جامعة بيرزيت

BIRZEIT UNIVERSITY

COMP1331

COMPUTER AND PROGRAMMING

Prepared by:
Dr. Mamoun Nawahdah

Approved by:
Computer Science Department
February 16, 2022

Contents

1 Elementary Java Programming 1

2 Selections 5

3 Loops 12

4 Methods 15

5 Recursion 19

6 Objects and Classes Part(I) 22

7 Objects and Classes Part(II) 27

8 Single-Dimensional Arrays 32

9 Multidimensional Arrays 37

10 Strings 42

11 Introduction to Exception Handling and Text I/O 47

12 Object-Oriented Thinking 50

Introduction

The aim of this lab manual is to help COMP1331 students to understand and apply a variety of fundamentals of object oriented programming concepts. Every lab session is provided with lab objectives, a brief context about the experiment's topic(s) to strength the student understanding to the lab material; a Java language syntax for the commands or statements that will be used; and a set of activities that allow the students to completely understand the topic. The activities in this manual are carefully prepared, studied and revised for students practice.

This lab manual starts by preparing the students to embark on the journey of learning Java programming language fundamentals. The students will begin to practice about Java and fundamental programming techniques with primitive data types, variables, constants, assignments, expressions, and operators (Lab 1), selection statements (Lab 2), loops (Lab 3), methods and mathematical functions (Lab 4), and an introduction to recursion (Lab 5). Then the students will practice object-oriented programming. The students will practice programming with objects and classes (Lab 6 and Lab 7), Single-Dimensional arrays (Lab 8), Multidimensional arrays (Lab 9), strings (Lab 10), introduction to exception handling and Text I/O (Lab 11), and class abstraction and basic relationships (Lab 12).

The material included in this manual has been adopted from the course's text-book:
Y. Daniel Liang, *Introduction to Java programming and data structures*, Twelfth edition, Pearson, 2019. (ISBN-13: 978-0-13-651996-6)

1 Elementary Java Programming

1.1 Objectives

- To understand the meaning of Java language specification, API, JDK™, and JRE™.
- To write a simple Java program.
- To display output on the console.
- To explain the basic syntax of a Java program.
- To create, compile, and run Java programs.
- To write Java programs to perform simple computations.
- To obtain input from the console using the **Scanner** class.
- To use identifiers to name variables, constants, methods, and classes.
- To use variables to store data.
- To program with assignment statements and assignment expressions.

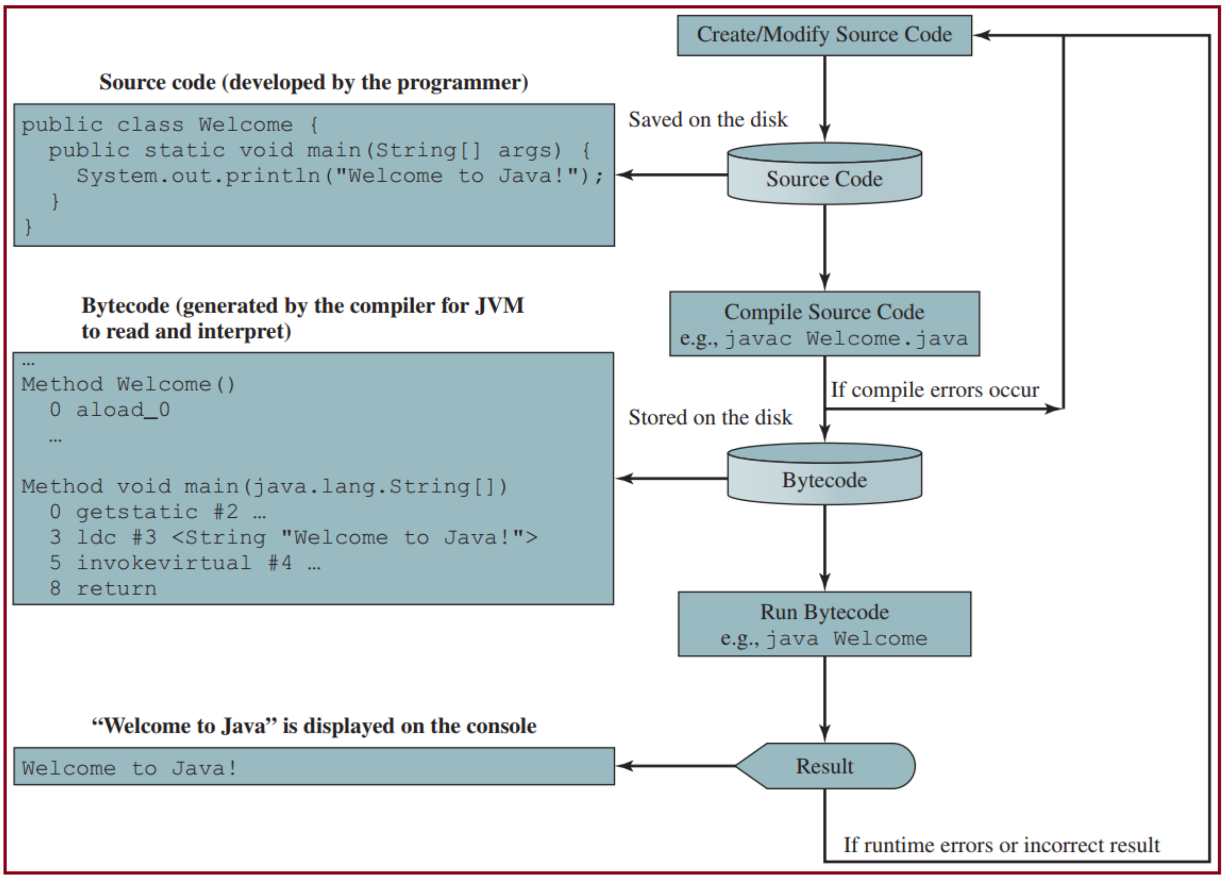
1.2 Context

Creating, Compiling, and Executing a Java Program

Figure 1 shows a simple Java program. You have to create your program and compile it before it can be executed. This process is repetitive, as shown in Figure 2. If your program has compile errors, you have to modify the program to fix them, then recompile it. If your program has runtime errors or does not produce the correct result, you have to modify the program, recompile it, and execute it again.

```java
public class Welcome {
  public static void main(String[] args) {
    // Display message Welcome to Java! on the console
    System.out.println("Welcome to Java!");
  }
}
```

Figure 1: A Simple Java Program



Create/Modify Source Code
Source code (developed by the programmer)
Saved on the disk
Source Code
Bytecode (generated by the compiler for JVM to read and interpret)
Compile Source Code e.g., javac Welcome.java
If compile errors occur
Stored on the disk
Bytecode
Run Bytecode e.g., java Welcome
"Welcome to Java" is displayed on the console
Result
If runtime errors or incorrect result

```
public class Welcome {
  public static void main(String[] args) {
    System.out.println("Welcome to Java!");
  }
}
```

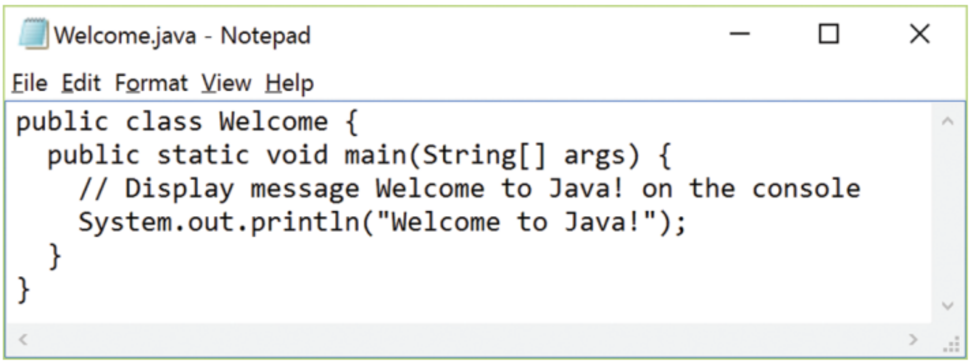
```
…
Method Welcome()
  0 aload_0
  …

Method void main(java.lang.String[])
  0 getstatic #2 …
  3 ldc #3 <String "Welcome to Java!">
  5 invokevirtual #4 …
  8 return
```

```
Welcome to Java!
```

Figure 2: The Java program-development process consists of repeatedly creating/modifying source code, compiling, and executing programs.

You can use any text editor to create and edit a Java source-code file. From the command window (CMD), you can use a text editor such as **Notepad** to create the Java source-code file, as shown in Figure 3.



```
public class Welcome {
  public static void main(String[] args) {
    // Display message Welcome to Java! on the console
    System.out.println("Welcome to Java!");
  }
}
```

Figure 3: You can create a Java source file using Windows Notepad.

***Note**: The source file must end with the extension **.java** and must have the same exact name as the public class name.*

A Java compiler translates a Java source file into a Java bytecode file. The following command compiles Welcome.java:

```
javac Welcome.java
```

If there aren't any syntax errors, the compiler generates a **bytecode** file with a **.class** extension. The following command runs the bytecode:

```
java Welcome
```

Reading Input from the Console

Reading input from the console enables the program to accept input from the user. Java uses **System.out** to refer to the standard output device, and **System.in** to the standard input device. To perform console output, you simply use the **println** method to display a primitive value or a string to the console. To perform console input, you need to use the **Scanner** class to create an object to read input from **System.in**, as follows:

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

The **Scanner** class is in the **java.util** package.

```
import java.util.Scanner;
```

You can invoke the **nextDouble()** method to read a double value as follows:

```
double radius = input.nextDouble();
```

This statement reads a number from the keyboard and assigns the number to radius.

You can invoke the **nextInt()** method to read an integer value as follows:

```
int width = input.nextInt();
```

1.3 Activities

- **Activity 1:** Write a program that displays **Welcome to Java**, **Welcome to Computer Science**, and **Programming is fun**.
- **Activity 2:** Write a program that displays the area and perimeter of a rectangle with a width of **5.5** and a height of **4.3** using the following formula:

```
area = width * height
```

- **Activity 3:** Assume that a runner runs **10** kilometers in **25** minutes and **30** seconds. Write a program that displays the average speed in miles per hour. *(Note **1** mile is equal to **1.6** kilometers.)*
- **Activity 4:** Write a program that prompts the user to enter the *width* and *height* of a rectangle and displays the perimeter and the area. Here is a sample run:

```
Enter the width of a rectangle: 3
Enter the height of a rectangle: 4
      The perimeter is 14
      The area is 12
```

- **Activity 5:** Write a program that reads a *Celsius* degree in a double value from the console, then converts it to *Fahrenheit*, and displays the result. The formula for the conversion is as follows:

$$\text{fahrenheit} = (9\ /\ 5) * \text{celsius} + 32$$

Hint: In Java, 9 / 5 is 1, but 9.0 / 5 is 1.8.

2 Selections

2.1 Objectives

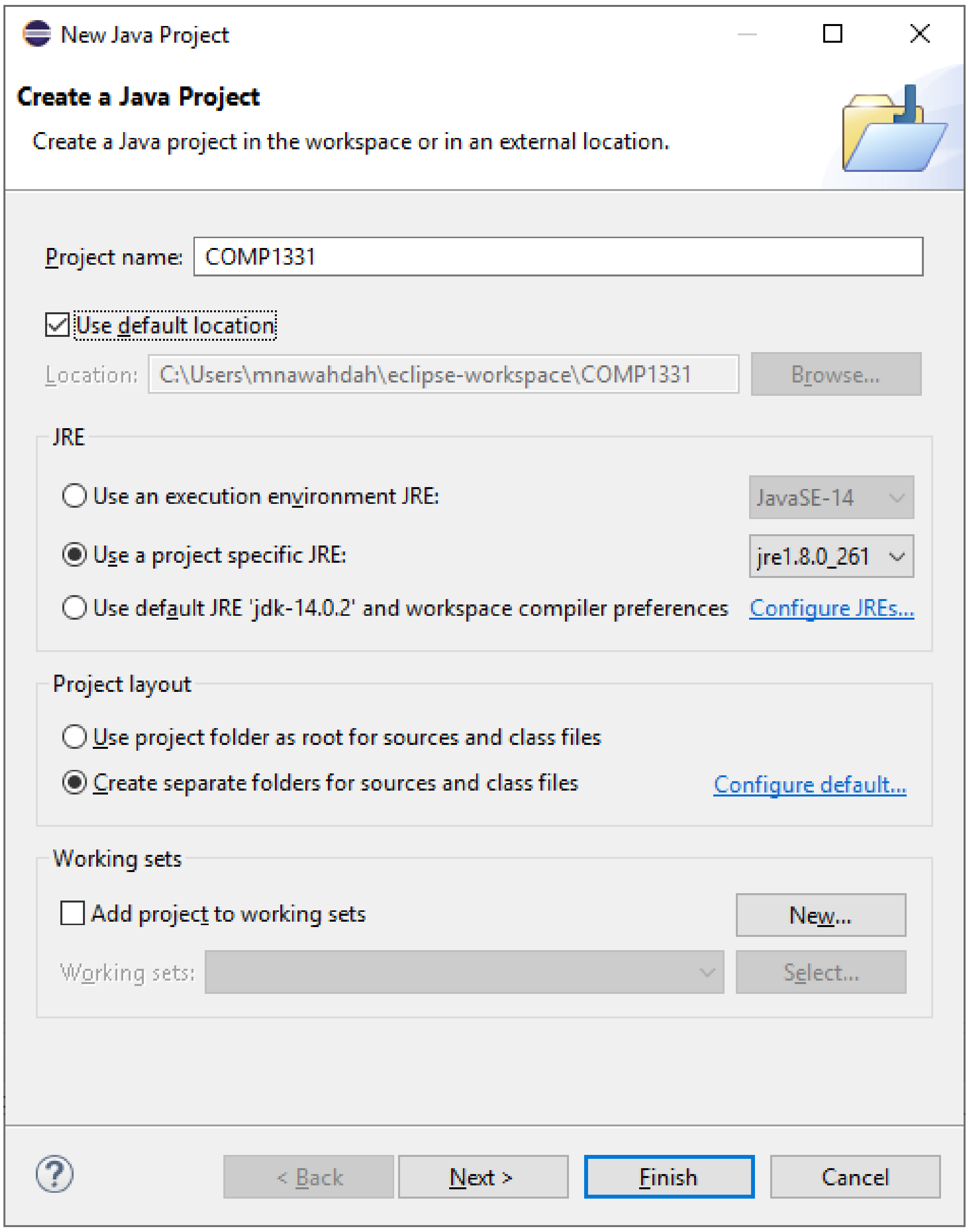
- To understand the meaning of IDE.
- To develop Java programs using Eclipse™.
- To implement selection control using one-way *if* statements.
- To combine conditions using logical operators.
- To implement selection control using *switch* statements.

2.2 Context

An **IDE** is an **I**ntegrated **D**evelopment **E**nvironment for rapidly developing programs.

Developing Java Programs Using Eclipse™
You can edit, compile, run, and debug Java Programs using Eclipse™.

- Creating a Java Project: you need to first create a project to hold all files. See Figure 4.



New Java Project

Create a Java Project

Create a Java project in the workspace or in an external location.

Project name: COMP1331

[✓] Use default location

Location: C:\Users\mnawahdah\eclipse-workspace\COMP1331

JRE

- ○ Use an execution environment JRE: JavaSE-14
- ● Use a project specific JRE: jre1.8.0_261
- ○ Use default JRE 'jdk-14.0.2' and workspace compiler preferences

Project layout

- ○ Use project folder as root for sources and class files
- ● Create separate folders for sources and class files

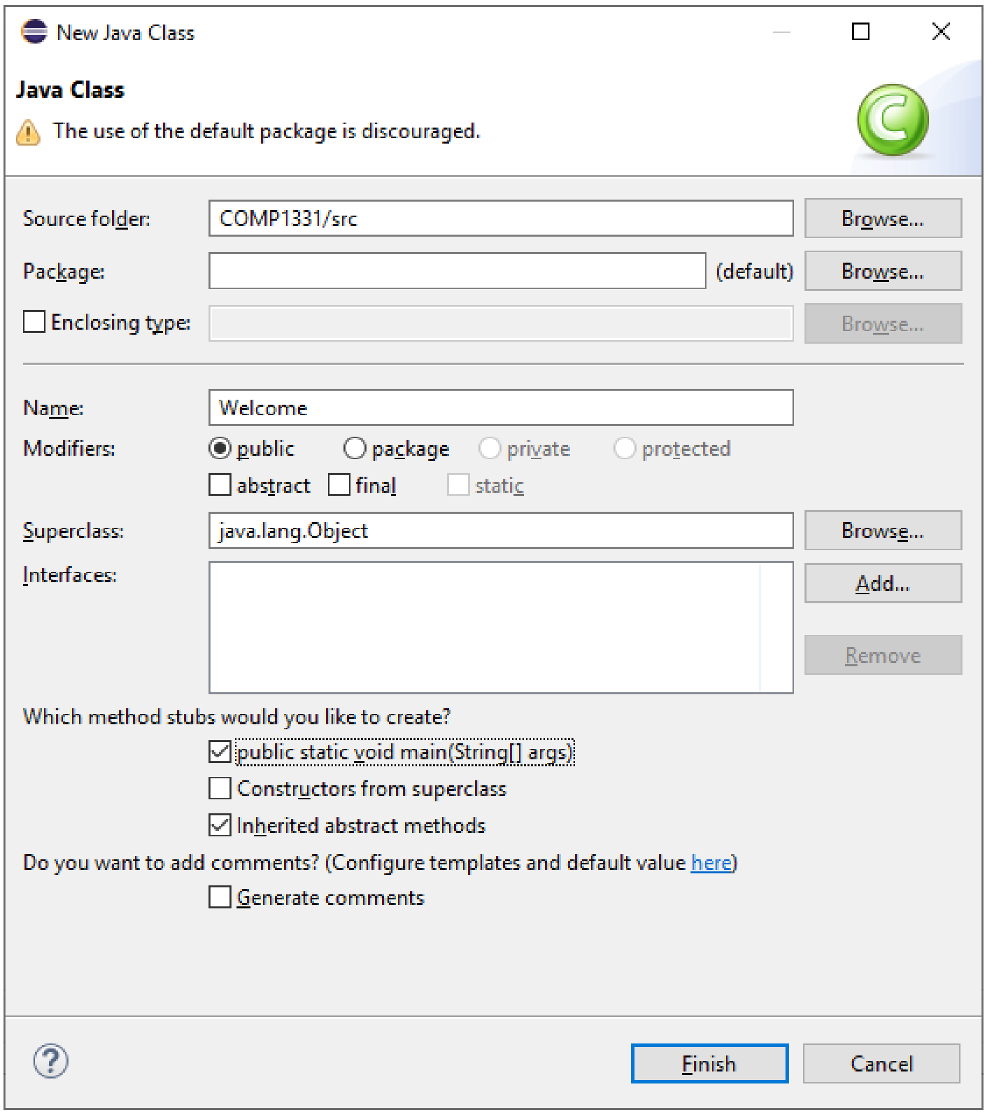
Working sets

[ ] Add project to working sets

Working sets:

Figure 4: The New Java Project dialog is for specifying a project name and the properties.

- Creating a Java Class: After a project is created, you can create Java programs (classes) in the project. see Figure 5



New Java Class

Java Class

The use of the default package is discouraged.

Source folder: COMP1331/src

Package: (default)

☐ Enclosing type:

Name: Welcome

Modifiers: [✓ public] package private protected

☐ abstract ☐ final ☐ static

Superclass: java.lang.Object

Interfaces:

Which method stubs would you like to create?

☑ public static void main(String[] args)

☐ Constructors from superclass

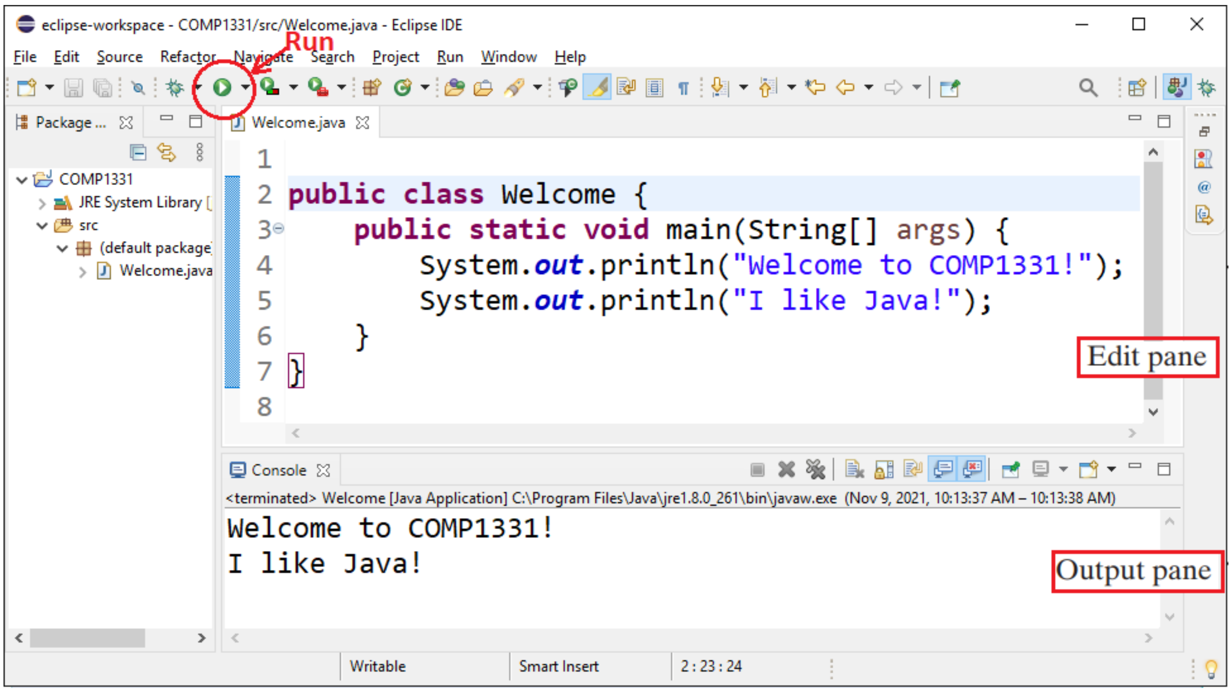
☑ Inherited abstract methods

Do you want to add comments? (Configure templates and default value here)

☐ Generate comments

Figure 5: The New Java Class dialog box is used to create a new Java class.

- Compiling and Running a Class: The Run icon automatically compiles the program if the program has been changed and run it. The output is displayed in the Console pane, as shown in Figure 6.



```java

public class Welcome {
    public static void main(String[] args) {
        System.out.println("Welcome to COMP1331!");
        System.out.println("I like Java!");
    }
}

```

```
<terminated> Welcome [Java Application] C:\Program Files\Java\jre1.8.0_261\bin\javaw.exe (Nov 9, 2021, 10:13:37 AM – 10:13:38 AM)
Welcome to COMP1331!
I like Java!
```

Figure 6: You can edit a program and run it in Eclipse™.

Boolean Data Type, Values, and Expressions

The program can decide which statements to execute based on a condition. Selection statements use conditions that are **Boolean** expressions. A Boolean expression is an expression that evaluates to a Boolean value: *true* or *false*. A variable that holds a Boolean value is known as a Boolean variable. Java provides six relational operators (also known as comparison operators), shown in Table 1, which can be used to compare two values.

Table 1: Relational Operators

Java Operator	Name
<	Less than.
<=	Less than or equal to.
>	Greater than.
>=	Greater than or equal to.
==	Equal to.
!=	Not equal to.

_if_ Statements

An if statement is a construct that enables a program to specify alternative paths of execution. Java has several types of selection statements:

- **One-way if statements**: An if statement executes statements if the boolean-expression evaluates to true.
```
    if (boolean-expression){
         statement(s);
    }
```
- **Two-way if-else statements**: An if-else statement decides the execution path based on whether the condition is true or false.
```
    if (boolean-expression){
         statement(s)-for-the-true-case;
    }
    else {
         statement(s)-for-the-false-case;
    }
```
- **Nested if statements**: An if statement can be inside another if statement to form a nested if statement.

switch Statements

A *switch* statement executes statements based on the value of a variable or an expression. Here is the full syntax for the *switch* statement:

```
switch (switch-expression) {
      case value1:  statement(s)1;
           break;
      case value2:  statement(s)2;
           break;
      ...
      case valueN: statement(s)N;
           break;
      default:  statement(s)-for-default;
 }
```

2.3 Activities

- **Activity 1:** Write a program that prompts the user to enter an integer between 1 and 12 and displays the English month names January, February, . . . , December for the numbers 1, 2, . . . , 12, accordingly.

- **Activity 2:** Write a program that prompts the user to enter three integers and display the integers in decreasing order.

- **Activity 3:** Write a program that prompts the user to enter a three-digit integer (e.g. *123*) and determines whether it is a palindrome integer. An integer is palindrome if it reads the same from right to left and from left to right. Here are sample runs of this program:

Sample run 1:

```
Enter a three-digit integer: 121
121 is a palindrome
```

Sample run 2:

```
Enter a three-digit integer: 123
123 is not a palindrome
```

- **Activity 4:** Write a program that reads three edges for a triangle and computes the perimeter if the input is valid. Otherwise, display that the input is invalid.
  The input is valid if the sum of every pair of two edges is greater than the remaining edge.

- **Activity 5:** Write a program that prompts the user to enter the coordinates of two points (*x1*, *y1*) and (*x2*, *y2*), and displays the line equation in the slope intercept form, i.e., $y = mx + b$. m and b can be computed using the following formula:

$$m = \frac{y_2 - y_1}{x_2 - x_1}$$

$$b = y_1 - mx_1$$

- **Activity 6:** Write a program that prompts the user to enter the exchange rate from currency in U.S. dollars $ to Jordanian dinar JOD. Prompt the user to enter 0 to convert from $ to JOD and 1 to convert from JOD to $. Prompt the user to enter the amount in $ or JOD to convert it to JOD or $, respectively. Here are the sample runs:

Sample run 1:

```
Enter the exchange rate from $ to JOD: 0.709
Enter 0 to convert $ to JOD and 1 vice versa: 0
Enter the $ amount: 100
100.0 $ is 70.9 JOD
```

Sample run 2:

```
Enter the exchange rate from $ to JOD: 0.709
Enter 0 to convert $ to JOD and 1 vice versa: 1
Enter the JOD amount: 100
100.0 JOD is 141.0438$
```

Sample run 3:

```
Enter the exchange rate from $ to JOD: 0.709
Enter 0 to convert $ to JOD and 1 vice versa: 5
Incorrect input: 5
```

3 Loops

3.1 Objectives

- To write programs for executing statements repeatedly using a *while* loop.
- To write loops using *do-while* statements.
- To write loops using *for* statements.
- To write nested loops.
- To implement program control with *break* and *continue*.

3.2 Context

The *while* Loop

A while loop executes statements repeatedly while the condition is true.The syntax for the while loop is as follows:

```
while (loop-continuation-condition) {
      // Loop body
      statement(s);
 }
```

The part of the loop that contains the statements to be repeated is called the *loop body*. A one-time execution of a loop body is referred to as an *iteration* of the loop.

The *do-while* Loop

A do-while loop is the same as a while loop except that it executes the loop body first then checks the loop continuation condition. Its syntax is as follows:

```
do {
      // Loop body
      statement(s);
 } while (loop-continuation-condition);
```

You can write a loop using either the while loop or the do-while loop. Sometimes one is a more convenient choice than the other.

The *for* Loop

The for loop statement starts with the keyword *for*, followed by a pair of parentheses enclosing the control structure of the loop. The syntax of a for loop is as follows:

```
for (initial-action; loop-continuation-condition;
      action-after-each-iteration) {
      // Loop body
      statement(s);
 }
```

Nested Loops

A loop can be nested inside another loop. Nested loops consist of an outer loop and one or more inner loops. Each time the outer loop is repeated, the inner loops are re-entered, and started a new.

Keywords *break* and *continue*

The *break* and *continue* keywords provide additional controls in a loop. You have used the keyword *break* in a *switch* statement. You can also use *break* in a loop to immediately terminate the loop. You can also use the *continue* keyword in a loop. When it is encountered, it ends the current iteration and program control goes to the end of the loop body. In other words, *continue* breaks out of an iteration, while the *break* keyword breaks out of a loop.

3.3 Activities

- **Activity 1:** Write a program that displays all the numbers from 1 to 1,000 (10 per line) that are divisible by 3 and 4.

- **Activity 2:** Write a program that displays all the numbers from 1 to 1,000 (10 per line) that are divisible by 3 or 4, but not both.

- **Activity 3:** The greatest common divisor (GCD) of two integers *n1* and *n2* is as follows: First find *d* to be the minimum of *n1* and *n2*, then check whether *d*, *d-1*, *d-2*, ..., *2*, or *1* is a divisor for both *n1* and *n2* in this order. The first such common divisor is the greatest common divisor for *n1* and *n2*. Write a program that prompts the user to enter two positive integers and displays the GCD.

- **Activity 4:** You can approximate π by using the following summation:

  $$\pi = 4(1 - \frac{1}{3} + \frac{1}{5} - \frac{1}{7} + \frac{1}{9} - \frac{1}{11} + ... + \frac{(-)^{i+1}}{2i-1})$$

  Write a program that displays the π value for i = 100, 1000, and 10000.

- **Activity 5:** You can approximate **e** using the following summation:

  $$e = 1 + \frac{1}{1!} + \frac{1}{2!} + \frac{1}{3!} + \frac{1}{4!} + ... + \frac{1}{i!}$$

  Write a program that displays the **e** value for i = 1, 2, 3, ... , and 10.

- **Activity 6:** A positive integer is called a *perfect number* if it is equal to the sum of all of its positive divisors, excluding itself. For example, 6 is the first perfect number because *6 = 3 + 2 + 1.* The next is *28 = 14 + 7 + 4 + 2 + 1.* Write a program to find all the perfect numbers < 10,000.

4 Methods

4.1 Objectives

- To define methods with formal parameters.
- To invoke methods with actual parameters (i.e., arguments).
- To define methods with a return value.
- To define methods without a return value and distinguish the differences between void methods and value-returning methods.
- To pass arguments by value.
- To determine the scope of variables.
- To solve mathematical problems by using the methods in the **Math** class.

4.2 Context

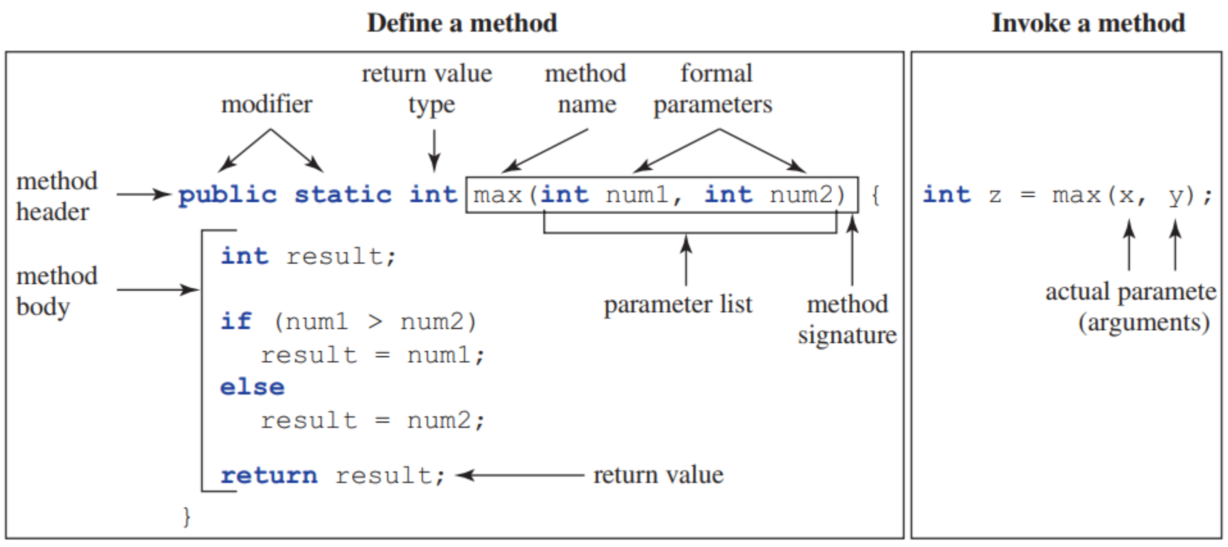
Methods can be used to define reusable code and organize and simplify coding, and make code easy to maintain.

Defining and Calling a Method

A method definition consists of method *name*, *parameters*, *return* value type, and *body*. The syntax for defining a method is as follows:

```
modifier returnValueType methodName(list of parameters) {
  // Method body;
  }
```

Let's look at a method defined to find the larger between two integers. This method, named max, has two int parameters, *num1* and *num2*, the larger of which is returned by the method. Figure 7 illustrates the components of this method.



```
public static int max(int num1, int num2) {
  int result;

  if (num1 > num2)
    result = num1;
  else
    result = num2;

  return result;
}
```

```
int z = max(x, y);
```

Figure 7: A method definition consists of a method header and a method body.

If a method returns a value, it is called a *value-returning method*; otherwise, it is called a *void method*. The variables defined in the method header are known as *formal parameters* or simply *parameters*. A parameter is like a placeholder: when a method is invoked, you pass a value to the parameter. This value is referred to as an *actual parameter* or *argument*. The method name and the parameter list together constitute the *method signature*.

To execute the method, you have to *call* or *invoke* it. There are two ways to call a method, depending on whether the method returns a value or not. If a method returns a value, a call to the method is usually treated as a value. For example:

```
int larger = max(3, 4);
```

Passing Arguments by Values

When you invoke a method with an argument, the value of the argument is passed to the parameter. This is referred to as *pass-by-value*. If the argument is a variable rather than a literal value, the value of the variable is passed to the parameter. The variable is not affected, regardless of the changes made to the parameter inside the method.

The Scope of Variables

A variable defined inside a method is referred to as a *local variable*. The scope of a local variable starts from its declaration and continues to the end of the block that contains the variable.

Common Mathematical Functions

Java provides many useful methods in the **Math** class for performing common mathematical functions. They can be categorized as *trigonometric* methods (see Table 2), *exponent* methods (see Table 3), and *service* methods (see Table 4). In addition to methods, the **Math** class provides two useful double constants, *PI* and *E*.

Table 2: Trigonometric Methods in the **Math** Class

Method	Description
sin(radians)	Returns the trigonometric sine of an angle in radians.
cos(radians)	Returns the trigonometric cosine of an angle in radians.
tan(radians)	Returns the trigonometric tangent of an angle in radians.
toRadians(degree)	Returns the angle in radians for the angle in degrees.
toDegrees(radians)	Returns the angle in degrees for the angle in radians.

Table 3: Exponent Methods in the **Math** Class

Method	Description
exp(x)	Returns e raised to power of x (e^x).
log(x)	Returns the natural logarithm of x ($ln(x) = log_e(x)$).
log10(x)	Returns the base 10 logarithm of x ($log_{10}(x)$).
pow(a, b)	Returns a raised to the power of b (a^b).
sqrt(x)	Returns the square root of x ($\sqrt{x}$) for x>=0.

Table 4: Rounding Methods in the **Math** Class

Method	Description
ceil(x)	x is rounded up to its nearest integer. This integer is returned as a double value.
floor(x)	x is rounded down to its nearest integer. This integer is returned as a double value.
rint(x)	x is rounded to its nearest integer. If x is equally close to two integers, the even one is returned as a double value.
round(x)	Returns (int)Math.floor(x + 0.5) if x is a float and returns (long)Math.floor(x + 0.5) if x is a double.

4.3 Activities

- **Activity 1:** Write a method that computes the sum of the digits in an integer. Use the following method header:
  ```
  public static int sumDigits(long n)
  ```
  For example, sumDigits(234) returns 9 (= 2 + 3 + 4). Use a loop to repeatedly extract and remove the digit until all the digits are extracted.
  Write a test program that prompts the user to enter an integer then displays the sum of all its digits.

- **Activity 2:** Write a method with the following header to display an integer in reverse order:
  ```
  public static void reverse(int number)
  ```
  For example, reverse(3456) displays 6543.
  Write a test program that prompts the user to enter an integer then displays its reversal.

- **Activity 3:** Write a method with the following header to display if an integer is prime or not:
  ```
  public static boolean isPrime(int number)
  ```
  Use this method to find the number of prime numbers less than 10000.

- **Activity 4:** A regular polygon is an **n**-sided polygon in which all sides are of the same length and all angles have the same degree. The formula for computing the area of a regular polygon is:

  $$Area = \frac{n * s^2}{4 * tan(\frac{\pi}{n})}$$

  Here, **s** is the length of a side.
  Write a program that prompts the user to enter the number of sides and their length of a regular polygon and displays its area. Here is a sample run:

  ```
  Enter the number of sides: 5
  Enter the side: 6.5
      The area of the polygon is 72.69017017488385
  ```

- **Activity 5:** Write a program that prompts the user to enter a decimal number between 0 and 255 and displays its corresponding binary value.

5 Recursion

5.1 Objectives

- To describe what a recursive method is and the benefits of using recursion.
- To develop recursive methods for recursive mathematical functions.
- To explain how recursive method calls are handled in a call stack.
- To solve problems using recursion.

5.2 Context

Recursion is a technique that leads to elegant solutions to problems that are difficult to program using simple loops.
To use recursion is to program using *recursive methods* that is, to use methods that invoke themselves. Recursion is a useful programming technique. In some cases, it enables you to develop a natural, straightforward, simple solution to an otherwise difficult problem.

A recursive call can result in many more recursive calls because the method keeps on dividing a subproblem into new subproblems. For a recursive method to terminate, the problem **must** eventually be reduced to a stopping case, at which point the method returns a result to its caller. The caller then performs a computation and returns the result to its own caller. This process continues until the result is passed back to the original caller.

Problem Solving Using Recursion

If you think recursively, you can solve many problems using recursion.
All recursive methods have the following characteristics:

- The method is implemented using an *if-else* or a *switch* statement that leads to different cases.
- One or more *base cases* (the *simplest case*) are used to stop recursion.
- Every recursive call reduces the original problem, bringing it increasingly closer to a base case until it becomes that case.

In general, to solve a problem using recursion, you break it into subproblems. Each subproblem is the same as the original problem, but smaller in size. You can apply the same approach to each subproblem to solve it recursively.

5.3 Activities

- **Activity 1:** The factorial of a number **n** can be recursively defined as follows:
  - 0! = 1;
  - n! = n * (n - 1)!   n > 0

  Implement the factorial method using recursion.
  Write a program that prompts the user to enter an integer and displays its factorial.

- **Activity 2:** The greatest common divisor **gcd(m, n)** is defined recursively as follows:
  - If m % n is 0, **gcd(m, n)** is n.
  - Otherwise, **gcd(m, n)** is **gcd(n, m % n)**.

  Write a recursive method to find the GCD.
  Write a test program that prompts the user to enter two integers and displays their GCD.

- **Activity 3:** Write a recursive method to compute the following series:

  $$m(i) = 1 + \frac{1}{2} + \frac{1}{3} + \frac{1}{4} + ... + \frac{1}{i}$$

  Write a test program that displays m(i) for i = 1, 2, 3,. . . , 20.

- **Activity 4:** Write a recursive method that displays an int value reversely on the console using the following header:
  ```
      public static void reverseDisplay(int value)
  ```
  For example, reverseDisplay(12345) displays 54321.
  Write a test program that prompts the user to enter an integer and displays its reversal.

- **Activity 5:** Write a recursive method that computes the sum of the digits in an integer. Use the following method header:
  ```
      public static int sumDigits(long n)
  ```
  For example, *sumDigits(234)* returns 2 + 3 + 4 = 9.
  Write a test program that prompts the user to enter an integer and displays its sum.

6 Objects and Classes Part(I)

6.1 Objectives

- To describe objects and classes, and use classes to model objects.
- To use **UML** graphical notation to describe classes and objects.
- To demonstrate how to define classes and create objects.
- To create objects using constructors.
- To define a reference variable using a reference type and access objects via object reference variables.
- To access an object's data and methods using the object member access operator (**.**).
- To define data fields of reference types and assign default values for an object's data fields.
- To distinguish between object reference variables and primitive data-type variables.

6.2 Context

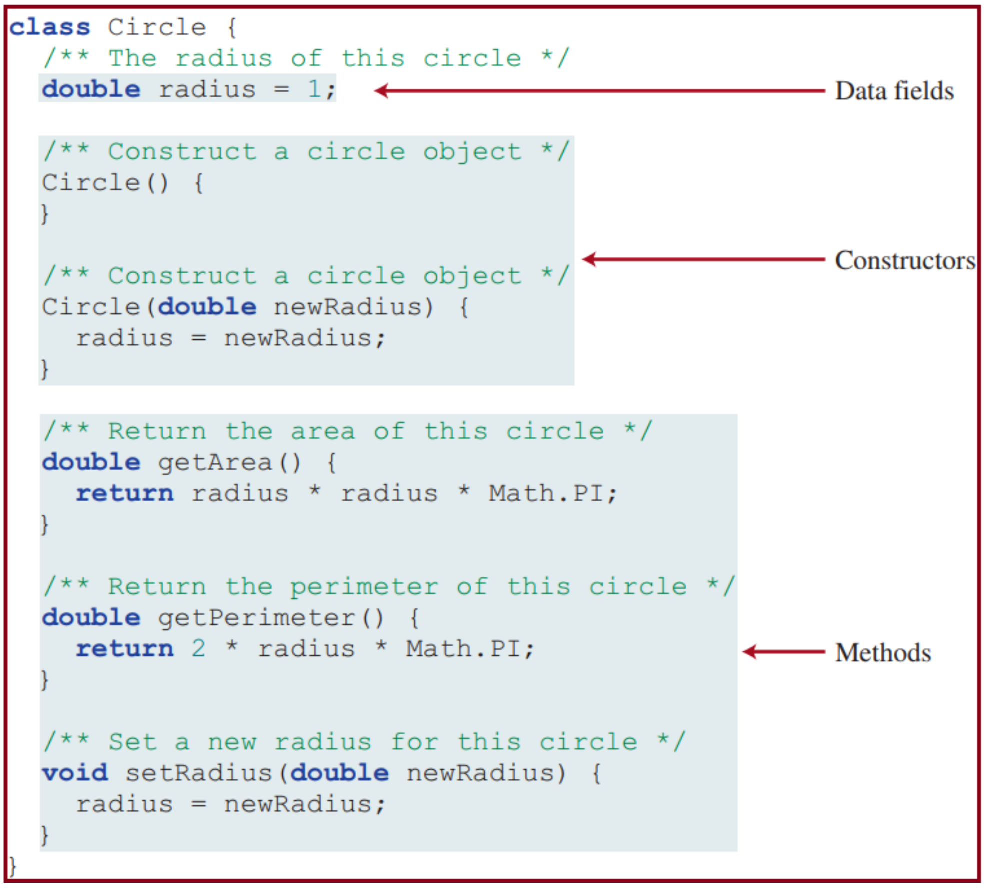
Defining Classes for Objects Object-oriented programming (**OOP**) involves programming using objects. An object represents an entity in the real world that can be distinctly identified. An object has a unique *identity*, *state*, and *behavior*.

- The *state* of an object (also known as its *properties* or *attributes*) is represented by data fields with their current values.
- The *behavior* of an object (also known as its *actions*) is defined by methods. To invoke a method on an object is to ask the object to perform an action.

Objects of the same type are defined using a common class. A *class* is a template that defines what an object's data fields and methods will be. An object is an instance of a class. You can create many instances of a class. Creating an instance is referred to as *instantiation*.

A class provides methods of a special type, known as *constructors*, which are invoked to create a new object. A constructor can perform any action, but constructors are designed to perform initializing actions, such as initializing the data fields of objects. Figure 8 shows an example of defining the class for **Circle** objects.

The illustration of the class in Figure 8 can be standardized using Unified Modeling Language (**UML**) notation. This notation, as shown in Figure 9, is called a *UML class diagram*, or simply a *class diagram*.
In the class diagram, the data field is denoted as
dataFieldName: dataFieldType
The constructor is denoted as
ClassName(parameterName: parameterType)
The method is denoted as
methodName(parameterName: parameterType): returnType



```java
class Circle {
  /** The radius of this circle */
  double radius = 1;

  /** Construct a circle object */
  Circle() {
  }

  /** Construct a circle object */
  Circle(double newRadius) {
    radius = newRadius;
  }

  /** Return the area of this circle */
  double getArea() {
    return radius * radius * Math.PI;
  }

  /** Return the perimeter of this circle */
  double getPerimeter() {
    return 2 * radius * Math.PI;
  }

  /** Set a new radius for this circle */
  void setRadius(double newRadius) {
    radius = newRadius;
  }
}
```

Figure 8: A class is a construct that defines objects of the same type.

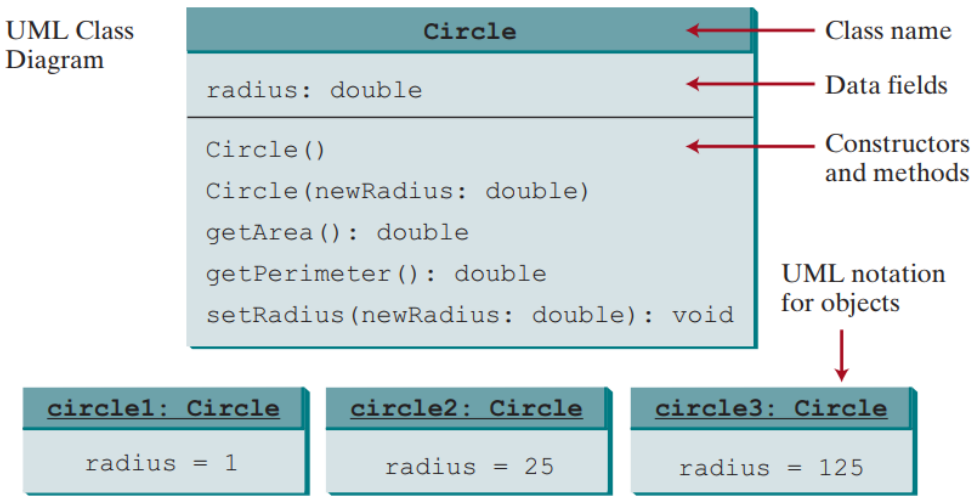


Figure 9: Classes and objects can be represented using UML notation.

6.3 Activities

- **Activity 1:** Design a class named **Rectangle** to represent a rectangle. The class contains:

  - Two double data fields named *width* and *height* that specify the width and height of the rectangle. The default values are `1` for both width and height.
  - A no-arg constructor that creates a default rectangle.
  - A constructor that creates a rectangle with the specified width and height.
  - A method named *getArea()* that returns the area of this rectangle.
  - A method named *getPerimeter()* that returns the perimeter.

  Draw the UML diagram for the class then implement the class.
  Write a test program that creates two **Rectangle** objects - one with width `4` and height `40`, and the other with width `3.5` and height `35.9`.
  Display the width, height, area, and perimeter of each rectangle in this order.

- **Activity 2:** Design a class named **Fan** to represent a fan. The class contains:

  - An int data field named *speed* that specifies the speed of the fan (the default is `0`).
  - A boolean data field named *on* that specifies whether the fan is on (the default is `false`).
  - A double data field named *radius* that specifies the radius of the fan (the default is `5`).
  - A string data field named *color* that specifies the color of the fan (the default is "`blue`").
  - A no-arg constructor that creates a default fan.
  - A method named *toString()* that returns a string description for the fan. If the fan is `on`, the method returns the fan speed, color, and radius in one combined string. If the fan is not `on`, the method returns the fan color and radius along with the string "`fan is off`" in one combined string.

  Draw the UML diagram for the class then implement the class.
  Write a test program that creates two **Fan** objects. Assign maximum speed of `100`, radius `10`, color `yellow`, and turn it on to the first object.
  Assign medium speed of `50`, radius `5`, color `blue`, and turn it off to the second object.
  Display the objects by invoking their *toString* method.

- **Activity 3:** In an n-sided regular polygon, all sides have the same length and all angles have the same degree (i.e., the polygon is both equilateral and equiangular). Design a class named **RegularPolygon** that contains:
  - An int data field named *n* that defines the number of sides in the polygon with default value `5`.
  - A double data field named *side* that stores the length of the side with default value `2`.
  - A double data field named *x* that defines the x-coordinate of the polygon's center with default value `10`.
  - A double data field named *y* that defines the y-coordinate of the polygon's center with default value `10`.
  - A no-arg constructor that creates a regular polygon with default values.
  - A constructor that creates a regular polygon with the specified number of sides and length of side, centered at (`10`, `10`).
  - A constructor that creates a regular polygon with the specified number of sides, length of side, and x- and y-coordinates.
  - The method *getPerimeter()* that returns the perimeter of the polygon.
  - The method *getArea()* that returns the area of the polygon. The formula for computing the area of a regular polygon is:

$$Area = \frac{n * s^2}{4 * tan(\frac{\pi}{n})}$$

  Draw the UML diagram for the class then implement the class.
  Write a test program that creates three **RegularPolygon** objects, created using the no-arg constructor, using *RegularPolygon(4, 3)*, and using *RegularPolygon(6, 4.5, 0, 0)*. For each object, display its perimeter and area.

7 Objects and Classes Part(II)

7.1 Objectives

- To use the Java library classes **Date** and **Random**.
- To distinguish between *instance* and *static* variables and methods.
- To define *private* data fields with appropriate getter and setter methods.
- To distinguish between *public*, *private*, and default visibility modifiers.
- To encapsulate data fields to make classes easy to maintain.
- To determine the scope of variables in the context of a class.
- To use the keyword *this* to refer to the calling object itself.

7.2 Context

Using Classes from the Java Library

The Java API contains a rich set of classes for developing Java programs. For example Java provides a system-independent encapsulation of date and time in the **java.util.Date** class, as shown in Table 5.

Table 5: A **Date** object represents a specific date and time.

Method	Description
Date()	Constructs a Date object for the current time.
Date(elapseTime: long)	Constructs a Date object for a given time in milliseconds elapsed since `January 1, 1970, GMT`.
toString()	Returns a string representing the date and time.
getTime()	Returns the number of milliseconds since `January 1, 1970, GMT`.
setTime(elapseTime: long)	Sets a new elapse time in the object.

Another useful class is **java.util.Random** class. This class used to generate random numbers, as shown in Table 6, which can generate a random *int*, *long*, *double*, *float*, and *boolean* value.

Table 6: A **Random** object can be used to generate random values.

Method	Description
Random()	Constructs a Random object with the current time as its seed.
nextInt()	Returns a random int value.
nextInt(n: int)	Returns a random int value between 0 and n (excluding n).
nextLong()	Returns a random long value.
nextDouble()	Returns a random double value between 0.0 and 1.0 (excluding 1.0).
nextFloat()	Returns a random float value between 0.0F and 1.0F (excluding 1.0F).
nextBoolean()	Returns a random boolean value.

Static Variables, Constants, and Methods

A *static* variable is shared by all objects of the class. A *static* method cannot access *instance* members (i.e., *instance* data fields and methods) of the class.

Note: static *variables and methods are underlined in the* ***UML*** *class diagram.*

Constants in a class are shared by all objects of the class. Thus, constants should be declared as *final static*. For example, the constant **PI** in the **Math** class is defined as follows:

```
final static double PI = 3.14159265358979323846;
```

Visibility Modifiers

You can use the *public* visibility modifier for classes, methods, and data fields to denote they can be accessed from any other classes. If no visibility modifier is used, then by default the classes, methods, and data fields are accessible by any class in the same package. In addition to the *public* and *default* visibility modifiers, Java provides the *private* modifier for class members. The *private* modifier makes methods and data fields accessible only from within its own class.

Data Field Encapsulation

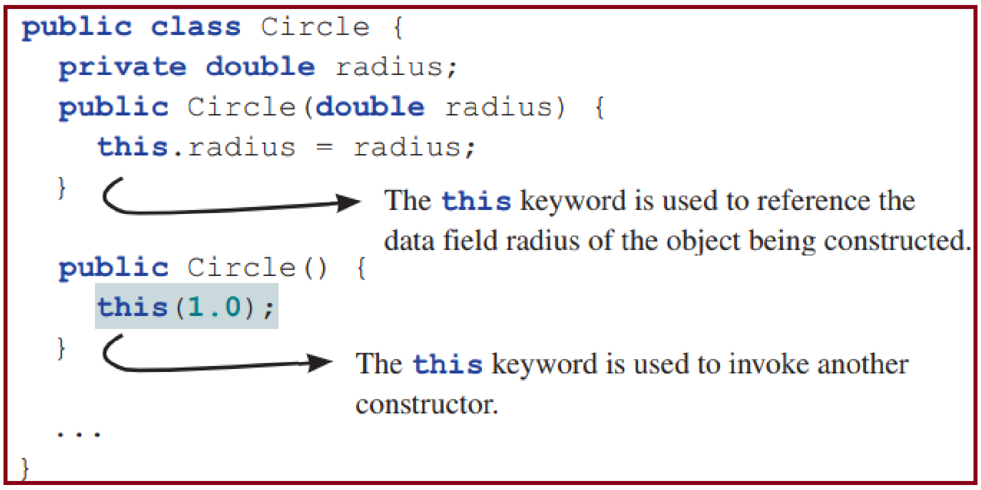
Making data fields private protects data and makes the class easy to maintain. To prevent direct modifications of data fields, you should declare the data fields *private*, using the private modifier. To make a private data field accessible, provide a *getter* method to return its value. To enable a private data field to be updated, provide a *setter* method to set a new value. A *getter* method is also referred to as an *accessor* and a *setter* to a *mutator*.

The Scope of Variables

Instance and *static* variables in a class are referred to as the *class's variables* or *data fields*. A variable defined inside a method is referred to as a *local variable*. The scope of a class's variables is the entire class. If a local variable has the same name as a class's variable, the local variable takes precedence and the class's variable with the same name is *hidden*.

The this Reference

The keyword *this* refers to the calling object. It can also be used inside a constructor to invoke another constructor of the same class. Figure 10 shows the different usage of *this* keyword.



```
public class Circle {
  private double radius;
  public Circle(double radius) {
    this.radius = radius;
  }

  public Circle() {
    this(1.0);
  }

  ...
}
```

The **this** keyword is used to reference the data field radius of the object being constructed.

The **this** keyword is used to invoke another constructor.

Figure 10: Using *this* to Reference Data Fields and to Invoke a Constructor.

7.3 Activities

- **Activity 1:** Write a program that creates a **Date** object, sets its elapsed time to 10000, 100000, 1000000, 10000000, 100000000, 1000000000, 10000000000, 100000000000 , and 1000000000000, and displays the date and time using the *toString()* method, respectively.

- **Activity 2:** Write a program that creates a **Random** object and displays the first 50 random integers between 0 and 100 using the *nextInt(100)* method.

- **Activity 3:** Design a class named **StopWatch**. The class contains:
  - Private long data fields *startTime* and *endTime* with getter methods.
  - A no-arg constructor that initializes *startTime* and *endTime* with the current time.
  - A method named *start()* that resets the *startTime* to the current time.
  - A method named *stop()* that sets the *endTime* to the current time.
  - A method named *getElapsedTime()* that returns the elapsed time for the stop-watch in milliseconds.

  Draw the **UML** diagram for the class then implement the class.
  Write a test program that measures the execution time of generating `100,000` random numbers.

- **Activity 4:** Design a class named **Account** that contains:
  - A private int data field named *id* for the account (default `0`).
  - A private double data field named *balance* for the account (default `0`).
  - A private double data field named *annualInterestRate* that stores the current interest rate (default 0). Assume that all accounts have the same interest rate.
  - A private Date data field named *dateCreated* that stores the date when the account was created.
  - A no-arg constructor that creates a default account.
  - A constructor that creates an account with the specified *id* and initial *balance.*
  - The *accessor* and *mutator* methods for *id*, *balance*, and *annualInterestRate.*
  - The accessor method for *dateCreated.*
  - A method named *getMonthlyInterestRate()* that returns the monthly interest rate.
  - A method named *getMonthlyInterest()* that returns the monthly interest.
  - A method named *withdraw* that withdraws a specified amount from the account.
  - A method named *deposit* that deposits a specified amount to the account.

  Draw the **UML** diagram for the class then implement the class.
  Write a test program that creates an **Account** object with an account ID of `1122`, a balance of `$20000`, and an annual interest rate of `4.5%`. Use the *withdraw* method to withdraw $2500, use the *deposit* method to deposit $3000, and print the balance, the monthly interest, and the date when this account was created.

  Hint: The method *getMonthlyInterest()* is to return monthly interest, not the interest rate. Monthly interest is *balance * monthlyInterestRate.* monthlyInterestRate is *annualInterestRate / 12.* Note annualInterestRate is a percentage, for example `4.5%`. You need to divide it by `100`.

8 Single-Dimensional Arrays

8.1 Objectives

- To declare array reference variables and create arrays.
- To obtain array size using *arrayRefVar.length* and know default values in an array.
- To access array elements using indexes.
- To declare, create, and initialize an array using an array *initializer*.
- To copy contents from one array to another.
- To develop and invoke methods with array arguments and return values.

8.2 Context

Array Basics

An array is used to store a collection of data, but often we find it more useful to think of an array as a collection of variables of the same type. To use an array in a program, you must declare a variable to reference the array and specify the array's element type. Here is the syntax for declaring an array variable.

```
elementType[] arrayRefVar;
```

After an array variable is declared, you can create an array by using the ***new*** operator and assign its reference to the variable with the following syntax:

```
arrayRefVar = new elementType[arraySize];
```

To assign values to the elements, use the syntax:

```
arrayRefVar[index] = value;
```

When space for an array is allocated, the array size must be given, specifying the number of elements that can be stored in it. The size of an array cannot be changed after the array is created. Size can be obtained using *arrayRefVar.length*. When an array is created, its elements are assigned the default value of **0** for the numeric primitive data types, **\U0000** for char types, and **false** for boolean types.
Here is an example of creating an array:

```
double[] myList = new double[6];
```

For example, the following code initializes the above array:

```
myList[0] = 5.6;
myList[1] = 4.5;
myList[2] = 3.3;
myList[3] = 13.2;
myList[4] = 4.0;
myList[5] = 34.33;
```

This array is illustrated in Figure 11.

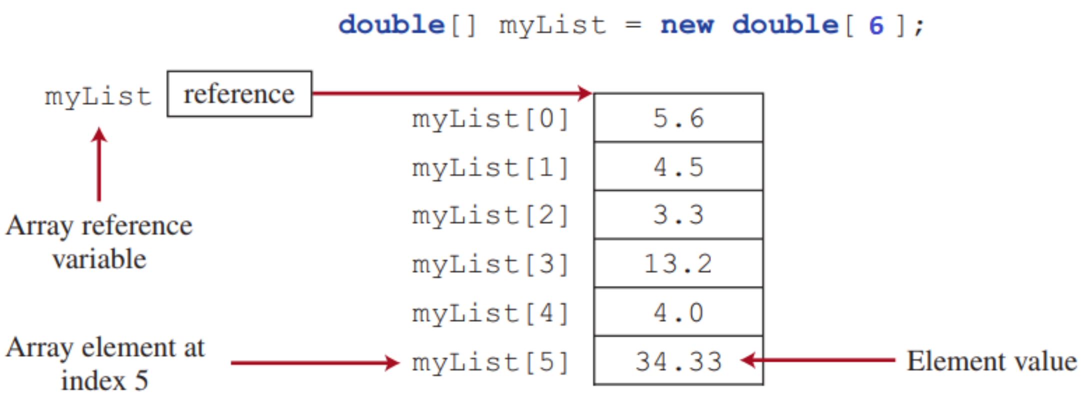


Figure 11: The array myList has 6 elements of double type and int indices from 0 to 5.

Java has a shorthand notation, known as the array *initializer*, which combines the declaration, creation, and initialization of an array in one statement using the following syntax:

```
elementType[] arrayRefVar = {value0, value1, ..., valuek};
```

Assigning one array variable to another array variable actually copies one reference to another and makes both variables point to the same memory location.To copy the contents of one array into another, you have to copy the array's individual elements into the other array. You can write a loop to copy every element from the source array to the corresponding element in the target array. Another approach is to use the *arraycopy* method in the **java.lang.System** class to copy arrays instead of using a loop. The syntax for *arraycopy* is:

```
arraycopy(sourceArray, srcPos, targetArray, tarPos, length);
```

Java uses *pass-by-value* to pass arguments to a method. There are important differences between passing the values of variables of primitive data types and passing arrays.

- For an argument of a primitive type, the argument's value is passed.
- For an argument of an array type, the value of the argument is a reference to an array;

this reference value is passed to the method. Thus, if you change the array in the method, you will see the change outside the method.
You can pass arrays when invoking a method. A method may also return an array. When a method returns an array, the reference of the array is returned.

8.3 Activities

- **Activity 1:** Write a program that reads `10` integers then displays them in the reverse of the order in which they were read.

- **Activity 2:** Write a program that generates `100` random integers between `0` and `9` and displays the count for each number.
  (Hint: Use an array of 10 integers, say counts, to store the counts for the number of 0s, 1s, . . . , 9s.)

- **Activity 3:** Write a method that finds the smallest element in an array of double values using the following header:
  ```
  public static double min(double[] array)
  ```
  Write a test program that prompts the user to enter `10` numbers, invokes this method to return the minimum value, and displays the minimum value.

- **Activity 4:** Write a method that returns a new array by eliminating the duplicate values in the array using the following method header:
  ```
  public static int[] eliminateDuplicates(int[] list)
  ```
  Write a test program that reads in `10` integers, invokes the method, and displays the distinct numbers separated by exactly one space.
  Here is a sample run of the program:
  ```
  Enter 10 numbers: 1 2 3 2 1 6 3 4 5 2
  The distinct numbers are: 1 2 3 6 4 5
  ```

- **Activity 5:** The arrays *list1* and *list2* are identical if they have the same contents. Write a method that returns **true** if *list1* and *list2* are identical, using the following header:

  ```
  public static boolean equals(int[] list1, int[] list2)
  ```

  Write a test program that prompts the user to enter two lists of integers and displays whether the two are identical.
  Here are the sample runs.

Sample run 1:

```
Enter list1 size : 5
Enter list2 size : 5
Enter list1 contents: 2 5 6 1 6
Enter list2 contents: 2 5 6 1 6
Two lists are strictly identical
```

Sample run 2:

```
Enter list1 size : 5
Enter list2 size : 5
Enter list1 contents: 2 5 6 6 1
Enter list2 contents: 2 5 6 1 6
Two lists are not strictly identical
```

- **Activity 6:** Write the following method that merges two sorted lists into a new sorted list:

  ```
  public static int[] merge(int[] list1, int[] list2)
  ```

9 Multidimensional Arrays

9.1 Objectives

- To declare variables for two-dimensional arrays, create arrays, and access array elements in a two-dimensional array using row and column indices.
- To pass two-dimensional arrays to methods.
- To use multidimensional arrays.
- To store and process objects in arrays.

9.2 Context

A two-dimensional array is an array that contains other arrays as its elements. You can use a two-dimensional array to store a matrix or a table. The syntax for declaring a two-dimensional array is as follows:

```
elementType[][] arrayRefVar;
```

As an example, here is how you would declare a two-dimensional array variable matrix of **int** values:

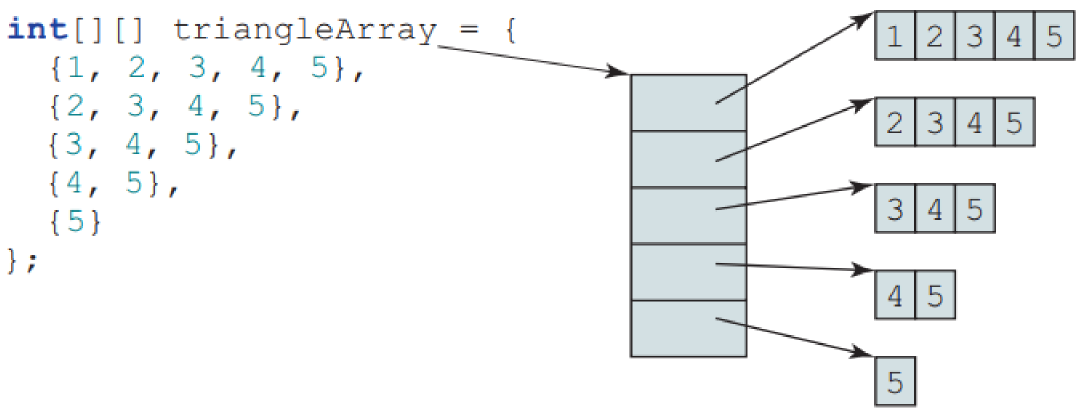
```
int[][] matrix;
```

You can create a two-dimensional array of 5-by-5 int values and assign it to matrix using this syntax:

```
matrix = new int[5][5];
```

A two-dimensional array is actually an array in which each element is a one-dimensional array. The length of an array **x** is the number of elements in the array, which can be obtained using *x.length*.
x[0], *x[1]*, . . . , and *x[x.length - 1]* are arrays. Their lengths can be obtained using *x[0].length*, *x[1].length*, . . . , and *x[x.length - 1].length*.

Each row in a two-dimensional array is itself an array. Thus, the rows can have different lengths. An array of this kind is known as a *ragged array*. Figure 12 shows an example of creating a ragged array.



```
int[][] triangleArray = {
  {1, 2, 3, 4, 5},
  {2, 3, 4, 5},
  {3, 4, 5},
  {4, 5},
  {5}
};
```

Figure 12: An example of creating a ragged array.

Nested for loops are often used to process a two-dimensional array.
You can pass a two-dimensional array to a method just as you pass a one-dimensional array. You can also return an array from a method.

Multidimensional Arrays

A two-dimensional array is an array of one-dimensional arrays, and a three-dimensional array is an array of two-dimensional arrays. The following syntax declares a three-dimensional array variable scores, creates an array, and assigns its reference to scores.

```
double[][][] scores = new double[6][5][4];
```

Array of Objects

An array can hold objects as well as primitive-type values. For example, the following statement declares and creates an array of 10 **Circle** objects:

```
Circle[] circleArray = new Circle[10];
```

An array of objects is actually an array of reference variables. Thus, invoking *circleArray[1].getArea()* involves two levels of referencing, as shown in Figure 13.

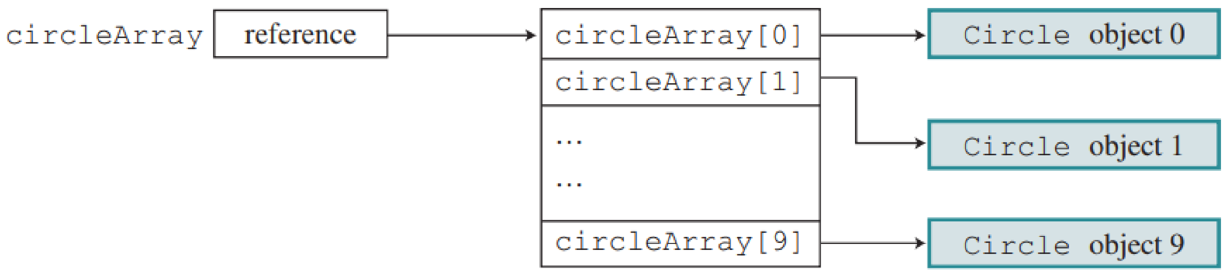


Figure 13: In an array of objects, an element of the array contains a reference to an object.

9.3 Activities

- **Activity 1:** Write a method that returns the sum of all the elements in a specified column in a matrix using the following header:

  ```
  public static double sumColumn(double[][] m, int columnIndex)
  ```

  Write a test program that reads a 3-by-4 matrix and displays the sum of each column. Here is a sample run:

  ```
  Enter a 3-by-4 matrix row by row:
  1.5 2 3 4
  5.5 6 7 8
  9.5 1 3 1
  Sum of the elements at column 0 is 16.5
  Sum of the elements at column 1 is 9.0
  Sum of the elements at column 2 is 13.0
  Sum of the elements at column 3 is 13.0
  ```

- **Activity 2:** Write a method to add two matrices. The header of the method is as follows:

  ```
  public static double[][] addMatrix(double[][] a, double[][] b)
  ```

  In order to be added, the two matrices must have the same dimensions and the same or compatible types of elements. For example, The following figure shows the results of adding two 3 * 3 matrices:

$$\begin{pmatrix} a_{11} & a_{12} & a_{13} \\ a_{21} & a_{22} & a_{23} \\ a_{31} & a_{32} & a_{33} \end{pmatrix} + \begin{pmatrix} b_{11} & b_{12} & b_{13} \\ b_{21} & b_{22} & b_{23} \\ b_{31} & b_{32} & b_{33} \end{pmatrix} = \begin{pmatrix} a_{11} + b_{11} & a_{12} + b_{12} & a_{13} + b_{13} \\ a_{21} + b_{21} & a_{22} + b_{22} & a_{23} + b_{23} \\ a_{31} + b_{31} & a_{32} + b_{32} & a_{33} + b_{33} \end{pmatrix}$$

  Write a test program that prompts the user to enter two 3 * 3 matrices and displays their sum.

- **Activity 3:** Write a program that randomly fills in 0s and 1s into a 4-by-4 matrix, prints the matrix, and finds the first row and column with the most 1s. Here is a sample run of the program:

```
0 0 1 1
1 0 1 1
0 1 0 1
1 0 1 0
The largest row index: 1
The largest column index: 2
```

- **Activity 4:** Design a class named **LinearEquation** for a 2 * 2 system of linear equations:

$$\begin{aligned} ax + by &= e \\ cx + dy &= f \end{aligned} \qquad x = \frac{ed - bf}{ad - bc} \qquad y = \frac{af - ec}{ad - bc}$$

  The class contains:

  - A private 2 * 3 array data field called *data* (*data[0,0]* = *a*, *data[0,1]* = *b*, *data[1,0]* = *c*, *data[1,1]* = *d*, *data[0,2]* = *e*, and *data[1,2]* = *f*).
  - A constructor with the argument of 2 * 3 array.
  - A method named *isSolvable()* that returns true if *ad* - *bc* is not 0.
  - Methods *getX()* and *getY()* that return the solution for the equation.

  Write a test program that prompts the user to enter the information of 4 linear equations and displays the solution of each equation.
  If *ad* - *bc* is 0, report that "`The equation has no solution.`"

10 Strings

10.1 Objectives

- To represent strings using the **String** object.
- To return the string length using the *length()* method.
- To return a character in the string using the *charAt(i)* method.
- To use the + operator to concatenate strings.
- To return an *uppercase* string or a *lowercase* string and to *trim* a string.
- To read strings from the console.
- To compare strings using the *equals* and the *compareTo* methods.
- To obtain substrings.
- To find a character or a substring in a string using the *indexOf* method.
- To use the **StringBuilder** class to process mutable strings.

10.2 Context

String is a predefined class in the Java library. The String type is not a primitive type. A **String** object is immutable; its contents cannot be changed.

Table 7 lists the **String** methods for obtaining string length, for accessing characters in the string, for concatenating string, for converting string to uppercases or lowercases, and for trimming a string.

Table 7: Simple Methods for **String** Objects.

Method	Description
length()	Returns the number of characters in this string.
charAt(index)	Returns the character at the specified index from this string.
concat(s1)	Returns a new string that concatenates this string with string s1.
toUpperCase()	Returns a new string with all letters in uppercase.
toLowerCase()	Returns a new string with all letters in lowercase.
trim()	Returns a new string with whitespace characters trimmed on both sides.

Reading a String from the Console

To read a string from the console, invoke the *next()* method on a **Scanner** object. For example, the following code reads a word string from the keyboard:

```
Scanner input = new Scanner(System.in);
System.out.print("Enter a word:  ");
String s = input.next();
System.out.println("Word is " + s);
```

The *next()* method reads a string that ends with a whitespace character. You can use the *nextLine()* method to read an entire line of text. The *nextLine()* method reads a string that ends with the `Enter` key pressed.

Comparing Strings

The **String** class contains the methods, as listed in Table 8, for comparing two strings.

Table 8: Comparison Methods for **String** Objects.

Method	Description
equals(s1)	Returns true if this string is equal to string s1.
equalsIgnoreCase(s1)	Returns true if this string is equal to string s1; it is case insensitive.
compareTo(s1)	Returns an integer greater than 0, equal to 0, or less than 0 to indicate whether this string is greater than, equal to, or less than s1.
compareToIgnoreCase(s1)	Same as compareTo except that the comparison is case insensitive.
startsWith(prefix)	Returns true if this string starts with the specified prefix.
endsWith(suffix)	Returns true if this string ends with the specified suffix.
contains(s1)	Returns true if s1 is a substring in this string.

Note: the == operator checks only whether 2 strings refer to the same object.

Obtaining Substrings

You can obtain a single character from a string using the *charAt* method. You can also obtain a substring from a string using the *substring* method in the **String** class, as given in Table 9.

Table 9: The **String** Class Contains the Methods for Obtaining Substring.

Method	Description
substring(beginIndex)	Returns this string's substring that begins with the character at the specified beginIndex and extends to the end of the string.
substring(beginIndex, endIndex)	Returns this string's substring that begins at the specified beginIndex and extends to the character at index endIndex - 1. Note the character at endIndex is not part of the substring.

Finding a Character or a Substring in a String

The **String** class provides several versions of *indexOf* and *lastIndexOf* methods to find a character or a substring in a string, as listed in Table 10.

Table 10: The **String** Class Contains the Methods for Finding Substrings.

Method	Description
indexOf(ch)	Returns the index of the first occurrence of ch in the string. Returns -1 if not matched.
indexOf(ch, fromIndex)	Returns the index of the first occurrence of ch after fromIndex in the string. Returns -1 if not matched.
indexOf(s)	Returns the index of the first occurrence of string s in this string. Returns -1 if not matched.
indexOf(s, fromIndex)	Returns the index of the first occurrence of string s in this string after fromIndex. Returns -1 if not matched.
lastIndexOf(ch)	Returns the index of the last occurrence of ch in the string. Returns -1 if not matched.
lastIndexOf(ch, fromIndex)	Returns the index of the last occurrence of ch before fromIndex in this string. Returns -1 if not matched.
lastIndexOf(s)	Returns the index of the last occurrence of string s. Returns -1 if not matched.
lastIndexOf(s, fromIndex)	Returns the index of the last occurrence of string s before fromIndex. Returns -1 if not matched.

Replacing and Splitting Strings

The String class provides the methods for replacing and splitting strings, as shown in Table 11.

Table 11: The **String** class contains the methods for replacing and splitting strings.

Method
replace(oldChar: char,newChar: char): String
replaceFirst(oldString: String, newString: String): String
replaceAll(oldString: String, newString: String): String
split(delimiter: String): String[]

Conversion between Strings and Arrays

Strings are not arrays, but a string can be converted into an array and vice versa. To convert a string into an array of characters, use the *toCharArray* method. To convert an array of characters into a string, use the *String(char[])* constructor or the *valueOf(char[])* method.

The StringBuilder Class

In general, the **StringBuilder** class can be used wherever a string is used. **StringBuilder** is more flexible than **String**. You can add, insert, or append new contents into **StringBuilder** objects, whereas the value of a **String** object is fixed once the string is created. You can create an empty string builder using *new StringBuilder()* or a string builder from a string using *new StringBuilder(String)*. Table 12 shows the functions to modify a string in the **StringBuilder**.

Table 12: The **StringBuilder** class contains the methods for modifying string builders.

Method
append(data: char[]): StringBuilder
append(v: aPrimitiveType): StringBuilder
append(s: String): StringBuilder
delete(startIndex: int, endIndex: int):StringBuilder
deleteCharAt(index: int): StringBuilder
insert(offset: int, data: char[]): StringBuilder
insert(offset: int, b: aPrimitiveType): StringBuilder
insert(offset: int, s: String): StringBuilder len: int): StringBuilder
replace(startIndex: int, endIndex: int, s: String): StringBuilder
reverse(): StringBuilder
setCharAt(index: int, ch: char): void

10.3 Activities

- **Activity 1:** Write a program that prompts the user to enter two strings, and reports whether the second string is a substring of the first string.

- **Activity 2:** Write a program that prompts the user to enter three cities and displays them in ascending order.

- **Activity 3:** Write a program that prompts the user to enter a binary number as a string (e.g. "`110011`") and display its corresponding decimal value (e.g. `51`).

- **Activity 4:** The **String** class is provided in the Java library. Provide your own implementation for the following methods (name the new class **MyString1**):
  - *public MyString1(char[] chars);*
  - *public char charAt(int index);*
  - *public int length();*
  - *public MyString1 substring(int begin, int end);*
  - *public MyString1 toLowerCase();*
  - *public boolean equals(MyString1 s);*
  - *public static MyString1 valueOf(int i);*

- **Activity 5:** The **StringBuilder** class is provided in the Java library. Provide your own implementation for the following methods (name the new class **MyStringBuilder1**):
  - *public MyStringBuilder1(String s);*
  - *public int length();*
  - *public char charAt(int index);*
  - *public String toString();*
  - *public MyStringBuilder1 append(MyStringBuilder1 s);*
  - *public MyStringBuilder1 append(int i);*
  - *public MyStringBuilder1 toLowerCase();*
  - *public MyStringBuilder1 substring(int begin, int end);*

11 Introduction to Exception Handling and Text I/O

11.1 Objectives

- To get an overview of exceptions and exception handling.
- To write a *try-catch* block to handle exceptions.
- To discover file/directory properties using the **File** class.
- To write data to a file using the **PrintWriter** class.
- To read data from a file using the **Scanner** class.

11.2 Context

Exception-Handling Overview

Exception handling enables a program to deal with runtime errors and continue its normal execution. Java enables a method to throw an exception that can be caught and handled by the caller. When an exception is thrown, the normal execution flow is interrupted. The statement for invoking the method is contained in a *try* block. The *try* block contains the code that is executed in normal circumstances. The exception is caught by the *catch* block. The code in the *catch* block is executed to handle the exception. Afterward, the statement after the *catch* block is executed. In summary, a template for a try-throw-catch block may look as follows:

```
try {
     Code to run;
     A statement or a method that may throw an exception;
     More code to run;
}
catch (type ex) {
     Code to process the exception;
}
```

The File Class

The **File** class contains the methods for obtaining the properties of a file/directory. The **File** class is intended to provide an abstraction that deals with most of the machine-dependent complexities of files and path names in a machine-independent fashion. However, the **File** class does not contain the methods for reading and writing file contents.

Caution: The directory separator for Windows is a backslash (\). The backslash is a special character in Java and should be written as (\\) in a string literal.

Writing Data Using PrintWriter

The **java.io.PrintWriter** class can be used to create a file and write data to a text file. First, you have to create a **PrintWriter** object for a text file as follows:

```
PrintWriter output = new PrintWriter(filename);
```

Then, you can invoke the *print*, *println*, and *printf* methods on the **PrintWriter** object to write data to a file.
The *close()* method must be used to close the opened file. If this method is not invoked, the data may not be saved properly in the file.

Reading Data Using Scanner

The **java.util.Scanner** class was used to read strings and primitive values from the console. To read from a file, create a **Scanner** for a file, as follows:

```
Scanner input = new Scanner(new File(filename));
```

Invoking the constructor *new Scanner(File)* may throw an *I/O exception*, so the main method declares *throws Exception*.

11.3 Activities

- **Activity 1:** Write a program that meets the following requirements using exception handling technique:
  - Creates an array with 100 randomly chosen integers.
  - Prompts the user to enter the index of the array, then displays the corresponding element value. If the specified index is out of bounds, display the message "*Out of Bounds*".

- **Activity 2:** Write a program that will count the number of characters, words, and lines in a file. Words are separated by whitespace characters.

- **Activity 3:** Suppose a text file contains an unspecified number of scores separated by spaces. Write a program that prompts the user to enter the file, reads the scores from the file, and displays their total and average.

- **Activity 4:** Write a program that reads the strings from a file contains names and reports whether the names in the files are stored in increasing order. If the names are not sorted in the file, it displays the first two names that are out of the order.

- **Activity 5:** Write a program that prompts the user to enter a file name and displays the occurrences of each letter in the file. Letters are case insensitive.

12 Object-Oriented Thinking

12.1 Objectives

- To apply class abstraction to develop software.
- To discover the relationships between classes.
- To create objects for primitive values using the wrapper classes.

12.2 Context

Class Abstraction and Encapsulation

Class abstraction is separation of class implementation from the use of a class. The details of implementation are encapsulated and hidden from the user. This is known as class encapsulation.

Thinking in Objects

The procedural paradigm focuses on designing methods. The object-oriented paradigm couples data and methods together into objects. Software design using the object-oriented paradigm focuses on objects and operations on objects.

Class Relationships

The common relationships among classes are association, aggregation, composition, and inheritance. The following are the list of class relationships to be covered in this lab:

- **Association**: Association is a general binary relationship that describes an activity between two classes.
- **Aggregation**: Aggregation models has-a relationships. The owner object is called an aggregating object, and its class is called an aggregating class. The subject object is called an aggregated object, and its class is called an aggregated class.
- **Composition**: We refer aggregation between two objects as composition if the existence of the aggregated object is dependent on the aggregating object. In other words, if a relationship is composition, the aggregated object cannot exist on its own.

Wrapper Classes

A primitive-type value is not an object, but it can be wrapped in an object using a wrapper class in the Java API. Java provides **Boolean**, **Character**, **Double**, **Float**, **Byte**, **Short**, **Integer**, and **Long** wrapper classes in the **java.lang** package for primitive data types. The instances of all wrapper classes are immutable; this means that, once the objects are created, their internal values cannot be changed.

The BigInteger and BigDecimal Classes

The **BigInteger** and **BigDecimal** classes can be used to represent integers or decimal numbers of any size and precision. Both are immutable. An instance of **BigInteger** can represent an integer of any size.

12.3 Activities

- **Activity 1:** Implement a class named **Time** for encapsulating a time. The class contains the following:
  - A data field of the **BigInteger** *time* that stores the elapsed time in milliseconds since midnight, `Jan 1, 1970`.
  - A no-arg constructor that constructs a **Time** for the current system time.
  - A constructor with the specified time string to create a **Time**.
    A time string format is "`yyyy:mm:dd-hh:mm:ss`"
    such as "`2022:5:13-14:40:20`".
  - A constructor with the specified elapsed time in seconds since midnight, `Jan 1, 1970`.
  - The *getHour()* method that returns the current hour in the range `0-23`.
  - The *getMinute()* method that returns the current minute in the range `0-59`.
  - The *getSecond()* method that returns the current second in the range `0-59`.
  - The *getSeconds()* method that returns the elapsed total seconds.
  - The *toString()* method that returns a string time
    such as "`2022:5:13-14:40:20`".

  Write a Driver class to test **Time** class.

- **Activity 2:** Design two classes: **Flight**[1] and **Itinerary**[2]. The **Flight** class stores the information about a flight with the following members:
  - A data field named *flightNo* of the **String** type with getter method.
  - A data field named *departureTime* of the **Time** type (The one created in **Activity 1**) with getter and setter methods.
  - A data field named *arrivalTime* of the **Time** type with getter and setter methods.
  - A constructor that creates a **Flight** with the specified number, *departureTime*, and *arrivalTime*.
  - A method named *getFlightTime()* that returns the flight time in minutes.

  The **Itinerary** class stores the information about itinerary with the following members:
  - A data field named *flights* of **Flight**[] type. The array contains the flights for the itinerary.
  - A constructor that creates an **Itinerary** with the specified flights.
  - A method named *getTotalTime()* that returns the total travel time in minutes from the departure time of the first flight to the arrival time of the last flight in the itinerary.

  Implement these two classes and a **Driver** class to test these classes.

[1]Flight: a trip made by or in an airplane or spacecraft
[2]Itinerary: the route of a journey, which might consists of several flights