COSC 1020

Yves Lespérance

Lecture Notes

Week 4 — The Object Abstraction

Recommended Readings:
Horstmann: Ch. 2 Sec. 1, 2, & 6 (rest of Ch. 2 optional)
Lewis & Loftus: Ch. 2 Sec. 0, 6, & 7
Horstmann may be hard to follow!

Objects and Classes with Non-Static Features

Besides modules, there is another kind of abstraction that you often need to use.

An *object* is an abstraction that combines some data (attributes) with some methods to operate on it. E.g. an object for a stock “NT” with attributes symbol, name, price, and delimiter.

There are many similar such objects that differ in the data they contain, e.g. “NT”, “BMO”, etc.

All these similar objects belong to a *class*, e.g. `Stock`. The class acts as a kind of template for the objects. When you want to deal with a particular individual of this sort (e.g. a stock), you create a new object that is an instance of the class and customize it. The class can also be viewed as a factory for such objects.

To create a new object that is an instance of the class, you call one of its constructors, e.g.

```
new Stock("NT")
```

If you want to be able to refer to the object, you have to save a reference to it in a variable, e.g.

```
Stock stk1 = new Stock("NT");
```

The “`Stock stk1`” part declares the variable `stk1` to have the type `Stock`. “`new Stock("NT")`” constructs the new object and returns a reference to it (its address), which is stored in the variable.

Once we have created an instance of a class can use methods defined by the class to operate on the instance. E.g. we can retrieve the price of the `stk1` object by calling its `getPrice` method:

```
IO.println(stk1.getPrice());
```

We can also retrieve and change the value of any `public` instance variable (attribute) that it has, e.g. the `name` instance variable of `stk1`:

```
IO.println(stk1.name);
```

Note that `public` instance variables are usually avoided because they allow users to put the object in an inconsistent state, e.g. change the name of the “BMO” symbol stock to “York University”. It is better to keep instance variables/attributes private and have users change their values by calling methods that ensure the object stays consistent.

Classes with Non-Static Features

The methods and fields that an object will have are defined by that class it belongs to. A class can have:

- *constructors,* that are used to create instances, objects that belong to the class, e.g. the two `Stock` constructors;
- *instance methods* (non-`static`), i.e., operations that belong to each object that is an instance of the class and can be called by users, e.g. `stk1.getPrice()`;
- *instance variables* (non-`static`), that store data associated with the object and whose values can be retrieved and changed by users, e.g. `stk1.name`;
- `static` *methods,* that belong to the class, e.g. could have `Stock.getTSXcompIndexValue()`;

- `static` *variables* and *constants,* that belong to the class, e.g. `Stock.titleCaseName`, `Stock.TSE_URL`.

Objects and classes are *abstractions/black boxes* with *public interfaces*. Information on the *public* methods and fields is collected in the class's API, which the users can consult. The public methods' implementation and the class's non-public fields and methods are kept *private* and hidden from users.

A class can be used to encapsulate a type, e.g. `Fraction`. When the API of a class with non-static features is compatible with several different implementation approaches, one says that it is an *abstract data type*.

E.g.

```
import type.lang.*;
import type.lib.*;

public class StockTest
{  public static void main(String[] args)
   {  Stock stk1 = new Stock("NT");
      IO.println(stk1.getName());
      IO.println(stk1.getPrice(),".2");
      IO.println("Check price again now?");
      IO.readLine();
      stk1.refresh();
      double price = stk1.getPrice();
      IO.println(price,".2");
      Stock stk2 = new Stock("BMO");
      IO.println(stk2.getName());
      IO.println(stk2.getPrice(),".2");
      stk2.setSymbol("BCE");
      IO.println(stk2.getName());
      IO.println(stk2.getPrice(),".2");
      IO.println(stk2.name);
      stk2.name = "York University";
      IO.println(stk2.getName());
      IO.println(stk2.getPrice(),".2");
   }
}
```

```
Script started on Wed Oct 02 14:14:11 2002
zebra 301 % java StockTest
Nortel Networks Corp
0.80
Check price again now?

0.80
Bank of Montreal
36.55
BCE Inc.
27.89
BCE Inc.
York University
27.89
zebra 302 % exit
exit

script done on Wed Oct 02 14:14:57 2002
```

Constructors

When you create an instance of a class by calling for e.g. `new Stock()`, you are actually calling one of the class's constructors.

Constructors have the same name as their class. A class can have several constructors. This is an example of "overloading", i.e. having several methods with the same name in one class. The overloaded methods must have a different number or types of arguments (signature).

For e.g., the class `Stock` has 2 constructors:

1. a `1` argument constructor that initializes the new stock's symbol attribute to the value supplied and uses the `refresh` method to initialize the name and value attribute, e.g.

   `Stock stk1 = new Stock("BMO");`

2. a `0` argument constructor that initializes the attributes to default values (null for strings, zero for numbers), e.g.

   `Stock stk1 = new Stock();`

Note that objects exist independently of the variables that refer to them. E.g.

```
Stock stk1 = new Stock("NT");
// now stk1 refers to Stock object with symbol NT
Stock stk2 = stk1;
// now both stk1 & stk2 refer to single object ‘‘NT’’
stk1 = new Stock("BMO");
// now stk1 refers to Stock object ‘‘BMO’’
// while stk2 still refers to ‘‘NT’ object
stk2 = null;
// now stk2 does not refer to anything &
// ‘‘NT’’ object will be garbage collected;
// stk1 still refers to ‘‘BMO’’ object
```

Note: calling a method on an object variable that contains `null` is an error.

Types of Methods

- *accessor methods*, which are used to retrieve the value of an attribute of the object, e.g. `getSymbol, getName, getDelimiter, getPrice`;
- *mutator methods*, which are used to change the value of an attribute of the object, e.g. `setSymbol, setDelimiter`;
- *standard methods*, e.g. `toString(), equals`, etc.;
- *specialized methods*, which are particular to the class, e.g. `refresh`.

The `toString()` Method

A standard method that returns a string representing the object as desired.

E.g. the `Stock` class defines one:

```
import type.lang.*;
import type.lib.*;
public class StockTest2
{  public static void main(String[] args)
   {  Stock stk1 = new Stock("NT");
      IO.println(stk1.toString());
      IO.println(stk1); // print calls toString
      stk1.setDelimiter('*');
      IO.println(stk1.toString());
      StockNS stkNS1 = new StockNS("NT");
      IO.println(stkNS1.toString());
   }
}
```

```
zebra 305 % java StockTest2
NT Nortel Networks Corp
NT Nortel Networks Corp
NT*Nortel Networks Corp
type.lib.StockNS@86d4c1
zebra 306 %
```

If a class does not define a `toString()` method, one is automatically inherited from the class `Object`; e.g. the `StockNS` class.

The `equals` Method

A standard method that allows the object to be compared to another to determine whether it is identical.

E.g. the `Stock` class defines a method
`public boolean equals(java.lang.Object other)`
that tests whether the `other` object is equal to `this` object based on their symbol attributes:

```
import type.lang.*;
import type.lib.*;
public class StockTest3
{  public static void main(String[] args)
   {  Stock stk1 = new Stock("NT");
      IO.println("constructed stk1 with symbol NT");
      Stock stk2 = new Stock("NT");
      IO.println("constructed stk2 with symbol NT");
      Stock stk3 = new Stock("BMO");
      IO.println("constructed stk3 with symbol BMO");
      IO.print("stk1.equals(stk2) is ");
      IO.println(stk1.equals(stk2));
      IO.print("stk1.equals(stk3) is ");
      IO.println( stk1.equals(stk3));
      stk3.setSymbol("NT");
      IO.println("set stk3 to have symbol NT");
      IO.print("stk1.equals(stk3) is ");
      IO.println(stk1.equals(stk3));
      IO.print("stk1 == stk3 is ");
      IO.println(stk1 == stk3);
      IO.print("stk1 == stk1 is ");
      IO.println(stk1 == stk1);
      StockNS stkNs1 = new StockNS("NT");
      IO.println("constructed stkNS1 with symbol NT");
      StockNS stkNs2 = new StockNS("NT");
      IO.println("constructed stkNS2 with symbol NT");
      IO.print("stkNs1.equals(stkNs2) is ");
      IO.println(stkNs1.equals(stkNs2));
      IO.print("stkNs1.equals(stkNs1) is ");
      IO.println(stkNs1.equals(stkNs1));
   }
}
```

```
zebra 316 % java StockTest3
constructed stk1 with symbol NT
constructed stk2 with symbol NT
constructed stk3 with symbol BMO
stk1.equals(stk2) is true
stk1.equals(stk3) is false
set stk3 to have symbol NT
stk1.equals(stk3) is true
stk1 == stk3 is false
stk1 == stk1 is true
constructed stkNS1 with symbol NT
constructed stkNS2 with symbol NT
stkNs1.equals(stkNs2) is false
stkNs1.equals(stkNs1) is true
zebra 317 %
```

Note that == can be used with objects. But it only considers objects identical if they have the same address. This is usually not what we want.

If a class does not define an `equals` method, one is automatically inherited from the class `Object` that functions like ==; e.g. `StockNS`.

E.g. LottoNumbers

Can use `Random` class from `java.util` to generate pseudo-random numbers. E.g.

```
import type.lang.*;
import java.util.Random;
public class LottoNumbers
{  public static void main(String[] args)
    {  final int MAX = 49;
       Random gen = new Random();
       int rn = gen.nextInt();  // get random int
       rn = Math.abs(rn); // make non-negative
       rn = rn % MAX;     // scale to be in [0,48]
       rn = rn + 1;       // shift to be in [1,49]
       IO.print(rn + " ");
       rn = gen.nextInt();// get another random int
       rn = Math.abs(rn); // make non-negative
       rn = rn % MAX;      // scale to be in [0,48]
       rn = rn + 1;       // shift to be in [1,49]
       IO.print(rn + " ");
       rn = gen.nextInt(MAX);// get rand int in [0,49)
       rn = rn + 1;         // shift to be in [1,49]
       IO.print(rn + " ");
       rn = gen.nextInt(MAX);
       rn = rn + 1;
       IO.print(rn + " ");
```

```
      rn = gen.nextInt(49);
      rn = rn + 1;
      IO.print(rn + " ");
      rn = gen.nextInt(49);
      rn = rn + 1;
      IO.println(rn);
   }
}
```

```
zebra 306 % java LottoNumbers
11 42 18 24 3 7
zebra 307 % java LottoNumbers
31 36 4 46 43 39
zebra 308 % java LottoNumbers
25 33 26 38 41 37
zebra 309 %
```