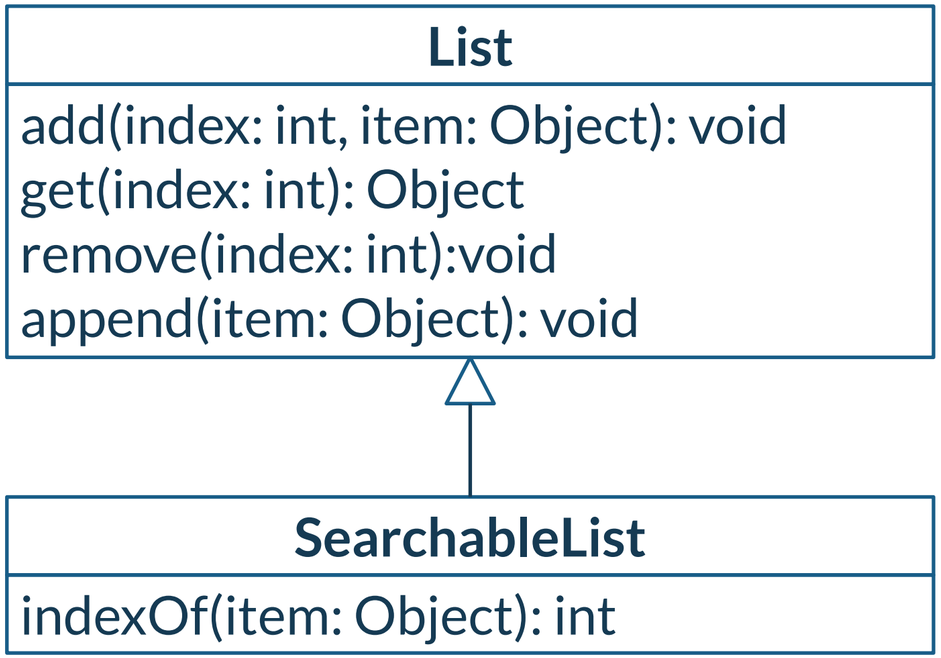
- Often used in domain modelling
- The subclass is a specialization of the general parent class (1-to-1)
- The superclass is a generalization of the subclasses (1-to-many)



© 2019 ESTECO SpA

Additional content – Extension

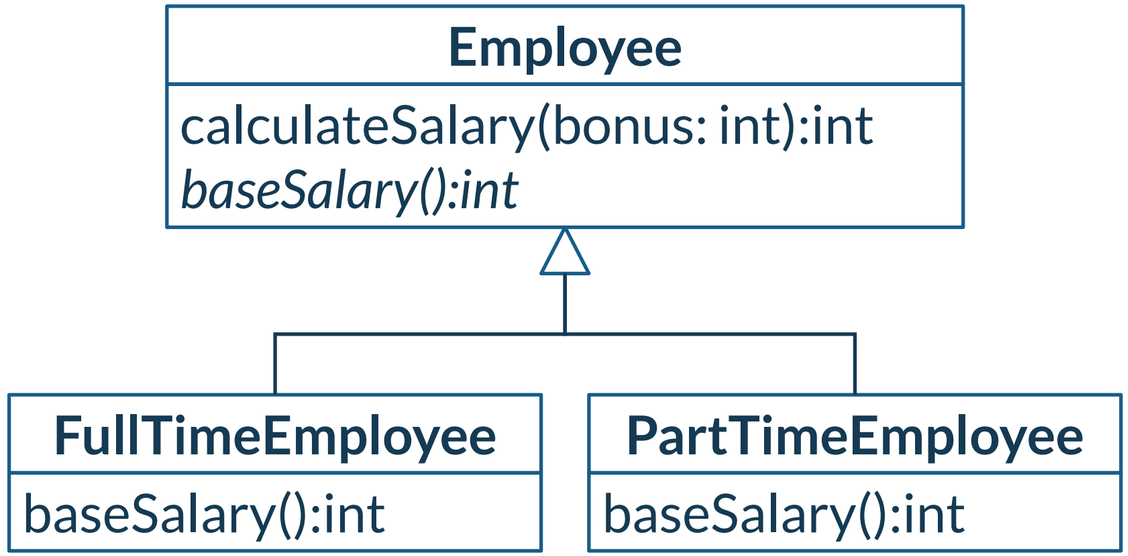
- The subclass adds new functionalities to the parent class
- It adds new methods to those of the parent
- In general, you will use the subclass as substitute of the superclass



© 2019 ESTECO SpA

Additional content – 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.specification.Employee

```
public abstract class Employee {

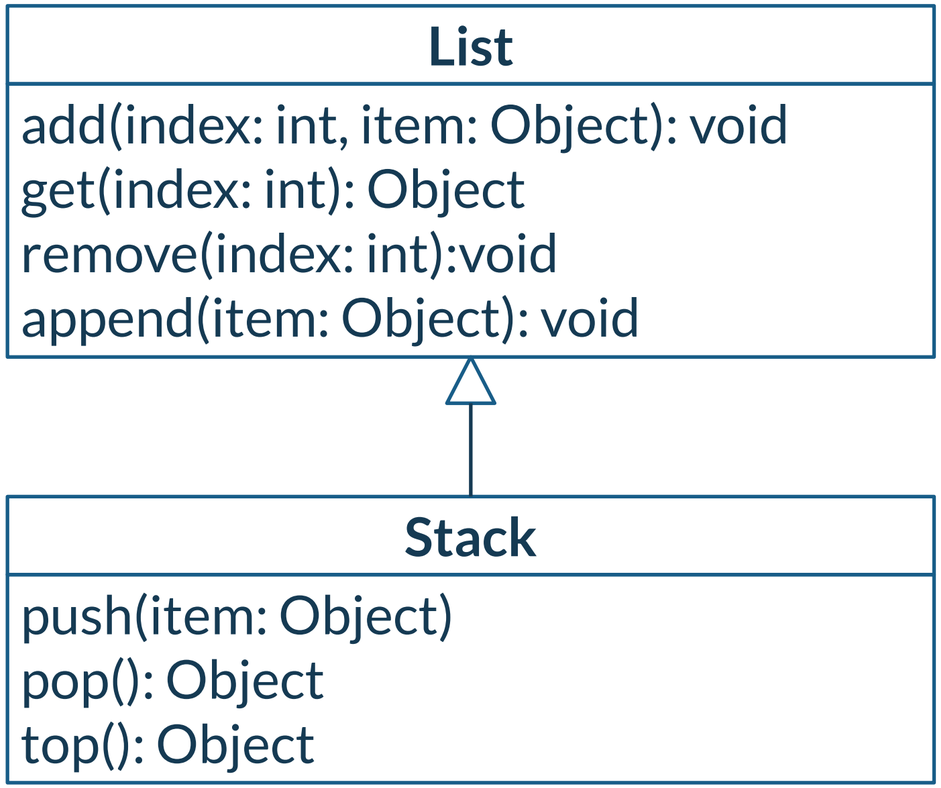
    protected abstract int baseSalary();

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

© 2019 ESTECO SpA

Additional content – Construction

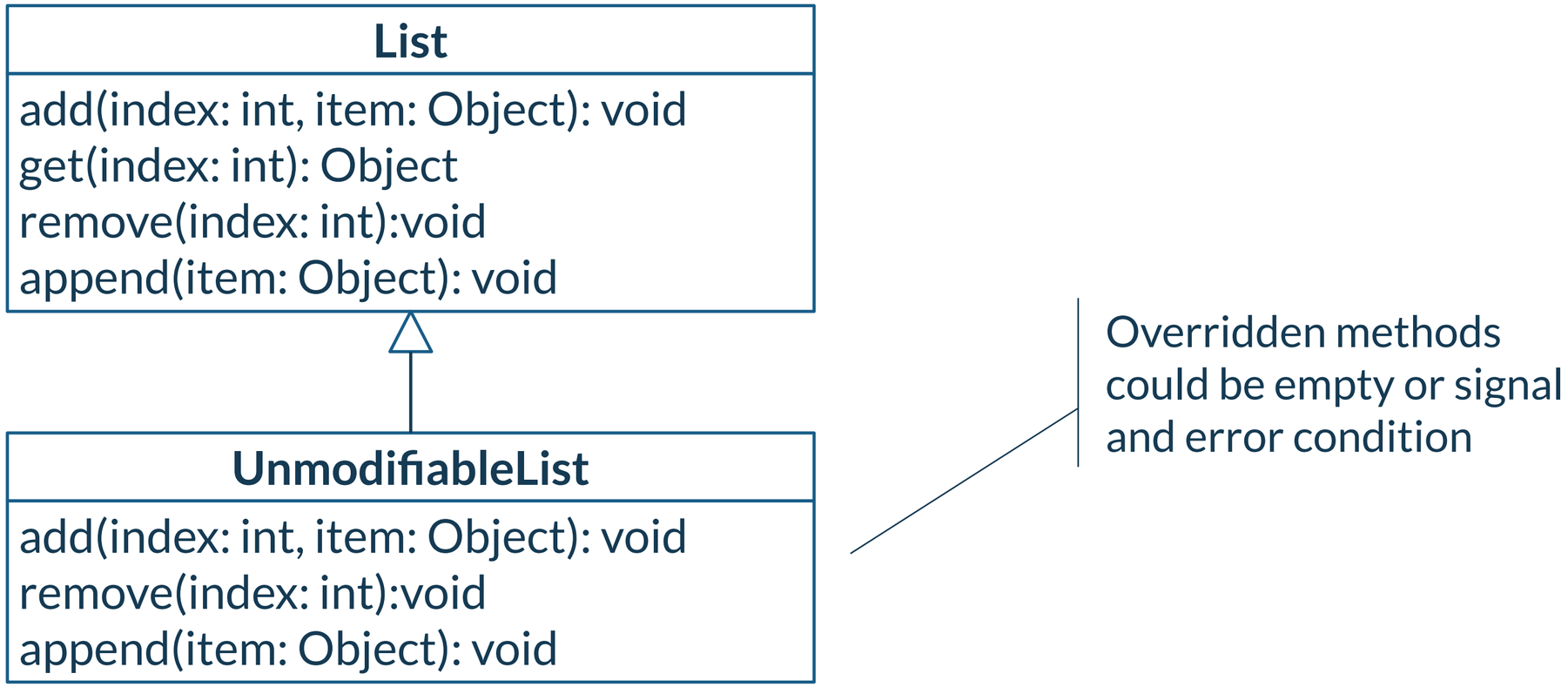
- The subclass and superclass share some behavior
- In general, you will not use the subclass as substitute of the superclass
- Composition could play a better role here!



© 2019 ESTECO SpA

Additional content – Limitation

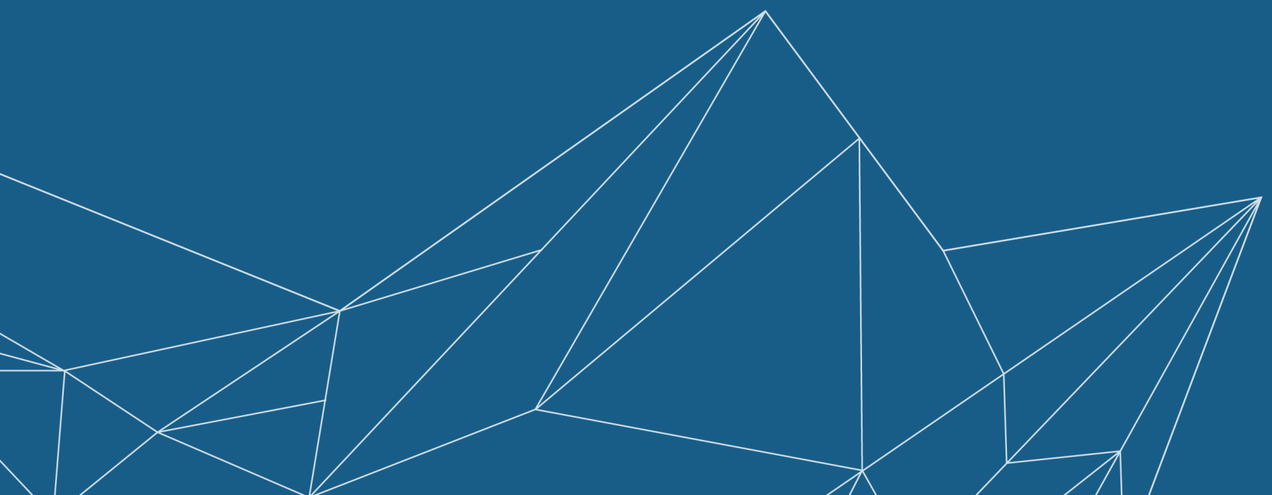
- The subclass restrict some properties of the superclass by overriding some methods
- In general, you cannot use the subclass as substitute of the superclass



© 2019 ESTECO SpA



Object equivalence & identity



© 2019 ESTECO SpA

Object.equals()

```
public boolean equals(Object obj)
```

Indicates whether some other object is “equal to” this one

The equals method implements an equivalence relation on non-null object references:

- It is *reflexive*: for any non-null reference value x, x.equals(x) should return true
- It is *symmetric*: for any non-null reference values x and y, x.equals(y) should return true if and only if y.equals(x) returns true
- It is *transitive*: for any non-null reference values x, y, and z, if x.equals(y) returns true and y.equals(z) returns true, then x.equals(z) should return true
- It is *consistent*: for any non-null reference values x and y, multiple invocations of x.equals(y) consistently return true or consistently return false, provided no information used in equals comparisons on the objects is modified
- For any non-null reference value x, x.equals(null) should return false

An equivalence relation partitions the elements it operates on into *equivalence classes*; all the members of an equivalence class are equal to each other. Members of an equivalence class are substitutable for each other, at least for some purposes

The default implementation is the most discriminating possible equivalence relation on objects; that is, for any non-null reference values x and y, this method returns true if and only if x and y refer to the same object (x == y has the value true). In other words, under the reference equality equivalence relation, each equivalence class only has a single element



© 2019 ESTECO SpA

Digression - Equivalence of Strings

StringEquals.java

```
static void main() {
  String s1 = "Java is great!";
  String s2 = "Java" + " is great!";
  String a = "Java";
  String b = " is great!";
  String s3 = a + b;

  IO.println("s1: " + s1);
  IO.println("s2: " + s2);
  IO.println("s3: " + s3);

  IO.println("s1.equals(s2): " + s1.equals(s2));
  IO.println("s1 == s2: " + (s1 == s2));
  IO.println("s1.equals(s3): " + s1.equals(s3));
  IO.println("s1 == s3: " + (s1 == s3));
}
```

```
$ java StringEquals.java
s1: Java is great!
s2: Java is great!
s3: Java is great!

s1.equals(s2): true
s1 == s2: true

s1.equals(s3): true
s1 == s3: false
```

Since Strings are objects, with == you test referential equality

What if a and b are declared final?



© 2019 ESTECO SpA

Comparable

The `Comparable` interface is implemented by classes that want to define a “natural” order.

The `String` class implements `Comparable`, so String objects can be compared one with each other

```
static void main() {
    Set<String> citiesOfFvg = Set.of("Trieste", "Udine", "Gorizia", "Pordenone");

    TreeSet<String> naturalOrder = new TreeSet<>(citiesOfFvg);

    IO.println(naturalOrder);
}
```

```
[Gorizia, Pordenone, Trieste, Udine]
```

The result is the list of the cities in the `String` lexicographical order



© 2019 ESTECO SpA

Comparator 1/3

By implementing the `Comparator` interface we can define an order for classes that doesn't implement `Comparable`, or we can override the "natural" order of classes that already implements `Comparable`

```
static void main() {
    Set<String> citiesOfFvg = Set.of("Trieste", "Udine", "Gorizia", "Pordenone");

    TreeSet<String> lengthOrder = new TreeSet<>(new Comparator<String>() {
        @Override
        public int compare(String o1, String o2) {
            return o1.length() - o2.length();
        }
    });
    lengthOrder.addAll(citiesOfFvg);

    IO.println(lengthOrder);
}
```

```
[Udine, Trieste, Pordenone]
```

The result is the list of the cities in order of name length

What about Gorizia?

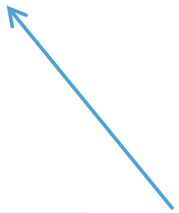


© 2019 ESTECO SpA

Comparator 2/3

```
Comparator<String> comparator = new Comparator<>() {
    @Override
    public int compare(String o1, String o2) {
        return o1.length() - o2.length();
    }
};
TreeSet<String> triesteOnly = new TreeSet<>(comparator);
triesteOnly.add("Trieste");

IO.println(triesteOnly.contains("Gorizia")); // true
```



The result is `true` because, in a `TreeSet`, the order relation defined by the comparator is used as equivalence relation

© 2019 ESTECO SpA

Comparator 3/3

```
public static void main(String[] args) {
    Set<String> citiesOfFvg = Set.of("Trieste", "Udine", "Gorizia", "Pordenone");

    Comparator<String> comparator = new Comparator<>() {
        @Override
        public int compare(String o1, String o2) {
            return o1.length() == o2.length() ? o1.compareTo(o2) : o1.length() - o2.length();
        }
    };

    TreeSet<String> lengthOrder = new TreeSet<>(comparator);
    lengthOrder.addAll(citiesOfFvg);

    System.out.println(lengthOrder);
}
```

```
[Udine, Gorizia, Trieste, Pordenone]
```

Gorizia is back!



© 2019 ESTECO SpA

Equals and ordering

The implementations of the methods of the `Comparable` and `Comparator` interfaces must be consistent with equals

The "natural" ordering for a class C (implementing `Comparable`) is said to be *consistent with equals* if and only if `e1.compareTo(e2) == 0` has the same boolean value as `e1.equals(e2)` for every `e1` and `e2` of class C.

Note that `null` is not an instance of any class, and `e.compareTo(null)` should throw a `NullPointerException` even though `e.equals(null)` returns `false`.



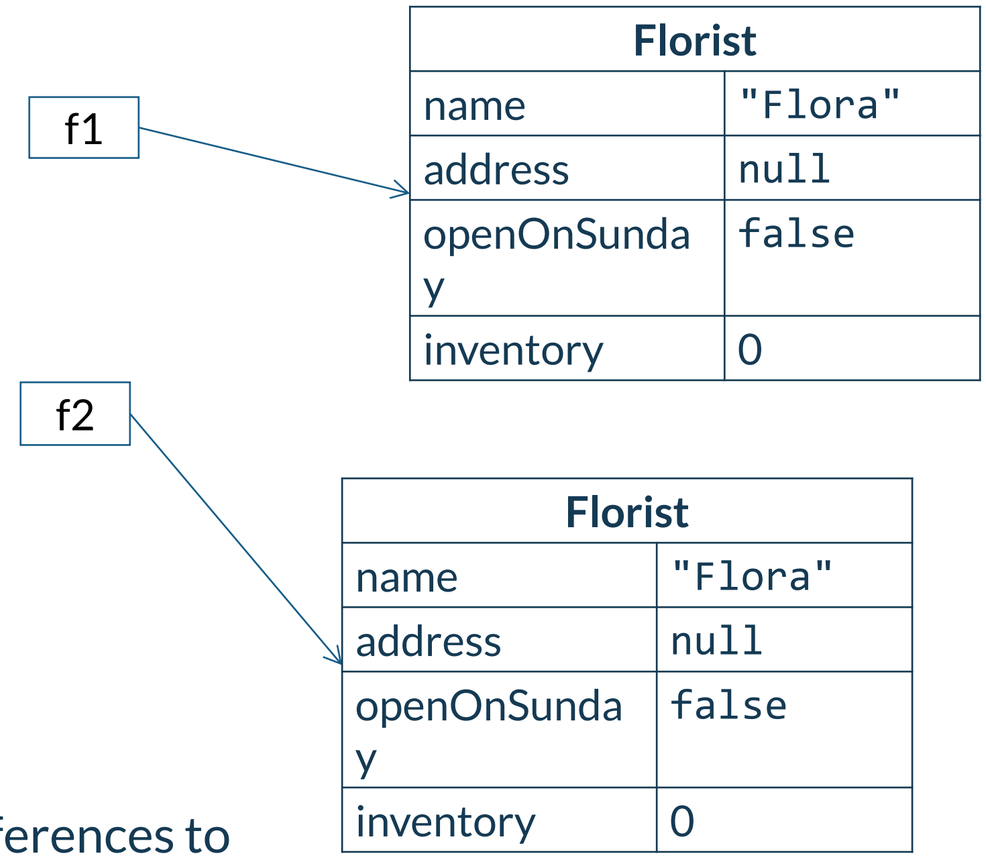
© 2019 ESTECO SpA

Referential equality 1/2

ReferentialInequality.java

```
void main() {
    Florist f1 = new Florist("Flora");
    Florist f2 = new Florist("Flora");

    if (f1 == f2) {
        IO.println("Same");
    } else {
        IO.println("Different");
    }
}
```




```
$ java ReferentialInequality
Different
```

f1 and f2 are references to different objects

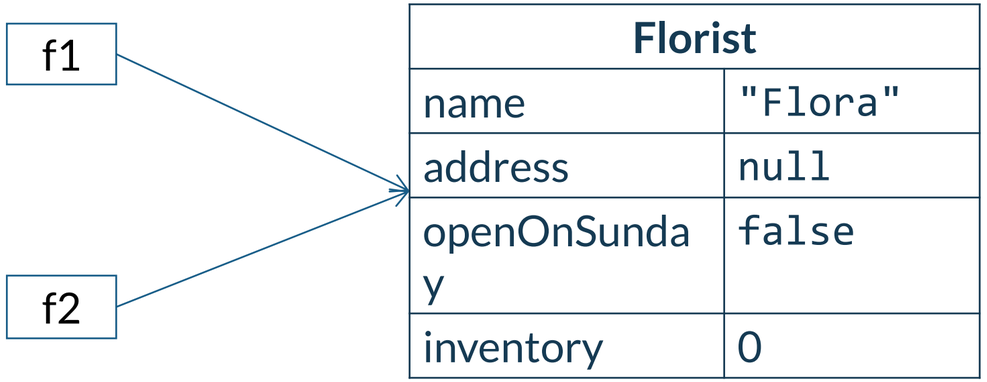
© 2019 ESTECO SpA

Referential equality 2/2

ReferentialEquality.java

```
void main() {
    Florist f1 = new Florist("Flora");
    Florist f2 = f1;

    if (f1 == f2) {
        IO.println("Same");
    } else {
        IO.println("Different");
    }
}
```




```
$ java ReferentialEquality
Same
```

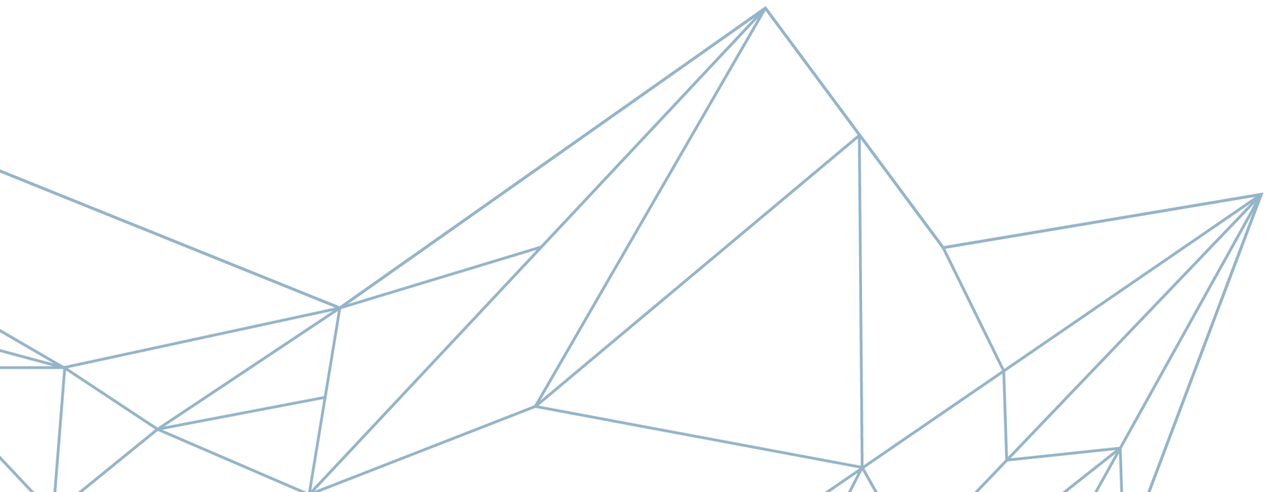
f1 and f2 are references to the same object

© 2019 ESTECO SpA





More on object equivalence



© 2019 ESTECO SpA

Object.hashCode()

```
public int hashCode()
```

Returns a hash code value for the object. This method is supported for the benefit of hash tables such as those provided by HashMap

The general contract of hashCode is:

- Whenever it is invoked on the same object more than once during an execution of a Java application, the hashCode method must consistently return the same integer, provided no information used in equals comparisons on the object is modified. This integer need not remain consistent from one execution of an application to another execution of the same application
- If two objects are equal according to the equals method, then calling the hashCode method on each of the two objects must produce the same integer result
- It is *not* required that if two objects are unequal according to the equals method, then calling the hashCode method on each of the two objects must produce distinct integer results. However, the programmer should be aware that producing distinct integer results for unequal objects may improve the performance of hash tables



© 2019 ESTECO SpA

HashCode and equals 1/4

Suppose that two `Florist` objects with the same name represent the same florist

And we want to store the number of orders of each florist into a `Map`, so that we can retrieve that information from the map

```
public class Florist {

    private String name;

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

```
void main() {
    var orders = new HashMap<Florist, Integer>();
    orders.put(new Florist("Flora"), 12);
    orders.put(new Florist("Iris"), 7);

    Florist florist = new Florist("Flora");
    IO.println(orders.get(florist));
}
```

This example doesn't work as expected

Why?



© 2019 ESTECO SpA

HashCode and equals 2/4

```
public class Florist {
    private String name;

    public Florist(String name) {
        this.name = name;
    }

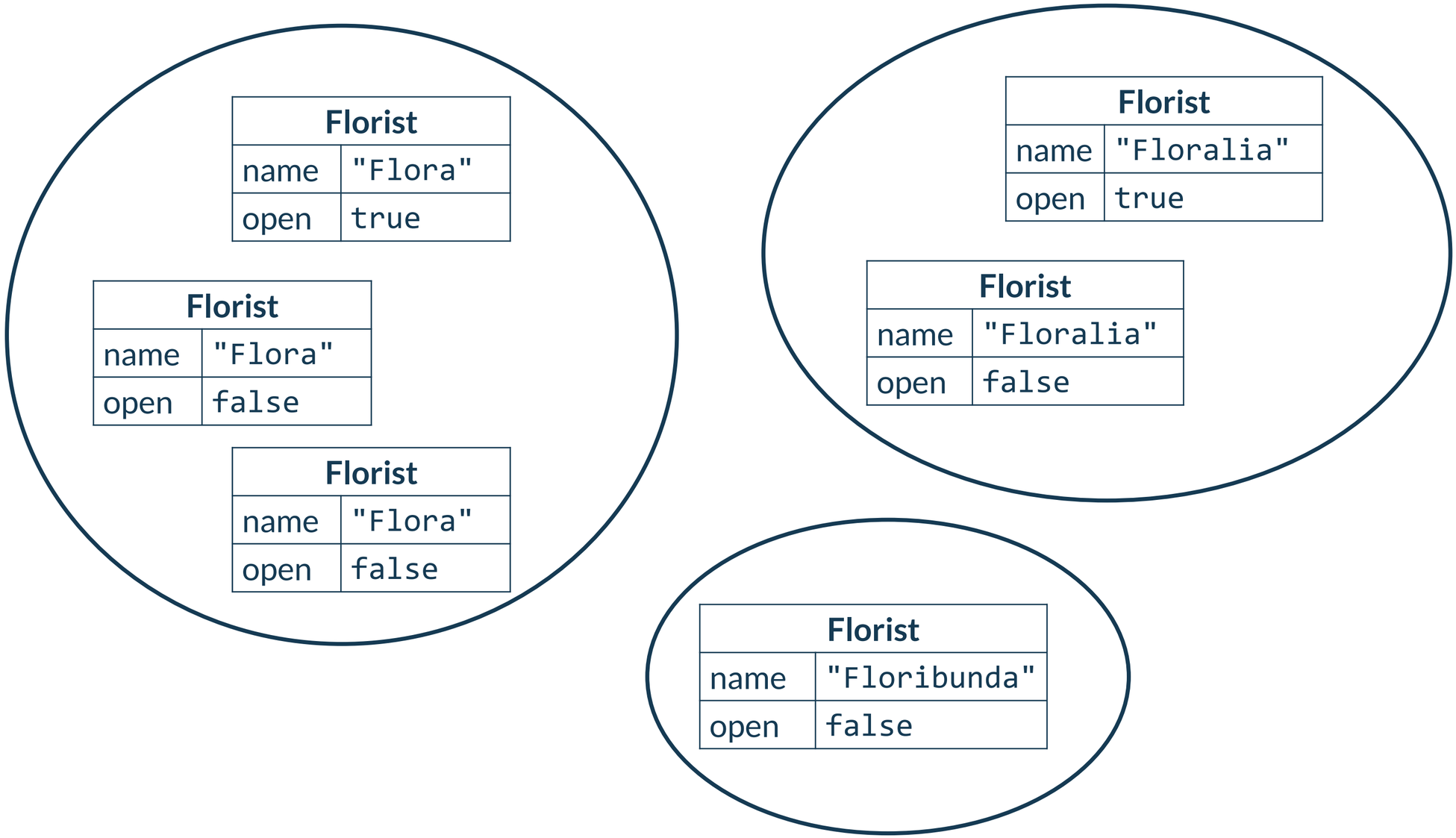
    @Override
    public boolean equals(Object o) {
        if (this == o) return true;
        if (o == null || getClass() != o.getClass()) return false;
        Florist florist = (Florist) o;
        return Objects.equals(name, florist.name);
    }

    @Override
    public int hashCode() {
        return name.hashCode();
    }
}
```



© 2019 ESTECO SpA

Florist equivalence classes



© 2019 ESTECO SpA

HashCode and equals 3/4

Now we add a rebrand(String) method because we want to rename a `Florist`

```
public class Florist {

    private String name;

    public Florist(String name) {
        this.name = name;
    }

    public void rebrand(String name) {
        this.name = name;
    }

    public String getName() {
        return name;
    }

    …
}
```



© 2019 ESTECO SpA

HashCode and equals 4/4

```
void main() {
    var orders = new HashMap<Florist, Integer>();
    orders.put(new Florist("Flora"), 12);
    orders.put(new Florist("Iris"), 7);

    for (Florist florist : orders.keySet()) {
        if (florist.getName().equals("Flora")) {
            florist.rebrand("Flora's Flowers");
            break;
        }
    }

    Florist florist = new Florist("Flora's Flowers");
    IO.println(orders.get(florist));
}
```

After rebranding, the hash code is changed but the object position in the hash table is not updated!

The API of the Florist class is flawed: equals() and hashCode() depend on a mutable field, so we cannot use it as a key in hash-based collections

A `Florist` is an entity: you should base its equality on something that never changes (e.g., a VAT number), not on its name



© 2019 ESTECO SpA

More on equals() and hashCode()

`equals()` and `hashCode()` are defined in the `Object` class, but in practice they are used only by the collections

- `equals()` is used by the collections to test the presence of an object
- `hashCode()` is used by the hash-based collections only

The `Object` class provides a consistent (default) implementation for both the methods

Many Java classes overrides `equals()` and `hashCode()`, for very good reasons

Every time you override `equals()` and `hashCode()` you are changing the semantic of the equivalence for that class, e.g., moving from identity-based to value-based equivalence

For example, records and enums provide built-in consistent implementations of `equals()` and `hashCode()`, that are identity-based for enums and value-based for records



© 2019 ESTECO SpA

Digression – Equivalence of Lists

`boolean equals(Object o)`
Compares the specified object with this list for equality. Returns `true` if and only if the specified object is also a list, both lists have the same size, and all corresponding pairs of elements in the two lists are *equal*. (Two elements `e1` and `e2` are *equal* if `Objects.equals(e1, e2)`.) In other words, two lists are defined to be equal if they contain the same elements in the same order.
This definition ensures that the equals method works properly across different implementations of the `List` interface.

```
static void main() {
    List al = new ArrayList();
    al.add("SDM");

    List ll = new LinkedList();
    ll.add("SDM");

    IO.println(al.equals(ll)); // true
}
```

al of class `ArrayList` is equivalent to ll of class `LinkedList`

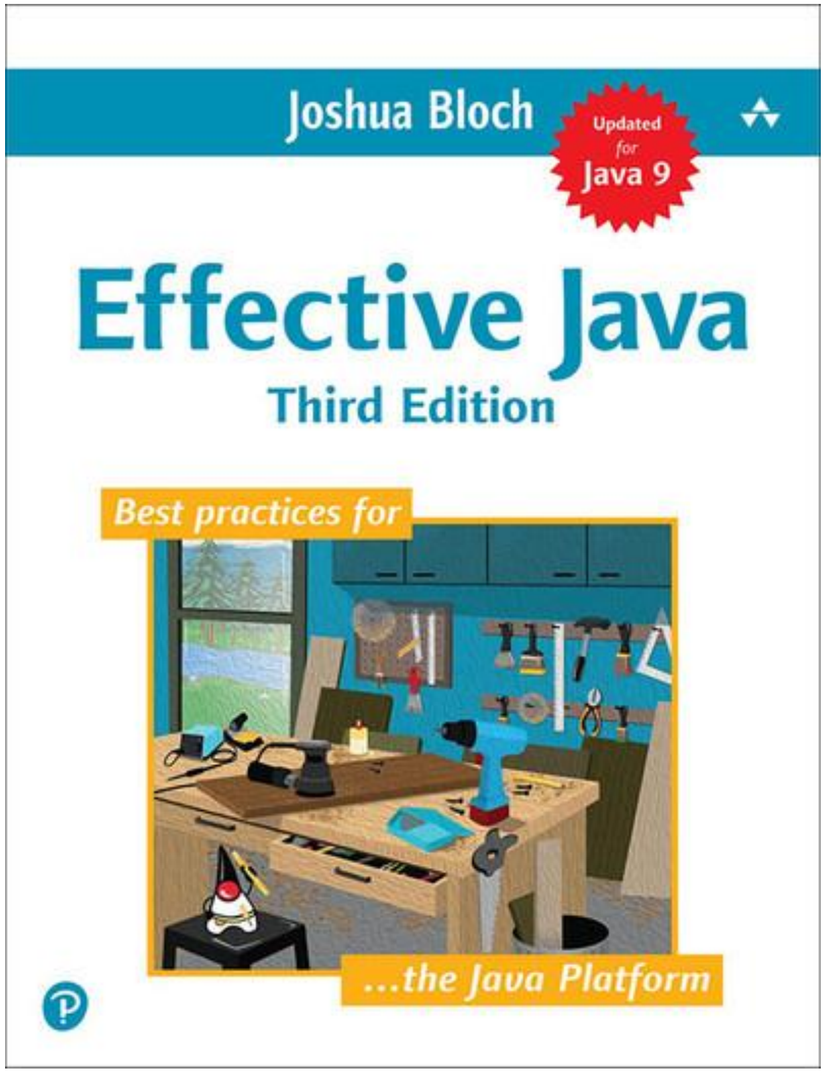
The `List` “family” defines the equivalence of objects belonging to different classes

`ArrayList` and `LinkedList` are not even in the same hierarchy, there is no hierarchy between them, `ArrayList` is not a subclass of `LinkedList` nor vice versa



© 2019 ESTECO SpA

Reference on overwriting equals() and hashCode()



Bloch, Joshua J. **Effective Java**, 3rd Edition. Addison-Wesley Professional, 2018
Chapter 3 “Methods Common to All Objects”

© 2019 ESTECO SpA

Values & immutability

© 2019 ESTECO SpA

Identity, state and time

Identity is the relationship between state and time

9:00

Florist	
name	`"Flora"`
address	`null`
openOnSunday	`false`
inventory	`0`

11:00

Florist	
name	`"Flora"`
address	`null`
openOnSunday	`false`
inventory	`40`

18:00

Florist	
name	`"Flora"`
address	`null`
openOnSunday	`true`
inventory	`40`



The state of the object changes over time, yet it is always the same florist

Java tracks identity with references: `==` compares identities, not states



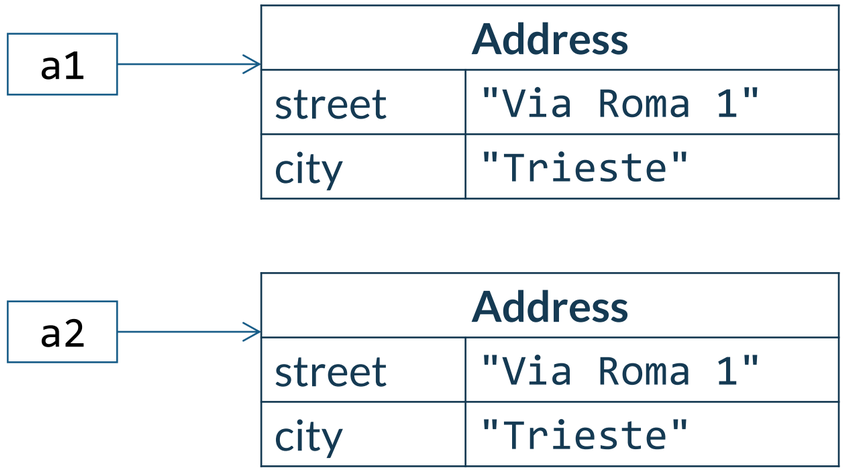
© 2019 ESTECO SpA

Values: state without time

Addresses.java

```
record Address(String street, String city) {}

void main() {
    var a1 = new Address("Via Roma 1", "Trieste");
    var a2 = new Address("Via Roma 1", "Trieste");
    IO.println(a1 == a2);
    IO.println(a1.equals(a2));
}
```




```
$ java Addresses
false
true
```

Records are only shallowly immutable: a component of type `List` can still change

- An `Address` never changes: without time, its identity carries no information
- Equal values are interchangeable: compare them with `equals()`
- To “change” a value, create a new one: `flora.moveTo(new Address(...))`

© 2019 ESTECO SpA

Entities vs values

	`Florist` (entity)	`Address` (value)
State over time	mutable	immutable
Does identity matter?	yes	no
`equals()`	identity-based (Object's default)	value-based
Safe as a HashMap key?	yes	yes

Remember the rebranded `Florist`: a mutable class with value-based `equals()` and `hashCode()` is the dangerous combination

Questions

- Is `address` part of Flora's identity?
- Two `Florist` objects named Flora: are they the same florist?



© 2019 ESTECO SpA

Additional content – Value classes in Java

Project Valhalla is making values explicit: JEP 401, a preview feature targeting JDK 28 (March 2027)

```
value record Address(String street, String city) {}
```

- Value objects have only `final` fields and no identity
- `==` on value objects compares their field values, not their references
- `equals()` remains the way to compare objects: `==` does not replace it
- Some JDK classes, such as `Integer` and `LocalDate`, become value classes
- Preview features are disabled by default: compile and run with `--enable-preview`

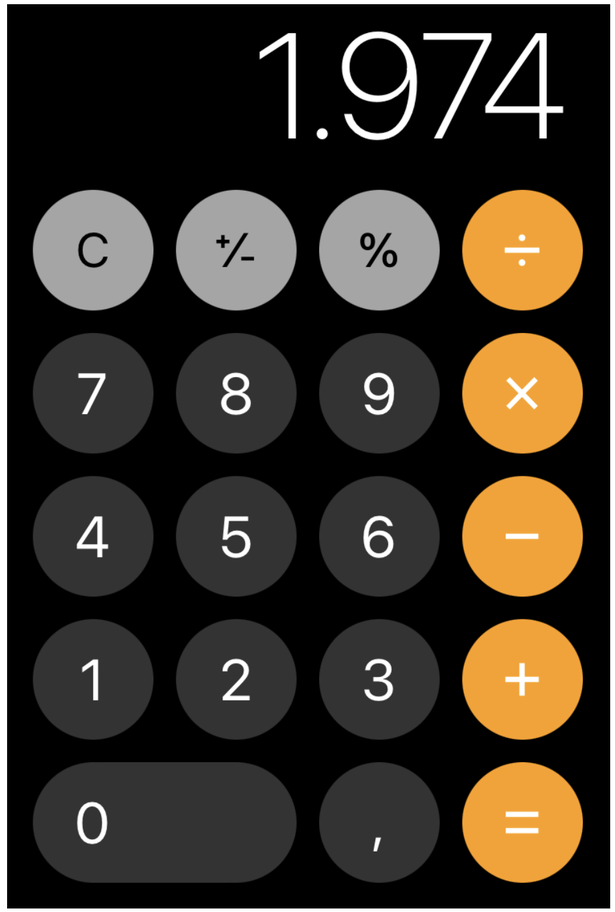


© 2019 ESTECO SpA

Exercises

© 2019 ESTECO SpA

Exercise 1



Define a calculator class that

1. receives “events” from a calculator keyboard
2. sends the output to a Display object

```
interface Display {
    void display(String text);
}
```

```
class Calculator {

    final Display display;
    //...

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

    void plusPressed() {
        //...
    }

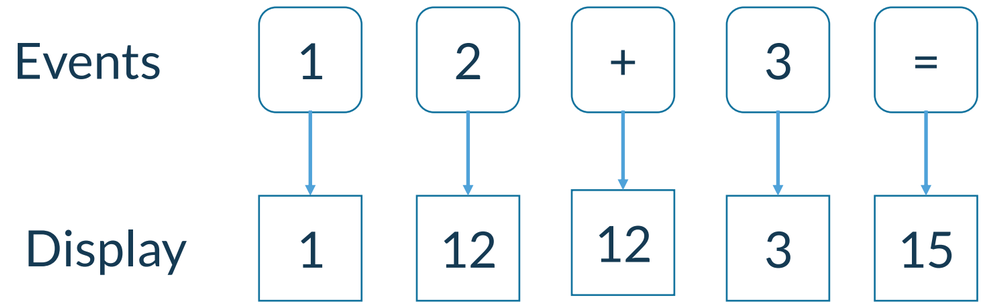
    void zeroPressed() {
        //...
    }

    //...
}
```

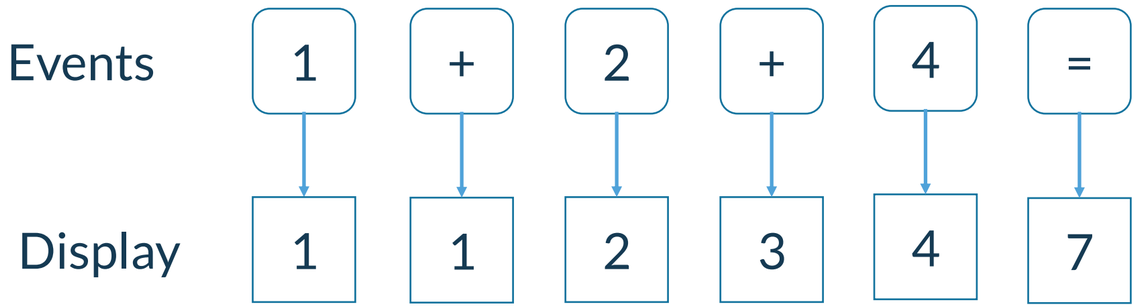
© 2019 ESTECO SpA

A couple of scenarios

Sample scenario



Operations chaining



© 2019 ESTECO SpA

Exercise

Write a class (or a set of classes) that given a text it produces a Term Frequency table.
Consider the option to provide a list of stop words, normalization, etc.
Provide an option to print the table in alphabetical order and by frequency.

"Term frequency (TF) means how often a term occurs in a document. In the context of natural language, terms correspond to words or phrases ..."



Term	Frequency
english	8
language	7
words	12

...

input	7
cactus	1
fireworks	3

© 2019 ESTECO SpA

Self assessment

© 2019 ESTECO SpA

Quiz 1: Object-oriented programming

1. **What is the central idea of the Object-Oriented Programming (OOP) paradigm?**
   A. A program is a sequence of reusable procedures that perform specific tasks
   B. A program is a combination of function calls that use immutable data
   C. A program is a community of interacting objects, where each object has both a state and a defined behavior
   D. A program is a set of logical rules that a computer follows to reach a conclusion

2. **What is a key difference between procedural programming and OOP?**
   A. Procedural programming uses late binding (dynamic dispatch), while OOP uses early binding (static dispatch)
   B. In OOP, the specific action taken by a method call is determined at runtime based on the object, a concept known as late binding
   C. Procedural programming does not allow for any form of code reuse
   D. In OOP, methods operate on data passed to them as arguments, not on the object's own members



© 2019 ESTECO SpA

Quiz 1: Object-oriented programming

3. **Which is the OOP concept that separates an object's interface from its internal implementation?**
   A. Polymorphism
   B. Abstraction
   C. Inheritance
   D. Encapsulation

4. **The primary purpose of inheritance is for what?**
   A. Classifying objects into a strict taxonomy
   B. Hiding the internal secrets of an object
   C. Allowing objects of different classes to be treated as objects of a common superclass
   D. Reusing code from an existing parent (class)



© 2019 ESTECO SpA

Quiz 2: Reflection and best practices

1. **A well-designed class should:**
   A. Have one clear responsibility and hide internal data
   B. Expose as many public fields as possible for flexibility
   C. Avoid constructors to reduce complexity
   D. Depend heavily on global variables

2. **Which statement about final is true?**
   A. A final class can still be subclassed
   B. A final method can be overridden in a subclass
   C. final prevents further inheritance or overriding
   D. final means the class has no constructors

3. **Which correctly describes polymorphism in Java?**
   A. Multiple methods with the same name but different signatures
   B. Treating objects of different subclasses through a common interface or superclass reference
   C. Automatic type casting of primitives
   D. Using multiple inheritance of classes.



© 2019 ESTECO SpA

Quiz 2: Reflection and best practices

4. **What happens if a subclass overrides a non-final method and forgets to call super.method() where needed?**
   A. The superclass behavior is skipped; possible logic bug
   B. Compilation error
   C. The superclass method runs automatically
   D. The JVM inserts a warning only
5. **Which statement about interfaces is correct?**
   A. Interfaces can hold instance fields
   B. Interfaces can have constructors
   C. All interface fields are public static final by default
   D. Classes can extend multiple interfaces



© 2019 ESTECO SpA

Quiz 3: Object equivalence

1. **In the context of the `Object` class, what is the default behavior of the `equals()` method?**
   A. It returns true if all instance fields have the same value
   B. It returns true if the objects are of the same class
   C. It returns true only if both references point to the same object (reference equality)
   D. It throws a `NullPointerException`

2. **To define a natural order for objects of a specific class in Java, which interface must that class implement?**
   A. `Comparator`
   B. `Serializable`
   C. `Iterator`
   D. `Comparable`



© 2019 ESTECO SpA

Quiz 3: Object equivalence

3. **What is the fundamental contract between the `equals()` and `hashCode()` methods?**
   A. If two objects are equal according to the `equals()` method, then calling `hashCode()` on each must produce the same integer result
   B. If two objects are unequal by `equals()`, their `hashCode()` methods must produce different results
   C. The `hashCode()` value must remain consistent across different executions of the application
   D. The `hashCode()` is only used by `TreeSet` and `TreeMap`

4. **What potential issue can arise if an object used as a key in a hash-based collection (like `HashMap`) is modified in a way that changes the information used in its `hashCode()` calculation after it has been put into the collection?**
   A. A `ConcurrentModificationException` will be thrown
   B. The collection will automatically update the object's position
   C. The `equals()` method will no longer work correctly
   D. The object may become irretrievable (lost) from the collection



© 2019 ESTECO SpA

Correct answers

Quiz 1

1. C
2. B
3. B
4. D

Quiz 2

1. A
2. C
3. B
4. A
5. C

Quiz 3

1. C
2. D
3. A
4. D

© 2019 ESTECO SpA



Thank you!

esteco.com

Read the ESTECO Copyright Policy © 2019 ESTECO SpA