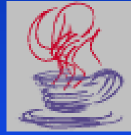
A. Bechini
Course: Java

Special thanks to A. Vecchio for the previous version of these slides

Introducing Java

Abstract Classes, Interfaces and Enhancement of Polymorphism



Introducing Java

Abstract Classes

Designing a hierarchy, it's current (and proper) practice placing in the super-classes all the common methods and data structures required by subclasses.

Sometimes a super-class is aimed only at acting as a "common model" for the sub-classes, and no reason exists to actually instantiate it. In this case, it may be declared as "abstract."

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

Abstract classes CANNOT be instantiated

Introducing Java

Abstract Classes

You create an abstract class when you want to manipulate a set of classes through this common interface

Introducing Java

Abstract Classes

- Abstract classes can contain whatever an "ordinary" class can: instance and class variables, instance and class methods, with whatever modifiers
- Moreover, abstract classes can contain ***abstract methods***
- An abstract method is given the signature only. An abstract method is not equipped with a body, i.e. no implementation is given for it.
- The implementation of the body of an abstract method is provided in sub-classes of the abstract class.

Special thanks to A. Vecchio for the previous version of these slides

Introducing Java

Abstract classes

A method can be declared abstract iff it is contained in an abstract class.

Abstract methods "roughly" describe within a super-class behaviors that are exhibited by sub-classes.
Each subclass is in charge of providing the correct specific implementation for such behaviors

```
public abstract class Shape {
  protected double x,y;
  public void whatPlace() {
    System.out.println("My position: "+x+","+y);
  }
  public abstract double area();
}
```

Sub-classes of Shape inherit the concrete method whatPlace() and variables x and y

They MUST implement the method area() as well

Introducing Java

Abstract Classes

```
class Circle extends Shape {
  protected double radius;
  public Circle(double r){
    radius=r;
  }
  public double area() {
    return(radius*radius*3.1415);
  }
}
```

```
class Square extends Shape{
  protected double edge;
  public Square(double e){
    edge=l;
  }
  public double area() {
    return(edge*egde);
  }
}
```

Classes Circle and Square are sub-classes of Shape;
Thus, they inherit variables x and y, and method whatPlace() as well.
Moreover, they implement method area() declared as abstract in class Shape

A. Bechini
Course: Java

Special thanks to A. Vecchio for the previous version of these slides

Introducing Java

Abstract Classes

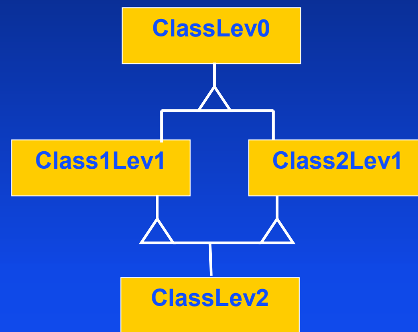
```
...
Shape[] shp=new Shape[2];
shp[0]=new Circle(1.0);
shp[1]=new Square(2.0);

for(int i=0; i< shp.length; i++) {
  System.out.println("Area: "
          + shp[i].area());
  shp[i].whatPlace();
}
...
```

Here, a two-element array is created, and it is assigned two objects: one of type Circle and the other of type Square

The methods whatPlace() and area() are invoked on the array elements: the former is inherited from class Shape, where it has been implemented; the latter has been declared abstract

Introducing Java

The Diamond Problem



Special thanks to A. Vecchio for the previous version of these slides

Introducing Java

Interfaces

In Java (differently by other OO languages such as C++) multiple inheritance among classes is not allowed. Whenever multiple inheritance is present, a single class may "extend" multiple super-classes.

In Java a single class can extend just one single super-class.

This feature makes the language easier to learn and to implement.

One of the drawbacks of this approach is the following: it's not possible to specify that classes on separate sub-trees share some behaviors.

E.g., the two classes Car and Factory may share the behavior of PollutingObject characterized by the methods emissionOfPollutingGases (), etc.

Introducing Java

Interfaces

Java provides a solution to this issue by means of the introduction of interfaces

An interface is a collection of method definitions with no implementation; no instance variable is present within an interface

An interface can be associated to whatever class, in order to provide it with a behavior which is possibly not inherited by a super-class

Interfaces are not integral part of the ordinary class hierarchy.

Special thanks to A. Vecchio for the previous version of these slides

Introducing Java

Interfaces

An interface definition is carried out according to the same rules used for classes: it must be placed in a file named as the interface itself, and with the extension .java; once it gets compiled, it is contained in a .class file.

The keyword interface is used to create a new interface

```
package geometry;
public interface Measurable {
  public static final double PI=3.1415;
  public abstract double area();
  double perimeter();
}
```

Interfaces may be placed in a package

As for classes, interfaces must have a "public" or "package" protection level.

Introducing Java

Interfaces

```
package geometry;
public interface Measurable {
  public static final double PI=3.1415;
  public abstract double area();
  double perimeter();
}
```

Methods can be declared as public and abstract

They cannot be neither protected nor private

If no modifier is provided (as in perimeter()), a member assumes the same visibility as the class (in this case, public)

The defined variable must be public, static, final

In case mo modifier is specified, the same rules as for methods apply.

A. Bechini
Course: Java

Special thanks to A. Vecchio for the previous version of these slides

Introducing Java

Interfaces

An interface can be defined to be an extension of another by using the keyword extends:

```
public interface InterfaceB extends InterfaceB {
  …
```

Interface hierarchy, differently from the class hierarchy, has no root (neither explicit nor implicit; no counterpart of the java.lang.Object exists).

Multiple inheritance is used in the interface hierarchy

```
public interface InterfaceX extends InterfaceA, InterfaceB {
  …
```

InterfaceX contains all the method/variable definitions and constants that are present both in InterfaceA and in InterfaceB

Introducing Java

Interfaces: the keyword *implements*

Because of the use of the keyword implements, a class is in charge of implementing all the methods defined in the interface.

```
package geometry;
public interface Measurable {
  public static final double PI=3.1415;
  public abstract double area();
  double perimeter();
}
```

```
class Square implements Measurable{
  protected double edge;
  public Quadrato(double e){
    edge=e;
  }
  public double area() {
    return(edge*edge);
  }
  public double perimeter() {
    return(edge*4);
  }
}
```

Sub-classes of a class that implements a given interface, inherit the methods in the implemented interface.

Special thanks to A. Vecchio for the previous version of these slides

Introducing Java

Interfaces

A class can implement more than one interface.

class Apple extends Fruit implements Peelable, Eatable, Sellable, …

- If two implemented interfaces have the same method with the same signature, a single implementation for both has to be specified.
- If the two methods have the same name and different signatures, both of them have to be implemented.
- If the two methods have the same name, the same signature but different types are returned, an error occurs.

Introducing Java

Interfaces

It's possible to declare a variable whose type is an interface. Such a reference variable can be assigned object of classes that implement such an interface.

Measurable mea=new Square(2.0)

As mea is an object of type Measurable, it is possible to invoke over it the two methods perimeter() and area()

Interfaces can be used to group a number of constants to be imported into several different classes

```
public interface Constants {
  double PI=3.1415, sqrtTwo=1.4142, …
}
```

Special thanks to A. Vecchio for the previous version of these slides

Introducing Java

Interfaces: More Polymorphism is Added

The fact that it's possible to declare a variable whose type is an interface, gives us much more flexibility in using polymorphic behaviors.

A certain method a() in an interface X can be invoked over all the objects whose classes implements X, REGARDLESS of the placement of the classes in the class hierarchy.

In this context, it is common to deal with arrays (or more complex data structures of this kind) with elements of type "interface"

Introducing Java

Interfaces: What about Cast?

- What about cast among interfaces?
- What about cast among interfaces and classes?
- What about cast among interfaces and abstract classes

Let's make some experiments!

Special thanks to A. Vecchio for the previous version of these slides

Introducing Java

Exercise

- Create the class Rodent and the two classes Mouse and Beaver, making possibly used of an abstract class
- Create class Cage, containing three object of type Rodent
- Cage must implement the interface java.util.Enumeration (present in the core APIs) which is made of two methods:
  - Object nextElement() Returns an element
  - boolean hasMoreElements() Returns true if there are more elements to be enumerated

Each element can be returned just once.

Print the attributes of the three Rodent objects within an instance of Cage