ISL ENGINEERING COLLEGE
Approved By AICTE, Affiliated to Osmania University

International Airport Road, Chandrayangutta, Bandlaguda, Vyasapuri, Hyderabad - 500 005.

UNIT – I

Introduction

Classes and Objects

String Handling

UNDERSTANDING JAVA'S OBJECT-ORIENTED DEVELOPMENT

- ✓ Object Oriented Programming is a programming concept that works on the principle that objects are the most important part of your program.
- ✓ It allows users create the objects that they want and then create methods to handle those objects. Manipulating these objects to get results is the goal of Object Oriented Programming.
- ✓ Object Oriented Programming popularly known as OOP, is used in a modern programming language like Java. Java is object-oriented.
- ✓ Object-oriented languages are better than "Do this/Do that" languages because they organize data in a way that lets people do all kinds of things with it.
- ✓ To modify the data, you can build on what you already have, rather than scrap everything you've done and start over each time you need to do something new.
- ✓ Although computer programmers are generally smart people, they took a while to figure this out.

OBJECTS AND THEIR CLASSES

- ✓ In an object-oriented language, you use objects and classes to organize your data. Imagine that you're writing a computer program to keep track of the houses in a new condominium development.
- ✓ The houses differ only slightly from one another. Each house has a distinctive siding color, an indoor paint color, a kitchen cabinet style, and so on. In your object-oriented computer program, each house is an object.
- ✓ But objects aren't the whole story. Although the houses differ slightly from one another, all the houses share the same list of characteristics. For instance, each house has a characteristic known as siding color. Each house has another characteristic known as kitchen cabinet style. In your object-oriented program, you need a master list containing all the characteristics that a house object can possess. This master list of characteristics is called a class. So there you have it.
- ✓ Object-oriented programming is misnamed. It should really be called "programming with classes and objects."
- ✓ Notice that the word classes were listed first? Think again about a housing development that's under construction. Somewhere on the lot, in a rickety trailer parked on bare dirt, is a master list of characteristics known as a blueprint. An architect's blueprint is like an object-oriented programmer's class.
- ✓ A blueprint is a list of characteristics that each house will have. The blueprint says, "siding." The actual house object has gray siding. The blueprint says, "kitchen cabinet."

- ✓ The actual house object has Louis XIV kitchen cabinets. A year after you create the blueprint, you use it to build ten houses. It's the same with classes and objects. First, the programmer writes code to describe a class. Then when the program runs, the computer creates objects from the class.

Core OOPS concepts are

1. **Class:** The class is a group of similar entities. It is only a logical component and not the physical entity. For example, if you had a class called "Expensive Cars" it could have objects like Mercedes, BMW, Toyota, etc. Its properties (data) can be price or speed of these cars. While the methods may be performed with these cars are driving, reverse, braking etc.
2. **Object:** An object can be defined as an instance of a class, and there can be multiple instances of a class in a program. An Object contains both the data and the function, which operates on the data. For example - chair, bike, marker, pen, table, car, etc.
3. **Inheritance:** Inheritance is an OOPS concept in which one object acquires the properties and behaviors of the parent object. It's creating a parent-child relationship between two classes. It offers robust and natural mechanism for organizing and structure of any software.
4. **Polymorphism:** Polymorphism refers to the ability of a variable, object or function to take on multiple forms. For example, in English, the verb "run" has a different meaning if you use it with "a laptop," "a foot race" and "a business". Here, we understand the meaning of "run" based on the other words used along with it. The same also applied to Polymorphism.
5. **Abstraction:** An abstraction is an act of representing essential features without including background details. It is a technique of creating a new data type that is suited for a specific application. For example, while driving a car, you do not have to be concerned with its internal working. Here you just need to concern about parts like steering wheel, Gears, accelerator, etc.
6. **Encapsulation:** Encapsulation is an OOP technique of wrapping the data and code. In this OOPS concept, the variables of a class are always hidden from other classes. It can only be accessed using the methods of their current class. For example - in school, a student cannot exist without a class.
7. **Association:** Association is a relationship between two objects. It defines the diversity between objects. In this OOP concept, all object have their separate lifecycle, and there is no owner. For example, many students can associate with one teacher while one student can also associate with multiple teachers.
8. **Aggregation:** In this technique, all objects have their separate lifecycle. However, there is ownership such that child object can't belong to another parent object. For example consider class/objects department and teacher. Here, a single teacher can't belong to

multiple departments, but even if we delete the department, the teacher object will never be destroyed.

9. **Composition:** A composition is a specialized form of Aggregation. It is also called "death" relationship. Child objects do not have their lifecycle so when parent object deletes all child object will also delete automatically. For that, let's take an example of House and rooms. Any house can have several rooms. One room can't become part of two different houses. So, if you delete the house room will also be deleted.

Advantages of OOPS:

1. OOP offers easy to understand and a clear modular structure for programs.
2. Objects created for Object-Oriented Programs can be reused in other programs. Thus it saves significant development cost.
3. Large programs are difficult to write, but if the development and designing team follow OOPS concept then they can better design with minimum flaws.
4. It also enhances program modularity because every object exists independently.

Structured Programming	Object Oriented Programming
Structured Programming is designed which focuses on **process**/ logical structure and then data required for that process.	Object Oriented Programming is designed which focuses on **data**.
Structured programming follows **top-down approach**.	Object oriented programming follows **bottom-up approach**.
Structured Programming is also known as **Modular Programming** and a subset of **procedural programming language**.	Object Oriented Programming supports **inheritance, encapsulation, abstraction**, **polymorphism**, etc.
In Structured Programming, Programs are divided into small self-contained **functions**.	In Object Oriented Programming, Programs are divided into small entities called **objects**.
Structured Programming is **less** secure as there is no way of **data hiding**.	Object Oriented Programming is more secure as having data hiding feature.
Structured Programming can solve **moderately** complex programs.	Object Oriented Programming can solve any **complex** programs.
Structured Programming provides **less reusability**, more function dependency.	Object Oriented Programming provides more reusability, less function **dependency**.
Less abstraction and less flexibility.	More abstraction and more **flexibility**.

The Benefits of OOP

OOP offers several benefits to the program designer and the user. Object-orientation contributes to the solutions of many problem associated with the development and quality of software products. The new technology promises greater programmer productivity, better quality of software and lesser maintenance cost. The principal advantages are:

1. Through inheritance, we can eliminate redundant code and extend the use of existing classes.
2. We can build programs from standard working modules that communicate with one another rather than, having to start writing the code from scratch. This leads to saving of development time and higher productivity.
3. The principle of data hiding helps the programmers to build secure program that can't be invaded by code in other parts of the program.
4. It is possible to have multiple objects to coexist without any interference.
5. It is possible to map objects in the problem domain to those objects in the program.
6. It is easy to partition the work in a project based on objects.
7. The data-centered design approach enables us to capture more details of the model in an implementable form.
8. Object-oriented systems can be easily upgraded from small to large system
9. Message passing technique for communication between objects makes the interface descriptions with external system much simpler.
10. Software complexity can be easily managed.

History of JAVA Programming

- Java was conceived by **James Gosling, Patrick Naughton, Chris Warth, Ed Frank, and Mike Sheridan** at Sun Microsystems, Inc. in 1991.
- It took 18 months to develop the first working version.
- This language was initially called "Oak," but was renamed "Java" in 1995.
- Between the initial implementation of Oak in 1992 and the public announcement of Java in 1995, many people contributed to the design and evolution of the language.
- Bill Joy, Arthur van Hoff, Jonathan Payne, Frank Yellin, and Tim Lindholm were the key contributors.
- Much of Java is inherited from two languages: C & C+. From C, Java derives its syntax and from C++ it took object-oriented features.

- JAVA was designed primarily with a motivation to develop a platform independent (Architecture – neutral) language that could be used to create software for many consumer electronics like microwave ovens and remote controls.
- But later with the invention of World Wide Web (internet) the language was adapted to implement web applications, where the size and portability of the content matters.

JAVA Buzzwords:

Java buzzwords are the key considerations for adopting the JAVA language.

1. Simple
2. Secure
3. Portable
4. Object-oriented
5. Robust
6. Multithreaded
7. Architecture-neutral
8. Interpreted
9. High performance
10. Distributed
11. Dynamic`

1. Simple:

- ✓ Java was designed to be easy for the professional programmer to learn and use effectively.
- ✓ As many of the JAVA concepts are inherited from C & C++. Having the prior knowledge of these two languages will make our learning job easier.
- ✓ One will not find Java hard to learn. Moving to Java will require very little effort.

2. Secure:

- ✓ When we download a “normal” program, we are taking a risk, because the code we are downloading might contain a virus, Trojan horse, or other harmful code.
- ✓ Malicious code can cause damage, because it can gain unauthorized access to system resources. For example, a virus program might gather private information, such as credit

card numbers, bank account balances, and passwords, by searching the contents of any computer local file system.

- ✓ Java achieved this protection by limiting an applet to the Java execution environment and not allowing it access to other parts of the computer.

3. Portable

- ✓ Portability is a major aspect of the Internet because there are many different types of computers and operating systems connected to it.
- ✓ A Java program can run on any computer connected to the Internet, there needed to be some way to enable that program to execute on different systems. The Java Virtual Machine is the solution for providing the portability.

4. Object Oriented

- ✓ Java manages to maintain a balance between the "everything is an object" paradigm and the non-object model.
- ✓ The object model in Java is simple and easy to extend, while primitive types, such as integers, are kept as high-performance non-objects.

5. Robust

- ✓ The multi-platform environment of the Web increases the demands on a robust program, because the program must execute reliably in a variety of systems.
- ✓ Thus, the ability to create robust programs was given a high priority.
- ✓ Java checks the code at compile time and also checks it at run time.
- ✓ Many hard-to-track-down bugs are simply impossible to create in Java.
- ✓ Knowing that what you have written will behave in a predictable way under diverse conditions is a key feature of Java.

6. Multi-Threaded

- ✓ Java was designed to meet the real-world requirement of creating interactive, networked programs.
- ✓ To accomplish this, Java supports multithreaded programming, which allows us to write programs that do many things simultaneously.

7. Architecture-Neutral

- ✓ Programmers face a problem that if they write a program today, there is no guarantee that it will run tomorrow—even on the same machine.
- ✓ Operating system upgrades, processor upgrades, and changes in core system resources can make a program malfunction.
- ✓ To solve this issue, the Java designers made several hard decisions in the Java language and the Java Virtual Machine.
- ✓ Their goal was "write once; run anywhere, anytime, forever." To a great extent, this goal was accomplished.

8 & 9. Interpreted and High Performance

- ✓ Java enables the creation of cross-platform programs by compiling into an intermediate representation called Java bytecode.
- ✓ This code can be executed on any system that implements the Java Virtual Machine.
- ✓ Many previous attempts were made to get cross-platform solutions, but resulted degrade in performance.
- ✓ Java bytecode was designed very carefully so that it can easily translate directly into native machine code with very high performance by using a just-in-time compiler.

10. Distributed

- ✓ Java is designed for the distributed environment of the Internet.
- ✓ It supports TCP/IP protocols for accessing a resource using a URL and Remote Method Invocation (RMI) to invoke methods across a network.

11. Dynamic

- ✓ Java programs carry substantial amounts of run-time information with them, which is used to verify and resolve accesses to objects at run time.
- ✓ This enables a possibility to dynamically link code in a safe manner.

DATA TYPES:

Java defines eight primitive types of data: byte, short, int, long, char, float, double, and Boolean. These can be put in four groups:

- **Integers:** This group includes byte, short, int, and long, which are for whole-valued signed numbers.
- **Floating-point numbers:** This group includes float and double, which represent numbers with fractional precision.
- **Characters:** This group includes char, which represents symbols in a character set, like letters and numbers.
- **Boolean:** This group includes Boolean, which is a special type for representing true/false values.

Integers:

- ✓ Java defines four integer types: byte, short, int, and long.
- ✓ All of these are signed, positive and negative values.
- ✓ Java does not support unsigned, positive-only integers.

Name	Width in Bits	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

Floating-point numbers:

- ✓ Floating-point numbers, also known as real numbers, are used when evaluating expressions that require fractional precision. For example, calculations such as square root.
- ✓ There are two kinds of floating-point types, float and double, which represent single- and double-precision numbers, respectively.

Name	Width in Bits	Approximate Range
double	64	4.9e–324 to 1.8e+308
float	32	1.4e–045 to 3.4e+038

Characters

- ✓ In Java, the data type used to store characters is char.
- ✓ Java uses Unicode to represent characters. Unicode defines a fully international character set that can represent all of the characters found in all human languages.
- ✓ It is a unification of dozens of character sets, such as Latin, Greek, Arabic, Cyrillic, Hebrew, Katakana, Hangul, and many more.
- ✓ For this purpose, it requires 16 bits. Thus, in Java char is a 16-bit type.
- ✓ The range of a char is 0 to 65,536. There are no negative chars.
- ✓ The standard set of characters known as ASCII still ranges from 0 to 127 as always, and the extended 8-bit character set, ISO-Latin-1, ranges from 0 to 255.

Boolean:

- ✓ Java has a primitive type, called Boolean, for logical values.
- ✓ It can have only one of two possible values, true or false.
- ✓ This is the type returned by all relational operators and conditional expressions.

LITERALS

Integer literals

- ✓ Integers are probably the most commonly used type in the typical program. Any whole number value is an integer literal.
  Examples are 1, 2, 3, and 42.
- ✓ These are all decimal values, meaning they are describing a base 10 number.

Floating-Point Literals

- ✓ Floating-point numbers represent decimal values with a fractional component.
  Examples include 6.022E23, 314159E–05 and 2e+100.

Boolean Literals

- ✓ Boolean literals are simple. There are only two logical values that a Boolean value can have, true and false.
- ✓ The values of true and false do not convert into any numerical representation.
- ✓ The true literal in Java does not equal 1, nor does the false literal equal 0.

Character Literals

- ✓ Characters in Java are indices into the Unicode character set.
- ✓ They are 16-bit values that can be converted into integers and manipulated with the integer operators, such as the addition and subtraction operators.
- ✓ Aliteral character is represented inside a pair of single quotes. All of the visible ASCII characters can be directly entered inside the quotes, such as 'a', 'z', and '@'.

String Literals

String literals in Java are specified like they are in most other languages—by enclosing a sequence of characters between a pair of double quotes. Examples of string literals are

"Hello World"

"two\nlines"

"\"This is in quotes\""

VARIABLES

- ✓ The variable is the basic unit of storage in a Java program.
- ✓ A variable is defined by the combination of an identifier, a type, and an optional initializer.
- ✓ In addition, all variables have a scope, which defines their visibility, and a lifetime.

Declaring a Variable

- ✓ In Java, all variables must be declared before they can be used.
- ✓ The basic form of a variable declaration is shown here:

type identifier [= value][, identifier [= value] ...] ;

Dynamic Initialization

- ✓ Java allows variables to be initialized dynamically, using any expression valid at the time the variable is declared.

Example:

```
class DynInit {
        public static void main(String args[])
        {
                int a = 3, b = 4;
                // c is dynamically initialized
```

```
                int c = a * b);
                System.out.println("Hypotenuse is " + c);
        }
}
```

The Scope and Lifetime of Variables

- ✓ Java allows variables to be declared within any block.
- ✓ A block is begun with an opening curly brace and ended by a closing curly brace.
- ✓ A block defines a scope. Thus, each time you start a new block, you are creating a new scope.
- ✓ Ascope determines what objects are visible to other parts of your program. It also determines the lifetime of those objects.
- ✓ In Java, the two major scopes are those defined by a class and those defined by a method.
- ✓ The scope defined by a method begins with its opening curly brace. If that method has parameters, they too are included within the method's scope.
- ✓ Variables declared inside a scope are not visible/accessible to code that is defined outside that scope.
- ✓ When we declare a variable within a scope, we are localizing that variable and protecting it from unauthorized access and/or modification.
- ✓ Scopes can be nested. For example, each time we create a block of code, we are creating a new, nested scope. When this occurs, the outer scope encloses the inner scope. This means that objects declared in the outer scope will be visible to code within the inner scope. The reverse is not true. Objects declared within the inner scope will not be visible outside it.

Example:

```
class Scope {
    public static void main(String args[])
    {
            int x;
            x = 10;
            if(x == 10)
```

```
            {
                int y = 20;
                System.out.println("x and y: " + x + " " + y);
                x = y * 2;
            }
            System.out.println("x is " + x);
        }
    }
```

TYPE CONVERSION AND CASTING

- ✓ It is fairly common to assign a value of one type to a variable of another type. If the two types are compatible, then Java will perform the conversion automatically. For example, it is always possible to assign an int value to a long variable.
- ✓ Not all types are compatible, so not all type conversions are implicitly allowed. For instance, there is no automatic conversion defined from double to byte.
- ✓ Fortunately, it is still possible to obtain a conversion between incompatible types. To do so, you must use a cast, which performs an explicit conversion between incompatible types.

Java's Automatic Conversions

- ✓ When one type of data is assigned to another type of variable, an automatic type conversion will take place if the following two conditions are met:
    i. The two types are compatible.
    ii. The destination type is larger than the source type.
- ✓ When these two conditions are met, a widening conversion takes place.
  For example: the **int** type is always large enough to hold all valid **byte** values, so no explicit cast statement is required.
- ✓ The numeric types, including **integer** and **floating-point** types, are compatible with each other.
- ✓ There are no automatic conversions from the numeric types to **char** or **boolean**.
- ✓ **char** and **boolean** are not compatible with each other.

- ✓ Java also performs an automatic type conversion when storing a literal integer constant into variables of type byte, short, long, or char.

Casting Incompatible Types

- ✓ To create a conversion between two incompatible types, you must use a cast. A cast is simply an explicit type conversion.
- ✓ It has the following general form: (target-type) value

  For example: if we assign an **int** value to a **byte** variable. This conversion will not be performed automatically, because a **byte** is smaller than an **int**. This kind of conversion is called a narrowing conversion.

  int a;

  byte b;

  b = (byte) a;
- ✓ A different type of conversion will occur when a floating-point value is assigned to an integer type: truncation.
- ✓ **For example:** if the value 1.23 is assigned to an integer, the resulting value will simply be 1. The 0.23 will have been truncated.

Automatic Type Promotion in Expressions

- ✓ In addition to assignments in expressions type conversions may occur automatically.
- ✓ For example: Examine the following expression:

  byte a = 40;

  byte b = 50;

  byte c = 100;

  int d = a * b / c;
- ✓ The result of the intermediate term a * b easily exceeds the range of either of its byte operands. To handle this kind of problem, Java automatically promotes each byte to int when evaluating an expression.

The Type Promotion Rules

Java defines several type promotion rules that apply to expressions.

They are as follows:

- ✓ First, all byte, short, and char values are promoted to int, as just described.
- ✓ Then, if one operand is a long, the whole expression is promoted to long.
- ✓ If one operand is a float, the entire expression is promoted to float.
- ✓ If any of the operands is double, the result is double.

Example: The following program demonstrates how each value in the expression gets promoted to match the second argument to each binary operator:

```
class Promote {
        public static void main(String args[])
        {
                byte b = 42;
                char c = 'a';
                short s = 1024;
                int i = 50000;
                float f = 5.67f;
                double d = .1234;
                double result = (f * b) + (i / c) - (d * s);
                System.out.println((f * b) + " + " + (i / c) + " - " + (d * s));
                System.out.println("result = " + result);
        }
}
```

COMMENTS:

- ✓ Comments are used for documenting the code for understanding.
- ✓ There are three types of comments defined by Java.
    1. Single-line Comment

// Single Line Comment

2. Multiline Comment

/* Line – 1

Line – 2

Line – 3 Multiline Comments */

3. Documentation comment

/** Documentation Comment

Documentation Comment */

SEPARATORS:

- ✓ In Java, there are a few characters that are used as separators.
- ✓ The most commonly used separator in Java is the semicolon.

SYMBOL	NAME	PURPOSE
( )	Parentheses	Used to contain lists of parameters in method definition and invocation. Also used for defining precedence in expressions, containing expressions in control statements, and surrounding cast types.
{ }	Braces	Used to contain the values of automatically initialized arrays. Also used to define a block of code, for classes, methods, and local scopes.
[ ]	Brackets	Used to declare array types. Also used when dereferencing array values.
;	Semicolon	Terminates statements.
,	Comma	Separates consecutive identifiers in a variable declaration. Also used to chain statements together inside a **for** statement.
.	Period	Used to separate package names from subpackages and classes. Also used to separate a variable or method from a reference variable.

ARRAYS

- ✓ An array is a group of like-typed variables that are referred to by a common name.
- ✓ Arrays of any type can be created and may have one or more dimensions.
- ✓ A specific element in an array is accessed by its index.
- ✓ Arrays offer a convenient means of grouping related information.

One-Dimensional Arrays

- ✓ To create an array, you first must create an array variable of the desired type.
- ✓ The general form of a one-dimensional array declaration is

  type var-name[ ];
- ✓ Here, type declares the base type of the array.
- ✓ **For example:** the following declares an array named month_days with the type "array of int":

  int month_days[];
- ✓ After this declaration month_days is an array variable with value set to null, which represents an array with no value.
- ✓ To link month_days with an actual, physical array of integers, we must allocate one using **new** and assign it to month_days.
- ✓ **new** is a special operator that allocates memory.
- ✓ The general form of **new** to one-dimensional arrays:

  array-var = new type[size];
- ✓ *type* specifies the type of data being allocated, *size* specifies the number of elements in the array, and *array-var* is the array variable that is linked to the array.
- ✓ The elements in the array allocated by new will automatically be initialized to zero.
- ✓ This following example allocates a 12-element array of integers and links them to month_days.

  month_days = new int[12];
- ✓ After this statement executes, **month_days** will refer to an array of 12 integers. Further, all elements in the array will be initialized to zero.
- ✓ Obtaining an array is a two-step process.

- ✓ First, you must declare a variable of the desired array type.
- ✓ Second, you must allocate the memory that will hold the array, using new, and assign it to the array variable.
- ✓ Thus, in Java all arrays are dynamically allocated.

Accessing Array Elements

- ✓ Once you have allocated an array, you can access a specific element in the array by specifying its index within square brackets.
- ✓ All array indexes start at zero.
  - ✓ **For example:** to assigns the value 28 to the second element of **month_days.** month_days[1] = 28;
- ✓ One can combine the declaration of the array variable with the allocation of the array itself, int month_days[] = new int[12];
- ✓ If we do the declaration, definition & Initialization all together there is no need to use the new keyword to allocate the memory.
- ✓ **For example:** to store the number of days in each month, the following code creates an initialized array of integers:

```
class AutoArray {
        public static void main(String args[])
        {
                int month_days[] = { 31, 28, 31, 30, 31, 30, 31, 31, 30, 31,30, 31 };
                System.out.println("April has " + month_days[3] + " days.");
        }
}
```

Multidimensional Arrays

- ✓ In Java, multidimensional arrays are actually arrays of arrays.
- ✓ To declare a multidimensional array variable, specify each additional index using another set of square brackets.

- ✓ **For example:** to declare a two-dimensional array variable called twoD.

  int twoD[][] = new int[4][5];

  This allocates a 4 by 5 array and assigns it to twoD.

```
class TwoDArray {
    public static void main(String args[]) {
        int twoD[][]= new int[4][5];
        int i, j, k = 0;
        for(i=0; i<4; i++)
        for(j=0; j<5; j++)
        {
                twoD[i][j] = k;
                k++;
        }
        for(i=0; i<4; i++)
        {
                for(j=0; j<5; j++)
                        System.out.print(twoD[i][j] + " ");
                        System.out.println();
        }
        }
}
```

- ✓ To allocate memory for a multidimensional array, we need to specify the memory only for the first (leftmost) dimension. We can allocate the remaining dimensions separately.
- ✓ For example, to allocate memory for the first dimension of twoD. It allocates the second dimension manually.

  int twoD[][] = new int[4][];

  twoD[0] = new int[5];

  twoD[1] = new int[5];

  twoD[2] = new int[5];

  twoD[3] = new int[5];

- ✓ (Ragged Array)**For example:** to create a two-dimensional array in which the sizes of the second dimension are unequal.

  int twoD[][] = new int[4][];

```
twoD[0] = new int[1];
twoD[1] = new int[2];
twoD[2] = new int[3];
twoD[3] = new int[4];
```

Example:

```
class TwoDAgain {
    public static void main(String args[]) {
        int twoD[][] = new int[4][];
        twoD[0] = new int[1];
        twoD[1] = new int[2];
        twoD[2] = new int[3];
        twoD[3] = new int[4];
        int i, j, k = 0;
        for(i=0; i<4; i++)
            for(j=0; j<i+1; j++) {
                twoD[i][j] = k;
                k++;
            }
            for(i=0; i<4; i++) {
                for(j=0; j<i+1; j++)
                    System.out.print(twoD[i][j] + " ");
                    System.out.println();
            }
        }
    }
```

Output:

```
0
1 2
3 4 5
6 7 8 9
```

Three-dimensional array

- ✓ Let's look at one more example that uses a multidimensional array.
- ✓ The following program creates a 3 by 4 by 5, three-dimensional array.

Example: // Demonstrate a three-dimensional array.

```
class ThreeDMatrix {
        public static void main(String args[]) {
                int threeD[][][] = new int[3][4][5];
                int i, j, k;
                for(i=0; i<3; i++)
                        for(j=0; j<4; j++)
                                for(k=0; k<5; k++)
                                        threeD[i][j][k] = i * j * k;
                for(i=0; i<3; i++) {
                        for(j=0; j<4; j++) {
                                for(k=0; k<5; k++)
                                        System.out.print(threeD[i][j][k] + " ");
                System.out.println();
                }
                System.out.println();
                }
        }
}
```

Output

```
0 0 0 0 0
0 0 0 0 0
0 0 0 0 0
0 0 0 0 0
0 0 0 0 0
0 1 2 3 4
0 2 4 6 8
0 3 6 9 12
0 0 0 0 0
0 2 4 6 8
0 4 8 12 16
0 6 12 18 24
```

Alternative Array Declaration Syntax

- ✓ There is a second form that may be used to declare an array: type[ ] var-name;
- ✓ The square brackets follow the type specifier, and not the name of the array variable.

  int[] a2 = new int[3]; (SAME) int al[] = new int[3];

  char twod1[][] = new char[3][4]; (SAME) char[][] twod2 = new char[3][4];
- ✓ This alternative declaration is a convenience, when declaring several arrays at the same time.

  int[] nums, nums2, nums3; (SAME) int nums[], nums2[], nums3[];

OPERATORS

- ✓ Java provides a rich operator environment. Most of its operators can be divided into the following four groups: arithmetic, bitwise, relational, and logical.
- ✓ Java also defines some additional operators that handle certain special situations.

Arithmetic Operators

- ✓ Arithmetic operators are used in mathematical expressions in the same way that they are used in algebra.
- ✓ The following table lists the arithmetic operators:

Operator	Results
+	Addition
-	Subtraction (also unary minus)
*	Multiplication
/	Division
%	Modulus
++	Increment
+=	Addition assignment
-=	Subtraction assignment
*=	Multiplication assignment
/=	Division assignment
%=	Modulus assignment
--	Decrement

- ✓ The operands of the arithmetic operators must be of a numeric type.

- ✓ You cannot use them on Boolean types, but you can use them on char types, since the char type in Java is, essentially, a subset of int.

The Basic Arithmetic Operators

- ✓ The basic arithmetic operations—addition, subtraction, multiplication, and division— all behave as one would expect for all numeric types.
- ✓ The minus operator also has a unary form that negates its single operand.
- ✓ Remember that when the division operator is applied to an integer type, there will be no fractional component attached to the result.
- ✓ The following simple example program demonstrates the arithmetic operators.

Example: // Demonstrate the basic arithmetic operators.

```
class BasicMath {
public static void main(String args[]) {
// arithmetic using integers
        System.out.println("Integer Arithmetic");
        int a = 1 + 1;
        int b = a * 3;
        int c = b / 4;
        int d = c - a;
        int e = -d;
        System.out.println("a = " + a);
        System.out.println("b = " + b);
        System.out.println("c = " + c);
        System.out.println("d = " + d);
        System.out.println("e = " + e);
        // arithmetic using doubles
        System.out.println("\nFloating Point Arithmetic");
        double da = 1 + 1;
        double db = da * 3;
        double dc = db / 4;
        double dd = dc - a;
        double de = -dd;
        System.out.println("da = " + da);
        System.out.println("db = " + db);
        System.out.println("dc = " + dc);
```

```
        System.out.println("dd = " + dd);
        System.out.println("de = " + de);
        }
}
```

Output

```
Integer Arithmetic
a = 2
b = 6
c = 1
d = -1
e = 1
Floating Point Arithmetic
da = 2.0
db = 6.0
dc = 1.5
dd = -0.5
de = 0.5
```

The Modulus Operator

- ✓ The modulus operator, %, returns the remainder of a division operation.
- ✓ It can be applied to floating-point types as well as integer types.

Example: // Demonstrate the % operator.

```
class Modulus {
        public static void main(String args[]) {
        int x = 42;
        double y = 42.25;
        System.out.println("x mod 10 = " + x % 10);
        System.out.println("y mod 10 = " + y % 10);
        }
}
```

Output:

```
x mod 10 = 2
y mod 10 = 2.25
```

Arithmetic Compound Assignment Operators

- ✓ Java provides special operators that can be used to combine an arithmetic operation with an assignment.

  a = a + 4; <same> a+=4;

  a = a % 2; <same> a%=2;

- ✓ There are compound assignment operators for all of the arithmetic, binary operators. Thus, any statement of the form

  var = var op expression; <same> var op= expression;

- ✓ The compound assignment operators provide two benefits.
  - First, they save you a bit of typing, because they are “shorthand” for their equivalent long forms.
  - Second, they are implemented more efficiently by the Java run-time system than are their equivalent long forms.

Example: // Demonstrate several assignment operators.

```
class OpEquals {
public static void main(String args[]) {
        int a = 1;
        int b = 2;
        int c = 3;
        a += 5;
        b *= 4;
        c += a * b;
        c %= 6;
        System.out.println("a = " + a);
        System.out.println("b = " + b);
        System.out.println("c = " + c);
        }
}
```

Output:

```
a = 6
b = 8
c = 3
```

The Bitwise Operators

- ✓ Java defines several bitwise operators that can be applied to the integer types, long, int, short, char, and byte.
- ✓ These operators act upon the individual bits of their operands.

Operator	Result
~	Bitwise unary NOT
&	Bitwise AND
|	Bitwise OR
^	Bitwise exclusive OR
>>	Shift right
<<	Shift left
&=	Bitwise AND assignment
|=	Bitwise OR assignment
^=	Bitwise exclusive OR assignment
>>=	Shift right assignment
<<=	Shift left assignment

Representing Signed Integers in JAVA

- ✓ All of the integer types (except char) are signed integers. This means that they can represent negative values as well as positive ones.
- ✓ Java uses an encoding known as two's complement, which means that negative numbers are represented by inverting (changing 1's to 0's and vice versa) all of the bits in a value, then adding 1 to the result.
- ✓ **For example:** –42 is represented by inverting all of the bits in 42, or 00101010, which yields 11010101, then adding 1, which results in 11010110, or –42. To decode a negative number, first invert all of the bits, and then add 1.
- ✓ **For example:** –42, or 11010110 inverted, yields 00101001, or 41, so when you add 1 you get 42.

The Bitwise Logical Operators

- ✓ The bitwise logical operators are &, |, ^, and ~.
- ✓ The following table shows the outcome of each operation.

A	B	A | B	A & B	A ^ B	~A
0	0	0	0	0	1
1	0	1	0	1	0
0	1	1	0	1	1
1	1	1	1	0	0

The Bitwise NOT

- ✓ Also called the bitwise complement, the unary NOT operator, ~, inverts all of the bits of its operand.
- ✓ **For example:** the number 42 00101010

~42 ~00101010

Result: 11010101

The Bitwise AND

- ✓ The AND operator, &, produces a 1 bit if both operands are also 1.
- ✓ A zero is produced in all other cases.
- ✓ **For example:**

```
  00101010   42
& 00001111   15
----------------------
  00001010   10
```

The Bitwise XOR

- ✓ The XOR operator, ^, combines bits such that if exactly one operand is 1, then the result is 1. Otherwise, the result is zero.
- ✓ **For example:**

```
  00101010   42
^ 00001111   15
--------------------------
  00100101   37
```

Using the Bitwise Logical Operators

Example: // Demonstrate the bitwise logical operators.

```
class BitLogic {
public static void main(String args[]) {
        String binary[] = {
        "0000", "0001", "0010", "0011", "0100", "0101", "0110", "0111",
        "1000", "1001", "1010", "1011", "1100", "1101", "1110", "1111"
        };
                int a = 3; // 0 + 2 + 1 or 0011 in binary
                int b = 6; // 4 + 2 + 0 or 0110 in binary
                int c = a | b;
                int d = a & b;
                int e = a ^ b;
                int f = (~a & b) | (a & ~b);
                System.out.println(" a = " + binary[a]);
                System.out.println(" b = " + binary[b]);
                System.out.println(" a|b = " + binary[c]);
                System.out.println(" a&b = " + binary[d]);
                System.out.println(" a^b = " + binary[e]);
                System.out.println("~a&b|a&~b = " + binary[f]);
        }
}
```

Output:

```
a = 0011
b = 0110
a|b = 0111
a&b = 0010
a^b = 0101
~a&b|a&~b = 0101
```

The Left Shift

- ✓ The left shift operator, <<, shifts all of the bits in a value to the left a specified number of times. It has this general form:

  value << num

- ✓ That is, the << moves all of the bits in the specified value to the left by the number of bit positions specified by num. For each shift left, the high-order bit is shifted out (and lost), and a zero is brought in on the right.
- ✓ This means that when a left shift is applied to an int operand, bits are lost once they are shifted past bit position 31. If the operand is a long, then bits are lost after bit position 63.
- ✓ The outcome of a left shift on a byte or short value will be an int, and the bits shifted left will not be lost until they shift past bit position 31.

Example: // Left shifting a byte value.

```
class ByteShift {
        public static void main(String args[]) {
                byte a = 64, b;
                int i;
                i = a << 2;
                b = (byte) (a << 2);
                System.out.println("Original value of a: " + a);
                System.out.println("i and b: " + i + " " + b);
        }
}
```

Output:

Original value of a: 64
i and b: 256 0

Note: Since **a** is promoted to int for the purposes of evaluation, left-shifting the value 64 (0100 0000) twice results in i containing the value 256 (1 0000 0000). However, the value in b contains 0 because after the shift, the low-order byte is now zero.

The Right Shift

- ✓ The right shift operator, >>, shifts all of the bits in a value to the right a specified number of times. Its general form is shown here:

  value >> num

- ✓ That is, the >> moves all of the bits in the specified value to the right the number of bit positions specified by num.

Example:

```
int a = 32;
```

a = a >> 2; // a now contains 8
int a = 35; a = a >> 2; // a still contains 8
00100011 **35** >> 2 00001000 **8**

Bitwise Operator Compound Assignments

- ✓ All of the binary bitwise operators have a compound form similar to that of the algebraic operators, which combines the assignment with the bitwise operation.

a = a >> 4;	<same>	a >>= 4;
a = a | b;	<same>	a |= b;

Example:

```
class OpBitEquals {
public static void main(String args[]) {
                int a = 1;
                int b = 2;
                int c = 3;
                a |= 4;
                b >>= 1;
                c <<= 1;
                a ^= c;
                System.out.println("a = " + a);
                System.out.println("b = " + b);
                System.out.println("c = " + c);
        }
}
```

Output:

```
a = 3
b = 1
c = 6
```

Relational Operators

- ✓ The relational operators determine the relationship that one operand has to the other. Specifically, they determine equality and ordering.

Operator	Result
==	Equal to
!=	Not equal to
>	Greater than
<	Less than
>=	Greater than or equal to
<=	Less than or equal to

- ✓ The outcome of these operations is a boolean value. The relational operators are most frequently used in the expressions that control the if-statement and the various loop statements.
- ✓ Any type in Java, including integers, floating-point numbers, characters, and Booleans can be compared using the equality test, ==, and the inequality test, !=.

Example:

int a = 4;
int b = 1;
boolean c = a < b;

The result of a<b (which is false) is stored in c.

Boolean Logical Operators

- ✓ The Boolean logical operators shown here operate only on boolean operands.
- ✓ All of the binary logical operators combine two boolean values to form a resultant boolean value.

Operator	Result
&	Logical AND
|	Logical OR
^	Logical XOR (exclusive OR)
||	Short-circuit OR

&&	Short-circuit AND
!	Logical unary NOT
&=	AND assignment
|=	OR assignment
^=	XOR assignment
==	Equal to
!=	Not equal to
?:	Ternary if-then-else

- ✓ The logical Boolean operators, &, |, and ^, operate on boolean values in the same way that they operate on the bits of an integer.
- ✓ The logical ! operator inverts the Boolean state: !true == false and !false == true.

A	B	A | B	A & B	A ^ B	!A
False	False	False	False	False	True
True	False	True	False	True	False
False	True	True	False	True	True
True	True	True	True	False	False

Example: // Demonstrate the boolean logical operators.

```
class BoolLogic {
        public static void main(String args[]) {
                boolean a = true;
                boolean b = false;
                boolean c = a | b;
                boolean d = a & b;
                boolean e = a ^ b;
                boolean f = (!a & b) | (a & !b);
                boolean g = !a;
                System.out.println(" a = " + a);
                System.out.println(" b = " + b);
                System.out.println(" a|b = " + c);
                System.out.println(" a&b = " + d);
                System.out.println(" a^b = " + e);
                System.out.println("!a&b|a&!b = " + f);
                System.out.println(" !a = " + g);
        }
}
```

The ?: Operator

- ✓ Java includes a special ternary (three-way) operator that can replace certain types of if-then-else statements. This operator is the ?:.
- ✓ The ?: has this general form: expression1 ? expression2 : expression3.
- ✓ Here, expression1 can be any expression that evaluates to a boolean value. If expression1 is true, then expression2 is evaluated; otherwise, expression3 is evaluated.

Example: // Demonstrate ?:

```
class Ternary {
public static void main(String args[]) {
        int i, k;
        i = 10;
        k = i < 0 ? -i : i; // get absolute value of i
        System.out.print("Absolute value of ");
        System.out.println(i + " is " + k);
        i = -10;
        k = i < 0 ? -i : i; // get absolute value of i
        System.out.print("Absolute value of ");
        System.out.println(i + " is " + k);
        }
}
```

Output:

```
Absolute value of 10 is 10
Absolute value of -10 is 10
```

Operator Precedence

- ✓ The following table shows the order of precedence for Java operators, from highest to lowest.

Highest			
( )	[ ]	.	
++	--	~	!
*	/	%	
+	-		
>>	>>>	<<	
>	>=	<	<=

==	!=		
&			
^			
|			
&&			
||			
?:			
=	OP=		
Lowest			

Example:
int a=10, b=10, c=30;
int res=a+b*c; <same> int res=a+(b*c);
To override the precedence int res=(a+b)*c;

CONTROL STATEMENTS

- ✓ Programming language uses control statements to control the flow of execution of program.
- ✓ Java's program control statements can be put into the following categories: selection, iteration, and jump.
- ✓ Selection statements allow your program to choose different paths of execution based upon the outcome of an expression or the state of a variable.
- ✓ Iteration statements enable program execution to repeat one or more statements.
- ✓ Jump statements allow your program to execute in a nonlinear fashion.

Java's Selection Statements

- ✓ Java supports two selection statements: if and switch.
- ✓ These statements allow you to control the flow of your program's execution based upon conditions.

The if else statement

- ✓ The *if* statement is Java's conditional branch statement.
- ✓ It can be used to route program execution through two different paths.

General syntax of the *if* statement:
if (condition) statement1; else statement2;
Example:

int a, b;
if(a < b) a = 0;
else b = 0;

- ✓ Here, each statement may be a single statement or a compound statement enclosed in curly braces (that is, a block). The else part is optional.
- ✓ If the condition is true, then statement1 is executed. Otherwise, statement2 (if it exists) is executed.

Nested if else statement

- ✓ Having if else inside another if or else is known as nested if else statement.
- ✓ When we nest ifs, the main thing to remember is that an else statement always refers to the nearest if statement that is within the same block.

Example:

if(i == 10) {
if(j < 20) a = b;
if(k > 100) c = d; // this if is associated with this else
else a = c;
}
else a = d;

The if-else-if Ladder

- ✓ A common programming construct that is based upon a sequence of nested ifs is the if-else-if ladder.

General Systax:
if(condition) statement; else if(condition)

```
        statement;
else if(condition)
        statement;
        ...
else
        statement;
```

- ✓ The *if* statements are executed from the top down.
- ✓ As soon as one of the conditions controlling the *if* is true, the statement associated with that *if* is executed, and the rest of the ladder is bypassed.
- ✓ If none of the conditions is true, then the final *else* statement will be executed.

Example: // Demonstrate if-else-if statements.

```
class IfElse {
        public static void main(String args[]) {
                int month = 4; // April
                String season;
                        if(month == 12 || month == 1 || month == 2)
                                season = "Winter";
                        else if(month == 3 || month == 4 || month == 5)
                                season = "Spring";
                        else if(month == 6 || month == 7 || month == 8)
                                season = "Summer";
                        else if(month == 9 || month == 10 || month == 11)
                                season = "Autumn";
                        else
                                season = "Bogus Month";
                        System.out.println(month+ “is in the " + season + ".");
        }
}
```

Output:

4 is in the Spring.

switch statement

- ✓ The switch statement is a multiway branch statement.
- ✓ It is used to execute different parts of the program code based on the value of an expression.
- ✓ It is a better alternative to if-else-if statements.

General Sytax:

```
switch (expression) {
case value1:
        // statement sequence
        break;
case value2:
        // statement sequence
        break;
...
case valueN:
        // statement sequence
        break;
default:
        // default statement sequence
}
```

- ✓ The expression must be of type byte, short, int, or char; each of the values specified in the case statements must be of a type compatible with the expression.
- ✓ The value of the expression is compared with each of the literal values in the case statements. If a match is found, the code sequence following that case statement is executed.
- ✓ If none of the constants matches the value of the expression, then the default statement is executed.
- ✓ The break statement is used inside the switch to terminate a statement sequence.

Example: // A simple example of the switch.

```
class SampleSwitch {
        public static void main(String args[]) {
        for(int i=0; i<6; i++)
                switch(i) {
```

```
                    case 0:
                        System.out.println("i is zero.");
                        break;
                    case 1:
                        System.out.println("i is one.");
                        break;
                    case 2:
                        System.out.println("i is two.");
                        break;
                    case 3:
                        System.out.println("i is three.");
                        break;
                    default:
                        System.out.println("i is greater than 3.");
                }
            }
}
```

Output:

```
i is zero.
i is one.
i is two.
i is three.
i is greater than 3.
i is greater than 3.
```

Nested switch Statements

- ✓ If we use a switch statement as a part of an outer switch statement, then we call it a nested switch statement.

Example:

```
switch(count) {
        case 1:
                switch(target) { // nested switch
                    case 0:
                        System.out.println("target is zero");
                        break;
                    case 1: // no conflicts with outer switch
                        System.out.println("target is one");
```

```
                    break;
                }
                break;
        case 2: // ...
```

ITERATION STATEMENTS

- ✓ In Java iteration statements are for, while, and do-while.
- ✓ We commonly call them loops.
- ✓ A loop repeatedly executes the same set of instructions until a termination condition is met.

while Statement

- ✓ The while loop repeats a statement or block until its controlling expression is true.

General form:

```
while(condition) {
        // body of loop
}
```

- ✓ The *condition* can be any Boolean expression.
- ✓ The body of the loop will be executed as long as the conditional expression is true.
- ✓ When condition becomes false, control passes to the next line of code after the loop.
- ✓ The curly braces are unnecessary if only a single statement is being repeated.

Example: // Demonstrate the while loop.

```
class While {
        public static void main(String args[]) {
                int n = 10;
                while(n > 0) {
                        System.out.println(n);
                        n--;
                }
        }
}
```

Output:

```
10
9
.
```

```
.
1
```

do-while Statement

- ✓ When it is desirable to execute the body of a loop at least once, even if the conditional expression is false, in such situations we use do-while loop.
- ✓ In do-while loop the termination expression is tested at the end of the loop rather than at the beginning.

General form:

```
do {
        // body of loop
} while (condition);
```

- ✓ Each iteration of the do-while loop first executes the body of the loop and then evaluates the conditional expression.
- ✓ If this expression is true, the loop will repeat. Otherwise, the loop terminates.

Example: // Demonstrate the do-while loop.

```
class DoWhile {
        public static void main(String args[]) {
                int n = 10;
                do {
                        System.out.println("tick " + n);
                        n--;
                } while(n > 0);
        }
}

do {
        System.out.println("tick " + n);
} while(--n > 0);
```

Output:

- ✓ In this example, the expression (– –n > 0) combines the decrement of n and the test for zero into one expression.

- ✓ The do-while loop is especially useful when you process a menu selection, because you will usually want the body of a menu loop to execute at least once.

Example: // Using a do-while to process a menu selection

```
class Menu {
    public static void main(String args[]) throws java.io.IOException {
        char choice;
        do {
            System.out.println("Help on:");
            System.out.println(" 1. if");
            System.out.println(" 2. switch");
            System.out.println(" 3. while");
            System.out.println(" 4. do-while");
            System.out.println(" 5. for\n");
            System.out.println("Choose one:");
            choice = (char) System.in.read();
        } while( choice < '1' || choice > '5');
        System.out.println("\n");
        switch(choice) {
            case '1':
                System.out.println("The if:\n");
                System.out.println("if(condition) statement;");
                System.out.println("else statement;");
                break;
            case '2':
                System.out.println("The switch:\n");
                System.out.println("switch(expression) {");
                System.out.println(" case constant:");
                System.out.println(" statement sequence");
                System.out.println(" break;");
                System.out.println(" // ...");
                System.out.println("}");
                break;
            case '3':
                System.out.println("The while:\n");
                System.out.println("while(condition) statement;");
                break;
            case '4':
                System.out.println("The do-while:\n");
```

```
                        System.out.println("do {");
                        System.out.println(" statement;");
                        System.out.println("} while (condition);");
                        break;
                case '5':
                        System.out.println("The for:\n");
                        System.out.print("for(init; condition; iteration)");
                        System.out.println(" statement;");
                        break;
            }
        }
}
```

Output:

Here is a sample run produced by this program:

```
Help on:
1. if
2. switch
3. while
4. do-while
5. for
Choose one:
4
The do-while:
do {
statement;
} while (condition);
```

for Loop Statement

- ✓ There are two forms of the *for* loop.
- ✓ The first is the traditional form that has been in use since the original version of Java.
- ✓ The second is the new "*for-each*" form.

Traditional form of for loop:

General form:

for(initialization; condition; iteration) {
// body
}

If we have only one statement to be repeated, there is no need for the curly braces.

The *for* loop operates as follows:

- ✓ When the loop first starts, the *initialization* portion of the loop is executed.
- ✓ Generally, this is an expression that sets the value of the loop counter variable.
- ✓ The *initialization* expression is only executed once.
- ✓ Next, *condition* is evaluated. This must be a Boolean expression. It usually tests the loop counter variable against a target value. If this expression is true, then the body of the loop is executed.
- ✓ If it is false, the loop terminates.
- ✓ Next, the *iteration* portion of the loop is executed. This is usually an expression that increments or decrements the loop counter variable.
- ✓ The loop then iterates, first evaluating the *conditional* expression, then executing the body of the loop, and then executing the iteration expression with each pass.
- ✓ This process repeats until the controlling expression is false.

Example: // Demonstrate the *for* loop.

```
class ForTick {
        public static void main(String args[]) {
                int n;
                for(n=10; n>0; n--)
                        System.out.println("tick " + n);
        }
}
```

Declaring Loop Control Variables Inside the *for* Loop

- ✓ The variable that controls a *for* loop is only needed for the purposes of the loop and should not be used elsewhere.
- ✓ In this case, it is possible to declare the variable inside the initialization portion of the *for*.

Example: // Declare a loop control variable inside the *for*.

```
class ForTick {
        public static void main(String args[]) {
                // here, n is declared inside of the for loop
                for(int n=10; n>0; n--)
                        System.out.println("tick " + n);
```

```
        }
}
```

Using the Comma

- ✓ If we want to include more than one statement in the initialization and iteration portions of the *for* loop. We can use comma to separate them.
- ✓ Java permits us to include multiple statements in both the initialization and iteration portions of the *for*.
- ✓ Each statement is separated from the next by a comma.

Example: // Using the comma.

```
class Comma {
        public static void main(String args[]) {
                int a, b;
                for(a=1, b=4; a<b; a++, b--) {
                        System.out.println("a = " + a);
                        System.out.println("b = " + b);
                }
        }
}
```

- ✓ Here is one more for loop variation. You can intentionally create an infinite loop (a loop that never terminates) if you leave all three parts of the *for* empty.

For example:

```
for( ; ; ) {
        // ...
}
```

- ✓ This loop will run forever because there is no condition under which it will terminate.

The *For-Each* Version of the *for* Loop

- ✓ Beginning with JDK 5, a second form of for was defined that implements a "*for-each*" style loop.
- ✓ A *for-each* style loop is designed to cycle through a collection of objects, such as an array, in strictly sequential fashion, from start to finish.

- ✓ Java adds the *for-each* capability by enhancing the for statement.
- ✓ The advantage of this approach is that no new keyword is required, and no preexisting code is broken.
- ✓ The *for-each* style of for is also referred to as the enhanced for loop.

General form:
for(type itr-var : collection) statement-block;

- ✓ Here, type specifies the type and itr-var specifies the name of an iteration variable that will receive the elements from a collection, one at a time, from beginning to end.
- ✓ The collection being cycled through is specified by collection.
- ✓ There are various types of collections that can be used with the *for*, but the only type used in this chapter is the array.
- ✓ With each iteration of the loop, the next element in the collection is retrieved and stored in itr-var.
- ✓ The loop repeats until all elements in the collection have been obtained.
- ✓ Because the iteration variable receives values from the collection, type must be the same as (or compatible with) the elements stored in the collection.

Examples:

Regular *for* loop

```
int nums[] = { 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 };
int sum = 0;
for(int i=0; i < 10; i++) sum += nums[i];
```

The *for-each* form

```
int nums[] = { 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 };
int sum = 0;
for(int x: nums) sum += x;
```

Example Program for *for-each*: // Use a *for-each* style *for* loop.

```
class ForEach {
        public static void main(String args[]) {
                int nums[] = { 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 };
                int sum = 0;
                // use for-each style for to display and sum the values
```

```
                for(int x : nums) {
                        System.out.println("Value is: " + x);
                        sum += x;
                }
        System.out.println("Summation: " + sum);
        }
}
```

Output:

```
Value is: 1
Value is: 2
Value is: 3
Value is: 4
Value is: 5
Value is: 6
Value is: 7
Value is: 8
Value is: 9
Value is: 10
Summation: 55
```

✓ It is possible to terminate the loop early by using a *break* statement.

Example: // Use break with a *for-each* style for.

```
class ForEach2 {
        public static void main(String args[]) {
                int sum = 0;
                int nums[] = { 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 };
                // use for to display and sum the values
                for(int x : nums) {
                        System.out.println("Value is: " + x);
                        sum += x;
                        if(x == 5) break; // stop the loop when 5 is obtained
                }
        System.out.println("Summation of first 5 elements: " + sum);
        }
}
```

Output:

```
Value is: 1
Value is: 2
```

```
Value is: 3
Value is: 4
Value is: 5
Summation of first 5 elements: 15
```

- ✓ The iteration variable in for-each style loop is "read-only" as it relates to the underlying array.
- ✓ An assignment to the iteration variable has no effect on the underlying array.
- ✓ We can't change the contents of the array by assigning the iteration variable a new value.

Example: // The *for-each* loop is essentially read-only.

```
class NoChange {
        public static void main(String args[]) {
                int nums[] = { 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 };
                for(int x : nums) {
                        System.out.print(x + " ");
                        x = x * 10; // no effect on nums
                }
                System.out.println();
                for(int x : nums)
                        System.out.print(x + " ");
                        System.out.println();
        }
}
```

Output:

Iterating Over Multidimensional Arrays

- ✓ The enhanced version of the *for* also works on multidimensional arrays.

Example: // Use *for-each* style *for* on a two-dimensional array.

```
class ForEach3 {
        public static void main(String args[]) {
                int sum = 0;
                int nums[][] = new int[3][5];
                // give nums some values
                for(int i = 0; i < 3; i++)
```

```
                for(int j=0; j < 5; j++)
                        nums[i][j] = (i+1)*(j+1);
                // use for-each for to display and sum the values
                for(int x[] : nums) {
                        for(int y : x) {
                                System.out.println("Value is: " + y);
                                sum += y;
                        }
                }
                System.out.println("Summation: " + sum);
        }
}
```

Output:

```
Value is: 1
Value is: 2
Value is: 3
Value is: 4
Value is: 5
Value is: 2
Value is: 4
Value is: 6
Value is: 8
Value is: 10
Value is: 3
Value is: 6
Value is: 9
Value is: 12
Value is: 15
Summation: 90
```

Nested Loops

- ✓ Java allows loops to be nested. That is, one loop may be inside another.

For example: // Loops may be nested.

```
class Nested {
        public static void main(String args[]) {
                int i, j;
                for(i=0; i<10; i++) {
                        for(j=i; j<10; j++)
```

```
                    System.out.print(".");
                System.out.println();
            }
        }
}
```

Output:

```
.....
.....
.....
.....
.....
.....
....
...
..
.
```

JUMP STATEMENTS

- ✓ Java supports three jump statements: break, continue, and return.
- ✓ These statements transfer control to another part of your program.

Using break

- ✓ In Java, the break statement has three uses:
  a. First, as you have seen, it terminates a statement sequence in a switch statement.
  b. Second, it can be used to exit a loop.
  c. Third, it can be used as a "civilized" form of goto.
- ✓ The last two uses are explained here:

Using break to Exit a Loop

- ✓ By using break, we can force termination of a loop, bypassing the conditional expression and any remaining code in the body of the loop.
- ✓ When a break statement is used inside a loop, the loop is terminated and program control resumes at the next statement following the loop.

Example: // Using break to exit a loop.

```
class BreakLoop {
        public static void main(String args[]) {
```

```
                for(int i=0; i<100; i++) {
                        if(i == 10) break; // terminate loop if i is 10
                                System.out.println("i: " + i);
                }
                        System.out.println("Loop complete.");
        }
}
```

Output:

```
i: 0
i: 1
i: 2
i: 3
i: 4
i: 5
i: 6
i: 7
i: 8
i: 9
Loop complete.
```

Using break as a Form of Goto

- ✓ Java defines an expanded form of the break statement.
- ✓ By using this form of break, you can, for example, break out of one or more blocks of code.
- ✓ These blocks need not be part of a loop or a switch.
- ✓ They can be any block. Further, you can specify precisely where execution will resume, because this form of break works with a label.

General form:

break *label*;

- ✓ *label* is the name of a label that identifies a block of code.
- ✓ When this form of break executes, control is transferred out of the named block.
- ✓ To name a block, put a label at the start of it. A label is any valid Java identifier followed by a colon.

Example: // Using break as a civilized form of goto.

```
class Break {
        public static void main(String args[]) {
                boolean t = true;
                first: {
                        second: {
                                third: {
                                        System.out.println("Before the break.");
                                        if(t) break second; // break out of second block
                                        System.out.println("This won't execute");
                                        }
                                        System.out.println("This won't execute");
                                }
                                System.out.println("This is after second block.");
                        }
        }
}
```

Output:

```
Before the break.
This is after second block.
```

✓ One of the most common uses for a labeled break statement is to exit from nested loops.

Example: // Using break to exit from nested loops

```
class BreakLoop4 {
        public static void main(String args[]) {
                outer: for(int i=0; i<3; i++) {
                                System.out.print("Pass " + i + ": ");
                                for(int j=0; j<100; j++) {
                                        if(j == 10) break outer; // exit both loops
                                        System.out.print(j + " ");
                                }
                        System.out.println("This will not print");
                }
                System.out.println("Loops complete.");
        }
}
```

Output:

```
Pass 0: 0 1 2 3 4 5 6 7 8 9 Loops complete.
```

Using continue

- ✓ The *continue* statement is useful to continue running the loop but stop processing the remainder of the code in its body for this particular iteration.
- ✓ The *continue* statement performs such an action.
- ✓ In *while* and *do-while* loops, a continue statement causes control to be transferred directly to the conditional expression that controls the loop.
- ✓ In a *for* loop, control goes first to the iteration portion of the *for* statement and then to the conditional expression.
- ✓ For all three loops, any intermediate code is bypassed.

Example: // Demonstrate continue.

```
class Continue {
        public static void main(String args[]) {
                for(int i=0; i<10; i++) {
                        System.out.print(i + " ");
                        if (i%2 == 0) continue;
                                System.out.println("");
                        }
        }
}
```

Output:

```
0 1
2 3
4 5
6 7
8 9
```

- ✓ As with the *break* statement, *continue* may specify a label to describe which enclosing loop to continue.

Example: // Using *continue* with a label.

```
class ContinueLabel {
        public static void main(String args[]) {
                outer: for (int i=0; i<10; i++) {
                                for(int j=0; j<10; j++) {
```

```
                    if(j > i) {
                        System.out.println();
                        continue outer;
                    }
                    System.out.print(" " + (i * j));
                }
            }
            System.out.println();
        }
}
```

Output:

```
0
0 1
0 2 4
0 3 6 9
0 4 8 12 16
0 5 10 15 20 25
0 6 12 18 24 30 36
0 7 14 21 28 35 42 49
0 8 16 24 32 40 48 56 64
0 9 18 27 36 45 54 63 72 81
```

return Statement:

- ✓ The *return* statement is used to explicitly return from a method.
- ✓ That is, it causes program control to transfer back to the caller of the method.
- ✓ At any time in a method the *return* statement can be used to cause execution to branch back to the caller of the method.
- ✓ Thus, the *return* statement immediately terminates the method in which it is executed.

Example: // Demonstrate return.

```
class Return {
    public static void main(String args[]) {
        boolean t = true;
        System.out.println("Before the return.");
        if(t) return; // return to caller
        System.out.println("This won't execute.");
    }
```

```
}
```

Output:

```
Before the return.
```

- ✓ Here, *return* causes execution to return to the Java run-time system, since it is the run-time system that calls main( ).

INTRODUCING CLASSES

- ✓ The class is at the core of Java.
- ✓ It is the logical construct upon which the entire Java language is built because it defines the shape and nature of an object.
- ✓ The class forms the basis for object-oriented programming in Java.
- ✓ Any concept you wish to implement in a Java program must be encapsulated within a class.

Class Fundamentals

- ✓ Classes have been used since the beginning of this book, simply to encapsulate the main( ) method, which has been used to demonstrate the basics of the Java syntax.
- ✓ Class is used to define a new data type. Once defined, this new type can be used to create objects of that type.
- ✓ Thus, a class is a template for an object, and an object is an instance of a class.
- ✓ The two words object and instance used interchangeably.

The General Form of a Class

- ✓ When you define a class, you declare its exact form and nature.
- ✓ You do this by specifying the data that it contains and the code that operates on that data.
- ✓ While very simple classes may contain only code or only data, most real-world classes contain both.
- ✓ A class is declared by use of the ***class*** keyword.

General form:

```
class classname {
        type instance-variable1;
```

```
        type instance-variable2;
        // ...
        type instance-variableN;
        type methodname1(parameter-list) {
                // body of method
        }

        // ...
        type methodnameN(parameter-list) {
        // body of method
        }
}
```

- ✓ The data, or variables, defined within a class are called instance variables.
- ✓ The code is contained within methods.
- ✓ Collectively, the methods and variables defined within a class are called members of the class.
- ✓ Variables defined within a class are called instance variables because each instance of the class (that is, each object of the class) contains its own copy of these variables.
- ✓ Thus, the data for one object is separate and unique from the data for another.
- ✓ All methods have the same general form as main( ), they will not be specified as static or public. Java classes do not need to have a main( ) method.
- ✓ Only one of the program class is the starting point for your program and is going to have a main() method. Further, applets don't require a main( ) method at all.

A Simple Class

- ✓ Here is a class called Box that defines three instance variables: width, height, and depth.
- ✓ Currently, Box does not contain any methods.

```
class Box {
        double width;
        double height;
        double depth;
}
```

- ✓ In this case, the new data type is called Box.

- ✓ You will use this name to declare objects of type Box.
- ✓ It is important to remember that a class declaration only creates a template; it does not create an actual object.

To create a Box object:

```
Box mybox = new Box();
```

- ✓ After this statement executes, mybox will be an instance of Box. Thus, it will have "physical" reality.
- ✓ Every Box object will contain its own copies of the instance variables width, height, and depth. To access these variables, you will use the dot (.) operator.
- ✓ The dot operator links the name of the object with the name of an instance variable.

mybox.width = 100;

Example Program

```
class Box {
        double width;
        double height;
        double depth;
}
// This class declares an object of type Box.
class BoxDemo {
        public static void main(String args[]) {
                Box mybox = new Box();
                double vol;
                // assign values to mybox's instance variables
                        mybox.width = 10;
                        mybox.height = 20;
                        mybox.depth = 15;
                // compute volume of box
                vol = mybox.width * mybox.height * mybox.depth;
                System.out.println("Volume is " + vol);
        }
}
```

Output:

```
Volume is 3000.0
```

- ✓ File that contains this program should br BoxDemo.java, because the main( ) method is in the class called BoxDemo, not the class called Box.
- ✓ When you compile this program, you will find that two .class files have been created, one for Box and one for BoxDemo.
- ✓ The Java compiler automatically puts each class into its own .class file.
- ✓ To run this program, you must execute BoxDemo.class.
- ✓ Each object has its own copies of the instance variables.
- ✓ This means that if you have two Box objects, each has its own copy of depth, width, and height.
- ✓ It is important to understand that changes to the instance variables of one object have no effect on the instance variables of another.

Example: Demonstrating the effect on the instance variables.

```
class Box {
        double width;
        double height;
        double depth;
}
class BoxDemo2 {
        public static void main(String args[]) {
        Box mybox1 = new Box();
        Box mybox2 = new Box();
        double vol;
        // assign values to mybox1's instance variables
        mybox1.width = 10;
        mybox1.height = 20;
        mybox1.depth = 15;
        /* assign different values to mybox2's
        instance variables */
        mybox2.width = 3;
        mybox2.height = 6;
        mybox2.depth = 9;
        // compute volume of first box
        vol = mybox1.width * mybox1.height * mybox1.depth;
```

```
        System.out.println("Volume is " + vol);
        // compute volume of second box
        vol = mybox2.width * mybox2.height * mybox2.depth;
        System.out.println("Volume is " + vol);
        }
}
```

Output:

```
Volume is 3000.0
Volume is 162.0
```

Declaring Objects

- ✓ When we create a class, we are creating a new data type.
- ✓ We can use this type to declare objects of that type.
- ✓ Obtaining objects of a class is a two-step process.
- ✓ First, we must declare a variable of the class type. This variable does not define an object. Instead, it is simply a variable that can refer to an object.
- ✓ Second, we must acquire an actual, physical copy of the object and assign it to that variable. We can do this using the new operator. The new operator dynamically allocates (that is, allocates at run time) memory for an object and returns a reference to it.
- ✓ The following is used to declare an object of type Box:

```
Box mybox = new Box();
```

- ✓ This statement combines the two steps just described:

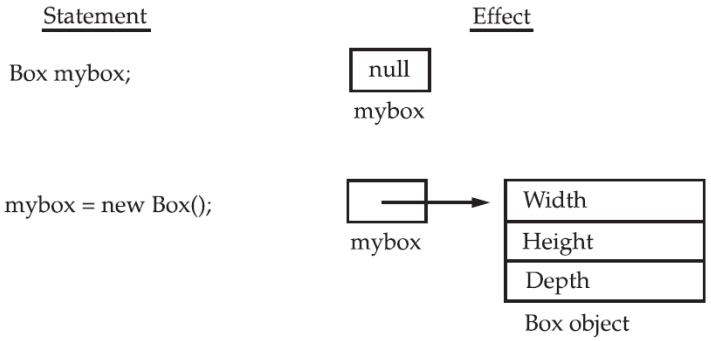
```
Box mybox; // declare reference to object
mybox = new Box(); // allocate a Box object
```

A Closer Look at new

- ✓ The new operator dynamically allocates memory for an object.

It has this general form:

```
class-var = new classname( );
```



- ✓ The *classname* is the name of the class that is being instantiated. The class name followed by parentheses specifies the constructor for the class.
- ✓ A constructor defines what occurs when an object of a class is created.
- ✓ Constructors are an important part of all classes and have many significant attributes.
- ✓ Most real-world classes explicitly define their own constructors within their class definition. If no explicit constructor is specified, then Java will automatically supply a default constructor. This is the case with Box.

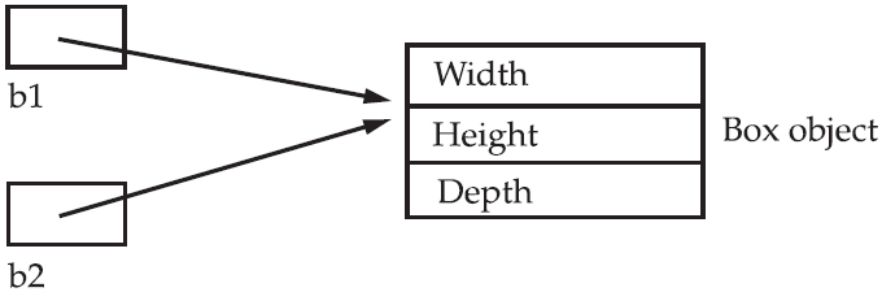
The difference between a class and an object is that:

- ✓ A class creates a logical framework that defines the relationship between its members. An object of a class is an instance of that class.
- ✓ A class is a logical construct. An object has physical reality. (That is, an object occupies space in memory.)

Assigning Object Reference Variables

```
Box b1 = new Box();
Box b2 = b1;
```

- ✓ After this fragment executes, b1 and b2 will both refer to the same object. The assignment of b1 to b2 did not allocate any memory or copy any part of the original object. It simply makes b2 refer to the same object as b1.
- ✓ Any changes made to the object through b2 will affect the object to which b1 is referring, since they are the same object.



- ✓ Although b1 and b2 both refer to the same object, they are not linked in any other way.
- ✓ A subsequent assignment to b1 will simply unhook b1 from the original object without affecting the object or affecting b2.

Example:

```
Box b1 = new Box();
Box b2 = b1;
// ...
b1 = null;
```

Introducing Methods

- ✓ Classes usually consist of two things: instance variables and methods.
- ✓ A method is a set of statements or logic encoded as a single unit.

General form:

```
type name(parameter-list) {
    // body of method
}
```

- ✓ The *type* specifies the type of data returned by the method. This can be any valid type, including class types that you create. If the method does not return a value, its return type must be void.
- ✓ The name of the method *name* can be any legal identifier.
- ✓ The *parameter-list* is a sequence of type and identifier pairs separated by commas. Parameters are essentially variables that receive the value of the arguments passed to the method when it is called. If the method has no parameters, then the parameter list will be empty.

Adding a Method to the Box Class

- ✓ Methods are used to access the instance variables defined by the class.
- ✓ Methods define the interface to most classes.

Example:

```
class Box {
        double width;
        double height;
        double depth;
        // display volume of a box
        void volume() {
        System.out.print("Volume is ");
        System.out.println(width * height * depth);
        }
}
class BoxDemo3 {
        public static void main(String args[]) {
                Box mybox1 = new Box();
                Box mybox2 = new Box();
                        // assign values to mybox1's instance variables
                mybox1.width = 10;
                mybox1.height = 20;
                mybox1.depth = 15;
                        /* assign different values to mybox2's instance variables */
                mybox2.width = 3;
```

```
                mybox2.height = 6;
                mybox2.depth = 9;
                        // display volume of first box
                mybox1.volume();
                        // display volume of second box
                mybox2.volume();
        }
}
```

Output:

```
Volume is 3000.0
Volume is 162.0
```

Example: Returning a Value

```
class Box {
        double width;
        double height;
        double depth;
                // compute and return volume
        double volume() {
        return width * height * depth;
        }
}
class BoxDemo4 {
        public static void main(String args[]) {
                Box mybox1 = new Box();
                Box mybox2 = new Box();
                double vol;
                        // assign values to mybox1's instance variables
                mybox1.width = 10;
                mybox1.height = 20;
                mybox1.depth = 15;
                        /* assign different values to mybox2's instance variables */
                mybox2.width = 3;
                mybox2.height = 6;
                mybox2.depth = 9;
```

```
                    // get volume of first box
                vol = mybox1.volume();
                System.out.println("Volume is " + vol);
                    // get volume of second box
                vol = mybox2.volume();
                System.out.println("Volume is " + vol);
        }
}
```

Output:

Example: Adding a Method That Takes Parameters

```
int square()
{
        return 10 * 10;
}
int square(int i)
{
        return i * i;
}
```

Now, square( ) will return the square of whatever value it is called with.

```
int x, y;
x = square(5); // x equals 25
x = square(9); // x equals 81
y = 2;
x = square(y); // x equals 4
```

CONSTRUCTORS

- ✓ Java allows objects to initialize themselves when they are created. This automatic initialization is performed through the use of a constructor.
- ✓ A constructor initializes an object immediately upon creation. It has the same name as the class in which it resides and is syntactically similar to a method.
- ✓ Once defined, the constructor is automatically called immediately after the object is created, before the new operator completes.
- ✓ The parentheses are needed after the class name to call the constructor for that class.
- ✓ When we do not explicitly define a constructor for a class, then Java creates a default constructor for the class. The default constructor automatically initializes all instance variables to zero.

Example: //Constructor demonstration.

```
class Box {
        double width;
        double height;
        double depth;
                // This is the constructor for Box.
        Box() {
                System.out.println("Constructing Box");
                width = 10;
                height = 10;
                depth = 10;
        }
        // compute and return volume
        double volume() {
                return width * height * depth;
        }
}

class BoxDemo6 {
        public static void main(String args[]) {
```

```
                // declare, allocate, and initialize Box objects
                Box mybox1 = new Box();
                Box mybox2 = new Box();
                double vol;
                // get volume of first box
                vol = mybox1.volume();
                System.out.println("Volume is " + vol);
                // get volume of second box
                vol = mybox2.volume();
                System.out.println("Volume is " + vol);
        }
}
```

Output:

Parameterized Constructors

- ✓ To initialize object properties with various values, rather than having the values for all objects, the parameterized constructor is used.

For example: // Parameterized Constructors

```
class Box {
        double width;
        double height;
        double depth;
        // This is the constructor for Box.
        Box(double w, double h, double d) {
                width = w;
                height = h;
                depth = d;
        }
        // compute and return volume
        double volume() {
                return width * height * depth;
        }
}
```

```
class BoxDemo7 {
        public static void main(String args[]) {
                // declare, allocate, and initialize Box objects
                Box mybox1 = new Box(10, 20, 15);
                Box mybox2 = new Box(3, 6, 9);
                double vol;
                // get volume of first box
                vol = mybox1.volume();
                System.out.println("Volume is " + vol);
                // get volume of second box
                vol = mybox2.volume();
                System.out.println("Volume is " + vol);
        }
}
```

Output:

```
Volume is 3000.0
Volume is 162.0
```

GARBAGE COLLECTION

- ✓ Objects are dynamically allocated by using the *new* operator.
- ✓ Java takes a different approach to destroy the objects and release the memory; it handles deallocation automatically. This technique is called garbage collection.
- ✓ When no references to an object exist, that object is assumed to be no longer needed, and the memory occupied by that object can be released.
- ✓ There is no need to destroy the objects explicitly. Garbage collection only occurs during the execution of your program.

The finalize( ) Method

- ✓ Sometimes when we have something to perform while an object is destroying itself, we can do so by calling the *finalize()* method.
- ✓ By using finalization, we can define specific actions that will occur when an object is just about to be reclaimed by the garbage collector.

- ✓ To add a finalizer to a class, simply define the *finalize*( ) method. The Java run time calls that method whenever it is about to recycle an object of that class.

General form:

```
protected void finalize( )
{
        // finalization code here
}
```

OVERLOADING METHODS

- ✓ If two or more methods within the same class that share the same name, with different parameter declarations, the methods are said to be overloaded, and the process is referred to as method overloading.
- ✓ Method overloading is one of the ways that Java supports polymorphism.
- ✓ When Java encounters a call to an overloaded method, it checks the type and/or number of arguments as its guide to determine the version of the overloaded method to actually call.

Example: // Overloading Methods

```
class OverloadDemo {
        void test() {
                System.out.println("No parameters");
        }
        void test(int a) {
                System.out.println("a: " + a);
        }
        void test(int a, int b) {
                System.out.println("a and b: " + a + " " + b);
        }

        double test(double a) {
                System.out.println("double a: " + a);
                return a*a;
        }
}

class Overload {
```

```
    public static void main(String args[]) {
        OverloadDemo ob = new OverloadDemo();
        double result;
        ob.test();
        ob.test(10);
        ob.test(10, 20);
        result = ob.test(123.25);
        System.out.println("Result of ob.test(123.25): " + result);
    }
}
```

Output:

OVERLOADING CONSTRUCTORS

- ✓ Like methods the constructors can also be overloaded with different type and/or number of arguments.
- ✓ This process is called constructor overloading.

Example: // Constructor Overloading.

```
class Box {
    double width;
    double height;
    double depth;
    Box(double w, double h, double d) {
        width = w;
        height = h;
        depth = d;
    }
    Box() {
        width = -1; // use -1 to indicate
        height = -1; // an uninitialized
        depth = -1; // box
    }
    Box(double len) {
        width = height = depth = len;
    }
    double volume() {
        return width * height * depth;
```

```
        }
}

class OverloadCons {
        public static void main(String args[]) {
                Box mybox1 = new Box(10, 20, 15);
                Box mybox2 = new Box();
                Box mycube = new Box(7);
                double vol;
                vol = mybox1.volume();
                System.out.println("Volume of mybox1 is " + vol);
                vol = mybox2.volume();
                System.out.println("Volume of mybox2 is " + vol);
                vol = mycube.volume();
                System.out.println("Volume of mycube is " + vol);
        }
}
```

Output:

ARGUMENT PASSING/PARAMETER PASSING

There are two ways in which a computer language can pass an argument to a method:

- ✓ The first way is call-by-value. This approach copies the value of an argument into the formal parameter of the method. Changes made to the formal parameter of the method do not have any effect on the argument.
- ✓ The second way an argument can be passed is call-by-reference. In this approach, a reference to an argument (not the value of the argument) is passed to the formal parameter. Inside the method, this reference is used to access the actual argument specified in the call. This means that changes made to the formal parameter will affect the argument used to call the method. Java supports both approaches, depending upon what is passed.

Call-by-value mechanism:

- ✓ In Java, when you pass a primitive type to a method, it is passed by value. Changes made to the formal once do not affect the actual once.

Example: // Call-by-value mechanism.

```
class Test {
        void meth(int i, int j) {
                i *= 2;
                j /= 2;
        }
}
class CallByValue {
        public static void main(String args[]) {
                Test ob = new Test();
                int a = 15, b = 20;
                System.out.println("a and b before call: " +a + " " + b);
                ob.meth(a, b);
                System.out.println("a and b after call: " +a + " " + b);
        }
}
```

Output:

Call-by-reference mechanism:

- ✓ When an object is passed the reference of a class is passed.
- ✓ Creating a variable of a class type only creates a reference to an object. Passing this reference to a method, the parameter that receives it will refer to the same object as that referred to by the argument.
- ✓ This effectively means that objects are passed to methods by use of call-by-reference. Changes made to the object inside the method do affect the object used as an argument.

Example: // Call-by-reference mechanism.

```
class Test {
        int a, b;
        Test(int i, int j) {
                a = i;
                b = j;
        }
        void meth(Test o) {
                o.a *= 2;
                o.b /= 2;
```

```
        }
}

class CallByRef {
        public static void main(String args[]) {
                Test ob = new Test(15, 20);
                System.out.println("ob.a and ob.b before call: " +
                ob.a + " " + ob.b);
                ob.meth(ob);
                System.out.println("ob.a and ob.b after call: " +
                ob.a + " " + ob.b);
        }
}
```

Output:

RETURNING OBJECTS

- ✓ A method can return any type of data, including class types that we create.

Example: // Demonstrating Returning Objects.

```
class Test {
        int a;
        Test(int i) {
                a = i;
        }
        Test incrByTen() {
                Test temp = new Test(a+10);
                return temp;
        }
}
class RetOb {
        public static void main(String args[]) {
                Test ob1 = new Test(2);
                Test ob2;
                ob2 = ob1.incrByTen();
                System.out.println("ob1.a: " + ob1.a);
                System.out.println("ob2.a: " + ob2.a);
                ob2 = ob2.incrByTen();
```

```
                System.out.println("ob2.a after second increase: "+ ob2.a);
        }
}
```

Output:

```
ob1 == ob2: true
ob1 == ob3: false
```

RECURSION

- ✓ Recursion is the process of defining something in terms of itself.
- ✓ As it relates to Java programming, recursion is the attribute that allows a method to call itself.
- ✓ A method that calls itself is said to be recursive.
- ✓ The classic example of recursion is the computation of the factorial of a number.

Example: // Demonstrating Recursion.

```
class Factorial {
        int fact(int n) {
        int result;
        if(n==1) return 1;
        result = fact(n-1) * n;
        return result;
        }
}
class Recursion {
        public static void main(String args[]) {
                Factorial f = new Factorial();
                System.out.println("Factorial of 3 is " + f.fact(3));
                System.out.println("Factorial of 4 is " + f.fact(4));
                System.out.println("Factorial of 5 is " + f.fact(5));
        }
}
```

Output:

```
Factorial of 3 is 6
Factorial of 4 is 24
Factorial of 5 is 120
```

INTRODUCING ACCESS CONTROL

- ✓ Encapsulation links data with the code that manipulates it.
- ✓ Encapsulation provides another important attribute: access control. Through encapsulation, we can control what parts of a program can access the members of a class. By controlling access, we can prevent misuse.
- ✓ **For example:** allowing access to data only through a well-defined set of methods, we can prevent the misuse of that data. If properly implemented, a class creates a "black box" which may be used, but the inner workings of which are not open to tampering.
- ✓ How a member can be accessed is determined by the *access specifier* that modifies its declaration. Java supplies a rich set of access specifiers. Some aspects of access control are related mostly to inheritance or packages.
- ✓ Java's access specifiers are public, private, and protected. Java also defines a default access level.
- ✓ When a member of a class is modified by the public specifier, then that member can be accessed by any other code.
- ✓ When a member of a class is specified as private, then that member can only be accessed by other members of its class.
- ✓ Protected applies only when inheritance is involved.

Example: // Demonstrating Access control.

```
/* This program demonstrates the difference between public and private.*/
class Test {
        int a; // default access
        public int b; // public access
        private int c; // private access
        // methods to access c
        void setc(int i) { // set c's value
        c = i;
        }
        int getc() { // get c's value
        return c;
        }
}
```

```
class AccessTest {
        public static void main(String args[]) {
                Test ob = new Test();
                // These are OK, a and b may be accessed directly
                ob.a = 10;
                ob.b = 20;
                // This is not OK and will cause an error
                // ob.c = 100; // Error!
                // You must access c through its methods
                ob.setc(100); // OK
                System.out.println("a, b, and c: " + ob.a + " " +ob.b + " " + ob.getc());
        }
}
```

Output:

NESTED AND INNER CLASSES

- ✓ It is possible to define a class within another class; such classes are known as *nested classes*.
- ✓ The scope of a nested class is bounded by the scope of its enclosing class. If class B is defined within class A, then B does not exist independently of A.
- ✓ Nested class has access to the members, including private members, of the class in which it is nested. The enclosing class does not have access to the members of the nested class.
- ✓ There are two types of nested classes: static and non-static.
- ✓ A static nested class is one that has the static modifier applied. It must access the members of its enclosing class through an object. That is, it cannot refer to members of its enclosing class directly.
- ✓ The most important type of nested class is the inner class. An inner class is a non-static nested class. It has access to all of the variables and methods of its outer class and may refer to them directly in the same way that other non-static members of the outer class do.

Example: // Demonstrating Nested and Inner classes

```
class Outer {
        int outer_x = 100;
        void test() {
                Inner inner = new Inner();
                inner.display();
        }
        // this is an inner class
        class Inner {
                void display() {
                        System.out.println("display: outer_x = " + outer_x);
                }
        }
}
class InnerClassDemo {
        public static void main(String args[]) {
                Outer outer = new Outer();
                outer.test();
        }
}
```

Output:

```
display: outer_x = 100
```

✓ **Note:** It is important to understand that an instance of Inner can be created only within the scope of class Outer. The Java compiler generates an error message if any code outside of class Outer attempts to instantiate class Inner. To create an instance of Inner outside of Outer one must qualify its name with Outer, as Outer.Inner.

Example: // Demonstrating Nested and Inner classes

```
// This program will not compile.
class Outer {
        int outer_x = 100;
        void test() {
                Inner inner = new Inner();
                inner.display();
        }
                // this is an inner class
                class Inner {
                        int y = 10; // y is local to Inner
                        void display() {
```

```
                    System.out.println("display: outer_x = " + outer_x);
                }
            }
        void showy() {
            System.out.println(y); // error, y not known here!
        }
}

class InnerClassDemo {
        public static void main(String args[]) {
            Outer outer = new Outer();
            outer.test();
        }
}
```

Output:
This program will not compile.

EXPLORING THE STRING CLASS

- ✓ String is probably the most commonly used class in Java's class library. The obvious reason for this is that strings are a very important part of programming.
- ✓ Every string you create is actually an object of type String. Even string constants are actually String objects.

  For example: in System.out.println("This is a String, too"); the string "This is a String, too" is a String constant.
- ✓ Strings can be constructed in a variety of ways.

  The easiest way is: String myString = "this is a test";
- ✓ Once a *String* object is created it can be used anywhere a string is allowed.

  For example: System.out.println(myString);
- ✓ Java defines a special operator for String objects: +. It is used to concatenate two strings.

  For example: String myString = "I" + " like " + "Java.";
- ✓ The *String* class contains several methods.
  - o To test the equality of two strings the equals( ) method can be used.
  - o To obtain the length of a string the length( ) method can be used.

- To obtain the character at a specified index within a string the charAt( ) method can be used.

General forms:

```
boolean equals(String object);
int length( );
char charAt(int index);
```

Example: // Demonstrating some String methods.

```
class StringDemo2 {
    public static void main(String args[]) {
        String strOb1 = "First String";
        String strOb2 = "Second String";
        String strOb3 = strOb1;
        System.out.println("Length of strOb1: " +
        strOb1.length());
        System.out.println("Char at index 3 in strOb1: " +
        strOb1.charAt(3));
        if(strOb1.equals(strOb2))
            System.out.println("strOb1 == strOb2");
        else
            System.out.println("strOb1 != strOb2");
        if(strOb1.equals(strOb3))
            System.out.println("strOb1 == strOb3");
        else
            System.out.println("strOb1 != strOb3");
    }
}
```

Output:

```
Length of strOb1: 12
Char at index 3 in strOb1: s
strOb1 != strOb2
strOb1 == strOb3
```