



Programming in Java – Inheritance



Paolo Vercesi
Technical Program Manager

Agenda



Extension and inheritance

Access control and encapsulation

Interfaces and polymorphism

The Three Musketeers of OOP

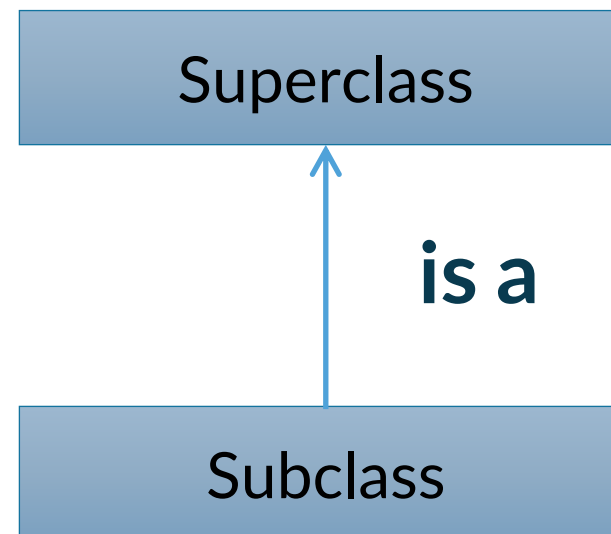


Extension and inheritance



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

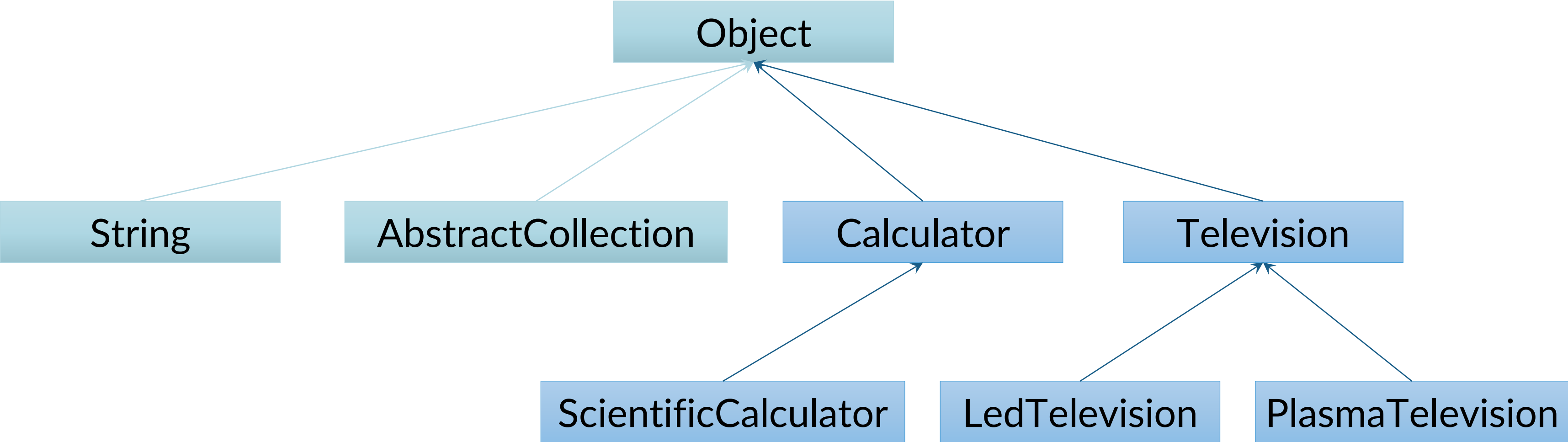
```
package it.units.sdm;  
public class PlasmaTelevision extends Television {  
    double usageHours;  
}
```

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

Java uses the **extends** keyword to indicate the **extended** superclass



Java classes hierarchy



Construction of subclasses

```
package it.units.sdm;

public class PlasmaTelevision extends Television {

    double usageHours;

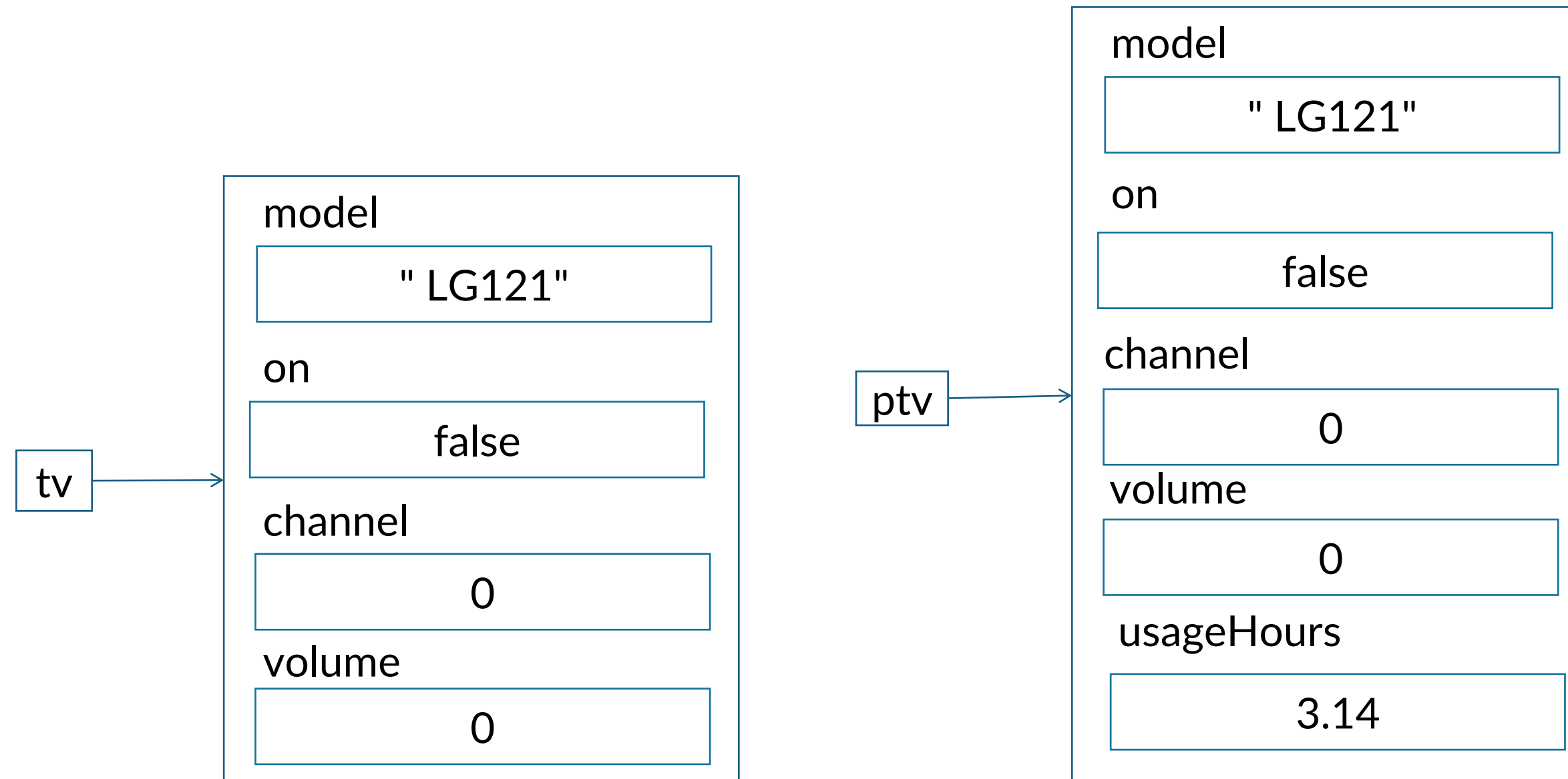
    public PlasmaTelevision(String model, double usageHours) {
        super(model);
        this.usageHours = usageHours;
    }
}
```

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");
```



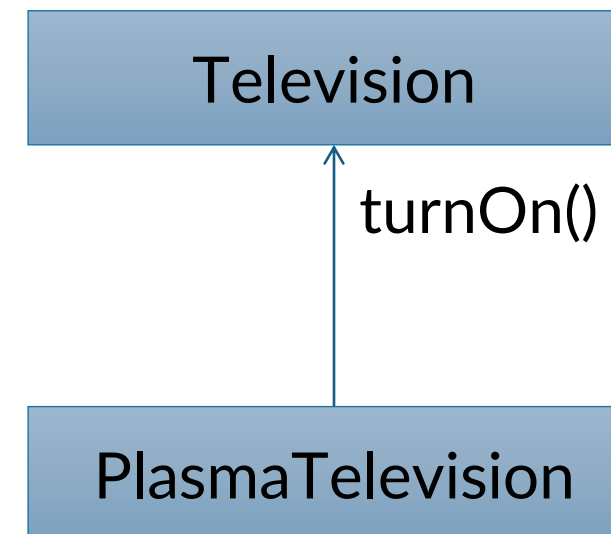
Inheritance with methods

- ✓ **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



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;

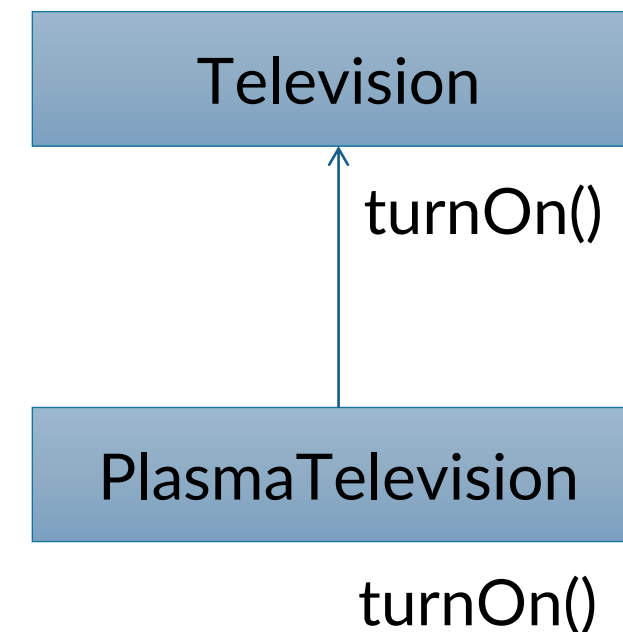
public class PlasmaTelevision extends Television {

    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

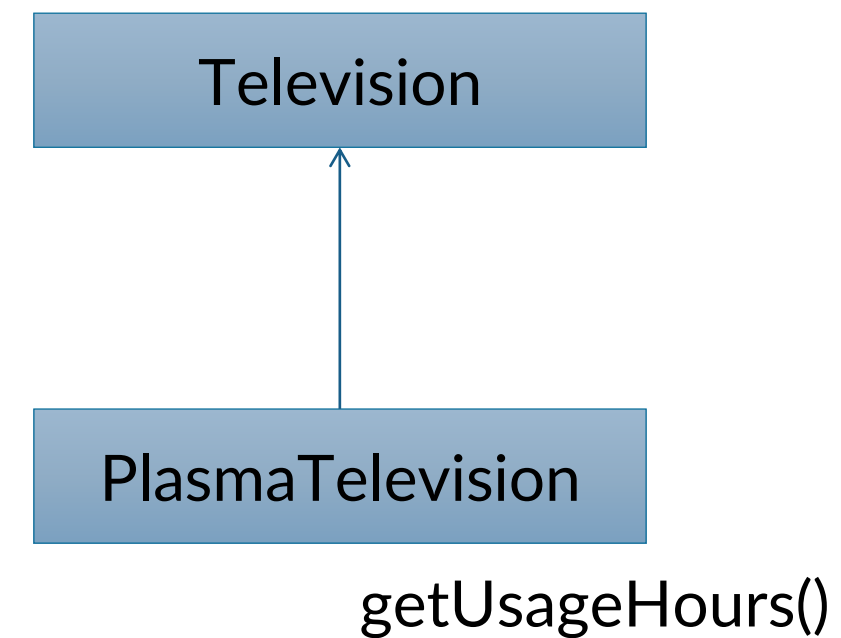
```
public class PlasmaTelevision extends Television {  
  
    private double usageHours;  
    private long startTime;  
  
    public PlasmaTelevision(String model) {  
        super(model);  
    }  
  
    void turnOn() {  
        super.turnOn();  
        startTime = System.currentTimeMillis();  
    }  
  
    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
```



Late binding

com.esteco.sdm.Television

```
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.PlasmaTelevision

```
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();`



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





Access control and encapsulation



Access control

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
<i>no modifier</i>	Y	Y	N	N
private	Y	N	N	N



Who can access this class? 1/3

```
com.esteco.sdm.Televsion
```

```
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.

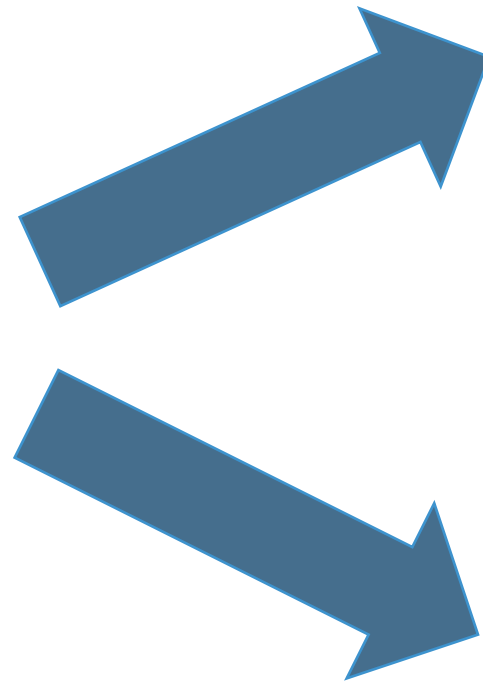
```
com.esteco.sdm.Television
```

```
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** does not allow sub-classing

A **final method** cannot be overridden in a subclass

An **abstract class** cannot be instantiated

An **abstract method** has no body, and 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 syntactically like 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);  
}
```

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

```
class ConsoleDisplay implements Display {  
    @Override  
    public void display(double d) {  
        System.out.println(format(d));  
    }  
  
    public String format(double d) {  
        return String.valueOf(d);  
    }  
}
```



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();  
}
```

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

```
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  
    }  
}
```

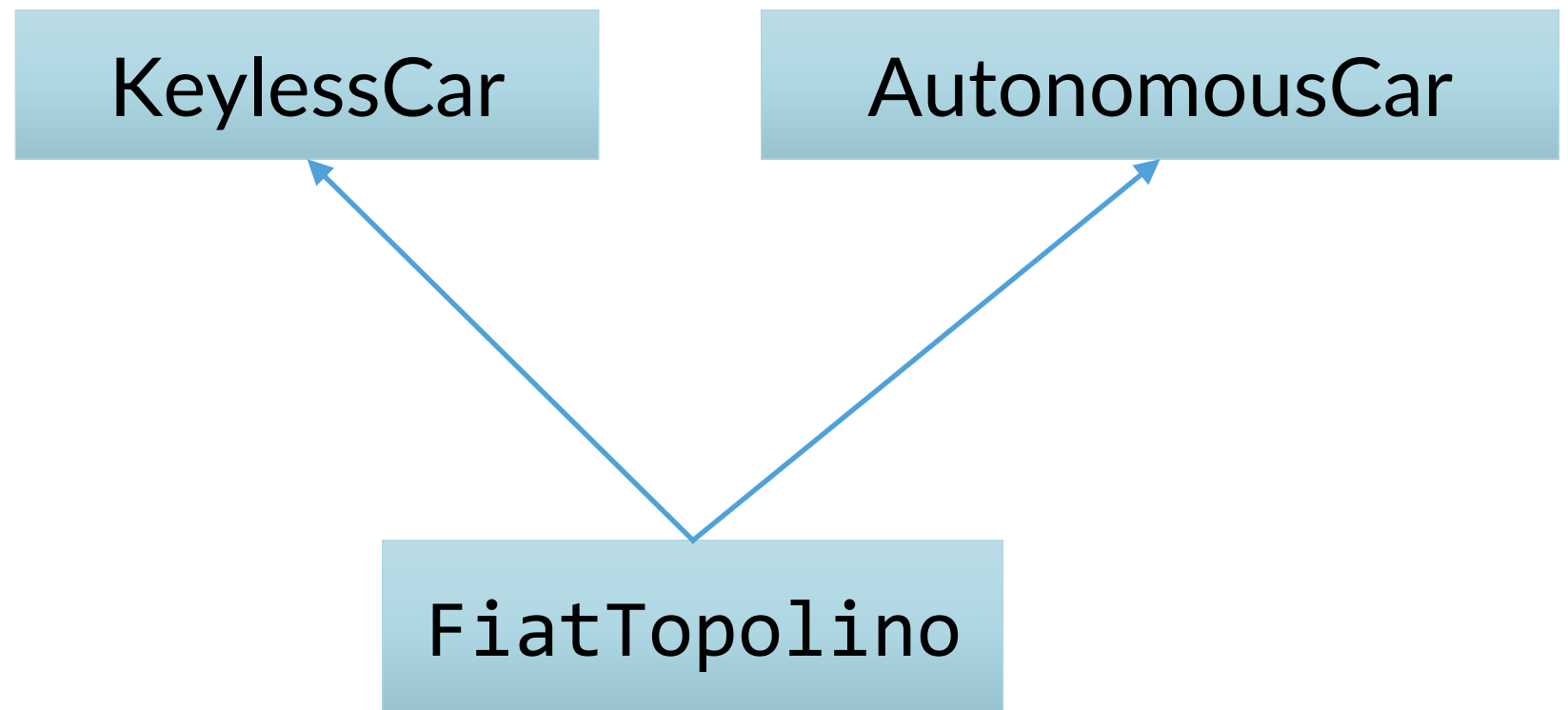


Implementation of multiple interfaces 3/4

```
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) c;  
Display d2 = (Display) kc;
```

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

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.

```
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`.



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();  
            }  
        };  
        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?
`a.getA()`

Anonymous classes “don’t define” a new type



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

Sample usage of DEFAULT_ADDRESS

```
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!  
    }  
}
```



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();  
            }  
        };  
    }  
}
```



Default methods 1/3

```
interface CollectionOfStrings {  
    default boolean isEmpty() {  
        return getSize() == 0;  
    }  
  
    int getSize();  
  
    String[] getValues();  
}
```

Java 8 introduced the default methods in interfaces.

Default methods can be overridden by classes implementing the interface.



Default methods 2/3

```
interface CollectionOfStrings {  
  
    default boolean isEmpty() {  
        return getSize() == 0;  
    }  
  
    int getSize();  
  
    String[] getValues();  
}
```



```
interface CollectionOfStrings {  
  
    default boolean isEmpty() {  
        return getSize() == 0;  
    }  
  
    int getSize();  
  
    String[] getValues();  
  
    boolean contains(String string);  
}
```

In a new version of our library, we want to add a **boolean contains(String string)** method to the interface. What does happen to all the implementing classes?



Default methods 2/3

```
interface CollectionOfStrings {  
    default boolean isEmpty() {  
        return getSize() == 0;  
    }  
  
    int getSize();  
  
    String[] getValues();  
  
    default boolean contains(String string) {  
        for (String s: getValues()) {  
            if (Objects.equals(s, string))  
                return true;  
        }  
        return false;  
    }  
}
```

When we add the method `boolean contains(String)` to the `CollectionOfStrings` interface, most of the classes implementing `CollectionOfStrings` become unusable, because their source code don't compile anymore.

By defining a default method, we preserve the compatibility with older code bases.



Why Java introduced default methods

A primary motivation for the default method was to provide a means by which **interfaces can be expanded without breaking existing code.**

Another motivation for the default method was the desire to **specify methods in an interface that are, essentially, optional,** depending on how the interface is used.

It is important to point out that the addition of default methods does not change a key aspect of an interface: its **inability to maintain state information.**



Default methods clash

```
interface AutonomousCar {  
    void driveTo(String address);  
    default void stop() {  
        //do something  
    }  
}  
  
interface KeylessCar {  
    void start();  
    default void stop() {  
        //do something  
    }  
}
```

```
class FiatTopolino implements KeylessCar, AutonomousCar {  
    @Override  
    public void driveTo(String address) {  
        //drive to address  
    }  
  
    @Override  
    public void start() {  
        //Do something  
    }  
  
    @Override  
    public void stop() {  
        //FiatTopolino must override this method  
    }  
}
```

When a class implements two interfaces that both have the **same default method**, the class **must override** that method, otherwise a compilation error will result.



Default methods overriding

```
interface KeylessCar {  
    void start();  
  
    default void stop() {  
        //do something  
    }  
}  
  
interface AutonomousCar extends KeylessCar {  
  
    void driveTo(String address);  
  
    default void stop() {  
        KeylessCar.super.stop();  
        //do something  
    }  
}
```

When one interface extends another, with both defining a common default method, the extending interface version of the method takes precedence.

It is possible to explicitly refer to a default implementation in an extended interface by using **super**.



Private interface methods

```
interface Interface {  
  
    default void defaultMethodA() {  
        //do something  
        privateMethod("A");  
    }  
  
    default void defaultMethodB() {  
        //do something  
        privateMethod("B");  
    }  
  
    private void privateMethod(String message) {  
        System.out.println(message);  
    }  
  
}
```

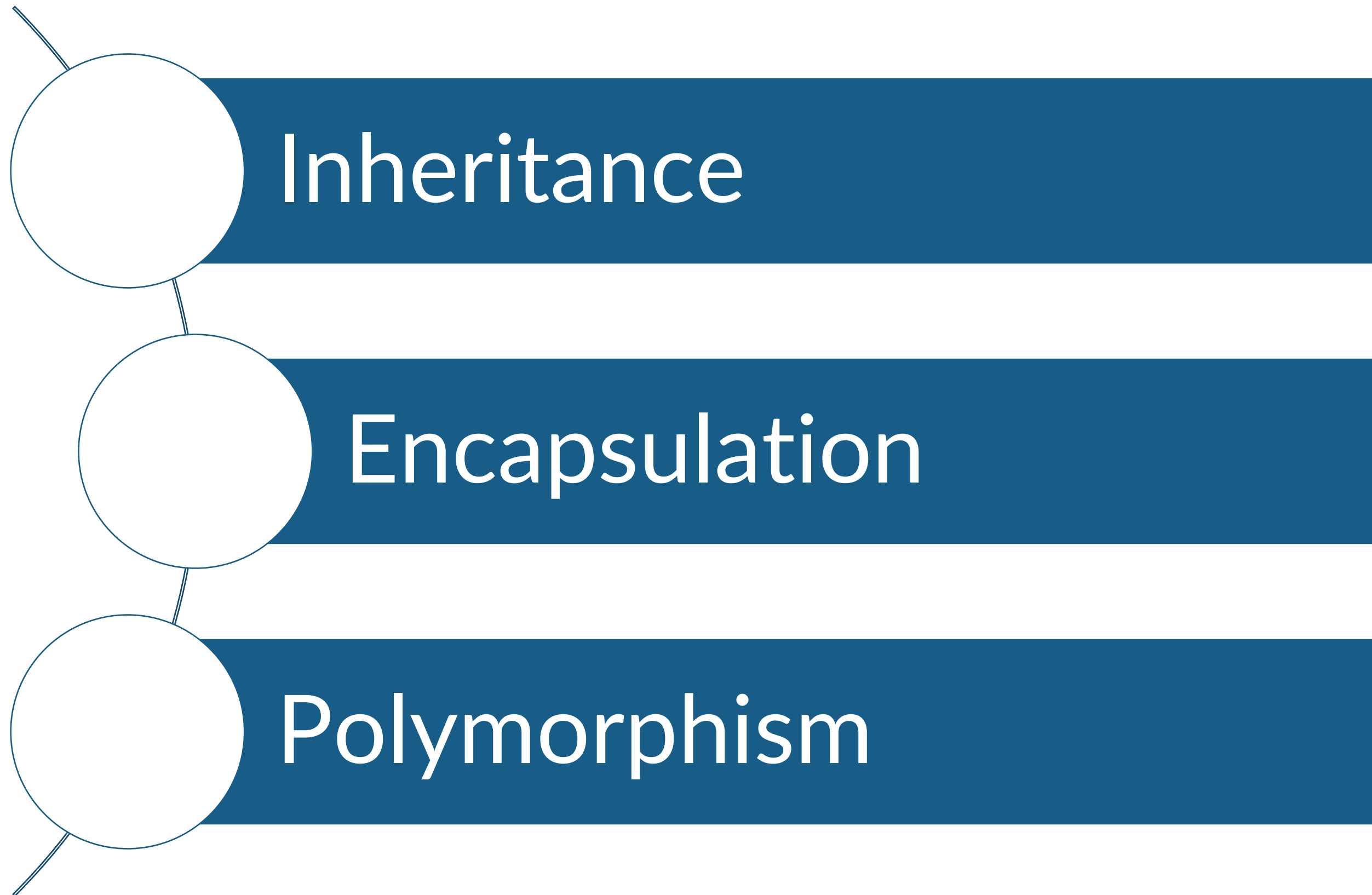
Private methods can be used only from the default methods of the same interface





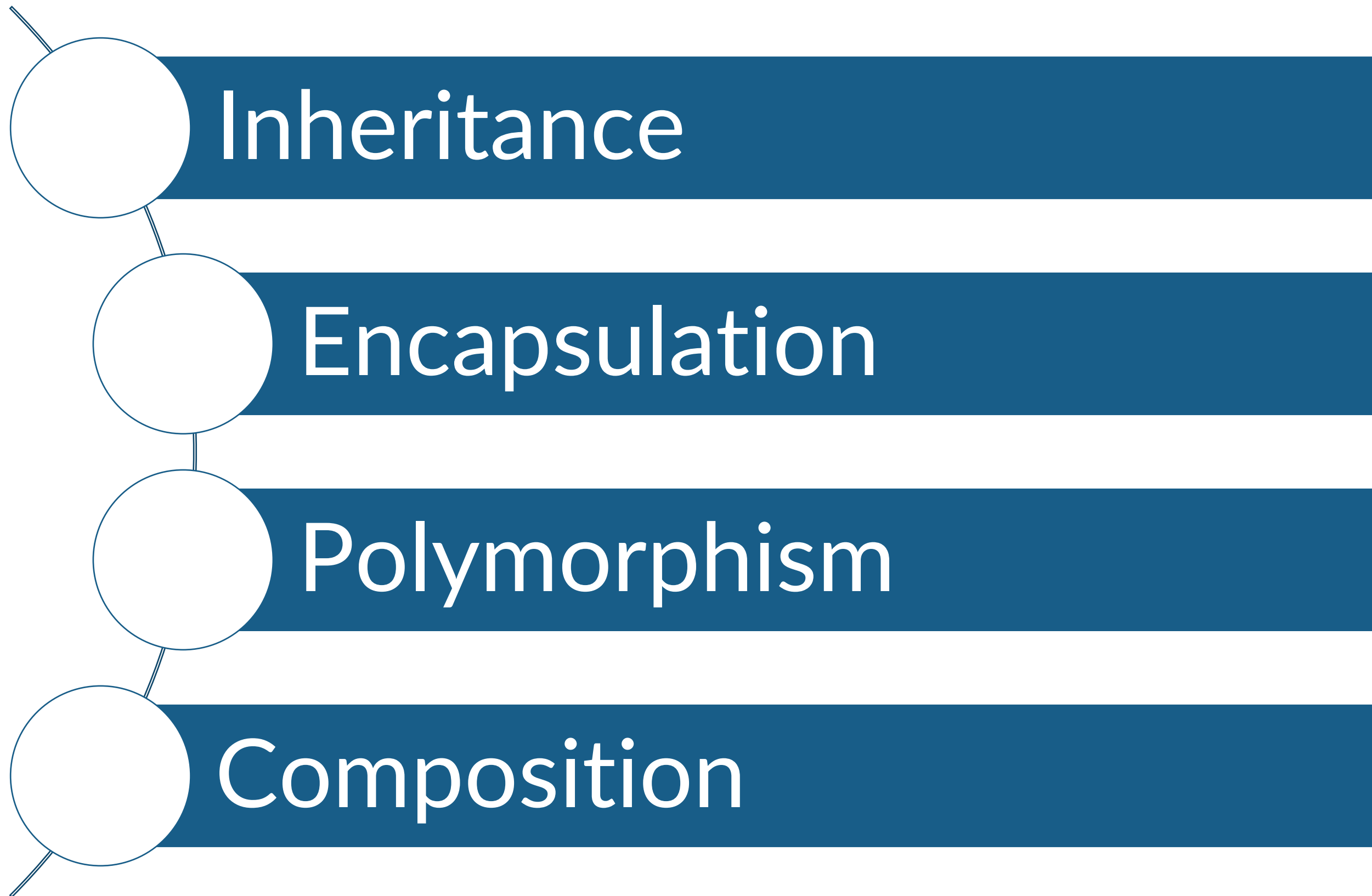
The Three Musketeers of OOP





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;  
}
```



When to use inheritance

Both classes are in the same logical domain

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

The subclass is a proper subtype of the superclass

The enhancements made by the subclass are primarily additive

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





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 . Minimize code duplication.

Hint: consider the usage of both inheritance and composition.



Tomorrow

- We will take a breath (not a break)
- We will review the assignments





Thank you!

esteco.com

