2.1 Functions
A Foundation for Programming

Any program you might want to write

- Objects
- Functions and modules
- Graphics, sound, and image I/O
- Arrays
- Conditionals and loops
- Math
- Text I/O
- Primitive data types
- Assignment statements

Build bigger programs and reuse code
Functions

• Take in input arguments (zero or more)
• Perform some computation
  – May have side-effects (such as drawing)
• Return one output value

\[
f(x, y, z)\]

Input Arguments
\[x\] \[y\] \[z\]

Return Value
\[f(x, y, z)\]
Functions (Static Methods)

• Applications:
  – Use mathematical functions to calculate formulas
  – Use functions to build modular programs

• Examples:
  – Built-in functions:
    
    Math.random(), Math.abs(), Integer.parseInt()

  – I/O libraries:
    
    ellipse(), beginShape(), StdAudio.play()

  – User-defined functions:
    
    main()
Anatomy of a Java Function

• Java functions – It is easy to write your own
  – Example: `double sqrt(double c)`

```
public static double sqrt(double c) {
  ...
}
```

Please note that the method’s signature is defined incorrectly in the figure on pg 188 of your textbook
Anatomy of a Java Function

- Java functions – It is easy to write your own
  - Example: `double sqrt(double c)`

```java
define sqrt(c) = \sqrt{c}
```

2.0 → input → `sqrt(c) = \sqrt{c}` → output → 1.414213...

```java
public static double sqrt(double c)
{
    if (c < 0) return Double.NaN;
    double err = 1e-15;
    double t = c;
    while (Math.abs(t - c/t) > err * t)
        t = (c/t + t) / 2.0;
    return t;
}
```

- Local variables
- Method body
- Return statement
- Call on another method
Flow of Control

Functions provide a new way to control the flow of execution.
Flow of Control

What happens when a function is called:

- Control transfers to the function
- Argument variables are assigned the values given in the call
- Function code is executed
- Return value is substituted in place of the function call in the calling code
- Control transfers back to the calling code

Note: This is known as "pass by value"
Organizing Your Program

• Functions help you organize your program by breaking it down into a series of steps
  – Each function represents some abstract step or calculation
  – Arguments let you make the function have different behaviors

• **Key Idea**: write something ONCE as a function then reuse it many times
Scope:

- The code that can refer to a particular variable

- A variable's scope is the entire code block (any nested blocks) after its declaration

Simple example:

```c
int count = 1;
for (int i = 0; i < 10; i++){
    count *= 2;
}
// using 'i' here generates
// a compiler error
```

Best practice: declare variables to limit their scope
Scope with Functions

```java
public class Newton {
    public static double sqrt(double c) {
        if (c < 0) return Double.NaN;
        double err = 1e-15;
        double t = c;
        while (Math.abs(t - c/t) > err * t)
            t = (c/t + t) / 2.0;
        return t;
    }

    public static void main(String[] args) {
        int N = args.length;
        double[] a = new double[N];
        for (int i = 0; i < N; i++)
            a[i] = Double.parseDouble(args[i]);
        for (int i = 0; i < N; i++)
            double x = sqrt(a[i]);
        StdOut.println(x);
    }
}
```
public class Cubes1 {

    public static int cube(int i) {
        int j = i * i * i;
        return j;
    }

    public static void main(String[] args) {
        int N = Integer.parseInt(args[0]);
        for (int i = 1; i <= N; i++)
            System.out.println(i + " " + cube(i));
    }
}

% javac Cubes1.java
% java Cubes1 6
1 1
2 8
3 27
4 64
5 125
6 216
Last In First Out (LIFO) Stack of Plates
Method Overloading

• Two or more methods in the same class may also have the same name

• This is called method overloading

<table>
<thead>
<tr>
<th>absolute value of an int value</th>
<th>absolute value of a double value</th>
</tr>
</thead>
<tbody>
<tr>
<td>public static int abs(int x)</td>
<td></td>
</tr>
<tr>
<td>{</td>
<td></td>
</tr>
<tr>
<td>if (x &lt; 0) return -x;</td>
<td></td>
</tr>
<tr>
<td>else return x;</td>
<td></td>
</tr>
<tr>
<td>}</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td>public static double abs(double x)</td>
<td></td>
</tr>
<tr>
<td>{</td>
<td></td>
</tr>
<tr>
<td>if (x &lt; 0.0) return -x;</td>
<td></td>
</tr>
<tr>
<td>else return x;</td>
<td></td>
</tr>
<tr>
<td>}</td>
<td></td>
</tr>
</tbody>
</table>
Method Signature

• A method is uniquely identified by
  - its **name** and
  - its **parameter list** (parameter types and their order)
• This is known as its **signature**

Examples:

```java
static int min (int a, int b)
static double min (double a, double b)
static float min (float a, float b)
```
Return Type is Not Enough

• Suppose we attempt to create an overloaded \( \text{circle}(\text{double } x, \text{double } y, \text{double } r) \) method by using different return types:

\[
\begin{align*}
\text{static void } & \text{circle } (\text{double } x, \text{double } y, \text{double } r) \{ \ldots \} \\
//\text{returns true if circle is entirely onscreen, false otherwise } & \text{static boolean } \text{circle } (\text{double } x, \text{double } y, \text{double } r) \{ \ldots \}
\end{align*}
\]

• This is NOT valid method overloading because the code that calls the function can ignore the return value

\[
\text{circle}(50, 50, 10);
\]

– The compiler can’t tell which \text{circle()} method to invoke
– Just because a method returns a value doesn’t mean the calling code has to use it
Too Much of a Good Thing

Automatic type promotion and overloading can sometimes interact in ways that confuse the compiler. For example:

```java
//version 1
static void printAverage (int a, double b) {
    ...
}

//version 2
static void printAverage (double a, int b) {
    ...
}
```

Why might this be problematic?
Too Much of a Good Thing

static void printAverage (int a, double b) { /*code*/}
static void printAverage (double a, int b) { /*code*/}

• Consider if we do this:

```java
public static void main (String[] args) {
    ...
    printAverage(4, 8);
    ...
}
```

• The Java compiler can’t decide whether to:
  – promote 7 to 7.0 and invoke the first version of printAverage(), or
  – promote 5 to 5.0 and invoke the second version
• It will throw up its hands and complain
• Take-home lesson: don’t be too clever with method overloading
More Documentation
Method-level Documentation

• Method header format:

/**
 * Name: circleArea
 * PreCondition: the radius is greater than zero
 * PostCondition: none
 * @param radius - the radius of the circle
 * @return the calculated area of the circle
 */

static double circleArea (double radius) {
    // handle unmet precondition
    if (radius < 0.0) {
        return 0.0;
    } else {
        return Math.PI * radius * radius;
    }
}
Method Documentation

• Clear communication with the class user is of paramount importance so that he can
  - use the appropriate method, and
  - use class methods properly.

• Method comments:
  - explain what the method does, and
  - describe how to use the method.

• Two important types of method comments:
  - precondition comments
  - post-conditions comments
Preconditions and Postconditions

• **Precondition**
  - What is assumed to be true when a method is called
  - If any pre-condition is not met, the method may not correctly perform its function.

• **Postcondition**
  - States what will be true after the method executes (assuming all pre-conditions are met)
  - Describes the side-effect of the method
An Example

Very often the precondition specifies the limits of the parameters and the postcondition says something about the return value.

/* Prints the specified date in a long format 
   e.g. 1/1/2000 -> January 1, 2000 
   Pre-condition: 
     1 <= month <= 12 
     day appropriate for the month 
     1000 <= year <= 9999 
   Post-condition: 
     Prints the date in long format 
*/
static printDate(int month, int day, int year)
{
  // code here
}