# CSE 1030

## Yves Lespérance

## Lecture Notes

## Week 3 — Implementing Non-Static Features

Recommended Readings:
Savitch Ch. 4 and Van Breugel & Roumani Ch. 2

## `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 lab handbook and Horstmann 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.

## 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.

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*, 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`. We say that such a class is a *utility*.

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 Class Definition

A class definition may include the following:

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

   // constructor definitions
     ...

   // method definitions
     ...

}// end class ClassName
```



## Attribute Declarations and Initialization

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

```
public class Person
{
   // attributes

   private String name;
   private int age;

   // constructors
     ...

   // methods
     ...

}// end class Person
```

If an attribute is `public`, clients can access and change its value directly. But if the attribute is `private`, clients 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;             // allowed
System.out.println(aCar.mileage); // allowed
aCar.model = "VW Beetle";         // error
System.out.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`.

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.



These are non-static/instance attributes; will see later about adding static/class attributes.

Initialization of non-static attributes is done in constructors.

Constant attributes are labeled `final`.



## 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 has the form:

*access [static] return-type signature body*



It specifies:

- whether the method is accessible to clients, i.e. `public` or `private`,
- the type of *result* it returns, `void` if none,
- whether it is an instance or class (`static`) method,
- the method's signature, i.e. its name and the types (and names) of *parameters* it takes,
- 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 this.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 age)
{
   this.age = age;
}

public void setNameAndAge(String name, int age)
{...}
```



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",49);
```

When a method is called, first the *parameters are passed, using pass by value)*, and then the body of the method is executed.



## Constructors

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

They use headers like methods, but without a return type, as it is always the constructor's class.

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 3 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();
   ```

3. a 1 argument copy constructor that initializes the attributes to that of an existing Person object, e.g.

   ```
   Person p1 = new Person("Yves", 44);
   Person p2 = new Person(p1);
   ```



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
{
   private String name;
   private int age;
   ...

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

}// 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`.

Note that `this(...)` is a call to the class's constructor.



To clearly distinguish them from local variables, we always refer to non-static attributes as `this.attributeName`, and to static attibutes as `ClassName.attributeName`.

When you call a method on an instance of a class `this` refers to that instance of the class; it is an *implicit parameter* of the method.

## Obligatory Methods

All classes inherit certain methods from the class `Object`. These include:

`toString()`, which returns a string representing the object,

`equals(Object otherObject)`, which returns a boolean according to whether the argument object is equal to this object,

`hashCode()`, which returns an integer that can serve as a hash code for the object — it should be the same for objects that are equals.

It is generally better to override these with versions that are appropriate to the class being defined, e.g. `toString()` returning a person's name, and `equals` comparing persons based on their names or social insurance numbers.



## Implementing Interfaces

A class may be declared to implement certain interfaces, e.g.

```
public class Rectangle implements Comparable<Rectangle>
```

Then, it has to define the methods declared in the interface, e.g.

```
int compareTo(Rectangle rectangle).
```



## E.g. Implementing a `CreditCard` Class

```
// file CreditCard.java

public class CreditCard
{
   // instance variables/attributes/fields

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

   // class/static constants

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



```
// constructors

 public CreditCard(int no, String aName, double aLimit)
  { assert 0 < no && no <= 999999 && aLimit > 0; // precondition
    this.number = String.format("%06d", no) + "-";
    int digitSum = 0;
    while(no > 0)
    {  digitSum = digitSum + no % 10;
       no = no / 10;
    }
    this.number = this.number + (MOD - digitSum % MOD);
    this.name = aName;
    this.limit = aLimit;
    this.balance = 0;
 }

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

```
   // instance methods - accessors

   public double getBalance()
   {  return this.balance;
   }

   public double getLimit()
   {  return this.limit;
   }

   public String getName()
   {  return this.name;
   }

   public String getNumber()
   {  return this.number;
   }
```



```
   // mutators

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

   // specialized methods

   public boolean charge(double amount)
    { assert amount >= 0; // precondition
      if(this.balance+amount > this.limit)
      {  return false;
      }
      else
      {  this.balance = this.balance + amount;
         return true;
      }
   }

   public void credit(double amount)
   {  assert amount >= 0; // precondition
      this.balance = this.balance - amount;
```



```
   }

   public void pay(double amount)
   {  assert amount >= 0; // precondition
      this.balance = this.balance - amount;
   }

   // standard methods

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

   public String toString()
   {  String res = "CARD [";
      res = res + "NO=" + this.number;
      res = res + ", BALANCE=";
      res = res + String.format("%.2f", this.balance) + "]";
      return res;
   }
}// end class CreditCard
```



## 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;
      System.out.println(v3); // ok
      System.out.println(v2); // ok
      System.out.println(v1); // ok
      meth2(v3);
   }
   private void meth2(int v4)
   {  int v5 = 5;
      System.out.println(v5); // ok
      System.out.println(v4); // ok
      System.out.println(v1); // ok
      System.out.println(v3); // error
      System.out.println(v2); // error
      while(...)
      {  int v6 = 6;
         System.out.println(v6); // ok
         ...
      }
      System.out.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 we will discuss later.



## Steps to Class Implementation

Study API. Should be documented using `javadoc`.

Write 1st version of class with fields and methods required by API, leaving out implementation for now.

Write test harness that tests every feature of the class.

Identify private attributes and declare them.

Implement constructors, accessors, mutators, standard methods, specialized methods. Avoid redundancy by delegating and defining private methods.

Add new test cases as you implement methods. Test methods as early as possible. Fix bugs and run all tests again (bug fix may introduce new bugs).