Object Oriented Java

I. Object ***based*** programming

II. Object ***oriented*** programing

M. Carmen Fernández Panadero

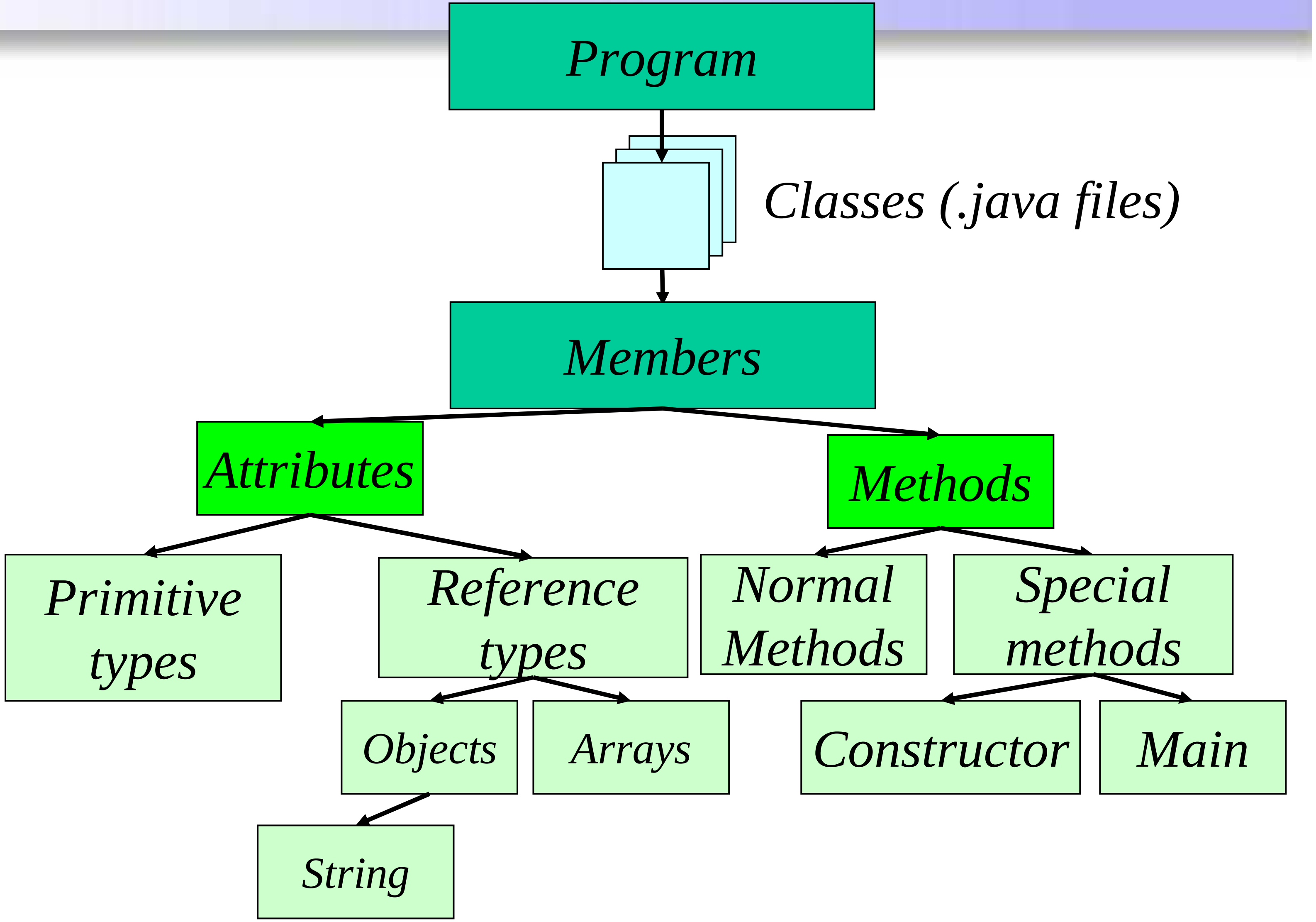
Raquel M. Crespo García

<mcfp, rcrespo@it.uc3m.es>

Contents

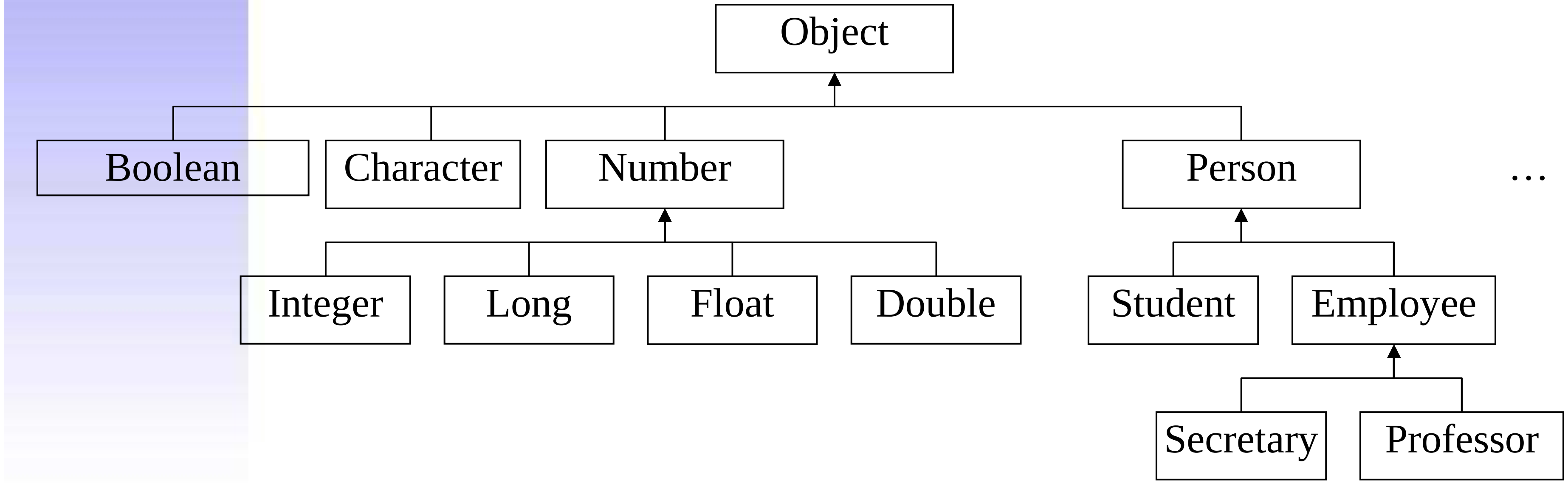
- Polymorphism
- Dynamic binding
- Casting. Types compatibility
- Abstract classes and methods
  - Partial implementations
  - Polymorphism with abstract classes
- Interfaces (concept and implementation)
  - Multiple inheritance
  - Polymorphism with interfaces
- Packages
- Exceptions

Review (Session 1)



Review (session 2)

Inheritance hierarchy in Java



Review (session 2)
Inheritance hierarchy in Java

- In Java, all classes are related in a unique inheritance hierarchy
- A class can:
  - **Explicitly inherit** of other class
  - or **implicitly inherit** of the ***Object*** class (defined in the Java core)
- This applies both for predefined classes as well as user-defined ones

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 designing and implementing extensible systems
  - Programs can process generic objects (described by references of the superclass)
  - The concrete behavior depends on the subclasses
  - New subclasses can be added later

Polymorphism

Exercise

- Program a class:
  - **Shape**, that 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**, that extends the **Shape** class and overrides the **area()** method
  - **Rectangle**, that extends **Shape** and overrides the **area()** method
  - **ShapesList**, that 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 **ShapesList** 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 overridden). 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

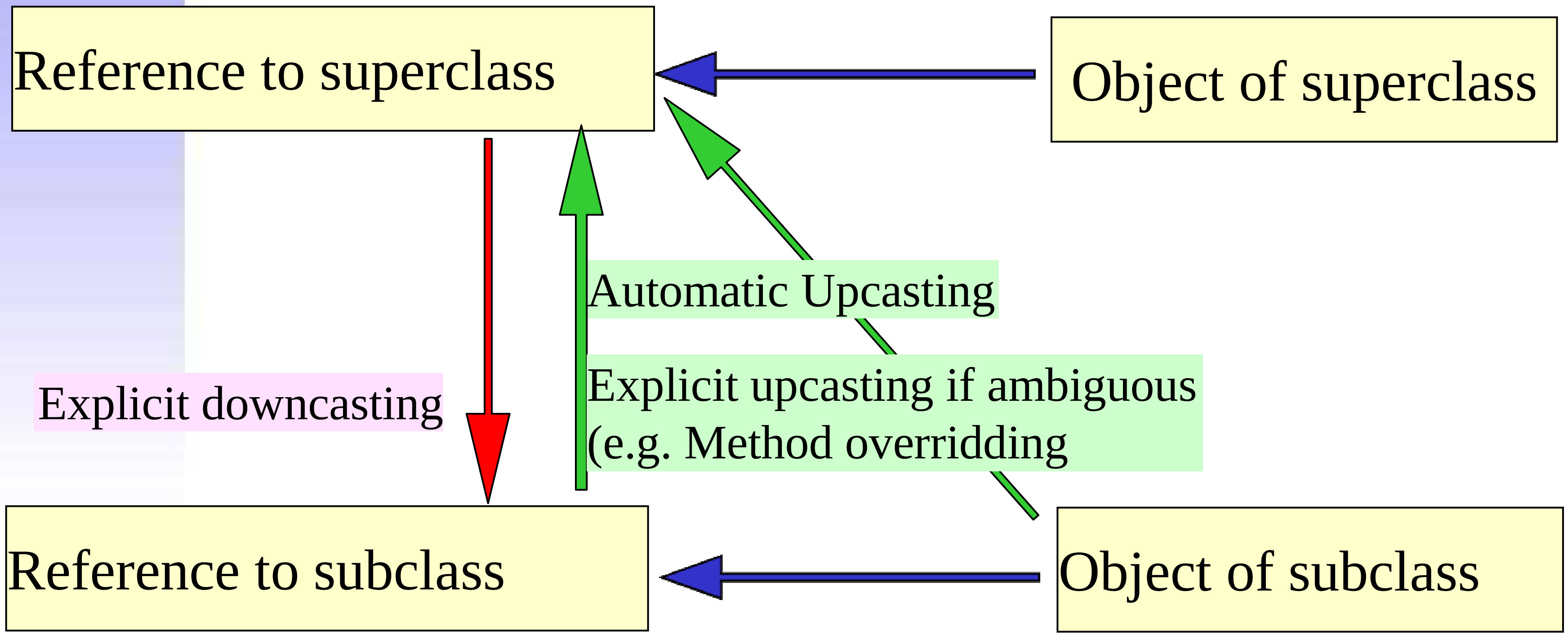
2. **Downcasting**: compatibility downwards (towards the derived classes)
   - Downcasting cannot be applied by default, because an object of the base class is not always an object of the derived class.

     ~~`Student s = new Person(); // error`~~

   - 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 overridding

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.

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

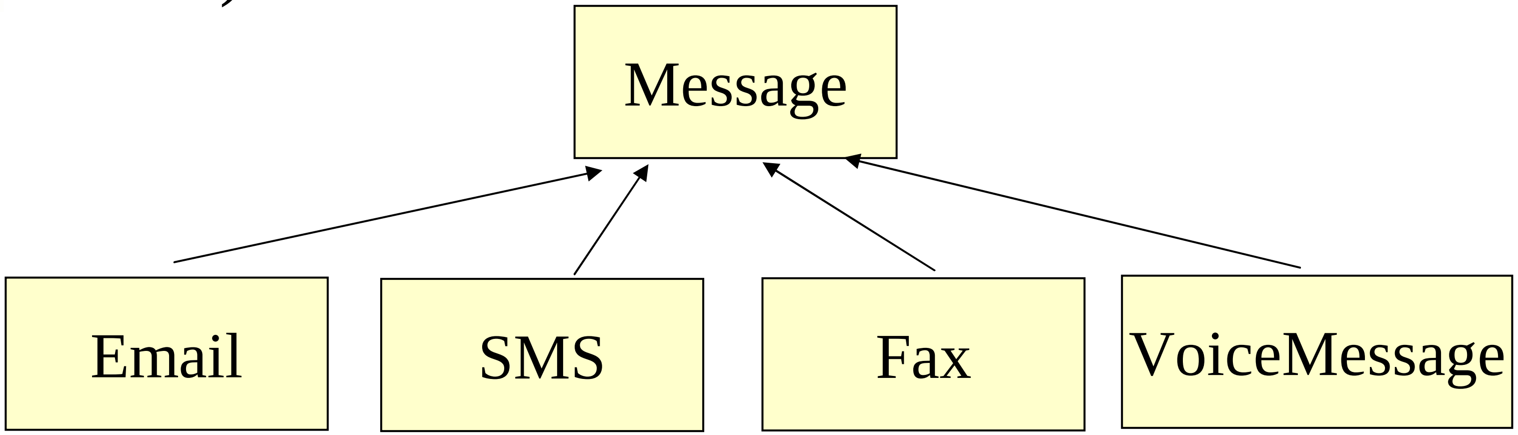
- Example:

```
  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 abstrac 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 of the abstract class must implement the abstract methods of the superclass
  - Or they will be abstract themselves too

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

Abstract classes

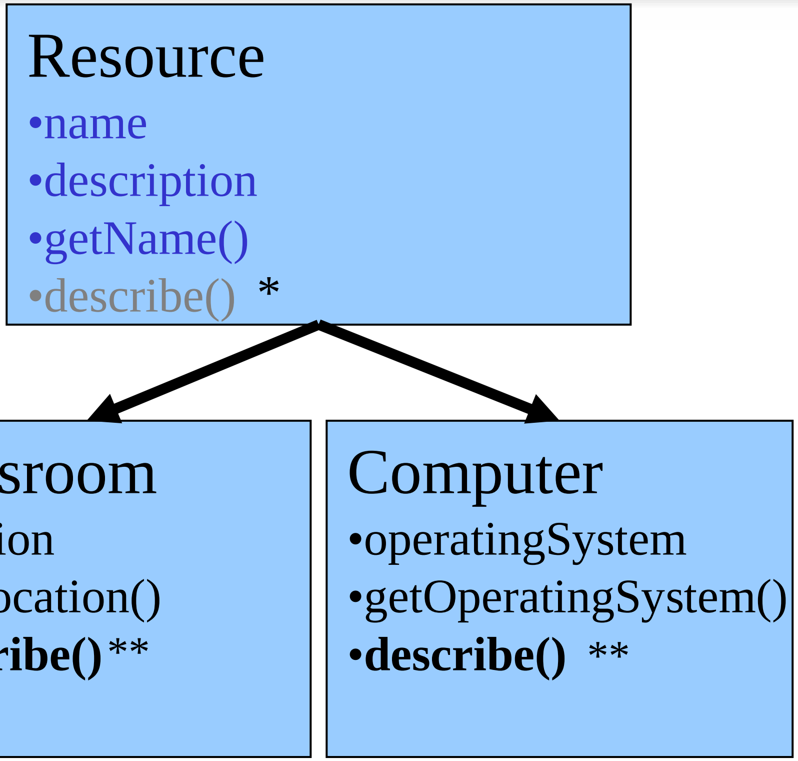
How are they used?

*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

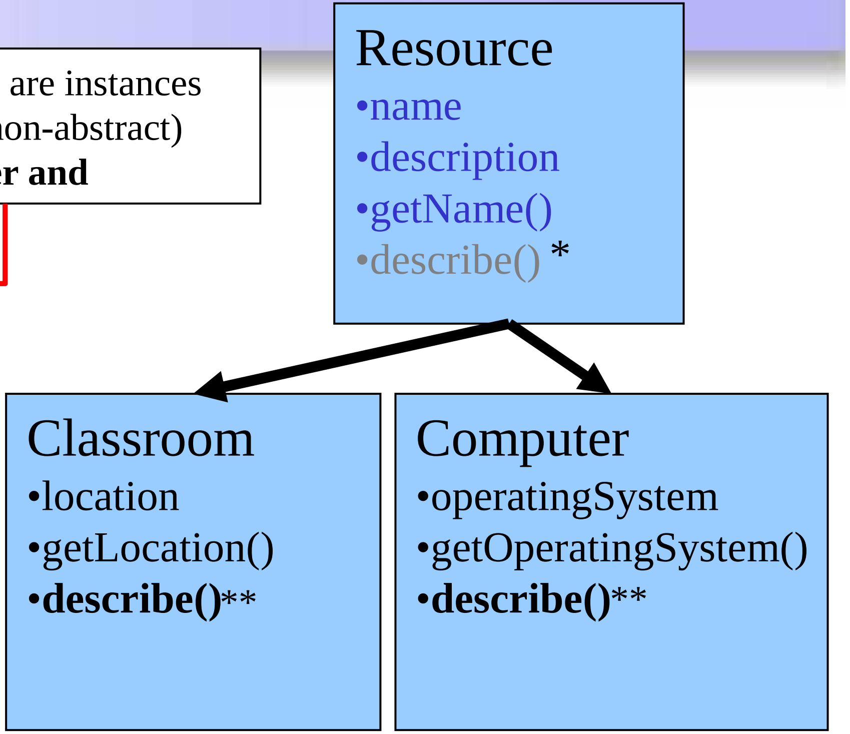
Polymorphism

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();
  }

 }
}
```

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**



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{…}

* Grey color means having no code

** Bold means having code

Interfaces

What is an interface?

- Interfaces take the abstract class concept one step further.
  - ***All** 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

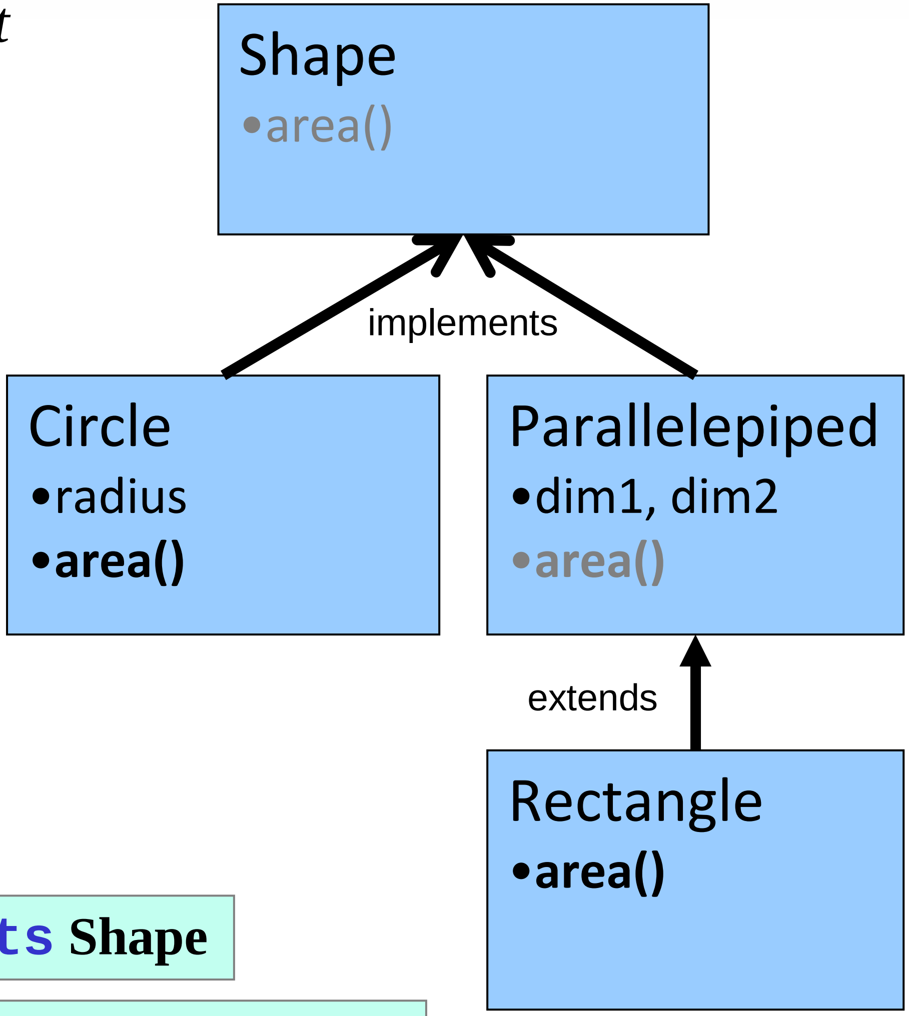
What is an interface?

- 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 behaviors 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 braces**!! 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 braces**!! 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

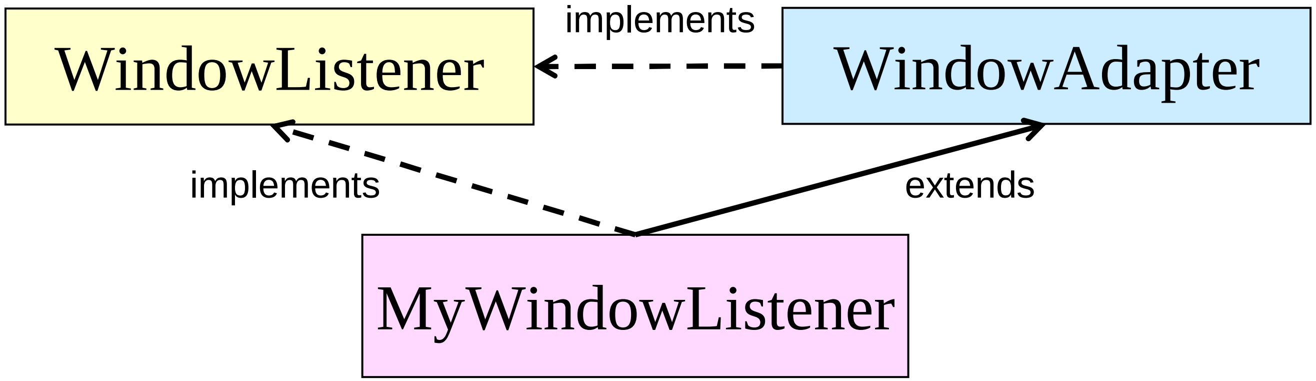
Use. Extending interfaces with inheritance

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



WindowListener

implements

WindowAdapter

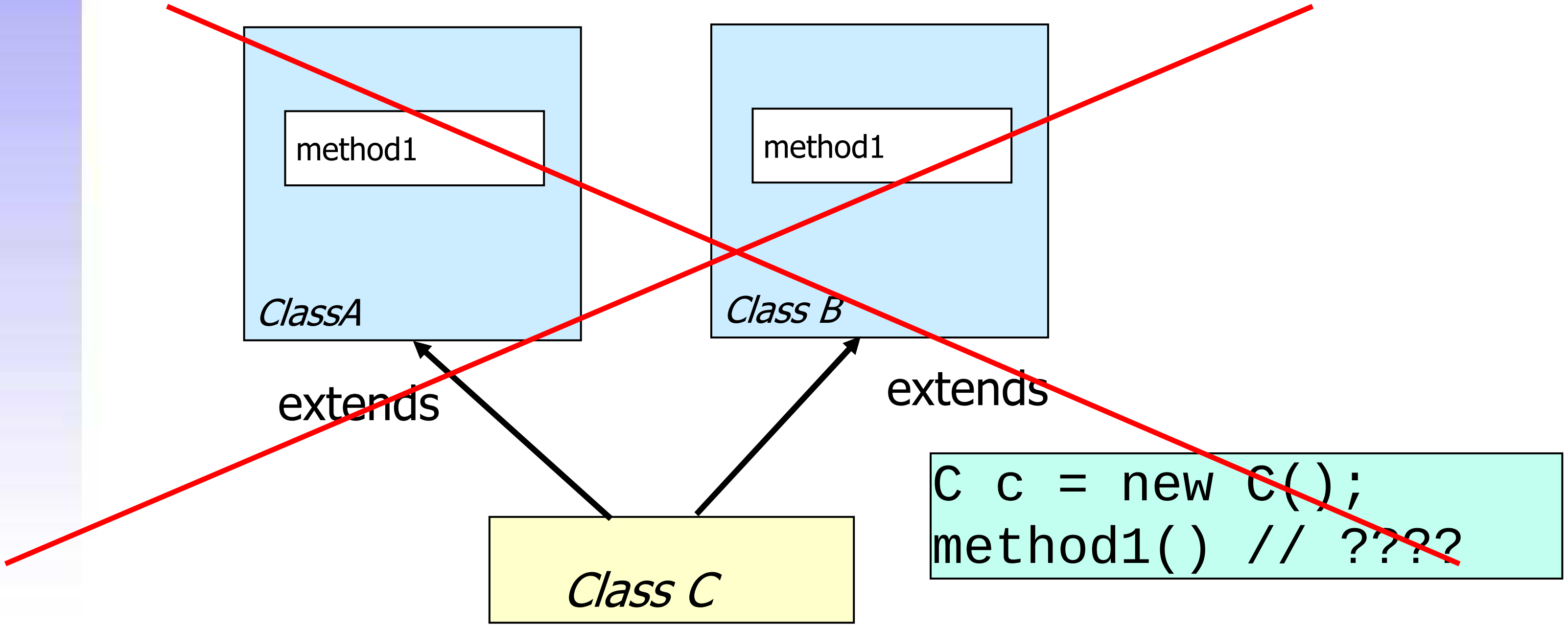
implements

extends

MyWindowListener

Interfaces

Purpose. Multiple inheritance

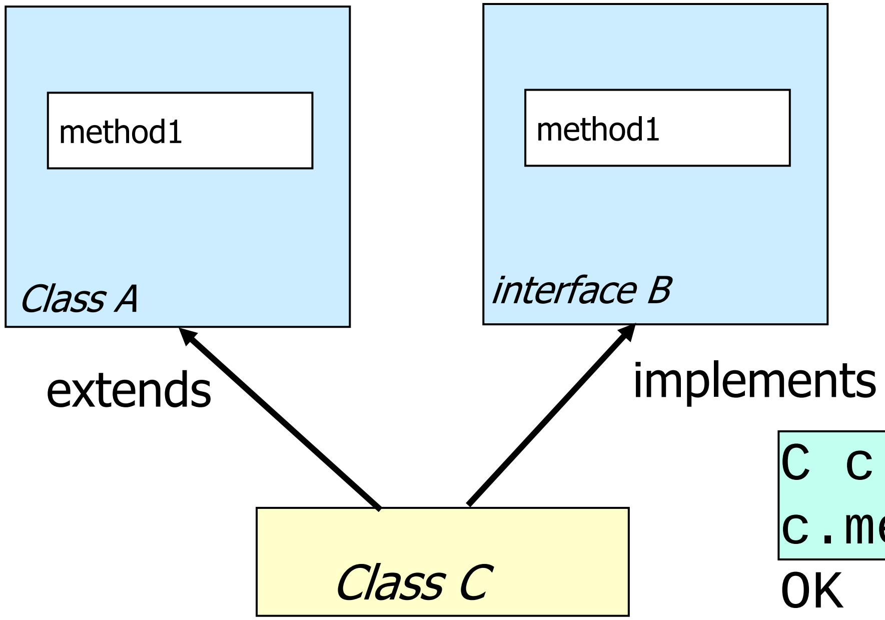


```
C c = new C();
method1() // ????
```

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

Interfaces

Purpose. Multiple inheritance



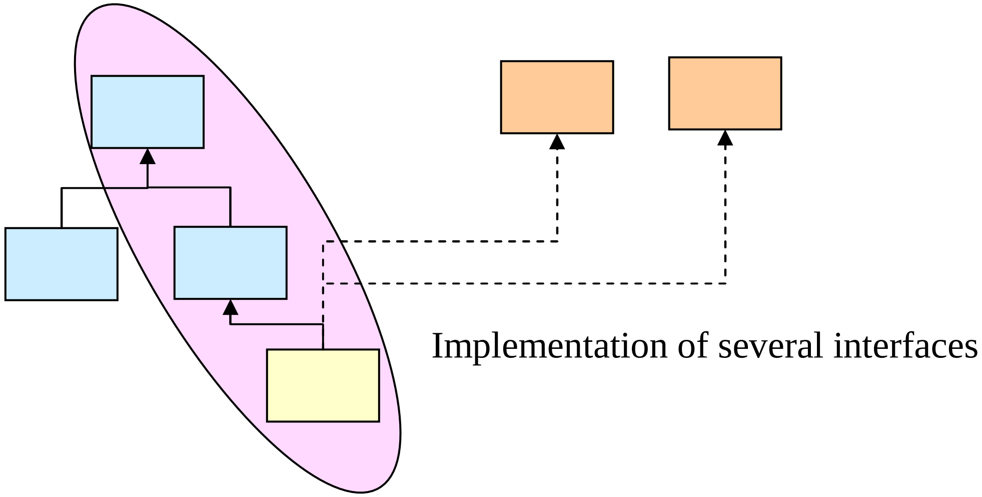
```
C c = new C();
c.method1() //
OK
```

- 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

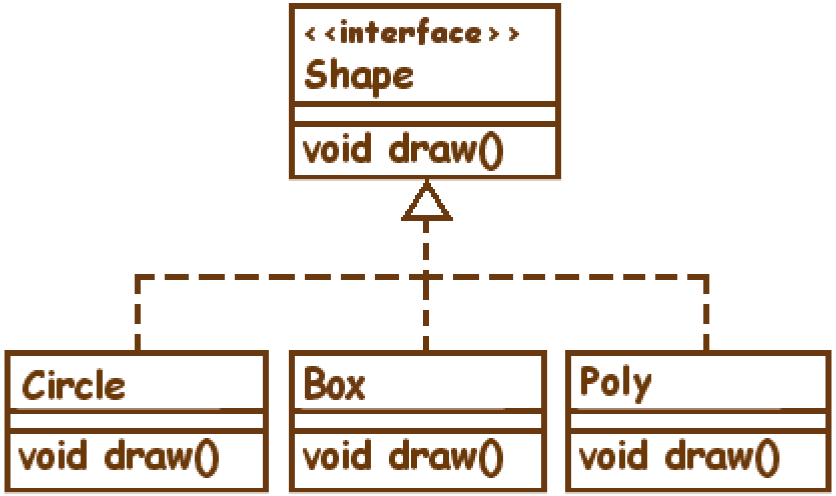


Interfaces

Purpose. Polymorphism

- Polymorphism: “one interface, multiple methods”
- Interfaces support dynamic resolution of methods during execution time (dynamic binding)
- What difference is there 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 ) ,
```

Exercise: JavaRanch

```
        // 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*** (concrete)
  - ***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* (concrete 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

M. Carmen Fernández Panadero

Raquel M. Crespo García

<mcfp, rcrespo@it.uc3m.es>

Packages

- A ***package*** groups ***classes*** and ***interfaces***
- The hierarchies in a package correspond to the hierarchies of folders
- 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) and /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 sentece 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	
others	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 leave 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. Applied to inner classes	It is the same for all of the class objects. Use: ClassName.method ();	It is the same for all of the class objects.

(*) http://www.coderanch.com/t/527185/java/java/Understanding-Modifiers-classes

Exceptions

M. Carmen Fernández Panadero

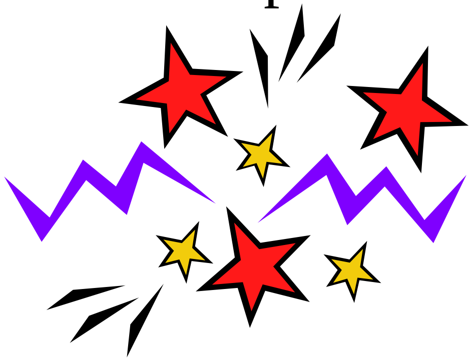
Raquel M. Crespo García

<mcfp, rcrespo@it.uc3m.es>

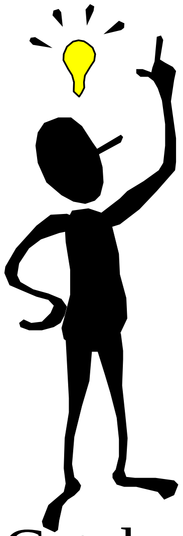
Exceptions

- What they are
- Purpose
- Type
- Use

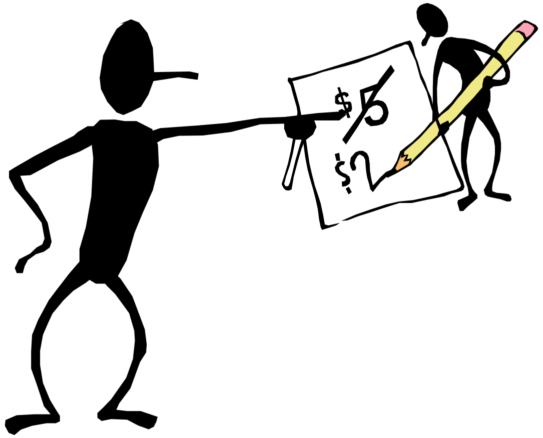
Exception



Ignore → End



Catch



Throw

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:
  - **Search** for a piece 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 compiling-time
    - E.g.: NullPointerException, ArithmeticException, NumberFormatException, IndexOutOfBoundException, etc.)
  - Exceptions checked during compiling time
    - E.g.: Input/output exceptions (`IOException, FileNotFoundException, EOFException`)
    - **User-defined** (`MyException`)
- During compiling 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 senteces 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