



الجامعة السعودية الإلكترونية
SAUDI ELECTRONIC UNIVERSITY
2011-1432

College of Computing and Informatics

Object Oriented Programming



Object Oriented Programming

Week 2

Introduction to Computer Science and Programming



Contents

1. Introduction to Hardware and Software
2. Low-Level and High-Level Programming Language
3. Operating System
4. Java Development Environment
5. Internet and World Wide Web



Weekly Learning Outcomes



- Illustrate Low-Level vs High-Level Programming Languages
- Recognize the difference between Compiler and Interpreter



Required Reading

1. Chapter 1 - (Java: How to program: Late Objects)

Recommended Reading

1. Chapter 1 - (Java The Complete Reference, Eleventh Edition)
2. Difference between Compiler and Interpreter:

<https://www.javatpoint.com/compiler-vs-interpreter>

3. NetBeans Tutorials:

<https://netbeans.apache.org//kb/index.html>



- Introduction to Hardware and Software



Introduction

- Java one of the world's most widely used computer programming languages. You're already familiar with the powerful tasks' computers perform.
- For many organizations, the preferred language for meeting their enterprise programming needs is Java.
- Java Editions:
 - Java Standard Edition
 - Contains the capabilities needed to develop desktop and server applications.
 - Java Enterprise Edition
 - Is geared toward developing large-scale, distributed networking applications and web-based applications

Hardware and Software

- Many of today's personal computers can perform billions of calculations in one second
 - A computer consists of various devices referred to as hardware
- Computers process data under the control of sequences of instructions called computer programs.
 - These software programs guide the computer through ordered actions specified by people called computer programmers.
- Hardware
 - Physical devices of computer system
- Software
 - Programs that run on computers

Hardware and Software

- **Computer Organization**
 - Six logical units of computer system
 - Input unit
 - Mouse, keyboard
 - Output unit
 - Printer, monitor, audio speakers
 - Memory unit
 - RAM
 - Arithmetic and logic unit (ALU)
 - Performs calculations
 - Central processing unit (CPU)
 - Supervises operation of other devices
 - Secondary storage unit
 - Hard drives, floppy drives

Hardware and Software

- **Computer Organization**

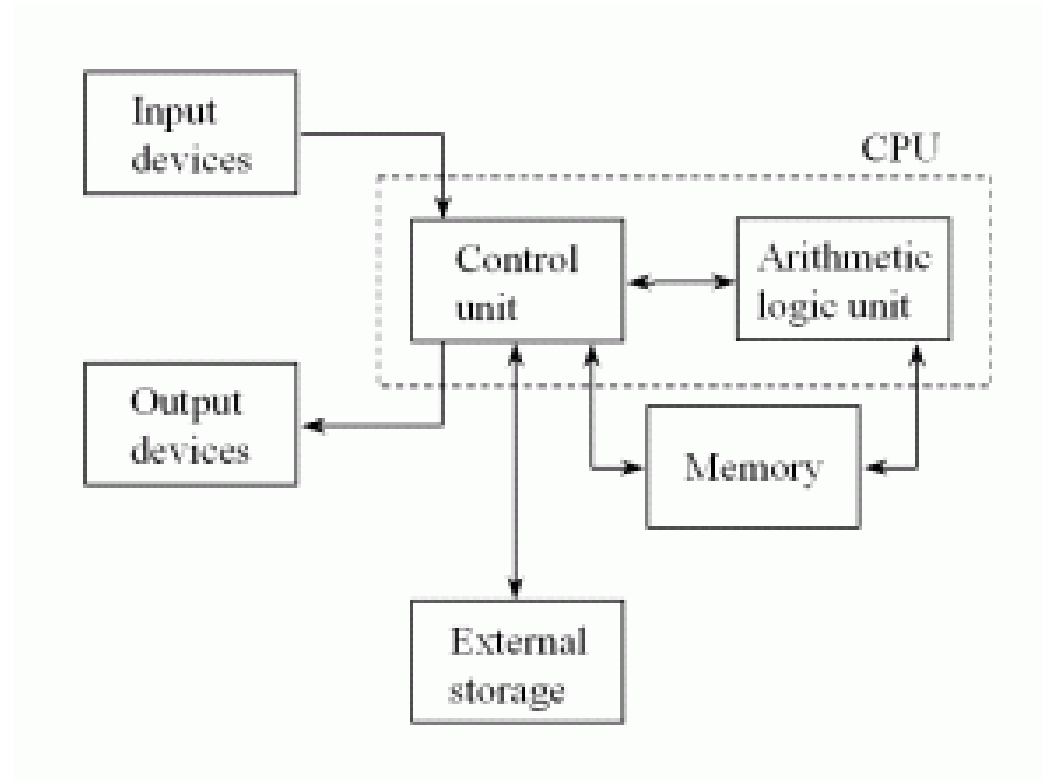


Figure: <https://naoreenchowdhury.wordpress.com/computer-organization-architecture/>

Hardware and Software

- **Data Hierarchy**

- Bits
- Characters
- Fields
- Records
- Files
- Database
- Big Data

Unit	Bytes	Which is approximately
1 kilobyte (KB)	1024 bytes	10^3 (1024 bytes exactly)
1 megabyte (MB)	1024 kilobytes	10^6 (1,000,000 bytes)
1 gigabyte (GB)	1024 megabytes	10^9 (1,000,000,000 bytes)
1 terabyte (TB)	1024 gigabytes	10^{12} (1,000,000,000,000 bytes)
1 petabyte (PB)	1024 terabytes	10^{15} (1,000,000,000,000,000 bytes)
1 exabyte (EB)	1024 petabytes	10^{18} (1,000,000,000,000,000,000 bytes)
1 zettabyte (ZB)	1024 exabytes	10^{21} (1,000,000,000,000,000,000,000 bytes)

Figure: Chapter 1 - (Java: How to program: Late Objects)

- Low-Level and High-Level Programming Language



Machine Languages



Any computer can directly understand only its own machine language, defined by its hardware design.



Machine languages generally consist of strings of numbers (ultimately reduced to 1s and 0s) that instruct computers to perform their most elementary operations one at a time.



Machine languages are machine dependent

Assembly Languages and Assemblers

- Instead of using the strings of numbers that computers could directly understand, programmers began using English-like abbreviations to represent elementary operations.
- These abbreviations formed the basis of assembly languages.
- Translator programs called assemblers were developed to convert early assembly-language programs to machine language at computer speeds.

High-Level Languages and Compilers

- To speed the programming process, high-level languages were developed in which single statements could be written to accomplish substantial tasks.
- Translator programs called compilers convert high-level language programs into machine language.
- High-level languages allow you to write instructions that look almost like everyday English and contain commonly used mathematical notations.

Interpreter

S

- Compiling a large high-level language program into machine language can take considerable computer time.
- Interpreter programs, developed to execute high-level language programs directly, avoid the delay of compilation, although they run slower than compiled programs.

- Operating System



Operating System



- It is software systems that make using computers more convenient for users, application developers and system administrators.
- They provide services that allow each application to execute safely, efficiently and concurrently (i.e., in parallel) with other applications.
- The software that contains the core components of the operating system is the kernel

Operating System

Windows



(2021)

Figure: <https://www.almr.sal.com/post/955105>

Operating System

Linux



Figure: <https://analyticsindiamag.com/10-most-stable-linux-distros-in-2021/>

Operating System

MAC

macOS History

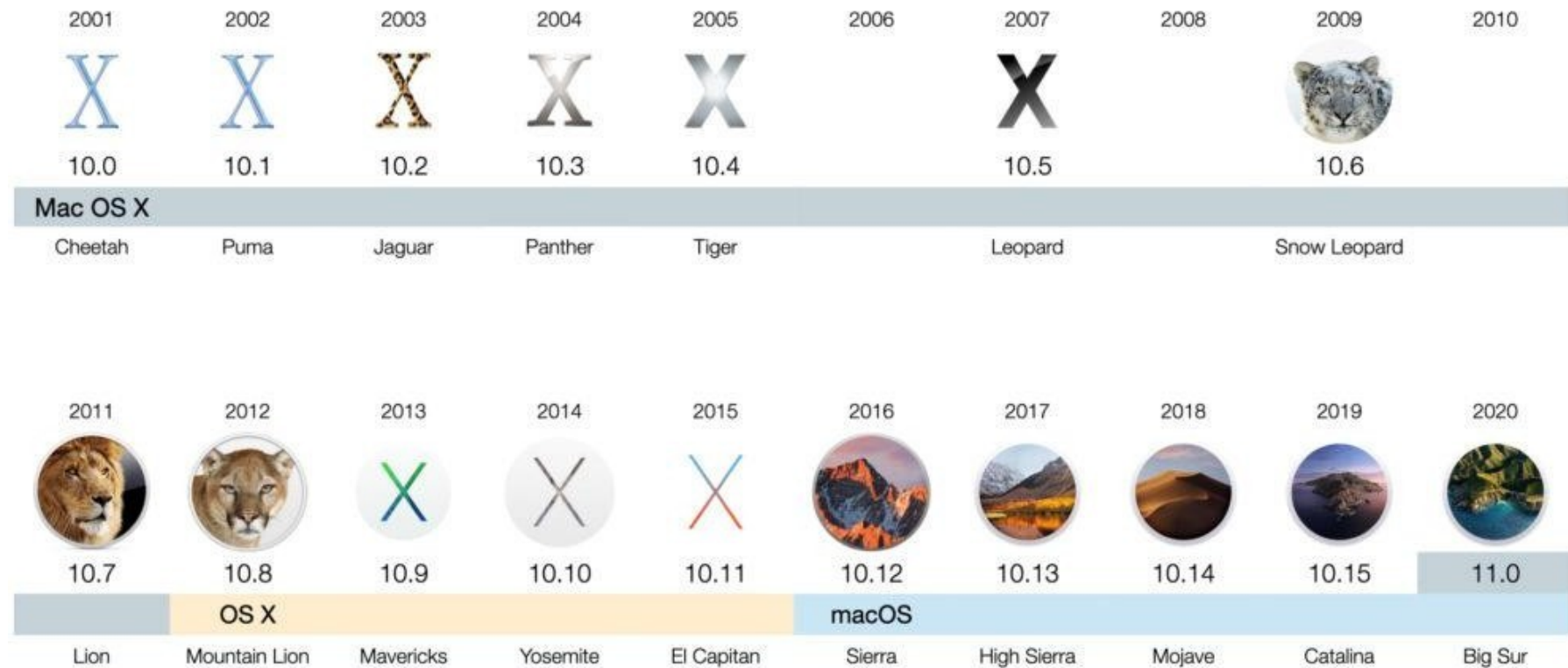


Figure: <https://scriptingosx.com/2021/03/twenty-years-of-mac-os-x/>

Operating System

Android

ANDROID VERSIONS AND THEIR RELEASE DATES



Figure: <https://csquery.com/android-versions-and-their-release-dates/>

- Programming Languages



Programming Languages

- **Java**
 - A key goal of Java is to be able to write programs that will run on a great variety of computer systems and computer-controlled devices. This is sometimes called “write once, run anywhere.”
 - Developed by Sun Microsystems in 1991. Sun Microsystems was acquired by Oracle in 2010.

Programming Languages

- **Java Class Libraries**

- **Classes**

- Contain methods that perform tasks
 - Return information after task completion
 - Used to build Java programs

- **Java contains class libraries**

- Known as Java APIs (Application Programming Interfaces)

- Java Development Environment



Java Development Environment

- Phase 1: Creating a Program
 - consists of editing a file with an editor program, normally known simply as an editor
 - Java source code files are given a name ending with the **.java** extension

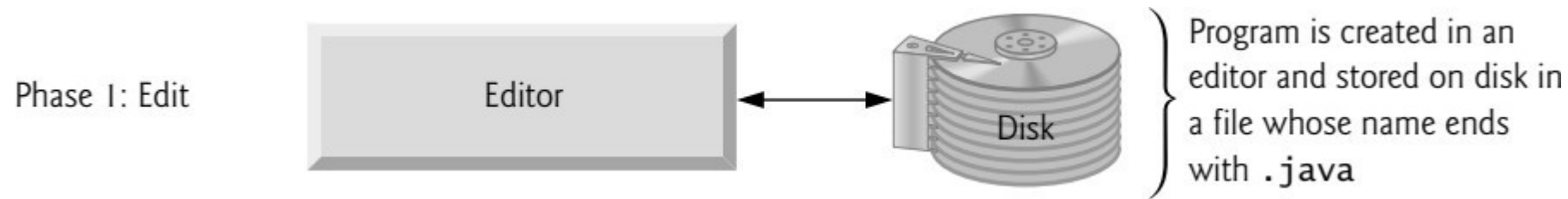


Figure 1.6: Chapter 1 - (Java: How to program: Late Objects)

Java Development Environment

- Phase 1: Creating a Program
 - Integrated development environments (IDEs) provide tools that support the software development process, such as editors, debuggers for locating logic errors.
 - Eclipse (www.eclipse.org)
 - NetBeans (www.netbeans.org)
 - IntelliJ IDEA (www.jetbrains.com)
 - BlueJ (www.bluej.org)

Java Development Environment

- Phase 2: Compiling a Java Program into Bytecodes
 - You use the command **javac** (the Java compiler) to compile a program
 - If the program compiles, the compiler produces a **.class** file called **Welcome.class** that contains the compiled version.
 - IDEs typically provide a menu item, such as Build or Make, that invokes the **javac** command for you.

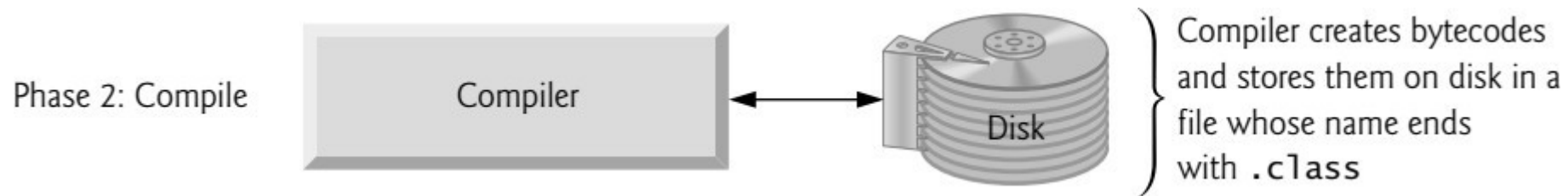


Figure 1.7: Chapter 1 - (Java: How to program: Late Objects)

Java Development Environment

- Phase 2: Compiling a Java Program into Bytecodes
 - The Java compiler translates Java source code into bytecodes that represent the tasks to execute in the execution phase.
 - The Java Virtual Machine (JVM)—a part of the JDK and the foundation of the Java platform—executes bytecodes.
 - A virtual machine (VM) is a software application that simulates a computer but hides the underlying operating system and hardware from the programs that interact with it.

machine-language instructions	bytecode instructions
platform dependent	platform independent
Non-portable	portable
You need to recompile the source code on new machine.	No need to recompile the source code on new machine

Java Development Environment

- Phase 3: Loading a Program into Memory
 - In this phase the JVM places the program in memory to execute it—this is known as loading .
 - The JVM's class loader takes the .class files containing the program's bytecodes and transfers them to primary memory.

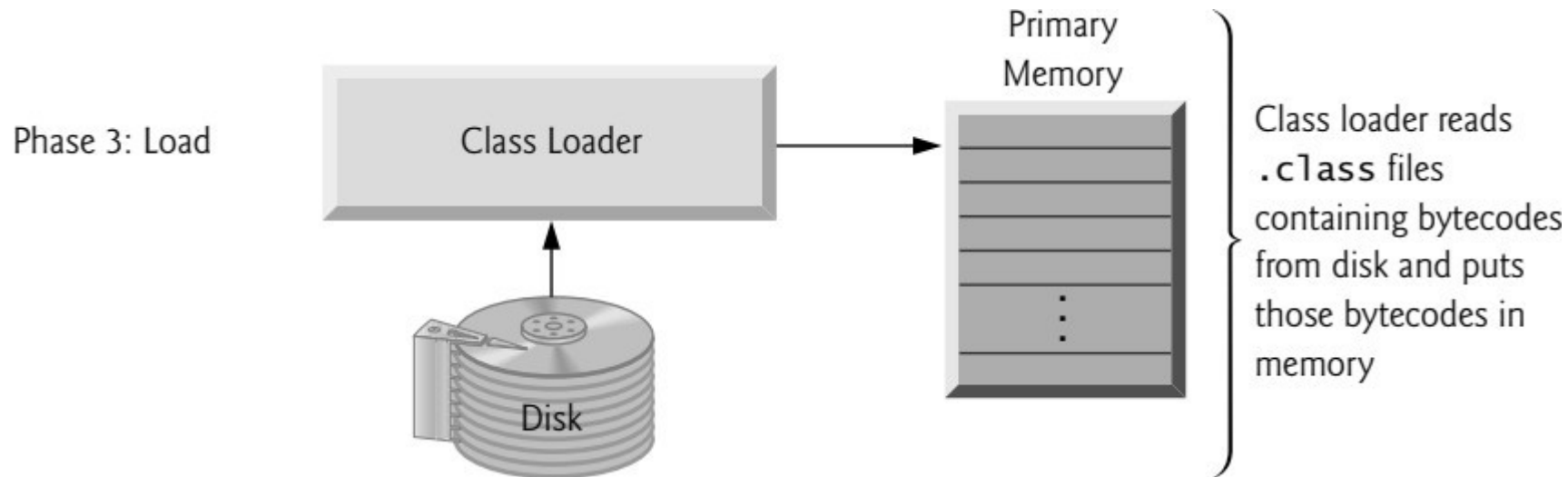


Figure 1.8: Chapter 1 - (Java: How to program: Late Objects)

Java Development Environment

- Phase 4: Bytecode Verification

- After the classes are loaded, the bytecode verifier examines their bytecodes to ensure that they're valid and do not violate Java's security restrictions.
- Java enforces strong security to make sure that Java programs arriving over the network do not damage your files or your system.

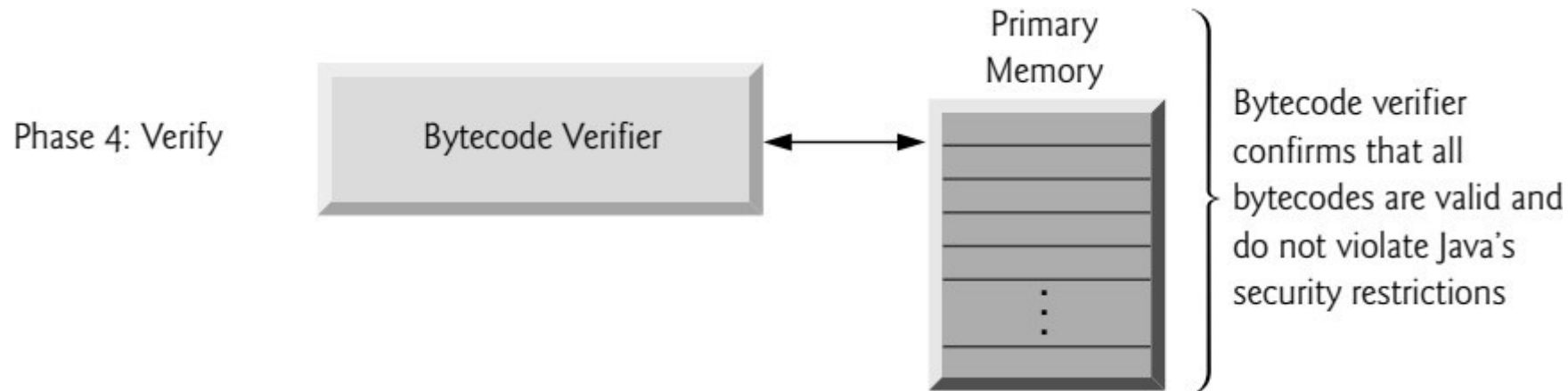


Figure 1.9: Chapter 1 - (Java: How to program: Late Objects)

Java Development Environment

- Phase 5: Execution

- The JVM executes the program's bytecodes, thus performing the actions specified by the program.
- Today's JVMs typically execute bytecodes using a combination of interpretation and so-called just-in-time (JIT) compilation.
 - Searching for hot spots—parts of the bytecodes that execute frequently. For these parts, a just-in-time (JIT) compiler, translates the bytecodes into the underlying computer's machine language.
 - When the JVM encounters these compiled parts again, the faster machine-language code executes.

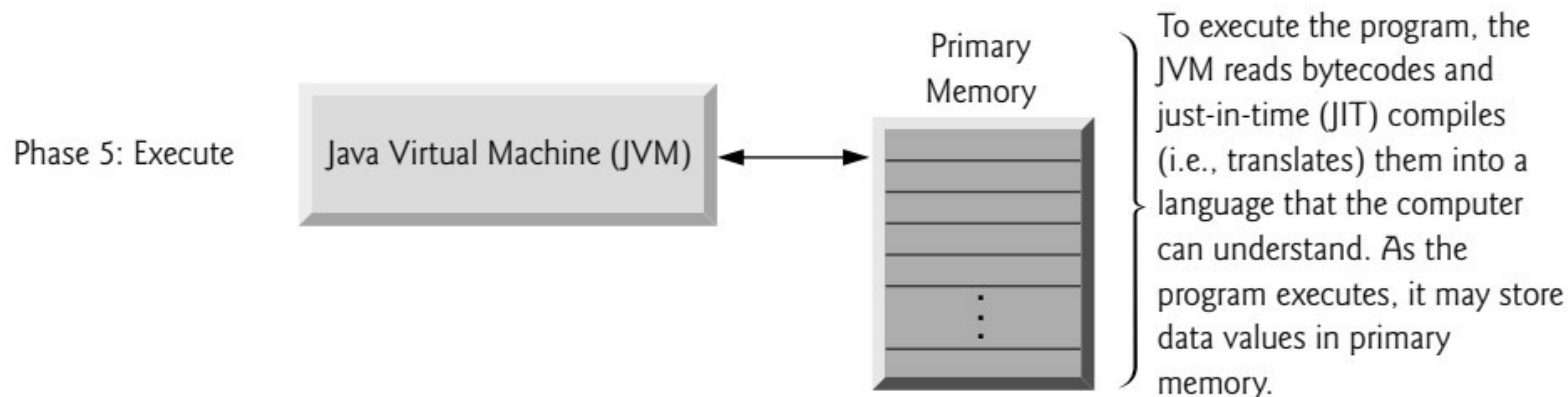


Figure 1.9: Chapter 1 - (Java: How to program: Late Objects)

Java Development Environment

- Phase 5: Execution
 - Two compilation phases
 - One in which source code is translated into bytecodes.
 - Second in which, during execution, the bytecodes are translated into machine language for the actual computer on which the program executes

Java Development Environment

- **Common Programming Error**

- Errors such as division by zero occur as a program runs, so they're called *runtime errors* or *execution-time errors*.
runtime errors or **execution-time errors**.
- *Fatal runtime errors*
- **Fatal runtime errors** cause programs to terminate immediately without having successfully performed their jobs.
- *Nonfatal runtime errors*
- **Nonfatal runtime errors** allow programs to run to completion, often producing incorrect results.

- Internet and World Wide Web



Internet and World Wide Web

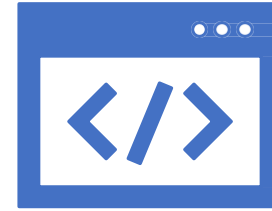


Internet

Developed over three decades ago with DOD funding

Originally for connecting few main computer systems

Now accessible by hundreds of millions of computers



World Wide Web (WWW)

Allows for locating/viewing multimedia-based documents

- Introduction to Object Technology



Introduction to Object Technology

- Objects
 - Reusable software components that model real-world items
 - Look all around you
 - People, cars, mobile phone, etc.
 - Attributes
 - Size, shape, color, weight, etc.
 - Behaviors
 - Attend lecture, make call, stop engine, etc.

Introduction to Object Technology

- Object-oriented design (OOD)
 - Models real-world objects
 - Models communication among objects
 - Encapsulates data (attributes) and functions (behaviors)
 - Information hiding
 - Communication through well-defined interfaces
- Object-oriented language
 - Programming is called object-oriented programming (OOP)
 - Java

Tutorial to Install JDK and IDE

1. Install JDK:

- <https://docs.oracle.com/en/java/javase/11/install/installation-jdk-microsoft-windows-platforms.html#GUID-A740535E-9F97-448C-A141-B95BF1688E6F>

2. Install any of the following IDE:

- Netbeans
 - <https://studyopedia.com/netbeans/installing-netbeans-ide-windows-10/>
- BlueJ
 - <https://www.bluej.org/download/install.html>
- Eclipse
 - <https://www.eclipse.org/downloads/packages/installer>
- IntelliJ IDEA
 - <https://www.jetbrains.com/help/idea/installation-guide.html#toolbox>



Reference

S

- Deitel, P. J., & Deitel, H. M. (2017). Java: how to program late objects. Pearson.

Thank You





الجامعة السعودية الإلكترونية
SAUDI ELECTRONIC UNIVERSITY
2011-1432

College of Computing and Informatics

Object Oriented Programming



Object Oriented Programming

Week 3

Introduction to Java applications I



Contents

1. Writing First Program in Java (Block of Code, Statements, Keywords)
2. Displaying Text (print, println, printf, escape sequences)
3. Obtain User Input from the Keyboard
4. Import Libraries
5. Commenting Code
6. Debug and Trace Errors



Weekly Learning Outcomes

- Write first program in Java
- Manipulate the debugging tools and trace errors
- Show the use of Java libraries



Required Reading

1. Chapter 2 - (Java: How to program: Late Objects)

Recommended Reading

1. Chapter 2 - (Java The Complete Reference, Eleventh Edition)
2. Importing Libraries:

<http://www.mathcs.emory.edu/~cheung/Courses/170/Syllabus/04/java-lib.html>

3. Debugging with Netbeans:

https://docs.oracle.com/javame/config/cdc/cdc-opt-impl/ojmeec/1.1/developer/html/debugging_netbeans.htm

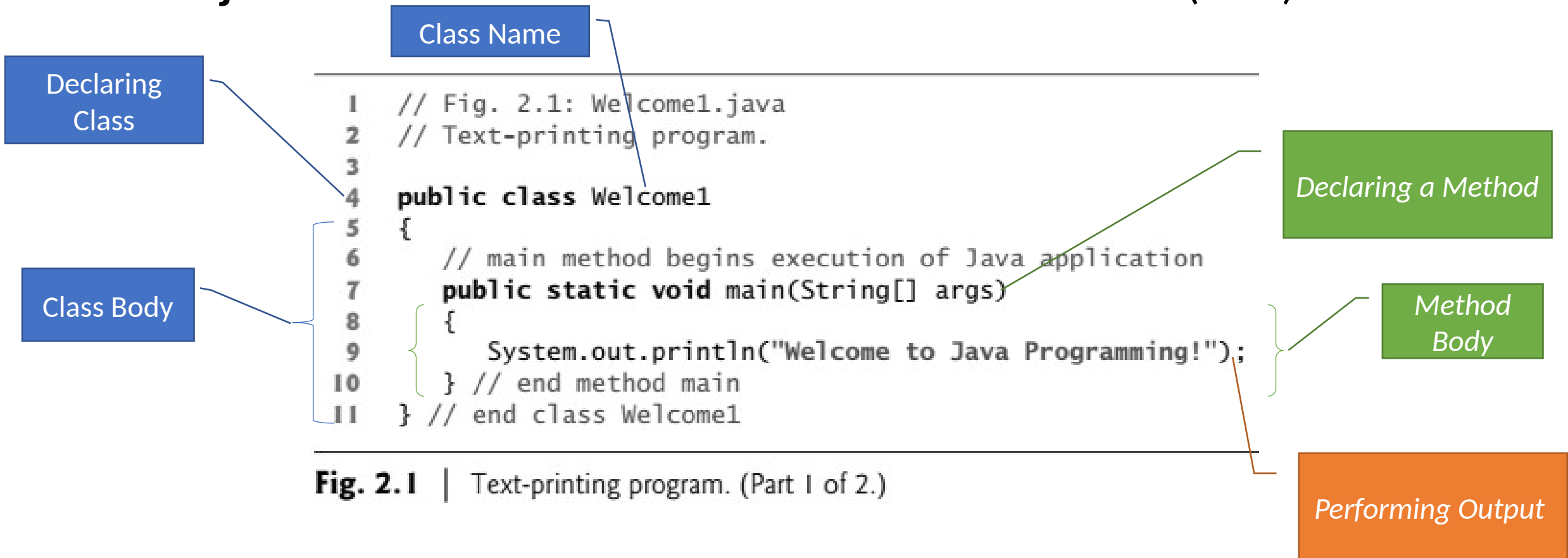


- Writing First Program in Java (Block of Code, Statements, Keywords)



Writing First Program in Java

- A Java application is a computer program that executes when you use the **java** command to launch the Java Virtual Machine (JVM).



Writing First Program in Java

- *Commenting Your Programs:*
 - Comments ignored during program execution
 - To document programs
 - To improve their readability
- 1. One Line:
 - `// Fig. 2.1: Welcome1.java`
- 2. Multi-Lines:
 - `/* This is a traditional comment. It
can be split over multiple lines */`
- 3. Javadoc comments
 - `/** and */`
 - to embed program documentation directly in your programs

Writing First Program in Java

Blank line

```
2 // Text-printing program.  
3
```

- Makes program more readable
- Blank lines, spaces, and tabs are white-space characters
 - Ignored by compiler

Begins class declaration for class Welcome1

```
4 public class Welcome1
```

- Every Java program has at least one user-defined class
- Keyword: words reserved for use by Java
 - class keyword followed by class name

Writing First Program in Java

Name of class
called identifier

7

```
public static void main(String[] args)
```

- Series of characters consisting of letters, digits, underscores (_) and dollar signs (\$)
- Does not begin with a digit, has no spaces
- Examples: Welcome1, \$value, _value, button7
 - **7button** is invalid
- Java is case sensitive (capitalization matters)
 - a1 and A1 are different

public keyword will be discussed in later chapters in detail.

Writing First Program in Java

Saving files

```
4 public class Welcome1
```

- File name must be class name with .java extension
- Welcome1.java

Left brace {

- Begins body of every class
- Right brace ends declarations (line 13)

Part of every Java application

```
7 public static void main(String[] args)
```

- Applications begin executing at main
 - Parenthesis indicate main is a method (ch. 6)
 - Java applications contain one or more methods

Writing First Program in Java

9

```
System.out.println("Welcome to Java Programming!");
```

Instructs computer to perform an action

- Prints string of characters
 - String - series characters inside double quotes
- White-spaces in strings are not ignored by compiler

System.out

- Standard output object
- Print to command window (i.e., MS-DOS prompt)

Method System.out.println

- Displays line of text
- Argument inside parenthesis

This line known as a statement

- Statements must end with semicolon ;

Writing First Program in Java

- *Compiling Your First Java Application*
 - ***javac Welcome1.java***
 - If no errors, `Welcome1.class` created
 - Has bytecodes that represent application
 - Bytecodes passed to Java interpreter
- *Executing the **Welcome1** Application*
 - ***java Welcome1***
 - Interpreter loads `.class` file for class `Welcome1`
 - `.class` extension omitted from command
 - Interpreter calls method `main`



You type this command to execute the application

The program outputs to the screen
Welcome to Java Programming!

Fig. 2.2 | Executing `Welcome1` from the Command Prompt.

Writing First Program in Java

- **Modifying Your First Java Program**
 - *Displaying a Single Line of Text with Multiple Statements*

`System.out.print` keeps the cursor on the same line, so `System.out.println` continues the same line.

```
1 // Fig. 2.3: Welcome2.java
2 // Printing a line of text with multiple statements.
3
4 public class Welcome2
5 {
6     // main method begins execution of Java application
7     public static void main(String[] args)
8     {
9         System.out.print("Welcome to ");
10        System.out.println("Java Programming!");
11    } // end method main
12 } // end class Welcome2
```

Fig. 2.3 | Printing a line of text with multiple statements. (Part 1 of 2.)

Welcome to Java Programming!

Fig. 2.3 | Printing a line of text with multiple statements. (Part 2 of 2.)

Writing First Program in Java

- *Displaying Multiple Lines of Text with a Single Statement*

```
1 // Fig. 2.4: Welcome3.java
2 // Printing multiple lines of text with a single statement.
3
4 public class Welcome3
5 {
6     // main method begins execution of Java application
7     public static void main(String[] args)
8     {
9         System.out.println("Welcome\nto\nJava\nProgramming!");
10    } // end method main
11 } // end class Welcome3
```

Newline character

```
Welcome
to
Java
Programming!
```

Fig. 2.4 | Printing multiple lines of text with a single statement.

Writing First Program in Java



Displaying Multiple Lines of Text with a Single Statement



Newline characters (\n)

Interpreted as “special characters” by methods
`System.out.print` and `System.out.println`
Indicates cursor should be on next line



Usage

Can use in `System.out.println` or
`System.out.print` to create new lines

Writing First Program in Java

Escape sequence	Description
<code>\n</code>	Newline. Position the screen cursor at the beginning of the <i>next</i> line.
<code>\t</code>	Horizontal tab. Move the screen cursor to the next tab stop.
<code>\r</code>	Carriage return. Position the screen cursor at the beginning of the <i>current</i> line—do <i>not</i> advance to the next line. Any characters output after the carriage return <i>overwrite</i> the characters previously output on that line.
<code>\\</code>	Backslash. Used to print a backslash character.
<code>\"</code>	Double quote. Used to print a double-quote character. For example, <code>System.out.println("\"in quotes\");</code> displays "in quotes".

Fig. 2.5 | Some common escape sequences.

- Displaying Text



Displaying Text

- **Displaying Text with printf**
 - **printf** method (f means “formatted”) displays *formatted* data

the format specifier %s
is a placeholder for a
string

```
1 // Fig. 2.6: Welcome4.java
2 // Displaying multiple lines with method System.out.printf.
3
4 public class Welcome4
5 {
6     // main method begins execution of Java application
7     public static void main(String[] args)
8     {
9         System.out.printf("%s%n%s%n",
10             "Welcome to", "Java Programming!");
11     } // end method main
12 } // end class Welcome4
```

the %n format
specifier,
which is a line
separator

```
Welcome to
Java Programming!
```

Fig. 2.6 | Displaying multiple lines with method `System.out.printf`.

Displaying Text

- **Displaying Text with printf**

- `System.out.printf(format, arguments);`

- Example:

- `System.out.printf("Hello %s!%n", "World");`
- Output:
 - Hello World!

- **Format Rules**

- `%[flags][width][.precision]conversion-character`
- The *[flags]* define standard ways to modify the output
- The *[width]* specifies the field width for outputting the argument.
- The *[.precision]* specifies the number of digits of precision

Displaying Text

- **Displaying Text with printf**
 - **Conversion Characters**
 - *s* formats strings.
 - *d* formats decimal integers.
 - *f* formats floating-point numbers.
 - *t* formats date/time values.

Displaying Text

- **Displaying Text with printf**
 - **Boolean Formatting**
 - `System.out.printf("%b%n", null);`
 - `System.out.printf("%B%n", false);`
 - `System.out.printf("%B%n", 5.3);`
 - `System.out.printf("%b%n", "random text");`
 - **Output:**
 - false
 - FALSE
 - TRUE
 - true

Displaying Text

- **Displaying Text with printf**
 - **String Formatting**
 - `printf("%s' %n", "baeldung");`
 - `printf("%S' %n", "baeldung");`
 - Output:
 - 'baeldung'
 - 'BAELDUNG'
 - `printf("%15s' %n", "baeldung");`
 - Output:
 - ' baeldung'

Displaying Text

- **Displaying Text with printf**
 - **Number Formatting**
 - `System.out.printf(Locale.US, "%,d %n", 10000);`
 - `System.out.printf(Locale.ITALY, "%,d %n", 10000);`
 - Output:
 - 10,000
 - 10.000
 - `System.out.printf("%5.2f%n", 5.1473);`
 - Output:
 - ' 5.15'

Displaying Text

- **Displaying Text with printf**

- **Date and Time Formatting**

- `System.out.printf("hours %tH: minutes %tM: seconds %tS%n", date, date, date);`
 - Having used *H*, *M* and *S*, we get this result:
 - hours 13: minutes 51: seconds 15
 - `System.out.printf("%1$tA, %1$tB %1$tY %n", date);`
 - Then, using *A*, *B* and *Y*, we'd get this output:
 - Thursday, November 2018

- Obtain User Input from the Keyboard



Obtain User Input from the Keyboard

- When you read input from the user, you need to store it in your program.
 - A variable is a location in the computer's memory where a value can be stored for use later in a program.
 - Variables will be discussed in details in the next lecture.
- To read an input you use A **Scanner** to enables a program to read data
 - `Scanner input = new Scanner(System.in);`
 - This expression uses the **new** keyword to create a Scanner object that reads characters typed by the user at the keyboard.
 - The standard input object, **System.in**, enables applications to read *bytes* of data typed by the user.

Obtain User Input from the Keyboard

Read an integer:

- **First you Prompt the User for Input:**
 - `System.out.print("Enter integer: "); // prompt`
- **Then Obtaining an int as Input from the User**
 - `int number1 = input.nextInt(); // read a number from user`
 - `nextInt` method to obtain an integer from the user at the keyboard.

Obtain User Input from the Keyboard

Read a double:

- **First you Prompt the User for Input:**
 - `System.out.print("Enter a double: "); // prompt`
- **Then Obtaining an int as Input from the User**
 - `double number1 = input.nextDouble(); // read a double from user`
 - nextDouble method to obtain an integer from the user at the keyboard.

Obtain User Input from the Keyboard

Read a string:

- **First you Prompt the User for Input:**
 - `System.out.print("Enter a string: "); // prompt`
- **Then Obtaining an int as Input from the User**
 - `String s = input.next(); // read a string from user`
 - next method to obtain an integer from the user at the keyboard.

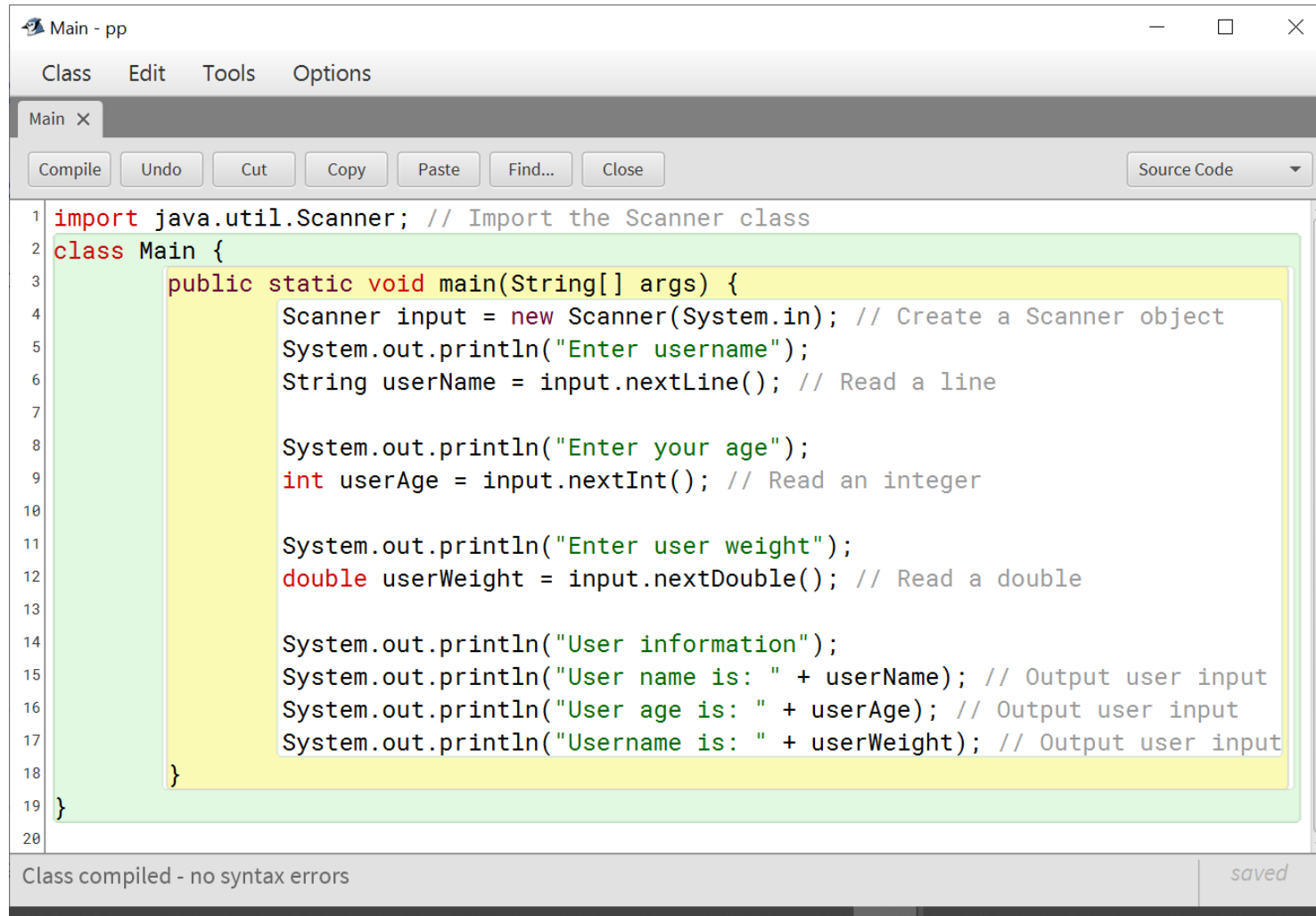
Obtain User Input from the Keyboard

Read a line:

- **First you Prompt the User for Input:**
 - `System.out.print("Enter a line: "); // prompt`
- **Then Obtaining an int as Input from the User**
 - `String s = input.nextLine(); // read a line from user`
 - `nextLine` method to obtain an integer from the user at the keyboard.

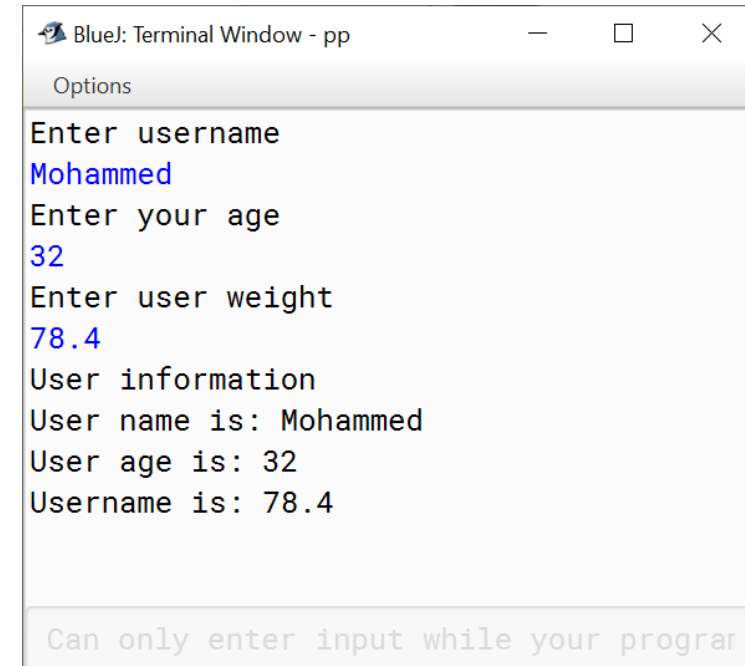
Obtain User Input from the Keyboard

- Simple Example:



```
1 import java.util.Scanner; // Import the Scanner class
2 class Main {
3     public static void main(String[] args) {
4         Scanner input = new Scanner(System.in); // Create a Scanner object
5         System.out.println("Enter username");
6         String userName = input.nextLine(); // Read a line
7
8         System.out.println("Enter your age");
9         int userAge = input.nextInt(); // Read an integer
10
11        System.out.println("Enter user weight");
12        double userWeight = input.nextDouble(); // Read a double
13
14        System.out.println("User information");
15        System.out.println("User name is: " + userName); // Output user input
16        System.out.println("User age is: " + userAge); // Output user input
17        System.out.println("Username is: " + userWeight); // Output user input
18    }
19 }
20
```

Class compiled - no syntax errors



```
Options
Enter username
Mohammed
Enter your age
32
Enter user weight
78.4
User information
User name is: Mohammed
User age is: 32
Username is: 78.4

Can only enter input while your program is running
```

Obtain User Input from the Keyboard

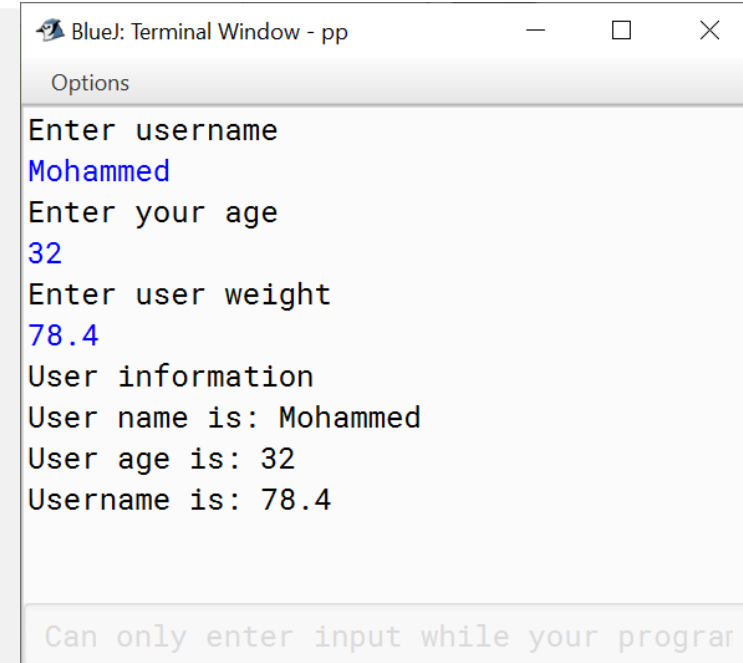
- **Simple Example:**

```
import java.util.Scanner; // Import the Scanner class
class Main {
    public static void main(String[] args) {
        Scanner input = new Scanner(System.in); // Create a Scanner object
        System.out.println("Enter username");
        String userName = input.nextLine(); // Read a line

        System.out.println("Enter your age");
        int userAge = input.nextInt(); // Read an integer

        System.out.println("Enter user weight");
        double userWeight = input.nextDouble(); // Read a double

        System.out.println("User information");
        System.out.println("User name is: " + userName); // Output user input
        System.out.println("User age is: " + userAge); // Output user input
        System.out.println("User weight is: " + userWeight); // Output user input
    }
}
```

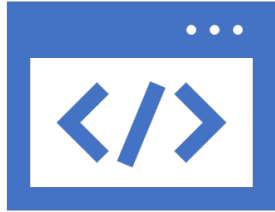


```
BlueJ: Terminal Window - pp
Options
Enter username
Mohammed
Enter your age
32
Enter user weight
78.4
User information
User name is: Mohammed
User age is: 32
User weight is: 78.4
Can only enter input while your program is running
```

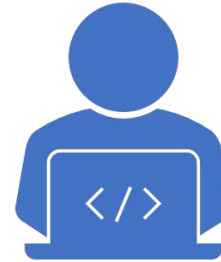
- Import Libraries



Import Libraries



A great strength of Java is its rich set of predefined classes that you can *reuse* rather than “reinventing the wheel.” These classes are grouped into packages.



referred to as the Java class library, or the Java Application Programming Interface (Java API)

```
import java.util.Scanner; // program uses class Scanner
```

- **import** declaration that helps the compiler locate a class that’s used in this program
- the package that is named **java.util**

Import Libraries

- **Java API Documentation**

- For each new Java API class we use, we indicate the package in which it's located.
- The Java API Documentation includes a detailed description of each package and its classes.
 - <https://docs.oracle.com/javase/7/docs/api/index.html>
- You can download it from the Additional Resources section at
 - <http://www.oracle.com/technetwork/java/javase/downloads/index.html>

- Commenting Code



Commenting Code

- Javadoc is a tool which comes with JDK and it is used for generating Java code documentation in HTML format from Java source code, which requires documentation in a predefined format.
 - Following is a simple example where the lines inside `/*...*/` are Java multi-line comments. Similarly, the line which precedes `//` is Java single-line comment.

```
/**
 * The HelloWorld program implements an application that
 * simply displays "Hello World!" to the standard output.
 *
 * @author  name
 * @version 1.0
 * @since 2014-03-31
 */
public class HelloWorld {
    public static void main(String[] args) {
        // Prints Hello, World! on standard output.
        System.out.println("Hello World!");
    }
}
```

Reference: https://www.tutorialspoint.com/java/java_documentation.htm

Commenting Code

```
/** The Calculator class provides methods to get addition and subtraction of given 2 numbers. */  
public class Calculator {  
    /** The add() method returns addition of given numbers. */  
    public static int add(int a, int b){return a+b;}  
    /** The sub() method returns subtraction of given numbers. */  
    public static int sub(int a, int b){return a-b;}  
}
```

Compile it by javac tool:

javac Calculator.java

Create Documentation API by javadoc tool:

javadoc Calculator.java

- Now, there will be HTML files created for your Calculator class in the current directory

- Debug and Trace Errors



Debug and Trace Errors

- **Debugging** is the routine process of locating and removing bugs, errors or abnormalities from programs.
 - **A conditional breakpoint:**
 - Applying breakpoints lets a debugger temporarily suspend the execution of your program at a certain point in the code.
 - This allows you to examine the state of the application, the values of local variables, the call stack, and to step through the code line-by-line.
 - This information gives you more clues to why the problem occurs.
 - To set a breakpoint, just toggle it by clicking on the left-hand side of the IDE screen.

Debug and Trace Errors

- **Exception breakpoints:**

- Runtime exceptions like `NullPointerException` and `ArrayIndexOutOfBoundsException` are the errors encountered most often.
- Figuring out the root cause of these errors can be difficult.
- IDEs offer exception breakpoints in which the execution of the program is stopped whenever the specified exception occurs.

- **Watch point:**

- It allows you to stop the execution of a program each time a targeted field or variable is accessed or changed.
- The watchpoint can be applied on a field or variable.

Debug and Trace Errors

- Check the following link for a complete tutorial for debugging steps in NetBeans:
 - <https://medium.com/@teguhteja/how-to-debugging-in-netbeans-java-c3087d9e2f6c>

References

- Deitel, P. J., & Deitel, H. M. (2017). *Java: how to program late objects*. Pearson.
- <https://www.baeldung.com/java-printstream-printf>
- <https://rollbar.com/guides/java/how-to-debug-java-errors/>

Thank You





الجامعة السعودية الإلكترونية
SAUDI ELECTRONIC UNIVERSITY
2011-1432

College of Computing and Informatics

Object Oriented Programming



Object Oriented Programming

Week 4

Introduction to Java applications II



Contents

1. Variables
2. Memory Concepts
3. Operators
4. Increment/Decrement Operators
5. Assignment & Expressions
6. Logic and Syntax errors & Debugging concepts



Weekly Learning Outcomes



- Construct different variables types in Java
- Illustrate the usage of operators, assignments and expressions



Required Reading

1. Chapter 2 - (Java: How to program: Late Objects)

Recommended Reading

1. Chapter 3, Chapter 4 - (Java The Complete Reference, Eleventh Edition)
2. Java Variables: <https://www.javatpoint.com/java-data-types>
3. Java Data Types: <https://www.javatpoint.com/java-data-types>



- Variable
s



Variables

Location in memory that stores a value

- Declare with **name** and **type** before use

A variable's **name** can be any valid identifier

Series of characters consisting of letters, digits, underscores (`_`) and dollar signs (`$`)

- Does not begin with a digit, has no spaces
- Examples: Welcome1, \$value, _value, button7
 - **7button** is invalid
- Java is case sensitive (capitalization matters)
 - a1 and A1 are different

A variable's **type** specifies what kind of information is stored at that location in memory.

Variables

- Declarations end with semicolons ;
 - `int number1;`
- Can declare multiple variables of the same type at a time
- Use comma separated list
 - `int number1, number2;`
- Can add comments to describe purpose of variables
 - `int number1;                   // first number to be stored`

Variables

Variables Type (Primitive Types):

- Integers
 - can hold integer values (whole numbers such as 72, -1127 and 0)
 - The range of values for an int is -2,147,483,648 to +2,147,483,647
- Float & Double
 - hold real numbers
 - contain decimal points, such as in 3.4, 0.0 and -11.19
- Char
 - hold character data
 - represent individual characters, such as an uppercase letter (e.g., A), a digit (e.g., 7), a special character (e.g., * or %)
- Boolean
 - Hold True/False

Appendix D summarizes the characteristics of the eight primitive types (Boolean, byte, char, short, int, long, float and double).

Variables

- Integer
S

Name	Width	Range
long	64	-9,223,372,036,854,775,808 to 9,223,372,036,854,775,807
int	32	-2,147,483,648 to 2,147,483,647
short	16	-32,768 to 32,767
byte	8	-128 to 127

- Declare a variable and assign value in one statement:

```
int myNum = 15;  
System.out.println(myNum);
```

- Declare a variable and assign value in multiple statement:

```
int myNum;  
myNum = 15;  
System.out.println(myNum  
);
```

Variables

- Integers

- Declare a variable and assign value, then change to a new value:

```
int myNum = 15;  
myNum = 20; // myNum is now 20  
System.out.println(myNum);
```

- Final Variables

- this will declare the variable as "final" or "constant", which means unchangeable and read-only

```
final int myNum = 15;  
myNum = 20; // will generate an error: cannot assign a value to a final  
variable
```

Variables

- Examples

```
String myText = "Hello"; //String - Not primitive Data Type
```

Variables

- Display Variables:

- String:

```
String firstName = "John ";  
String lastName = "Doe";  
String fullName = firstName +  
lastName;  
System.out.println(fullName);
```

- Integer:

```
int x = 5; int y = 6;  
System.out.println(x + y); // Print the value of x + y
```

- Memory Concepts



Memory Concepts

- Variable names such as `number1` actually correspond to locations in the computer's memory.
- Reading variables from memory does not change them
- The number typed by the user is placed into a memory location corresponding to the name `number1`.

- `int number1`

`number1`

45

Fig. 2.8 | Memory location showing the name and value of variable `number1`.

- The computer places that integer value into location `number1`, replacing the previous value (if any) in that location.
 - `number1 = 72;`

`number1`

72

- Operators



Operators

- Operators are used to perform operations on variables and values.
 - Example: + operator to add together two values
 - `int number = 10 + 40;`
 - Example:

```
int sum1 = 100 + 50; // 150 (100 + 50)
int sum2 = sum1 + 250; // 400 (150 +
250)
int sum3 = sum2 + sum2; // 800 (400 +
400)
```

Operators

- Arithmetic

Java operation	Operator	Algebraic expression	Java expression
Addition	+	$f + 7$	<code>f + 7</code>
Subtraction	-	$p - c$	<code>p - c</code>
Multiplication	*	bm	<code>b * m</code>
Division	/	x / y or $\frac{x}{y}$ or $x \div y$	<code>x / y</code>
Remainder	%	$r \bmod s$	<code>r % s</code>

Fig. 2.11 | Arithmetic operators.

Operators

- **Arithmetic**
 - Integer division truncates remainder
7 / 5 evaluates to 1
 - Remainder operator % returns the remainder
7 % 5 evaluates to 2

Operators

- **Arithmetic**

- Parentheses for Grouping Subexpressions

- $a * (b + c)$

- It multiply a times the quantity $b + c$, we write

- $((a + b) * c)$

- the expression in the innermost set of parentheses ($a + b$ in this case) is evaluated first

- Example:

- $((3+2)*4)$

- $(5 * 4)$

- 20

Operators

- **Arithmetic**

- Rules of Operator Precedence

- Some arithmetic operators act before others (i.e., multiplication before addition)
 - Multiplication, division and remainder operations are applied first.
 - If an expression contains several such operations, they're applied from left to right.
 - Addition and subtraction operations are applied next.
 - If an expression contains several such operations, the operators are applied from left to right.

Operator(s)	Operation(s)	Order of evaluation (precedence)
* / %	Multiplication Division Remainder	Evaluated first. If there are several operators of this type, they're evaluated from <i>left to right</i> .
+ -	Addition Subtraction	Evaluated next. If there are several operators of this type, they're evaluated from <i>left to right</i> .
=	Assignment	Evaluated last.

Fig. 2.12 | Precedence of arithmetic operators.

Operators

- **Arithmetic**
 - Sample Algebraic and Java Expressions

Algebra: $m = \frac{a + b + c + d + e}{5}$
Java: `m = (a + b + c + d + e) / 5;`

Algebra: $y = mx + b$
Java: `y = m * x + b;`

Operators

- **Arithmetic**

- Sample Algebraic and Java Expressions

<i>Algebra:</i>	$z = pr \% q + w/x - y$
<i>Java:</i>	<code>z = p * r % q + w / x - y;</code>
	6 1 2 4 3 5

- The circled numbers under the statement indicate the order in which Java applies the operators. The *, % and / operations are evaluated first in left-to-right order.

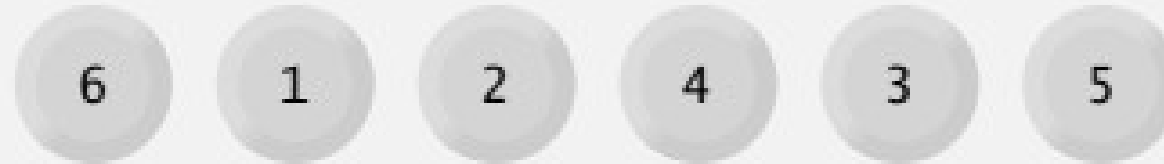
Operators

- **Arithmetic**

- Sample Algebraic and Java Expressions
 - Evaluation of a Second-Degree Polynomial

$$ax^2 + bx + c$$

```
y = a * x * x + b * x + c;
```



- More simpler writing:

```
y = (a * x * x) + (b * x) + c;
```

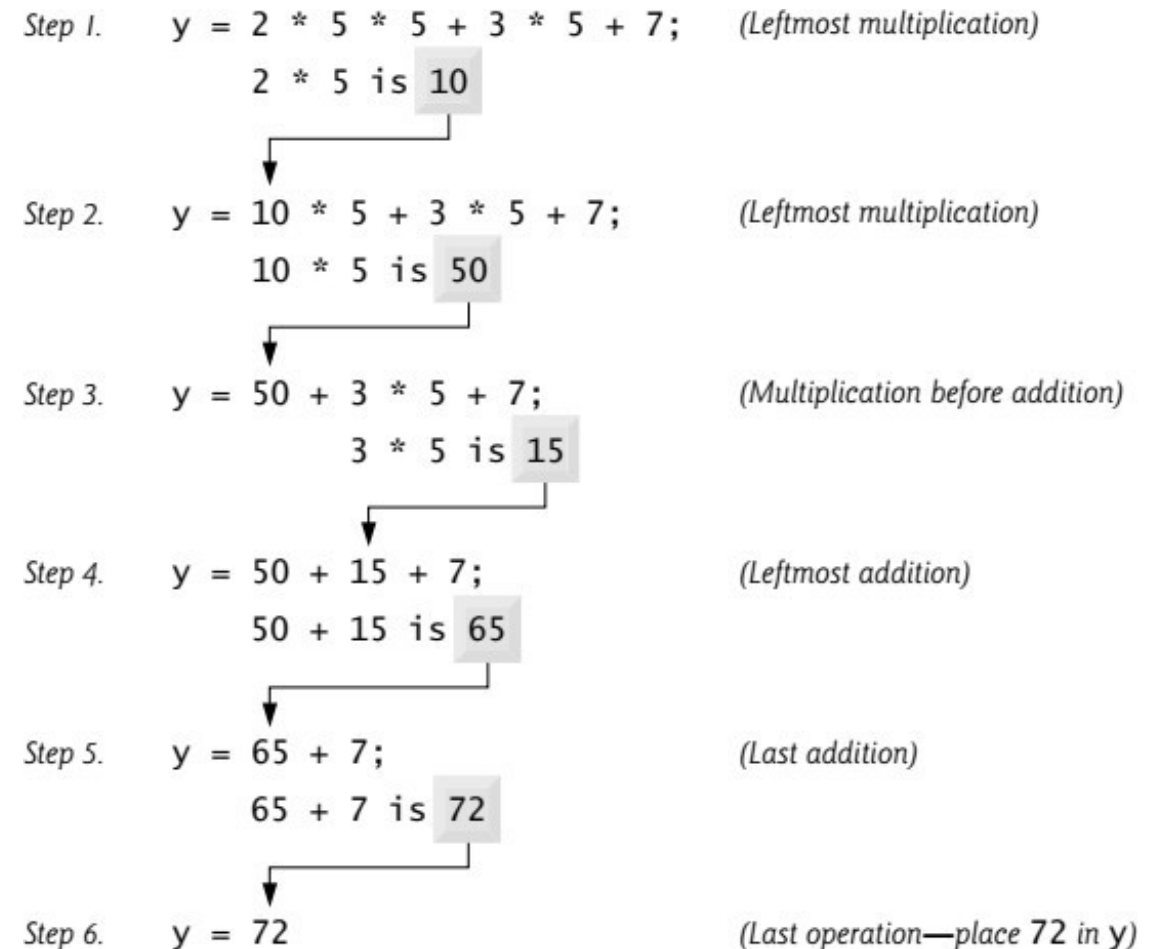

Operators

- **Arithmetic**

- Evaluate this equation:

$$y = (a * x * x) + (b * x) + c;$$

- With the following values $a = 2$,
 $b = 3$, $c = 7$ and $x = 5$.



Operators

- **Equality and Relational Operators**

- A condition is an expression that can be **true** or **false**.

Algebraic operator	Java equality or relational operator	Sample Java condition	Meaning of Java condition
<i>Equality operators</i>			
=	==	x == y	x is equal to y
1	!=	x != y	x is not equal to y
<i>Relational operators</i>			
>	>	x > y	x is greater than y
<	<	x < y	x is less than y
≥	>=	x >= y	x is greater than or equal to y
≤	<=	x <= y	x is less than or equal to y

Operators

- **Equality and Relational Operators**

- Example:

```
public class Main {  
    public static void main(String[] args) {  
        int x = 5;  
        int y = 3;  
        System.out.println(x == y); // returns false because 5 is  
        not equal to 3  
    }  
}
```

- Output: False **False**

Operators

- **Equality and Relational Operators**

- Example:

```
public class Main {  
    public static void main(String[] args) {  
        int x = 5;  
        int y = 3;  
        System.out.println(x > y); // returns true because 5 is  
        greater than 3  
    }  
}
```

- Output: True **True**

Operators

- **Equality and Relational Operators**

- Example:

```
public class Main {  
    public static void main(String[] args) {  
        int x = 5;  
        int y = 3;  
        System.out.println(x <= y); // returns false because 5 is  
        neither less than or equal to 3  
    }  
}
```

- Output: False **False**

Operators

- Operators
Precedence

Operators				Associativity	Type
*	/	%		left to right	multiplicative
+	-			left to right	additive
<	<=	>	>=	left to right	relational
==	!=			left to right	equality
=				right to left	assignment

Fig. 2.16 | Precedence and associativity of operators discussed.

- Increment/Decrement Operators



Increment/Decrement Operators

++	Increment	Increases the value of a variable by 1
----	-----------	--

- The pre increment operator is used to increment the value of some variable before using it in an expression.

```
public class IncDec {  
    public static void main(String[] args) {  
        int a, b = 15;  
        a = ++b;  
        System.out.println("a = " + a);  
    }  
}
```

- Output: 16
16

Increment/Decrement Operators

++	Increment	Increases the value of a variable by 1
----	-----------	--

- In the post increment the value is used inside the expression, then incremented by one.

```
public class IncDec {  
    public static void main(String[] args) {  
        int a, b = 15;  
        a = b++;  
        System.out.println("a = " + a);  
    }  
}
```

- Output: 15
15

Increment/Decrement Operators

--	Decrement	Decreases the value of a variable by 1
----	-----------	--

- If a decrement operator is used in front of an operand, then it is called Pre decrement operator.

```
class PreDecrement {  
    public static void main(String[]  
        args) {    int x = 10;  
    int y = --x;  
    System.out.println("y value is: " +  
        y);  
}  
}
```

- Output: **y value is: 9**

Increment/Decrement Operators

--	Decrement	Decreases the value of a variable by 1
----	-----------	--

- If a decrement operator is used after an operand, then it is called Post decrement operator.

```
class PostDecrement {  
    public static void main(String[]  
        args) {    int x = 10;  
        int y = x--;  
        System.out.println("y value is: " +  
            y);  
    }  
}
```

- Output: **y value is: 10**

- Assignment & Expressions



Assignment & Expressions

we use the assignment operator (=) to assign the value 10 to a variable called x:

- `int x = 10;`

The addition assignment operator (+=) adds a value to a variable:

- `int x = 10;`
- `x += 5;`

Assignment & Expressions

Operator	Example	Same As
=	x = 5	x = 5
+=	x += 3	x = x + 3
-=	x -= 3	x = x - 3
*=	x *= 3	x = x * 3
/=	x /= 3	x = x / 3
%=	x %= 3	x = x % 3

Reference: https://www.w3schools.com/java/java_operators.asp

- Logic and Syntax errors & Debugging concepts



Logic and Syntax errors & Debugging concepts

- Syntax errors
- These are errors where the compiler finds something wrong with your program, and you can't even try to execute it.
- For example, you may have incorrect punctuation, or may be trying to use a variable that hasn't been declared.

Logic and Syntax errors & Debugging concepts

- Syntax errors
 - Missing Parentheses
 - `System.out.print;`
 - Missing Semicolumn
 - `int x=10`

Logic and Syntax errors & Debugging concepts |

Runtime errors



If there are no syntax errors, Java may detect an error while your program is running.

You will get an error message telling you the kind of error

Logic and Syntax errors & Debugging concepts

- Runtime errors (Divide by Zero Error)

```
class DivByZero {  
    public static void main(String  
    args[])  
    {  
        int var1 15;  
        = int 5;  
        var2 = 0;  
        int var3 var1 /  
        = int var2;  
        ans1; This statement causes a runtime  
        error,  
        // as 15 is getting divided by 0 here  
        int ans2 = var1 / var3;  
  
        System.out.println(  
            "Division of  
            val"  
        + " by var2 is: "  
            + ans1);  
        System.out.printl  
        n(  
            "Division  
            of val"  
            + " by
```

Logic and Syntax errors & Debugging concepts |

Logic errors



A logic error, or bug, is when your program compiles and runs, but does the wrong thing.

The Java system, of course, has no idea what your program is supposed to do, so it provides no additional information to help you find the error.

Logic and Syntax errors & Debugging concepts

- Logic errors
 - **Using incorrect operator precedence**

```
public class OperatorError {  
    public static void main(String[] args) {  
        // Create some variables.  
        int MyVar1 = 5 + 4 * 3 / 2;  
        int MyVar2 = (5 + 4) * 3 / 2;  
        int MyVar3 = (5 + 4) * (3 / 2);  
        int MyVar4 = (5 + (4 * 3)) / 2;  
        // Output the result.  
        System.out.println( "MyVar1: " + MyVar1 + "nMyVar2: " + MyVar2 + "nMyVar3: " +  
MyVar3 + "nMyVar4: " + MyVar4);  
    }  
}
```

Reference

S

- Deitel, P. J., & Deitel, H. M. (2017). Java: how to program late objects. Pearson.

Thank You





الجامعة السعودية الإلكترونية
SAUDI ELECTRONIC UNIVERSITY
2011-1432

College of Computing and Informatics

Object Oriented Programming



Object Oriented Programming

Week 5

Control Statements I



Contents

1. Algorithms and Pseudocode
2. If statement
3. If....Else
4. Nested If Statement
5. Switch.....Case
6. Conditional Operator



Weekly Learning Outcomes



- Recognize the control statements for decision making
- Demonstrate the usage of nested if and switch statements



Required Reading

1. Chapter 3 - (Java: How to program: Late Objects)

Recommended Reading

1. Chapter 5 - (Java The Complete Reference, Eleventh Edition)
2. Java if...else Statement: <https://www.programiz.com/java-programming/if-else-statement>
3. The switch Statement:
<https://docs.oracle.com/javase/tutorial/java/nutsandbolts/switch.html>



- Algorithms and Pseudocode



Algorithms and Pseudocode

- A procedure for solving a problem in terms of
 1. the actions to execute and
 2. the order in which these actions execute
- called an **algorithm**.

Algorithms and Pseudocode

- **Algorithm Example: Add two numbers entered by the user**

Step 1: Start

Step 2: Declare variables num1, num2 and sum. Step 3: Read values num1 and num2.

Step 4: Add num1 and num2 and assign the result to sum. $\text{sum} \leftarrow \text{num1} + \text{num2}$

Step 5: Display sum Step 6: Stop

Reference: <https://www.programiz.com/dsa/algorithm>

Algorithms and Pseudocode

- **Pseudocode** is an informal language that helps you develop algorithms without having to worry about the strict details of Java language syntax.
- **Pseudocode** does not execute on computers.

•

Algorithms and Pseudocode

- **Pseudocode** Example:
 - If student's grade is greater than or equal to 60
 - Print "passed"
 - else
 - Print "failed"

Control Structure

- Sequential execution
 - Program statements execute one after the other

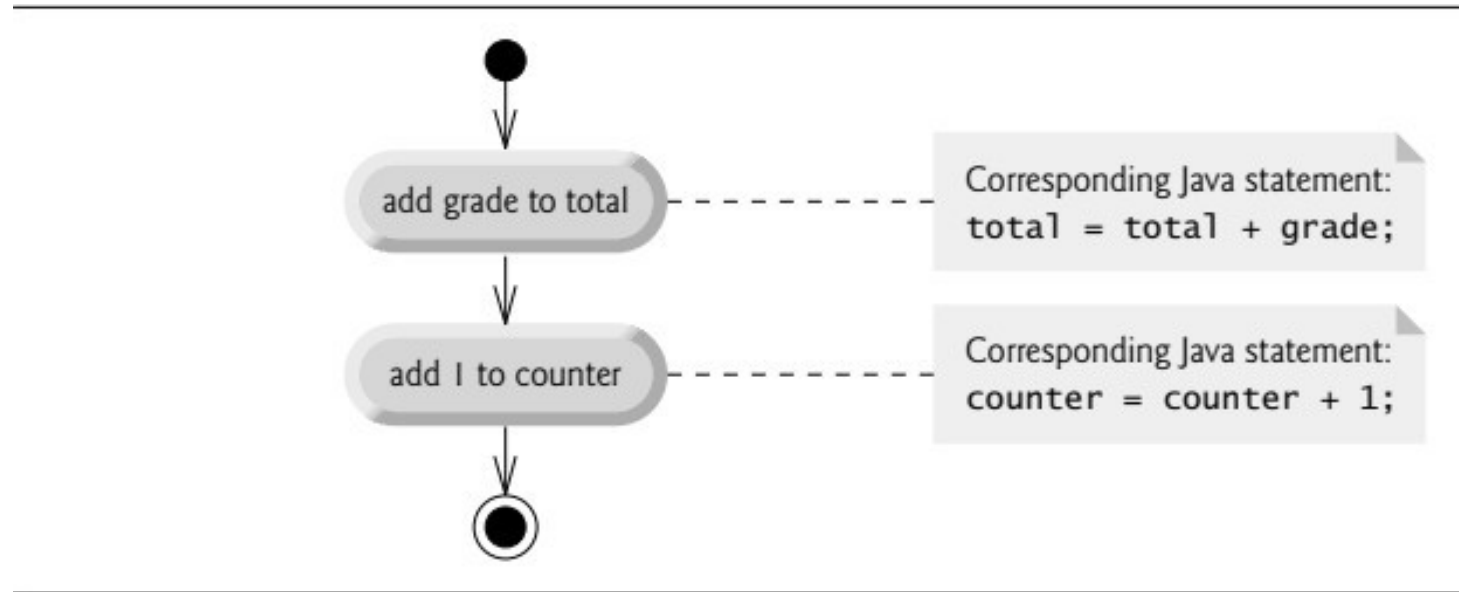


Fig. 3.1 | Sequence-structure activity diagram.

Control Structure



- Transfer of control
 - Three control statements can specify order of statements
 - Sequence structure
 - Selection structure
 - Repetition structure

Control Structure

Transfer of control

- Java has a sequence structure “built-in”
- Java provides three selection structures
 - if
 - If...else
 - switch
- Java provides three repetition structures
 - while
 - do...while
 - do

Each of these words is a Java keyword

- If statement



If statement

- Single-entry/single-exit control structure
- Perform action only when condition is true

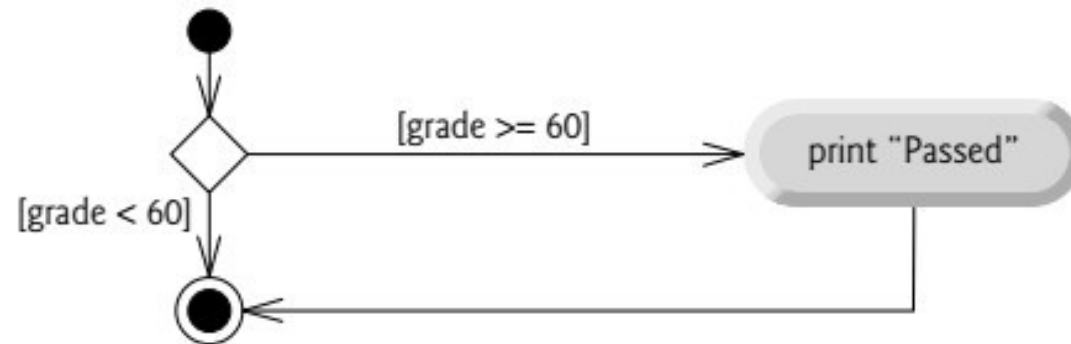


Fig. 3.2 | if single-selection statement UML activity diagram.

If statement

- Pseudocode
 - If student's grade is greater than or equal to 60
Print "Passed"
- Java Code
 - ***if*** (studentGrade >= 60)
System.out.println("***Passed***");

If statement

- Example:

//Java Program to demonstrate the use of if statement.

```
public class IfExample {  
    public static void main(String[] args) {  
        //defining an 'age' variable  
        int age=20;  
        //checking the age  
        if(age>18){  
            System.out.print("Age is greater than 18");  
        }  
    }  
}
```

- If....Else



If...Else

- Perform action only when condition is true
- Perform different specified action when condition is false

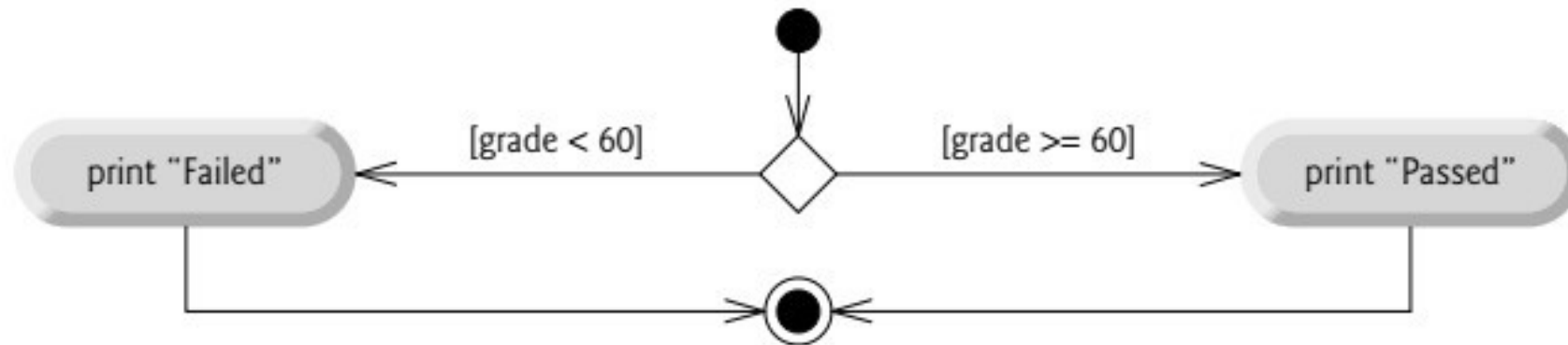


Fig. 3.3 | if...else double-selection statement UML activity diagram.

If....Else

- Pseudocode
 - If student's grade is greater than or equal to 60
Print "Passed"
 - Else
Print "Failed"
- Java Code
 - *if* (grade >= 60)
System.out.println("**Passed**");
 - *else*
System.out.println("**Failed**");

If....Else

- Example:

```
public class IfElseExample {  
    public static void main(String[] args) {  
        //defining a variable  
        int number=13;  
        //Check if the number is divisible by 2 or not  
        if(number%2==0)  
            { System.out.println("even  
            number");  
        }else{  
            System.out.println("odd number");  
        }  
    }  
}
```

- Nested If Statement



Nested If Statement

- A program can test multiple cases by placing if...else statements inside other if...else statements to create nested **if...else** statements.

- Pseudocode
 - If student's grade is greater than or equal to 90
Print "A"
 - else
 - If student's grade is greater than or equal to 80
Print "B"
 - else
 - If student's grade is greater than or equal to 70
Print "C"
 - else
 - If student's grade is greater than or equal to 60
Print "D"
 - else
Print "E"

Nested If Statement

- Java Code

```
    if (studentGrade >= 90)
```

```
        System.out.println("A");
```

```
    Else
```

```
        if (studentGrade >= 80)
```

```
            System.out.println("B");
```

```
        Else
```

```
            if (studentGrade >= 70)
```

```
                System.out.println("C");
```

```
            Else
```

```
                if (studentGrade >= 60)
```

```
                    System.out.println("D");
```

```
                Else
```

```
                    System.out.println("F");
```


Nested If Statement

- Simplify the Java Code

```
    if (studentGrade >= 90)  
        System.out.println("A")  
        ;
```

```
    else if (studentGrade >= 80)  
        System.out.println("B")  
        ;
```

```
else if (studentGrade >= 70)  
    System.out.println("C");
```

```
    else if (studentGrade >= 60)  
        System.out.println("D")  
        ;
```

```
else
```

Nested If Statement

- Dangling-**else** Problem
- Java compiler always associates an else with the immediately preceding if unless told to do otherwise by the placement of braces { }

```
if (x > 5)
    if (y > 5)
        System.out.println("x and y are > 5");
else
    System.out.println("x is <= 5");
```

- The compiler will interpret it as the following:

```
if (x > 5)
    if (y > 5)
        System.out.println("x and y are > 5");
    else
        System.out.println("x is <= 5");
```

Nested If Statement

- Dangling-**else** Problem
- To force the nested if...else statement to execute as it was originally intended.

```
if (x > 5)
{
    if (y > 5)
        System.out.println("x and y are > 5");
}
else
    System.out.println("x is <= 5");
```

Nested If Statement

- Blocks
- The if statement normally expects only one statement in its body.
- To include several statements in the body of an if enclose the statements in braces.

```
if (grade >= 60)
    System.out.println("Passed");
else
{
    System.out.println("Failed");
    System.out.println("You must take this course again.");
}
```

- Switch....Case



Switch.....Case

The **switch** multiple-selection statement performs different actions based on the possible values of a constant integral expression

The case values must be unique.

The Java switch expression must be of byte, short, int, long, enums and

string. Each case statement can have a break statement which is optional.

- It jumps the control after the switch expression.
- If a break statement is not found, it executes the next case.

The case value can have a default label which is optional.

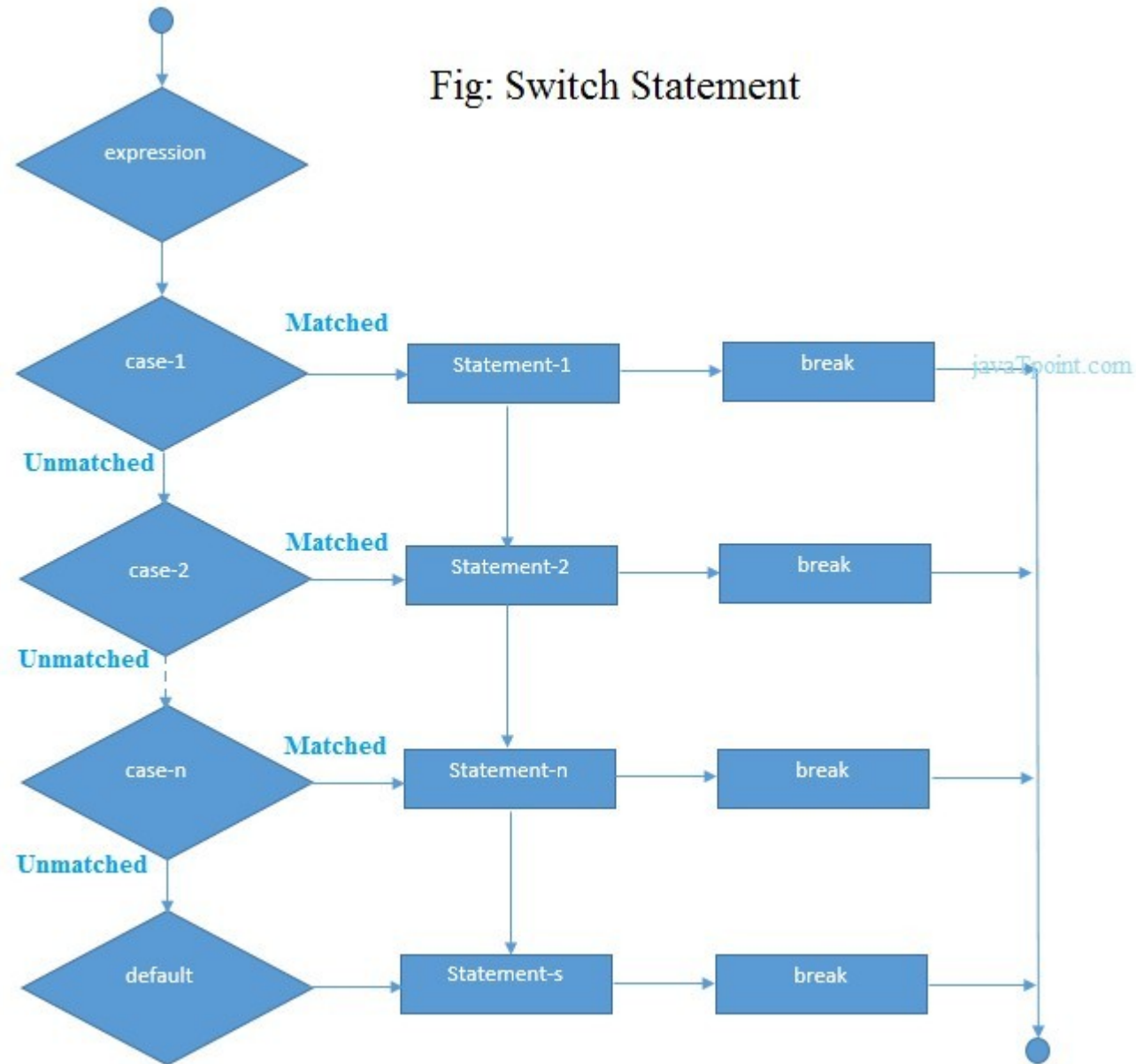
Switch....Case

```
switch(expression){  
    case value1:  
        //code to be executed;  
        break;                //optional  
    case value2:  
        //code to be executed;  
        break;                //optional  
    .....  
    default:  
        code to be executed if all cases are not matched;  
}
```

Reference: <https://www.javatpoint.com/java-switch>

Switch.....Case

Fig: Switch Statement



Reference: <https://www.javatpoint.com/java-switch>

Switch.....Case

- Example:

```
public class SwitchExample {  
    public static void main(String[] args) {  
        //Declaring a variable for switch expression  
        int number=20;  
        //Switch expression  
        switch(number){  
            //Case statements  
            case 10: System.out.println("10");  
                break;  
            case 20: System.out.println("20");  
                break;  
            case 30: System.out.println("30");  
                break;  
            //Default case statement  
            default: System.out.println("Not in 10, 20 or 30");  
        }  
    }  
}
```

Reference: <https://www.javatpoint.com/java-switch>

Switch....Case

- The Java switch statement is fall-through. It means it executes all statements after the first match if a break statement is not present.

```
public class SwitchExample2 {  
    public static void main(String[] args) {  
        int number=20;  
        //switch expression with int value  
        switch(number){  
            //switch cases without break  
            statements  
            case 10:  
                System.out.println("10");  
            case 20:  
                System.out.println("20");  
            case 30:  
                System.out.println("30");  
            default: System.out.println("Not in 10, 20 or 30")  
        }  
    }  
}
```

```
20  
30  
Not in 10, 20 or 30
```

Switch....Case

- Example:

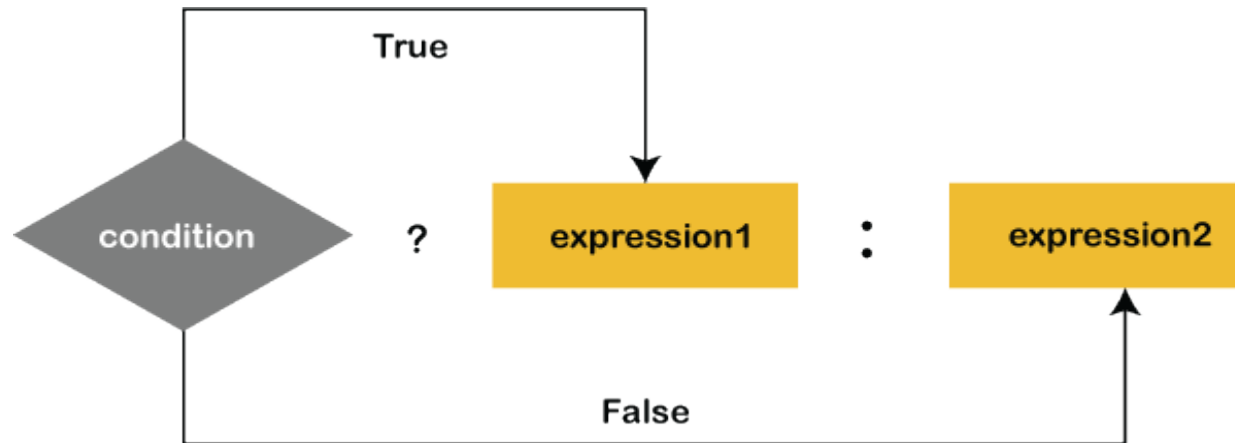
```
public class Test {  
    public static void main(String args[]) {  
        char grade = 'C';  
        switch(grade) {  
            case 'A' : System.out.println("Excellent!");  
                        break;  
  
            case 'B' :  
            case 'C' : System.out.println("Well done");  
                        break;  
  
            case 'D' : System.out.println("You passed");  
            case 'F' : System.out.println("Better try again");  
                        break;  
  
            default : System.out.println("Invalid grade");  
        }  
        System.out.println("Your grade is " + grade);  
    }  
}
```

- Conditional Operator



Conditional Operator (?:)

- It can be used in place of an if...else statement.
- This can make your code shorter and clearer.
- The conditional operator is Java's only ternary operator.



Reference: <https://www.javatpoint.com/conditional-operator-in-java>

Conditional Operator (?:)

- Example:

```
public class condiop {  
    public static void main(String[] args) {  
        String out;  
        int a = 6, b = 12;  
        out = a==b ? "Yes":"No";  
        System.out.println("Ans: "+out);  
    }  
}
```

Reference: <https://www.w3schools.in/java-tutorial/operators/conditional/>

Conditional Operator (?:)

- Example:

```
public class LargestNumberExample {  
    public static void main(String args[]) {  
        int x=69;  
        int  
        y=89;  
        int z=79;  
        int largestNumber= (x > y) ? (x > z ? x : z) : (y > z ? y : z);  
        System.out.println("The largest numbers is: "+largestNumber);  
    }  
}
```

Reference: <https://www.javatpoint.com/conditional-operator-in-java>

Reference

S

- Deitel, P. J., & Deitel, H. M. (2017). Java: how to program late objects. Pearson.

Thank You





الجامعة السعودية الإلكترونية
SAUDI ELECTRONIC UNIVERSITY
2011-1432

College of Computing and Informatics

Object Oriented Programming



Object Oriented Programming
Week 6
Control Statements II



Contents

1. For / While Loops
2. Nested Loops
3. Break and Continue Statements
4. Logical Operator



Weekly Learning Outcomes



- Operate different Loops statements techniques
- Differentiate between Break and Continue statements



Required Reading

1. Chapter 4 - (Java: How to program: Late Objects)

Recommended Reading

1. Chapter 5 - (Java The Complete Reference, Eleventh Edition)
2. Java Loops Tutorial: <https://www.learnjavaonline.org/en/Loops>
3. Branching Statements:
<https://docs.oracle.com/javase/tutorial/java/nutsandbolts/branch.html>



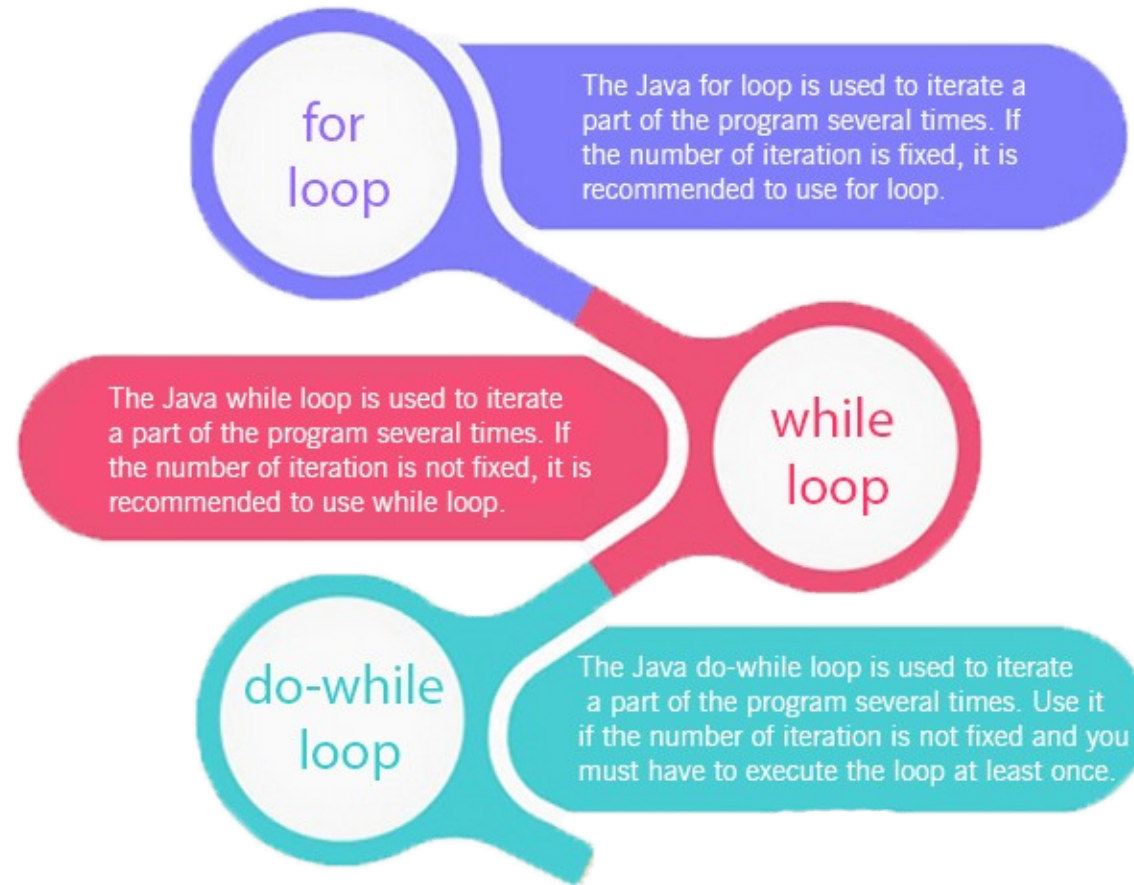
- For / While Loops



For / While Loops

- A repetition statement allows you to specify that a program should repeat an action.
- Loops can execute a block of code as long as a specified condition is reached.
- Loops are handy because they save time, reduce errors, and they make code more readable.

For / While Loops



Reference: <https://www.javatpoint.com/java-for-loop>

For / While Loops

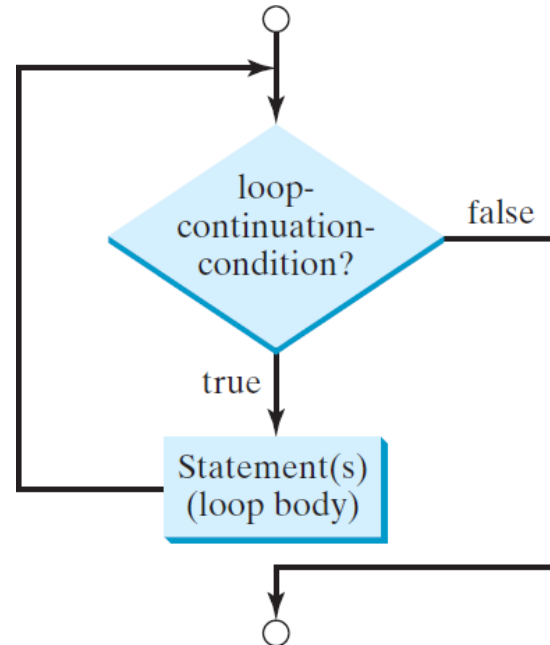
- Loops Requirements:
 1. A control variable (or loop counter)
 2. The initial value of the control variable
 3. The increment by which the control variable is modified each time through the loop (also known as each iteration of the loop)
 4. The loop-continuation condition that determines if looping should continue.

For / While Loops

- **while** Repetition Statement

- The pseudocode statement

*While there are more items on my shopping list
Purchase next item and cross it off my list*



For / While Loops

```
int counter = 1;           // declare and initialize control variable
while (counter <= 10 ) {   // loop-continuation condition
    System.out.printf("%d ", counter);
    ++counter;             // increment control
    variable
}
System.out.println();
```

1	2	3	4	5	6	7	8	9	10
---	---	---	---	---	---	---	---	---	----

For / While Loops

- **for Repetition Statement**

- specifies the counter-controlled- details in a single line of repetition code.

```
for (initialization; loopContinuationCondition; increment)  
    statement
```

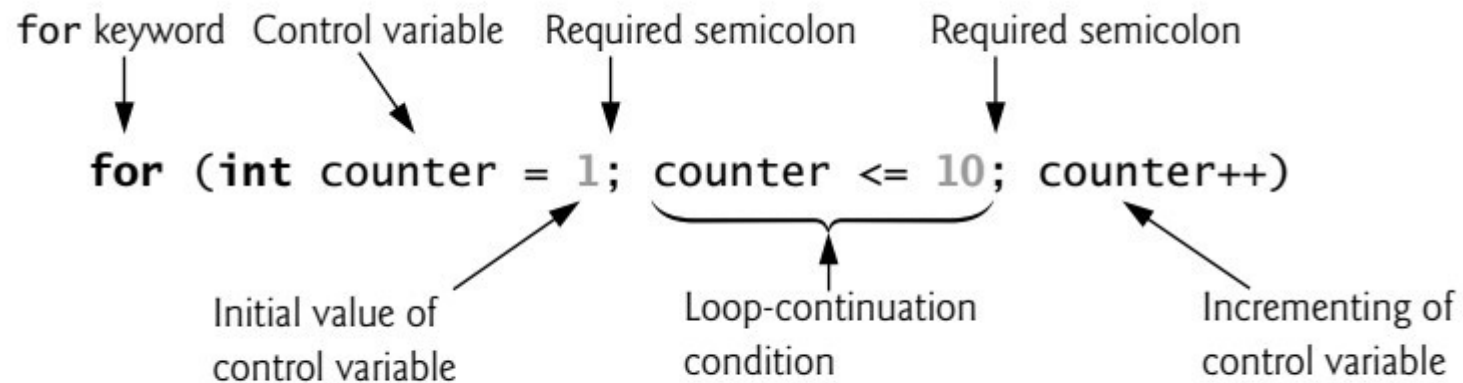


Fig. 4.3 | for statement header components.

For / While Loops

- **for** Repetition Statement

```
1  // Fig. 4.2: ForCounter.java
2  // Counter-controlled repetition with the for repetition statement.
3
4  public class ForCounter
5  {
6      public static void main(String[] args)
7      {
8          // for statement header includes initialization,
9          // loop-continuation condition and increment
10         for (int counter = 1; counter <= 10; counter++)
11             System.out.printf("%d  ", counter);
12
13         System.out.println();
14     }
15 } // end class ForCounter
```

1 2 3 4 5 6 7 8 9 10

Fig. 4.2 | Counter-controlled repetition with the for repetition statement.

For / While Loops

- **for** Repetition Statement

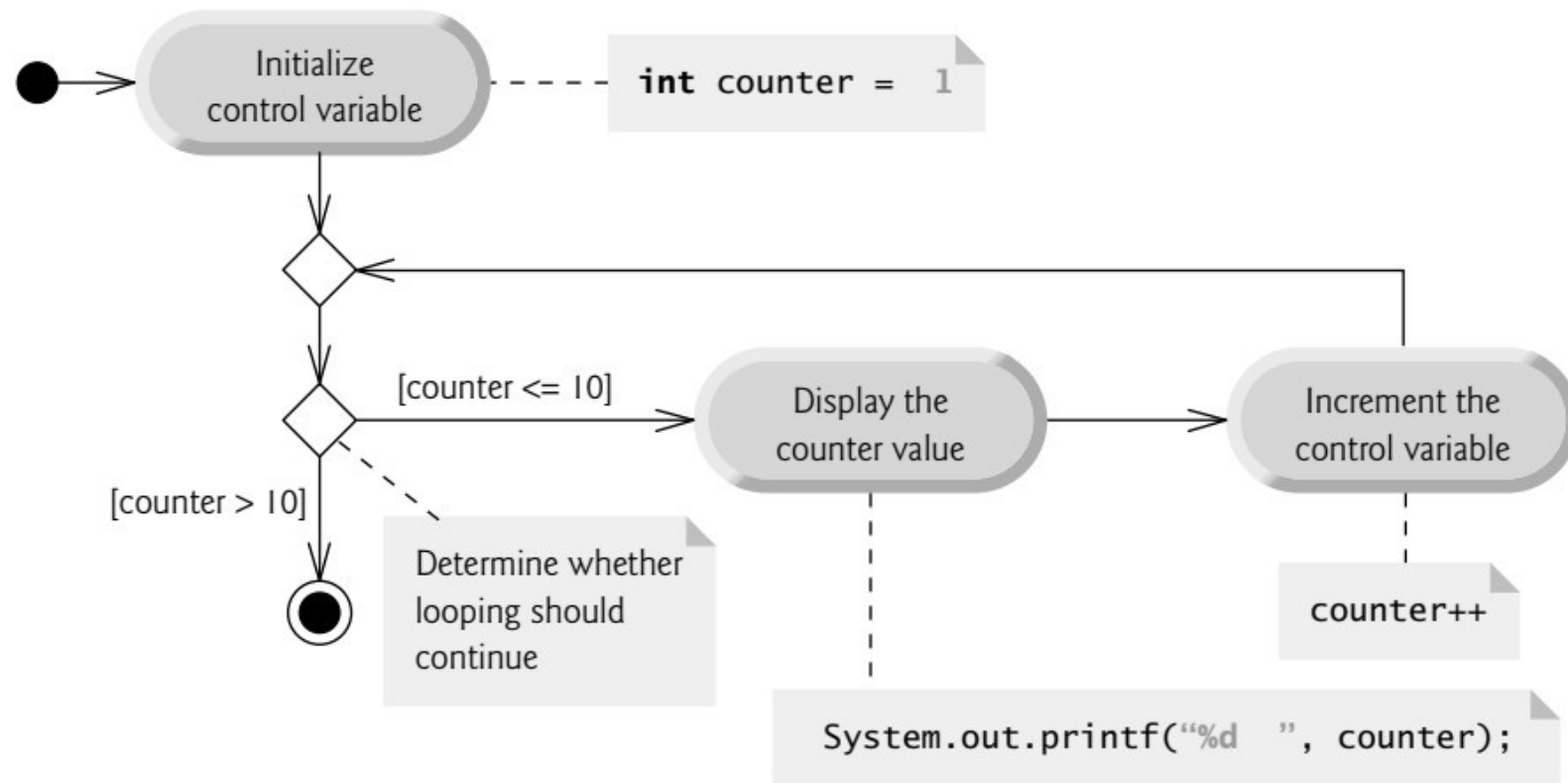


Fig. 4.4 | UML activity diagram for the for statement in Fig. 4.2.

For / While Loops

- **for Repetition Statement**

- The increment expression in a for acts as if it were a standalone statement at the end of the for's body. Therefore, the expressions.

```
counter = counter + 1  
counter += 1  
++counter  
counter++
```

For / While Loops

- **for Repetition Statement**

- Placing Arithmetic Expressions in a **for** Statement's Header

- For example, assume that $x = 2$ and $y = 10$. If x and y are not modified in the body of the loop, the statement

```
for (int j = x; j <= 4 * x * y; j += y / x)
```

- is equivalent to the statement

```
for (int j = 2; j <= 80; j += 5)
```

For / While Loops

- Examples Using the for Statement

- a) Vary the control variable from 1 to 100 in increments of 1.

```
for (int i = 1; i <= 100; i++)
```

- b) Vary the control variable from 100 to 1 in *decrements* of 1.

```
for (int i = 100; i >= 1; i--)
```

- c) Vary the control variable from 7 to 77 in increments of 7.

```
for (int i = 7; i <= 77; i += 7)
```

- d) Vary the control variable from 20 to 2 in *decrements* of 2.

```
for (int i = 20; i >= 2; i -= 2)
```

- e) Vary the control variable over the values 2, 5, 8, 11, 14, 17, 20.

```
for (int i = 2; i <= 20; i += 3)
```

- f) Vary the control variable over the values 99, 88, 77, 66, 55, 44, 33, 22, 11, 0.

```
for (int i = 99; i >= 0; i -= 11)
```

For / While Loops

- Summing the Even Integers from 2 to 20

```
1  // Fig. 4.5: Sum.java
2  // Summing integers with the for statement.
3
4  public class Sum
5  {
6
7      public static void main(String[] args)
8      {
9
10         // total even integers from 2 through 20
11         for (int number = 2; number <= 20; number += 2)
12             total += number;
13
14         System.out.printf("Sum is %d\n", total);
15     }
16 } // end class Sum
```

Output

Sum is 110

For / While Loops

- **do...while Repetition Statement**
- In the **while**, the program tests the loop-continuation condition
 - at the beginning of the loop, before executing the loop's body;
 - if the condition is false, the body never executes.
- The **do...while** statement tests the loop continuation condition
 - after executing the loop's body;
 - therefore, the body always executes at least once.

For / While Loops

- **do...while** Repetition Statement

```
1 // Fig. 4.7: DoWhileTest.java
2 // do...while repetition statement.
3
4 public class DoWhileTest
5 {
6     public static void main(String[] args)
7     {
8         int counter = 1;
9
10        do
11        {
12            System.out.printf("%d ", counter);
13            ++counter;
14        } while (counter <= 10); // end do...while
15
16        System.out.println();
17    }
18 } // end class DoWhileTest
```

1	2	3	4	5	6	7	8	9	10
---	---	---	---	---	---	---	---	---	----

For / While Loops

- **do...while** Repetition Statement

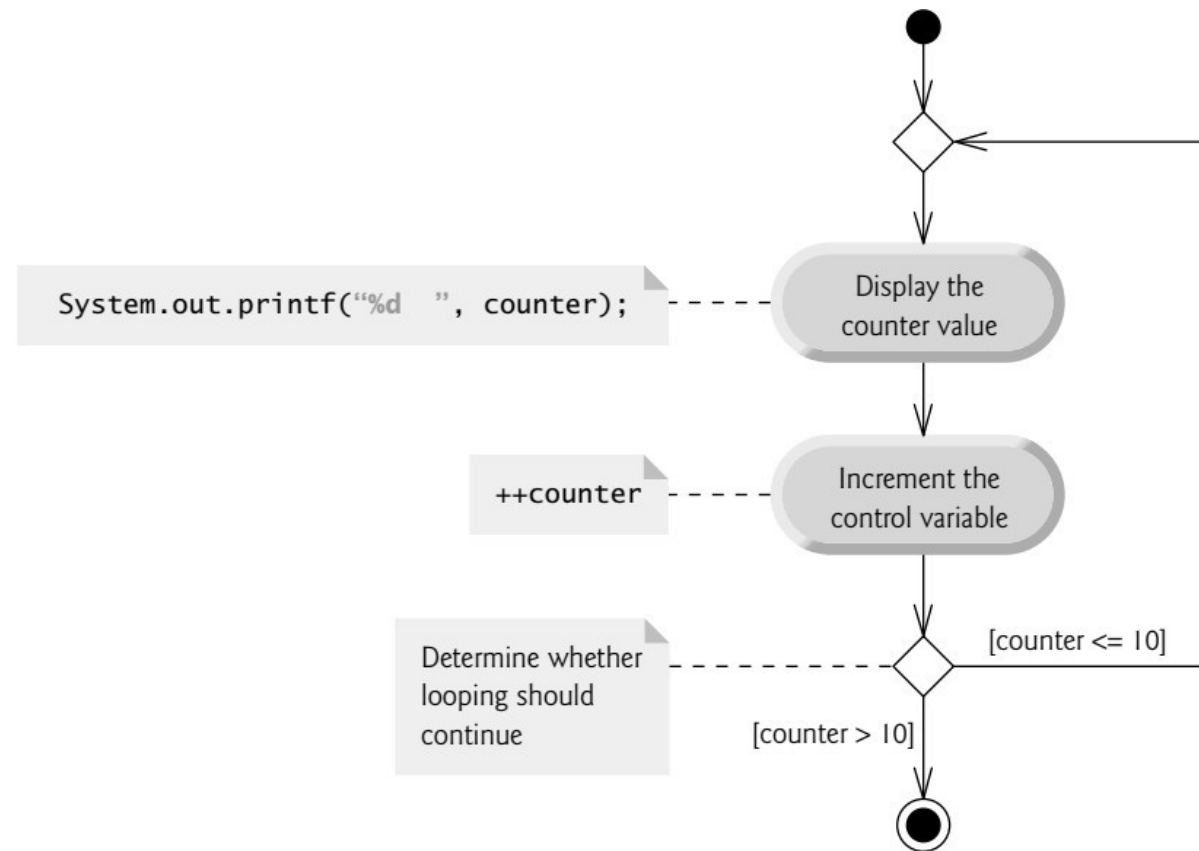


Fig. 4.8 | do...while repetition statement UML activity diagram.

For / While Loops

- Comparison

Comparison	for loop	while loop	do while loop
Introduction	The Java for loop is a control flow statement that iterates a part of the programs multiple times.	The Java while loop is a control flow statement that executes a part of the programs repeatedly on the basis of given boolean condition.	The Java do while loop is a control flow statement that executes a part of the programs at least once and the further execution depends upon the given boolean condition.
When to use	If the number of iteration is fixed, it is recommended to use for loop.	If the number of iteration is not fixed, it is recommended to use while loop.	If the number of iteration is not fixed and you must have to execute the loop at least once, it is recommended to use the do-while loop.
Syntax	<pre>for(init;condition;incr/decr){ // code to be executed }</pre>	<pre>while(condition){ //code to be executed }</pre>	<pre>do{ //code to be executed }while(condition);</pre>
Example	<pre>//for loop for(int i=1;i<=10;i++){ System.out.println(i); }</pre>	<pre>//while loop int i=1; while(i<=10){ System.out.println(i); i++; }</pre>	<pre>//do-while loop int i=1; do{ System.out.println(i); i++; }while(i<=10);</pre>
Syntax for infinitive loop	<pre>for(;;){ //code to be executed }</pre>	<pre>while(true){ //code to be executed }</pre>	<pre>do{ //code to be executed }while(true);</pre>

- Nested Loops



Nested Loops

- If a loop exists inside the body of another loop, it's called a nested loop.

```
// outer loop
for (int i = 1; i <= 5; ++i) {
    // codes

    // inner loop
    for(int j = 1; j <= 2; ++j) {
        // codes
    }
    ..
}
```

Reference: <https://www.programiz.com/java-programming/nested-loop>

Nested Loops

- Example 1:

```
class Main {  
    public static void main(String[] args) {  
  
        int weeks = 3;  
        int days = 7;  
  
        // outer loop prints weeks  
        for (int i = 1; i <= weeks; ++i) {  
            System.out.println("Week: " + i);  
  
            // inner loop prints days  
            for (int j = 1; j <= days; ++j) {  
                System.out.println("    Day: " + j);  
            }  
        }  
    }  
}
```

Output

```
Week: 1  
    Day: 1  
    Day: 2  
    Day: 3  
    ..... ..  
Week: 2  
    Day: 1  
    Day: 2  
    Day: 3  
    ..... ..  
..... ..
```

Nested Loops

- Example 2:

```
class Main {  
    public static void main(String[] args) {  
  
        int weeks = 3;  
        int days = 7;  
        int i = 1;  
  
        // outer loop  
        while (i <= weeks) {  
            System.out.println("Week: " + i);  
  
            // inner loop  
            for (int j = 1; j <= days; ++j) {  
                System.out.println("    Days: " + j);  
            }  
            ++i;  
        }  
    }  
}
```

Output

```
Week: 1  
    Day: 1  
    Day: 2  
    Day: 3  
        ..... ..  
Week: 2  
    Day: 1  
    Day: 2  
    Day: 3  
        .....  
..... ..
```

Nested Loops

- Example 3:

```
class Main {  
    public static void main(String[] args) {  
  
        int rows = 5;  
  
        // outer loop  
        for (int i = 1; i <= rows; ++i) {  
  
            // inner loop to print the numbers  
            for (int j = 1; j <= i; ++j) {  
                System.out.print(j + " ");  
            }  
            System.out.println("");  
        }  
    }  
}
```

Output

```
1  
1 2  
1 2 3  
1 2 3 4  
1 2 3 4 5
```

- Break and Continue Statements



Break and Continue Statements

- ***break*** Statement
 - When executed in a while, for, do...while or switch,
 - causes immediate exit from that statement.
 - Execution continues with the first statement after the control statement.
 - Common uses of the break statement are
 - to escape early from a loop
 - to skip the remainder of a switch

Break and Continue Statements

- ***break***

```
1 // Fig. 4.11: BreakTest.java
2 // break statement exiting a for statement.
3 public class BreakTest
4 {
5     public static void main(String[] args)
6     {
7         int count; // control variable also used after loop terminates
8
9         for (count = 1; count <= 10; count++) // loop 10 times
10        {
11            if (count == 5)
12                break; // terminates loop if count is 5
13
14            System.out.printf("%d ", count);
15        }
16
17        System.out.printf("\nBroke out of loop at count = %d\n", count);
18    }
19 } // end class BreakTest
```

Output

```
1 2 3 4
Broke out of loop at count = 5
```


Break and Continue Statements

- ***continue*** Statement
 - when executed in a while, for or do...while,
 - skips the remaining statements in the loop body
 - and proceeds with the next iteration of the loop.
 - In **while** and **do...while** statements, the program evaluates the loop continuation test immediately after the continue statement executes.
 - In a **for** statement, the increment expression executes, then the program evaluates the loop continuation test.

Break and Continue Statements

- **continue**

```
1 // Fig. 4.12: ContinueTest.java
2 // continue statement terminating an iteration of a for statement.
3 public class ContinueTest
4 {
5     public static void main(String[] args)
6     {
7         for (int count = 1; count <= 10; count++) // loop 10 times
8         {
9             if (count == 5)
10                continue; // skip remaining code in loop body if count is 5
11
12             System.out.printf("%d ", count);
13         }
14
15         System.out.printf("\nUsed continue to skip printing 5\n");
16     }
17 } // end class ContinueTest
```

Output

```
1 2 3 4 6 7 8 9 10
Used continue to skip printing 5
```

- Logical Operator



Logical Operator

- Enable you to form more complex conditions by combining simple conditions.
- The logical operators are
 - && (conditional AND)
 - || (conditional OR)
 - & (boolean logical AND)
 - | (boolean logical inclusive OR)
 - ^ (Boolean logical exclusive OR)
 - ! (logical NOT).

Logical Operator

- Conditional AND (&&) Operator

- To ensure at some point in a program that two conditions are both true before we choose a certain path of execution.

```
if (gender == FEMALE && age >= 65)
    ++seniorFemales;
```

- The if statement condition is true if and only if both simple conditions are true.

Logical Operator

- Conditional AND (&&) Operator

expression1	expression2	expression1 && expression2
false	false	false
false	true	false
true	false	false
true	true	true

Fig. 4.13 | && (conditional AND) operator truth table.

Logical Operator

- Conditional OR (||)

Operator We use that either or both of two conditions are true before choose a certain path of execution.

```
if ((semesterAverage >= 90) || (finalExam >= 90))  
    System.out.println ("Student grade is A");
```

- The if statement condition awards the student an A if either or both of the simple conditions are true.

Logical Operator

- Conditional OR (||)
Operator

expression1	expression2	expression1 expression2
false	false	false
false	true	true
true	false	true
true	true	true

Fig. 4.14 | || (conditional OR) operator truth table.

Logical Operator

- Boolean Logical AND (&) and Boolean Logical Inclusive OR (|) Operators
 - are identical to the && and || operators, except that the & and | operators always evaluate both of their operands.

```
(gender == 1) & (age >= 65)
```

- evaluates age >= 65 regardless of whether gender is equal to 1. This is useful if the right operand has a required side effect.

```
(birthday == true) | (++age >= 65)
```

- guarantees that the condition ++age >= 65 will be evaluated.
 - Thus, the variable **age is incremented**, regardless of whether the overall expression is true or false.

Logical Operator

- Boolean Logical Exclusive OR (^)
 - is true if and only if one of its operands is true and the other is false.

expression1	expression2	expression1 ^ expression2
false	false	false
false	true	true
true	false	true
true	true	false

Fig. 4.15 | ^ (boolean logical exclusive OR) operator truth table.

Logical Operator

- Logical Negation (!) Operator

- This operator “reverses” the meaning of a condition.

```
if (! (grade == sentinelValue))  
    System.out.printf("The next grade is %d%n", grade);
```

- which executes the printf call only if grade is not equal to sentinelValue.

```
if (grade != sentinelValue)  
    System.out.printf("The next grade is %d%n", grade);
```

expression	! expression
false	true
true	false

Fig. 4.16 | ! (logical NOT) operator truth table.

Logical Operator

- Precedence and Associativity of the Operators Presented So Far

Operators	Associativity	Type
++ --	right to left	unary postfix
++ -- + - ! (type)	right to left	unary prefix
* / %	left to right	multiplicative
+ -	left to right	additive
< <= > >=	left to right	relational
== !=	left to right	equality
&	left to right	boolean logical AND
^	left to right	boolean logical exclusive OR
	left to right	boolean logical inclusive OR
&&	left to right	conditional AND
	left to right	conditional OR
?:	right to left	conditional
= += -= *= /= %=	right to left	assignment

Fig. 4.18 | Precedence/associativity of the operators discussed so far.

Reference

S

- Deitel, P. J., & Deitel, H. M. (2017). Java: how to program late objects. Pearson.

Thank You





الجامعة السعودية الإلكترونية
SAUDI ELECTRONIC UNIVERSITY
2011-1432

College of Computing and Informatics

Object Oriented Programming



Object Oriented Programming

Week 7
Methods
(Part 1)



Contents

1. Modules Approach
2. Declaring Methods
3. Arguments and Casting



Weekly Learning Outcomes



- Describe Methods and calling stack in Java



Required Reading

1. Chapter 5 - (Java: How to program: Late Objects)

Recommended Reading

1. Chapter 6 - (Java The Complete Reference, Eleventh Edition)

2. Defining Methods:

<https://docs.oracle.com/javase/tutorial/java/javaOO/methods.html>

3. Java API Packages: <https://www.javatpoint.com/package>



- Modules Approach



Modules Approach

- Suppose you want to build a program as follows:

```
int sum = 0;  
for (int i = 1; i <= 10; i++)  
    sum += i;
```

```
System.out.println("Sum from 1 to 10 is " + sum);
```

```
sum = 0;  
for (int i = 20; i <= 30; i++)  
    sum += i;
```

```
System.out.println("Sum from 20  
to 30 is " + sum);
```

```
sum = 0;  
for (int i = 35; i <= 45; i++)  
    sum += i;
```

```
System.out.println("Sum from 35
```

Modules Approach

- Defining a method will make you code more simpler

```
public static int sum(int i1, int i2) {  
    int sum = 0;  
    for (int i = i1; i <= i2; i+  
        +) sum += i;  
    return sum;  
}
```

```
public static void main(String[] args)  
{  
    System.out.println("Sum from 1 to 10 is " + sum(1, 10));  
    System.out.println("Sum from 20 to 30 is " + sum(20,  
30)); System.out.println("Sum from 35 to 45 is " +  
sum(35, 45));  
}
```

Modules Approach



Java programs by combining new methods and classes with predefined ones available in the Java Application Programming Interface.



Methods help you modularize a program by separating its tasks into self-contained units.

Divide and Conquer

- It makes program development more manageable by constructing programs from small, simple pieces.
- Software reusability—using existing methods as building blocks to create new programs.

Modules Approach

- Hierarchical Relationship Between Method Calls
 - A method is invoked by a method call
 - when the called method completes its task
 - it returns control—and possibly a result—to the caller.

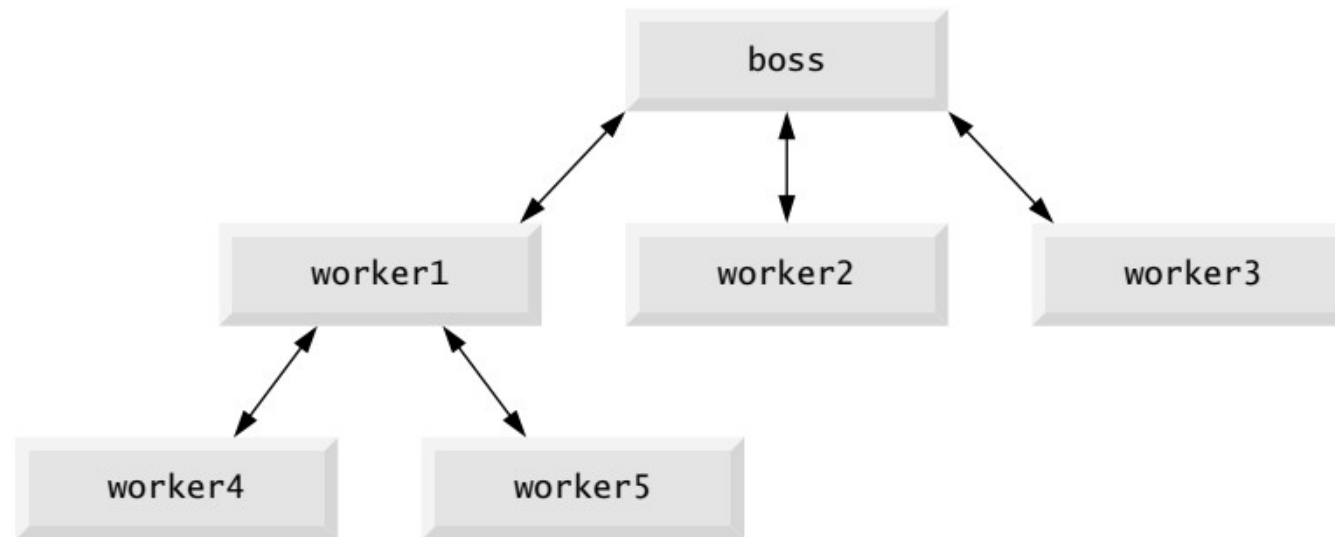


Fig. 5.1 | Hierarchical boss-method/worker-method relationship.

- Declaring Methods



Declaring Methods

```
public static int methodName(int a, int b) {  
    // body  
}
```

public static – modifier

//it will be discussed in detail in week 9

int – return type

methodName – name of the method

a, b – formal parameters

int a, int b – list of parameters

Declaring Methods

- The method name should be a **verb** and start with a **lowercase** letter.
 - If the method name has more than two words, the first name must be a verb followed by adjective or noun.
 - In the multi-word method name, the first letter of each word must be in **uppercase** except the first word. For example:
- **Single-word method name:** sum(), area()
- **Multi-word method name:** areaOfCircle(), stringComparision()

Declaring Methods

- Types of Method
 - Predefined Method
 - User-defined Method

Declaring Methods

- Predefined Method
 - are the method that is already defined in the Java class libraries.
 - known as the **standard library method** or **built-in method**.

```
public class Demo
{
    public static void main(String[] args)
    {
        // using the max() method of Math class
        System.out.print("The maximum number is: " + Math.max(9,7));
    }
}
```

Declaring Methods

- User-defined Method
 - The method written by the user or programmer

```
import java.util.Scanner;
public class EvenOdd {
    public static void main (String args[]) {
        //creating Scanner class object
        Scanner scan=new Scanner(System.in);
        System.out.print("Enter the number: ");
        //reading value from the user
        int num=scan.nextInt();
        //method calling
        findEvenOdd(num);
    }
}
```

```
//user defined method
public static void findEvenOdd(int num)
{
    //method body
    if(num%2==0)
        System.out.println(num+" is even");
    else
        System.out.println(num+" is odd");
}
```

Declaring Methods

- Example:

```
public class Addition {  
    public static void main(String[] args) {  
        int a = 19;  
        int b = 5;  
        //method calling  
        int c = add(a, b); //a and b are  
        //actual parameters  
        System.out.println("The sum of a and b is= " + c);  
    }  
  
    //user defined method  
    public static int add(int n1, int n2) //n1 and n2  
    //are formal para  
    meters {  
        int s;  
        s=n1+n2;  
        return s; //returning the sum  
    }  
}
```


Declaring Methods

- Return Values

- The void keyword, used in the examples above, indicates that the method should not return a value.
- If you want the method to return a value, you can use a primitive data type (such as int, char, etc.) instead of void, and use the return keyword inside the method:

```
public class Main
{
    static int myMethod(int x) {
        return 5 + x;
    }
    public static void main(String[]
args) {
        System.out.println(myMeth
od(3));
    }
}
// Outputs 8 (5 + 3)
```

- Arguments and Casting



Arguments and Casting

- Information can be passed to methods as parameter
 - Parameters are specified after the method name, inside the parentheses.
 - You can add as many parameters as you want, just separate them with a comma.

```
public class Main {  
    static void myMethod(String fname)  
    { System.out.println(fname + "  
    Refsnes");  
}  
  
    public static void main(String[]  
    args) { myMethod("Liam");  
    myMethod("Jenny");  
    myMethod("Anja");  
}  
}  
  
// Liam Refsnes  
// Jenny Refsnes  
// Anja Refsnes
```

Arguments and Casting

- Multiple Parameters

```
public class Main {  
    static void myMethod(String fname, int age) {  
        System.out.println(fname + " is " +  
            age);  
    }  
  
    public static void main(String[] args) {  
        myMethod("Liam", 5);  
        myMethod("Jenny", 8);  
        myMethod("Anja", 31);  
    }  
}  
  
// Liam is 5  
// Jenny is 8  
// Anja is 31
```

Arguments and Casting

- **pass-by-value**

- When Java passes a variable to a method via a parameter, the method itself receives a copy of the variable's value, not the variable itself.

```
public class ChangeParameters {  
    public static void main(String[] args)  
    {  
        int number = 1;  
        tryToChangeNumber(number);  
        System.out.println(number);  
    }  
  
    public static void tryToChangeNumber(int i) {  
        i = 2;  
    }  
}
```

Arguments and Casting

- **Argument Promotion**

- Coercion of arguments

- Forcing arguments to appropriate type to pass to method

- e.g., `System.out.println( Math.sqrt( 4 ) );`

- Evaluates `Math.sqrt( 4 )`

- Then evaluates `System.out.println()`

- Promotion rules

- Specify how to convert types without data loss

Arguments and Casting

- **Argument Promotion**

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)

Fig. 5.4 | Promotions allowed for primitive types.

Arguments and Casting

- **Argument Promotion**

- Suppose method **square** calculates the square of an integer
- Thus requires an **int** argument.
- To call **square** with a **double** argument named doubleValue, we would be required to write the method call as

```
square((int) doubleValue)
```

- This method call explicitly casts (converts) doubleValue's value to a temporary integer for use in method square.
- Thus, if doubleValue's value is 4.5, the method receives the value 4 and returns 16, not 20.25.

Reference

S

- Deitel, P. J., & Deitel, H. M. (2017). Java: how to program late objects. Pearson.

Thank You





الجامعة السعودية الإلكترونية
SAUDI ELECTRONIC UNIVERSITY
2011-1432

College of Computing and Informatics

Object Oriented Programming



Object Oriented Programming

Week 8

Methods (Part 2)



Contents

1. Java API Packages
2. Scope of Declaration
3. Method Overloading



Weekly Learning Outcomes



- Summarize methods and Java API packages features



Required Reading

1. Chapter 5 - (Java: How to program: Late Objects)

Recommended Reading

1. Chapter 6 - (Java The Complete Reference, Eleventh Edition)

2. Defining Methods:

<https://docs.oracle.com/javase/tutorial/java/javaOO/methods.html>

3. Java API Packages: <https://www.javatpoint.com/package>



- Java API Packages



Java API Packages

- Packages
 - Classes grouped into categories of related classes
 - Promotes software reuse
 - import statements specify classes used in Java programs
 - e.g., `import javax.swing.JApplet;`

Java API Packages

Package	Description
<code>java.awt.event</code>	The Java Abstract Window Toolkit Event Package contains classes and interfaces that enable event handling for GUI components in both the <code>java.awt</code> and <code>javax.swing</code> packages. (See Chapter 12, GUI Components: Part 1, and Chapter 22, GUI Components: Part 2.)
<code>java.awt.geom</code>	The Java 2D Shapes Package contains classes and interfaces for working with Java's advanced two-dimensional graphics capabilities. (See Chapter 13, Graphics and Java 2D.)
<code>java.io</code>	The Java Input/Output Package contains classes and interfaces that enable programs to input and output data. (See Chapter 15, Files, Streams and Object Serialization.)
<code>java.lang</code>	The Java Language Package contains classes and interfaces (discussed throughout the book) that are required by many Java programs. This package is imported by the compiler into all programs.
<code>java.net</code>	The Java Networking Package contains classes and interfaces that enable programs to communicate via computer networks like the Internet. (See online Chapter 28, Networking.)
<code>java.security</code>	The Java Security Package contains classes and interfaces for enhancing application security.
<code>java.sql</code>	The JDBC Package contains classes and interfaces for working with databases. (See Chapter 24, Accessing Databases with JDBC.)

Fig. 5.5 | Java API packages (a subset). (Part 1 of 2.)

Java API Packages

Package	Description
<code>java.util</code>	The Java Utilities Package contains utility classes and interfaces that enable storing and processing of large amounts of data. Many of these classes and interfaces have been updated to support Java SE 8's new lambda capabilities. (See Chapter 16, Generic Collections.)
<code>java.util.concurrent</code>	The Java Concurrency Package contains utility classes and interfaces for implementing programs that can perform multiple tasks in parallel. (See Chapter 23, Concurrency.)
<code>javax.swing</code>	The Java Swing GUI Components Package contains classes and interfaces for Java's Swing GUI components that provide support for portable GUIs. This package still uses some elements of the older <code>java.awt</code> package. (See Chapter 12, GUI Components: Part 1, and Chapter 22, GUI Components: Part 2.)
<code>javax.swing.event</code>	The Java Swing Event Package contains classes and interfaces that enable event handling (e.g., responding to button clicks) for GUI components in package <code>javax.swing</code> . (See Chapter 12, GUI Components: Part 1, and Chapter 22, GUI Components: Part 2.)
<code>javax.xml.ws</code>	The JAX-WS Package contains classes and interfaces for working with web services in Java. (See online Chapter 32, REST-Based Web Services.)
<code>javafx</code> packages	JavaFX is the preferred GUI technology for the future. We discuss these packages in Chapter 25, JavaFX GUI: Part 1 and in the online JavaFX GUI and multimedia chapters.
<i>Some Java SE 8 Packages Used in This Book</i>	
<code>java.time</code>	The new Java SE 8 Date/Time API Package contains classes and interfaces for working with dates and times. These features are designed to replace the older date and time capabilities of package <code>java.util</code> . (See Chapter 23, Concurrency.)
<code>java.util.function</code> and <code>java.util.stream</code>	These packages contain classes and interfaces for working with Java SE 8's functional programming capabilities. (See Chapter 17, Java SE 8 Lambdas and Streams.)

Fig. 5.5 | Java API packages (a subset). (Part 2 of 2.)

- Scope of Declaration



Scope of Declaration

- The scope of a declaration is the portion of the program that can refer to the declared entity by its name.
 - Portion of the program that can reference an entity by its name
- Basic scope rules
 - Scope of a parameter declaration
 - Scope of a local-variable declaration
 - Scope of a label in a labeled break or continue statement
 - Scope of a local-variable declaration that appears in the initialization section of a for statement's header
 - Scope of a method or field of a class

Scope of Declaration

- Example:

```
1 // Fig. 5.9: Scope.java
2 // Scope class demonstrates field and local variable scopes.
3
4 public class Scope
5 {
6     // field that is accessible to all methods of this class
7     private static int x = 1;
8
9     // method main creates and initializes local variable x
10    // and calls methods useLocalVariable and useField
11    public static void main(String[] args)
12    {
13        int x = 5; // method's local variable x shadows field x
14
15        System.out.printf("local x in main is %d\n", x);
16
17        useLocalVariable(); // useLocalVariable has local x
18        useField(); // useField uses class Scope's field x
19        useLocalVariable(); // useLocalVariable reinitializes local x
20        useField(); // class Scope's field x retains its value
21
22        System.out.printf("\nlocal x in main is %d\n", x);
23    }
24
25    // create and initialize local variable x during each call
26    public static void useLocalVariable()
27    {
28        int x = 25; // initialized each time useLocalVariable is called
29
30        System.out.printf(
31            "\nlocal x on entering method useLocalVariable is %d\n", x);
32        ++x; // modifies this method's local variable x
33        System.out.printf(
34            "local x before exiting method useLocalVariable is %d\n", x);
35    }
36
37    // modify class Scope's field x during each call
38    public static void useField()
39    {
40        System.out.printf(
41            "\nfield x on entering method useField is %d\n", x);
42
43        x *= 10; // modifies class Scope's field x
44        System.out.printf(
45            "field x before exiting method useField is %d\n", x);
46    }
47 } // end class Scope
```

Scope of Declaration

- Example:

```
local x in main is 5

local x on entering method useLocalVariable is 25
local x before exiting method useLocalVariable is 26

field x on entering method useField is 1
field x before exiting method useField is 10

local x on entering method useLocalVariable is 25
local x before exiting method useLocalVariable is 26

field x on entering method useField is 10
field x before exiting method useField is 100

local x in main is 5
```


- Method Overloading



Method Overloading



Several methods of the same name



Different parameter set for each method

Number of parameters

Parameter types

Method Overloading

```
1 // Fig. 5.10: MethodOverload.java
2 // Overloaded method declarations.
3
4 public class MethodOverload
5 {
6     // test overloaded square methods
7     public static void main(String[] args)
8     {
9         System.out.printf("Square of integer 7 is %d\n", square(7));
10        System.out.printf("Square of double 7.5 is %f\n", square(7.5));
11    }
12
13    // square method with int argument
14    public static int square(int intValue)
15    {
16        System.out.printf("%nCalled square with int argument: %d\n",
17                           intValue);
18        return intValue * intValue;
19    }
20
21    // square method with double argument
22    public static double square(double doubleValue)
23    {
24        System.out.printf("%nCalled square with double argument: %f\n",
25                           doubleValue);
26        return doubleValue * doubleValue;
27    }
28 } // end class MethodOverload
```

Output

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

Fig. 5.10 | Overloaded method declarations. (Part 1 of 2.)

Method Overloading

- Example:

```
class Adder{  
    static int add(int a, int b){  
        return a+b;  
    }  
    static double add(double a, double b){  
        return a+b;  
    }  
    public static void main(String[] args)  
    { System.out.println(add(11,11));  
      System.out.println(add(12.3,12.6));  
    }  
}
```

Method Overloading

- Example:

```
class Adder{  
    static int add(int a,int b){  
        return a+b;  
    }  
    static int add(int a,int b,int c){  
        return a+b+c;  
    }  
  
    public static void main(String[] args)  
    { System.out.println(add(11,11));  
      System.out.println(add(11,11,11));  
    }  
}
```

Method Overloading

- Example of Method Overloading with Type Promotion

```
class OverloadingCalculation1{
    static void sum(int a,long b){
        System.out.println(a+b);
    }
    static void sum(int a,int b,int c){
        System.out.println(a+b+c);
    }

    public static void main(String args[]){
        sum(20,20); //now second int literal will be promoted to long
        sum(20,20,20);
    }
}
```

Reference

S

- Deitel, P. J., & Deitel, H. M. (2017). Java: how to program late objects. Pearson.

Thank You





الجامعة السعودية الإلكترونية
SAUDI ELECTRONIC UNIVERSITY
2011-1432

College of Computing and Informatics

Object Oriented Programming



Object Oriented Programming

Week 10

Arrays



Contents

1. Primitive vs. Reference type
2. Creating, Initializing, and modifying arrays
3. Exception Handling (Try-Catch Statement)
4. Enhanced for Statement
5. Passing Arrays to Methods
6. Pass-By-Value vs. Pass-By-Reference
7. Multidimensional Arrays



Weekly Learning Outcomes



- Make use of Arrays to solve different Java problems
- Differentiate between Pass-By-Value vs. Pass-By-Reference



Required Reading

1. Chapter 6 - (Java: How to program: Late Objects)

Recommended Reading

1. Chapter 3 - (Java The Complete Reference, Eleventh Edition)

2. Java Arrays Tutorial:

<https://docs.oracle.com/javase/tutorial/java/nutsandbolts/arrays.html>



- Primitive vs. Reference type



Primitive vs. Reference type

- A primitive-type variable
 - can hold exactly one value of its declared type at a time. For example, an int variable can store one integer at a time.
 - When another value is assigned to that variable, the new value replaces the previous one.

Primitive vs. Reference type

- Programs use variables of reference types
 - to store the addresses of objects in the computer's memory.

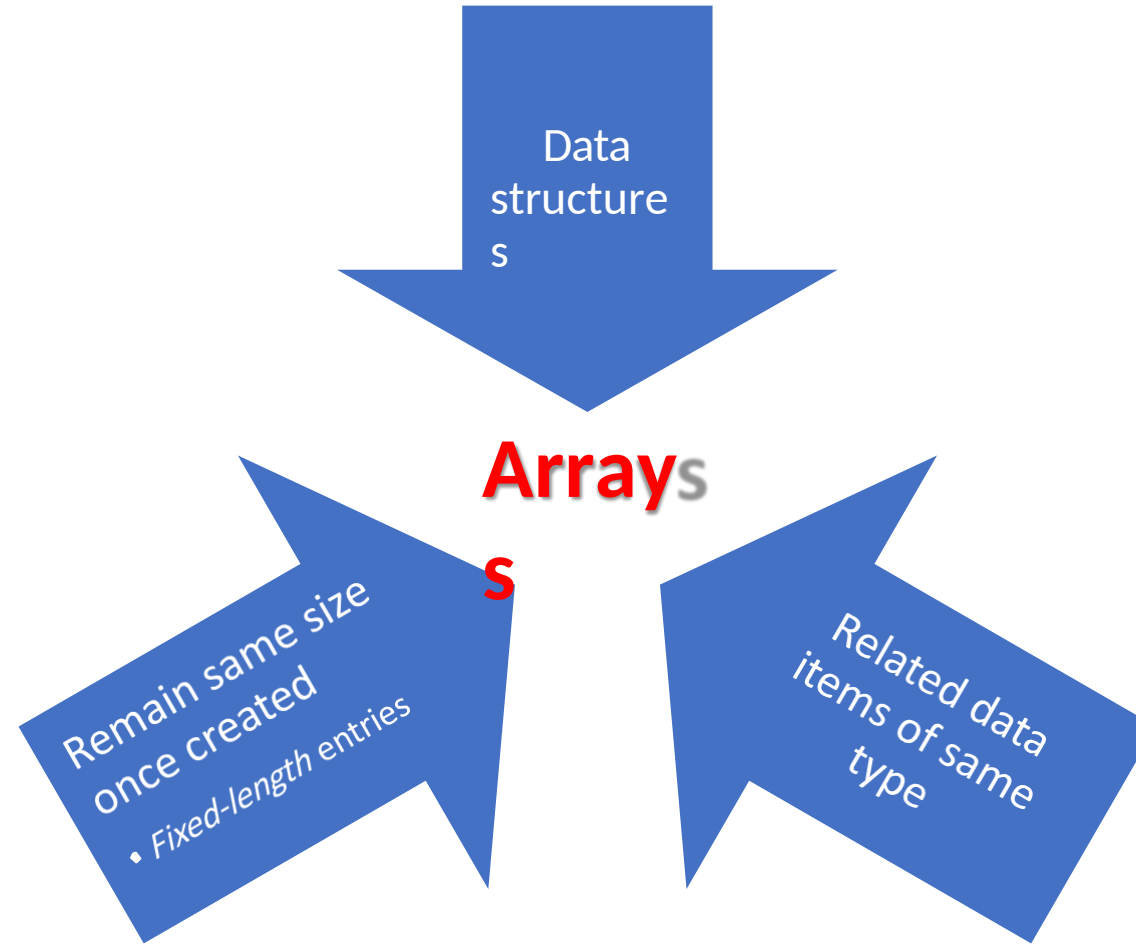
```
Scanner input = new Scanner(System.in);
```

- creates a Scanner object, then assigns to the variable input a reference to that Scanner object.

- Creating, Initializing, and modifying arrays



Creating, Initializing, and modifying arrays



Creating, Initializing, and modifying arrays

- Arrays

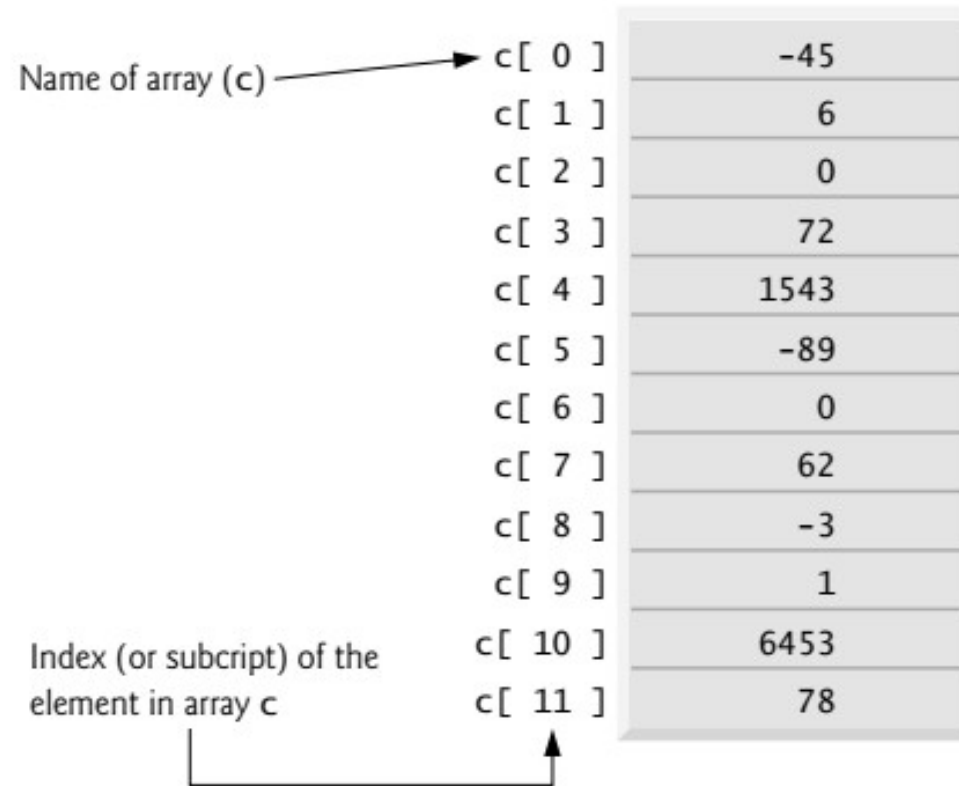


Fig. 6.1 | A 12-element array.

Creating, Initializing, and modifying arrays

- Index

- Also called subscript
- Position number in square brackets
- Must be positive integer or integer expression

```
a = 5;  
b = 6;  
c[ a + b ] += 2;
```

- Adds 2 to
c[11]

Creating, Initializing, and modifying arrays

- The name of the array is `c`
- `c.length` returns array `c`'s length
- This array's 12 elements are referred to as `c[0]`, `c[1]`, `c[2]`, ..., `c[11]`
- The value of `c[0]` is -45
- The value of `c[1]` is 6
- The value of `c[2]` is 0
- `sum = c[0] + c[1] + c[2];`

Creating, Initializing, and modifying arrays

- **Declaring and Creating Arrays**

- Array objects occupy space in memory.
- Arrays are created with keyword new.

```
int[] c = new int[12];
```

- Equivalent to:

```
int[] c;                                // declare the array
```

variable

```
c = new int[12]; // create the array; assign to array variable
```

- Create several arrays in a single declaration

```
String[] b = new String[100], x = new String[27];
```

Creating, Initializing, and modifying arrays

- Creating and Initializing an Array

Example

```
1 // Fig. 6.2: InitArray.java
2 // Initializing the elements of an array to default values of zero.
3
4 public class InitArray
5 {
6     public static void main(String[] args)
7     {
8         // declare variable array and initialize it with an array object
9         int[] array = new int[10]; // create the array object
10
11         System.out.printf("%s%8s%n", "Index", "Value"); // column headings
12
13         // output each array element's value
14         for (int counter = 0; counter < array.length; counter++)
15             System.out.printf("%5d%8d%n", counter, array[counter]);
16     }
17 } // end class InitArray
```

Output

Index	Value
0	0
1	0
2	0
3	0
4	0
5	0
6	0
7	0
8	0
9	0

Creating, Initializing, and modifying arrays

- Using an Array_INITIALIZER

```
1 // Fig. 6.3: InitArray.java
2 // Initializing the elements of an array with an array initializer.
3
4 public class InitArray
5 {
6     public static void main(String[] args)
7     {
8         // initializer list specifies the initial value for each element
9         int[] array = { 32, 27, 64, 18, 95, 14, 90, 70, 60, 37 };
10
11         System.out.printf("%s%8s%n", "Index", "Value"); // column headings
12
13         // output each array element's value
14         for (int counter = 0; counter < array.length; counter++)
15             System.out.printf("%5d%8d%n", counter, array[counter]);
16
17     } // end class InitArray
```

Output

Index	Value
0	32
1	27
2	64
3	18
4	95
5	14
6	90
7	70
8	60
9	37

Creating, Initializing, and modifying arrays

- Calculating the Values to Store in an Array

```
1 // Fig. 6.4: InitArray.java
2 // Calculating the values to be placed into the elements of an array.
3
4 public class InitArray
5 {
6     public static void main(String[] args)
7     {
8         final int ARRAY_LENGTH = 10; // declare constant
9         int[] array = new int[ARRAY_LENGTH]; // create array
10
11         // calculate value for each array element
12         for (int counter = 0; counter < array.length; counter++)
13             array[counter] = 2 + 2 * counter;
14
15         System.out.printf("%s%8s\n", "Index", "Value"); // column headings
16
17         // output each array element's value
18         for (int counter = 0; counter < array.length; counter++)
19             System.out.printf("%5d%8d\n", counter, array[counter]);
20     }
21 } // end class InitArray
```

Output

Index	Value
0	2
1	4
2	6
3	8
4	10
5	12
6	14
7	16
8	18
9	20

Creating, Initializing, and modifying arrays

- Summing the Elements of an Array

```
1 // Fig. 6.5: SumArray.java
2 // Computing the sum of the elements of an array.
3
4 public class SumArray
5 {
6     public static void main(String[] args)
7     {
8         int[] array = { 87, 68, 94, 100, 83, 78, 85, 91, 76, 87 };
9         int total = 0;
10
11         // add each element's value to total
12         for (int counter = 0; counter < array.length; counter++)
13             total += array[counter];
14
15         System.out.printf("Total of array elements: %d\n", total);
16     }
17 } // end class SumArray
```

Output

Total of array elements: 849

Creating, Initializing, and modifying arrays

• Using Bar Charts to Display Array Data Graphically

```
1 // Fig. 6.6: BarChart.java
2 // Bar chart printing program.
3
4 public class BarChart
5 {
6     public static void main(String[] args)
7     {
8         int[] array = {0, 0, 0, 0, 0, 0, 1, 2, 4, 2, 1};
9
10        System.out.println("Grade distribution:");
11
12        // for each array element, output a bar of the chart
13        for (int counter = 0; counter < array.length; counter++)
14        {
15            // output bar label ("00-09: ", ..., "90-99: ", "100: ")
16            if (counter == 10)
17                System.out.printf("%5d: ", 100);
18            else
19                System.out.printf("%02d-%02d: ",
20                                counter * 10, counter * 10 + 9);
21
22            // print bar of asterisks
23            for (int stars = 0; stars < array[counter]; stars++)
24                System.out.print("*");
25
26            System.out.println();
27        }
28    }
29 } // end class BarChart
```

Output

```
Grade distribution:
00-09:
10-19:
20-29:
30-39:
40-49:
50-59:
60-69: *
70-79: **
80-89: ****
90-99: **
100: *
```

Creating, Initializing, and modifying arrays

- Using the Elements of an Array as Counters

```
1 // Fig. 6.7: RollDie.java
2 // Die-rolling program using arrays instead of switch.
3 import java.security.SecureRandom;
4
5 public class RollDie
6 {
7     public static void main(String[] args)
8     {
9         SecureRandom randomNumbers = new SecureRandom();
10        int[] frequency = new int[7]; // array of frequency counters
11
12        // roll die 6,000,000 times; use die value as frequency index
13        for (int roll = 1; roll <= 6000000; roll++)
14            ++frequency[1 + randomNumbers.nextInt(6)];
15
16        System.out.printf("%s%10s\n", "Face", "Frequency");
17
18        // output each array element's value
19        for (int face = 1; face < frequency.length; face++)
20            System.out.printf("%4d%10d\n", face, frequency[face]);
21    }
22 } // end class RollDie
```

Output

Face	Frequency
1	999690
2	999512
3	1000575
4	999815
5	999781
6	1000627

- Exception Handling (Try-Catch Statement)



Exception Handling (Try-Catch Statement)

An exception indicates

- A problem that occurs while a program executes.
- To handle an exception, place any code that might throw an exception in a **try-Catch** statement.

The try Statement

- contains the code that might throw an exception.

The Catch Statement

- contains the code that handles the exception if one occurs.

Exception Handling (Try-Catch Statement)

- Try-Catch Syntax

```
try {  
    // Block of code to try  
}  
catch(Exception e) {  
    // Block of code to handle errors  
}
```


Exception Handling (Try-Catch Statement)

- Exception Example

```
public class Main
{
    public static void main(String[] args) {
        int[] myNumbers = {1, 2, 3};
        System.out.println(myNumbers[10]); //
                                         error!
    }
}
```

Output

Exception in thread "main" java.lang.ArrayIndexOutOfBoundsException: 10
at Main.main(Main.java:4)

Exception Handling (Try-Catch Statement)

- finally statement lets you execute code, after try...catch, regardless of the result:

```
public class Main {  
    public static void main(String[] args) {  
        try {  
            int[] myNumbers = {1, 2, 3};  
            System.out.println(myNumbers[10]  
            );  
        }  
        catch (Exception e) {  
            System.out.println("Something  
            went wrong.");  
        }  
        finally {  
            System.out.println("The try  
            catch is finished.");  
        }  
    }  
}
```

Something went wrong.

Exception Handling (Try-Catch Statement)

- Exception Example

```
public class Main {  
    public static void main(String[ ]  
        args) { try {  
        int[] myNumbers = {1, 2, 3};  
        System.out.println(myNumbers[10  
            ]));  
    }  
    catch (Exception e) {  
        System.out.println("Something went  
            wrong.");  
    }  
}
```

Output

Something went wrong.

- Enhanced for Statement



Enhanced for Statement

- Iterates through the elements of an array without using a counter.
 - thus avoiding the possibility of “stepping outside” the array

```
for (parameter : arrayName)  
    statement
```

- parameter has a type and an identifier
- arrayName is the array through which to iterate.

Enhanced for Statement

```
1 // Fig. 6.9: EnhancedForTest.java
2 // Using the enhanced for statement to total integers in an array.
3
4 public class EnhancedForTest
5 {
6     public static void main(String[] args)
7     {
8         int[] array = { 87, 68, 94, 100, 83, 78, 85, 91, 76, 87 };
9         int total = 0;
10
11         // add each element's value to total
12         for (int number : array)
13             total += number;
14
15         System.out.printf("Total of array elements: %d\n", total);
16     }
17 } // end class EnhancedForTest
```

Output

Total of array elements: 849

- Passing Arrays to Methods



Passing Arrays to Methods

- To pass an array argument to a method, specify the name of the array without any brackets.
 - For example to pass the array `hourlyTemperatures`

```
double[] hourlyTemperatures = new double[24];
```

- The method call will be:

```
modifyArray(hourlyTemperatures);
```

- The method header will be:

```
void modifyArray(double[] b)
```


Passing Arrays to Methods

- Example:

```
1 // Fig. 6.10: PassArray.java
2 // Passing arrays and individual array elements to methods.
3
4 public class PassArray
5 {
6     // main creates array and calls modifyArray and modifyElement
7     public static void main(String[] args)
8     {
9         int[] array = { 1, 2, 3, 4, 5 };
10
11         System.out.printf(
12             "Effects of passing reference to entire array:%n" +
13             "The values of the original array are:%n");
14
15         // output original array elements
16         for (int value : array)
17             System.out.printf("  %d", value);
18
19         modifyArray(array); // pass array reference
20         System.out.printf("%n%nThe values of the modified array are:%n");
21
```

```
22         // output modified array elements
23         for (int value : array)
24             System.out.printf("  %d", value);
25
26         System.out.printf(
27             "%n%nEffects of passing array element value:%n" +
28             "array[3] before modifyElement: %d%n", array[3]);
29
30         modifyElement(array[3]); // attempt to modify array[3]
31         System.out.printf(
32             "array[3] after modifyElement: %d%n", array[3]);
33     }
34
35     // multiply each element of an array by 2
36     public static void modifyArray(int[] array2)
37     {
38         for (int counter = 0; counter < array2.length; counter++)
39             array2[counter] *= 2;
40     }
41
42     // multiply argument by 2
43     public static void modifyElement(int element)
44     {
45         element *= 2;
46         System.out.printf(
47             "Value of element in modifyElement: %d%n", element);
48     }
49 } // end class PassArray
```

Passing Arrays to Methods

- Example:

Output

Effects of passing reference to entire array:

The values of the original array are:

1 2 3 4 5

The values of the modified array are:

2 4 6 8 10

Effects of passing array element value:

array[3] before modifyElement: 8

Value of element in modifyElement: 16

array[3] after modifyElement: 8

- Pass-By-Value vs. Pass-By-Reference



Pass-By-Value vs. Pass-By-Reference

- Two ways to pass arguments in method calls in many programming languages are
 - pass-by-value (call-by-value).
 - pass-by-reference (call-by-reference).

Pass-By-Value vs. Pass-By-Reference

- pass-by-value
 - 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 called method's copy do not affect the original variable's value in the caller

Pass-By-Value vs. Pass-By-Reference

- pass-by-reference
 - When an argument is passed by reference:
 - The called method can access the argument's value in the caller directly and modify that data, if necessary.
 - It improves performance by eliminating the need to copy possibly large amounts of data.

Pass-By-Value vs. Pass-By-Reference

Java does not allow you to choose pass-by-value or pass-by-reference—all arguments are passed by value.

- Dealing with reference object type will be discussed in detail in later chapters.

- Multidimensional Arrays



Multidimensional Arrays

Multidimensional arrays

- Tables with rows and columns
 - Two-dimensional array
 - Declaring two-dimensional array `b[2][2]`
 - `int b[][] = { { 1, 2 }, { 3, 4 } };`
 - 1 and 2 initialize `b[0][0]` and `b[0][1]`
 - 3 and 4 initialize `b[1][0]` and `b[1][1]`
 - `int b[][] = { { 1, 2 }, { 3, 4, 5 } };`
 - row 0 contains elements 1 and 2
 - row 1 contains elements 3, 4 and 5

Multidimensional Arrays

	Column 0	Column 1	Column 2	Column 3
Row 0	a[0][0]	a[0][1]	a[0][2]	a[0][3]
Row 1	a[1][0]	a[1][1]	a[1][2]	a[1][3]
Row 2	a[2][0]	a[2][1]	a[2][2]	a[2][3]

The diagram illustrates a 2D array structure. Below the table, three arrows point to the cell 'a[2][1]'. The longest arrow, labeled 'Array name', points to the 'a' part of the notation. The middle arrow, labeled 'Row index', points to the '[2]' part. The shortest arrow, labeled 'Column index', points to the '[1]' part.

Fig. 6.11 | Two-dimensional array with three rows and four columns.

Multidimensional Arrays

```
1 // Fig. 6.12: InitArray.java
2 // Initializing two-dimensional arrays.
3
4 public class InitArray
5 {
6     // create and output two-dimensional arrays
7     public static void main(String[] args)
8     {
9         int[][] array1 = {{1, 2, 3}, {4, 5, 6}};
10        int[][] array2 = {{1, 2}, {3}, {4, 5, 6}};
11
12        System.out.println("Values in array1 by row are");
13        outputArray(array1); // displays array1 by row
14
15        System.out.printf("\nValues in array2 by row are\n");
16        outputArray(array2); // displays array2 by row
17    }
18
19    // output rows and columns of a two-dimensional array
20    public static void outputArray(int[][] array)
21    {
22        // loop through array's rows
23        for (int row = 0; row < array.length; row++)
24        {
25            // loop through columns of current row
26            for (int column = 0; column < array[row].length; column++)
27                System.out.printf("%d ", array[row][column]);
28
29            System.out.println();
30        }
31    }
32 } // end class InitArray
```

Output

Values in array1 by row are

```
1 2 3
4 5 6
```

Values in array2 by row are

```
1 2
3
4 5 6
```

Reference

S

- Deitel, P. J., & Deitel, H. M. (2017). Java: how to program late objects. Pearson.

Thank You





الجامعة السعودية الإلكترونية
SAUDI ELECTRONIC UNIVERSITY
2011-1432

College of Computing and Informatics

Object Oriented Programming



Object Oriented Programming
Week 11
Classes and Objects I



Contents

1. Declaring classes, instance variables, methods
2. Default and Explicit Initialization for Instance Variables
3. Constructors, private, public, static.
4. Instantiating and using objects
5. Implementing a test class



Weekly Learning Outcomes



- Construct Java Classes and Instantiating objects
- Plan and design a multi class application



Required Reading

1. Chapter 7 - (Java: How to program: Late Objects)

Recommended Reading

1. Chapter 6 - (Java The Complete Reference, Eleventh Edition)

2. Java Classes and Objects:

https://www.w3schools.com/java/java_classes.asp

p



- Declaring classes, instance variables, methods



Declaring classes, instance variables, methods

Object Oriented Programming (OOP)

Encapsulates data (attributes) and methods (behaviors)

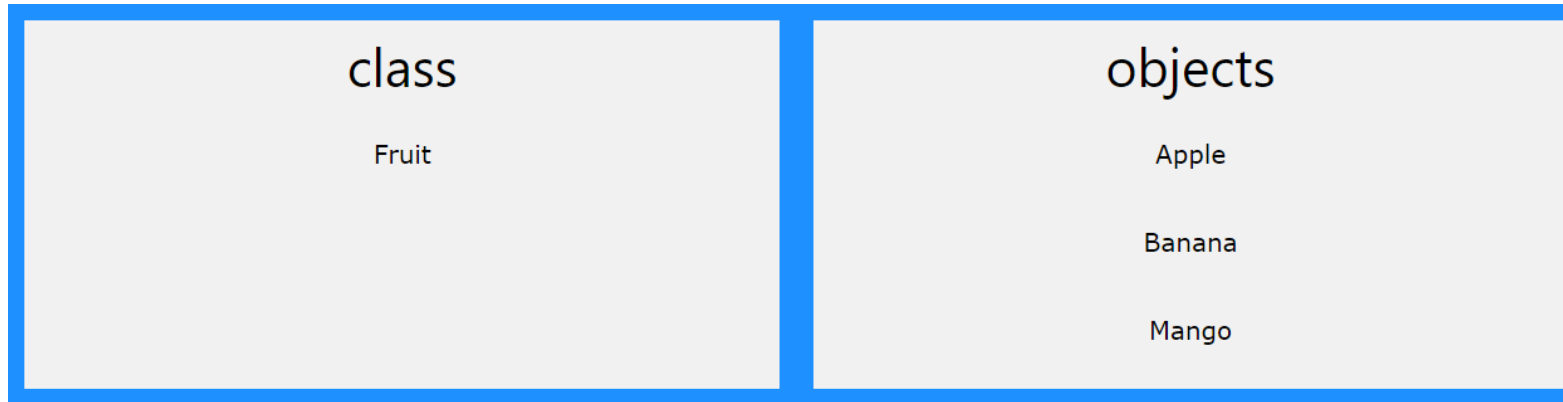
- Objects

Allows objects to communicate

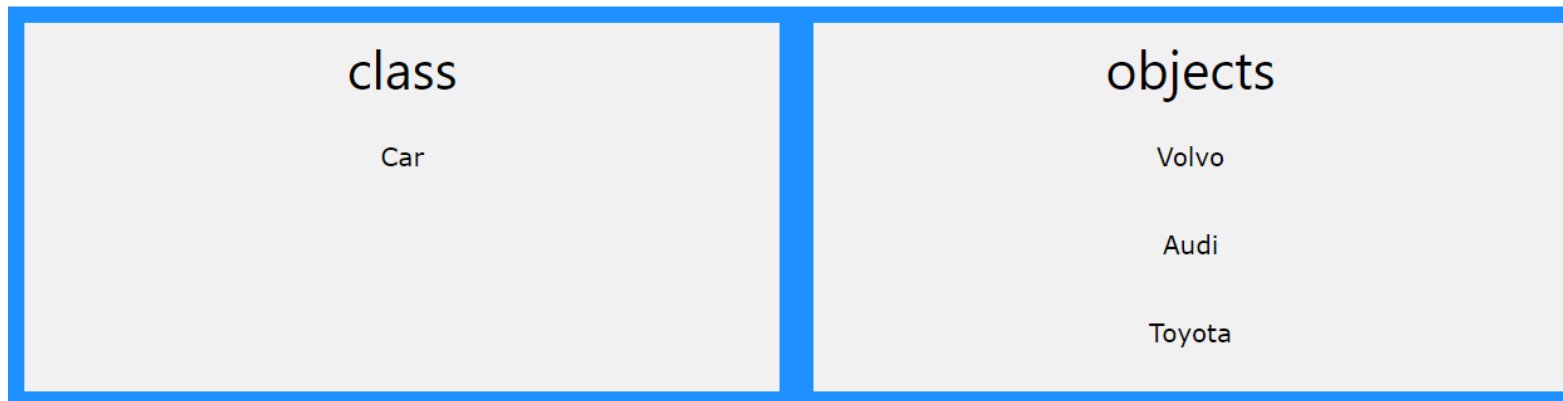
- Well-defined interfaces

Declaring classes, instance variables, methods

- What are Classes and Objects?



Another example:



Reference: https://www.w3schools.com/java/java_oop.asp

Declaring classes, instance variables, methods

- A class is a group of objects which have common properties. It is a template or blueprint from which objects are created. It is a logical entity. **It can't be physical.**
- A class in Java can contain:
 - Fields
 - Methods
 - Constructors
 - Blocks
 - Nested class and interface

Declaring classes, instance variables, methods

- **An object is an instance of a class.** A **class is a template or blueprint** from which objects are created. So, an object is the instance(result) of a class.
- **Object Definitions:**
 - An object is a real-world entity.
 - An object is a runtime entity.
 - The object is an entity which has state and behavior.
 - The object is an instance of a class.

Declaring classes, instance variables, methods

- Instance variable in Java
 - A variable which is created inside the class but outside the method is known as an instance variable.
 - Instance variable doesn't get memory at compile time.
 - It gets memory at runtime when an object or instance is created.
 - That is why it is known as an instance variable.

Declaring classes, instance variables, methods

Method in Java

In Java, a method is like a function which is used to expose the behavior of an object.

Methods in classes represent a behavior of an object from the class

Declaring classes, instance variables, methods

- Example:

```
1  // Fig. 7.1: Account.java
2  // Account class that contains a name instance variable
3  // and methods to set and get its value.
4
5  public class Account
6  {
7      private String name; // instance variable
8
9      // method to set the name in the object
10     public void setName(String name)
11     {
12         this.name = name; // store the name
13     }
14
15     // method to retrieve the name from the object
16     public String getName()
17     {
18         return name; // return value of name to caller
19     }
20 } // end class Account
```

Fig. 7.1 | Account class that contains a **name** instance variable and methods to *set* and *get* its value.

Declaring classes, instance variables, methods

- Instance Variable *name*

```
private String name; // instance variable
```

- Instance variables are declared inside a class declaration but outside the bodies of the class's methods
- Access Modifiers *public* and *private*
 - Variables or methods declared with access modifier public are accessible from outside the class in which they're declared.
 - Variables or methods declared with access modifier private are accessible only to methods of the class in which they're declared.

Declaring classes, instance variables, methods

- Instance Method ***setName*** of Class

Account

```
public void setName(String name)
{
    this.name = name; // store the name
}
```

- indicates that setName receives parameter name of type String—which represents the name that will be passed to the method as an argument.
- If a method contains a local variable with the same name as an instance variable, the method's body can use the keyword **this** to refer to the shadowed instance variable explicitly

Declaring classes, instance variables, methods

• AccountTest Class That Creates and Uses an Object of Class Account

```
1 // Fig. 7.2: AccountTest.java
2 // Creating and manipulating an Account object.
3 import java.util.Scanner;
4
5 public class AccountTest
6 {
7     public static void main(String[] args)
8     {
9         // create a Scanner object to obtain input from the command window
10        Scanner input = new Scanner(System.in);
11
12        // create an Account object and assign it to myAccount
13        Account myAccount = new Account();
14
15        // display initial value of name (null)
16        System.out.printf("Initial name is: %s\n\n", myAccount.getName());
17
18        // prompt for and read name
19        System.out.println("Please enter the name:");
20        String theName = input.nextLine(); // read a line of text
21        myAccount.setName(theName); // put theName in myAccount
22        System.out.println(); // outputs a blank line
23
24        // display the name stored in object myAccount
25        System.out.printf("Name in object myAccount is:\n%s\n",
26            myAccount.getName());
27    }
28 } // end class AccountTest
```

Output

Initial name is: null

Please enter the name:
Jane Green

Name in object myAccount is:
Jane Green

Declaring classes, instance variables, methods

- Instantiating an Object—Keyword **new**

```
// create an Account object and assign it to myAccount  
Account myAccount = new Account();
```

- Keyword **new** creates a new object of the specified class—in this case, Account.
- The parentheses to the right of Account are required.

Declaring classes, instance variables, methods

static Methods



Recall that static methods relate to a class as a whole



Whereas instance methods are associated with a specific instance (object) of the class and may manipulate the instance variables of that object.

- Default and Explicit Initialization for Instance Variables



Default and Explicit Initialization for Instance Variables

- Recall that local variables are not initialized by default.
 - Primitive-type instance variables are initialized by default
 - Instance variables of types byte, char, short, int, long, float and double are initialized to 0, and variables of type boolean are initialized to false.
- You can initialize an instance variable as:

```
private int numberOfStudents = 10;
```

- Constructors, setters, getters, private, public.



Constructors, private, public, static.

- Each class you declare can optionally provide a constructor with parameters that can be used to initialize an object of a class when the object is created.
 - It has the same name as its class and is syntactically similar to a method.
 - However, constructors have no explicit return type.

Constructors, private, public, static.

- Example:

```
Public class MyClass {  
    Int num; MyClass() {  
        num = 100;  
    }  
}
```

```
public class ConsDemo {  
    public static void main(String args[]) {  
        MyClass t1 = new MyClass();  
        MyClass t2 = new MyClass();  
        System.out.println(t1.num + " " +  
            t2.num);  
    }  
}
```

Output
100 100

Constructors, private, public, static.

- Parameterized Constructors

```
// A simple constructor.  
class MyClass {  
    int x;  
    MyClass(int i ) {  
        x = i;  
    }  
}
```

```
public class ConsDemo {  
    public static void main(String args[])  
    {  
        MyClass t1 = new MyClass( 10 );  
        MyClass t2 = new MyClass( 20 );  
        System.out.println(t1.x + " " + t2.x);  
    }  
}
```

Output
10

Constructors, private, public, static.

- If you declare any variable as static, it is known as a static variable.
 - The static variable can be used to refer to the common property of all objects
 - The static variable gets memory only once in the class area at the time of class loading.

```
class
Student{ int
rollno;
String name;
String
college="ITS
";
}
```

Constructors, private, public, static.

```
class
Student{ int
rollno;
String name;
String
college="ITS
";
}
```

Output

```
111 Karan ITS
222 Aryan ITS
```

```
class Student{
    int rollno;//instance variable
    String name;
    static String college
="ITS";//static variable
    Student(int r, String n){
        rollno = r;
        name = n;
    }
    void display ()
    {System.out.println(rollno+"
"+name+" "+college);}
}
```

public class

```
TestStaticVariable1{
    public static void
    main(String args[]){
        Student s1 = new Student(111,"Karan");
        Student s2 = new Student(222,"Aryan");
        s1.display();
```

Reference: <https://www.javatpoint.com>

Constructors, private, public, static.

- Program of the counter without static variable

```
class Counter{
    int count=0;//will get memory each time when the instance is created
    Counter(){
        count++;//incrementing value
        System.out.println(count);
    }

    public static void main(String args[]){
        //Creating objects
        Counter c1=new Counter();
        Counter c2=new
        Counter();      Counter
        c3=new Counter();
    }
}
```

Output

1
1
1

Constructors, private, public, static.

- Program of the counter with static variable

```
class Counter{
    static int count=0;
    Counter(){
        count+
        +; //
        increme
        nting
    public static void main(String args[]){
        //Starting objects
        Counter c1=new Counter();
        Counter c2=new Counter();
        Counter c3=new Counter();
    }
}
```

Output

1
2
3

Constructors, private, public, static.

- Static Method Example:

```
class Calculate{  
    static int cube(int x){  
        return x*x*x;  
    }  
  
    public static void main(String args[])  
    { int  
      result=Calculate.cube(5);  
      System.out.println(result);  
    }  
}
```

Output

125

Constructors, private, public, static.

- Public & Private Modifiers

	default	private	protected	public
Same Class	Yes	Yes	Yes	Yes
Same package subclass	Yes	No	Yes	Yes
Same package non-subclass	Yes	No	Yes	Yes
Different package subclass	No	No	Yes	Yes
Different package non-subclass	No	No	No	Yes

Constructors, private, public, static.

- Public Modifier

```
package abcpackage;  
public class Addition {  
    Public int addTwoNumbers(int a, int b){  
        return a+b;  
    }  
}
```

```
package xyzpackage;  
import abcpackage.*;  
public class Test {  
    public static void main(String args[]){  
        Addition obj = new Addition();  
        obj.addTwoNumbers(10, 21);  
    }  
}
```

Output

31

Constructors, private, public, static.

- Private Modifier

```
class ABC{
    private double num = 100;
    private int square(int a){
        return a*a;
    }
}

public class Example{
    public static void main(String args[])
    {
        ABC obj = new ABC();
        System.out.println(obj.num);
        System.out.println(obj.square(10));
    }
}
```

Output

Compile - time error

- Instantiating and using objects



Instantiating and using objects

- Each of the following statements creates an object and assigns it to a variable:
 - `Point originOne = new Point(23, 94);`
 - `Rectangle rectOne = new Rectangle(originOne, 100, 200);`
 - `Rectangle rectTwo = new Rectangle(50, 100);`
- Each of these statements has three parts:
 - **Declaration**
 - **Declaration**: The code set in bold are all variable declarations that associate a variable name with an object type.
 - **Instantiation**: The new keyword is a Java operator that creates the object.
 - **Initialization**: The new operator is followed by a call to a constructor, which initializes the new object.

- Implementing a test class



Implementing a test class

- Example 1:

```
class Lamp {  
    // stores the value for light  
    // true if light is on  
    // false if light is off  
    boolean isOn;  
    // method to turn on the  
    light void turnOn() {  
        isOn = true;  
        System.out.pr  
        intln("Light  
        on? " + isOn);  
    }  
    // method to turnoff the  
    light void turnOff() {  
        isOn = false;  
        System.out.p  
        rintln("Light  
        on? " + isOn);  
    }  
}  
class Main {  
    public static void  
    main(String[] args) {  
        // create objects led and  
        halogen Lamp led = new  
        Lamp();  
        Lamp halogen = new Lamp();  
        // turn on the light by calling  
        method turnOn()  
        led.turnOn();  
        halogen.turnOff();  
    }  
}
```

Output

Light on? true
Light on? False

Implementing a test class

- Example 2:

```
class Account{
    int acc_no;
    String name;
    float amount;
    void insert(int a,String n,float amt){
        acc_no=a;
        name=n; amount=amt;
    }
    void deposit(float amt){
        amount=amount+amt; System.out.println(amt+"
        deposited");
    }
    void withdraw(float amt){
        if(amount<amt){
            System.out.println("Insufficient
            Balance");
        }else{
            amount=amount-amt; System.out.println(amt+"
            withdrawn");
        }
    }
    //method to check the balance of the account
    void checkBalance(){System.out.println("Balance is: "+amount);}
    //method to display the values of an object
    void display(){System.out.println(acc_no+" "+name+"
    "+amount);}
}
```

Reference: <https://www.javatpoint.com/object-and-class-in-java>

Implementing a test class

- Example 2:

```
class TestAccount{  
    public static void main(String[] args)  
    {  
        Account a1=new Account();  
        a1.insert(832345,"Ankit",1000);  
        a1.display();  
        a1.checkBalance();  
        a1.deposit(40000);  
        a1.checkBalance();  
        a1.withdraw(15000)  
        ; a1.checkBalance();  
    }  
}
```

Output

```
832345 Ankit 1000.0  
Balance is: 1000.0  
40000.0 deposited  
Balance is: 41000.0  
15000.0 withdrawn  
Balance is: 26000.0
```

Reference

S

- Deitel, P. J., & Deitel, H. M. (2017). Java: how to program late objects. Pearson.

Thank You





الجامعة السعودية الإلكترونية
SAUDI ELECTRONIC UNIVERSITY
2011-1432

College of Computing and Informatics

Object Oriented Programming



Object Oriented Programming
Week 12
Classes and Objects II



Contents

1. this Reference
2. Overloaded Constructors
3. Default and No-Argument Constructors
4. Composition
5. enum Types
6. Garbage Collection
7. static Class Members & static import
8. final Instance Variables
9. Creating packages



Weekly Learning Outcomes



- Demonstrate encapsulation and data hiding concepts
- Illustrate final and static modifiers



Required Reading

1. Chapter 8 - (Java: How to program: Late Objects)

Recommended Reading

1. Chapter 7 - (Java The Complete Reference, Eleventh Edition)
2. Final static variables: <https://www.geeksforgeeks.org/final-static-variable-java/>
3. Garbage Collection in Java: <https://beginnersbook.com/2013/04/java-garbage-collection/>



- this Reference



this Reference

- The problem if we don't use this keyword by the example given below:

```
class Student{
    int rollno;
    String name;
    float fee;
    Student(int
rollno,String
name,float
fee){
        rollno=rollno;

        name=name;
        fee=fee;
}

class TestThis1{
    public static void main(String args[]){
        void display() {Student s1=new Student(111,"ankit",5000f);
        {System.out.println("Student 2"+new Student(112,"sumit",6000f);
        no+" "+name+" "+fee);} s1.display();
        "+fee);} s2.display();
    }
}
```

Output

0 null 0.0

0 null 0.0

this Reference

- The problem if we don't use this keyword by the example given below:

```
class Student{
    int rollno;
    String name;
    float fee;
    Student(int rollno,String name,float fee){
        this.rollno=rollno;
        this.name=name;
        this.fee=fee;
    }
    void display(){System.out.println(rollno+"
        "+name+" "+fee);}
}
class TestThis1{
    public static void main(String args[]){
        Student s1=new Student(111,"ankit",5000f);
        Student s2=new Student(112,"sumit",6000f);
        s1.display();
        s2.display();
    }
}
```

Output

```
111 ankit 5000.0
112 sumit 6000.0
```

this Reference

- this() : to invoke current class constructor

```
class A{
    A(){
        System.out.println("hello a");
    }
    A(int x){
        this();
        System.out.println(x);
    }
}
class TestThis5{
    public static void main(String args[]){
        A a=new A(10);
    }
}
```

Output

hello a
10

this Reference

- Example:

```
1 // Fig. 8.4: ThisTest.java
2 // this used implicitly and explicitly to refer to members of an object.
3
4 public class ThisTest
5 {
6     public static void main(String[] args)
7     {
8         SimpleTime time = new SimpleTime(15, 30, 19);
9         System.out.println(time.buildString());
10    }
11 } // end class ThisTest
12
13 // class SimpleTime demonstrates the "this" reference
14 class SimpleTime
15 {
16     private int hour; // 0-23
17     private int minute; // 0-59
18     private int second; // 0-59
19
20     // if the constructor uses parameter names identical to
21     // instance variable names the "this" reference is
22     // required to distinguish between the names
23     public SimpleTime(int hour, int minute, int second)
24     {
25         this.hour = hour; // set "this" object's hour
26         this.minute = minute; // set "this" object's minute
27         this.second = second; // set "this" object's second
28     }
29 }
```

this Reference

```
30 // use explicit and implicit "this" to call toUniversalString
31 public String buildString()
32 {
33     return String.format("%24s: %s%n%24s: %s",
34         "this.toUniversalString()", this.toUniversalString(),
35         "toUniversalString()", toUniversalString());
36 }
37
38 // convert to String in universal-time format (HH:MM:SS)
39 public String toUniversalString()
40 {
41     // "this" is not required here to access instance variables,
42     // because method does not have local variables with same
43     // names as instance variables
44     return String.format("%02d:%02d:%02d",
45         this.hour, this.minute, this.second);
46 }
47 } // end class SimpleTime
```

Output

```
this.toUniversalString(): 15:30:19
toUniversalString(): 15:30:19
```

- Overloaded Constructors



Overloaded Constructors

- Java Constructor overloading is a technique in which a class can have any number of constructors that differ in parameter list.
- The compiler differentiates these constructors by taking into account the number of parameters in the list and their type.
- Example of valid constructors for class Account:
 - `Account(int a);`
 - `Account (int a,int b);`
 - `Account (String a,int b);`

Overloaded Constructors

```
class Demo{  
    int  
    value1;  
    int  
    value2;  
    Demo(){  
        v  
        a  
        l  
        u  
        e  
        1  
  
        =  
  
        1  
        0  
        ;  
        v  
        a  
        l  
        u  
        e  
        2  
  
        =  
  
        2  
        0  
        ;  
        S  
        v
```

Output

Inside 1st Constructor
Inside 2nd Constructor
Inside 3rd Constructor
Value1 === 10

Value2 === 20
Value1 === 30
Value2 === 0
Value1 === 30
Value2 === 40

Overloaded Constructors

- **Time Class Case Study**

- Check the full case study in Chapter 8, pages from 316 to 322

- Default and No-Argument Constructors



Default and No-Argument Constructors

- Every class must have at least one constructor.
 - If you do not provide any in a class's declaration, the compiler creates a default constructor that takes no arguments when it's invoked.
- The default constructor initializes the instance variables to the initial values specified in their declarations or to their default values

If we do not create any constructor, the Java compiler automatically create a no-arg constructor during the execution of the program. This constructor is called default constructor.

- Composition



Composition

- A class can have references to objects of other classes as members.
 - This is called composition and is sometimes referred to as a has-a relationship.
- Example:
 - An AlarmClock object needs to know the current time and the time when it's supposed to sound its alarm
 - it's reasonable to include two references to Time objects in an AlarmClock object.
 - A car has-a steering wheel, a break pedal and an accelerator pedal.

Composition

- Example (Class Date):

```
1 // Fig. 8.7: Date.java
2 // Date class declaration.
3
4 public class Date
5 {
6     private int month; // 1-12
7     private int day; // 1-31 based on month
8     private int year; // any year
9
10    private static final int[] daysPerMonth =
11        { 0, 31, 28, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31 };
12
13    // constructor: confirm proper value for month and day given the year
14    public Date(int month, int day, int year)
15    {
16        // check if month in range
17        if (month <= 0 || month > 12)
18            throw new IllegalArgumentException(
19                "month (" + month + ") must be 1-12");
20
21        // check if day in range for month
22        if (day <= 0 ||
23            (day > daysPerMonth[month] && !(month == 2 && day == 29)))
24            throw new IllegalArgumentException("day (" + day +
25                ") out-of-range for the specified month and year");
26
27        // check for leap year if month is 2 and day is 29
28        if (month == 2 && day == 29 && !(year % 400 == 0 ||
29            (year % 4 == 0 && year % 100 != 0)))
30            throw new IllegalArgumentException("day (" + day +
31                ") out-of-range for the specified month and year");
32
33        this.month = month;
34        this.day = day;
35        this.year = year;
36
37        System.out.printf(
38            "Date object constructor for date %s\n", this);
39    }
40
41    // return a String of the form month/day/year
42    public String toString()
43    {
44        return String.format("%d/%d/%d", month, day, year);
45    }
46 } // end class Date
```

Composition

- Class

Employee

```
1 // Fig. 8.8: Employee.java
2 // Employee class with references to other objects.
3
4 public class Employee
5 {
6     private String firstName;
7     private String lastName;
8     private Date birthDate;
9     private Date hireDate;
10
11     // constructor to initialize name, birth date and hire date
12     public Employee(String firstName, String lastName, Date birthDate,
13                     Date hireDate)
14     {
15         this.firstName = firstName;
16         this.lastName = lastName;
17         this.birthDate = birthDate;
18         this.hireDate = hireDate;
19     }
20
21     // convert Employee to String format
22     public String toString()
23     {
24         return String.format("%s, %s Hired: %s Birthday: %s",
25                               lastName, firstName, hireDate, birthDate);
26     }
27 } // end class Employee
```

Composition

- Class

EmployeeTest

```
1 // Fig. 8.9: EmployeeTest.java
2 // Composition demonstration.
3
4 public class EmployeeTest
5 {
6     public static void main(String[] args)
7     {
8         Date birth = new Date(7, 24, 1949);
9         Date hire = new Date(3, 12, 1988);
10        Employee employee = new Employee("Bob", "Blue", birth, hire);
11
12        System.out.println(employee);
13    }
14 } // end class EmployeeTest
```

Output

```
Date object constructor for date 7/24/1949
Date object constructor for date 3/12/1988
Blue, Bob  Hired: 3/12/1988  Birthday: 7/24/1949
```

- **enum Types**



enum Types

- It is a special class
 - which defines a set of constants represented as unique identifiers

```
enum Level {  
    LOW,  
    MEDIUM  
    ,  
    HIGH  
}
```

```
Level myVar =  
Level.MEDIUM;
```

enum Types

- Example:

```
class EnumExample1{  
    //defining the enum inside the class  
    public enum Season { WINTER, SPRING, SUMMER, FALL }  
    //main method  
    public static void main(String[] args) {  
        //traversing the enum  
        for (Season s : Season.values())  
            System.out.println(s);  
    }  
}
```

Output

WINTER
SPRING
SUMMER
FALL

enum Types

- Example:

```
class EnumExample1{
    //defining enum within class
    public enum Season { WINTER, SPRING, SUMMER, FALL }
    //creating the main method
    public static void main(String[] args) {
        //printing all enum
        for (Season s : Season.values()){
            System.out.println(s);
        }
        System.out.println("Value of WINTER is: "+Season.valueOf("WINTER"));
        System.out.println("Index of WINTER is: "+Season.valueOf("WINTER").ordinal());
        System.out.println("Index of SUMMER is: "+Season.valueOf("SUMMER").ordinal());
    }
}
```

Output

WINTER
SPRING
SUMMER
FALL

Value of WINTER is: WINTER
Index of WINTER is: 0
Index of SUMMER is: 2

- Garbage Collection



Garbage Collection



Object uses system resources, such as memory.



We need a disciplined way to give resources back to the system when they're no longer needed.



Garbage collection is the process that

Returns memory to system

Java performs this automatically

- object marked for garbage collection if no references to object



Finalizer method

Returns resources to system

Java provides method finalize

- Defined in java.lang.Object
- Receives no parameters
- Returns void

- static Class Members & static import



static Class Members (Revisit) & static import

static keyword

- static class variable
 - Class-wide information
 - All class objects share same data

Access to a class's public static members

- Qualify the member name with the class name and a dot

static Class Members (Revisit) & static import

- Example:

```
1 // Fig. 8.12: Employee.java
2 // static variable used to maintain a count of the number of
3 // Employee objects in memory.
4
5 public class Employee
6 {
7     private static int count = 0; // number of Employees created
8     private String firstName;
9     private String lastName;
10
11     // initialize Employee, add 1 to static count and
12     // output String indicating that constructor was called
13     public Employee(String firstName, String lastName)
14     {
15         this.firstName = firstName;
16         this.lastName = lastName;
17
18         ++count; // increment static count of employees
19         System.out.printf("Employee constructor: %s %s; count = %d\n",
20             firstName, lastName, count);
21     }
22
23     // get first name
24     public String getFirstName()
25     {
26         return firstName;
27     }
28
29     // get last name
30     public String getLastName()
31     {
32         return lastName;
33     }
34
35     // static method to get static count value
36     public static int getCount()
37     {
38         return count;
39     }
40 } // end class Employee
```

static Class Members (Revisit) & static import

- Example:

```
1 // Fig. 8.13: EmployeeTest.java
2 // static member demonstration.
3
4 public class EmployeeTest
5 {
6     public static void main(String[] args)
7     {
8         // show that count is 0 before creating Employees
9         System.out.printf("Employees before instantiation: %d\n",
10             Employee.getCount());
11
12         // create two Employees; count should be 2
13         Employee e1 = new Employee("Susan", "Baker");
14         Employee e2 = new Employee("Bob", "Blue");
15
16         // show that count is 2 after creating two Employees
17         System.out.printf("\nEmployees after instantiation:\n");
18         System.out.printf("via e1.getCount(): %d\n", e1.getCount());
19         System.out.printf("via e2.getCount(): %d\n", e2.getCount());
20         System.out.printf("via Employee.getCount(): %d\n",
21             Employee.getCount());
22
23         // get names of Employees
24         System.out.printf("\nEmployee 1: %s %s\nEmployee 2: %s %s\n",
25             e1.getFirstName(), e1.getLastName(),
26             e2.getFirstName(), e2.getLastName());
27     }
28 } // end class EmployeeTest
```

Output

```
Employees before instantiation: 0
Employee constructor: Susan Baker; count = 1
Employee constructor: Bob Blue; count = 2
```

static Class Members (Revisit) & static import

- A **static** import declaration enables you to import the static members of a class or interface so you can access them via their unqualified names in your class
 - that is, the class name and a dot (.) are not required when using an imported static member.
 - A static import declaration has two forms
 - One that imports a particular static member

```
import static packageName.ClassName.staticMemberName;
```

- One that imports all static members of a class

```
import static packageName.ClassName.*;
```


static Class Members (Revisit) & static import

- Example:

```
1 // Fig. 8.14: StaticImportTest.java
2 // Static import of Math class methods.
3 import static java.lang.Math.*;
4
5 public class StaticImportTest
6 {
7     public static void main(String[] args)
8     {
9         System.out.printf("sqrt(900.0) = %.1f%n", sqrt(900.0));
10        System.out.printf("ceil(-9.8) = %.1f%n", ceil(-9.8));
11        System.out.printf("E = %f%n", E);
12        System.out.printf("PI = %f%n", PI);
13    }
14 } // end class StaticImportTest
```

Output

```
sqrt(900.0) = 30.0
ceil(-9.8) = -9.0
E = 2.718282
PI = 3.141593
```

- final Instance Variables



final Instance Variables

- final keyword
 - Indicates that variable is not modifiable
 - Any attempt to modify final variable results in error
 - `private final int INCREMENT = 5;`
 - Declares variable INCREMENT as a constant
- Enforces principle of least privilege

final Instance Variables

- Example:

```
class Bike9{
    final int speedlimit=90; //final variable
    void run(){
        speedlimit=400;
    }

    public static void main(String args[]){
        Bike9 obj=new Bike9();
        obj.run();
    }
} //end of class
```

Output

Compile Time Error

- Creating packages



Creating packages

- Package is a:
 - Group of related classes and interfaces
 - Help manage complexity of application components
 - Facilitate software reuse
 - Provide convention for unique class names
 - Popular package-naming convention
 - Reverse Internet domain name
 - e.g., com.deitel

Creating packages

- Create Package pack

```
package pack;  
public class A{  
    public void msg(){  
        System.out.println("Hello");  
    }  
}
```

- Create Package mypack

```
package mypack;  
import pack.*;  
class B{  
    public  
    static  
    void  
    main(S  
    tring  
    args[]){
```

```
    A obj = new A();
```

Output

Hello

Reference: <https://www.javatpoint.com/package>

Reference

S

- Deitel, P. J., & Deitel, H. M. (2017). Java: how to program late objects. Pearson.

Thank You

