Lecture 16
Method

1. **First**, you must **define** the **method**.
   
   How to **define a method**:
   
   - **Write down** the steps (= statements) contained in the **method**
   - **Attach** a **name** to the steps (= statements)

   **Notes:**
   
   - You **only** need to **define** a method **once**
     
     (Remember that in **Java**, you must **define the method inside** some **class**.)

2. After **defining the method**, you can then **call a method** using the **name of the method**
   
   - When a **method** is **called**, the **statements** inside the corresponding method are **executed**
   - When all **statements** in the method has been **executed**, the **execution** will **resume** at the **program location of the method call**

   This **mechanism** is called **method invocation** (an older term is **procedure call**)

   **Note:**
   
   - You can **invoke** a method **as many times as you wish**
The 2 types (kinds) of methods in Java

- **Older** (non-object-oriented) **programming languages** have only *one type (kind) of method*
- **Newer programming languages** (so called *object-oriented* languages, explained later) have *2 types (kinds) of methods:*

  1. **Class methods**
  2. **Instance methods**
Example: defining a method

```java
public class ToolBox  // Methods must be contained inside a class
{
    /* ---------------------------------------------
       A method called "min" that contains
       the statements to find the smaller
       of 2 values a and b
       --------------------------------------------- */
    public static double min ( double a, double b )
    {
        double m = 0;
        if ( a < b )
        {
            m = a;  // a is the smaller value
        }
        else
        {
            m = b;  // b is the smaller value
        }
        return(m);  // Output of the method
    }
    ...
    (You can define more methods inside a class)
}
```
Example: using the method `ToolBox.min`:

```java
public class MyProgram {
    public static void main(String[] args) {
        double r;
        r = ToolBox.min(1.0, 4.0);
        System.out.println(r);
        r = ToolBox.min(3.7, -2.9);
        System.out.println(r);
        r = ToolBox.min(-9.9, 3.8);
        System.out.println(r);
    }
}
```

Notice the advantage of using methods:

- A non-savvy user that wants to use the `ToolBox.min` method does not need to know the statements contained inside the method `ToolBox.min` !!

- A non-savvy user will only need to know the following in order to use the method:
  
  1. The (complete) name of the method (i.e.: `ToolBox.min`)
  2. What information the method needs to do the task (i.e.: 2 numbers)
  3. What information the method returns to the user (i.e.: 1 number)

- In fact, you were a non-savvy user of the methods in Java's scanner class

  E.g.: You have used `nextDouble()` without knowing exactly what statements this method contains!
Program starts executing in the main method:

```java
public class MyProgram {
    public static void main(String[] args) {
        double r;
        r = Toolbox.min(1.0, 4.0);
        ....
    }
}
```

```
public class Toolbox {
    public static double min(double a, double b) {
        double m = 0;
        if (a < b) {
            m = a;
        } else {
            m = b;
        }
        return m;
    }
}
```
The **method invocation** `ToolBox.min(1.0, 4.0)` **transfers** the **program execution** to the **method** `ToolBox.min`

```java
public class MyProgram {
    public static void main(String[] args) {
        double r;
        r = ToolBox.min(1.0, 4.0);
        ....
    }
}

public class ToolBox {
    public static double min(double a, double b) {
        double m = 0;
        if (a < b) {
            m = a;
        } else {
            m = b;
        }
        return m;
    }
}
```
The `min` method executes and reaches the `return` statement.

```java
public class MyProgram {
    public static void main(String[] args) {
        double r;
        r = Toolbox.min(1.0, 4.0);
        ....
    }
}

public class Toolbox {
    public static double min(double a, double b) {
        double m = 0;
        if (a < b) {
            m = a;
        } else {
            m = b;
        }
        return m;
    }
}
```

**Finds min of a and b**

```
\text{Finds min of a and b}
```

```
a = 1.0, b = 4.0
```

```
m = 1.0
```
The `return` statement transfers program execution back to the point of invocation.

```java
public class MyProgram {
    public static void main(String[] args) {
        double r;
        r = Toolbox.min(1.0, 4.0); // 1.0
        ....
    }
}
```

```java
public class Toolbox {
    public static double min(double a, double b) {
        double m = 0;
        if (a < b) {
            m = a;
        } else {
            m = b;
        }
        return(m);
    }
}
```
Defining a (class) method

- Syntax used to define a method:

  ```java
  public static RETURN-TYPE method-Name ( FORMAL-PARAMETER-LIST )
  {
      (variables definitions and/or statements)
  }
  ```

  **Note:** the construct *must* appear *inside some class*, so it will look like this:

  ```java
  public class SomeClassName
  {
      ...
      public static RETURN-TYPE method-Name ( FORMAL-PARAMETER-LIST )
      {
          (variables definitions and/or statements)
      }
      ...
  }
  ```
- The **keyword public** tells the Java compiler that we are defining *something* that had **no access restriction**

  The "something" can be one of 4 things:

  - A **method**, or
  - A **variable**
  - A **constant** (final)
  - A **class** (will **not** be discussed in this course)

  In other words, you can **only** define these 4 **kinds of things** inside a **class**

- The **keyword static** tells the Java compiler that we are defining a **class typed** method (or variable).

  **Note:**

  - If you **omit** the keyword **static**, you will define an instance method or variable or constant which will be discussed later.

- The **return-type** is a Java data type (e.g., `int`, `double`, etc.)

  - The **return-type** specifies the **type of the data** that is **returned** by the method.

- The **method-Name** is an **identifier** that is used to **identify this method**.

  - You use the **method-Name** to **identify the method** in the **method invocation**
The pair of brackets (....) tells the Java compiler that you want to define a method.

- You can in fact define 2 different things inside a class:
  - methods
  - variables

We will discuss variables defined inside a class later in the course.

- A definition without the brackets (....) will define a variable !!!
- Notice how the Java compiler can tell the difference:

```java
public class SomeClassName {
    ...
    public static double min;
    public static double min(double a, double b) {
        ...
    }
    ...
}
```

No brackets: defines a variable

With brackets: defines a method
The **FORMAL-PARAMETER-LIST** is a *comma-separated* list of *parameter variables* that is **passed** (= given to) the method as **additional information**.

- The **items** in the **FORMAL-PARAMETER-LIST** has the following form:
  
  ```
  TYPE variable-name
  ```

- A **formal parameter variable** is **initialized** with the **value given in the method call**.

  It is a way to **pass information** to the **method** so it can use the **information** to **perform its designated task**.

- The **method body** completes the **method definition**.

  The **method body** is a **block** (enclosed between `{ ... }`)

  Within the **method body**, you can write any number of the following things:

  - **statements**
  - **variable definitions** --- variables defined **inside a method body** are called **local variables**
The **method header** is the **first part** of the **method definition** without the method body.

```java
public class SomeClassName {
    ...

    public static RETURN-TYPE method-Name ( FORMAL-PARAMETER-LIST ) {
        (variables definitions and/or statements)
    }

    ...
```

Method header
Example:

```
public class ToolBox {

    public static double min(double a, double b) {
        double m = 0;
        if (a < b) {
            m = a;
        } else {
            m = b;
        }
        return m;
    }
}
```

- **Method Header**: public static double min(double a, double b)
- **Return Type**: double
- **Method Name**: min
- **2 Formal Parameters**: a, b
- **Definition of a Local Variable**: m
- **Statement (an if-else statement)**:
  - if (a < b)
  - else
- **2 Kinds of Things in Method Body**:
  1. Local variable definition
  2. (Program) Statement

The diagram illustrates the structure and components of a method definition in a class, highlighting the roles of different elements within the code.
Multiple methods with the same name in the same class

• When you use/invoke a method in your Java program, the Java compiler will use the following information to identify which method you want to use:

  – The method name (i.e., ClassName.MethodName)
  – The number and the type of the parameters that you specify in the method invocation.
Example: suppose we have the following 2 methods named `min` in the same class:

```java
public class ToolBox { // contains 2 min methods
    /* -----------------------------------------------
       min with 2 double formal parameters
       ----------------------------------------------- */
    public static double min ( double a, double b )
    {
        double m = 0;
        System.out.println("You are using min1");
        if ( a < b )
            m = a;
        else
            m = b;
        return(m);
    }

    /* -----------------------------------------------
       min with 2 int formal parameters
       ----------------------------------------------- */
    public static int min ( int a, int b )
    {
        int m = 0;
        System.out.println("*** You are using min2 !!!");
        if ( a < b )
            m = a;
        else
            m = b;
        return(m);
    }
}
```
The different *kinds* of variables in a Java program

- Different kinds of *information*
  - Computer program needs *information* to operate correctly
    - The *information* used by computer programs are *stored* inside *variables*
  - We can *divide* the *information* broadly into:
    - *Short term information*
    - *Long(er) term information*

As a result, computer programs use *different kinds of variables* specialized to store *short term* and *long term information*
The different kinds of *variables* in a Java program

- **Java** has 4 different kinds of *variables* (See: [click here](#))
  
  - Class variables
  - Instance variables
  - Local variables
  - Parameter variables

General information on *variables*:

- Each kind of *variable* has its own characteristics *(properties)*

- The kind of *variable* is determined by:
  - *Where* the *variable* is defined
  - Whether the keyword *static* was *used* in the *variable* definition.
• *Kind and Type*

○ I deliberately used the term "*kind* of variable" to distinguish this *property* from the "*type* of the variable"

○ *Kind* and type:

- The *type* of a *variable* determines the *encoding method used* to *interpret* the information stored inside a *variable*

- The *kind* of a *variable* determines the *life time* and *scope* of a *variable*

The *life time* and *scope* of variables will be *discussed later*.

What I want to *make clear* here is:

- *kind of variable ≠ type of variable*
Properties of variables

- Every variable has 2 properties:
  - **lifetime** = the **duration** that a variable **exists**
  - **scope** = the **region in the program** where the variable is **accessible (can be used)**
Why do programs have different kinds of variables

- The reason is to accommodate different needs:
  - **Short term** information:
    - Some information need *only* be retained *within a (one) method*
      (We call this *kind* of information *local information*)
  - **Long term** information:
    - Some information must be retained *across several methods*

- *Local variables* and *parameter variables* are used to store *local* (or short term) information
- *Class variables* and *instance variables* are used to store *longer term* information