CSE 1030

Yves Lespérance

Lecture Notes

Week 1 — Implementing Static Features

Recommended Readings:
Savitch Ch. 5 and Van Breugel & Roumani Ch. 1

Object-Oriented Programming

Object-Oriented Programming (OOP) was developed to make it easier to develop and maintain large software systems.

A key concern there is ensuring that the software is decomposed into modules that different developers can work on independently.

OOP supports this by grouping data and related operations into classes and encapsulating the implementation of the classes.

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, courses, etc. 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/clients 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* or *API*. The rest of the class's definition is *private* and hidden from users. The API is like a contract between the clients and the implementer.

When such a class allows many different possible implementations, one says that the class defines an *abstract data type*; e.g. stack, list, binary tree, etc.

2. You want to *group together* a set of related operations in a *module* or *utility*, 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.

Elements of a Utility Class Definition

A utility class (i.e. one with only static features) definition may include the following:

```
// any needed package statement
// any needed import statements

public class ClassName
{
   // static attribute declarations
      ...

   // constructor definition - disallow use by clients
   private ClassName(){};


   // static method definitions
     ...

}// end class ClassName
```


Example: The `Math` Class

A typical example of a utility class is the `Math` class of the standard Java library, which collects various commonly used mathematical methods (e.g. absolute value, rounding, etc.) and constants (e.g. π).

A simplified `Math` class is discussed in the lecture notes. Have a look.

Declaring and Initializing Attributes

Data associated with a utility class is stored in its attributes.

A static/class attribute declaration has the form:

access `static` [`final`] type name [= value];

E.g.

```
public static final double PI = 3.141592653589793;
```


It specifies:

- whether the attribute is accessible to clients, i.e. `public` or `private`,
- that it is a `static` class attribute,
- whether it is `final`, i.e. a constant,
- the type of the attribute,
- the name of the attribute,
- and optionally, the initial value of the attribute.

A key guideline is to keep non-final attributes `private`. In this way, the class designer retains the right to change the way the data in the attributes is represented.

Constructors

Constructors have the same name as their class. Their job is to adequately initialize newly created instances of a class.

A utility class with only static features should not have instances.

But Java automatically creates a 0-arguments constructor if we don't define one.

To prevent clients from ever creating instances of a utility class, we can provide a constructor that we make private, e.g.

```
private Math(){};
```


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 (`void` if none),
- its access/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.

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 min;`.

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 numbers to compare to the `Math.min` method; the method uses the parameters `a` and `b` for this.

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

```
public static int min(int a, int b)
{
   int min;
   if (a <= b)
   {
      min = a;
   }
   else
   {
      min = b;
   }
   return min;
}
```


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.

```
int paulsAge = 23;
int marysAge = 17;
int minAge;
minAge = Math.min(paulsAge,marysAge);
int roundedAge;
roundedAge = Math.round(23.6);
```

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

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 `Math.min(paulsAge,marysAge)`, the value of the argument `paulsAge`, say `23` is first obtained, then a new formal parameter variable `a` is created, and then the value of the `paulsAge` argument, `23`, is copied into this parameter variable. Similarly for `marysAge`, its value is copied into the new parameter variable `b`. After this has been done, the body of the `Math.min` method is executed.

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

```
paulsAge + 5
```


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 ParamPassPrimApp
{
   public static void main(String[] args)
   {
      int n = 41;
      int m = 23;
      System.out.println("in main n = " + n);
      System.out.println("in main m = " + m);
      m = MyUtility.utilMethod(n);
      System.out.println("in main n = " + n);
      System.out.println("in main m = " + m);
   }
}
```

```
public class MyUtility
{
   private MyUtility(){}

   public static int utilMethod(int n)
   {
      System.out.println("in utilMethod n = " + n);
      n++;
      System.out.println("in utilMethod n = " + n);
      return(n);
   }
}
```


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.


```
import java.awt.geom.Point2D.Double;
import java.awt.geom.Point2D;

public class ParamPassingObjApp
{
   public static void main(String[] args)
   {
      Point2D.Double p1 = new Point2D.Double(1.5,3.0);
      System.out.println("in main p1 = " + p1);
      MyUtility.utilMethod(p1);
      System.out.println("in main p1 = " + p1);
   }
}

import java.awt.geom.Point2D;
import java.awt.geom.Point2D.Double;

public class MyUtility
{
   private MyUtility(){}

   public static void utilMethod(Point2D.Double p)
   {
      System.out.println("in utilMethod p = " + p);
      p.setLocation(1.6,3.1);
      System.out.println("in utilMethod p = " + p);
   }
}
```


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.

```
import java.awt.geom.Point2D;
import java.awt.geom.Point2D.Double;

public class MyUtility
{
   private MyUtility(){}

   public static void utilMethod(Point2D.Double p)
   {
      System.out.println("in utilMethod p = " + p);
      p.setLocation(1.6,3.1);
      p = new Point2D.Double(1.6,3.1);
      System.out.println("in utilMethod p = " + p);
   }
}
```


Control Flow and the Execution Stack

```
public class MethodCallingApp
{
   public static void main(String[] args)
   {
      int n = 41;
      int m = 23;
      System.out.println("in main n = " + n);
      System.out.println("in main m = " + m);
      System.out.println("in main MyUtility.getCount() = " +
                         MyUtility.getCount());
      m = MyUtility.utilMethod1(n);
      System.out.println("in main n = " + n);
      System.out.println("in main m = " + m);
      System.out.println("in main MyUtility.getCount() = " +
                         MyUtility.getCount());
   }
}
```



```
public class MyUtility
{
   private static int count = 0;

   private MyUtility(){}

   public static int getCount()
   {
      return count;
   }

   public static void setCount(int count)
   {
      MyUtility.count = count;
   }

   public static void incrementCount()
   {
      MyUtility.count++;
   }

   public static int utilMethod1(int n)
   {
      incrementCount();
      System.out.println("in utilMethod1 n = " + n);
      n++;
      n = utilMethod2(n);
      System.out.println("in utilMethod1 n = " + n);
      return(n);
   }
```



```
   private static int utilMethod2(int n)
   {
      incrementCount();
      System.out.println("in utilMethod2 n = " + n);
      n+= 10;
      n = utilMethod3(n);
      System.out.println("in utilMethod2 n = " + n);
      return(n);
   }

   private static int utilMethod3(int n)
   {
      incrementCount();
      System.out.println("in utilMethod3 n = " + n);
      n+= 100;
      System.out.println("in utilMethod3 n = " + n);
      //uncomment to print stack trace
      //Throwable t = new Throwable();
      //t.printStackTrace(System.out);
      return(n);
    }
}
```

When a method, say `main`, calls a method `utilMethod1` (on `MyUtility`), the execution of `main` is suspended and `utilMethod1` starts executing. Only when the execution of `utilMethod1` terminates will the execution of `main` resume. If `utilMethod1` calls a third method `utilMethod2`, `utilMethod1` is also suspended until `utilMethod2` 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.

`javadoc`: A Documentation Utility

Important to have good documentation of classes. API/external doc for clients/users and internal doc for implementers.

Can use `javadoc` utility to help produce external doc.

You put special comments in the class's file and then run `javadoc` on it to produce an HTML API documentation file.

`javadoc` comments start with `/**`. Put one immediately before each method, non-private field, and before the class itself.

Special tags (must start line):

`@param` parameter-name description

`@pre.` precondition

`@return` description

`@throws` exception if condition

etc.

Can include other HTML tags e.g. `<code>`, `<it>`, etc.

See lecture notes and textbook for examples.

`javadoc` automatically adds links to existing classes.

When designing a class, document API using `javadoc` before writing code.

Use normal comments to document class implementation.