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

![Diagram of function]

\[ f(x, y, z) \]
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(), size()
  
  – User-defined functions:
    setup(), draw(), mousePressed()
Anatomy of a Java Function

• Functions – It is easy to write your own

  – Example: `double sqrt(double c)`

    
    ```java
    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

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

```java
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;
}
```

Example:
- `double sqrt(double c)`
  - `sqrt(2.0)`
    - `input`: 2.0
    - `output`: 1.414213…
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

Scope: the code that can refer to a particular variable
- A variable's scope is the entire code block (any any nested blocks) after its declaration

Simple example:

```plaintext
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
Tracing Functions

```java
String arg = "6"

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

void setup() {
    int N = Integer.parseInt(arg);
    for (int i = 1; i <= N; i++)
        System.out.println(i + " " + cube(i));
}
```

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*

```java
int abs(int x)
{
    if (x < 0) return -x;
    else return x;
}

double abs(double x)
{
    if (x < 0.0) return -x;
    else return x;
}
```
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:

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

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

```java
void circle (double x, double y, double r) {...}

//returns true if circle is entirely onscreen, false otherwise
boolean circle (double x, double y, double r) {...}
```

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

```java
circle(50, 50, 10);
```

  – The compiler can’t tell which `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:

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

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

Why might this be problematic?
void printAverage (int a, double b) {/*code*/}
void printAverage (double a, int b) {/*code*/}

• Consider if we do this:

```java
void setup() {
    ...  
    printAverage(4, 8);
    ...  
}
```

• The 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:

```java
/**
 * 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
 */

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
*/
void printDate(int month, int day, int year)
{
    // code here
}