Systems Programming

Object Oriented Programming

Julio Villena Román (Lecturer)
<jvillena@it.uc3m.es>

Contents are mostly based on the work by:

M.Carmen Fernández Panadero, Raquel M. Crespo García
Carlos Delgado Kloos and Natividad Martínez Madrid

Systems Programming

Object BASED programming

Julio Villena Román (Lecturer)

<jvillena@it.uc3m.es>

Contents are mostly based on the work by:

M.Carmen Fernández Panadero, Raquel M. Crespo García
Carlos Delgado Kloos and Natividad Martínez Madrid

Scenario IV:
Declare and implement a class

- Now that you know how to read code and implement your own methods you will have to design a new class in order to create a new data type with its characteristics and behavior.

- Objective:
  - Be able to **declare a class** with a set of characteristics (**attributes**) and behaviour (**methods**)
  - Be able to **create objects** and modify or restrict access to their state and their behavior
- Workplan:
  - Memorize the basic **nomenclature** of the object-oriented programming
  - Practice **modeling objects** with simple examples to distinguish between a class, an object, its state and behavior
  - Review the **Java syntax** for declaring **class attributes, constructors** and **methods**
  - Review the mechanism and syntax for **message passing** between objects

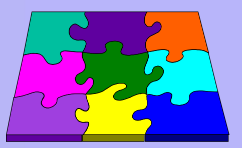
Contents

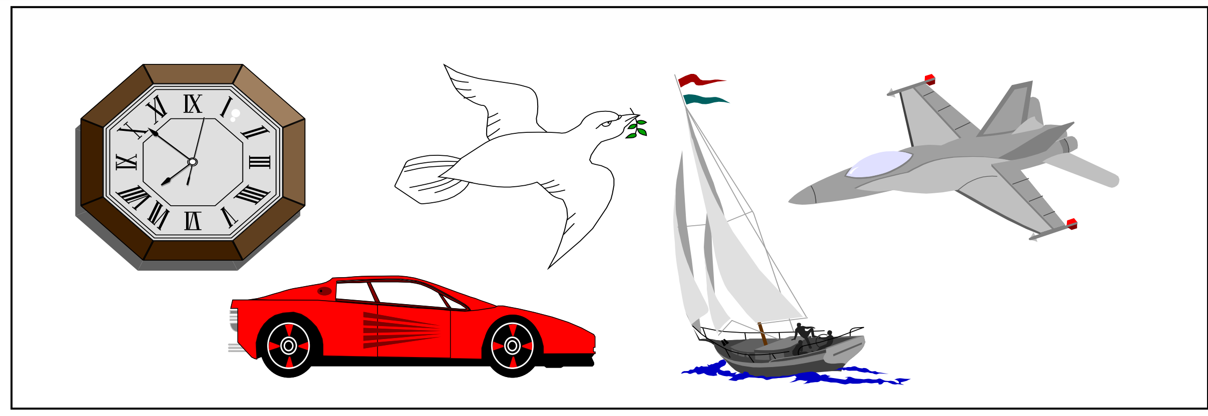
- Classes and Objects
- Object encapsulation
  - Functional abstraction
  - Data abstraction
- Class members (attributes and methods)
- Message passing
- Constructors
- Overloading

Objectives

- Define ***basic concepts*** of object ***based*** programming
  - Classes, objects
  - Members (variables, methods)
  - Abstraction and shadowing of information
- Learn the ***relationship*** between object and class
- ***Create*** a simple object and be able to ***model:***
  - its attributes (with variables)
  - its behaviour (with methods)

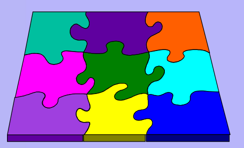
What is an object?

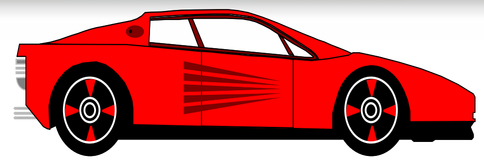




- **Objects** are (simple/complex) (real/imagined) representations of things: clock, airplane, bird, etc.

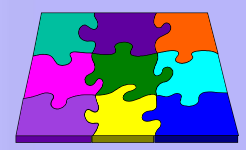
What is an object?





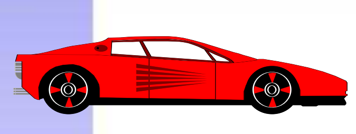
- ***Functional abstraction***
  - Things that we know that cars do:
    - advance
    - stop
    - turn right
    - turn left
- ***Data abstraction***
  - Properties (attributes) of a car:
    - color
    - speed
    - size

What is an object?



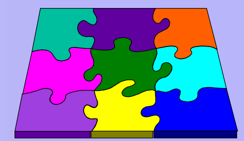
- It is a way to group a set of data (**state**) and functionality (**behaviour**) in the same block of code that can then be referenced from other parts of a program

Classification of objects



- **Class**: Set of objects with similar states and behavior
  - We can refer to the "Car" class (any instance in the classification of cars)
- "My car" is an **object**, i.e. a particular **instance** of the Car class
- The **class** to which the object belongs is considered as a new **data type**

Objects vs. Classes



A **class** is an abstract entity

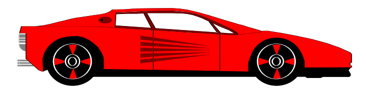
- It is a kind of data classification
- Defines the behaviour and attributes of a group of objects with similar structure and similar behaviour

Class car	→ Class name
Methods: turn on, advance, stop, ...	→ Methods (functions)
Atributtes: color , speed, etc.	→ Attributes (data)

An **object** is an instance of a class

⌘ An object can be distinguished from other members of the class by the value of its attributes

Object MyFerrari
Belongs to the class Car



Name: MyFerrari
Methods: turn on, advance, stop, ...
Attributes : color = "red";
speed = 300Km/h

- A **class** is declared, an **object** is created

Object encapsulation

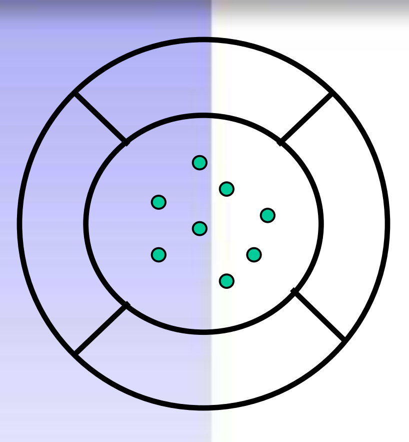
- ***Encapsulation:*** explains the links between **behaviour** and **state** to a particular object
- ***Information hiding***: Define which parts of the object are visible (the public interface) and which parts are hidden (private)

- *The ignition is a **public interface** mechanism to start a vehicle*
- *The implementation of how to really start a car is **private.** We only can access this information introducing the key into the ignition*

Pros

The object may change but its public interface remains compatible with the original. This fact facilitates reuse of code

Object encapsulation



CLASS MEMBERS

Objects encapsulate attributes, allowing access to them only through methods

- **Attributes (Variables)**: Containers of values
- **Methods**: Containers of functions

An object has:

- **State**: represented by the values of its attributes
- **Behaviour**: defined by its methods

Usually:

- Methods are public
- Attributes are private
- There can be private methods
- It is dangerous to have public attributes

Object Definition

Public Members

- Public members (describe **what** an object can do)
  - What the object can do (methods)
  - What the object is (its abstraction)

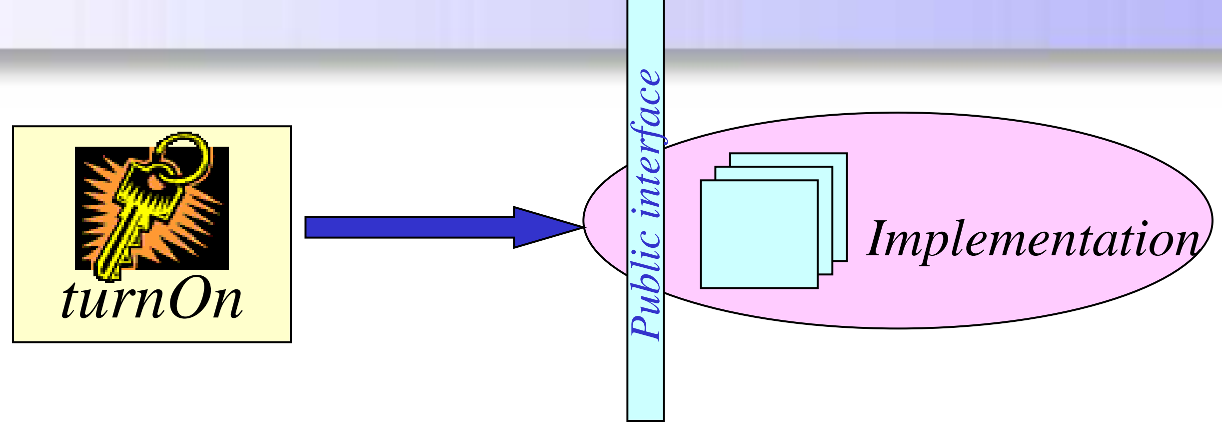
Private Members

- **How** the object does its work (how it is implemented)
  - For example, the ignition key interacts with the electric circuit of the vehicle, the engine, etc.
  - ***In pure object-oriented systems, state is completely private and can only be modified through the public interface***
  - Eg: public method stop can change the value of the private attribute speed

Interactions between objects

- **Object modeling** describes:
  - Objects and
  - Their interrelations
- To do a task, an object can **delegate** some work to another object, that can be part of itself, or can be any other object in the system
- Objects interact with each other by sending **messages**

Message passing



- An object sends a **message** to another object
  - By **calling** a method (method call / method invocation)
- Messages are handled by the **public interface** of the receiving object
  - We can only call methods from another object that are **public** or **accessible** from the calling object
- The receiving (called) object will react:
  - **Changing its state**, i.e. modifying its attributes, and/or
  - **Sending other messages**, i.e. calling other (public or private) methods from the same object (from himself) or calling other methods from other objects (public or accessible from that object)

Constructors
Ideas to recall

- When an object is created, its members are ***initialized*** using a constructor method
- Constructors:
  - Have the ***same name*** as the class
  - They have ***no return type*** (not even void)
- At least 1 constructor is recommended to exist
- Several constructors can exist that are distinguished by their parameters (***overloading***)
- A ***default constructor*** without parameters is created if no explicit constructors are defined, that initializes the attributes to their default values
- If there is a constructor in the class, the default constructor no longer exits. In that case, if a constructor without parameters is desired, it needs to be explicitly declared

Overloading
What is it?

- Two methods with the ***same name*** can be defined in a class if they have ***different parameters***
- It is widely used for constructors
- The method actually executed depends on the parameters passed when it is called
- In this case, ***no*** information ***hiding*** exists, both methods can be accessed

Overloading
What is it used for?

Classroom
- name
- description
- location
- printName()
- printDescription()
- printDescription(String furniture)
- printLocation()

*Although they have equal **names**, they are two different methods, because they have **different parameters***

*They have **different functionality** as shown in the example*

printDescription() → describe the classroom in general

printDescription(String furniture) → describe the furniture inside the classroom that is passed as a parameter

Systems Programming

Object ORIENTED programming (basic)

Julio Villena Román (LECTURER)

<jvillena@it.uc3m.es>

CONTENTS ARE MOSTLY BASED ON THE WORK BY:

M.Carmen Fernández Panadero, Raquel M. Crespo García
Carlos Delgado Kloos and Natividad Martínez Madrid

Scenario V:
Reusing code. Inheritance

- Once you are able to create your own classes, you are ready to work in teams and reuse code developed by your colleagues. Your team will provide you with a set of classes and you are required to create specializations or generalizations of them.
- Objective:
  - Be able to create a **derived class** adding some characteristics (attributes) and behavior (methods) to an existing class.
  - Be able to extract all the common code from a set of similar classes in order to group it into a new **parent class** so that it is easier to maintain.
  - Be able to create objects, and reference and access their attributes and methods, depending on their position in the inheritance hierarchy and their modifiers.
- Work plan:
  - Memorize the **naming** related to inheritance
  - Memorize the Java **syntax** related to inheritance (**extends**) and to reference (**super**, **this**) and access (**modifiers**) to the different members.
  - Know basic inheritance mechanisms, such as attribute **hiding**, **overriding** of methods and **overloading** of constructors, and know what they are used for and how they are used.

Contents

- Basic inheritance concepts
- Inheritance hierarchy
- Overriding I: Attribute Hiding
- Overriding II: Method Overriding
- Constructors of derived classes
- Static and Final Modifiers
- Scope and access

Inheritance
What is it?

- Mechanism for software reuse
- Allows to define from a class other related classes that can be a:
  - ***Specialization*** of the given class. (e.g. “Car” class is a specialization of the class “Vehicle”)
    - **Scenario**: We have to implement a new class that is very similar to a previous one but it needs additional information (characteristics and behaviour)
    - **Solution**: Create a class derived from the old one and add it new functionality without having to rewrite the common code
  - ***Generalization*** of the given class (e.g. the “Vehicle” class is a generalization of the “Car” class)
    - **Scenario**: We have a set of similar classes with code that is repeated in every class and thus difficult to update and maintain (e.g. a letter should be added to the serial number)
    - **Solution**: We move the code that is repeated to a single site (the parent class)

Inheritance

What is it?

Resource
- name
- description
- getName()
- getDescription()

Classroom
- name
- description
- location
- getName()
- getDescription()
- getLocation()

computer
- name
- description
- operatingSystem
- getName()
- getDescription()
- getOS()

Resource
- name
- description
- getName()
- getDescription()

Classroom
- location
- getLocation()

Computer
- operatingSystem
- getOS()

public class Classroom `extends` **Resource**

public class Computer `extends` **Resource**

The attributes and methods that appear in blue in the parent class are repeated in the subclasses. (Left picture)
It is not necessary to repeat the code, you only have to say that a class extends the other or inherits from it. (Right picture)

Inheritance
Naming

- If we define the car class from the vehicle class, it is said that:
  - “car” ***inherits*** attributes and methods from “vehicle”
  - “car” ***extends*** “vehicle”
  - “car” is a ***subclass*** of “vehicle”

    Derived class

    Child class
  - “vehicle” is a ***superclass*** of “car”

    Base class

    Parent class
- Inheritance implements the ***is-a*** relation
  - A car **is-a** vehicle; a dog **is-a** mammal, etc.

Inheritance

Declaration of subclasses

- The syntax for declaring subclasses is:

```
class Subclass extends Superclass { ... }
```

Person
Student
Employee
Professor
Secretary

```
class Person { ... }
class Student extends Person { ... }
class Employee extends Person { ... }
class Professor extends Employee { ... }
class Secretary extends Employee { ... }
```

Inheritance
Subclass

Attributes | Methods

firstName
lastName
yearBirth

print

Inherited from class Person

group
timetable

setGroup
printGroup

Class Student

Inheritance

How is it used?

```
public class Person {
  protected String firstName;
  protected String lastName;
  protected int birthYear;

  public Person () {
  }
  public Person (String firstName, String lastName,
                 int birthYear){
    this.firstName = firstName;
    this.lastName = lastName;
    this.birthYear = birthYear;
  }

  public void print(){
    System.out.print("Personal data: " + firstName
           + " " + lastName + " ("
           + birthYear + ")");
  }
}
```

Inheritance

How is it used?

```
public class Student extends Person {
    protected String group;
    protected char timetable;

    public Student() {
    }

    public Student(String firstName, String lastName,
                   int birthYear) {
        super(firstName, lastName, birthYear);
    }

    public void setGroup(String group, char timetable)
       throws Exception {
        if (group == null || group.length() == 0)
            throw new Exception ("Invalid group");
        if (timetable != 'M' && timetable != 'A')
            throw new Exception ("Invalid timetable");

        this.group = group;
        this.timetable = timetable;
    }

    public void printGroup(){
        System.out.print(" Group " + group + timetable);
    }
}
```

Inheritance

How is it used?

```
public class Test {
     public static void main (String[] args) throws Exception{

          Person neighbour = new Person ("Luisa", "Asenjo Martínez",
 1978);

          Student aStudent = new Student ("Juan", "Ugarte López",
 1985);
          aStudent.setGroup("66", 'M');

          neighbour.print();

          aStudent.print();
          aStudent.printGroup();
     }
 }
```

Inheritance

Consequences of extension of classes

- Inheritance of the interface
  - The public part of the subclass contains the public part of the superclass
    - The **`Student`** class contains the method **`print()`**
- Inheritance of the implementation
  - The implementation of the subclass contains the implementation of the superclass
    - When calling the method of the superclass on an object of the subclass (**`aStudent.print()`**) the expected behaviour takes place

Inheritance

Inheritance hierarchy in Java

- In Java, all classes are related through a **single inheritance hierarchy**
- A class can:
  - Explicitly inherit from another class, or
  - Implicitly inherit from the Object class (defined in the Java core)
- This is the case both for predefined classes and for user-defined classes

Inheritance hierarchy

- Object
  - Boolean
  - Character
  - Number
    - Integer
    - Long
    - Float
    - Double
  - Person
    - Student
    - Employee
      - Secretary
      - Professor
  - …

this and super references

- **this** references to the current class object
- **super**
  - references the current object casted as if it was an instance of its superclass
  - With the **super** reference, the methods of the base class can be explicitly accessed
  - **super** is useful when overriding methods

```
public class Student extends Person {
 // the rest remains the same
  public void print(){
     super.print();
     System.out.print("Group:" + group+ schedule);
  }
}
```

Inheritance
Overriding

- Modification of the elements of the base class inside the derived class
- The derived class can define:
  - An attribute with the same name as one of the base class → ***Attribute hiding***
  - A method with the same signature as one of the base class → ***Method overriding***
- The second case is more usual

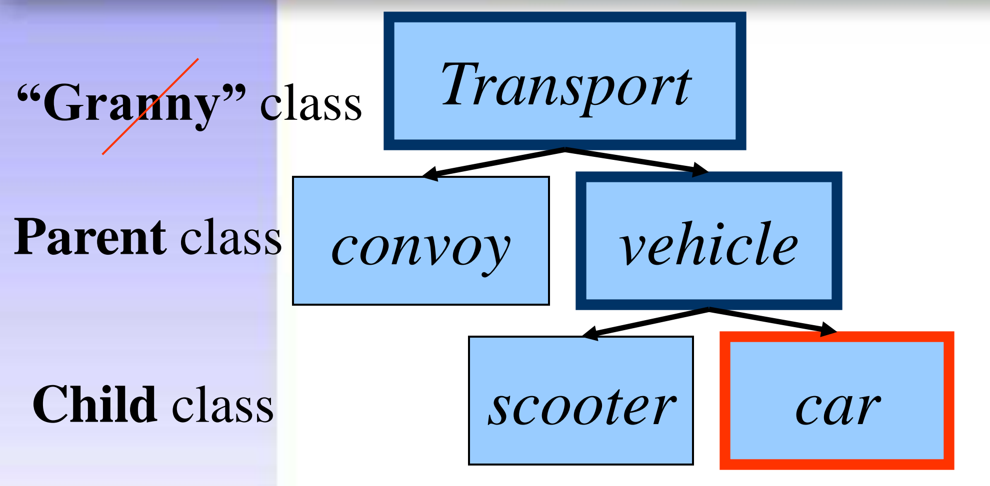
Overriding I (Shadowing)

Attribute hiding

- If we define an ***attribute*** (***variable***) in a subclass with the same name and type that other one in the superclass, the variable in the superclass remains hidden.
- We can access one variable or the other using ***this*** and ***super***.
  - E.g.: “Car" ***extends*** from “Vehicle" and “Vehicle" ***extends*** from “Transport”
  - We define the variable String name in the three classes
  - How can we know if we are referring to the name of transport, the name of the vehicle or the name of the car?

Overriding I (Shadowing)

Attribute hiding



`String name = "ground"`

`String name = "car"`

`String name = "Ferrari"`

- How can I access hidden variables?
  - ***name*** (car name)
  - ***this.name*** (car name)
  - ***super.name*** (vehicle name)
  - ***((vehicle)this).name*** (vehicle name)
  - ~~***super.super.name*** (WRONG)~~
  - ***((transport)this).name*** (transport name)

Variables Child class: visibles

Variables Parent class hidden

Overriding I (Shadowing)

Attribute hiding

- When accessing an attribute, the type of the reference is used for deciding which value to access
- The attribute of the subclass needs to have the same name as the one of the superclass
  - But not necessarily the same type
- Not very useful in practice
  - Allows the superclasses to define new attributes without affecting the subclass

Overriding I (Shadowing)

Attribute hiding. Example 1

```
Student a = new Student(...);
System.out.println(a.firstName);

Person p = a;
System.out.println(p.firstName);
```

Attributes Methods

firstName
lastName
yearBirth

print()

Inherited from class Person

firstName
group
timetable

setGroup(String s)
printGroup()

Class Student

Overriding I (Shadowing)

Attribute hiding. Example 2

```
class SuperShow {
  public String str = "SuperStr";
}

class ExtendShow extends SuperShow {
  public int str = 7;
}

class Show {
   public static void main (String[] args) {
      ExtendShow ext = new ExtendShow();
      SuperShow sup = ext;
      System.out.println(sup.str);
      System.out.println(ext.str);
   }
}
```

prints SuperStr

prints 7

Overriding II

Method overriding

- If the subclass defines a ***method*** with the same **signature** (name + number and type of the parameters) the method in the superclass is hidden
- If the ***final*** modifier is used in a method, this method can not be overriden
- How can we access hidden methods?
  - *start()* (run the start method of the car)
  - *this.start()* (run the start method of the car)
  - *super.start()* (run the start method of the vehicle)
  - ~~*super.super.start()*~~ (WRONG)

Methods of the child class: visible

Methods of the parent class: hidden

Overriding II
Method overriding

Resource
- name
- description
- getName()
- getDescription()

Classroom
- name
- description
- location
- getName()
- getDescriptoin()
- getLocation()

Computer
- name
- description
- operatingSystem
- getName()
- getDescription()
- getOS()

Resource
- name
- description
- getName()
- getDescription()

Classroom
- description
- location
- getLocation()
- getDescription()

Computer
- operatingSystem
- getOS()

`this.getDescription()`

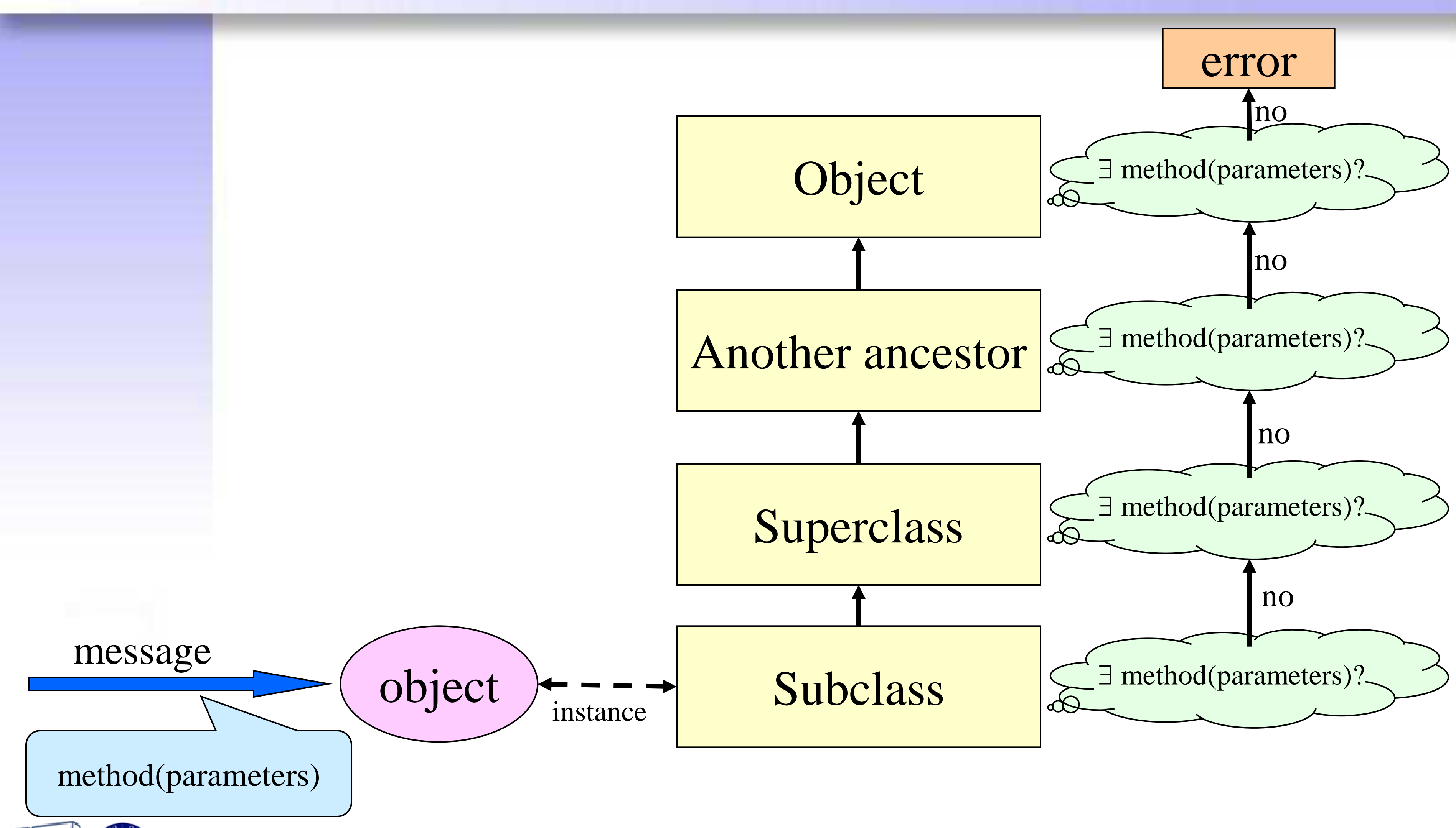
`super.getDescripcion`

public class Classroom `extends` Resource

public class Computer `extends` Resource

Overriding II

Method overriding



Universidad Carlos III de Madrid

Overriding II

Method overriding

- When sending a message to an object, the selected method:
  - Depends on the class which the object is an instance of
  - Does not depend on the reference class to which it is assigned, as in the case of attributes

Overriding II

Method overriding. Example

```
 class SuperShow {
  public String str = "SuperStr";
   public void show() {
      System.out.println("Super.show: " + str);
   }
}
class ExtendShow extends SuperShow {
  public String str = "ExtendStr";
   public void show() {
      System.out.println("Extend.show: " + str);
   }
}
class TestShow {
   public static void main (String[] args) {
      ExtendShow ext = new ExtendShow();
      SuperShow sup = ext;
      sup.show();
      ext.show();
   }
}
```

Both print:
"Extend.show: ExtendStr"

Overriding II
Final methods

- Method overriding is useful for
  - Extending the functionality of a method
  - Particularizing the functionality of a method to the derived class
- If it is not desired that subclasses are able to modify a method or an attribute of the base class, the reserved word **`final`** should be applied to the method or attribute

Overriding vs. overloading

- **Overriding**: The subclass substitutes the implementation of a method of the superclass
  - Both methods need to have the same signature
- **Overloading**: There is more than one method with the same name but different signature
  - The overloaded methods can be declared in the same class or in different classes in the inheritance hierarchy

Constructors and inheritance

- To create an object, the following steps are done:
  1. The base part is created
  2. The derived part is added
  – If the base class of the object inherits from another class, step 1 is applied in the order of the inheritance chain, until we reach **`Object`**
- For example, when creating a **`Student`** object, that extends **`Person`**, the steps would be:
  1. The part corresponding to **`Person`** is created. To do so:
     1. The part corresponding to **`Object`** is created.
     2. The **`Person`** elements are added
  2. The **`Student`** elements are added

Constructors and inheritance

- A call to the constructor of the base class is always done in the constructor of the derived class.
- This is the first action of the constructor (always in the first line)
- Two possibilities:
  - Not explicitly indicate it
  - Explicitly indicate it (mandatory in the first line)

Constructors and inheritance

1. If it is not explicit, Java automatically inserts a call to **super()** in the first line of the constructor of the derived class

```
public Student (String firstName, String lastName,
    int yearBirth, String group, char timetable) {
   // Java inserts here a call to super()
   this.firstName = firstName;
   this.lastName = lastName;
   this.birthYear = birthYear;
   this.group = group;
   this.timetable = timetable;
}
```

Constructors and inheritance

2. Explicitly coded

```
public Student (String firstName, String lastName,
            int birthYear, String group, char
   timetable) {
   super(firstName, lastName, birthYear);
   this.group = group;
   this.timetable = timetable;
}
```

Modifiers and access
Final

- The **final** modifier can be applied to:
  - **Parameters**: Means that the value of such parameter cannot be changed inside the method

```
public void myMethod(final int p1, int p2){} //p1 value cannot be changed
```

  - **Methods**: Means that it cannot be overridden in derived classes

```
public final void myMethod(){} //myMethod cannot be overridden
```

  - **Classes**: Avoid extending the class. It cannot be inherited.

```
public final class myClass(){} //myClass cannot be extended
```

Modifiers and access

Static (static members)

- ***static*** modifier
- Static members exist only ***once per class***, independently of the number of instances (objects) of the class that have been created or even if none (instances) exists.
- Static members can be accessed using the ***class name***.
- An static method ***cannot*** access non-static members directly, it must first create an object.

Modifiers and access

Static. Some rules

– Static members are invoked with:

```
ClassName.staticMethod();
ClassName.staticAttribute;
```

– Non static members require an instance (object) in order to be accessed.

```
ClassName objectName = new ClassName();
```

– Non static members are invoked with:

```
objectName.normalMethod();
objectName.normalAttribute;
```

– When a static member is invoked (called) from inside the same class, the class name can be deleted. I.e. it can be written:

```
staticMethod();
staticAttribute;
```

instead of:

```
ClassName.staticMethod();
ClassName.staticAttribute;
```

MODIFIERS		class	method	attribute
access	public	Accesible to any other class		
	(friendly)	Accessible only to classes in the same package		
	protected	🚫	Accessible to the class and its subclasses	
	private	Applied to inner classes	Accessible only inside the class	
other	abstract	Cannot be instantiated For **inheriting** from them At least 1 abstract method	Has no code It is implemented in the subclasses or child classes	🚫
	final	Cannot be extended. It is a leaf in the inheritance tree.	Cannot be overridden. It is **constant** and cannot be modified in the child classes.	Its value cannot be changed, it is **constant** . It is normally used together with static.
	static	Maximum level class.	It is the same for all of the class objects. Use: ClassName.method ();	It is the same for all of the class objects.

Systems Programming

Object ORIENTED programming (advanced)

Julio Villena Román (LECTURER)

<jvillena@it.uc3m.es>

CONTENTS ARE MOSTLY BASED ON THE WORK BY:

M.Carmen Fernández Panadero, Raquel M. Crespo García
Carlos Delgado Kloos and Natividad Martínez Madrid

Contents

- Polymorphism
- Dynamic binding
- Casting. Types compatibility
- Abstract classes and methods
  - Partial implementations
  - Polymorphism with abstract classes
- Interfaces (concept and implementation)
  - Multiple inheritance
  - Polymorphism with interfaces
- Packages
- Exceptions

Inheritance

Inheritance hierarchy in Java

- In Java, all classes are related through a **single inheritance hierarchy**
- A class can:
  - Explicitly inherit from another class, or
  - Implicitly inherit from the Object class (defined in the Java core)
- This is the case both for predefined classes and for user-defined classes

Polymorphism
What is it?

- Capacity of an object for deciding which method to apply, depending on the class it belongs to
  - A call to a method on a reference of a generic type (e.g. base class or interface) executes different implementations of the method depending on which class the object was created as
- Poly (many) + morph (form)
  - One function, different implementations
- Allows to design and implement extensible systems
  - Programs can process generic objects (described by references of the superclass)
  - The specific behaviour depends on the subclasses
  - New subclasses can be added later

Polymorphism
Exercise

- Program a class:
  - `Shape`, which represents a bi-dimensional shape (parallelepiped), with two attributes, one per each dimension, and an `area()` method that calculates the area. Its default return value is 0.
  - `Triangle`, which extends the `Shape` class and overrides the `area()` method
  - `Rectangle`, which extends `Shape` and overrides the `area()` method
  - `ShapeList`, which has an attribute of type array of `Shape`, and a method `totalArea()` that returns the sum of the areas of all the shapes
- What should be changed in `ShapeList` if a new class `Ellipse` is added?

Polymorphism: dynamic binding

- The power of method overriding is that the correct method is properly called, even though when referencing the object of the child class through a reference of the base class
- This mechanism is called "***dynamic binding***"
  - Allows detecting **during running time** which method is the proper one to call
- The compiler does not generate the calling code during compiling time
  - It generates code for calculating which method to call

Casting (Type conversion)
Syntax and terminology

- Syntax:

```
(type) identifier
```

- Two types of casting:
  - ***widening***: a subclass is used as an instance of the superclass (e.g.: calling a method of the parent class which has not been overriden). Implicit.
  - ***narrowing***: The superclass is used as an instance of one subclass. Explicit conversion.
- Casting can only be applied to parent and child classes, not to sibling classes

Casting (Type conversion)

Widening or upcasting

1. **Upcasting**: compatibility upwards (towards the base class)
   – An object of the derived class can always be used as an object of the base class (because it implements an "*is-a*" relationship)

```
Person p = new Student();
```

Casting (Type conversion)

Narrowing or downcasting

2. **Downcasting**: compatibility downwards (towards the derived classes)
   - Downcasting cannot be applied by default, because an object of the base class is not always an object of the derived class

```
Student s = new Person(); // wrong
```

   - It is only possible when the reference of the base class actually points to an object of the derived class
   - In these cases, an explicit *casting* must be applied

Casting (Type conversion)
Explicit and implicit

Reference to superclass

Object of superclass

Automatic upcasting

Explicit downcasting

Explicit upcasting if ambiguous
(e.g. Method overriding

Reference to subclass

Object of subclass

Casting (Type conversion)

Example

```
public class Test2 {
  public static void main (String[] args) {
    Person p1;
    //implicit upcasting - works
    Student s1 = new Student();
    p1 = s1;

    Student s2;
    //implicit downcasting - does not work
    s2 = p1; //error because no explicit casting is done

    //explicit downcasting - works
    s2 = (Student) p1;    // p1 actually references an
                          // instance of class Student
```

A student is always a person (**implicit**)

A person is not always a student

If someone, besides being a person, is also a student (not always happens), (s)he can be required stuff as a student, but must be **explicitly** stated that (s)he will be treated as a student.

Universidad Carlos III de Madrid

Casting (Type conversion)
Example

```
   Person p2 = new Person();
   Student s3;
//implicit downcasting - does not work
   s3 = p2;    //compiler error

//explicit downcasting - does not works sometimes
//ClassCastException will be thrown
//because p2 does not refer to a Student object
   s3 = (Student) p2; //error

//implicit downcasting - does not work
   Student s4 = new Person(); //error
  }
}
```

A person not always is a student. It cannot be **implicitly** assumed.

A person is sometimes a student, but if not (it has not been created as such), it cannot be treated as such, not even though **explicitly** trying.

A person is not always a student. It cannot be assumed **implicitly**.

Casting (Type conversion)

instanceof operator

- **Syntax:**

```
object instanceOf class
```

  – Checks if an object is really an instance of a given class

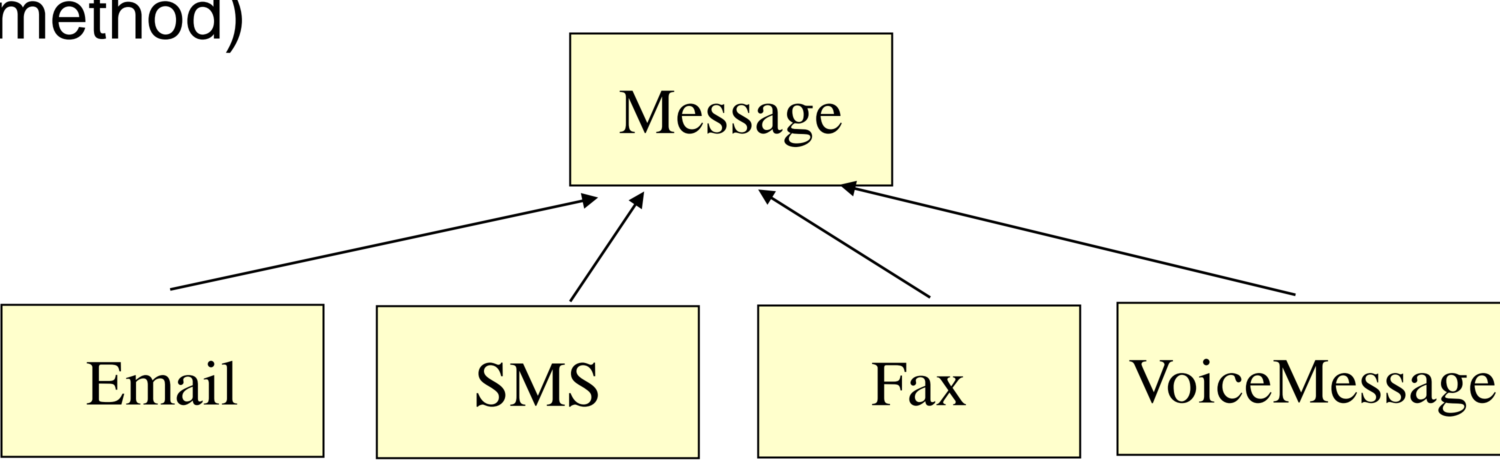
- Example:

```
public Student check(Person p) {
  Student s = null;
  if (p instanceOf Student)
     s = (Student) p;
  return s;
}
```

Abstract classes

What is an abstract class?

- An abstract class is a class that has ***at least one abstract method*** (not implemented, without code).
- It declares the ***structure*** of a given ***abstraction***, without providing all the implementation details (i.e. without implementing completely every method)



Abstract classes

Characteristics

- Classes and methods are defined as abstract using the reserved word ***abstract***

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

- The abstract modifier cannot be applied to:
  - **constructors**
  - **static** methods
  - **private** methods

Abstract classes
Characteristics

- Abstract classes ***cannot be instantiated***
  - References to abstract classes can exist
  - But they point to objects of classes derived of the abstract class

```
Shape fig = new Rectangle(2,3);
```

- Abstract classes ***can be extended***
- In an abstract class, there can be both
  - **abstract** methods
  - **non abstract** methods

Abstract classes

Purpose: partial implementations

- Abstract classes are normally used for representing ***partially implemented*** classes
  - Some methods are not implemented but declared
- The objective of partial implementations is to provide a ***common interface*** to all derived classes
  - Even though in cases when the base class has not information enough to implement the method

Abstract classes

abstract methods

- Methods declared but no implemented in abstract classes

```
abstract returnType name(parameters);
```

  - Methods are declared abstract using the reserved word **`abstract`**
- Classes inheriting from the abstract class must implement the abstract methods of the superclass
  - Or they will be abstract themselves too

NOTE: **No brackets**! They are not implemented, thus only a semicolon (**;**) follows the declaration

Abstract classes

How are they used?

*The Resource class is **abstract** because **one of its methods*** **`describe()`** ***has no code***

* Grey color indicates “having no code”

***All classes extending** Resource **must provide** the implementation of the* **`describe()`** *method (with its **code**)*

** Bold indicates “having code”

Resource
- name
- description
- getName()
- describe()*

Classroom
- location
- getLocation()
- **describe()****

Computer
- operatingSystem
- getOperatingSystem()
- **describe()****

```
public abstract class Resource
public class Classroom extends Resource
public class Computer extends Resource
```

Abstract classes

How are they used? Example

```
abstract class Shape {     // any parallelepiped
    double dim1;
    double dim2;

     Shape(double dim1, double dim2) {
       this.dim1 = dim1;
       this.dim2 = dim2;
    }

    abstract double area();
}
```

```
class Rectangle extends Shape {
    Rectangle(double dim1, double dim2) {
       super(dim1,dim2);
    }
    double area() {
       return dim1*dim2;    // rectangle area
    }
}
```

Abstract classes

Polymorphism

An array of objects of type **Resource (abstract)**

Array elements are instances of a concrete (non-abstract) class (**computer and classroom**)

```
public class ResourcesTest{

 public static void main(String args[]){
  Resource[] myResources = new Resource[3];

  myResources[0] = new Classroom("classroom1");
  myResources[1] = new Computer("PC1");
  myResources[2] = new Classroom("classroom2");

  for(int i=0; i<myResources.length;i++){
   myResources[i].describe();
  }

 }
}
```

Resource
- name
- description
- getName()
- describe() *

Classroom
- location
- getLocation()
- **describe()****

Computer
- operatingSystem
- getOperatingSystem()
- **describe()****

A call to the describe method on objects of type **Resource** and during running time it will be checked which type of object is contained (**Computer or Classroom**) and the proper method will be called. – **Dynamic binding**

public `abstract` class Resource{…}

public class Classroom `extends` Resource{…}

public class Computer `extends` Resource{…}

* Grey color means "having no code"

** Bold means "having code"

Interfaces
What is an interface?

- Interfaces take the abstract class concept one step further
  - ***<u>All</u> methods in the interface are abstract***
  - They could be thought of as "*like*" a "pure" abstract class
- Interfaces are always ***public***
  - Interface attributes are implicitly public, static and final
  - Interface methods have no access modifiers, they are public
- Interfaces are ***implemented*** by classes
  - A ***class*** implements an interface defining the body of ***all*** the methods
  - An ***abstract class*** implements an interface implementing or declaring abstracts the methods
  - A class can implement one or more interfaces (~multiple inheritance)

Interfaces
What is an interface?

- An ***interface*** is a pure **design** element
  - **What** to do
- A ***class*** (including abstract ones) is a mix of **design and implementation**
  - **What** to do and **how**
- Interfaces represent a complete abstraction of a class
  - An interface abstracts the public characteristics and behaviors of their implementations (how those behaviours are executed)
- Different classes can implement the same interface in different ways

Interfaces

How are they used?

*Shape is not a class, it is an **interface**, it just defines the behavior but not the implementation*

Shape
•area()

implements

Circle
•radius
•**area()**

Parallelepiped
•dim1, dim2
•area()

extends

Rectangle
•**area()**

*All classes implementing Shape must **provide an implementation** for **all methods** declared in Shape (or declared them abstract)*

```
public interface Shape
```

```
public class Circle implements Shape
```

```
public class Parallelepiped implements Shape
```

Interfaces

Declaration

- Syntax:

```
<public> interface name {
  type variable = value;
  returnType method(parameters);
}
```

  - **`public`** modifier is optional (interfaces are public)
  - All methods are implicitly **`abstract`** and **`public`**
  - Interface attributes are **`public`**, **`static`** and **`final`**
    - They represent constants

NOTE: **No brackets**!! As the method is not implemented, only a semicolon (**;**) follows the declaration

Interfaces

Implementation

- If a class implements an interface, it implements all abstract methods declared in such interface
- Represented with the reserved word **implements**

```
class MyClass implements Interface1, Interface2 {
   ...
}
```

```
class DerivedClass extends BaseClass
    implements Interface1, Interface2 {
   ...
}
```

Interfaces
Use. Example

- Define an interface for printable objects
  - Method `void print()`

- Modify the `Rectangle` and `Email` classes so that they implement the `Printable` interface

Interfaces

Use. Example

```
interface Printable {
  void print();
}
```

```
class Email extends Message
            implements Printable {

  public void print() {
    System.out.println("Printing email");
    System.out.println(message);
  }
```

NOTE: **No brackets**!! It is not implemented, thus just a semicolon (;) follows the declaration

Interfaces
Use. Example

```
public class Rectangle extends Shape implements Printable {
    [...]
    public void print(){
        System.out.println("Printing Rectangle (" + dim1 + "x" + dim2 + ")");
        StringBuffer res = new StringBuffer();
        for (int i = 0; i <= dim1+1; i++)
            res.append("* ");
        res.append("\n");
        for (int j = 0; j < dim2; j++){
            res.append("* ");
            for (int i = 1; i <= dim1; i++)
                res.append("  ");
            res.append("*");
            res.append("\n");
        }
        for (int i = 0; i <= dim1+1; i++)
            res.append("* ");
        System.out.println(res);
    }
}
```

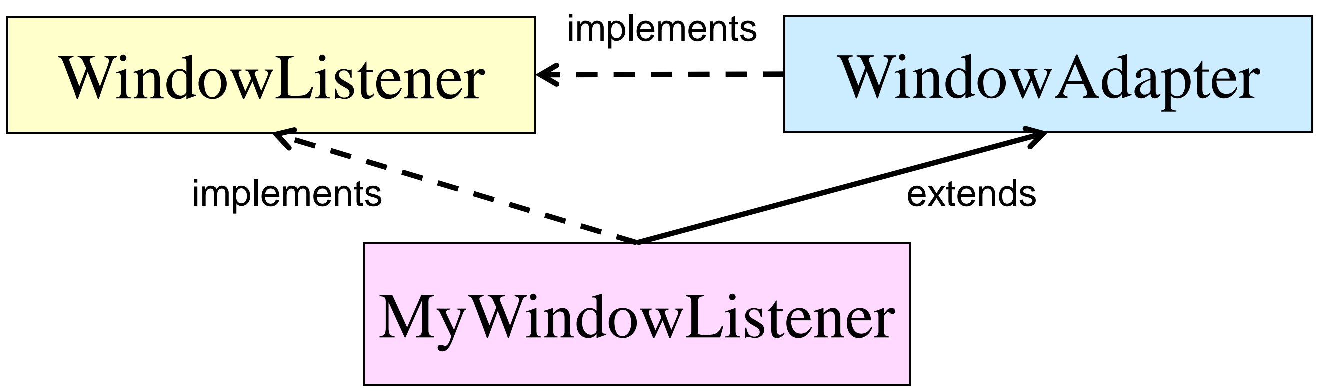
Interfaces

Use. Extending interfaces with inheritance

- Interfaces can be extended (inherited) too
- Interface inheritance adds the methods to be included in the classes implementing the interfaces
  - The class implementing the derived interface must include all the methods declared in both the derived as well as the base interfaces

Interfaces
Use. Example

- A well-designed program will include **interfaces** and **extensions** of classes
- In the future, programmers can easily modify it:
  - **Extending** the implementation, or
  - **Implementing** the interface



Interfaces

Purpose. Multiple inheritance

method1

ClassA

method1

Class B

extends

extends

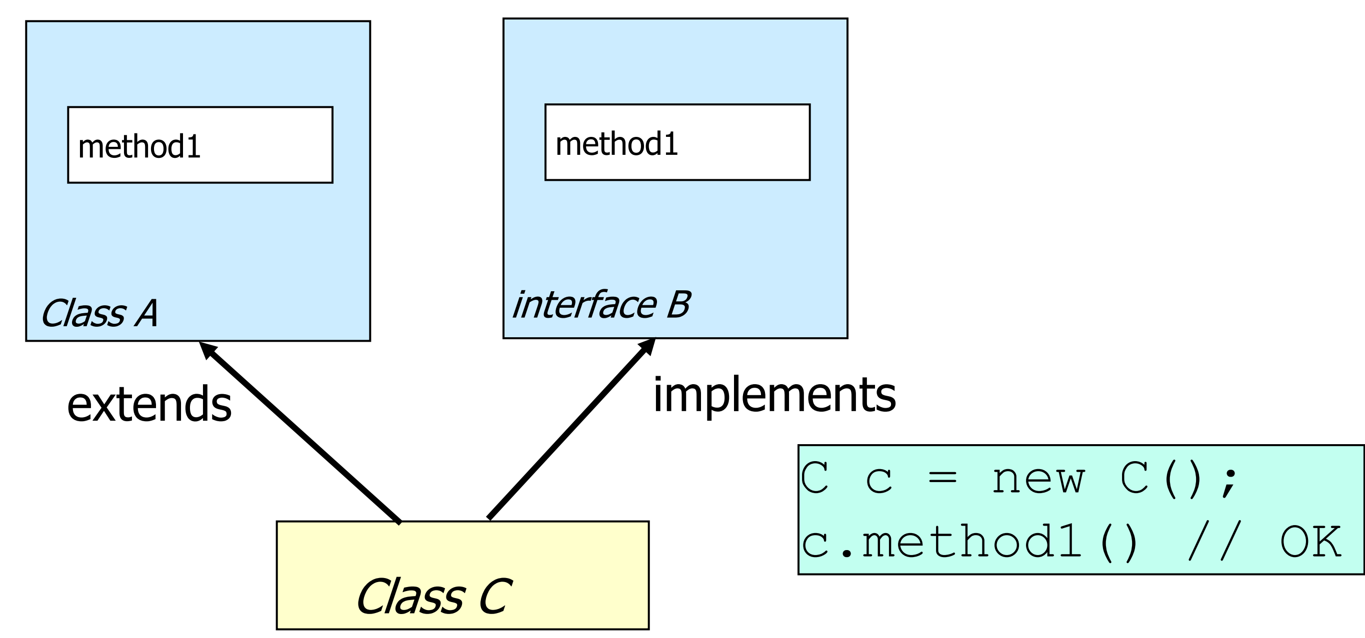
Class C

```
C c = new C();
method1() // ????
```

- Java does **not** allow multiple inheritance
- Similar functionality is provided with **interfaces**

Interfaces

Purpose. Multiple inheritance

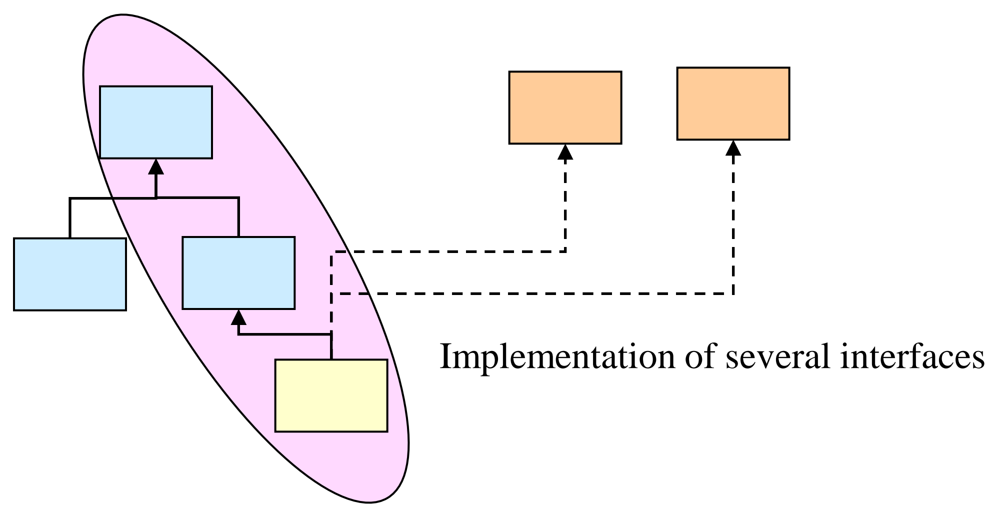


- A class extends only one base class
- But can implement several interfaces

Interfaces

Purpose. Multiple inheritance

- ***Simple inheritance*** of **implementations**
  - Extension on just one class
- ***Multiple inheritance*** of **interfaces**
  - Implementation of several interfaces

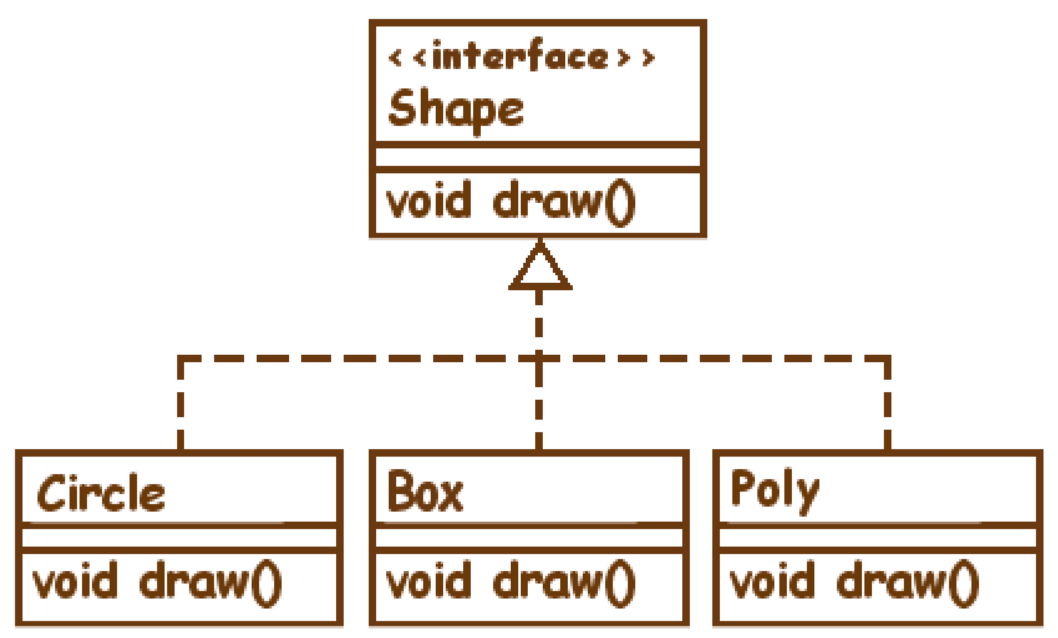


Interfaces

Purpose. Polymorphism

- Polymorphism: "one interface, multiple methods"
- Interfaces support dynamic resolution of methods during execution time (dynamic binding)
- What is the difference between interface implementation and inheritance?
  - Interfaces do not belong to the hierarchy of inheritance

Exercise: JavaRanch



Exercise: JavaRanch

```
import java.awt.* ;
public interface Shape
{
    public void draw( Graphics g );
}
```

```
import java.awt.* ;
public class Circle implements Shape
{
    private int x ;
    private int y ;
    private int wide ;
    private int high ;
    private Color color ;

    Circle( int x , int y , int wide , int high , Color color )
    {
        this.x = x ;
        this.y = y ;
        this.wide = wide ;
        this.high = high ;
        this.color = color ;
    }

    public void draw( Graphics g )
    {
        g.setColor( color );
        g.fillOval( x , y , wide , high );
    }

}
```

Exercise: JavaRanch

```
import java.awt.* ;
public class Box implements Shape
{
        private int x ;
        private int y ;
        private int wide ;
        private int high ;
        private Color color ;
        Box( int x , int y , int wide , int high , Color color )
        {
            this.x = x ;
            this.y = y ;
            this.wide = wide ;
            this.high = high ;
            this.color = color ;
        }

        public void draw( Graphics g )
        {
            g.setColor( color );
            g.fillRect( x , y , wide , high );
        }
}
```

Exercise: JavaRanch

```
import java.awt.* ;
public class Poly implements Shape
{
    int[] x ;
    int[] y ;
    private Color color ;

    Poly( int[] x , int[] y , Color color )
    {
        this.x = x ;
        this.y = y ;
        this.color = color ;
    }

    public void draw( Graphics g )
    {
        g.setColor( color );
        g.fillPolygon( x , y , x.length );
    }
}
```

Exercise: JavaRanch

```
import java.awt.* ;
public class ShowShapes extends Frame {
    static int[] vx = { 200 , 220 , 240 , 260 , 280 , 250 , 230 };
    static int[] vy = { 150 , 150 , 190 , 150 , 150 , 210 , 210 };

    static Shape[] shapes = {
        // J
        new Box( 50 , 70 , 100 , 20 , Color.red ) ,
        new Box( 90 , 70 , 20 , 110 , Color.blue ) ,
        new Circle( 50 , 150 , 60 , 60 , Color.green ) ,
        new Circle( 70 , 170 , 20 , 20 , Color.white ) ,
        new Box( 50 , 90 , 40 , 90 , Color.white ) ,

        // a
        new Circle( 130 , 150 , 60 , 60 , Color.green ) ,
        new Box( 170 , 180 , 20 , 30 , Color.blue ) ,
        new Circle( 150 , 170 , 20 , 20 , Color.white ) ,

        // v
        new Poly( vx , vy , Color.black ) ,

        // a
        new Circle( 290 , 150 , 60 , 60 , Color.green ) ,
        new Box( 330 , 180 , 20 , 30 , Color.blue ) ,
        new Circle( 310 , 170 , 20 , 20 , Color.white ) ,
    };

    ShowShapes() {
        setBounds( 200 ,150 , 400 , 250 );
        setVisible( true );
    }
    public void paint( Graphics g ) {
        for( int i = 0 ; i < shapes.length ; i++ )
            shapes[ i ].draw( g );
    }
    public static void main( String[] args ) {
        new ShowShapes();
    }
}
```

Object Orientation Summary

- ***Class*** (actual)
  - ***All*** methods are implemented
- ***Abstract class***
  - At least one method is **not implemented** but just declared
  - abstract modifier
- ***Interface***
  - ***No*** implementation at all
  - Reserved word: interface

Object Orientation Summary

- ***Class*** (actual or abstract)
  - Can ***extend*** (extends) ***only one*** base class (simple inheritance)
  - Can ***implement*** (implements) ***one or more*** interfaces (multiple inheritance)
    - Reserved word: extends
- ***Interface***
  - Can extend (extends) ***one or more*** interfaces

Packages

- A ***package*** groups ***classes*** and ***interfaces***
- The hierarchies in a package correspond to the hierarchies of folders in disk
- Dots are used for referring to subpackages, classes and interfaces in a package
  - E.g.: The Applet class in package **`java.applet`** provided by Java is imported when programming an applet

```
import java.applet.Applet;
```

  - The **`java.applet.Applet`** class is in the java/applet folder

Packages

- ***Using packages created by others***
  - Include in the classpath the path to the folder containing the package. E.g.: assuming **`PackageByOther`** is in **`c:\java\lib`** (Windows) or **`/opt/lib/`** (Linux)

```
set CLASSPATH=c:\PackageByOther;%CLASSPATH%                (Windows)
setenv CLASSPATH /opt/lib/PackageByOther:$CLASSPATH  (Linux)
```

  - In the class using the package, the corresponding import sentence must be included before the class declaration

```
import PackageByOther.*;
```

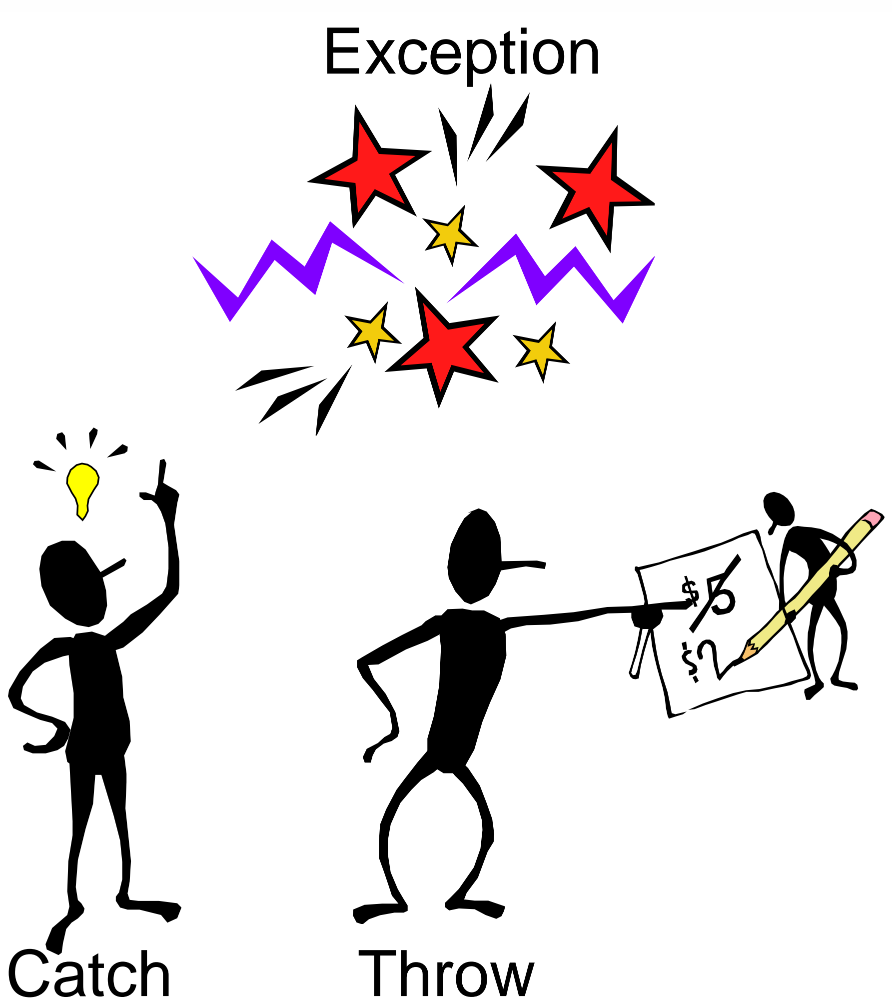
- ***Creating my own packages***
  - Save the classes in a folder named as the package
  - All classes belonging to the package must include the following sentence as the first one:

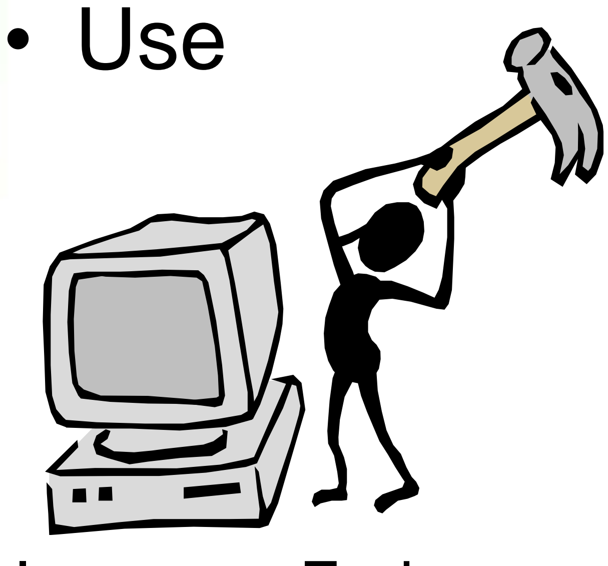
```
package myOwnPackage;
```

MODIFIERS		***class***	***method***	***attribute***
access	public	Accesible to any other class		
	(friendly)	Accessible only to classes in the same package		
	protected	🚫	Accessible to the class and its subclasses	
	private	Applied to inner classes	Accessible only inside the class	
other	abstract	Cannot be instantiated For **inheriting** from them At least 1 abstract method	Has no code It is implemented in the subclasses or child classes	🚫
	final	Cannot be extended. It is a leaf in the inheritance tree.	Cannot be overridden. It is **constant** and cannot be modified in the child classes.	Its value cannot be changed, it is **constant** . It is normally used together with static.
	static	Maximum level class.	It is the same for all of the class objects. Use: ClassName.method ();	It is the same for all of the class objects.

Exceptions

- What they are
- Purpose
- Type
- Use





Exceptions: What are they?

- **Events** that prevent the normal execution of the program
- When an exception occurs, an **exception object** is created and it is passed to the **execution control** system
- The execution control system:
  - **Searches** for a fragment of code that handles the exception
  - If no handling code is found, the program **ends**

Exceptions: Purpose

- For **separating** the code for error handling (`try-catch`) from the normal code
- For **propagating** errors in the calls stack (`throws`)
- For **grouping** and differentiating types of errors (as exceptions are objects, they can be grouped into classes)
- Every method must:
  - Either `catch` (catch)
  - Or `throw` (throws)

  any exception happening during its execution

Exceptions: Types

- Two main types:
  - Runtime exceptions (`RuntimeException`)
    - Not checked in compile time
    - E.g.: `NullPointerException`, `ArithmeticException`, `NumberFormatException`, `IndexOutOfBoundException`, etc.)
  - Exceptions checked during compile time
    - E.g.: Input/output exceptions (`IOException`, `FileNotFoundException`, `EOFException`)
    - **User-defined** (`MyException`)
- During compile time, it is checked that any exception (except runtime exceptions) are:
  - either **caught**
  - or **declared** to be thrown in the methods where they can happen

Exceptions: Use

- Exceptions appear:
  - Implicitly (when an error happens)
  - Explicitly: `throw new MyException(message)`
- What to do:
  - **Handle the exception**:
    - Enclose in a `try{}` block sentences that may generate exceptions
    - Enclose in the `catch(MyException e){}` block the sentences to be executed for handling the exception
  - **Throw the exception:**
    - `public void myMethod throws MyException`
- The `finally{}` block encloses the code that should always be executed