COSC 1020

Yves Lespérance

Lecture Notes

Week 11 — Implementing Classes I

Recommended Readings:
Horstmann: Ch. 2
Lewis & Loftus: Ch. 4 Sec. 0 to 6

Defining a Class — A Very Simple E.g.

```
// file Person.java

public class Person
{
   // constructor methods

   public Person()
   {
      name = "UNKNOWN";
      age = 0;
   }

   public Person(String n, int a)
   {
      name = n;
      age = a;
   }

   // instance methods

   public void setNameAndAge(String n, int a)
   {
      name = n;
      age = a;
   }

   public void setName(String name)
   {
      this.name = name;
   }
```

```
   public void setAge(int n)
   {
      age = n;
   }

   public void incrementAge()
   {
      age = age + 1;
   }

   public String getName()
   {
      return name;
   }

   public int getAge()
   {
      return age;
   }

   // instance variables/attributes

   private String name;
   private int age;

}// end class Person
```



```
// file TestPerson.java

import type.lang.*;

public class TestPerson
{  public static void main(String[] args)
   {
      Person p1 = new Person("John", 21);
      IO.println("p1's name is: " + p1.getName());
      IO.println("p1's age is: " + p1.getAge());
      p1.incrementAge();
      IO.println("p1's age is: " + p1.getAge());
      p1.setAge(18);
      IO.println("p1's age is: " + p1.getAge());
    }
}

zebra 43 % java TestPerson
p1's name is: John
p1's age is: 21
p1's age is: 22
p1's age is: 18
zebra 44 %
```

Elements of a Class Definition

A class definition may include the following:

- instance methods,
- constructors,
- instance variables/attributes/fields,
- class/static methods,
- class/static variables/constants.

Defining Methods

When you define a method, you take the steps required to solve a subproblem and give them a *name*. Afterwards, the method can be called without knowing how it is implemented. This is called *procedural abstraction*.

A method definition specifies:

- the name of the method,
- the name and types of *parameters* it takes,
- the type of *result* it returns,
- its visibility, i.e. `public`, `private`, etc.
- whether it is an instance or class (`static`) method,
- the steps required to execute it — the *body* of the method.

In programming languages that are not object-oriented, methods are often called *subprograms*.

Instance Variables/Attributes

For each piece of information that needs to be maintained about the instances of a class, you need to define an *instance variable/attribute/field* in the class, e.g.

```
public class Person
{
   // methods
   ...

   // instance variables/attributes

   private String name;
   private int age;

}// end class Person
```

If an attribute is `public`, users can access and change its value directly. But if the attribute is `private`, users can only access it indirectly by calling methods. E.g.


```
public class Car
{
   ...

   private String model;
   public int mileage;

}

...

Car aCar = new Car();
aCar.mileage = 23000;       // ok
IO.println(aCar.mileage);   // ok
aCar.model = "VW Beetle";   // error
IO.println(aCar.model);     // error
```

These access restrictions only apply to code outside the class. The methods of the class always have access to the class's attributes, e.g. `setAge` can change the value of the `age` attribute in a `Person` even though it is `private`.

Instance attributes are usually `private`. In this way, the class designer retains the right to change the way the data in the attributes is represented.

Returning Results from Methods

A method's header specifies whether or not it returns a *result*, and if it does, what the result's type is. When no result is returned, the method's result type is declared to be `void`.

After a method has been called and its body has finished executing, the execution of the program continues from the point where the method was called. If a method is to return a value to the place where it was called, it must terminate by executing the statement "`return` *expression*;", e.g. `return age;`.

Then, the expression is evaluated and its value is passed back to the point of the call as the method terminates.

Methods that return a value are often called *functions* and methods that do not are often called *procedures*.

Parameters

When we call a method, we often want to pass some data to it; the method can then use the data, save it in an attribute, or examine it to decide what actions to take. We do this by having the method take *parameters*. E.g. we need to pass the person's age to the `setAge` method; the age can be *any* value we want; the method uses the parameter n for this.

Parameters are declared in the header of the method definition. Both the parameter *name* and *type* are given, e.g.

```
public void setAge(int n)
{
   age = n;
}

public void setNameAndAge(String n, int a)
{...}
```


When you call a method, you supply an *argument* or *actual parameter* for each *formal parameter* in the method definition header. Arguments are associated to parameters by the order in which they appear. The number and type of arguments must match that of the parameters. E.g.

```
Person p1 = new Person();
int uAge = 44;
p1.setAge(uAge);
p1.setNameAndAge("Yves",44);
```

When a method is called, first the *parameters are passed*, and then the body of the method is executed.

Constructors

Constructors have the same name as their class. Their job is adequately initializing the new object's attributes.

A class can have several constructor methods. 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. For e.g., the class `Person` has 2 constructors:

1. a 2 arguments constructor that initializes the name and age of the new object to the values supplied, e.g.

   ```
   Person p1 = new Person("Yves", 44);
   ```

2. a 0 arguments constructor that initializes the attributes to default values, e.g.

   ```
   Person p1 = new Person();
   ```

In defining constructors, the result type need not be specified as it is always the constructor's class.

If you don't define any constructors, a 0 arguments constructor is automatically provided; it initializes the numeric attributes to 0, booleans to `false`, and objects to `null`.

About `this`

Within a class definition, `this` without parentheses always refers to the current instance of the class. It can be used to refer to the instance's attributes in a method that has a variable or parameter with the same name, e.g.

```
public class Person
{
   ...
   public void setName(String name)
   {
      this.name = name;
   }
   ...
   private String name;
   private int age;

}// end class Person
```

Here the parameter `name` defined in the method hides the attribute `name` defined in the class; but you can refer to the latter using `this.name`.

`this(...)` is a call to the class's constructor; see Horstmann p. 69.

E.g. Implementing the `CreditCard` Class

```
// file CreditCard.java
import type.lang.*;
public class CreditCard
{
   // constructors

   public CreditCard(int no, String aName, double aLimit)
   {  SE.require(0 < no && no <= 999999 && aLimit > 0);
      number = IO.format(no, "6z") + "-";
      int digitSum = 0;
      while(no > 0)
      {  digitSum = digitSum + no % 10;
         no = no / 10;
      }
      number = number + (MOD - digitSum % MOD);
      name = aName;
      limit = aLimit;
   }

   public CreditCard(int no, String aName)
   {  this(no, aName, DEFAULT_LIMIT);
   }

   // instance methods - accessors

   public double getBalance()
   {  return balance;
   }

   public double getLimit()
   {  return limit;
   }
```



```
   public String getName()
   {  return name;
   }

   public String getNumber()
   {  return number;
   }

   // mutators

   public boolean setLimit(double newLimit)
   {  if(newLimit >= 0 && newLimit >= balance)
      {  limit = newLimit;
         return true;
      }
      else
      {  return false;
      }
   }

   // specialized methods

   public boolean charge(double amount)
   {  SE.require(amount >= 0);
      if(balance+amount > limit)
      {  return false;
      }
      else
      {  balance = balance + amount;
         return true;
      }
   }

   public void credit(double amount)
   {  SE.require(amount >= 0);
      balance = balance - amount;
```

```
   }

   public void pay(double amount)
   {  SE.require(amount >= 0);
      balance = balance - amount;
   }

   // standard methods

   public boolean equals(Object anObject)
   {  return (anObject instanceof CreditCard &&
              number.equals(((CreditCard)anObject).number));
   }

   public String toString()
   {  String res = "CARD [";
      res = res + "NO=" + number;
      res = res + ", BALANCE=";
      res = res + IO.format(balance, ".2") + "]";
      return res;
   }

   // instance variables/attributes/fields

   private String number;
   private String name;
   private double limit;
   private double balance;

   // class/static variables/attributes/fields

   public static double DEFAULT_LIMIT = 1000.0;
   public static int MIN_NAME_LENGTH = 3;
   public static int MOD = 9;
   public static int SEQUENCE_NUMBER_LENGTH = 6;

}// end class CreditCard
```


Parameter Passing — The Details

As we saw earlier, parameter passing proceeds as follows:

1. the arguments are evaluated,
2. parameter variables are created,
3. the values of the arguments are *copied* into the parameter variables.

E.g. in the method call `c1.charge(amt)`, the value of the argument `amt`, say `20.0` is first obtained, then a new formal parameter variable `amount` is created, and then the value of the `amt` argument, `20.0`, is copied into this parameter variable. After this has been done, the body of the `charge` method is executed.

Same when the argument is an expression, e.g.

```
c1.charge(c1.getLimit() - c1.getBalance())
```


If method has several parameters, they are all passed in this way, e.g.

```
CreditCard c1 = new CreditCard(703,"John",5000.0).
```

Since the method is working with a copy of the argument, any changes made to the parameter variable don't affect the argument. So you cannot use parameters of primitive types to return values in Java. E.g.

```
public class App
{  public static void main(String[] args)
   {  int n = 99;
      IO.println("in main n = " + n);
      EgCl e = new EgCl();
      e.increment(n);
      IO.println("in main n = " + n);
   }
}

public class EgCl
{  public void increment(int m)
   {  IO.println("in increment m = " + m);
      m = m + 1;
      IO.println("in increment m = " + m);
   }
}
```


The mode of parameter passing used by Java is named *call by value* because it is the value of the argument which is passed to the formal parameter.

When the type of a parameter is an object type, only the reference in the argument gets copied in the parameter, and both the argument and parameter refer to the same object. So the method is working on the original object, and any change to its attributes persists when the method returns. E.g.

```
public class App
{  public static void main(String[] args)
   {  CreditCard c = new CreditCard(703,"John");
      IO.println("in main c = " + c.toString());
      EgCl e = new EgCl();
      e.increment(c);
      IO.println("in main c = " + c.toString());
   }
}

public class EgCl
{  public void increment(CreditCard ci)
   {  IO.println("in increment ci = " + ci.toString());
      ci.charge(100.0);
      IO.println("in increment ci = " + ci.toString());
   }
}
```

So, object parameters can be used by a method to send results back to the caller just as well as to receive data from the caller.

In languages like C++ and Pascal, there is a parameter passing mode named *call by reference* where the parameter receives a reference to the argument. The fact that object-type variables always contain references in Java makes object parameters behave somewhat as if they had been passed using call by reference.

But unlike true call by reference, changing which object the parameter is referring to does not change which object the argument is referring to. E.g.

```
public class App
{  public static void main(String[] args)
   {  CreditCard c = new CreditCard(703,"John");
      IO.println("in main c = " + c.toString());
      EgCl e = new EgCl();
      e.makeNewCard(c);
      IO.println("in main c = " + c.toString());
   }
}

public class EgCl
{  public void makeNewCard(CreditCard cm)
   {  cm =  new CreditCard(704,"Mary");
      IO.println("in makeNewCard cm = " + cm.toString());
   }
}
```


Control Flow and the Execution Stack

```
public class App
{  public static void main(String[] args)
   {  int n = 1;
      EgCl e = new EgCl();
      e.m1(n);
   }

}

public class EgCl
   public void m1(int n1)
   {  n1 = n1 + 1;
      m2(n1);
   }

   private void m2(int n2)
   {  n2 = n2 + 1;
      IO.println(n2);
   }
}
```

When a method, say `main`, calls a method `m1` (on e), the execution of `main` is suspended and `m1` starts executing. Only when the execution of `m1` terminates will the execution of `main` resume. If `m1` calls a third method `m2`, `m1` is also suspended until `m2` terminates. The chain of method calls can get arbitrarily long.

Since the first method to be called is always the last to resume, the Java interpreter uses a stack to keep track of control flow in a program — the *execution stack*. We will see that this is important when encounter methods that call themselves, i.e. recursion. The stack is also used to store a method's local variables.

Scope of Variables

The *scope* of a variable is the part of the program where it is *visible*, where it can be accessed. The variables declared inside a method, as well as its parameters, are said to be *local* to the method. One can only refer to them in the method or code block where they are declared. E.g.

```
public class Eg
{  public int meth1(int v2)
   {  int v3 = 3;
      IO.println(v3); // ok
      IO.println(v2); // ok
      IO.println(v1); // ok
      meth2(v3);
   }
   private void meth2(int v4)
   {  int v5 = 5;
      IO.println(v5); // ok
      IO.println(v4); // ok
      IO.println(v1); // ok
      IO.println(v3); // error
      IO.println(v2); // error
      while(...)
      {  int v6 = 6;
         IO.println(v6); // ok
         ...
      }
      IO.println(v6); // error
   }
   private int v1 = 1;
}
```


Access Control Revisited

For attributes and methods, one specifies where they are visible using access control modifiers such as `public` and `private`.

`public` means that the attribute or method is accessible everywhere. Normally we use this only for methods and class constants that are made available to users of the class.

`private` means that the attribute or method is only accessible inside the class where it is declared. Normally we use this for all attributes and for methods that are defined by the implementor for his own use and are not provided to users of the class.

Besides these, there are other access control modifiers such as `protected` (accessible in subclasses and other classes in the same package) and the default/no modifier (accessible other classes in the same package), which you will learn about in COSC 1030.

Why Define a Class?

There are two cases where defining a class is useful.

1. Your program needs to work with some kind of data, e.g. Persons. You want to *group together* the *data* and the *operations* that manipulate it.

You also want to *hide* the details of how the data is represented and how the operations are implemented from users of the class. The class will make some operations *public*, i.e. available to the users, and provide information on how to use them. This is the class's *interface*. The rest of the class's definition is *private* and hidden from users.

Sometimes, one says that the class defines an *abstract data type*. It is abstract because it you don't have to know the details to use it.

2. You want to *group together* a set of related operations in a *module*, e.g. the `Math` class. In this case, class users won't create instances of the class. The methods are associated with the class itself. In Java, they are labeled `static`.

Here too, the class supplies some *public* operations to users and provides information on how to use them in its *interface*. The rest of its definition is *private*.

In both cases, we say that the class *encapsulates*, i.e. hides, the details of its definition.