Introduction to Computer Programming (CS102A)

Lecture 6: Methods: A Deeper Look

Yuxin Ma

Department of Computer Science and Engineering
Southern University of Science and Technology

Objectives

- Modular Programming
- How to Use Methods
- Method-call Stack (Program Execution Stack)
- Method Overloading

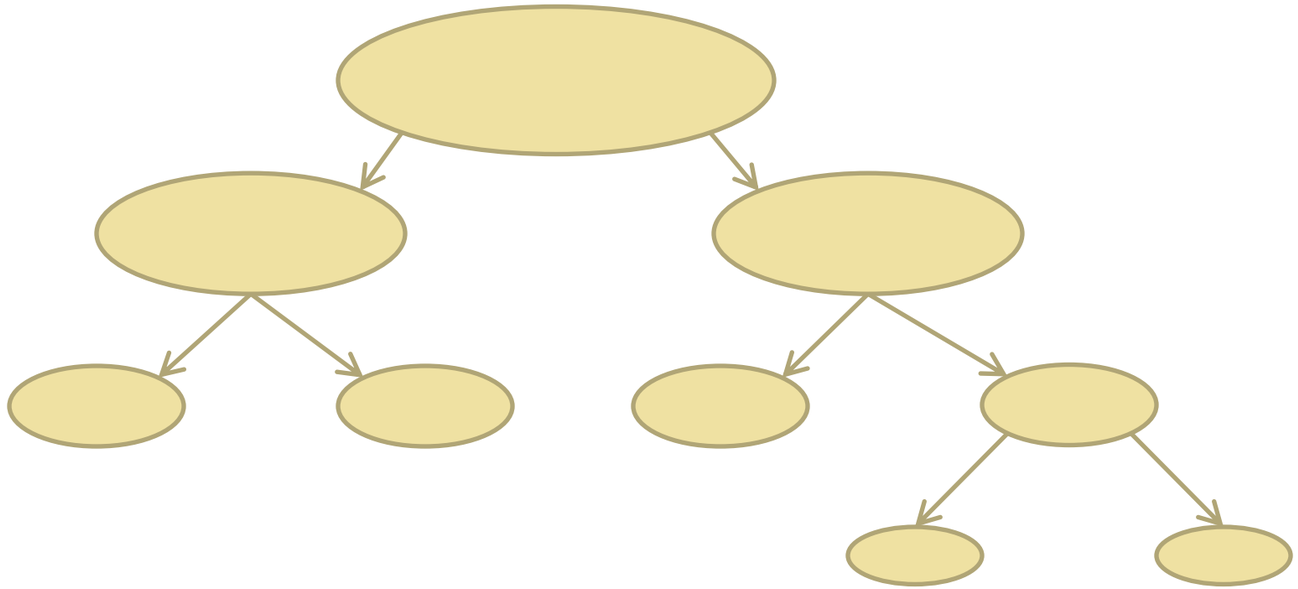
Problem Solving

- The programs we have written so far only solve simple problems (find the maximum value in an array of numbers)
  - They are short and everything fits well in a `main` method

- What if you are asked to solve complex problems, e.g., building a climate model from big data?
  - A giant main method?

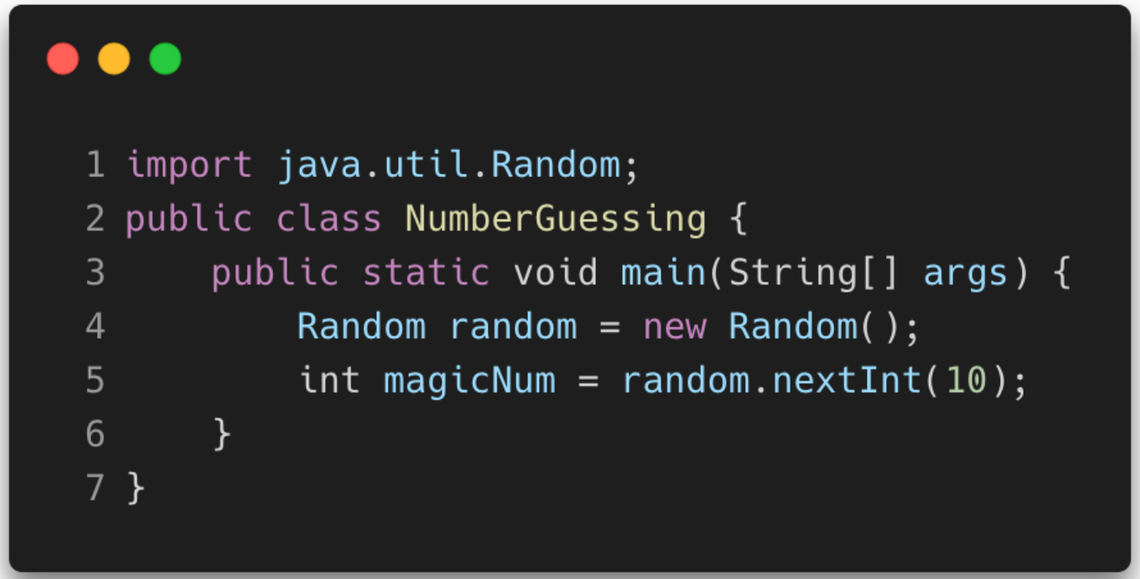
Divide and Conquer

- Decompose a big/complex task into smaller one and solve each of them



Methods

- Methods facilitate the design, implementation, operation and maintenance of large programs
  - E.g., `random.nextInt(10)`: calling a method to generate a random number
  - ! (We don't need to know how random numbers are generated)



```java
import java.util.Random;
public class NumberGuessing {
    public static void main(String[] args) {
        Random random = new Random();
        int magicNum = random.nextInt(10);
    }
}
```

Why Use Methods?

- For reusable code, reducing code duplication
  - If you need to do the same thing many times, write a method to do it, then call the method each time you have to do that task.
- To parameterize code
  - You will often use parameters that change the way the method works.
- For top-down programming (divide and conquer)
  - You solve a big problem (the "top") by breaking it down into small problems. To do this in a program, you write a method for solving your big problem by calling other methods to solve the smaller parts of the problem, which similarly call other methods until you get down to simple methods which solve simple problems.

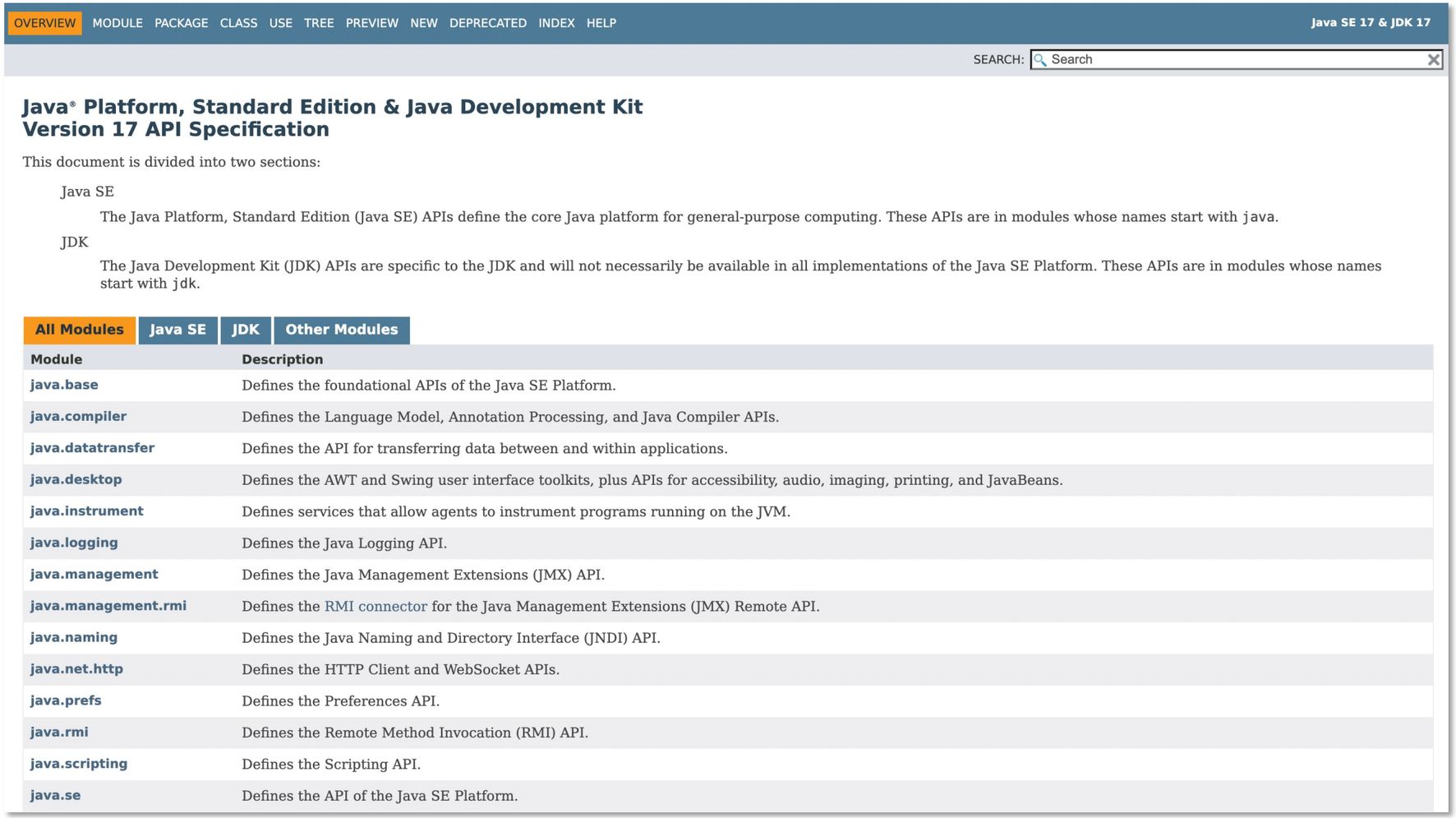
Why Use Methods?

- To create conceptual units
  - Create methods to do something that is one action in your mental view of the problem. This will make it much easier for you to program.
- To simplify
  - Because local variables and statements of a method can not been seen from outside the method, they (and their complexity) are hidden from other parts of the program, which prevents accidental errors or confusion (e.g., random number generation method)
- To ease debugging and maintenance
  - You don’t want to debug a main method with 100K lines of code

Program Modules in Java

- Java programs are written by combining new methods and classes that you write with predefined methods and classes available in the *Java Application Programming Interface (Java API)* and in various other libraries
  - Related classes are typically grouped into packages so that they can be imported into programs and reused
  - The Java API provides a rich collection of predefined classes, e.g.,
    - `java.util.Scanner`
    - `java.lang.Math`

Java API Documentation



OVERVIEW MODULE PACKAGE CLASS USE TREE PREVIEW NEW DEPRECATED INDEX HELP — Java SE 17 & JDK 17

SEARCH: Search

Java® Platform, Standard Edition & Java Development Kit Version 17 API Specification

This document is divided into two sections:

Java SE

The Java Platform, Standard Edition (Java SE) APIs define the core Java platform for general-purpose computing. These APIs are in modules whose names start with `java`.

JDK

The Java Development Kit (JDK) APIs are specific to the JDK and will not necessarily be available in all implementations of the Java SE Platform. These APIs are in modules whose names start with `jdk`.

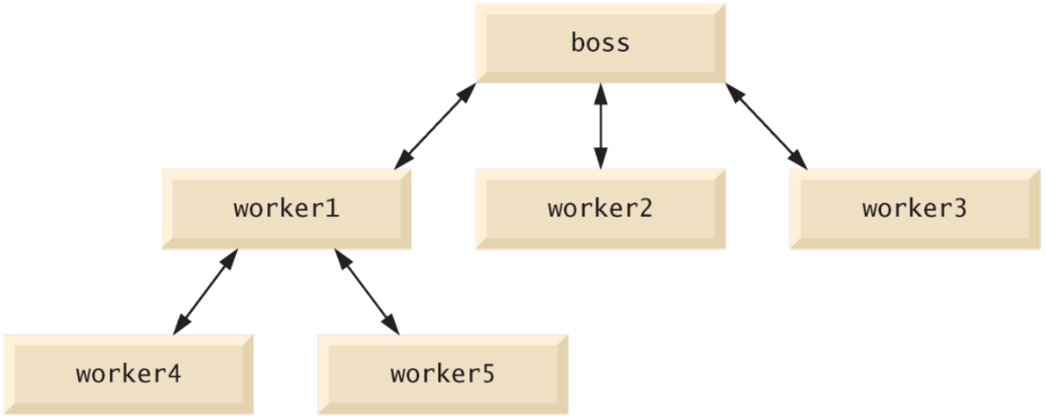
All Modules | Java SE | JDK | Other Modules

Module	Description
java.base	Defines the foundational APIs of the Java SE Platform.
java.compiler	Defines the Language Model, Annotation Processing, and Java Compiler APIs.
java.datatransfer	Defines the API for transferring data between and within applications.
java.desktop	Defines the AWT and Swing user interface toolkits, plus APIs for accessibility, audio, imaging, printing, and JavaBeans.
java.instrument	Defines services that allow agents to instrument programs running on the JVM.
java.logging	Defines the Java Logging API.
java.management	Defines the Java Management Extensions (JMX) API.
java.management.rmi	Defines the RMI connector for the Java Management Extensions (JMX) Remote API.
java.naming	Defines the Java Naming and Directory Interface (JNDI) API.
java.net.http	Defines the HTTP Client and WebSocket APIs.
java.prefs	Defines the Preferences API.
java.rmi	Defines the Remote Method Invocation (RMI) API.
java.scripting	Defines the Scripting API.
java.se	Defines the API of the Java SE Platform.

https://docs.oracle.com/en/java/javase/17/docs/api/index.html

Program Modules in Java

- Similar to the hierarchical form of management
  - A boss (the caller) asks a worker (the callee) to perform a task and report back (return) the results after completing the task
  - The boss method does not know how the worker method performs its designated tasks (method complexity is hidden)
  - The worker may also call other worker methods, unknown to the boss



`static` Methods

- Sometimes a method performs a task that does not depend on the contents of any object
  - Known as a static method or a class method
  - Place the keyword `static` before the return type in the declaration
  - Called via the class name and a dot (`.`) separator

`static` Methods

- Many more useful static methods in `java.lang.Math` class:

Method	Description	Example

`abs(` *x* `)`	absolute value of *x*	`abs( 23.7 )` is `23.7`
		`abs( 0.0 )` is `0.0`
		`abs( -23.7 )` is `23.7`

`ceil(` *x* `)`	rounds *x* to the smallest integer not less than *x*	`ceil( 9.2 )` is `10.0` `ceil( -9.8 )` is `-9.0`
`cos(` *x* `)`	trigonometric cosine of *x* (*x* in radians)	`cos( 0.0 )` is `1.0`
`exp(` *x* `)`	exponential method e^x	`exp( 1.0 )` is `2.71828` `exp( 2.0 )` is `7.38906`
`floor(` *x* `)`	rounds *x* to the largest integer not greater than *x*	`floor( 9.2 )` is `9.0` `floor( -9.8 )` is `-10.0`
`log(` *x* `)`	natural logarithm of *x* (base *e*)	`log( Math.E )` is `1.0` `log( Math.E * Math.E )` is `2.0`
`max(` *x*, *y* `)`	larger value of *x* and *y*	`max( 2.3, 12.7 )` is `12.7` `max( -2.3, -12.7 )` is `-2.3`
`min(` *x*, *y* `)`	smaller value of *x* and *y*	`min( 2.3, 12.7 )` is `2.3` `min( -2.3, -12.7 )` is `-12.7`
`pow(` *x*, *y* `)`	*x* raised to the power *y* (i.e., x^y)	`pow( 2.0, 7.0 )` is `128.0` `pow( 9.0, 0.5 )` is `3.0`
`sin(` *x* `)`	trigonometric sine of *x* (*x* in radians)	`sin( 0.0 )` is `0.0`
`sqrt(` *x* `)`	square root of *x*	`sqrt( 900.0 )` is `30.0`
`tan(` *x* `)`	trigonometric tangent of *x* (*x* in radians)	`tan( 0.0 )` is `0.0`

`static` Fields and Class `Math`

- Class Math declares commonly used mathematical constants
  - `Math.PI` (3.141592653589793)
  - `Math.E` (2.718281828459045) is the base value for natural logarithms
- These fields are declared in class Math with the modifiers `public`, `final` and `static`
  - `public` allows you to use these fields in your own classes
  - `final` indicates a constant—value cannot change
  - `static` makes them accessible via the class name `Math` and a dot (.) separator
  - `static` fields are also known as class variables (in contrast to instance variables)

Why `main` Method Has to be `static`?

- When you execute the Java Virtual Machine (JVM) with the java command, the JVM attempts to invoke the `main` method of the class you specify.
- Declaring `main` as `static` allows the JVM to invoke `main` without creating an object of the class

* You may have a deeper understanding when we introduce Classes and Objects later

Declaring Methods

- Two `static` methods are defined
  - `main`
  - `maximum`

```java
import java.util.Scanner;

public class MaximumFinder {
    public static void main(String[] args) {
        Scanner input = new Scanner(System.in);
        System.out.print("enter three floating-point values: ");
        double number1 = input.nextDouble();
        double number2 = input.nextDouble();
        double number3 = input.nextDouble();
        double result = maximum(number1, number2, number3);
        System.out.println("max is " + result);
    }

    public static double maximum(double x, double y, double z) {
        double max = x;
        if (y > max) max = y;
        if (z > max) max = z;
        return max;
    }
}
```

Details of Methods

```
14      public static double maximum(double x, double y, double z) {
15          double max = x;
16          if (y > max) max = y;
17          if (z > max) max = z;
18          return max;
19      }
```

- Find the largest of the 3 double values

```
10          double result = maximum(number1, number2, number3);
```

  - You need to call it explicitly to tell it to perform its task
  - Method don’t get called automatically after declaration
  - `static` methods in the same class can call each directly

Details of Methods

```
10          double result = maximum(number1, number2, number3);
```

- A method call supplies arguments for each of the method's parameters
  - One to one correspondence
  - The types must be consistent

```
1 Math.pow(Math.pow(x2 - x1, 2) + Math.pow(y2 - y1, 2), 0.5);
```

- Expressions and method calls in the arguments
  - Before any method can be called, its arguments must be evaluated to determine their values
  - If an argument is a method call, the method call must be performed to determine its return value

Details of Methods

```
14    public static double maximum(double x, double y, double z) {
15        double max = x;
16        if (y > max) max = y;
17        if (z > max) max = z;
18        return max;
19    }
```

- Return type: the type of data the method returns to its caller
  - `void` means returning nothing
- The method name follows the return type
  - Naming convention: `lowerCamelCase`
- A comma-separated list of parameters mean that the method requires additional information from the caller to perform its task.
  - Each parameter must specify a type and an identifier
  - A method's parameters are local variables of that method and can be used only in that method's body

Details of Methods

```
14    public static double maximum(double x, double y, double z) {
15        double max = x;
16        if (y > max) max = y;
17        if (z > max) max = z;
18        return max;
19    }
```

- Method header = modifiers + return type + method name + parameters
- Method body contains one or more statements that perform the method's task
- The `return` statement returns a value (or just control) to the point in the program from which the method is called.
  - It is good to have "`return`;" for a method with a return type `void`. This means that the method terminates without returning data.

Returning Results

- If the method does not return a result, control returns when the program flow reaches the method-ending right brace
  - Or when the statement `return`; executes

- If the method returns a result, the statement
  - `return expression;`

  evaluates the expression, and then returns the result to the caller

Method-Call Stack

- Stack data structure: analogous to a pile of dishes
  - When a dish is placed on the pile, it's normally placed at the top (referred to as pushing onto the stack)
  - Similarly, when a dish is removed from the pile, it's always removed from the top (referred to as popping off the stack)
- Last-in, first-out (LIFO) — the last item pushed (inserted) on the stack is the first item popped (removed) from the stack

Method-Call Stack

- When a program calls a method, the called method must know how to return to its caller, so the return address of the calling method is pushed onto the method-call stack (also known as program execution stack)
- If a series of method calls occurs, the successive return addresses are pushed onto the stack in last-in, first-out order
- The program-execution stack also contains the memory for the local variables used in each invocation of a method
  - Stored in the activation record (or stack frame) of the method call
  - When a method call is made, the activation record for that method call is pushed onto the method-call stack
- When a method returns to its caller, the activation record for the method call is popped off the stack and those local variables are no longer known to the program

Passing Arguments in Method Calls

- Typically, two ways: pass-by-value and pass-by-reference

- When an argument is passed by value, a copy of the argument's value is passed to the called method
  - The called method works exclusively with the copy
  - Changes to the copy do not affect the original variable's value in the caller

- When an argument is passed by reference, the called method can directly access the argument's value in the caller and modify that data, if necessary.
  - Improves performance by avoiding copying possibly large amounts of data.

Pass-by-value in Java

- In Java, all arguments are passed by value
- A method call can pass two types of values to the called method: copies of primitive values and copies of references to objects
- Although an object's reference is passed by value, a method can still interact with the referenced object using the copy of the object's reference (arrays are also objects)
  - The parameter in the called method and the argument in the calling method refer to the same object in memory

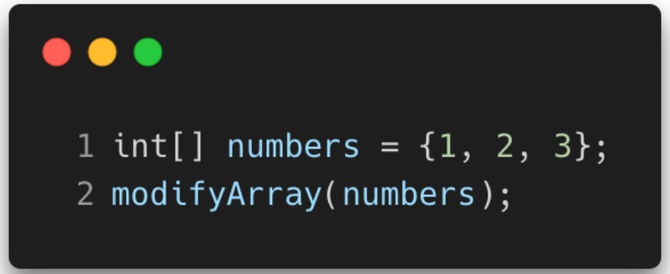
Examples

```java
public class PassingByValue {
    public static void main(String[] args) {
        int a = 3;
        System.out.println("Before: " + a);
        triple(a);
        System.out.println("After: " + a);
    }

    public static void triple(int x) {
        x *= 3;
    }
}
```

```java
public class PassingByReference {
    public static void main(String[] args) {
        int[] a = {1, 2, 3};
        System.out.print("Before: ");
        for (int value : a) {
            System.out.printf("%d ", value);
        }
        triple(a);
        System.out.print("\nAfter: ");
        for (int value : a) {
            System.out.printf("%d ", value);
        }
    }

    public static void triple(int[] x) {
        for (int i = 0; i < x.length; i++)
            x[i] *= 3;
    }
}
```

Passing Arrays to Methods

- To pass an array argument to a method, specify the name of the array without any brackets



```
1 int[] numbers = {1, 2, 3};
2 modifyArray(numbers);
```

- When we pass an array object's reference into a method, we don't need to pass the array length as an additional argument because every array knows its own length
- For a method to receive an array reference through a method call, the method's parameter list must specify an array parameter

Passing Arrays to Methods

- When a method argument is an entire array or an array element of a reference type, the called method receives a copy of the reference

```
1 int[] numbers = {1, 2, 3};
2 modifyArray(numbers);
```

Using Command-line Arguments: Revisited

- It's possible to pass arguments from the command line (these are known as command-line arguments) to an application by including a parameter of type `String[]` in the parameter list of `main`

```
1 public static void main(String[] args) {
2     ...
3 }
```

- By convention, this parameter is named `args`
- When an application is executed using the `java` command, Java passes the command-line arguments that appear after the class name in the java command to the application's `main` method as `String`s in the array `args`

Using Command-line Arguments: Revisited

```java
// Initializing an array using command-line arguments.
public class InitArray {
    public static void main(String[] args) {
        // check number of command-line arguments
        if ( args.length != 3 )
            System.out.println(
                "Error: Please re-enter the entire command, including\n" +
                "an array size, initial value and increment."
            );
        else {
            // get array size from first command-line argument
            int arrayLength = Integer.parseInt(args[0]);
            int[] array = new int[arrayLength]; // create array
            // get initial value and increment from command-line arguments
            int initialValue = Integer.parseInt(args[1]);
            int increment = Integer.parseInt(args[2]);
            // calculate value for each array element
            for (int counter = 0; counter < array.length; counter++)
                array[counter] = initialValue + increment * counter;
            System.out.printf("%s%8s\n", "Index", "Value");
            // display array index and value
            for (int counter = 0; counter < array.length; counter++)
                System.out.printf("%5d%8d\n", counter, array[ counter ]);
        }
    }
}
```

```
> java InitArray
Error: Please re-enter the entire command,
including an array size, initial value and
increment.
```

```
> java InitArray 5 0 4
Index Value
    0     0
    1     4
    2     8
    3    12
    4    16
```

Variable-length Argument Lists

- With variable-length argument lists, you can create methods that receive an unspecified number of arguments
- A type followed by an ellipsis (. . .) in a method's parameter list indicates that the method receives a variable number of arguments of that particular type

```
1 public static double average(double... numbers) {
```

  - Can occur only once in a parameter list, and the ellipsis, together with its type, must be placed at the end of the parameter list
- Java treats the variable-length argument list as an array of the specified type

Example

```java
public class VariableLengthArgumentList {
    public static double average(double... numbers) {
        double total = 0.0;
        for (double d : numbers)
            total += d;
        return total / numbers.length;
    }

    public static void main(String[] args) {
        double d1 = 10.0, d2 = 20.0, d3 = 30.0;
        System.out.printf("average of d1 and d2: %f\n", average(d1, d2));
        System.out.printf("average of d1 ~ d3: %f\n", average(d1, d2, d3));
    }
}
```

```
average of d1 and d2: 15.000000
average of d1 ~ d3: 20.000000
```

Argument Promotion

- Argument promotion: Converting an argument's value, if possible, to the type that the method expects to receive in its corresponding parameter
  - `Math.sqrt()` expects to receives a double argument, but it is ok to write
  - `Math.sqrt(4)` : java converts the `int` value 4 to the `double` value 4.0

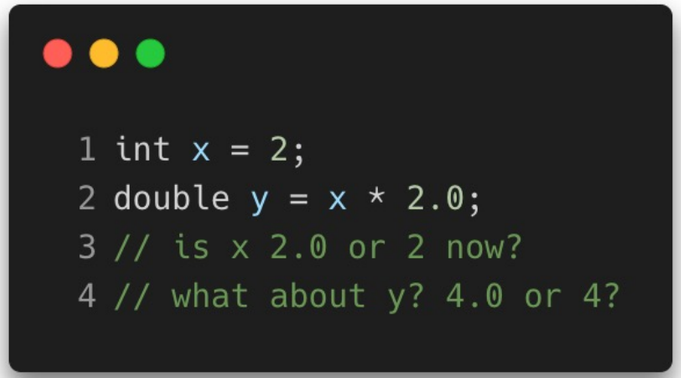
Promotion Rules

- Specify which conversions are allowed (which conversions can be performed without losing data)

Type	Valid promotions
`double`	None
`float`	`double`
`long`	`float` or `double`
`int`	`long`, `float` or `double`
`char`	`int`, `long`, `float` or `double`
`short`	`int`, `long`, `float` or `double` (but not `char`)
`byte`	`short`, `int`, `long`, `float` or `double` (but not `char`)
`boolean`	None (`boolean` values are not considered to be numbers in Java)

Promotion Rules

- Besides arguments passed to methods, the rules also apply to expressions containing values of two or more primitive types
  - In the following case, `2 * 2.0` becomes `4.0`



```
1 int x = 2;
2 double y = x * 2.0;
3 // is x 2.0 or 2 now?
4 // what about y? 4.0 or 4?
```

** x is still of `int` type, the expression uses a temporary copy of x's value for promotion

Method Overloading

- Methods of the same name can be declared in the same class, as long as they have different sets of parameters
- Used to create several methods that perform the same/similar tasks on different types or different numbers of arguments
  - Java compiler selects the appropriate method to call by examining the number, types and order of the arguments in the call

Method Overloading

- Compiler distinguishes overloaded methods by their signature
  - A combination of the method's name and the number, types and order of its parameters.

```
1 public double calculateAnswer(double wingSpan, int numberOfEngines,
2       double length, double grossTons) {
3
4     //do the calculation here
5     return 0.0;
6 }
7
8 // Signature: calculateAnswer(double, int, double, double)
```

Method Overloading

- Method calls cannot be distinguished by return type. If you have overloaded methods only with different return types:
  - `int square(int a)`
  - `double square(int a)`

and you called the method by `square(2);`, the compiler will be confused (since return value ignored).

Method Overloading

```java
// Overloaded method declarations.
public class MethodOverload {
    // square method with int argument
    public static int square(int intValue) {
        System.out.printf("%nCalled square with int argument: %d%n", intValue
        );
        return intValue * intValue;
    }

    // square method with double argument
    public static double square(double doubleValue) {
        System.out.printf("%nCalled square with double argument: %f%n",
                doubleValue);
        return doubleValue * doubleValue;
    }

    // test overloaded square methods
    public static void main(String[] args) {
        System.out.printf("Square of integer 7 is %d%n", square(7));
        System.out.printf("Square of double 7.5 is %f%n", square(7.5));
    }
} // end class MethodOverload
```

```
Called square with int argument: 7
Square of integer 7 is 49

Called square with double argument: 7.500000
Square of double 7.5 is 56.250000
```

Extra: Debugging

- Common methods
  - Use the print functions to display values in the terminal
  - Debugger in IDEA
    - Step through the program
    - “*Step over*”, “*Step into*”, and “*Step out*”