COSC 1020

Yves Lespérance

Lecture Notes

Week 9 — Inheritance & Polymorphism

Recommended Readings:
Horstmann: Ch. 9 Sec. 1 to 3 & Ch. 11
Lewis & Loftus: Ch. 7 Sec. 0 to 4

Inheritance

Often, we need to define a class that is similar to an existing class; it just adds a few new attributes or methods, or implements existing methods differently.

Instead of having to define the new class from scratch, OOP supports declaring the new class as a *subclass* or *specialization* of the old one.

Then the subclass *inherits* all the attributes and methods of the original class, can add new ones, as well as *override* the implementation of existing methods.

This is called *inheritance*.

We say that the subclass *is a* the superclass.

Code reuse is important in software engineering. This saves time and you don't have to keep track of all the places where the same code appears so you keep them "in sync".

Can represent is-a/subclass relationships in UML class diagrams.

E.g. `RewardCard` *is a* `CreditCard`,
`GlobalCredit` *has a* collection of `CreditCard`.

E.g. a hierarchy of classes:
- `Undergrad` *is a* `Student`,
- `Grad` *is a* `Student`,
- `MSC` *is a* `Grad`,
- `PHD` *is a* `Grad`.

The class `Undergrad` is a subclass or specialization of `Student`. Inversely, we say that `Student` is a *superclass* or *generalization* of `Undergrad`.

Every instance of a class is also an instance of all it superclasses (similar to sets). So an instance of `Undergrad` is also an instance of `Student`. An instance of `MSC` is also an instance of `Grad` and `Student`.

Since `RewardCard` is a subclass of `CreditCard`, it inherits `CreditCard`'s methods, e.g. `pay, getNumber`, etc., and fields (including encapsulated attributes).

`RewardCard` also defines some new methods, e.g. `getRewardPoints` and fields/attributes.

```
import type.lang.*;
import type.lib.*;
public class CardTest
{  public static void main(String[] args)
   {  CreditCard cc1 = new CreditCard(703,"John");
      IO.println("name: " + cc1.getName());
      IO.println("bal: " + cc1.getBalance());
      IO.println(cc1.toString());
      cc1.charge(120.0);
      cc1.charge(70.0);
      IO.println(cc1.getBalance());
      cc1.pay(50.0);
      IO.println(cc1.getBalance());
      IO.println();
      RewardCard rc1 = new RewardCard(704,"Paul",2000.0);
      IO.println("name: " + rc1.getName());
      IO.println("bal: " + rc1.getBalance());
      IO.println(rc1.toString());
      rc1.charge(120.0);
      IO.println("bal: " + rc1.getBalance());
      IO.println("points: " + rc1.getRewardPoints());
      rc1.charge(70.0);
      IO.println("bal: " + rc1.getBalance());
      IO.println("points: " + rc1.getRewardPoints());
      rc1.pay(50.0);
      IO.println("bal: " + rc1.getBalance());
      IO.println("points: " + rc1.getRewardPoints());
      rc1.credit(70.0);
      IO.println("bal: " + rc1.getBalance());
      IO.println("points: " + rc1.getRewardPoints());
      rc1.redeemPoints();
      IO.println("bal: " + rc1.getBalance());
      IO.println("points: " + rc1.getRewardPoints());
   }
}
```

```
zebra 315 % java CardTest
name: John
bal: 0.0
CARD [NO=000703-8, BALANCE=0.00]
190.0
140.0

name: Paul
bal: 0.0
RWRD [NO=000704-7, BALANCE=0.00, POINTS=0]
bal: 120.0
points: 6.0
bal: 190.0
points: 9.0
bal: 140.0
points: 9.0
bal: 70.0
points: 6.0
bal: 70.0
points: 0.0
zebra 316 %
```

Overriding Methods

Sometimes when we define a subclass, a method inherited from the superclass is inappropriate and we want to change it. We can do this by providing a new definition for the method in the subclass. The new definition *overrides*, i.e. replaces the inherited one.

E.g. in `RewardCard`:
`toString` is overridden to display the reward points,
`charge` is overridden to increment the reward points,
`credit` is overridden to decrement the reward points.

Overloaded vs Overridden Methods

It is important to distinguish between overloaded and overridden methods. As we have seen earlier, we can define several versions of a method that work with different types of arguments. This is called *overloading* the methods. E.g. the overloaded constructors:

```
CreditCard(int no, java.lang.String aName)
CreditCard(int no, java.lang.String aName, double aLimit)
```

Another e.g.:

We already have

```
boolean setLimit(double newLimit)
```

We could add:

```
boolean setLimit()
```

which sets the limit to a default value.

Then the user could write:

```
cc1.setLimit(2000.0);
cc1.setLimit();
```

The *signature* of a method is the number and types of its parameters and their ordering. E.g.

1st `CreditCard` constructor:	`(int,String)`
2nd `CreditCard` constructor:	`(int,String,double)`
original `setLimit`:	`(double)`
2nd `setLimit`:	`()`
perhaps a 3rd `setLimit`:	`(String)`

When overloading methods, you are defining several methods with the same name that will be available *in the same class*. This is only possible when the signatures of the methods are *different*.

To decide which overloaded method to call, the compiler looks at the number and types of the arguments. If the signatures are the same, it cannot determine which method is called.

In contrast, *overridden* methods have the *same signature*. The method in the subclass replaces that in the superclass (even though the superclass's version is still available using `super`). E.g.

class `Parent` with methods:

`void meth()` #1
`void meth(int n)` #2, overloads #1

class `Offspring extends Parent` with methods:

`void meth(int n)` #3, overrides #2

In an app, can write:

```
Parent o1 = new Parent();
Offspring o2 = new Offspring();
o1.meth();   // calls #1
o1.meth(31); // calls #2
o2.meth();   // calls #1
o2.meth(29); // calls #3
```

Note: constructors are never inherited.

Polymorphism

In OOP, classes correspond to types. When a class `Co` is a subclass of a class `Cp`, then the type `Co` is a subtype of `Cp`. If you have a reference to an instance of `Co`, it can be assigned to a variable of type `Cp` and manipulated as if it were an instance of the class `Cp`. E.g.

```
CreditCard cc1;
cc1 = new RewardCard(704,"Paul",1000.0);

Student s1;
s1 = new Undergrad(
  "Mary",201234567,"compsci",1);
```

The ability to use a value of a more specific type where one of a more general type is expected is called *polymorphism*, since the general type comes in "many forms".

You can also assign a variable declared as belonging to a superclass (a supertype) to a variable of a subclass (a subtype) provided you perform a type cast. E.g.

```
CreditCard cc1;
cc1 = new RewardCard(704,"Paul",1000.0);
RewardCard rc1;
rc1 = (RewardCard) cc1;

Student s1 = new Undergrad(
  "Mary",201234567,"compsci",1);
Undergrad u1 = (Undergrad) s1;
```

Note that if the object being cast does not belong to the type given (e.g. `s1` is not an instance of `Undergrad` or one of it's subclasses) an exception will be thrown when the program is run.

To be safe, you can check whether the object belongs to the right class using the `instanceof` operator before you perform the cast, e.g.

```
if (s1 instanceof Undergrad)
   u1 = (Undergrad) s1;
```

```
import type.lang.*;
import type.lib.*;
public class GlobalCreditEg
{  public static void main(String[] args)
   {  GlobalCredit yc = new GlobalCredit("York Credit",10);
      IO.println(yc.toString());
      yc.issue(703,"John",2000.0,'O');
      CreditCard cc1 = yc.getFirst();
      IO.println(yc.toString());
      yc.issue(704,"Paul",1000.0,'R');
      CreditCard cc2 = yc.getNext();
      yc.issue(705,"Jane",2500.0,'R');
      CreditCard cc3 = yc.getNext();
      IO.println(yc.toString());
      boolean res;
      res = yc.charge(cc2.getNumber(),500.0);
      IO.println("charging 500.0 to 704 = " + res);
      res = yc.charge(cc1.getNumber(),600.0);
      IO.println("charging 600.0 to 703 = " + res);
      for(CreditCard cc = yc.getFirst(); cc != null;
          cc = yc.getNext())
      {  IO.println(cc.toString());
      }
      res = yc.charge(cc2.getNumber(),800.0);
      IO.println("charging 800.0 to 704 = " + res);
      res = yc.charge(cc2.getNumber(),400.0);
      IO.println("charging 400.0 to 704 = " + res);
      for(CreditCard cc = yc.getFirst(); cc != null;
          cc = yc.getNext())
      {  IO.println(cc.toString());
      }
```

```
      res = yc.pay(cc1.getNumber(),200.0);
      IO.println("paying 200.0 to 703 = " + res);
      res = yc.redeemPoints(cc2.getNumber());
      IO.println("redeeming points on 704 = " + res);
      for(CreditCard cc = yc.getFirst(); cc != null;
          cc = yc.getNext())
      {  IO.println(cc.toString());
      }
      res = yc.charge(cc3.getNumber(),800.0);
      IO.println("charging 800.0 to 705 = " + res);
      for(CreditCard cc = yc.getFirst(); cc != null;
          cc = yc.getNext())
      {  if(cc instanceof RewardCard)
         {  RewardCard rc = (RewardCard) cc;
            IO.println(rc.getNumber() + " " +
                       rc.getRewardPoints());
         }
      }
      res = yc.getFirst().charge(200.0);
      IO.println("charging 200.0 directly to 703 = " + res);
      IO.println(yc.getFirst().toString());
   }
}
```

```
zebra 325 % java GlobalCreditEg
Global Credit Company [York Credit]: CARDS=0/10
Global Credit Company [York Credit]: CARDS=1/10
Global Credit Company [York Credit]: CARDS=3/10
charging 500.0 to 704 = true
charging 600.0 to 703 = true
CARD [NO=000703-8, BALANCE=600.00]
RWRD [NO=000704-7, BALANCE=500.00, POINTS=25]
RWRD [NO=000705-6, BALANCE=0.00, POINTS=0]
charging 800.0 to 704 = false
charging 400.0 to 704 = true
CARD [NO=000703-8, BALANCE=600.00]
RWRD [NO=000704-7, BALANCE=900.00, POINTS=45]
RWRD [NO=000705-6, BALANCE=0.00, POINTS=0]
paying 200.0 to 703 = true
redeeming points on 704 = true
CARD [NO=000703-8, BALANCE=400.00]
RWRD [NO=000704-7, BALANCE=900.00, POINTS=0]
RWRD [NO=000705-6, BALANCE=0.00, POINTS=0]
charging 800.0 to 705 = true
000704-7 0.0
000705-6 40.0
charging 200.0 directly to 703 = true
CARD [NO=000703-8, BALANCE=600.00]
zebra 326 %
```

Dynamic Method Binding

When you call an overridden method on a reference of polymorphic type (e.g. `cc1.toString()`), the version of the method that gets called depends on the actual type of the object referred to at run time; it is not necessarily that of the declared type. E.g.

```
CreditCard cc1;
cc1 = new CreditCard(703,"John",2000.0);
IO.println(cc1.toString());// calls CreditCard's
cc1 = new RewardCard(704,"Paul",1000.0);
IO.println(cc1.toString());// calls RewardCard's
```

This approach to selecting which version of an overridden method to call is named *dynamic* or *late method binding*. The term polymorphism is often used to refer to this specific feature of polymorphic method calls.

The Class `Object`

There is a root to the class hierarchy in Java, a class called `Object`. Every class that is defined without a superclass being specified is automatically made to be a subclass of `Object`.

Because it is so general, the class `Object` does not provide much. Its most useful methods are discussed below. It is generally a good idea to override these when you define a class.

`String toString()` generates a string representing the object, by default *ClassOfObject@AddressOfObject*, e.g. `type.lib.StockNS@86d4c1`. This is not very meaningful and we have seen how it is overriden in `Stock`.

`boolean equals(Object other)` tests whether this instance is equal to `other`. By default it returns true iff `this` and `other` are the same reference. It is better to override it to make a comparison based on the objects' attributes, e.g. as done in `Stock`.

`Object clone()` returns a new object which is a copy of this instance. By default this is a *shallow copy*, i.e. attributes which are themselves objects are not cloned, only their references are copied. If an object has attributes which are mutable objects, it is generally better to make a *deep copy*.

The `Object` class is also useful for defining very general data structures, e.g. a queue whose elements are of type `Object`, a `Vector` of `Objects`, etc. If you have such a data structure, then you can put any kind of object into it, `Stocks`, `Students`, `Integers` (wrapper class), etc.