2.1 Functions
Functions

• Take in input arguments (zero or more)
• Perform some computation
  - May have side-effects (such as drawing)
• Return one output value
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()
      These methods return, respectively, a double, double, and int value.
  - I/O libraries:
    - PennDraw.circle(x, y, halfRadius),
      PennDraw.line(x0, y0, x1, y1)
  - User-defined functions:
    - main()
Why do we need functions?

• Break code down into logical sub-steps

• Readability of the code improves

• Testability - focus on getting each individual function correct
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 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)`

```
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;
}
```
Flow of Control

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

```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]);
            System.out.println(x);
        }
    }
}
```

implicit return statement at end of void function
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"
Example

• Function to reverse a word

• Apply this word reversal function to reverse a sentence that is entered via command line arguments.

Live coding time .....
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
Functions are useful!

- Common adage in programming – DRY principle

- DRY = Don’t Repeat Yourself

As opposed to

WET = Write Everything Twice

- Remember, if you are writing too much code that looks similar, it is time to think about a function!
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:

```java
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
Q. What happens when you compile and run the following code?

```java
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));
    }
}
```
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));
    }
}
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*

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

public static double abs(double x)
{
    if (x < 0.0) return -x;
    else return x;
}
```
Method Overloading

• We need some way to uniquely identify a method
• The name of the method alone isn’t enough
  - PennDraw.square(0.5, 0.5, 0.25)
  - PennDraw.square(0.5, 0.5, 0.25, 45)

The methods have the same name, but do different things!
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 `circle(double x, double y, double r)` method by using different return types:

  ```java
  static void circle(double x, double y, double r) {...}
  //returns true if circle is entirely onscreen, false otherwise
  static 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
Automatic type promotion and overloading can sometimes interact in ways that confuse the compiler. For example:

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

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

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

• Consider if we do this

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

• The Java compiler can’t decide whether to:
  - promote 7 to 7.0 and invoke the first version of average(), or
  - promote 5 to 5.0 and invoke the second version

• Take-home lesson: don’t be too clever with method overloading
Function Examples

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

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

<table>
<thead>
<tr>
<th>primality test</th>
</tr>
</thead>
<tbody>
<tr>
<td>public static boolean isPrime(int N)</td>
</tr>
<tr>
<td>{</td>
</tr>
<tr>
<td>if (N &lt; 2) return false;</td>
</tr>
<tr>
<td>for (int i = 2; i &lt;= N/i; i++)</td>
</tr>
<tr>
<td>if (N % i == 0) return false;</td>
</tr>
<tr>
<td>return true;</td>
</tr>
<tr>
<td>}</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>hypotenuse of a right triangle</th>
</tr>
</thead>
<tbody>
<tr>
<td>public static double hypotenuse(double a, double b)</td>
</tr>
<tr>
<td>{</td>
</tr>
<tr>
<td>return Math.sqrt(a<em>a + b</em>b);</td>
</tr>
<tr>
<td>}</td>
</tr>
</tbody>
</table>
Q. What happens when you compile and run the following code?

```java
public class Cubes2 {

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

  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));
  }
}
```
Q. What happens when you compile and run the following code?

```java
public class Cubes3 {
    public static int cube(int i) {
        i = i * i * i;
    }

    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));
    }
}
```
Q. What happens when you compile and run the following code?

```java
public class Cubes4 {

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

    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));
    }
}
```
Q. What happens when you compile and run the following code?

```java
public class Cubes5 {
    public static int cube(int i) {
        return i * i * i;
    }

    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));
    }
}
```