

Lecture #1

Object Orientation as a New Paradigm . Introduction to Java programming language.

Material to read: Main Book, Ch. 1-2, up to page 38.

Tasks for training: Lab 1

"The object-oriented version of 'Spaghetti code' is, of course, 'Lasagna code'. (Too many layers)."*

Roberto Waltman

* Spaghetti code represents an unorganized code with no clear structure, beginning or ending.

Programming Paradigms

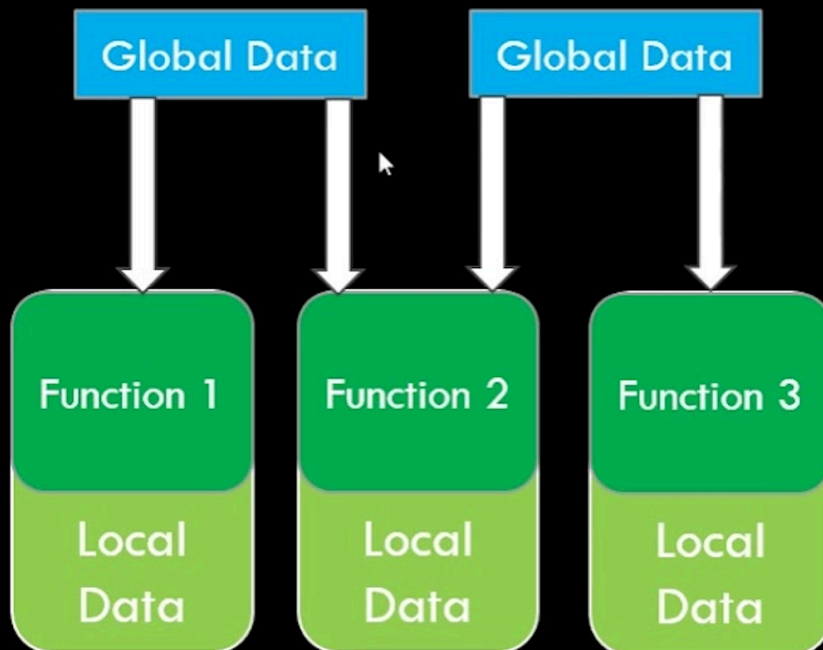
- In the world of programming there exist a number of paradigms and **object-oriented programming** is one of them.
- Other programming paradigms include:
 - the **imperative** programming paradigm (supported by languages such as Pascal or C)
 - the **logic** programming paradigm (exemplified by Prolog, FRIL)
 - the **functional** programming paradigm (represented by languages such as ML, Haskell, Lisp, F# and Scala).
- ***What is vs How to***

OOP

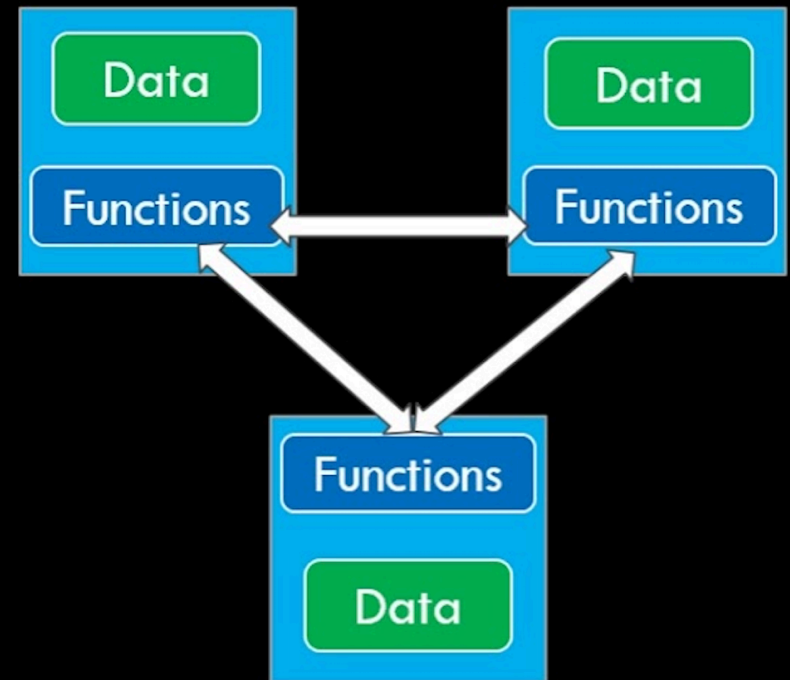
- an object-oriented program is seen as a **community of interacting agents called objects.**
- Each object has a role to play. It has some state and behavior. Each object provides a service or performs an action used by other members of the community.
- Main OOP principles: Inheritance, Encapsulation, Abstraction, Polymorphism (to be discussed later).

Procedural vs Object Oriented Programming

Procedural Oriented Programming



Object Oriented Programming



Java

- Java is a general-purpose object-oriented language. It is intended to let application developers "write once, run anywhere"
- Java applications are typically compiled to bytecode (class file) that can run on any Java Virtual Machine (JVM) regardless of computer architecture.

Java features

- **Simple (the language itself)**
- **Object oriented**
 - ▣ focus on the data (objects) and methods manipulating the data
 - ▣ all functions are associated with objects
 - ▣ almost all datatypes are objects (files, strings, etc.)
- **Portable**
 - ▣ same application runs on all platforms
- **Dynamic Memory Management**
 - ▣ garbage collection

Java features

□ **Reliable**

- ▣ extensive compile-time and runtime error checking (e.g. bytecode checking)
- ▣ no pointers but real arrays. Memory corruptions or unauthorized memory accesses are impossible
- ▣ automatic garbage collection tracks objects usage over time

□ **Interpreted**

- ▣ java compiler generate byte-codes, not native machine code
- ▣ the compiled byte-codes are platform-independent
- ▣ java bytecodes are translated on the fly to machine readable instructions in runtime (Java Virtual Machine)

□ **Network support**

Java features

□ **Secure**

- ▣ usage in networked environments requires more security
- ▣ memory allocation model is a major defense
- ▣ access restrictions are forced (private, public)
- ▣ array access bounds checking

□ **Multithreaded**

- ▣ multiple concurrent threads of executions can run simultaneously

□ **Dynamic**

- ▣ java is designed to adapt to evolving environment
- ▣ libraries can freely add new methods and instance variables without any effect on their clients
- ▣ interfaces promote flexibility and reusability in code by specifying a set of methods an object can perform, but leaves open how these methods should be implemented

Portability

- Portability means that programs must run similarly on any supported hardware/OS platform. This is achieved by compiling the Java code to an intermediate representation called **Java bytecode**, instead of directly to platform-specific **machine code**.
- Bytecode instructions are like machine code, but are intended to be interpreted by a JVM written specifically for the host hardware.
- A major benefit of using bytecode is porting. However, interpreted programs almost always run more slowly than programs compiled to native executables.
- **Just-in-Time (JIT)** compilers compile bytecodes to machine code during runtime.

JVM

- The **JVM** is a piece of software written specifically for a particular platform.
- At run time, the **JVM** reads and interprets **.class files** and executes the program's instructions on the native hardware platform for which the JVM was written.
- The JVM is the heart of the Java language's "**write-once, run-anywhere**" principle. Your code can run on any chipset for which a suitable JVM implementation is available.
- JVMs are available for major platforms like Linux and Windows

Java Runtime Environment - JRE



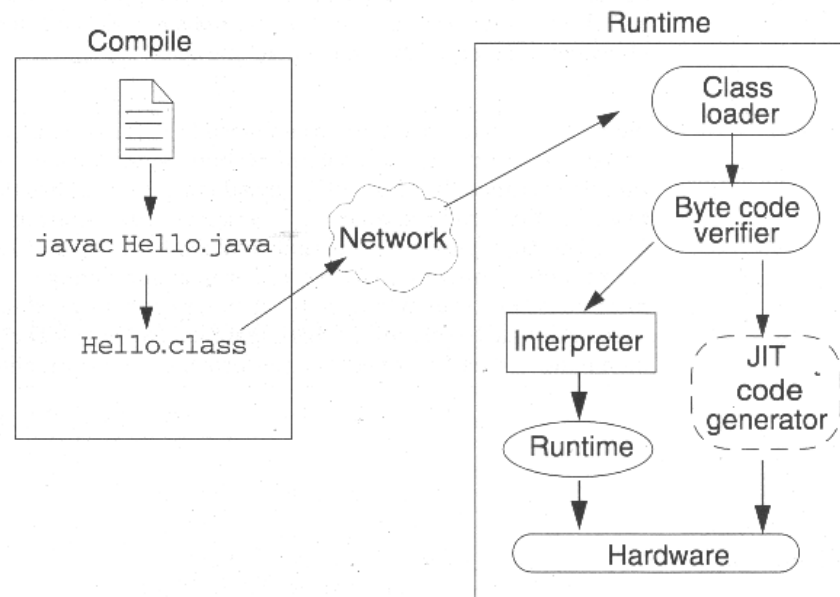
- The Java Runtime Environment includes the JVM, code libraries, and components that are necessary for running programs written in the Java language.
- It is available for multiple platforms

Performance issues

- Using a VM is slower than compiling to native instructions.
 - ▣ **JIT** compilers convert Java Bytecode to machine language during runtime.
- Safety/Security slow things down
 - ▣ all array accesses require bounds check
 - ▣ Many I/O operations require security checks

Compilation, launching

- **javac**: the Java compiler.
 - ▣ Reads source code and generates bytecode.
- **java**: the Java interpreter
 - ▣ Runs bytecode.



Java compiler

- We write source code in **.java** files and then compile them. The compiler checks your code against the language's syntax rules, then writes out **bytecodes** in **.class** files.
- **Bytecodes** are standard instructions targeted to run on a Java virtual machine (JVM). In adding this level of abstraction, the Java compiler differs from other language compilers, which write out instructions suitable for the CPU chipset the program will run on.
- Usage: `javac filename.java`
 - ▣ You can also do: `javac *.java`
 - ▣ Creates `filename.class` (if things work)

Java interpreter

- Usage: `java classname`

You tell the interpreter a class to run, not a file to run!

- The named class should have a method with prototype like:

- `public static void main()`

Comments

- A single line style marked with two slashes (//)
- A multiple line style opened with /* and closed with */
- The commenting style opened with /** and closed with */

```
// This is a comment
/* This is a comment too */
/* This is a
multiline
comment */
```

First java program - text-printing

```
public class Welcome1
{
    // main method begins execution of Java application
    public static void main( String args[] )
    {
        System.out.println( "Welcome to Java Programming!" );

    } // end method main
} // end class Welcome1
```

```
Welcome to Java Programming!
```

- When you save your public class declaration in a file, the file name must be the class name followed by the **".java"** file-name extension. For our application, the file name is Welcome1.java.
- **System.out** is known as the **standard output object**

First java program - text-printing

- To compile the program, type
`javac Welcome1.java`
- If the program contains no syntax errors, the preceding command creates a new file called **Welcome1.class** (known as the **class file** for Welcome1) containing the Java bytecodes that represent our application.
- When we use the **java** command to execute the application, these bytecodes will be executed by the JVM.
java Welcome1

- To print on separate lines, use

```
System.out.println( "Welcome\nto\nJava\nProgramming!" );
```

```
Welcome  
to  
Java  
Programming!
```

- `System.out.print()` method – just prints passed text without moving output cursor to a new line

Addition program

```
// Fig. 2.7: Addition.java
// Addition program that displays the sum of two numbers.
import java.util.Scanner; // program uses class Scanner

public class Addition
{
    // main method begins execution of Java application
    public static void main( String args[] )
    {
        // create Scanner to obtain input from command window
        Scanner input = new Scanner( System.in );

        int number1; // first number to add
        int number2; // second number to add
        int sum; // sum of number1 and number2

        System.out.print( "Enter first integer: " ); // prompt
        number1 = input.nextInt(); // read first number from user

        System.out.print( "Enter second integer: " ); // prompt
        number2 = input.nextInt(); // read second number from user

        sum = number1 + number2; // add numbers

        System.out.printf( "Sum is %d\n", sum ); // display sum

    } // end method main
} // end class Addition
```

Import declaration

- Helps the compiler to locate a class used in program.
- A great strength of Java is its rich set of predefined classes that programmers can reuse rather than "reinventing the wheel." These classes are grouped into **packages** named collections of classes. Collectively, Java's packages are referred to as the **Java Application Programming Interface** (Java API).
- In our example we use Java's predefined **Scanner** class (discussed shortly) from package **java.util**.

Variables

- A **variable** is a location in the computer's memory where a value can be stored for use later in a program.
- All variables must be declared with a name and a type before they can be used.
- **Variable declaration statement** specifies the name and type of a variable used in program.
`int count;`
- **Variable assignment statement**
`count = 8;`

Primitive & reference types

- **The primitive types** are the boolean type and the numeric types. The **numeric types** are the integral types - *byte, short, int, long, and char*, and the floating-point types - *float and double*.
- **The reference types** are *class* types, *interface* types, and *array* types. There is also a special null type.

```
class Point { int[] metrics; }
```

```
interface Move { void move(int deltax, int deltax); }
```

Primitive & reference types

- **Primitive values** . The value of a variable of primitive type can be changed only by assignment operations on that variable. ALWAYS use “==” (double equal) operator to compare for equality.
- An object is a class instance or an array. The **reference values** are pointers to these objects, and a special null reference, which refers to no object.
- Values of primitive types are either stored directly in fields (for objects) or on the stack rather than on the heap, as commonly true for objects. This was a conscious decision by Java's designers for performance reasons

JAVA PROGRAMMING BASICS

By Pakita Shamo

Identifiers

2

- ❑ **An identifier is a name given to a package, class, interface, method, or variable. It allows a programmer to refer to the item from other places in the program.**
- ❑ An identifier must start with a letter, an underscore, or a dollar sign.
- ❑ An identifier cannot contain operators, such as `+`, `-`, and so on.
- ❑ An identifier cannot be a reserved word (e.g. `int`)
- ❑ An identifier cannot be `true`, `false`, or `null`.
- ❑ An identifier can be of any length.

Declaring Variables

3

```
int x;           // Declare x to be an
                 // integer variable;

double radius;  // Declare radius to
                 // be a double variable;

char a;         // Declare a to be a
                 // character variable;
```

Assignment Statements

4

- `x = 1; // Assign 1 to x;`
- `radius = 1.0; // Assign 1.0 to radius;`
- `a = 'A'; // Assign 'A' to a;`

```
radius = 1.0;  
area = radius*radius*3.14159;  
System.out.println("The area is " + area + " for  
radius "+radius);
```

- Declaring and Initializing in 1 step
 - ▣ `int x = 1;`
 - ▣ `double d = 1.4;`

Constants

5

```
final datatype CONSTANTNAME = VALUE;
```

- `final double PI = 3.14159;`
- `final int SIZE = 3;`

Numerical Data Types

6

byte	8 bits
short	16 bits
int	32 bits
long	64 bits
float	32 bits
double	64 bits

Operators

7

- **+, -, *, /, and %**
- $5/2$ yields an integer 2.
- $5.0/2$ yields a double value 2.5
- $5 \% 2$ yields 1 (the remainder of the division)

Shortcut Operators

8

<i>Operator</i>	<i>Example</i>	<i>Equivalent</i>
<code>+=</code>	<code>i += 8</code>	<code>i = i + 8</code>
<code>-=</code>	<code>f -= 8.0</code>	<code>f = f - 8.0</code>
<code>*=</code>	<code>i *= 8</code>	<code>i = i * 8</code>
<code>/=</code>	<code>i /= 8</code>	<code>i = i / 8</code>
<code>%=</code>	<code>i %= 8</code>	<code>i = i % 8</code>

Increment and Decrement Operators

9

□ `x = 1;`

□ `y = 1 + x++;` What is the value of y?

□ `y = 1 + ++x;` What is the value of y?

□ `y = 1 + x--;`

□ `y = 1 + --x;`

What is the difference between `x--` and `--x`?

Numeric Type Conversion

10

Consider the following statements:

```
byte i = 100;
```

```
long myLong = i*3+4;
```

```
double d = i*3.1+1/2;
```

```
int x = myLong; (Wrong)
```

```
long l = x; (fine, implicit casting)
```

Type Casting

11

Implicit casting

```
double d = 3;
```

Explicit casting

```
int i = (int)3.0;
```

What is wrong? `int x = 5/2.0;`

The boolean Type and Operators

12

```
boolean lightsOn = true;
```

```
boolean lightsOn = false;
```

- `&&` (and) `(1 < x) && (x < 100)`
- `||` (or) `(lightsOn) || (isDayTime)`
- `!` (not) `!(isStopped)`

Programming Style and Documentation

13

- Appropriate Comments
- Naming Conventions
- Proper Indentation and Spacing Lines

Appropriate Comments

14

- Include a summary at the beginning of the program to explain what the program does, its key features, its supporting data structures, and any unique techniques it uses.
- Include your name, class section, instruction, date, and a brief description at the beginning of the program.

Naming Conventions

15

- Choose meaning and descriptive names.
- **Variables and method names:**
 - ▣ Use lowercase. If the name consists of several words, concatenate all in one, use lowercase for the first word, and capitalize the first letter of each subsequent word in the name. For example, the variables `radius` and `area`, and the method `computeArea`.
- **Class names:**
 - ▣ Capitalize the first letter of each word in the name. For example, the class name `MyCircle`.
- **Constants:**
 - ▣ Capitalize all letters in constants. For example, the constant `PI`

Programming Errors

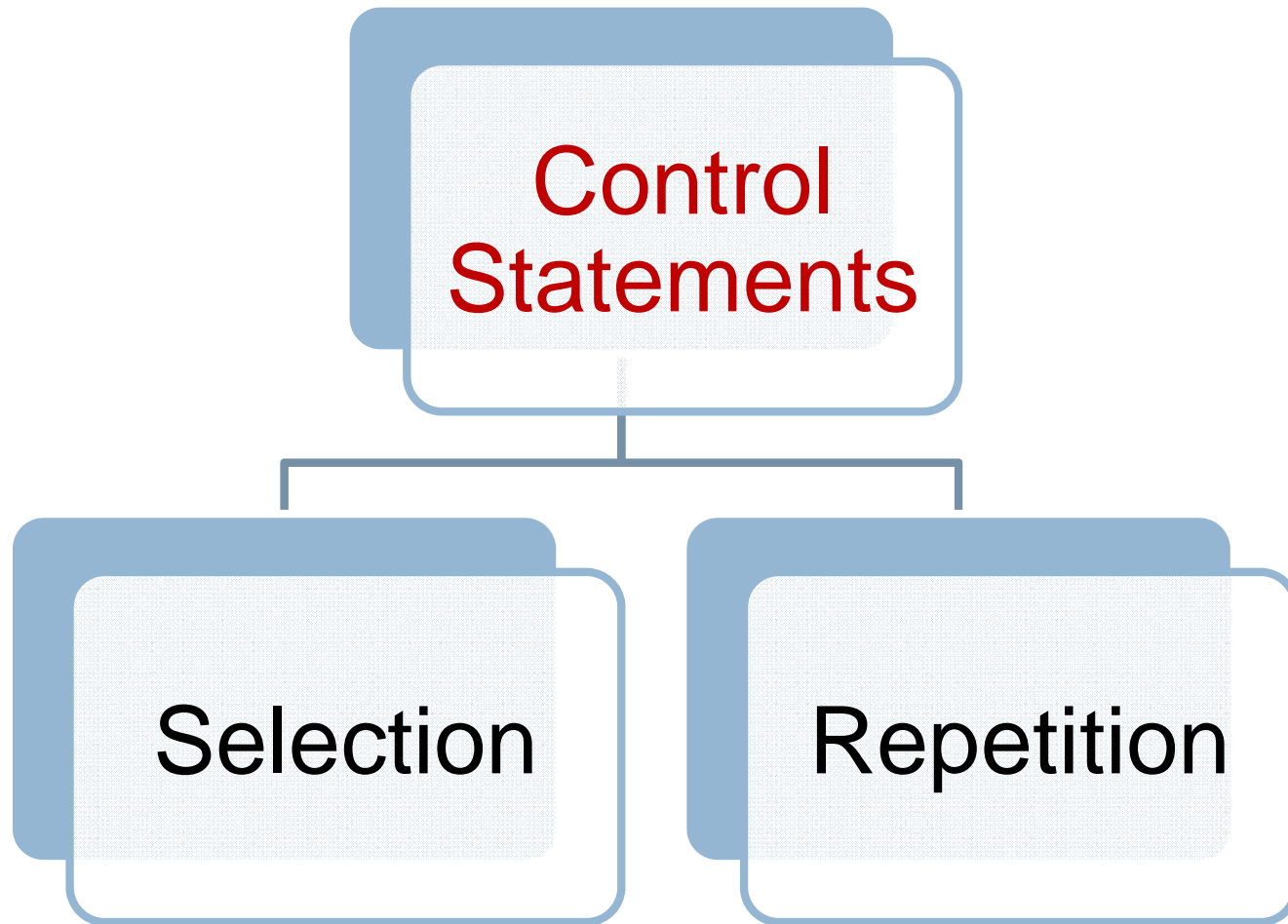
16

- Syntax Errors
 - ▣ Detected by the compiler.
- Runtime Errors
 - ▣ Causes the program to abort
- Logic Errors
 - ▣ Produces incorrect result

Think of examples for each error type

Control statements

17



Selection Statements

18

- `if` Statements
- `switch` Statements
- Conditional Operators

if , if..else statements

19

```
□ if (booleanExpression)
{
    statement(s);
}

□ if (booleanExpression)
{
    statement(s)-for-the-true-case;
}
else
{
    statement(s)-for-the-false-case;
}
```

if...else Example

20

```
if (radius >= 0)
{
    area = radius*radius*PI;
    System.out.println("The area for the "
        + "circle of radius " + radius +
        " is " + area);
}
else
{
    System.out.println("Negative input");
}
```

Conditional Operator

21

```
if (x > 0) y = 1  
else y = -1;
```

is equivalent to

```
y = (x > 0) ? 1 : -1;
```

switch Statements

22

```
switch (year)
{
    case 7:    annualInterestRate = 7.25;
               break;
    case 15:   annualInterestRate = 8.50;
               break;
    case 30:   annualInterestRate = 9.0;
               break;
    default:   System.out.println(
               "Wrong number of years, enter 7, 15, or 30");
}
```

Repetitions

23

- `while` Loops
- `do ...while` Loops
- `for` Loops
- `break` **and** `continue`

while, do..while Loops

24

```
while (continue-condition)
{
    // loop-body;
}
```

```
do
{
    // Loop body;
} while (continue-condition)
```

for Loops

25

```
for (control-variable-initializer;  
     continue-condition; adjustment-statement)  
{  
    //loop body;  
}
```

Example:

```
for (int i = 0; i<100; i++)  
{  
    System.out.println("Welcome to Java! " + i);  
}
```

□ break

□ continue

Methods

26

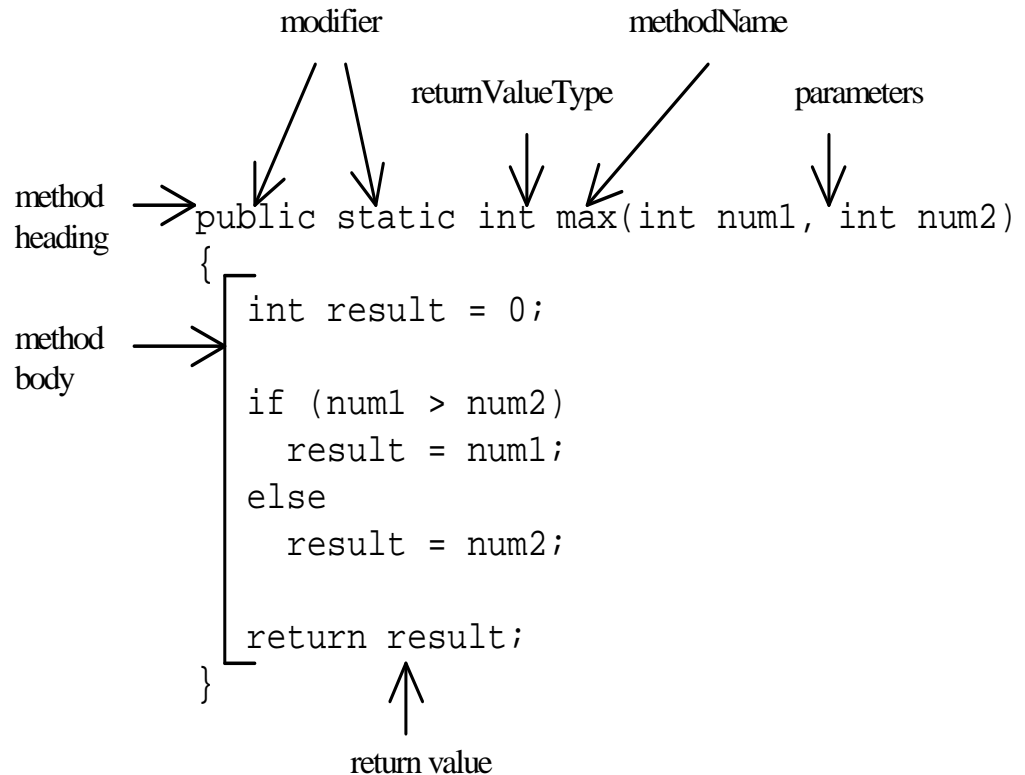
- Introducing Methods
- Declaring Methods
- Calling Methods
- Pass by Value
- Overloading Methods
- Method Abstraction
- The `Math` Class

Introducing Methods

27

A method is a collection of statements that are grouped together to perform an operation.

Method Structure



Declaring Methods

28

```
public static int max(int num1, int num2)
{
    if (num1 > num2)
        return num1;
    else
        return num2;
}
```

Overloading Methods

29

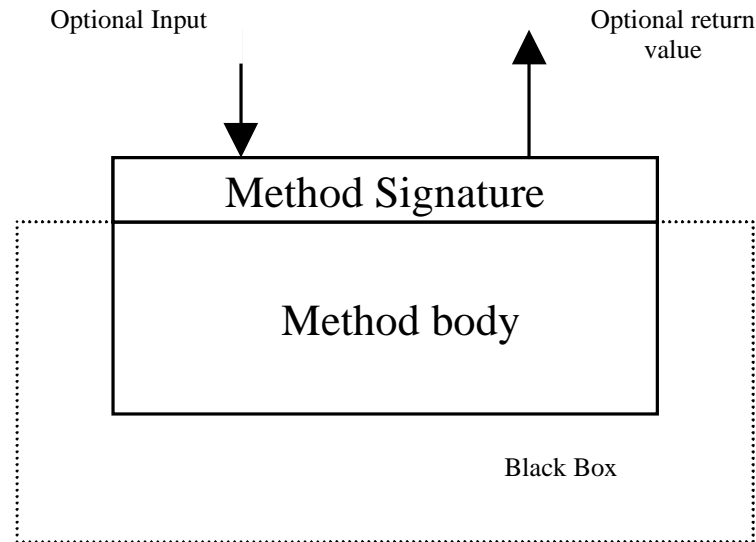
Overloading the `max` Method

```
double max(double num1, double num2)
{
    if (num1 > num2)
        return num1;
    else
        return num2;
}
```

Method Abstraction

30

You can think of the method body as a black box that contains the detailed implementation for the method.



The Math Class

31

- Class constants:
 - ▣ π , e
- Class methods:
 - ▣ Trigonometric Methods
 - ▣ Exponent Methods
 - ▣ Miscellaneous
- ▣ Pow, log, sqrt, sin, cos, abs, random, max, min etc.

The String Class

□ Declaring a String:

- ▣ `String message = "Welcome to Java!"`
- ▣ `String message = new String("Welcome to Java!");`

□ String Comparisons

- ▣ `if (s1.equals(s2)) { // s1 and s2 have the same contents }`
- ▣ `if (s1 == s2) { // s1 and s2 have the same reference }'`

□ String Concatenation

`String` is an immutable class, its values cannot be changed individually.

```
String s1 = "Welcome to Java";
```

```
String s2 = s1.substring(0,10) + "HTML";
```

```
String s3 = s1.concat(s2);
```

```
String s3 = s1 + s2;
```

String Concatenation

□ String Length

```
message = "Welcome";  
message.length() (returns 7)
```

□ Retrieving Individual Characters in a String

- ▣ Do not use `message[0]`
- ▣ Use `message.charAt(index)`
- ▣ Index starts from 0