Lecture 17
Last time

• Define and invoke (call) a method.
void Method Example

```java
public class TestVoidMethod {
    public static void main(String[] args) {
        System.out.print("The grade is ");
        printGrade(78.5);
        System.out.print("The grade is ");
        printGrade(59.5);
    }

    public static void printGrade(double score) {
        if (score >= 90.0) {
            System.out.println('A');
        } else if (score >= 80.0) {
            System.out.println('B');
        } else if (score >= 70.0) {
            System.out.println('C');
        } else if (score >= 60.0) {
            System.out.println('D');
        } else {
            System.out.println('F');
        }
    }
}
```
Passing Parameters by Values

- When you invoke a method with an argument, the value of the argument is passed to the parameter. This is referred to as \textit{pass-by-value}. If the argument is a variable rather than a literal value, the value of the variable is passed to the parameter. The variable is not affected, regardless of the changes made to the parameter inside the method.

- Passing Parameters by reference
Local variables

- A local variable is used to store information that is relevant for the duration of the execution of one method.
- A local variable will exist as long as the method in which they have been created is still running.
- As soon as the method terminates (i.e., returns), all the local variables defined inside the method are destroyed.
Defining local variables

- How to define (and recognize) a local variable:

  - A **local variable** is *defined* inside the **body** of a method

```
public class SomeClass
{
    public static void SomeMethod( ...) {
        double x;
        ...
        int y;
    }
}
```

(I.e., a local variable is *defined* between the opening and closing braces of a method)
The life time of a *local variable*

- A local variable is *created* at the moment that the *program execution* reaches the *definition* of the local variable
- A local variable is *destroyed* at the moment that the *program execution* reaches the *end* of the method in which the local variable is defined
The Scope of Variables

<table>
<thead>
<tr>
<th>Program &quot;Scope1&quot;</th>
<th>Program &quot;Scope2&quot;</th>
</tr>
</thead>
<tbody>
<tr>
<td>public class Scope1</td>
<td></td>
</tr>
<tr>
<td>{</td>
<td></td>
</tr>
<tr>
<td>public static void main(String[] args)</td>
<td></td>
</tr>
<tr>
<td>{</td>
<td></td>
</tr>
<tr>
<td>double r;</td>
<td></td>
</tr>
<tr>
<td>r = 3.14;</td>
<td></td>
</tr>
<tr>
<td>System.out.println(r);</td>
<td></td>
</tr>
<tr>
<td>}</td>
<td></td>
</tr>
<tr>
<td>}</td>
<td></td>
</tr>
<tr>
<td>public class Scope2</td>
<td></td>
</tr>
<tr>
<td>{</td>
<td></td>
</tr>
<tr>
<td>public static void main(String[] args)</td>
<td></td>
</tr>
<tr>
<td>{</td>
<td></td>
</tr>
<tr>
<td>double r;</td>
<td></td>
</tr>
<tr>
<td>r = 3.14;</td>
<td></td>
</tr>
<tr>
<td>}</td>
<td></td>
</tr>
<tr>
<td>System.out.println(r);</td>
<td></td>
</tr>
<tr>
<td>}</td>
<td></td>
</tr>
</tbody>
</table>
• **Scope** of a *variable* = the *region* in the *program* where the *variable* is *accessible*

• **Factors** that determine the *scope* of a *variable*:

  – *When* is the *variable* created (a *variable* must *exist* before it can be *accessible*)
  
  – *When* is the *variable* destroyed (a *variable* that has been *destroyed* is *not accessible*)
  
  – Between the 2 location above, the *scope* of a *variable* is *further limited* by the *block* in which it is *defined*
To avoid confusion, Java (in fact, every programming language) imposes the following rule:

- You cannot define different variables with the same name inside the same scope.

Example: this is not allowed

```java
public class Scope7 {
    public static void main(String[] args) {
        /* -----------------------------------------------
         2 variables named r inside the SAME scope
        ----------------------------------------------- */
        double r = 0.0;
        int r = 0; // Error!
        System.out.println(r);
    }
}
```
Special scope configurations

- A scope are delimited by a pair of (matching) braces `{ .... }`

- Configuration 1: non-overlapping

- Configuration 2: nested
You can **define different** variables with the **same** name in **disjoint** scopes

```java
public class Scope4 {
    public static void main(String[] args) {
        {
            double r;
            ...
        }
        ...
        {
            String r;
            ...
        }
    }
}
```
Nested Scopes

- Nested scopes = 2 or more scopes where one scope is contained in the other

Example: the following 2 blocks create a nested scopes

```java
public class Scope5 {
    public static void main(String[] args) {
        {
            ...
            {
                ...
                }
                ...
            }
            ...
        }
    }
}
```
A variable that is defined at location $x$ in an outer scope is accessible in all inner scopes following location $x$.

Example:

```java
public class Scope5 {
    public static void main(String[] args) {
        {
            double r;
            {
                inner scope
            }
            double t;
        }
    }
}
```
Variables with *same* name in *nested scopes*

Consider the following two variables definitions:

```java
public class Scope6 {
    public static void main(String[] args) {
        // Start of outer scope
        double r;
        // Start of inner scope
        String r;
        ...
    }
}
```

The first definition of the variable `r` takes place inside the *outer scope*.

The second definition of the variable with the *same name* `r` takes place inside the *inner scope*.

The designer of the Java programming language believes that this is confusing and *has decided* that this construct is not allowed in Java.

Other programming languages (C/C++) allow it.
It is fine to declare `i` in two nonnested blocks.

```java
public static void method1() {
    int x = 1;
    int y = 1;

    for (int i = 1; i < 10; i++) {
        x += i;
    }

    for (int i = 1; i < 10; i++) {
        y += i;
    }
}
```

It is wrong to declare `i` in two nested blocks.

```java
public static void method2() {
    int i = 1;
    int sum = 0;

    for (int i = 1; i < 10; i++) {
        sum += i;
    }
}
```
○ What will be printed by each of the print statements:

```java
public class Exercise1
{
    public static void main(String[] args)
    {
        {  
            double r = 3.14;
            
            {  
                String s = "1234";
                r = r + 2;
                System.out.println( s + r );
            }
            
            {  
                int s = 1234;
                System.out.println( s + r );
            }
        }
    }
}
```
Class variables

```java
public class SomeClass {
    public static int a;
    public static void SomeMethod( ... )
    {
        ...
    }

    public static double b;
    public static void SomeMethod( ... )
    {
        ...
    }

    public static double[] c;
}
```
Accessing a class variable

- A **class variable** is **accessed** using its **complete name**: `nameOfClass.nameOfClassVariable`

- **Example:**

```java
public class ClassVar1 {
    public static double a; // <----- Class variable
    
    public static void main(String[] args) {
        // Body of method "main"
        ClassVar1.a = 3.1415; // Accessing a class variable
        System.out.println(a);
    }
}
```
**Short hand** notation for class variables

- We can *invoke* methods from another method *inside the same class without* using the class name.
- We can *reference* a class variable defined *inside the same class without* using the class name.

```java
public class ClassVar2 {
    public static double a; // <------ Class variable

    public static void main(String[] args) {
        // Body of method "main"
        a = 3.1415; // We can omit the classname in this method
        System.out.println(a);
    }
}
```
The *life time* and *scope* of (public) class variables

- **Life time of class variables:**
  - Class variables are *created* when the Java program begins its execution
  - Class variables are *destroyed* when the Java program terminates

In other words:

- Class variables exist for the entire execution of the Java program
  (That's why they are used to store long term information)

- **Scope of class variables:**
  - Class variables are *accessible* in every method of every class

In other words:

- (Public) class variables can be used anywhere in any Java program
Shadowing a class variable

• When the method has a local variable or a parameter variable defined in a scope whose name is equal to the name of the class variable, then:

• The class variable with the same name can no longer accessible with the short hand notation in that scope !!!
public class ClassVar4
{
    public static double a = 3.1415;

    public static void main(String[] args)
    {
        System.out.println( a );  // prints 3.1415

        String a = "abc";    // Class var a is not accessible
        System.out.println( a ); // inside inner scope
        System.out.println( ClassVar4.a ); // prints 3.1415

        System.out.println( a );  // prints 3.1415

        boolean a = true;        // Class var a is not accessible
        System.out.println( a ); // in main any longer !!!
        System.out.println( ClassVar4.a ); // prints 3.1415
    }
}