Walk through 3 - Java Tutorial

Topic Functions

1. Introduction

In the first lab you wrote and run your first Java program (Hello World) using Eclipse. In the second lab you learned how to deal with:

- Primitive data types in Java
- Conditions and Loops
- Arrays

All examples that we used so far were using static methods defined in a Java class. In this lab you learn the anatomy of static methods, recursion and a Java application as a library of static methods. The next lab will focus on user defined data type, which is at the core of Object Oriented Programming (OOP).

2. Lab Objectives

The objective of this lab is to learn:

- Static methods in Java
- Recursion in Java
- Libraries and Clients

3. Lab Setup

Before beginning this lab, you should have:

1. The Eclipse and Java Runtime Environment on your computer.
2. Completed Lab walk-through 1 and 2 from the previous weeks.
3. Read chapter 1, pages 21-50 in your textbook.
4. **Before you start doing this lab, in your Eclipse workspace create a project named 2XB3_Lab3 inside a package named cas.2XB3.lab3.wt. Add classes to this project as you walk through this instruction.**

Note

1. For each task you are given the class template and helper codes and comments. The TA will give you a few minutes to complete the task on your own. Then you can verify your work, as the TA will walk you through the steps to complete the task.
2. YOU NEED TO HAVE YOUR OWN STORAGE DEVICE (E.G., A USB KEY) IN ORDER TO CREATE THE ECLIPSE WORKSPACE.
3. IT IS YOUR RESPONSIBILITY TO KEEP YOUR STORAGE DEVICE SAFE (WITH NECESSARY BACKUPS) FOR FUTURE USE OF THE PROJECTS YOU ARE CREATING IN THE LABS.

4. Lab Exercise

4.1 Static Methods

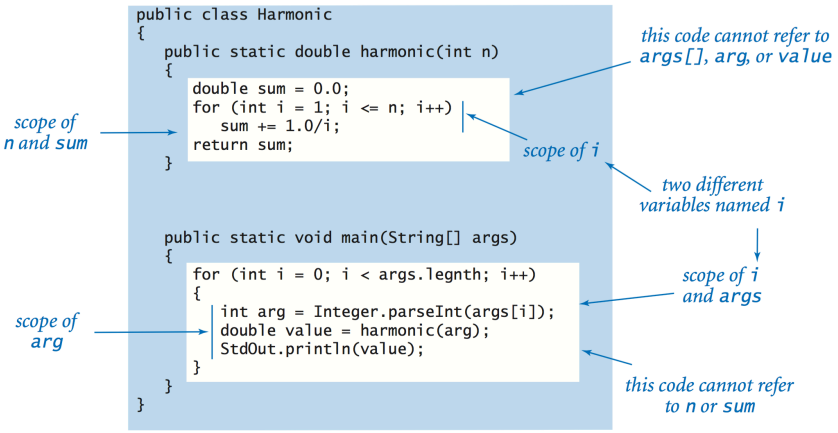
The Java construct for implementing functions is known as the *static method*.

Using and defining static methods.

The use of static methods is easy to understand. For example, when you write Math.abs(a-b) in a program, the effect is as if you were to replace that code with the *return value* that is produced by Java's Math.abs() method when passed the expression a-b as an *argument*.

Properties of static methods.

- *Multiple arguments.* Like a mathematical function, a Java static method can take on more than one argument, and therefore can have more than one parameter variable.
- *Multiple methods.* You can define as many static methods as you want in a .java file (a Java class). These methods are independent and can appear in any order in the file. A static method can call any other static method in the same file or any static method in a Java library such as Math.
- *Overloading.* Static methods with different signatures are different methods whose signatures differ. Using the same name for two static methods whose signatures differ is known as *overloading*.
- *Multiple return statements.* You can put return statements in a method wherever you need them: control goes back to the calling program as soon as the first return statement is reached.
- *Single return value.* A Java method provides only one return value to the caller, of the type declared in the method signature.
- *Scope.* The *scope* of a variable is the part of the program that can refer to that variable by name. The general rule in Java is that the scope of the variables declared in a block of statements is limited to the statements in that block. In particular, the scope of a variable declared in a static method is limited to that method's body. Therefore, you cannot refer to a variable in one static method that is declared in another.



- *Side effects.* A *pure function* is a function that, given the same arguments, always returns the same value, without producing any observable *side effects*, such as consuming input, producing output, or otherwise changing the state of the system. The

function harmonic() is an example of a pure function. However, in computer programming, we often define functions whose *only* purpose is to produce side effects. Can you think about some examples of these types of functions?

- In Java, a static method may use the keyword **void** as its return type, to indicate that it has no return value.

Task 1: Write a Java program to print the n'th harmonic number.

```
/*************************************************************************
 *  Compilation:  javac Harmonic.java
 *  Execution:    java Harmonic n
 *
 *  Prints the nth harmonic number: 1/1 + 1/2 + ... + 1/n.
 *
 *  % java Harmonic 10
 *  2.9289682539682538
 *
 *  % java Harmonic 10000
 *  9.787606036044348
 *
 *************************************************************************/

public class Harmonic {

    // returns 1/1 + 1/2 + 1/3 + ... + 1/n
    public static double harmonic(int n) {
        double sum = 0.0;
        //sum of 1/1 + 1/2 + 1/3 + ... + 1/n
        //your code here
        return sum;
    }

    public static void main(String[] args) {
        for (int i = 0; i < args.length; i++) {
            int arg = Integer.parseInt(args[i]);
            //call the harmonic method and store the return value into a
      double called value
            //your code here
            StdOut.println(value);
        }
    }
}
```

4.2 Recursion

The idea of calling one function from another immediately suggests the possibility of a function calling *itself*. The function-call mechanism in Java supports this possibility, which is known as *recursion*.

Your first recursive program.

The "Hello, World" for recursion is the *factorial* function, which is defined for positive integers *n* by the equation:

$$n! = n * (n-1) * (n-2) * \ldots * 2 * 1$$

The quantity *n*! is easy to compute with a for loop, but an even easier method in Factorial.java is to use the following recursive function:

```
public static long factorial(int n) {
   if (n == 1) return 1;
   return n * factorial(n-1);
}
```

We can trace this computation in precisely the same way that we trace any sequence of function calls.

```
factorial(5)
  factorial(4)
    factorial(3)
      factorial(2)
        factorial(1)
          return 1
        return 2*1 = 2
      return 3*2 = 6
    return 4*6 = 24
  return 5*24 = 120
```

Our factorial() implementation exhibits the two main components that are required for every recursive function.

- The *base case* returns a value without making any subsequent recursive calls. It does this for one or more special input values for which the function can be evaluated without recursion. For factorial(), the base case is *n* = 1.
- The *reduction step* is the central part of a recursive function. It relates the value of the function at one (or more) input values to the value of the function at one (or more) other input values. Furthermore, the sequence of input values values must *converge* to the base case. For factorial(), the value of *n* decreases by 1 for each call, so the sequence of input values converges to the base case.

Mathematical induction.

Recursive programming is directly related to *mathematical induction*, a technique for proving facts about natural numbers. Proving that a statement involving an integer *n* is true for infinitely many values of *n* by mathematical induction involves the following two steps:

- The *base case*: prove the statement true for some specific value or values of *n* (usually 0 or 1).
- The *induction step*: assume that the statement to be true for all positive integers less than *n*, then use that fact to prove it true for *n*.

Such a proof suffices to show that the statement is true for *infinitely* many values of *n*: we can start at the base case, and use our proof to establish that the statement is true for each larger value of *n*, one by one.

Task 2: Write a Java program to calculate the greatest common divisor

```
/******************************************************
 *  Compilation:  javac Euclid.java
 *  Execution:    java Euclid p q
 *
 *  Reads two command-line arguments p and q and computes the greatest
 *  common divisor of p and q using Euclid's algorithm.
 *
 *  Remarks
 *  -----------
 *   may return the negative of the gcd if p is negative
 *
 ******************************************************************/

public class Euclid {

    // recursive implementation
    public static int gcd(int p, int q) {
        //recursively find the greatest common divisor
        //your code here
    }
    // non-recursive implementation
    public static int gcd2(int p, int q) {
        //non-recursively find the greatest common divisor
        //your code here
    }
    public static void main(String[] args) {
        int p = Integer.parseInt(args[0]);
        int q = Integer.parseInt(args[1]);
        int d  = gcd(p, q);
        int d2 = gcd2(p, q);
        StdOut.println("gcd(" + p + ", " + q + ") = " + d);
        StdOut.println("gcd(" + p + ", " + q + ") = " + d2);
    }
}
```

Pitfalls of recursion.

With recursion, you can write compact and elegant programs that fail spectacularly at runtime.

- *Missing base case.* The recursive function in NoBaseCase.java is supposed to compute harmonic numbers, but is missing a base case:

```
public static double harmonic(int n) {
   return harmonic(n-1) + 1.0/n;
}
```

  If you call this function, it will repeatedly call itself and never return.

- *No guarantee of convergence.* Another common problem is to include within a recursive function a recursive call to solve a subproblem that is not smaller than the original problem. For example, the recursive function in NoConvergence.java goes into an infinite recursive loop for any value of its argument (except 1).

```
public static double harmonic(int n) {
   if (n == 1) return 1.0;
   return harmonic(n) + 1.0/n;
}
```

- *Excessive memory requirements.* If a function calls itself recursively an excessive number of times before returning, the memory required by Java to keep track of the recursive calls may be prohibitive. The recursive function in ExcessiveMemory.java correctly computes the nth harmonic number. However, calling it with a huge value of n will lead to a StackOverflowError.

```
public static double harmonic(int n) {
  if (n == 0) return 0.0;
  return harmonic(n-1) + 1.0/n;
}
```

- *Excessive recomputation.* The temptation to write a simple recursive program to solve a problem must always be tempered by the understanding that a simple program might require exponential time (unnecessarily), due to excessive recomputation. For example, the Fibonacci sequence

```
fibonacci(8)
   fibonacci(7)
      fibonacci(6)
         fibonacci(5)
            fibonacci(4)
               fibonacci(3)
                  fibonacci(2)
                     fibonacci(1)
                        return 1
                     fibonacci(0)
                        return 0
                     return 1
                  fibonacci(1)
                     return 1
                  return 2
               fibonacci(2)
                  fibonacci(1)
                     return 1
                  fibonacci(0)
                     return 0
                  return 1
               return 3
            fibonacci(3)
               fibonacci(2)
                  fibonacci(1)
                     return 1
                  fibonacci(0)
                     return 0
                  return 1
               fibonacci(1)
                  return 1
               return 2
            return 5
         fibonacci(4)
            fibonacci(3)
               fibonacci(2)
                  .
                  .
```

0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, 233, ...

is defined by the formula:

$$F_n = F_{n-1} + F_{n-2} \; for \; n \geq 2, with \; F_0 = 0 \; and \; F_1 = 1$$

A novice programmer might implement this recursive function to compute numbers in the Fibonacci sequence, as in Fibonacci.java:

```
// Warning: spectacularly inefficient.
public static long fibonacci(int n) {
   if (n == 0) return 0;
   if (n == 1) return 1;
   return fibonacci(n-1) + fibonacci(n-2);
}
```

Task 3: Experiment that this Java program is spectacularly inefficient! Run the program for fibonacci(1), fibonacci(10) and fibonacci(100). Can observe the inefficiency? Open a task3.txt file in your Eclipse project and write what the function does to compute fibonacci(8) = 21. Describe what does it compute first and identifiy as formula how may time fibonacci(1) is being computed.

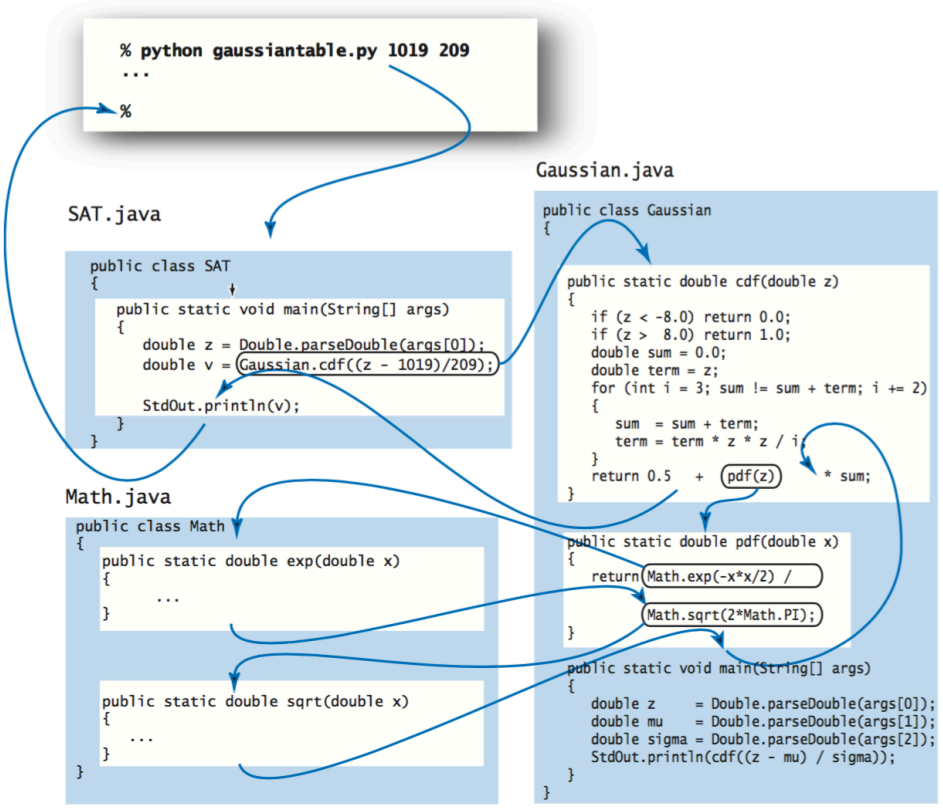
4.3 Libraries and Clients

Each program that you have written consists of Java code that resides in a single `.java` file. For large programs, keeping all the code in a single file is restrictive and unnecessary. Fortunately, it is very easy in Java to refer to a method in one file that is defined in another. This ability has two important consequences on our style of programming:

- It allows us to *extend the Java language* by developing libraries of static methods for use by any other program, keeping each library in its own file.
- It enables *modular programming*, where we divide a program up into static methods, grouped together in some logical way.

Using static methods in other programs.

To refer to a static method in one class that is defined in another, you must make both classes accessible to Java (for example, by putting them both in the same directory in your computer). Then, to call a method, prepend its class name and a period separator. For example, SAT.java calls the `cdf()` method inGaussian.java, which calls the `pdf()` method, which calls the `exp()` and `sqrt()`methods in Math.



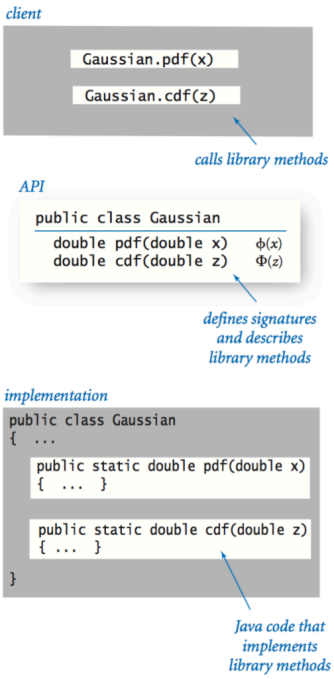
We describe several details about the process.

- *The public keyword.* The `public` modifier identifies the method as available for use by any other program. You can also identify methods as `private`, but you have no reason to do so at this point.
- *Each module is a class.* We use the term *module* to refer to all the code that we keep in a single file. By convention, each module is a Java class that is kept in a file with the same name of the class but has a `.java` extension. In this chapter, each class is merely a set of static methods.
- *The .class file.* When you compile the program, the Java compiler makes a file with the class name followed by a `.class` extension that has the code of your program in a language more suited to your computer.
- *Compile when necessary.* When you compile a program, Java typically compiles everything that needs to be compiled in order to run the program. For example, when you type `javac SAT.java`, the compiler will also check whether you modified `Gaussian.java` since the last time it was compiled. If so, it will also compile`Gaussian`.
- *Multiple `main()` methods.* Both SAT.java and Gaussian.java have their own `main()` method. When you type `java` followed by a class name, Java transfers control to the machine code corresponding to the `main()` method defined in that class.

Libraries.

We refer to a module whose methods are primarily intended for use by many other programs as a *library*.

- *Clients.* We use the term *client* to refer to a program that calls a given library method.
- *APIs.* Programmers normally think in terms of a *contract* between the client and the implementation that is a clear specification of what the method is to do.
- *Implementations.* We use the term *implementation* to describe the Java code that implements the methods in an API.



As an example, Gaussian.java is an implementation of the following API:

```
public class Gaussian
```

`double pdf(double x)`	$\phi(x)$
`double pdf(double x, double mu, double sigma)`	$\phi(x, \mu, \sigma)$
`double cdf(double z)`	$\Phi(z)$
`double cdf(double z, double mu, double sigma)`	$\Phi(z, \mu, \sigma)$

Task 4: Write a library of static method Hyperbolic.java that uses Math.exp() to implement the *hyperbolic* functions based on the definitions $\sinh(x)=(e^x-e^{-x})/2$ and $\cosh(x)= (e^x+e^{-x})/2$, with $\tanh(x)$, $\coth(x)$, $\operatorname{sech}(x)$, and $\operatorname{csch}(x)$ defined in a manner analogous to standard trigonometric functions.

```
/*************************************************************************
 *  Compilation:  javac Hyperbolic.java
 *  Execution:    java Hyperbolic x
 *
 *  Static library of hyperbolic trigonometric functions.
 *
 *  Remark
 *  ------
 *  Java 1.5 includes more robust methods Math.sinh(),
 *  Math.cosh(), and Math.tanh().
 *************************************************************************/

public class Hyperbolic {

   public static double cosh(double x) {
//your code here
   }

   public static double sinh(double x) {
 //your code here
```

```
   }

    public static double tanh(double x) {
 //your code here
   }


    public static void main(String[] args) {
        double x = Double.parseDouble(args[0]);
        StdOut.printf("sinh(%f) = %f\n", x, sinh(x));
        StdOut.printf("cosh(%f) = %f\n", x, cosh(x));
        StdOut.printf("tanh(%f) = %f\n", x, tanh(x));
    }
```

Task 5. Submit your work

Once all tasks are completed, you should submit your Eclipse project. Follow the instructions below for submission:

- Include a .txt file named last_name_initials.txt in the root of the project containing on separate lines: Full name, student number, any design decisions/assumptions you feel need explanation or attention.
- After checking the accuracy and completeness of your project, save everything and right-click on the name of the project, select Export->General->Archive File.
- Ensure that just your project has a check-mark beside it, and select a path to export the project to. The filename of the zipped project must follow this format: *macID_Lab3.zip.* Check the option to save the file in .zip format. Click Finish to complete the export.
- Go to Avenue and upload your zipped project to 'Lab Walk-through 3 – Lab Section X)
- IMPORTANT: You MUST export the FULL Eclipse project. Individual files (e.g. java/class files) will NOT be accepted as a valid submission.

5. Practice Problems

1. Write a static method `max3()` that takes three `int` arguments and returns the value of the largest one. Add an overloaded function that does the same thing with three `double` values.

2. Write a static method `majority()` that takes three `boolean` arguments and returns `true` if at least two of the arguments are `true`, and `false` otherwise. Do not use an `if` statement.

3. Consider the following recursive function.

```
public static int mystery(int a, int b) {
   if (b == 0)     return 0;
   if (b % 2 == 0) return mystery(a+a, b/2);
   return mystery(a+a, b/2) + a;
}
```

What are the values of mystery(2, 25) and mystery(3, 11)? Given positive integers a and b, describe what value mystery(a, b) computes.