Virtual Functions and Late Binding

Advanced Programming

Content

- Introduction
- Notion of virtual function
- Virtual functions in Java
- Simple exercises

Introduction

- OOP: behavior of objects determined at the **latest possible stage**, if possible at run time
- In conventional programming languages, this is usually possible only through **switch or if statements**
- The so-called **virtual functions** are an elegant and flexible solution to this problem
- Virtual functions implement a mechanism called **late binding**

Binding

- Association of function names and function bodies is called **binding**

binding *cannot be performed when a function or a method is defined, but when a function or a method is effectively* ***called***

Virtual Functions

- Virtual functions: functions for which the **association** between name and body is performed **at run time** to identify of the **most suited body** to use
  - The control on the presence of the body and of its conformance to the specifications is not performed at compile time but at execution time, thus reducing the checks available while the code is being developed

Overriding (1/2)

- A method of a (base) class can be “redefined” in derived classes
- Together with a method we have:
  - method name
  - parameters number and type(s)
  - return value type
- The name of the method and its parameter number and type(s) are called the signature of the method

Overriding (2/2)

- You can also override a method with the same signature that returns a **subclass** of the object returned by the original method.
- This is called **covariant return type**

Overriding and Overloading

- Overloading: the same name with a **different signature** in the base class and in the derived class
  - The decision of **which method to execute** is based on the **types of the actual parameters** – not on the type of the return!
  - **Same signature with different return type** (except covariant return type) returns a compiler error, this is because that the compiler cannot choose: choices have the same cost

Overriding and Overloading

- The access modifier for the **overriding method can allow more** but not less access than the overridden method:
  - For example, a protected method in the base class can be made public but not private method in the derived class
  - The derived class is at least as accessible as the base class

Note

- Remember that overriding pertains only the signature and the return value; it does not say anything on the modifiers or the type of methods (static, non-static)
- Therefore we need to reason on them separately

Questions and answers

- Can I override a static method with a static method?
  - **No**, it does not turn into a compiler error but in fact it does not overridden. The derived method **is hidden** by the base method. **A static method can hide only static methods**
- Can I override a **static** method with a **non-static** one?
  - **No** it is a compiler error as a child class is-a parent class
- Can I override a **non-static** method with a **static** method?
  - **No** it is a compiler error as a child class is-a parent class

Behaviour of virtual functions (1/5)

- Virtual functions are a techniques of object oriented programming languages that takes advantage of overriding

Behaviour of virtual functions (2/5)

- When a virtual function **f** is invoked for an object **o** of class D derived from B there are three (basic) possibilities:
  - **f is defined in D only**: compiler error

```
class B { }
class D extends B {
  void f() { }
}
B o = new D();
o.f(); // COMPILER ERROR!!
```

Behaviour of virtual functions (3/5)

- **f is defined in B only**: B.f

```
class B {
  void f() { }
}
class D extends B {
}
…
B o = new D();
o.f(); // Call to B.f()
```

Behaviour of virtual functions (4/5)

- **f is overridden in D**: D.f()

```
class B {
  void f() { <something> }
}
class D extends B {
    void f() { <something else> }
}

B o = new D();
o.f(); // Call to D.f()
```

Behaviour of virtual functions (5/5)

- To access the function f of B:

```
class B {
  void f() {...}
    void fOfB() { f(); }
}
class D extends B {
    void f() {...}
    void fOfB() { super.f(); }
}

B o = new D();
o.f(); // Call to D.f()
o.fOfB() ; // Call to B.f() using super
```

Example of virtual functions (Pseudocode)

```
class TransmissionChannel {
    public void transmit(String s) { ... }
}

class WirelessTransmissionChannel extends TransmissionChannel {
    public void transmit(String s) { ... }
}

class SmokeSignalsTransmissionChannel extends TransmissionChannel {
    public void transmit(String s) { ... }
}

class Environment {
    static public TransmissionChannel getConnection() {
        <determine the most suitable transmission channel>
        <establish the connection>
     return <the established connection>;
    }
}

... main() {
    TransmissionChannel tc;
    tc = Environment.getConnection();
    tc.transmit("Hello World ");
}
```

Is a “A a = new B()” with B determined at run-time; here we must include some sort of run-time exception!

The transmit protocol is decided at run-time

Declaration of virtual functions

- There is no need to declare a method to be virtual in Java
- Every (non-static) method is *always* virtual in Java
- Static methods are non-virtual (see previous example)
- It is possible to prevent the overriding of methods by declaring it “final”
- It is possible to prevent the extension of a class by declaring it “final”

Exercise

```
public class CoffeeMaker {
    ...
    public static void whoAreYou() {
        System.out.println("I am a coffee maker!");
    }
    public void whoAreYouReally() {
        System.out.println("I am a coffee maker!");
    }
     ...
}
```

Exercise

```
public class EspressoMaker extends CoffeeMaker {
    public static void whoAreYou() {
        System.out.println("I am an espresso maker!");
    }
    public void whoAreYouReally() {
         System.out.println("I am an espresso maker!");
    }
    public static void main(String[] args) {
        CoffeeMaker cm = new CoffeeMaker();
        CoffeeMaker eCM = new EspressoMaker();
        EspressoMaker em = new EspressoMaker();
        cm.whoAreYou();
        cm.whoAreYouReally();
        eCM.whoAreYou();
        eCM.whoAreYouReally();
        em.whoAreYou();
        em.whoAreYouReally();
   }
}
```

Exercise: Sample Output

- `I am a coffee maker!`
- `I am a coffee maker!`
- `I am a coffee maker!`
- `I am an espresso maker!`
- `I am an espresso maker!`
- `I am an espresso maker!`

Be careful (1/2)!

```
public class AmericanCoffeeMaker extends CoffeeMaker {
 public static void whoAreYou() {
    System.out.println("I am an American coffee maker!");
 }
 public void whoAreYouReally(int i) {
   System.out.println("I am American coffee maker code"+ i);
 }
```

Be careful (2/2)!

```
    public static void main(String[] args) {
        CoffeeMaker cm = new CoffeeMaker();
        CoffeeMaker aCM = new AmericanCoffeeMaker();
        AmericanCoffeeMaker am = new AmericanCoffeeMaker();
        cm.whoAreYou();
        cm.whoAreYouReally();
        aCM.whoAreYou();
        aCM.whoAreYouReally();
        aCM.whoAreYouReally(3);
        am.whoAreYou();
        am.whoAreYouReally();
        am.whoAreYouReally(3);
        AmericanCoffeeMaker am2;
        am2 = (AmericanCoffeeMaker) aCM; // Attention
        // Some compilers may not like this line
        // And it is dangerous anyway, for example if the
       //coffee maker is not an American coffee maker
        am2.whoAreYou();
        am2.whoAreYouReally();
        am2.whoAreYouReally(3);
   }
}
```

Exercise: Sample output & wrong lines

```
I am a coffee maker!
I am a coffee maker!
I am a coffee maker!
I am a coffee maker!
I am an American coffee maker!
I am a coffee maker!
I am American maker code 3
I am an American coffee maker!
I am a coffee maker!
I am American maker code 3
```

✗ `aCM.whoAreYouReally(3);` error because there is no such method in CoffeeMaker

Remember!

- We cannot access to local method of a derived class from a reference variable of the parent class if the local method does not overrides a base method!

Final methods do not allow overriding

```
  public class AmericanCoffeeMaker extends CoffeeMaker {
    ...
    public final static void whoAreYou() {
      System.out.println("I am an American coffee maker!");
    }
    public void whoAreYouReally(int i) {
     System.out.println("I am American coffee maker code"+
                           i);
    }
   ...
 }
 public class StarbucksCoffeeMaker extends
                                  AmericanCoffeeMaker {
   // public final static void whoAreYou() { ERROR!!!
   //    System.out.println("I am a Starbucks maker!");
   // }
    public void whoAreYouReally(int i) { // This is OK!
     System.out.println("I am Starbucks coffee maker code"+
                           i);
    }
 }
```