# Understanding Constructors; Function Objects

## 5.1 Standard Java

### Goals

Starting with this lab we will use standard Java, though we only know a small part of the language. We will learn new features when they are needed to support our program design process, but we will only use the parts of the language we have introduced in the lectures.

### Visibility modifiers

The first new feature of Java we need introduces is *visibility modifiers*. In Java every class, interface, field, method declaration, and method definition/header can start with **`public`**, **`private`**, or **`protected`**: **`public`** elements are visible in all other classes (the same way we fields and methods are used in *FunJava*); **`private`** elements can only be accessed within their defining class; and **`protected`** elements are visible in the defining class and it's subclasses.

If the visibility modifier is omitted, elements are visible to all classes within the same *package*. In our projects, all classes are defined in the same (*default*) package, so we only add visibility modifiers when it serves a purpose:

- Interface methods are implicitly **`public`**. When a class implements an interface and its methods, each one must be `public` since the implementing class *cannot reduce their visibility*.
- If a (possibly **`abstract`**) class defines a method, then any class that **`extends`** it also *cannot reduce the visibility* of the method. E.g., if the **`super`** class defines the method as **`public`**, all subclasses must also define it as **`public`**.
- We will discuss **`protected`** visibility more later, but fields are generally declared **`protected`** to hide implementation details from unrelated classes.

**File organization**

A standard Java *Project* differs little from what we've seen. The difference is that every **public** class and interface must be declared in a separate file whose name is the same as the class or interface, with an extension of `".java"`. For example, if our project contained **public** `Book`, `Author`, and `ExamplesBooks` classes, then we would need to define these classes in files `Book.java`, `Author.java`, and `ExamplesBooks.java`, respectively. Typically, each *Project* contains all files that are used to solve one problem.

*Note*: As long as they are not public, we can define several classes/interfaces in the same file, just like we did in `FunJava` (though you can fix the errors from interface methods now).

**Projects and Run Configurations**

If you haven't been already, from now on you should set up a new Eclipse *Project* for each lab, assignment, or portfolio. Create a Project named *Lab-05* and add a new file, `Date.java`, to the default package (in the `src` directory).

- Copy the following data definition into your **`Date.java`** file. Feel free to make the class **`public`**.

```
// to represent a calendar date
class Date{
   int year;
   int month;
   int day;

   Date(int year, int month, int day){
      this.year = year;
      this.month = month;
      this.day = day;
   }
}
```

- Define an examples class called `DateExamples`, add the usual constructor, and define three examples of valid dates.
- Get the `tester.jar` and add it as an *External Jar*.

- Create a new *Run Configuration* with `tester.Main` as the main class, and `DateExamples` entered in the *Arguments* tab. Note that the argument is the name of the class, not of the file.
- Select *Apply* and/or *Run*.
- Next time you want to run the same project, make sure *Date.java* is shown in the main pane, then hit the green circle with the white triangle, or press *Ctrl F11*.

## 5.2 Understanding Constructors

### Assuring Data Integrity

The data definitions at times do not capture the meaning of data and the restrictions on what values can be used to initialize different fields. For example, if we have a class that represents a date in the calendar using three integers for the day, month, and year, we know that our program is interested only in some years (maybe between the years 1500 and 2500), the month must be between 1 and 12, and the day must be between 1 and 31 (though there are additional restrictions on the day, depending on the month and whether we are in a leap year).

Suppose we `Date` examples:

```
// Good dates
Date d20100228 = new Date(2010, 2, 28); // Feb 28, 2010
Date d20091012 = new Date(2009, 10, 12);// Oct 12, 2009

// Bad date
Date dn303323 = new Date(-30, 33, 23);
```

Of course, the third example is just nonsense. While complete validation of dates (months, leap-years, etc...) is a course of material itself, for the purposes of practicing constructors, we will simply make sure that the month is between 1 and 12, the day is between 1 and 31, and the year is between 1500 and 50000 (*we're thinking ahead!*).

Did you notice the repetition in the description of validity? It suggests we start with a few helper methods (*pre-abstraction* if you will...):

- method `validNumber` that consumes a number and the low and high bound and returns **`true`** if the number is within the bounds (inclusive).

- methods `validDay`, `validMonth`, and `validYear` designed in a similar manner.

Quickly design these methods. they're pretty easy, but if you must, design at least one; you can finish the others at home... Friday night. For testing purposes, have the methods you skipped return **`true`** for now. (We call such temporary method definitions *stubs*.)

Once you're done, change the `Date` constructor to the following:

```
Date(int year, int month, int day){
   if(this.validYear(year))
      this.year = year;
   else
      throw new IllegalArgumentException("Invalid year");

   if(this.validMonth(month))
      this.month = month;
   else
      throw new IllegalArgumentException("Invalid month");

   if(this.validDay(day))
      this.day = day;
   else
      throw new IllegalArgumentException("Invalid day");
}
```

This is the same as the `Time` class we saw in lecture. To signal an error or some other *exceptional* condition, we **`throw`** an instance of `RuntimeException`, of which `IllegalArgumentException` is a subclass.

If the program ever executes a statement like:

```
    throw new ...Exception("... message ...");
```

then Java *raises* the constructed exception/error. For our purposes now, this is as good as terminating the program and printing the message string.

The *tester* library provides methods to test constructors that should throw exceptions:

```
boolean t.checkConstructorException(Exception e,
                                    String className,
                                    ... constr args ...);
```

For example, the following test case verifies that our constructor throws the correct exception with the expected message, if the supplied year is 53000:

```
t.checkConstructorException(
         new IllegalArgumentException("Invalid year"),
         "Date", 53000, 12, 30);
```

Run your program with this test. Now change the test by providing an incorrect message, incorrect exception (e.g. `NoSuchElementException`), or by supplying arguments that do not cause an error, and see that the test(s) fail.

Java provides the class `RuntimeException` with a number of subclasses that can be used to signal different types of dynamic errors. Later we will learn how to handle errors and design new subclasses of `RuntimeException` to signal errors specific to our programs.

### Overloading Constructors: Providing Defaults.

When entering dates for the current year it is tedious to continually enter `2011`. We can provide an additional constructor that only requires the `month` and `day`, assuming the year should be `2011`.

Remembering the *single point of control* rule, we make sure that the new *overloaded* constructor defers all of the work to the primary *full* constructor:

```
Date(int month, int day){
   this(2011, month, day);
}
```

Add examples that use only the month and day to see that the constructor works properly. Include tests with invalid month or year as well.

### Overloading Constructors: Expanding Options.

The user may want to enter the date in the form: `"Oct 20 2010"`. To make this possible, we can add another constructor:

```
Date(String month, int day, int year){
   ...
}
```

Our first task is to convert a `String` that represents a month into a number. We can do it in a helper method `getMonthNo`:

```
// Convert a three letter month into the numeric value
int getMonthNo(String month){
   if(month.equals("Jan")){ return 1; }
   else{ if (month.equals("Feb")){ return 2; }
```

```
    else{ if (month.equals("Mar")){ return 3; }
    else{ if (month.equals("Apr")){ return 4; }
        ...
    else
      throw new IllegalArgumentException("Invalid month");
}
```

Our constructor can then invoke this method as follows:

```
Date(String month, int day, int year){
   // Invoke the prinmary constructor, with a valid month
   this(year, 1, day);

   // Change the month to the given one
   this.month = this.getMonthNo(month);
}
```

Complete the implementation, and check that it works correctly.

## 5.3 Abstracting with Function Objects

Download the files in *Lab5.zip*. The folder contains the files *ImageFile.java*, *ISelectImageFile.java*, *SmallImageFile.java*, *IListImageFile.java*, *MTListImageFile.java*, *ConsListImageFile.java*, and *ExamplesImageFile.java*.

Starting with partially defined classes and examples will give you the opportunity to focus on the new material and eliminate typing in what you already know. However, make sure you understand how the class is defined, what does the data represent, and how the examples were constructed.

Create a new **Project** *Lab5-sp11* and import into it all of the given files. Also import *tester.jar*.

### Introduction - Tutorial

We start by designing three familiar methods that deal with lists of files: `filterSmallerThan40000`, `filterNamesShorterThan4`, and `countSmallerThan40000`.

Look at the first two methods. They should only differ in the body of the conditional in the class `ConsListImage`. The two versions look like this:

```
if (this.first.size() < 40000)
if (this.first.name.length() < 4)
```

Both represent a boolean expression that depends only on the value of `this.first`. Think about the *filter* loop function in *DrRacket*. Its contract and header were:

```
;; filter: (X -> boolean) [Listof X] -> [Listof X]
;; to construct a list from all those items
;; in alox for which p holds
(define (filter p alox)...)
```

The argument *p* was a function/predicate that consumed an item from the list (for example the *first*) and produced a boolean value that indicated whether the item is *acceptable*.

Java does not allow us to use functions or methods as arguments. To get around this problem we need to go through several steps:

- Define an interface that contains as its only method the header for the desired predicate: the interface `ISelectImageFile`:

  ```
  // to represent a predicate for ImageFile-s
  public interface ISelectImageFile{

    // Return true if the given ImageFile
    // should be selected
    public boolean select(ImageFile f);
  }
  ```

- Now any class that implements this interface will have this predicate method. Suppose our `filter` method consumes an object of the type `ISelectImageFile` as follows:

  ```
  // produce a list of ImageFiles from this list
  // that satisfy the given predicate
  public filter(ISelectImageFile pick);
  ```

  Inside the method `filter` our template now includes

  ```
  ...  pick.select(ImageFile) ...  -- boolean
  ```

  and so, we can replace the two conditionals by

  ```
  if (pick.select(this.first))
  ```

- We now need to define a class that implements this interface. It needs to define the method `select` that consumes an instance of `ImageFile` and returns `true` if the size of the given object is less than 40000. The following class definition accomplishes this task:

```
/* Select image files smaller than 40000 */
public class SmallImageFile implements ISelectImageFile {

  /* Return true if the given ImageFile is smaller than 40000 */
  public boolean select(ImageFile f) {
    return f.height * f.width < 40000;
  }
}
```

- In the `Examples` class we can now invoke the `filter` method on an `IListImageFile` with an instance of `SmallImageFile` as the argument:

  ```
  IListImageFile mtImagelist = new MTListImageFile();
  IListImageFile imagelist = .....
  IListImageFile smallImagelist = ...

  ISelectImageFile smallFiles = new SmallImageFile();

  // test the method filter on small image files
  boolean testFilter(Tester t){
    return
    t.checkExpect(mtImagelist.filter(this.smallFiles),
                  this.mtImagelist) &&
    t.checkExpect(imageList.filter(this.smallFiles),
                  this.smallImagelist);
  }
  ```

  The conditional inside the `filter` method:

  ```
  if (pick.select(this.first))
  ```

  will select the `ImageFile` objects for which the size is smaller than 40000.

### 5.3.1 Practice

We will now practice the use of *function objects*. The only purpose for defining the class `SmallImageFile` is to implement one method that determines whether the given `ImageFile` object has the desired property (a predicate method). An instance of this class can then be used as an argument to a method that deals with `ImageFile`s.

1. Start with defining in the `ExamplesImageFile` class the missing tests for the class `SmallImageFile`.

2. Design the method `allSmallerThan40000` that determines whether all items in a list are smaller that 40000 pixels. The method should take an instance of the class `SmallImageFile` as an argument.

3. We now want to determine whether the name in the given `ImageFile` object is shorter than 4. Design the class `NameShorterThan4` that implements the `ISelectImageFile` interface with an appropriate predicate method.

   Make sure in the class `ExamplesImageFile` you define an instance of this class and test the method.

4. Design the method `allNamesShorterThan4` that determines whether all items in a list have a name that is shorter than 4 characters. The method should take an instance of the class `NameShorterThan4` as an argument.

5. Design the method `allSuchImageFile` that that determines whether all items in a list satisfy the predicate defined by the `select` method of a given instance of the type `ISelectImageFile`. *Note: This resembles the andmap function in DrRacket.* In the `ExamplesImageFile` class test this method by abstracting over the method `allSmallerThan40000` and the method `allNamesShorterThan4`.

6. Design the class `GivenKind` that implements the `ISelectImageFile` interface with a method that produces `true` for all `ImageFile`s that are of the given `kind`. The desired `kind` is given as a parameter to the constructor, and so is specified when a new instance of the class `GivenKind` is created.

   *Hint:* Add a field to represent the desired `kind` to the class `GivenKind`.

7. In the `ExamplesImageFile` class use the method `allSuch` and the class `GivenKind` to determine whether all files in a list are *jpg* files. This should be written as a test case for the method `allSuchImageFile`.

   Do it again, but now ask about the *giff* files.

8. If you have some time left, design the method `filterImageFile` that produces a list of all `ImageFile`s that satisfy the

`ISelectImageFile` predicate. Test it with as many of your predicates as you can.

9. Follow the same steps as above to design the method `anySuchImageFile` that that determines whether there is an item a list that satisfies the predicate defined by the `select` method of a given instance of the type `ISelectImageFile`.

10. Finish the work at home and save it in your portfolio.

    *Food for thought:* Think how this program would be different if we have instead worked with lists of `Book`s, or lists of `Shape`s.