Module 07 — Methods

ITP 120 — Java Programming I

Northern Virginia Community College | Randy Michak

🎯 Learning Objectives

- Explain what a method is and why methods make programs easier to write and maintain
- Define both `void` and value-returning methods using correct Java syntax
- Call a method from `main` and trace the execution flow step by step
- Distinguish between formal parameters and actual arguments
- Explain pass-by-value and predict what happens to the caller's variables
- Write `return` statements and use returned values in expressions
- Overload a method by writing multiple versions with different parameter lists
- Explain variable scope and why two methods can use the same variable names without conflict
- Build a multi-method program that solves a realistic problem

📚 Key Terms

method · **void method** · **value-returning method** · **parameter** · **argument** · **return type** · **return statement** · **pass-by-value** · **method overloading** · **scope** · **local variable** · **access modifier**

1. What Are Methods?

A **method** is a named block of code that performs a specific task. You define it once and call it as many times as you need. That is the whole idea.

You have been using methods since day one: `main` is a method. So is `System.out.println()`. Every time you call `println`, you are handing some text to a block of code that knows how to write it to the screen. You did not write that block — Oracle did — but you use it constantly.

1.1 Why Methods Matter

Imagine writing a program that converts temperatures in five different places. Without methods, you copy-paste the same formula every time. When you find a bug, you fix it in five places. With a method, you fix it once and every call benefits.

Methods give you three things:

- **Code reuse** — write once, call many times
- **Organization** — each method has one job; the program is easier to navigate
- **Easier debugging** — a bug in a method is one bug to fix, not five

> 🧠 **Think of It This Way**
>
> A method is like a kitchen appliance. Your blender knows how to blend. You do not care about its internal motor — you just put ingredients in, press the button, and get a smoothie out. Methods work the same way: you pass in data (ingredients), the method does its work, and optionally hands something back (the result). One appliance, used a hundred times.

1.2 The main Method You Already Know

Every Java program starts at `main`. You have been writing it this way since Module 1:

```
public static void main(String[] args) {
    System.out.println("Hello from main!");
}
```

Every word in that signature means something. By the end of this module, you will understand all of it. For now, just recognize: `main` itself is a method. You have been defining and calling methods all along.

2. Defining a Method

Here is the general syntax for a method definition:

```
accessModifier  returnType  methodName(parameterList) {
    // method body -- statements go here
}
```

Part	What It Is	Example
Access modifier	Controls who can call this method. Use `public static` for standalone methods in this course.	`public static`
Return type	The data type the method sends back. Use `void` if it sends nothing back.	`int`, `double`, `String`, `void`
Method name	What you call the method. camelCase, verb-based names are the standard.	`calculateTax`, `printMenu`
Parameter list	The inputs the method receives. Can be empty — just use `()`.	`(double price, int qty)`
Method body	The statements that run when the method is called. Enclosed in `{ }`.	Any valid Java statements

2.1 void Methods — Methods That Do But Don't Return

A `void` method performs a task but does not hand any value back to the caller. Printing output is the classic example.

```
public static void printGreeting(String name) {
    System.out.println("Hello, " + name + "!");
    System.out.println("Welcome to ITP 120.");
}
```

The return type is `void`, so no `return` statement is needed. The method runs and control automatically goes back to the caller when the closing brace is reached.

2.2 Value-Returning Methods

A value-returning method computes something and *sends the result back* to whoever called it. The return type tells Java what kind of value to expect.

```
public static int add(int a, int b) {
    return a + b;   // sends the sum back to the caller
}
```

The return type is `int`. The `return` statement hands the value back. Every code path through the method must eventually reach a `return` statement.

2.3 Naming Conventions

- Use **camelCase**: first word lowercase, each additional word capitalized
- Start with a **verb**: `calculate`, `print`, `get`, `check`, `convert`, `validate`, `is`
- Be specific: `calculateSalesTax` beats `doStuff` every time

> 💡 **One Method, One Job**
>
> If you find yourself naming a method `printReceiptAndSaveToFile`, that is two jobs. Split it into `printReceipt` and `saveToFile`. Short, focused methods are easier to test, reuse, and debug.

3. Calling a Method

Defining a method does nothing on its own. You have to **call** it to make it run.

3.1 Basic Method Call

```
public class GreetingDemo {
    public static void main(String[] args) {
        printGreeting("Alice");   // first call
        printGreeting("Bob");     // second call -- same method, different
argument
    }

    public static void printGreeting(String name) {
        System.out.println("Hello, " + name + "!");
        System.out.println("Welcome to ITP 120.");
    }
}
```

```
Hello, Alice! Welcome to ITP 120. Hello, Bob! Welcome to ITP 120.
```

One method definition, two calls, two different results. That is reuse in action.

3.2 Execution Flow

When Java hits a method call, here is exactly what happens:

1. Java **pauses** execution in the current method
2. **Control jumps** to the method definition
3. The method runs its statements
4. When done (or at `return`), **control returns** to right after the call
5. Execution continues with the next statement

> 🧠 **Think of It This Way**
>
> A method call is like a detour on a road trip. You are driving down the highway (`main`), take an exit (method call), complete what you needed on that side road, then merge back onto the highway exactly where you exited. The highway was waiting for you the whole time.

3.3 Method Calls as Statements vs. in Expressions

A `void` method call is a complete statement by itself:

```
printGreeting("Alice");   // the call IS the statement
```

A value-returning method call can appear anywhere an expression of that type is valid:

```
int total = add(5, 3);                        // store in variable
System.out.println("Sum: " + add(5, 3));      // inside print
if (add(5, 3) > 7) {                          // in a condition
    System.out.println("Greater than 7");
}
System.out.println(add(add(1, 2), add(3, 4))); // nested: add(3,7)=10
```

```
8 Sum: 8 Greater than 7 10
```

> ⚠️ **Don't Use a void Method in an Expression**
>
> `int x = printGreeting("Alice");` is a compile error. `void` means nothing comes back. Only value-returning methods produce something you can store or use in an expression.

4. Parameters and Arguments

These two words get mixed up constantly. Here is the exact difference:

- **Parameters** (formal parameters) — the variables listed in the method *definition*. They are placeholders waiting to receive data.
- **Arguments** (actual arguments) — the actual values you pass when *calling* the method.

```
// DEFINITION: name and age are PARAMETERS
public static void introduce(String name, int age) {
    System.out.println("Hi, I'm " + name + " and I'm " + age + " years old.");
}

// CALL: "Alice" and 21 are ARGUMENTS
introduce("Alice", 21);
```

```
Hi, I'm Alice and I'm 21 years old.
```

Arguments must match parameters in **count, order, and compatible type**. If the definition expects a `String` then an `int`, your call must pass them in that exact order.

4.1 Multiple Parameters

```
public static double rectangleArea(double width, double height) {
    return width * height;
}

// In main:
double area = rectangleArea(5.0, 3.5);
System.out.printf("Area: %.2f square units%n", area);
```

```
Area: 17.50 square units
```

4.2 Pass-by-Value — Java Sends a Copy

In Java, arguments are always passed **by value**. The method receives a *copy* of the argument's value. Whatever the method does to that copy has no effect on the original variable in the caller.

```
public static void doubleIt(int x) {
    x = x * 2;   // changes only the LOCAL copy
    System.out.println("Inside method, x = " + x);
}

public static void main(String[] args) {
    int num = 10;
    doubleIt(num);
    System.out.println("After call, num = " + num);   // still 10!
}
```

```
Inside method, x = 20 After call, num = 10
```

The method received a copy of `10`, doubled that copy to `20`, and printed it. The original `num` in `main` was never touched.

> **⚠️ Common Misconception: Methods Can Change My Variable**
>
> Java sends a copy of the value — not the variable itself. Whatever happens inside the method stays inside the method. The original is safe. Lock this in: *primitives are passed by value; the original never changes inside the method.*

> **🔧 Try It — Pass-by-Value Experiment**
>
> Write a method called `triple(int n)` that multiplies `n` by 3 and prints the result inside the method. In `main`, declare `int myNum = 5`, call `triple(myNum)`, then print `myNum` again. Predict the output before you run it. Were you right?

5. Return Values

When a method computes a result the caller needs, it uses a `return` statement to send it back.

5.1 The return Statement

```
public static double celsiusToFahrenheit(double celsius) {
    double fahrenheit = (celsius * 9.0 / 5.0) + 32.0;
    return fahrenheit;   // send the result back
}
```

When Java hits `return fahrenheit`, it immediately exits the method and sends that value back to the caller. Any code after `return` in the same block is unreachable.

5.2 The Return Type Must Match

The value you return must be compatible with the declared return type. Return a `String` from a method declared as `int` and the compiler will not let it through.

```
public static int square(int n) {
    return n * n;          // int -- matches return type
}

public static String letterGrade(int score) {
    if (score >= 90) return "A";
    if (score >= 80) return "B";
    if (score >= 70) return "C";
    if (score >= 60) return "D";
    return "F";            // all paths covered -- compiler is happy
}

public static boolean isPassing(int score) {
    return score >= 60;    // the expression evaluates to boolean directly
}
```

5.3 Storing and Using Return Values

```
public static double celsiusToFahrenheit(double c) {
    return (c * 9.0 / 5.0) + 32.0;
}

public static void main(String[] args) {
    // Option 1: store in a variable
    double boiling = celsiusToFahrenheit(100.0);
    System.out.println("Boiling: " + boiling + " F");

    // Option 2: use directly in an expression
    System.out.printf("Body temp: %.1f F%n", celsiusToFahrenheit(37.0));

    // Option 3: use in a condition
    if (celsiusToFahrenheit(-10.0) < 32.0) {
        System.out.println("Below freezing!");
    }
}
```

```
Boiling: 212.0 F Body temp: 98.6 F Below freezing!
```

5.4 Early Return in void Methods

A bare `return;` (no value) in a `void` method exits it immediately. Use it to bail out early when a condition makes the rest of the method pointless:

```
public static void printPositive(int n) {
    if (n <= 0) {
        System.out.println("Not a positive number.");
        return;   // exit -- nothing else to do
    }
    System.out.println("Positive: " + n);
}
```

> 💡 **return Exits the Method Immediately**
>
> `return value;` does two things: specifies what to send back *and* immediately exits the method. No extra break or jump needed. Once Java sees `return`, the method is done.

6. Method Overloading

Method overloading lets you give multiple methods the same name, as long as their parameter lists are different. Java picks the right version at compile time based on the arguments you pass.

6.1 A Simple Overloading Example

```
public static int add(int a, int b) {
    return a + b;
}

public static double add(double a, double b) {
    return a + b;
}

public static int add(int a, int b, int c) {
    return a + b + c;
}

// In main:
System.out.println(add(3, 4));        // int version -- 7
System.out.println(add(3.5, 2.1));    // double version -- 5.6
System.out.println(add(1, 2, 3));     // three-int version -- 6
```

```
7 5.6 6
```

6.2 println Is Already Overloaded

You have been using an overloaded method since Module 1. `System.out.println()` has over a dozen versions — one for each type. Java picks the right one automatically:

```
System.out.println(42);       // calls println(int)
System.out.println(3.14);     // calls println(double)
System.out.println("Hello");  // calls println(String)
System.out.println(true);     // calls println(boolean)
```

6.3 Rules for Valid Overloads

Two methods can share a name if they differ in at least one of:

- Number of parameters
- Types of parameters
- Order of parameter types (when the types differ)

⚠️ Return Type Alone Is Not Enough

```
// COMPILE ERROR -- same name, same params, only return type differs
public static int    getValue(int x) { return x; }
public static double getValue(int x) { return x; }  // ERROR!
```

Java picks which method to call based on the *arguments*, not what you plan to do with the result. Return type gives the compiler no information for disambiguation.

🔧 Try It — Overload a Method

Write three overloaded versions of `describe`:

- `describe(String name)` — prints "Name: Alice"
- `describe(String name, int age)` — prints "Name: Alice, Age: 21"
- `describe(String name, int age, String city)` — prints "Name: Alice, Age: 21, City: Fairfax"

Call all three from `main` and verify the correct version runs each time.

7. Scope

Scope refers to where in the program a variable exists and can be used. In Java, a variable declared inside a method is called a **local variable**, and it exists only for the life of that method call. When the method returns, the local variable is gone.

7.1 Local Variables Live in Their Method

```
public static void methodA() {
    int x = 100;   // x exists only inside methodA
    System.out.println("methodA: x = " + x);
}

public static void methodB() {
    // System.out.println(x);  // COMPILE ERROR -- x doesn't exist here!
    int x = 999;   // a completely different x -- no conflict
    System.out.println("methodB: x = " + x);
}

public static void main(String[] args) {
    methodA();
    methodB();
}
```

```
methodA: x = 100 methodB: x = 999
```

Both methods have a variable named `x`, and there is zero conflict between them. They are completely separate variables that happen to share a name. Each one only exists inside its own method.

7.2 Parameters Are Local Variables Too

A method's parameters behave exactly like local variables declared at the top of the method. They receive their initial values from the arguments passed in, and they disappear when the method returns.

```
public static double applyDiscount(double price, double rate) {
    // price and rate are local to this method
    double savings = price * rate;
    double finalPrice = price - savings;
    return finalPrice;
    // price, rate, savings, finalPrice -- all gone after this return
}

public static void main(String[] args) {
    double result = applyDiscount(80.0, 0.20);
    System.out.printf("After 20%% discount: $%.2f%n", result);
    // price, rate, savings, finalPrice are NOT accessible here
}
```

```
After 20% discount: $64.00
```

7.3 Scope Within a Method

Scope also applies within blocks inside a method. A variable declared inside an `if` block or a `for` loop only exists within that block:

```
public static void scopeDemo() {
    int total = 0;

    for (int i = 1; i <= 5; i++) {
        int squared = i * i;   // squared only exists inside the for loop
        total += squared;
    }
    // i and squared are gone here -- cannot use them

    System.out.println("Sum of squares 1-5: " + total);
}
// total is gone after this method returns
```

```
Sum of squares 1-5: 55
```

💡 **Scope Keeps Methods Independent**

Scope is a feature, not a limitation. It means you can write a method without worrying about what variable names other methods are using. Each method is its own little world. This is a huge reason why programs with methods are easier to write and debug than programs without them.

⚠️ **Common Scope Mistake**

```
public static void main(String[] args) {
    if (true) {
        int msg = 42;   // msg lives only inside this if block
    }
    System.out.println(msg);  // COMPILE ERROR: msg cannot be found!
```

Declare variables in the *outermost scope* where they need to be visible. If you need a variable both inside and outside an `if` block, declare it before the `if`.

8. Practical Examples

Everything so far — method definitions, parameters, return values, overloading, scope — comes together here. These four examples mirror real problems you will write in class.

8.1 Temperature Converter (F to C and C to F)

Two conversion methods, each focused on one direction. Clean and reusable:

```
public static double fahrenheitToCelsius(double f) {
    return (f - 32.0) * 5.0 / 9.0;
}

public static double celsiusToFahrenheit(double c) {
    return (c * 9.0 / 5.0) + 32.0;
}

public static void main(String[] args) {
    System.out.printf("32 F = %.1f C%n",  fahrenheitToCelsius(32.0));   // 0.0
C
    System.out.printf("100 C = %.1f F%n", celsiusToFahrenheit(100.0));  //
212.0 F
    System.out.printf("98.6 F = %.1f C%n", fahrenheitToCelsius(98.6));  // 37.0
C
}
```

```
32 F = 0.0 C 100 C = 212.0 F 98.6 F = 37.0 C
```

8.2 Grade Calculator

One method handles the scoring logic. `main` just collects input and displays the result:

```
public static String letterGrade(int score) {
    if (score >= 90) return "A";
    if (score >= 80) return "B";
    if (score >= 70) return "C";
    if (score >= 60) return "D";
    return "F";
}

public static boolean isPassing(int score) {
    return score >= 60;
}

public static void main(String[] args) {
    int[] scores = {95, 83, 72, 58, 61};

    for (int score : scores) {
        String grade = letterGrade(score);
        String status = isPassing(score) ? "PASS" : "FAIL";
        System.out.printf("Score: %d  Grade: %s  Status: %s%n",
                          score, grade, status);
    }
}
```

```
Score: 95 Grade: A Status: PASS Score: 83 Grade: B Status: PASS Score: 72
Grade: C Status: PASS Score: 58 Grade: F Status: FAIL Score: 61 Grade: D
Status: PASS
```

8.3 Input Validation Method

A dedicated method that validates user input, called in a loop until valid data is entered. This pattern shows up in nearly every real program:

```
import java.util.Scanner;

public class ValidatedInput {

    public static boolean isValidAge(int age) {
        return age >= 0 && age <= 120;
    }

    public static int getValidAge(Scanner sc) {
        int age;
        do {
            System.out.print("Enter age (0-120): ");
            age = sc.nextInt();
            if (!isValidAge(age)) {
                System.out.println("Invalid. Try again.");
            }
        } while (!isValidAge(age));
        return age;
    }

    public static void main(String[] args) {
        Scanner sc = new Scanner(System.in);
        int age = getValidAge(sc);
        System.out.println("Valid age entered: " + age);
    }
}
```

```
Enter age (0-120): -5 Invalid. Try again. Enter age (0-120): 200 Invalid. Try again. Enter age (0-120): 25 Valid age entered: 25
```

8.4 Tax Calculator — Multiple Methods Working Together

This is the kind of program where methods shine. Each piece is isolated, testable, and reusable:

```
public class TaxCalculator {

    // Calculate sales tax amount
    public static double calculateTax(double subtotal, double taxRate) {
        return subtotal * taxRate;
    }

    // Calculate the total after tax
    public static double calculateTotal(double subtotal, double taxRate) {
        return subtotal + calculateTax(subtotal, taxRate);
    }

    // Print a formatted receipt line
    public static void printReceiptLine(String label, double amount) {
        System.out.printf("  %-15s $%7.2f%n", label, amount);
    }

    public static void main(String[] args) {
        double subtotal = 45.99;
        double taxRate  = 0.06;   // 6% Virginia sales tax

        double tax   = calculateTax(subtotal, taxRate);
        double total = calculateTotal(subtotal, taxRate);

        System.out.println("===== RECEIPT =====");
        printReceiptLine("Subtotal:",  subtotal);
        printReceiptLine("Tax (6%):",  tax);
        printReceiptLine("Total:",     total);
        System.out.println("==================");
    }
}
```

```
===== RECEIPT ===== Subtotal: $ 45.99 Tax (6%): $ 2.76 Total: $ 48.75
==================
```

Notice that `calculateTotal` calls `calculateTax` — methods can call other methods. The logic is layered and easy to follow. If the tax rate changes, you update one method. If the receipt format changes, you update `printReceiptLine`. Nothing else has to change.

🔧 Try It — Extend the Tax Calculator

Add these two methods to the tax calculator and call them from `main`:

- `applyDiscount(double subtotal, double discountRate)` — returns the price after the discount
- `printSeparator()` — a void method that just prints `"=================="`

Then print a receipt that shows: original price, discounted price, tax on the discounted price, and final total.

Vocabulary Review

Term	Definition
method	A named block of code that performs a specific task. Defined once, called as many times as needed.
void method	A method that performs a task but does not return a value to the caller. Declared with `void` as the return type.
value-returning method	A method that computes a result and sends it back to the caller using a `return` statement.
parameter	A variable in the method *definition* that receives data when the method is called. Also called a formal parameter.
argument	The actual value passed to a method when it is called. Also called an actual argument.
return type	The data type declared before the method name that specifies what kind of value the method returns. `void` means nothing is returned.
return statement	The statement that exits a method and optionally sends a value back to the caller: `return expression;`

Term	Definition
pass-by-value	Java's argument-passing mechanism: the method receives a copy of the argument's value. Changes to the parameter do not affect the original variable.
method overloading	Defining multiple methods with the same name but different parameter lists. Java picks the correct version based on the argument types at compile time.
scope	The region of a program where a variable is accessible. Local variables are in scope only within the method where they are declared.
local variable	A variable declared inside a method. It exists only during the execution of that method and cannot be accessed from other methods.
access modifier	A keyword that controls which code can call a method. `public` means any code can call it; `static` means it belongs to the class rather than an object instance.

📋 Module 07 Summary

- **Methods** are named blocks of code. Define once, call many times. You have been using them since Module 1.
- **Syntax:** `public static returnType methodName(parameterList) { body }`
- **void** methods perform a task and return nothing. Value-returning methods compute a result and send it back with `return`.
- **Calling a method** transfers control to the method, runs it, then returns control to the next statement after the call.
- **Parameters** are placeholders in the definition. **Arguments** are the real values you pass in the call. They must match in count, order, and compatible type.
- **Pass-by-value:** Java passes a copy of the argument. The original variable in the caller is never modified by the method.
- **Return values** can be stored in variables, used in expressions, passed as arguments, or used in conditions.
- **Method overloading:** same name, different parameter lists. Java picks the right version at compile time. Return type alone does not distinguish overloads.
- **Scope:** local variables exist only inside their method. Parameters are local variables. Two methods can use the same variable names without conflict.
- **Practical design:** one method, one job. Short, well-named methods make programs easier to test, debug, and reuse.

Module 07 Quiz

Choose the best answer for each question. Record your answers on a separate sheet.

1. Which of the following is a correct definition of a method that takes no parameters and returns nothing?

A) `public static int displayMenu() { System.out.println("Menu"); }`

B) `public static void displayMenu() { System.out.println("Menu"); }`

C) `static displayMenu() { System.out.println("Menu"); }`

D) `public void static displayMenu() { System.out.println("Menu"); }`

2. What is the output of the following code?

```
public static int mystery(int a, int b) {
    return a * 2 + b;
}

public static void main(String[] args) {
    int result = mystery(3, 4);
    System.out.println(result);
}
```

A) 12

B) 7

C) 14

D) 10

3. In Java, when you pass a primitive (like an `int`) to a method, what does the method receive?

A) A copy of the value; the original variable cannot be changed by the method

B) A reference to the original variable so it can modify it

C) The variable itself, which the method can rename

D) Nothing — primitives cannot be passed to methods

4. What is printed by this code?

```
public static void changeIt(int n) {
    n = n + 100;
}

public static void main(String[] args) {
    int x = 5;
    changeIt(x);
    System.out.println(x);
}
```

A) 105

B) 100

C) 0

D) 5

5. Which term describes the actual value you supply when calling a method?

A) Parameter

B) Argument

C) Return value

D) Identifier

6. A student writes the following two methods. Which statement is true?

```
public static int compute(int x)    { return x * 2; }
public static double compute(int x) { return x * 2.0; }
```

A) This is valid overloading because the return types are different

B) This is valid overloading because the method bodies differ

C) This will not compile because return type alone cannot distinguish overloaded methods

D) This is valid because Java will call the `double` version when the result is stored in a `double`

7. What is the output of the following program?

```
public static String classify(int n) {
    if (n > 0) return "positive";
    if (n < 0) return "negative";
    return "zero";
}

public static void main(String[] args) {
    System.out.println(classify(-7));
    System.out.println(classify(0));
}
```

A) positive
zero

B) negative
negative

C) negative
zero

D) zero
negative

8. Which of the following correctly demonstrates method overloading?

A) Two methods with the same name, same parameters, and same return type

B) Two methods with the same name but a different number of parameters

C) Two methods with different names and different parameter types

D) Two methods with different names but the same parameter list

9. What is the scope of a local variable declared inside a method?

A) It is accessible from any method in the same class

B) It is accessible from any method in the same file

C) It is accessible from `main` only

D) It exists only within the method where it is declared

10. Trace this code carefully. What is printed?

```
public static int doubleAndAdd(int x, int y) {
    return x * 2 + y;
}

public static void main(String[] args) {
    int a = 4;
    int b = 3;
    int result = doubleAndAdd(b, a);
    System.out.println(result);
}
```

A) 11

B) 14

C) 10

D) 7

✅ Answer Key

Question	Answer	Explanation
1	**B**	A method is a named block of code that performs a specific task.
2	**D**	`void` means the method does not return a value.
3	**A**	The `return` statement sends a value back to the calling code.
4	**D**	A method's parameter list defines the data types and names of inputs it accepts.
5	**B**	Method overloading means multiple methods share the same name but have different parameter lists.
6	**C**	A `static` method belongs to the class and can be called without creating an object.
7	**C**	Local variables exist only within the method where they are declared.
8	**B**	Arguments are the actual values passed to a method when it is called.
9	**D**	Java passes primitive types by value — the method receives a copy.
10	**A**	Breaking a program into methods improves readability, reusability, and testing.

ITP 120 — Java Programming I — Northern Virginia Community College
Author: Randy Michak | Contribute on GitHub