Inheritance, abstract classes and interfaces

Inheritance

Reusability

- If you get a compiled `A.class` you can use it:
  ```java
  A a = new A();
  int y = a.compute(x);
  ```
- But, sometimes, you would like to modify slightly the behavior of the instances of `A`
- It is not possible if you have not got the source code `A.java`

Inheritance

- With inheritance you can create an object whose behavior is slightly different from an instance of the `A` class:
  - an instance of a subclass of `A`, with
    - new methods
    - modified `A` methods

Inheritance in Java

- Each class has one and only one superclass: no multiple inheritance
- The class inherits the members of its superclass
- It is possible to:
  - add new methods
  - modify some methods of its superclass
- But it is not possible to remove methods of the superclass

Inheritance in Java

- The keyword `extends` indicates the superclass of a class:
  ```java
  class Subclass extends Superclass {
    ...
  }
  ```
- By default, a class inherits from the class `Object`
Example – superclass

```java
public class Rectangle {
    private int x, y; // upper left point
    private int width, height;
    // Constructors
    // Methods getX(), setX(int),...
    // getWidth(), getHeight(),...
    // contains(Point), intersects(Rectangle),...
    // translate(int, int), toString()...
    public void draw(Graphics g) {
        g.drawRect(x, y, width, height);
    }
}
```

Example – subclass

```java
public class ColoredRectangle extends Rectangle {
    private Color color; // new variable
    // Constructors
    // New methods
    public getColor() { return this.color; }
    public setColor(Color c) { this.color = c; }
    // Overrided methods
    public void draw(Graphics g) {
        g.setColor(color);
        g.fillRect(getX(), getY(),
                  getWidth(), getHeight());
    }
}
```

Examples

- Superclass `Vehicle` and subclasses `Bicycle`, `Car`, and `Truck`
- `Bicycle` inherits from `Vehicle` and has several subclasses: `MountainBike`, `RacingBike`,...
- `Employee` inherits from `Person`
- `Image` is the superclass of `GifImage` and `JpegImage`

Bad use of inheritance

- Use inheritance to represent the relation *is-a* between classes
- Don’t use inheritance just to reuse code
- For example, don’t do that: `ParallelepipedicRectangle` inherits from `Rectangle` and adds a depth

Overriding and Overloading

- Overriding is not the same as overloading:
  - you override a method when a method in a subclass has exactly the same signature as an inherited method
  - you overload a method when another method has the same name but a different signature
The `super` keyword

- In a method, the `super` keyword can be used to manipulate a method of the superclass, overridden (or not) in the current class:

```java
int m(int i) {
    return 500 + super.m(i);
}
```

Constraints about redefinitions (1)

- In Java, the return type must be exactly the same as the return type of the overridden method.

Constraints about redefinitions (2)

- Visibility keywords: the overriding method cannot be less accessible than the overridden method (for example it is not possible to declare `private` a method that overrides a public method).
- Exceptions: the overriding method cannot declare throwing more exceptions than the overridden method declares (subclasses are allowed).

Constructor Chaining

First instruction of a constructor

- The first instruction of a constructor is a call to:
  - a constructor of the superclass:
    ```java
    super(...)
    ```
  - another constructor of the current class:
    ```java
    this(...)
    ```
- Otherwise, the compiler adds an implicit call to the constructor without parameter of the superclass.
- `=>` the first instruction, explicit or implicit, is a call to a constructor of the mother class.

Superclass

```java
public class Rectangle {
    private int x, y, weight, height;

    public Rectangle(int x, int y, int weight, int height) {
        this.x = x;
        this.y = y;
        this.weight = weight;
        this.height = height;
    }
    . . .
}
```
Constructors of the subclass

```java
public class ColoredRectangle extends Rectangle {
    private Color couleur;
    public ColoredRectangle(int x, int y,
                           int weight, int height, Color color) {
        super(x, y, weight, height);
        this.color = color;
    }
    public ColoredRectangle(int x, int y,
                           int weight, int height) {
        this(x, y, weight, height, Color.black);
    }
    ...
}
```

Implicit call of the constructor without parameter of the superclass

- If the first instruction of a constructor is neither `super()` nor `this()`, the compiler adds an implicit call to the constructor without parameters of the superclass

⇒ A constructor of the superclass is always executed before the other instructions of the constructor

The really first instruction executed by a constructor

- But the first instruction executed by the constructor of the superclass is the call to a constructor of the superclass of the superclass
- So, the really first instruction executed by a constructor is the constructor of the class `Object`

Protected access

- If a class declares `protected` one of its members, this member will be accessible by its subclasses
- `protected` gives also access to the member to all the classes of the same package
- Precision: if the subclass is not in the same package, instances of the subclass will not have access to the protected members for the instances of the superclass

What is inherited?

- If B extends A, it inherits of all the members of A (but not the constructors)
- But, B cannot access private members of the A class; these members will be implicitly used by B, but B will not be able to name or to manipulate them

Access to protected members

- If a class declares `protected` one of its members (or constructor), this member will be accessible by its subclasses
Example

class A {
    ...
    protected int m() { ... }
}

class B extends A {
    ...
    public int m2() {
        int i = m(); // always allowed
        A a = new A();
        i += a.m(); // not always allowed
        ...
    }
}  

when is it allowed?

Object class

- java.lang.Object is the root of the inheritance hierarchy: the superclass of all the classes
- Object class has no instance or class variable
- All the methods of the Object class are inherited by all the classes
- The more often used methods of Object are:
  * toString()
  * equals()

toString method

- public String toString() returns a description of the instance
- Very useful for debugging
- Must be overridden by almost all the classes

equals method

- public boolean equals(Object object) returns true if object is equal to this, the current instance, according to an equality relation
- Example: 2 instances of Fraction are equal if f1.num * f2.den = f1.den * f2.num
- In the Object class,
  obj1.equals(obj2) ⇔ obj1 == obj2

equals and hashcode methods

- Specification of Java: if 2 objects are equal, they must have the same hashcode
- So, if equals is overridden, hashcode must be overridden too
- It is not always easy to override the hashcode method
Example of `toString` and `equals`

```java
class Fraction {
    private int num, den;
    . . .
    public String toString() {
        return num + "/" + den;
    }
    public boolean equals(Object o) {
        if (! (o instanceof Fraction))
            return false;
        return num * ((Fraction)o).den
            == den * ((Fraction)o).num;
    }
}
```

\[ \frac{a}{b} = \frac{c}{d} \iff a \cdot d = b \cdot c \]

Example of `hashCode`

```java
class Fraction {
    public int hashCode() {
        /* Reduce the fraction */
        Fraction f = reduce();
        return (17 + f.num * 37) * 37 + f.den;
    }
}
```

`getClass()` method

- `public Class getClass()` returns the class of the instance
- `java.lang.Class` contains a `getName()` method which returns the qualified name of the class

Polymorphism

A question...

- Suppose that `B` extends `A` and that the method `m()` of `A` is overridden in `B`
- Which method `m()` will be executed in the following code, the one of `A` or of `B`?

```java
A a = new B(5);
a.m();
```

The method that is called depends only on the real type (`B`) of `a` (not the declared type, here `A`). So, the method from `B` will be executed

Polymorphism

- It is the fact that the same call to a method in the code, can be executed by different methods
- For instance,

```java
A a = x.f();
a.m();
f can return an instance of `A`
method `m()` of `A` or of any subclass of `A`
```

```
a is an object of class `B` but declared as an `A`
```
Example of polymorphism

```java
public class Figure {
    public void draw(Graphics g) {}
}

public class Rectangle extends Figure {
    public void draw(Graphics g) {
        // ... Rectangle.java
    }
}

public class Circle extends Figure {
    public void draw(Graphics g) {
        // ... Circle.java
    }
}
```

Example of polymorphism (cont.)

```java
public class Drawing {
    private Figure[] figures;

    public void draw(Graphics g) {
        for (int i = 0; i < figures.length; i++) {
            figures[i].draw(g);
        }
    }

    public static void main(String[] args) {
        Drawing drawing = new Drawing(30);
        // ... Figure.java
        drawing.add(new Circle(center, radius));
        drawing.add(new Rectangle(p2, width, height));
        drawing.draw(g);
    }
}
```

Static type checking and polymorphism

- When it encounters `a.m()` the compiler must be sure that the class of the object `a` contains a method `m()`.
- For instance, because of `figures[i].draw(g)`, it is necessary that the class `Figure` contains the method `draw(Graphics)`.
- Else you will get an compilation error.

Advantage of polymorphism

- A good use of polymorphism can avoid a program with a lot of tests (if, switch/case...):
  ```java
  for (int i = 0; i < figures.length; i++) {
      if (figures[i] instanceof Rectangle) {
          // Draw a rectangle
      } else if (figures[i] instanceof Circle) {
          // Draw a circle
      } else if (figures[i] instanceof Rhomb) {
          // Draw a rhomb
      }
  }
  ```

- With polymorphism you can easily extend an application:
  ```java
  public void draw(Graphics g) {
      for (int i = 0; i < figures.length; i++) {
          figures[i].draw(g);
      }
  }
  ```

- Without polymorphism:
  ```java
  else if (figures[i] instanceof Rhomb) {
      // Draw a rhomb
  }
  ```
Conclusion

- Polymorphism is a fundamental issue in Object Oriented Programming and the basis for an efficient use of inheritance.
- Polymorphism is achieved by the late binding mechanism: the method that will be called is determined at run time, not at compilation time.

Cast

Cast: conversion between classes

- To cast is to enforce the compiler to see an object with a certain type.
- In Java, the only allowed casts between objects are between subclasses and superclasses.

UpCast: subclasses → superclass

- An object is considered with the type of an ancestor class instead of its real type.
- It is always possible to perform an upcast (relation is-a of inheritance).
- So, upcast can be implicit.

Example of upcast

```java
public class Engineer extends Employee {
    . . .
}
public class Project {
    public void add(Employee employee) {
        . . .
    }
}
// somewhere else...
Engineer i1 = new Engineer(2563, "John");
project.add(i1);
```

DownCast: superclass → subclass

- An object is considered of a type of a subclass of its declaration class.
- A downcast is not always possible; at runtime, the type of the object will be checked.
- Downcast is used to call a method of the subclass that does not exist in the superclass.
- A downcast must always be explicit.
Example of downcast

```java
// Figures are added in an ArrayList
ArrayList figures = new ArrayList();
figures.add(new Circle(center, radius));
figures.add(new Rectangle(p1, p2));
...
// get(i) return an Object; a cast is
// necessary to call the draw method
((Figure)figures.get(i)).draw(g);
```

Keyword `final`

- `final` class: cannot have subclasses
- `final` method: cannot be overridden
- `final` variable: cannot be modified after initialization
- `final` parameter: cannot be modified in the method body

Abstract classes and methods

- A method is abstract (modifier `abstract`) when you only give its signature and its return type, not its implementation
- A class must be declared `abstract` if it contains an abstract method
- Abstract classes are useful for classifying classes or for code factorization

Example of an abstract class

```java
public abstract class Figure {
    public abstract void draw(Graphics g);
    //...
}
```

Interfaces
**Interfaces**

- An interface is a 100% abstract class, without any variables.
- An interface can contain constants (static final).
- Methods and constants are necessarily (and implicitly) public.
- An interface can inherit (keyword extends) from several interfaces.

**Example of interface**

```java
public interface Figure {
    public void draw(Graphics g);
}
```

**Examples of interfaces from the API**

```java
public interface Comparable {
    int compareTo(Object o);
}
```

```java
public interface MouseListener extends EventListener {
    void mouseClicked(MouseEvent e);
    void mouseEntered(MouseEvent e);
    void mouseExited(MouseEvent e);
    void mousePressed(MouseEvent e);
    void mouseReleased(MouseEvent e);
}
```

**Implementation of an interface**

- A class can implement (keyword implements) one or several interfaces.
- If C is a class that implements an interface, it must implement all the methods of the interface.
- Otherwise C must be declared abstract.

**Example of an implementation**

```java
public class City implements Comparable {
    private String name;
    private int nbInhabitants;
    
    public int compareTo(Object object) {
        return nbInhabitants - (City)object.nbInhabitants;
    }
}
```

**Interfaces are types**

- An interface can be used exactly everywhere you can use a class name.
- An interface defines the services proposed by the class that implements it; it describes a behavior.
Polymorphism and interfaces

- For polymorphism, an interface plays the role of a superclass: the example of polymorphism works very well if the abstract class Figure is an interface.

Using interfaces

- You can apply polymorphism with classes from different inheritance hierarchies if they implement a same interface.
- The advantage of interface is that a class can implement several interfaces.

Cast and interfaces

- We can cast (upcast and downcast) between a class and an interface it implements (and upcast between an interface and Object):

```java
// upcast City -> Comparable
Comparable c1 = new City("Cannes", 200000);
Comparable c2 = new City("Nice", 500000);
...
if (c1.compareTo(c2)) // upcast Comparable -> Object
    System.out.println((Comparable)c2.getNbInhabitants());
```

Reuse

How class C2 can reuse code of a class C1

- Inherits from C1
- Composition: an instance variable of the C2 is of type C1
- Delegation: C2 delegates the execution of one of its methods to an instance of C1; this instance can be
  - an instance variable of C2
  - created by the method of C2 to do the work

Drawbacks with inheritance

- Static: if an object must play different roles at different moments, it is not possible to use inheritance.
  For instance, if a person can be a pilot or a client of an airplane company, 3 classes Person, with 2 subclasses Client and Pilot are no use.
- Poor encapsulation for the superclass: often difficult to change the superclass, especially if protected variables have been used.
- Not possible to inherit from a final class (String for instance).
Advantages of inheritance

- Easy to use: basic mechanism of the language
- Allow polymorphism (but also possible with interface)
- Easy to understand if it represents a is-a relation
- Flexible because you can modify inherited behavior (by overriding methods)

Conclusion

- To reuse classes,
  - Use inheritance to represent static is-a relations
  - Use delegation and composition in the other cases