



## **MODULE 1**

### **An Overview of Java:**

- Object Oriented Programming
- A First Simple Program
- A Second Short Program
- Two Control statements
- Using Blocks of code
- Lexical Issues
- The Java Class Libraries

### **Data Types, Variables and Arrays:**

- Java is a Strongly typed Language
- The Primitive types.
- Integers
- Floating-Point types
- Characters
- Booleans
- A Closer look at Literals
- Variables
- Type conversion and casting
- Automatic Type promotion in Expressions
- Arrays
- A few words about strings

## **Object Oriented Programming:**

- Object oriented programming (OOP) is the core of Java programming.
- Java is a general purpose, object- oriented programming language developed by Sun Microsystems. It was invented by James Gosling and his team and was initially called as Oak.
- The most important feature that made Java very popular was the “Platform-Independent” approach.
- It was the first programming language that did not tie-up with any particular operating system( or hardware) rather Java programs can be executed anywhere and on any system.
- Java was designed for the development of the software for consumer electronic devices like TVs, VCRs, etc.

## **Two Paradigms:**

- Every program contains 2 components code and data.
- Two approaches are there to solve the problem and in program writing: Procedure oriented and object oriented.

### ***Procedure Oriented:***

- Procedure oriented programs are written based on “whats happening” around, where the code acts on data. Ex: C etc
- Problems increases in procedure oriented as the program grows larger and more complex.

### ***Object Oriented:***

- Object oriented programs are written based on “Who is being affected” around, which manages the increasing complexity.
- It organises program around data and well defined interfaces of that data.
- Characterised as data controlling access to code. Ex: C++, JAVA, Small Talk etc

## **The Three OOP:**

The three important features of OOP are:

- Encapsulation
- Inheritance
- Polymorphism

**Encapsulation:**

- Encapsulation is the mechanism that binds together code and data it manipulates, and keeps both safe from outside interference and misuse.
- In Java the basis of encapsulation is the class. A class defines the state and behavior( data & code) that will be shared by set of objects.
- Each object contains the structure and behavior defined by the class. The data defined by the class are called instance variables(member variables), the code that operates on that data are called methods(member functions).

**Inheritance:**

- Inheritance is the process by which one object acquires the properties of another object. This is important as it supports the concept of hierarchical classification.
- By the use of inheritance, a class has to define only those qualities that make it unique. The general qualities can be derived from the parent class or base class.
- Ex: A child inheriting properties from parents.

**Polymorphism**

- Polymorphism (meaning many forms) is a feature that allows one interface to be used for a general class of actions. The specific action determined by the exact nature of the situation. This concept is often expressed as “ one interface, multiple methods”.
- Ex: “+” can be used for addition of 2 numbers and also concatenation of 2 strings.

`System.out.println(2+4); // outputs 6 as answer`

`System.out.println(“Hello” + “Gautham”); // outputs Hello Gautham as answer`

Apart from this the additional features include:

**Object:**

- An object can be any real world entity.
- Ex: an animal, bank, human, box, fan etc
- An object is a software bundle of related state and behavior.
- An object is an instance of class.

***Class:***

- A class is a blueprint or prototype from which objects are created.
- Its just a template for an object, which describes an object.
- Ex: a class describes how an animal looks like.
- A class is a user defined data type.

***Abstraction:***

- Data abstraction refers to providing only essential information to the outside world and hiding their background details i.e., to represent the needed informatin in program without presenting the details.
- Ex: a database system hides certain details of how data is stored and created and maintained.

***Polymorphism, Encapsulation and Inheritance work Together***

- The 3 principles of OOP Polymorphism, Encapsulation and Inheritance combines together to make the programming robust and scalable.
- Encapsulation allows to migrate the implementation without disturbing the code that depends on class.
- Polymorphism allows to create clean, sensible, readable, resilient code.
- Inheritance mainly deals with the code reusability.

***A First Simple Program***

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

1. Open the notepad and type the above program

2. Save the above program with **.java** extension, here file name and class name should be same,

**ex: Example.java**

3. Open the command prompt and **Compile** the above program

**javac Example.java**

From the above compilation the java compiler produces a bytecode(.class file)

4. Finally run the program through the  
**interpreter java Example.java**

### **Output of the program:**

Welcome to Programming in Java

### **Note:**

- In Java all code must reside inside a class and name of that class should match the name of the file that holds the program.
- Java is case-sensitive

### **Compiling the program**

- To compile the program, execute the compiler “javac”, specifying the name of the source file on the command line as shown below

**C:\> javac Example.java**

- The “javac: compiler creates a file called “Example.class” that contains the bytecode version of the program.
- To run the program, we must use the java interpreter called “java”. To do so we pass the class name “Example” as a command-line argument as shown below

**C:\> java Example**

- When you run the program we get the output:

**Welcome to Programming in Java**

### **Description:**

- (1) **Class declaration:** “class Example” declares a class, which is an object- oriented construct. Sampleone is a Java identifier that specifies the name of the class to be

defined.

- (2) **Opening braces:** Every class definition of Java starts with opening braces and ends with matching one.
- (3) **The main line:** the line “ public static void main(String args[]) “ defines a method name main. Java application program must include this main. This is the starting point of the interpreter from where it starts executing. A Java program can have any number of classes but only one class will have the main method.
- (4) **Public:** This key word is an access specifier that declares the main method as unprotected and therefore making it accessible to the all other classes.
- (5) **Static:** Static keyword defines the method as one that belongs to the entire class and not for a particular object of the class. The main must always be declared as static.
- (6) **Void:** the type modifier void specifies that the method main does not return any value.
- (7) **The println:** It is a method of the object out of system class. It is similar to the printf or cout of c or c++. This always appends a newline character to the end of the string i.e, any subsequent output will start on a new line.

### **A Second Short Program**

```
/* This is a short example
   Name of file : Example2.java */
class Example2{
    public static void main(String args[])
    {
        int n=3;
        System.out.println(" the value of n is "+n);
        n=n+5;
        System.out.print(" the new value is");
        System.out.println(n);
    }
}
```

Output:

the value of n is 3

the new value of n is 8

The statement `System.out.println(" the value of n is "+n)`, the sign "+" causes the value of "n" to be appended to the string that precedes it, and the resulting string is output. (Actually n is first converted from an integer into its string equivalent and then concatenated with the string that precedes it)

The `System.out.print( )` method is just like `println( )` except that it does not output a newline character after each call.

## **Two Control Statements**

Here in this chapter initially we focus on two control statements `if` and `for loop`, the detailed control statements will be discussed in module 2.

### **if statement**

- The `if`- statement is the most basic of all the control flow statements. It tells your program to execute a certain section of code *only if* a particular test evaluates to true.
- Here is the general form of the **`if`** statement:  
**`if (condition) statement;`**
- Here the condition is Boolean expression.
- If the condition is true then the statement is executed, if false then statement will be skipped.

Example:

```
class Example
{
    public static void main(String args[])
    {
        int a=10;
        if(a>0)
            System.out.println("a is positive number");
        System.out.println(" End of program");
    }
}
```

```
}
```

In the above program since a is greater than 0 it prints the output as  
a is positive number

End of program

If incase a is -1 or negative value the condition fails and it prints only

End of program

### **The for loop**

- The for loop is similar to that of C/C++
- Here is the general form of the traditional **for** statement:

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

- Initialization sets the loop control variable to initial value.
- Condition is a Boolean expression which tests the loop
- Iteration expression tells how the control variable has to change at each iteration.  
Generally the increment or decrement operator is used to perform iteration.

Example:

```
class Example  
{  
    public static void main(String args[])  
    {  
        int a;  
        for(a=0;a<5;a++)  
            System.out.println(a);  
        System.out.println(" End of program");  
    }  
}
```

Output:

0  
1



2

3

4

End of Program

### Using blocks of code

- Java supports **code blocks** - which means that two or more statements are grouped into blocks of code.
- Opening and closing braces is used to achieve this.
- Each block is treated as logical unit.
- Whenever two or more statements has to be linked blocks can be used.

Example:

```
class Example
{
    public static void main(String args[])
    {
        int a=10;
        if(a>0)
        { // begin of block
            System.out.println("a is positive number");
            System.out.println(" inside block");
        } // end of block
    }
}
```

### Lexical issues:

Java programs are a collection of whitespace, identifiers, literals, comments, operators, separators, and keywords.

### Whitespace:

- Java is a free from language- means no need to follow any indentation rules.
- Whitespace is a space, tab, or newline.

**Java character set:**

- The smallest unit of Java language are its character set used to write Java tokens. This character are defined by unicode character set that tries to create character for a large number of character worldwide.
- The Unicode is a 16-bit character coding system and currently supports 34,000 defined characters derived from 24 languages of worldwide.

**Key Words:**

Java program is basically a collection of classes. A class is defined by a set of declaration statements and methods containing executable statements. Most statement contains an expression that contains the action carried out on data. The compiler recognizes the tokens for building up the expression and statements. Smallest individual units of programs are known as tokens. Java language includes five types of tokens. They are

(a) Reserved Keyword

(b) Identifiers

(c) Literals.

(d) Operators

(e) Separators.

**Reserved keyword:**

Java language has 50 words as reserved keywords. They implement specific feature of the language. The keywords combined with operators and separators according to syntax build the Java language.

abstract	continue	for	new	switch
assert	default	goto	package	synchronized
boolean	do	if	private	this
break	double	implements	protected	throw
byte	else	import	public	throws
case	enum	instanceof	return	transient
catch	extends	int	short	try
char	final	interface	static	void
class	finally	long	strictfp	volatile
const	float	native	super	while

**Identifiers:**

Identifiers are programmer-designed token used for naming classes methods variable, objects, labels etc. The rules for identifiers are

1. They can have alphabets, digits, dollar sign and underscores.
2. They must not begin with digit.
3. Uppercase and lower case letters are distinct.
4. They can be any lengths.
5. Name of all public method starts with lowercase.
6. In case of more than one word starts with uppercase in next word.
7. All private and local variables use only lowercase and underscore.
8. All classes and interfaces start with leading uppercases.
9. Constant identifier uses uppercase letters only.

**Example for valid identifiers:**

Var\_1, count, \$value etc

**Example for invalid identifiers:**

6name, var@value, my/name etc

**Literals:**

Literals in Java are sequence of characters that represents constant values to be stored in variables. Java language specifies five major types of Literals. They are:

1. Integer Literals.
2. Floating-point Literals.
3. Character Literals.
4. String Literals.
5. Boolean Literals.

**Operators:**

An operator is a symbol that takes one or more arguments and operates on them to produce an result.

## **Separators:**

Separators are the symbols that indicates where group of code are divided and arranged.

Some of the operators are:

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 <b>for</b> statement.
.	Period	Used to separate package names from subpackages and classes. Also used to separate a variable or method from a reference variable.

## **Comments:**

- Java supports 3 styles of comments
- **Multiline comment:** this type of comment begins with `/*` and ends with `*/`

Ex: `/* Welcome to`

`Java Programming */`

- **Single line comments:** this type of comment begins with `//` and ends at the end of current line

Ex: `// Welcome to java Programming`

- **Documentation Comment:** this type of comment is used to produce an HTML file that documents your program. The documentation comment begins with `/**` and ends with `*/`

## **Java Class libraries:**

Java environment has several built in class libraries.

Java standard library includes hundreds of classes and methods grouped into several functional packages. Most commonly used packages are:

- (a) Language support Package.
- (b) Utilities packages.
- (c) Input/output packages

- (d) Networking packages
- (e) AWT packages.
- (f) Applet packages.

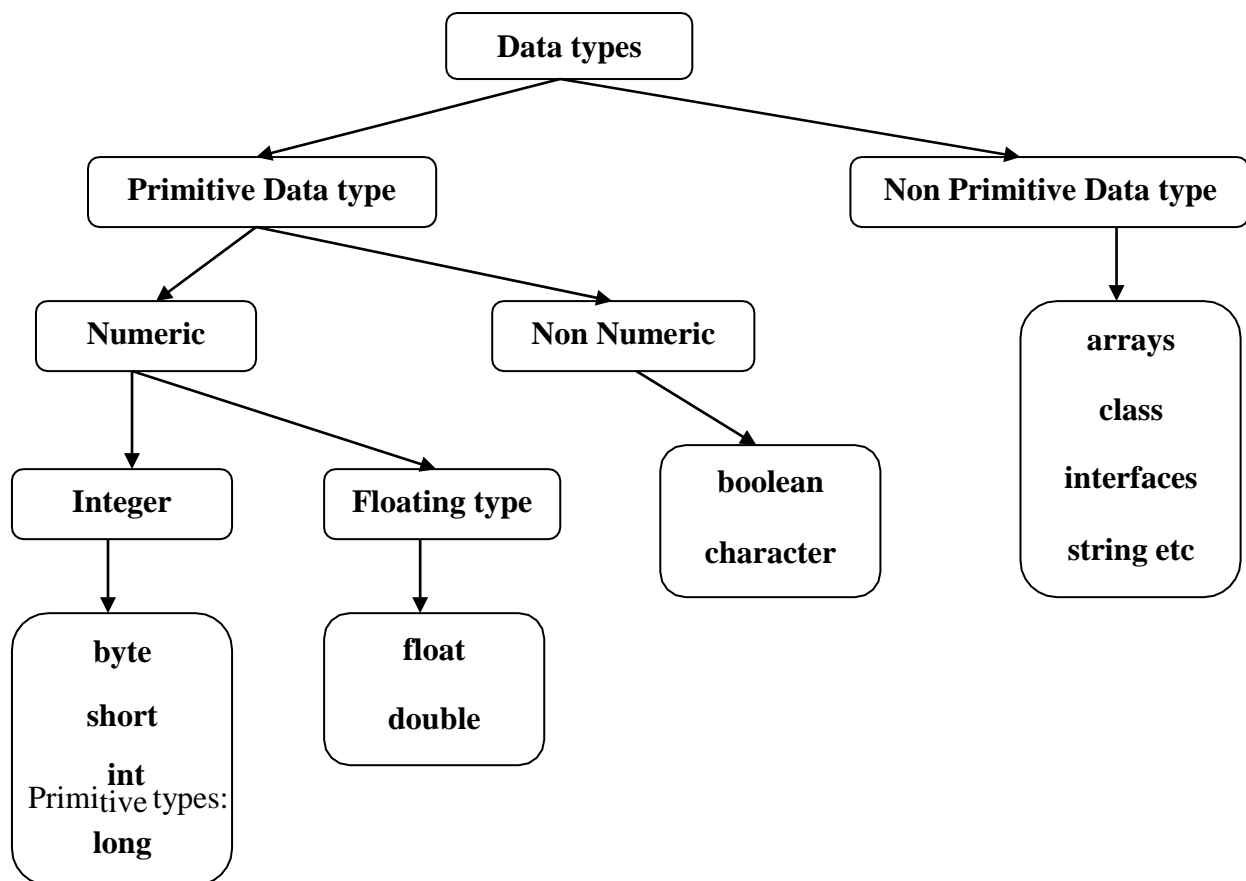
## Data types, variables and arrays

**Java is a strongly typed language:**

- The strongly typed nature of Java gives it the robustness and safety for it.
- Every variable and expression has strictly defined type.
- Assignments, parameter passing or explicit value passing are checked for type compatibility.
- Java compiler checks all expressions and parameters to ensure type compatibility.

### Data types

The various data types supported in java is as follows



Java defines eight *primitive* types of data: **byte, short, int, long, char, float, double**, and **boolean**. As shown in above figure.

- The primitive types represent single values—not complex objects. Although Java is otherwise completely object-oriented, the primitive types are not.

- They are analogous to the simple types found in most other non-object-oriented languages.
- The reason for this is efficiency. Making the primitive types into objects would have degraded performance too much. The primitive types are defined to have an explicit range and mathematical behavior.
- Because of Java's portability requirement, all data types have a strictly defined range. For example, an **int** is always 32 bits, regardless of the particular platform.

## 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.
- Many other computer languages support both signed and unsigned integers.
- However, Java's designers felt that unsigned integers were unnecessary. Specifically, they felt that the concept of *unsigned* was used mostly to specify the behavior of the *high-order bit*, which defines the *sign* of an integer value.

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

### byte

- The smallest integer type is **byte**.
- This is a signed 8-bit type that has a range from -128 to 127.
- Variables of type **byte** are especially useful when you're working with a stream of data from a network or file.
- Byte variables are declared by use of the **byte** keyword.
- For example, the following declares two **byte** variables called **b** and **c**: byte b, c;

### short

- **short** is a signed 16-bit type.
- It has a range from -32,768 to 32,767.
- It is probably the least-used Java type.

- Here are some examples of **short** variable declarations:

```
short s;
```

```
short t;
```

### *Floating-Point Types*

- Floating-point numbers, also known as *real* numbers, are used when evaluating expressions that require fractional precision.
- For example, calculations such as square root, or transcendental such as sine and cosine, result in a value whose precision requires a floating-point type.
- There are two kinds of floating-point types, **float** and **double**, which represent single- and double-precision numbers, respectively.

#### *float*

- The type **float** specifies a *single-precision* value that uses 32 bits of storage.

#### *double*

- Double precision, as denoted by the **double** keyword, uses 64 bits to store a value.
- Double precision is actually faster than single precision on some modern processors that have been optimized for high-speed mathematical calculations.

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**.
- However, C/C++ programmers beware: **char** in Java is not the same as **char** in C or C++.
- In C/C++, **char** is 8 bits wide. This is *not* the case in Java. Instead, 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

### ***Booleans:***

Java has a simple type called **boolean** for logical values. It can have only one of two possible values. They are true or false.

Data Type	Default Value	Default size
boolean	False	1 bit
Char	'\u0000'	2 byte
Byte	0	1 byte
short	0	2 byte
Int	0	4 byte
Long	0L	8 byte
Float	0.0f	4 byte
double	0.0d	8 byte

### ***Literals:***

A constant value in Java is created by using a literal representation of it. There are 5 types of literals.

- Integer Literals.
- Floating-point Literals.
- Character Literals.
- String Literals.
- Boolean Literals.

### ***Integer literals:***

- Any whole number value is an integer literal.
- These are all decimal values describing a base 10 number.
- There are two other bases which can be used in integer literal, octal( base 8) where 0 is prefixed with the value, hexadecimal (base 16) where 0X or 0x is prefixed with the integer value.

Example:

int decimal = 100;

int octal = 0144;

int hexa = 0x64;

***Floating point literals:***

- The default type when you write a floating-point literal is double, but you can designate it explicitly by appending the D (or d) suffix
- However, the suffix F (or f) is appended to designate the data type of a floating-point literal as float.
- We can also specify a floating-point literal in scientific notation using Exponent (short E ore), for instance: the double literal 0.0314E2 is interpreted as:

Example:

0.0314 \*10<sup>2</sup> (i.e 3.14).

6.5E+32 (or 6.5E32) Double-precision floating-point literal

7D Double-precision floating-point literal

.01f Floating-point literal

***Character literals:***

- char data type is a single 16-bit Unicode character.
- We can specify a character literal as a single printable character in a pair of single quote characters such as 'a', '#', and '3'.
- You must know about the ASCII character set. The ASCII character set includes 128 characters including letters, numerals, punctuation etc.
- Below table shows a set of these special characters.

Escape	Meaning
\n	New line
\t	Tab
\b	Backspace
\r	<u>Carriage return</u>
\f	Formfeed
\\	Backslash
\'	Single quotation mark
\"	Double quotation mark
\d	Octal
\xd	Hexadecimal
\ud	Unicode character

**Boolean Literals:**

- The values true and false are treated as literals in Java programming.
- When we assign a value to a boolean variable, we can only use these two values.
- Unlike C, we can't presume that the value of 1 is equivalent to true and 0 is equivalent to false in Java.
- We have to use the values true and false to represent a Boolean value.

Example

```
boolean chosen = true;
```

**String Literal**

- The set of characters is represented as String literals in Java.
- Always use "double quotes" for String literals.
- There are few methods provided in Java to combine strings, modify strings and to know whether two strings have the same values.

Example:

```
"hello world"
```

```
"Java"
```

**Variables:**

A variable is an identifier that denotes a storage location used to store a data value. A variable may have different value in the different phase of the program. To declare one identifier as a variable there are certain rules. They are:

1. They must not begin with a digit.
2. Uppercase and lowercase are distinct.
3. It should not be a keyword.
4. White space is not allowed.

**Declaring Variable:** One variable should be declared before using.

The syntax is

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

Example:

```
int a,b,c;
```

```
float quot, div;
```

**Initializing a variable:** A variable can be initialize in two ways. They are

- (a) Initializing by Assignment statements.
- (b) Dynamic Initialisation

**Initializing by assignment statements:**

- One variable can be initialize using assignment statements. The syntax is :

**Variable-name = Value;**

Example: int a=10,b,c=16;

Double pi=3.147;

**Dynamic initialization:**

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

Example:

```
class Example
{
    public static void main(String args[])
    {
        double a=10, b=2.6;
        double c=a/b;
        System.out.println("value of c 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*.
- A scope determines what objects are visible to other parts of your program. It also determines the lifetime of those objects.
- Many other computer languages define two general categories of scopes: **global** and **local**. However, these traditional scopes do not fit well with Java's strict, object-oriented model.
- As a general rule, variables declared inside a scope are not visible (that is, accessible) to code that is defined outside that scope. Thus, when you declare a variable within a

scope, you are localizing that variable and protecting it from unauthorized access and/or modification.

```
class Scope
{
public static void main(String args[])
{
    int x; // known to all code within main x = 10;
    if(x == 10)          // start new scope
    {
        int y = 20;  // known only to this block
                    // x and y both known here.
        System.out.println("x and y: " + x + " " + y); x = y * 2;
    }
    // y = 100; // Error! y not known here
    // x is still known here. System.out.println("x is " + x);
}
}
```

**Note:**

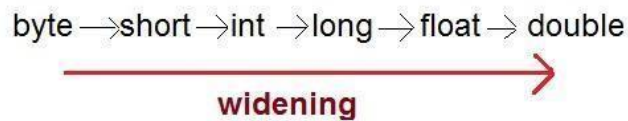
- *There should not be two variables with the same name in different scope.*
- *The variable at outer scope can be accessed in inner scope but vice versa is not possible.*

### **Type Conversion and casting**

It is often necessary to store a value of one type into the variable of another type. In these situations the value that to be stored should be casted to destination type. Assigning a value of one type to a variable of another type is known as **Type Casting** .Type casting can be done in two ways.

In Java, type casting is classified into two types,

### 1. Widening Casting(Implicit)



### 2. Narrowing Casting(Explicitly done)



### Widening or Automatic type conversion

Automatic Type casting take place when,

- ☐ the two types are compatible
- ☐ the target type is larger than the source type

### Example :

```
public class Test
{
    public static void main(String[] args)
    {
        int i = 100;
        long l = i;           //no explicit type casting required
        float f = l;          //no explicit type casting required
        System.out.println("Int value "+i);
        System.out.println("Long value "+l);
        System.out.println("Float value "+f);
    }
}
```

### Output :

Int value 100

Long value 100

Float value 100.0

---

### **Narrowing or Explicit type conversion**

When you are assigning a larger type value to a variable of smaller type, then you need to perform explicit type casting.

#### **Example :**

```
public class Test
{
    public static void main(String[] args)
    {
        double d = 100.04;
        long l = (long)d;      //explicit type casting required
        int i = (int)l;        //explicit type casting required
        System.out.println("Double value "+d);
        System.out.println("Long  value  "+l);
        System.out.println("Int value "+i);
    }
}
```

#### **Output :**

Double value 100.04

Long value 100

Int value 100

### **Automatic type promotion in expressions:**

- Type conversions also occurs in expressions.
- Java automatically promotes each byte, short, or char operand to int when evaluating an expression.

```
byte b = 50;
```

```
b = b * 2; // Error! Cannot assign an int to a byte!
```

the operands were automatically promoted to int when the expression was evaluated, the result has also been promoted to int. Thus, the result of the expression is now of type int, which cannot be assigned to a byte without the use of a cast.

```
byte b = 50;  
b = (byte)(b * 2); which yields the correct value of 100.
```

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.

## **Arrays in Java**

**Array** which stores a fixed-size sequential collection of elements of the same type.

An array is used to store a collection of data, but it is often more useful to think of an array as a collection of variables of the same type.

### **Declaring Array Variables:**

To use an array in a program, you must declare a variable to reference the array, and you must specify the type of array the variable can reference. Here is the syntax for declaring an array variable:

```
dataType[] arrayRefVar;           or           dataType arrayRefVar[];
```

### **Example:**

The following code snippets are examples of this syntax:

```
int[] myList;                       or                       int myList[];
```



## Creating Arrays:

You can create an array by using the new operator with the following syntax:

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

The above statement does two things:

- It creates an array using new **dataType[arraySize];**
- It assigns the reference of the newly created array to the variable **arrayRefVar**.

Declaring an array variable, creating an array, and assigning the reference of the array to the variable can be combined in one statement, as shown below:

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

Alternatively you can create arrays as follows:

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

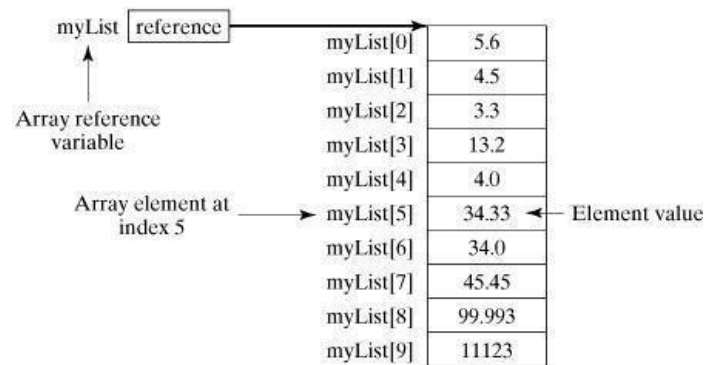
The array elements are accessed through the **index**. Array indices are 0-based; that is, they start from 0 to **arrayRefVar.length-1**.

## Example:

Following statement declares an array variable, myList, creates an array of 10 elements of double type and assigns its reference to myList:

```
double[] myList = new double[10];
```

Following picture represents array myList. Here, myList holds ten double values and the indices are from 0 to 9.



### Processing Arrays:

When processing array elements, we often use either for loop or foreach loop because all of the elements in an array are of the same type and the size of the array is known.

### Example:

Here is a complete example of showing how to create, initialize and process arrays:

```
class TestArray
```

```
{
    public static void main(String[] args)
    {
        double[] myList = {1.9, 2.9, 3.4, 3.5};

        // Print all the array elements
        for (int i = 0; i < 4; i++)
        {
            System.out.println(myList[i] + " ");
        }
    }
}
```

### Multidimensional Arrays

Java does not support multidimensional arrays. However, you can declare and create an array of arrays (and those arrays can contain arrays, and so on, for however many dimensions you need), and access them as you would C-style multidimensional arrays:

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

```
coords[0][0] = 1; coords[0][1] = 2;
```

### **A few words about strings:**

- Java supports string type which is an object. It is used to declare string variables
- Array of strings can also be declared.
- A string variable can be assigned to another string variable.
- String variable can also be used as argument.

Example:

```
String name1="gautham", name2;
```

```
Name2=name1; // sets name2 with value gautham
```

```
System.out.println(name2); // string variable passed as parameter.
```

### **Programs:**

#### **// Compute distance light travels using long variables.**

```
class Light
{
    public static void main(String args[])
    {
        int lightspeed;
        long days;
        long seconds;
        long distance; // approximate speed of light in miles per second
        lightspeed = 186000;
        days = 1000; // specify number of days here
        seconds = days * 24 * 60 * 60; // convert to seconds
        distance = lightspeed * seconds; // compute distance
        System.out.print("In " + days);
        System.out.print(" days light will travel about ");
        System.out.println(distance + " miles.");
    }
}
```

This program generates the following output:

**In 1000 days light will travel about 16070400000000 miles.**

**// Compute the area of a circle.**

```
class Area
{
    public static void main(String args[])
    {
        double pi, r, a; r = 10.8; // radius of circle
        pi = 3.1416; // pi, approximately
        a = pi * r * r; // compute area
        System.out.println("Area of circle is " + a);
    }
}
```

**// Demonstrate char data type.**

```
class CharDemo
{
    public static void main(String args[])
    {
        char ch1, ch2; ch1 = 88; // code for X
        ch2 = 'Y';
        System.out.print("ch1 and ch2: ");
        System.out.println(ch1 + " " + ch2);
    }
}
```

This program displays the following output:

**ch1 and ch2: X Y**

**// char variables behave like integers.**

```
class CharDemo2
{
    public static void main(String args[])
    {
        char ch1; ch1 = 'X';
    }
}
```

```
        System.out.println("ch1 contains " + ch1);
        ch1++; // increment ch1
        System.out.println("ch1 is now " + ch1);
    }
}
```

The output generated by this program is shown here:

**ch1 contains X ch1 is now Y**

**// Demonstrate boolean values.**

```
class BoolTest
{
    public static void main(String args[])
    {
        boolean b;
        b = false;
        System.out.println("b is " + b);
        b = true;
        System.out.println("b is " + b); // a boolean value can control the if statement
        if(b)
            System.out.println("This is executed.");
        b = false;
        if(b)
            System.out.println("This is not executed.");
        System.out.println("10 > 9 is " + (10 > 9));
    }
}
```

The output generated by this program is shown here:

**b is false**

**b is true**

**This is executed.**

**10 > 9 is true**

**Scope of variable**

```
class LifeTime
```

```
{  
    public static void main(String args[])  
    {  
        int x;  
        for(x = 0; x < 3; x++)  
        {  
            int y = -1; // y is initialized each time block is entered  
            System.out.println("y is: " + y); // this always prints -1  
            y = 100;  
            System.out.println("y is now: " + y);  
        }  
    }  
}
```

The output generated by this program is shown here:

```
y is: -1  
y is now: 100  
y is: -1  
y is now: 100  
y is: -1  
y is now: 100
```

**Type conversion**

```
class Conversion
```

```
{  
    public static void main(String args[])  
    {  
        byte b;  
        int i = 257;  
        double d = 323.142;  
        System.out.println("\nConversion of int to byte.");  
        b = (byte) i;  
        System.out.println("i and b " + i + " " + b);  
    }  
}
```

```
        System.out.println("\nConversion of double to int.");
        i = (int) d;
        System.out.println("d and i " + d + " " + i);
        System.out.println("\nConversion of double to byte.");
        b = (byte) d; System.out.println("d and b " + d + " " + b);
    }
}
```

This program generates the following output:

**Conversion of int to byte.**

**i and b 257 1**

**Conversion of double to int.**

**d and i 323.142 323**

**Conversion of double to byte.**

**d and b 323.142 67**

### **Additional:**

#### **Applications of Java**

- Some of the applications of Java is that internet users can use Java to create applet programs and run them using a web-browser.
- The first application program written in Java was HotJava, a web browser to run applet on internet.(An applet is a special kind of Java program that is designed to be transmitted over the internet and automatically executed by a Jva compatible web browser)
- Further internet users can also set up their websites containing java applets,that could be used by other remote users of the internet. Hence Java is popularly called as “Language of Internet”.
- Before the invention of Java, world wide web was limited to displaying text and still images. However, the incorporation of Java into web pages has made web capable of supporting animations, graphics, games and wide range of special effects.

We can develop two types of Java application. They are:

- (1). Stand alone Java application.
- (2). Web applets.

**Stand alone Java application:** Stand alone Java application are programs written in Java to carry out certain tasks on a certain stand alone system. Executing a stand-alone Java program contains two phases:

- (a) Compiling source code into bytecode using javac compiler.
- (b) Executing the bytecoded program using Java interpreter.

**Java applet:** Applets are small Java program developed for Internet application. An applet located on a distant computer can be downloaded via Internet and execute on local computer.

### Java Environment:

- Java environment includes a large number of development tools and hundred of classes and methods.
- The development tools are part of the system known as **Java Development Kit(JDK)**
- The classes and methods are apart of the **Java Standard Library (JSL)** also known as Application Program Interface (API)
- **JRE(java runtime environment)** consists of tools required to execute the java code, containing JVM, runtime class libraries, user interface toolkits.

### Java Development Kit:

The Java Development Kit comes with a collection of tools that are used for developing and running java programs. They include,

- ➔ **applet viewer:** Enables us to run java applet (without actually using a Java compatible browser)
- ➔ **java:** Java interpreter, which runs applets and applications by reading and interpreting bytecode files.
- ➔ **javac:** the Java compiler, which translates java source code to bytecode files that the interpreter can understand.
- ➔ **javadoc:** creates HTML format documentation from java source code.
- ➔ **javah:** produces header files for use with native methods.
- ➔ **javap:** Java disassembler, which enables us to convert bytecode files into a program description.



➔ **jdb**: Java debugger, which helps to find errors in programs.

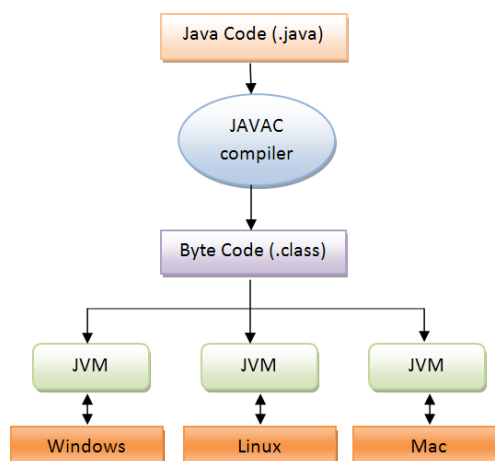
### Java Standard Library(JSL):

Java standard library includes hundreds of classes and methods grouped into several functional packages. Most commonly used packages are:

- (g) Language support Package.
- (h) Utilities packages.
- (i) Input/output packages
- (j) Networking packages
- (k) AWT packages.
- (l) Applet packages.

### JVM(Java Virtual Machine):

The concept of Write-once-run-anywhere (known as the Platform independent) is one of the important key feature of java language that makes java as the most powerful language. Not even a single language is idle to this feature but java is closer to this feature. The programs written on one platform can run on any platform provided the platform must have the JVM (**Java Virtual Machine**). A Java virtual machine (JVM) is a virtual machine that can execute Java bytecode. It is the code execution component of the Java software platform. The below figure shows the JVM



**Fig: Java Virtual Machine**

**Java is iterpreted:**

Java as a language initially gained popularity mainly due to its platform independent architecture or portability feature. The reason for Java to be portable is that it is interpreted. Firstly Java compiler translates source code into bytecode(an intermediate representation). This byte code is given as an input to the Java interpreter that generates the machine code that can be executed by the native system. So we call java as an interpreter language.

**The Bytecode:**

This concept allows java to solve both security and portability problems. When the source program is given as an input to the Java compiler, it will never generate the machine level language executable code, rather it generates “bytecode”.

Bytecodes are highly optimized set of instructions designed to be executed by the java runtime system called JVM. Translating a java program into bytecode makes it much easier to run a program in a wide variety of environment because only the JVM needs to be implemented for each platform.

**Java Features:**

- (1) Compiled and Interpreted
- (2) Architecture Neutral/Platform independent and portable
- (3) Object oriented
- (4) Robust and secure.
- (5) Distributed.
- (6) Familiar, simple and small.
- (7) Multithreaded and interactive.
- (8) High performance
- (9) Dynamic and extendible.

**1. Compiled and Interpreted**

Usually a computer language is either compiled or interpreted. Java combines both these approaches; first java compiler translates source code into bytecode instructions. Bytecodes are not machine instructions and therefore, in the second stage, java interpreter generates machine code that can be directly executed by the machine that is running the java program.

**2. Architecture Neutral/Platform independent and portable**

Java programs can be easily moved from one computer to another, anywhere and anytime. Changes in operating systems,

**3. Object oriented**

In java everything is an Object. Java can be easily extended since it is based on the Object model. Java is a true object oriented language. All data resides in objects and classes

**4. Robust and secure.**

Java is a robust language; Java makes an effort to eliminate error situations by emphasizing mainly on compile time error checking and runtime checking. Because of absence of pointers in java we can easily achieve the security.

**5. Distributed.**

Java is designed for the distributed environment of the internet. java applications can open and access remote objects on internet as easily as they can do in the local system.

**6. Familiar, simple and small.**

Java is designed to be easy to learn. If you understand the basic concept of OOP java would be easy to master.

**7. Multithreaded and interactive.**

With Java's multi-threaded feature it is possible to write programs that can do many tasks simultaneously. This design feature allows developers to construct smoothly running interactive applications.

**8. High performance**

Because of the intermediate bytecode java language provides high performance

**9. Dynamic and extendible.**

Java is considered to be more dynamic than C or C++ since it is designed to adapt to an evolving environment. Java programs can carry extensive amount of run-time information that can be used to verify and resolve accesses to objects on run-time.

**Implementing a Java program:** Java program implementation contains three stages.

They are:

1. Create the source code.
2. Compile the source code.
3. Execute the program.

**(1) Create the source code:**

1. Any editor can be used to create the Java source code.
2. After coding the Java program must be saved in a file having the same name of the class containing main() method.
3. Java code file must have .Java extension.

**(2) Compile the source code:**

1. Compilation of source code will generate the bytecode.
2. JDK must be installed before completion.
3. Java program can be compiled by typing `javac <filename>.java`
4. It will create a file called `<filename>.class` containing the bytecode.

**(3) Executing the program:**

1. Java program once compiled can be run at any system.
2. Java program can be execute by typing `Java <filename>`

## Module 2

### Control Statements

- 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 (that is, iteration statements form loops).
- *Jump* statements allow your program to execute in a nonlinear fashion.

#### Java's Selection Statements

- Java supports two selection statements: **if** and **switch**.

#### The if statement

- The **if** statement executes a block of code only if the specified expression is true.
- If the value is false, then the **if** block is skipped and execution continues with the rest of the program.
- You can either have a single statement or a block of code within an **if** statement.
- Note that the conditional expression must be a Boolean expression.

##### **Syntax:**

```
if (<conditional expression>) {  
    <statements>  
}
```

#### **Example:**

```
public class Example {  
    public static void main(String[] args) {  
        int a = 10, b = 20;  
        if (a > b)  
            System.out.println("a > b");  
        if (a < b)
```

```
        System.out.println("b > a");
    }
}
```

### 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.
- Here is the general form of the **if** statement:

**Syntax:**

```
if (condition)
    statement1;
else statement2;
```

- Here, each *statement* may be a single statement or a compound statement enclosed in curly braces (that is, a *block*).
- The *condition* is any expression that returns a **boolean** value. The **else** clause is optional.
- The **if** works like this: If the *condition* is true, then *statement1* is executed. Otherwise, *statement2* (if it exists) is executed.

Example:

```
public class Example {
    public static void main(String[] args) {
        int a = 10, b = 20;
        if (a > b)
            System.out.println("a > b");
        else
            System.out.println("b > a");
    }
}
```

### Nested ifs

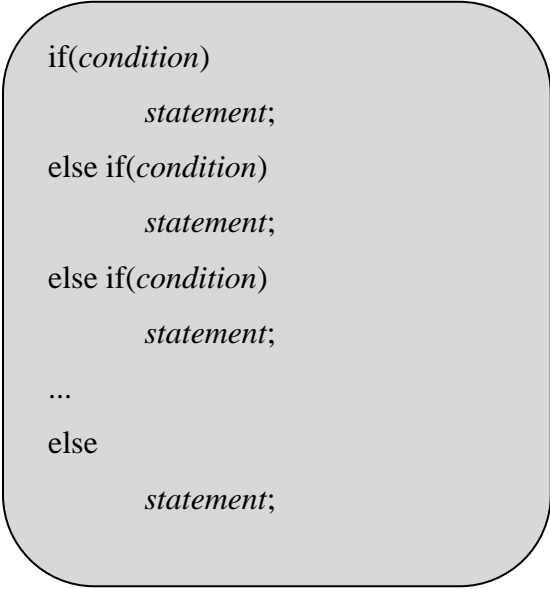
- A *nested if* is an **if** statement that is the target of another **if** or **else**.
- When you nest **ifs**, the main thing to remember is that an **else** statement always refers to the nearest **if** statement that is within the same block as the **else** and that is not already associated with an **else**.

Here is an example:

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

### The if-else-if Ladder

- A common programming construct that is based upon a sequence of nested **ifs** is the *if-else-if ladder*.
- It looks like this:



```
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:

```
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("April is in the " + season + ".");  
    }  
}
```

### **The switch statement**

- The **switch case** statement is a multi-way branch with several choices. A switch is easier to implement than a series of if/else statements.

#### **Structure of Switch:**

- The switch statement begins with a keyword, followed by an expression that equates to a no long integral value.
- Following the controlling expression is a code block that contains zero or more labeled cases.
- Each label must equate to an integer constant and each must be unique.

#### **Working of switch case:**

- When the switch statement executes, it compares the value of the controlling expression to the values of each case label.



- The program will select the value of the case label that equals the value of the controlling expression and branch down that path to the end of the code block.
- If none of the case label values match, then none of the codes within the switch statement code block will be executed. Java includes a **default** label to use in cases where there are no matches.
- We can have a nested switch within a case block of an outer switch.

**Syntax:**

```
switch (<non-long integral expression>) {  
    case label1: <statement1> ; break;  
    case label2: <statement2> ; break;  
    ...  
    case labeln: <statementn> ; break;  
    default: <statement>  
}
```

**Example:**

```
public class Example {  
    public static void main(String[] args) {  
        int a = 10, b = 20, c = 30;  
        int status = -1;  
        if (a > b && a > c) {  
            status = 1;  
        } else if (b > c) {  
            status = 2;  
        } else {  
            status = 3;  
        }  
        switch (status) {  
            case 1:  
                System.out.println("a is the greatest");  
                break;  
            case 2:
```

```
        System.out.println("b is the greatest");
        break;
    case 3:
        System.out.println("c is the greatest");
        break;
    default:
        System.out.println("Cannot be determined");
    }
}
```

- The **break** statement is optional. If you omit the **break**, execution will continue on into the next **case**.
- It is sometimes desirable to have multiple **cases** without **break** statements between them.
- For example, consider the following program:

// In a switch, break statements are optional.

```
class MissingBreak {
    public static void main(String args[]) {
        for(int i=0; i<12; i++)
            switch(i) {
                case 0:
                case 1:
                case 2:
                case 3:
                case 4:
                    System.out.println("i is less than 5");
                    break;
                case 5:
                case 6:
                case 7:
                case 8:
                case 9:
```

```
        System.out.println("i is less than 10");
        break;
    default:
        System.out.println("i is 10 or more");
    }
}
}
```

### Nested switch Statements

- You can use a **switch** as part of the statement sequence of an outer **switch**. This is called a *nested switch*.
- Since a **switch** statement defines its own block, no conflicts arise between the **case** constants in the inner **switch** and those in the outer **switch**.
- For example, the following fragment is perfectly valid:

```
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: // ...
}
```

In summary, there are three important features of the **switch** statement to note:

- The **switch** differs from the **if** in that **switch** can only test for equality, whereas **if** can evaluate any type of Boolean expression. That is, the **switch** looks only for a match between the value of the expression and one of its **case** constants.
- No two **case** constants in the same **switch** can have identical values. Of course, a **switch** statement and an enclosing outer **switch** can have **case** constants in common.

- A **switch** statement is usually more efficient than a set of nested **ifs**.

## Iteration Statements

### The while loop

- The **while** statement is a looping construct control statement that executes a block of code while a condition is true.
- You can either have a single statement or a block of code within the while loop.
- The loop will never be executed if the testing expression evaluates to false.
- The loop condition must be a **boolean** expression.

#### **Syntax:**

```
while (<loop condition>) {  
    <statements>  
}
```

#### **Example:**

```
public class Example {  
    public static void main(String[] args) {  
        int count = 1;  
        System.out.println("Printing Numbers from 1 to 10");  
        while (count <= 10) {  
            System.out.println(count++);  
        }  
    }  
}
```

### The do-while loop

- The **do-while** loop is similar to the **while** loop, except that the test is performed at the end of the loop instead of at the beginning.
- This ensures that the loop will be executed at least once.

**Syntax:**

```
do {  
    <loop body>  
} while (<loop condition>);
```

**Example:**

```
public class Example {  
    public static void main(String[] args) {  
        int count = 1;  
        System.out.println("Printing Numbers from 1 to 10");  
        do {  
            System.out.println(count++);  
        } while (count <= 10);  
    }  
}
```

**The for loop**

- The **for** loop is a looping construct which can execute a set of instructions a specified number of times. It's a counter controlled loop.

**Syntax:**

```
for (<initialization>; <loop condition>; <increment expression>) {  
    <loop body>  
}
```

**Example:**

```
public class Example {  
    public static void main(String[] args) {  
        System.out.println("Printing Numbers from 1 to 10");  
        for (int count = 1; count <= 10; count++) {  
            System.out.println(count);  
        }  
    }  
}
```

### **Declaring Loop Control Variables Inside the for Loop**

- Often the variable that controls a **for** loop is only needed for the purposes of the loop and is not used elsewhere.
- When this is the case, it is possible to declare the variable inside the initialization portion of 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);  
    }  
}
```

- When you declare a variable inside a **for** loop, there is one important point to remember: the scope of that variable ends when the **for** statement does

### **Using the Comma**

- There will be times when you will want to include more than one statement in the initialization and iteration portions of the **for** loop.

```
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);  
        }  
    }  
}
```

### **Some for Loop Variations**

- The **for** loop supports a number of variations that increase its power and applicability. The reason it is so flexible is that its three parts—the initialization, the conditional test, and the iteration—do not need to be used for only those purposes can be used for any purpose you desire.

- One of the most common variations involves the conditional expression.
- Specifically, this expression does not need to test the loop control variable against some target value. In fact, the condition controlling the **for** can be any Boolean expression. For example, consider the following fragment:

```
boolean done = false;
for(int i=1; !done; i++) {
    // ...
    if(interrupted()) done = true;
}
```

In this example, the **for** loop continues to run until the **boolean** variable **done** is set to **true**. It does not test the value of **i**.

- Here is another interesting **for** loop variation. Either the initialization or the iteration expression or both may be absent, as in this next program:

```
// Parts of the for loop can be empty.
class ForVar {
    public static void main(String args[]) {
        int i;
        boolean done = false;
        i = 0;
        for( ; !done; ) {
            System.out.println("i is " + i);
            if(i == 10) done = true;
            i++;
        }
    }
}
```

Here, the initialization and iteration expressions have been moved out of the **for**. Thus, parts of the **for** are empty

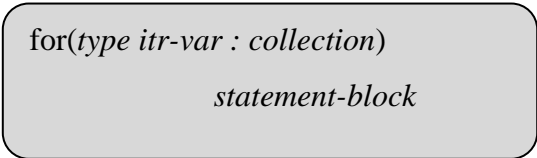
- Here is one more **for** loop variation. You can intentionally create an infinite loop (a loop that never terminates) if you leave all three parts of the **for** empty.
- For example:

```
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.
- The general form of the for-each version of the **for** is shown here:



```
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.

#### **Working:**

- 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.
- Thus, when iterating over arrays, *type* must be compatible with the base type of the array.

```
class ForEach {  
    public static void main(String args[]) {  
        int nums[] = { 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 };  
        int sum = 0;  
        for(int x : nums) {  
            sum += x;  
        }  
    }  
}
```



```
        System.out.println("Summation: " + sum);
    }
}
```

- With each pass through the loop, **x** is automatically given a value equal to the next element in **nums**. Thus, on the first iteration, **x** contains 1; on the second iteration, **x** contains 2; and so on.
- Not only is the syntax streamlined, but it also prevents boundary errors.

For example, this program sums only the first five elements of **nums**:

```
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) {
            sum += x;
            if(x == 5) break; // stop the loop when 5 is obtained
        }
        System.out.println("Summation of first 5 elements: " + sum);
    }
}
```

### Iterating Over Multidimensional Arrays

- The enhanced version of the **for** also works on multidimensional arrays.
- Remember, however, that in Java, multidimensional arrays consist of *arrays of arrays*. (For example, a two-dimensional array is an array of one-dimensional arrays.)

```
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) {
            sum += y;
        }
    }
    System.out.println("Summation: " + sum);
}
}

```

- In the program, pay special attention to this line:

```
for(int x[] : nums) {
```

- Notice how **x** is declared. It is a reference to a one-dimensional array of integers.
- This is necessary because each iteration of the **for** obtains the next *array* in **nums**, beginning with the array specified by **nums[0]**.
- The inner **for** loop then cycles through each of these arrays, displaying the values of each element.

### *Java program to search given key element*

```

class Search {
    public static void main(String args[]) {
        int nums[] = { 6, 8, 3, 7, 5, 6, 1, 4 };
        int val = 5;
        boolean found = false;
        // use for-each style for to search nums for val
        for(int x : nums) {
            if(x == val) {
                found = true;
                break;
            }
        }
        if(found)
            System.out.println("Value found!");
    }
}

```

```
    }  
}
```

### Nested Loops

- Like all other programming languages, Java allows loops to be nested.
- That is, one loop may be inside another. For example, here is a program that nests **for** loops:

```
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();  
        }  
    }  
}
```

### Jump Statements

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

#### The break statement

- The **break** statement transfers control out of the enclosing loop (for, while, do or switch statement).
- You use a **break** statement when you want to jump immediately to the statement following the enclosing control structure.
- You can also provide a loop with a label, and then use the label in your **break** statement.
- The label name is optional, and is usually only used when you wish to terminate the outermost loop in a series of nested loops.

#### **Syntax:**

```
break; // the unlabeled form  
break <label>; // the labeled form
```

**Example for break:**

```
public class Example {  
    public static void main(String[] args) {  
        System.out.println("Numbers 1 - 10");  
        for (int i = 1; ++i) {  
            if (i == 11)  
                break;  
            System.out.println(i + "\t");  
        }  
    }  
}
```

**Example for labeled break:**

```
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.");  
        }  
    }  
}
```

Running this program generates the following output:

Before the break.

This is after second block.

**The continue statement**

- A **continue** statement stops the iteration of a loop (while, do or for) and causes execution to resume at the top of the nearest enclosing loop.
- You use a **continue** statement when you do not want to execute the remaining statements in the loop, but you do not want to exit the loop itself.
- You can also provide a loop with a label and then use the label in your **continue** statement.
- The label name is optional, and is usually only used when you wish to return to the outermost loop in a series of nested loops.

**Syntax:**

```
continue; // the unlabeled form
continue <label>; // the labeled form
```

**Example for continue:**

```
public class Example {
    public static void main(String[] args) {
        System.out.println("Odd Numbers");
        for (int i = 1; i <= 10; ++i) {
            if (i % 2 == 0)
                continue;
            System.out.println(i + "\t");
        }
    }
}
```

**Example for labelled continue:**

```
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();  
}  
}
```

### The return statement

- The **return** statement exits from the current method, and control flow returns to where the method was invoked.

#### **Syntax:**

The **return** statement has two forms:

One that returns a value

**return val;**

One that doesn't return a value

**return;**

#### **Example:**

```
public class Example {  
    public static void main(String[] args) {  
        int res = sum(10, 20);  
        System.out.println(res);  
    }  
    private static int sum(int a, int b) {  
        return (a + b);  
    }  
}
```

## Module 3

### 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.

#### Class Fundamentals

- Class defines 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. Because an object is an instance of a class, you will often see 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.
- A class is declared by use of the **class** keyword

```
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.

### A Simple Class

- Here is a class called **Box** that defines three instance variables: **width**, **height**, and **depth**.

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

- As stated, a class defines a new type of data.
- 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

```
Box mybox = new Box(); // create a Box object called mybox
```

- After this statement executes, **mybox** will be an instance of **Box**.
- Thus, it will have “physical” reality.
- Thus, 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.

For example, to assign the **width** variable of **mybox** the value 100, you would use the following statement:

```
mybox.width = 100;
```



```
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);  
    }  
}
```

### **Declaring Objects**

- When you create a class, you are creating a new data type.
- However, obtaining objects of a class is a two-step process.
- First, you 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, you must acquire an actual, physical copy of the object and assign it to that variable. You 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.
- This reference is, more or less, the address in memory of the object allocated by **new**. This reference is then stored in the variable.
- Thus, in Java, all class objects must be dynamically allocated.

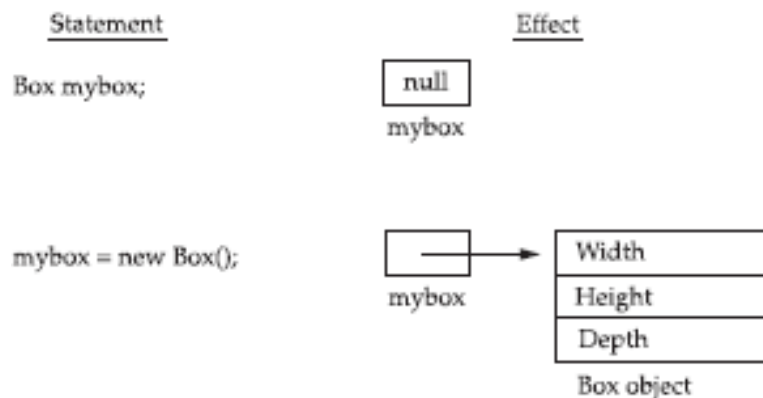
**Box mybox = new Box();**

- This statement combines the two steps just described. It can be rewritten like this to show each step more clearly:

**Box mybox; // declare reference to object**

**mybox = new Box(); // allocate a Box object**

- The first line declares **mybox** as a reference to an object of type **Box**.
- After this line executes, **mybox** contains the value **null**, which indicates that it does not yet point to an actual object.
- Any attempt to use **mybox** at this point will result in a compile-time error. The next line allocates an actual object and assigns a reference to it to **mybox**.
- After the second line executes, you can use **mybox** as if it were a **Box** object. But in reality, **mybox** simply holds the memory address of the actual **Box** object.



- Here, *class-var* is a variable of the class type being created. 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.
- It is important to understand that **new** allocates memory for an object during run time.
- The advantage of this approach is that your program can create as many or as few objects as it needs during the execution of your program.
- However, since memory is finite, it is possible that **new** will not be able to allocate memory for an object because insufficient memory exists.
- If this happens, a run-time exception will occur.
- A class creates a new data type that can be used to create objects.

- That is, a class creates a logical framework that defines the relationship between its members. When you declare an object of a class, you are creating an instance of that class.
- Thus, a class is a logical construct. An object has physical reality.

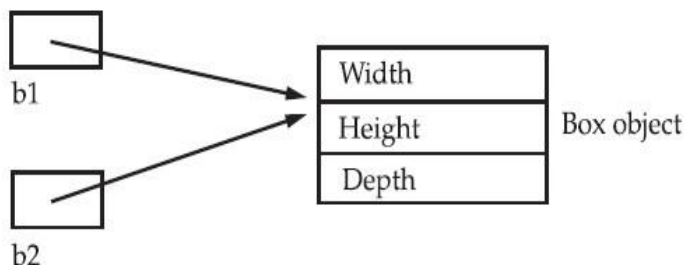
### Assigning Object Reference Variables

Object reference variables act differently when an assignment takes place.

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

```
Box b2 = b1;
```

This situation is depicted here:



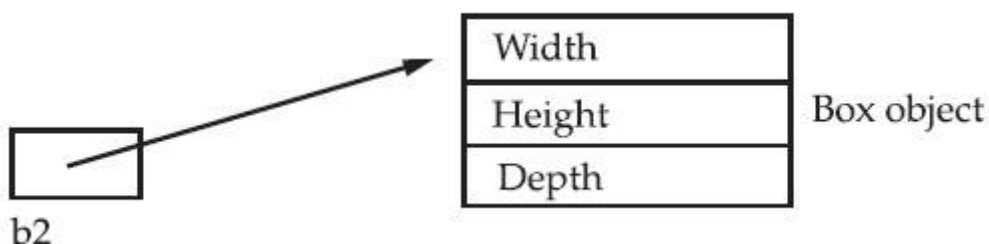
- After this fragment executes, **b1** and **b2** will both refer to the *same* object.
- The assignment of **b1** to **b2** did not allocate any memory or copy any part of the original object.
- It simply makes **b2** refer to the same object as does **b1**.
- Thus, any changes made to the object through **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.

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

```
// ...
```

```
b1 = null;
```

Here, **b1** has been set to **null**, but **b2** still points to the original object.



## Introducing methods

This is the general form of a method:

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

- Here, *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 is specified by *name*. This can be any legal identifier other than those already used by other items within the current scope.
- 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.
- Methods that have a return type other than **void** return a value to the calling routine using the following form of the **return** statement:

```
return value;
```

- Here, *value* is the value returned.

## Adding a Method to the Box Class

```
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();  
// assign values to mybox1's instance variables  
mybox1.width = 10;  
mybox1.height = 20;  
mybox1.depth = 15;  
// display volume of first box  
mybox1.volume();  
}  
}
```

This program generates the following output, which is the same as the previous version.

Volume is 3000.0

Volume is 162.0

Look closely at the following two lines of code:

```
mybox1.volume();
```

- The first line here invokes the **volume( )** method on **mybox1**.
- That is, it calls **volume( )** relative to the **mybox1** object, using the object's name followed by the dot operator.
- Thus, the call to **mybox1.volume( )** displays the volume of the box defined by **mybox1**,
- There is something very important to notice inside the **volume( )** method: the instance variables **width**, **height**, and **depth** are referred to directly, without preceding them with an object name or the dot operator.
- When a method uses an instance variable that is defined by its class, it does so directly, without explicit reference to an object and without use of the dot operator.
- This is easy to understand if you think about it. A method is always invoked relative to some object of its class. Once this invocation has occurred, the object is known.

### Returning a Value

- While the implementation of **volume( )** does move the computation of a box's volume inside the **Box** class where it belongs, it is not the best way to do it.

```
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();
        double vol;
        // assign values to mybox1's instance variables
        mybox1.width = 10;
        mybox1.height = 20;
        mybox1.depth = 15;
        // get volume of first box
        vol = mybox1.volume();
        System.out.println("Volume is " + vol);
    }
}
```

- As you can see, when **volume( )** is called, it is put on the right side of an assignment statement.
- On the left is a variable, in this case **vol**, that will receive the value returned by **volume( )**.
- Thus, after **vol = mybox1.volume();** executes, the value of **mybox1.volume( )** is 3,000 and this value then is stored in **vol**.

There are two important things to understand about returning values:

- The type of data returned by a method must be compatible with the return type specified by the method. For example, if the return type of some method is **boolean**, you could not return an integer.
- The variable receiving the value returned by a method (such as **vol**, in this case) must also be compatible with the return type specified for the method.

### Adding a Method That Takes Parameters

- While some methods don't need parameters, most do. Parameters allow a method to be generalized.
- That is, a parameterized method can operate on a variety of data and/or be used in a number of slightly different situations

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

- While this method does, indeed, return the value of 10 squared, its use is very limited.
- However, if you modify the method so that it takes a parameter, as shown next, then you can make **square( )** much more useful.

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

- Now, **square( )** will return the square of whatever value it is called with. That is, **square( )** is now a general-purpose method that can compute the square of any integer value, rather than just 10.

Here is an example:

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

- In the first call to **square( )**, the value 5 will be passed into parameter **i**.
- In the second call, **i** will receive the value 9.
- The third invocation passes the value of **y**, which is 2 in this example.
- As these examples show, **square( )** is able to return the square of whatever data it is passed
- A *parameter* is a variable defined by a method that receives a value when the method is called. For example, in **square( )**, **i** is a parameter.

- An *argument* is a value that is passed to a method when it is Invoked.
- For example, **square(100)** passes 100 as an argument. Inside **square( )**, the parameter **i** receives that value.
- Thus, a better approach to setting the dimensions of a box is to create a method that takes the dimensions of a box in its parameters and sets each instance variable appropriately.
- This concept is implemented by the following program:

// This program uses a parameterized method.

```
class Box {  
    double width;  
    double height;  
    double depth;  
    // compute and return volume  
    double volume() {  
        return width * height * depth;  
    }  
    // sets dimensions of box  
    void setDim(double w, double h, double d) {  
        width = w;  
        height = h;  
        depth = d;  
    }  
}  
  
class BoxDemo5 {  
    public static void main(String args[]) {  
        Box mybox1 = new Box();  
        Box mybox2 = new Box();  
        double vol;  
        // initialize each box  
        mybox1.setDim(10, 20, 15);  
        mybox2.setDim(3, 6, 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);  
    }  
}
```

- As you can see, the **setDim( )** method is used to set the dimensions of each box. For example, when `mybox1.setDim(10, 20, 15);` is executed, 10 is copied into parameter **w**, 20 is copied into **h**, and 15 is copied into **d**.
- Inside **setDim( )** the values of **w**, **h**, and **d** are then assigned to **width**, **height**, and **depth**, respectively.

## Constructors

- It can be tedious to initialize all of the variables in a class each time an instance is created.
- Even when you add convenience functions like **setDim( )**, it would be simpler and more concise to have all of the setup done at the time the object is first created.
- Because the requirement for initialization is so common, 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.
- Constructors look a little strange because they have no return type, not even **void**. This is because the implicit return type of a class' constructor is the class type itself.
- It is the constructor's job to initialize the internal state of an object so that the code creating an instance will have a fully initialized, usable object immediately.

```
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);
    }
}
```

When this program is run, it generates the following results:

Constructing Box

Constructing Box

Volume is 1000.0

Volume is 1000.0

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

- Now you can understand why the parentheses are needed after the class name. What is actually happening is that the constructor for the class is being called. Thus, in the line

Box mybox1 = new Box();

- **new Box( )** is calling the **Box( )** constructor **new** .
- When you do not explicitly define a constructor for a class, then Java creates a default constructor for the class

### Parameterized Constructors

- While the **Box( )** constructor in the preceding example does initialize a **Box** object, it is not very useful—all boxes have the same dimensions.
- What is needed is a way to construct **Box** objects of various dimensions.
- The easy solution is to add parameters to the constructor

```
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);
}
}
```

The output from this program is shown here:

Volume is 3000.0

Volume is 162.0

### **The this keyword**

- Sometimes a method will need to refer to the object that invoked it.
- To allow this, Java defines the **this** keyword. **this** can be used inside any method to refer to the *current* object

```
Box(double w, double h, double d) {
    this.width = w;
    this.height = h;
    this.depth = d;
}
```

Uses of this:

- To overcome shadowing or instance variable hiding.
- To call an overload constructor

### **Instance Variable Hiding**

- It is illegal in Java to declare two local variables with the same name inside the same or enclosing scopes.

- Interestingly, you can have local variables, including formal parameters to methods, which overlap with the names of the class' instance variables.
- However, when a local variable has the same name as an instance variable, the local variable *hides* the instance variable.

// Use this to resolve name-space collisions.

```
Box(double width, double height, double depth) {  
    this.width = width;  
    this.height = height;  
    this.depth = depth;  
}
```

**NOTE:** The use of **this** in such a context can sometimes be confusing, and some programmers are careful not to use local variables and formal parameter names that hide instance variables.

### **Garbage Collection**

- Since objects are dynamically allocated by using the **new** operator, you might be wondering how such objects are destroyed and their memory released for later reallocation.
- Java takes a different approach; it handles deallocation for you automatically.
- The technique that accomplishes this is called *garbage collection*.
- It works like this: when no references to an object exist, that object is assumed to be no longer needed, and the memory occupied by the object can be reclaimed.
- There is no explicit need to destroy objects as in C++.
- Garbage collection only occurs sporadically (if at all) during the execution of your program.
- It will not occur simply because one or more objects exist that are no longer used.

### **The finalize( ) Method**

- Sometimes an object will need to perform some action when it is destroyed. For example, if an object is holding some non-Java resource such as a file handle or character font, then you might want to make sure these resources are freed before an object is destroyed.
- To handle such situations, Java provides a mechanism called *finalization*.

- By using finalization, you can define specific actions that will occur when an object is just about to be reclaimed by the garbage collector.

The **finalize()** method has this general form:

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

- Here, the keyword **protected** is a specifier that prevents access to **finalize()** by code defined outside its class.
- It is important to understand that **finalize()** is only called just prior to garbage collection.
- It is not called when an object goes out-of-scope, for example. This means that you cannot know when—or even if—**finalize()** will be executed.
- Therefore, your program should provide other means of releasing system resources, etc., used by the object.
- It must not rely on **finalize()** for normal program operation.

### A Stack Class

```
class Stack {  
    int stck[] = new int[10];  
    int tos;  
    // Initialize top-of-stack  
    Stack() {  
        top = -1;  
    }  
    // Push an item onto the stack  
    void push(int item) {  
        if(top==9)  
            System.out.println("Stack is full.");  
        else  
            stck[++top] = item;  
    }  
    // Pop an item from the stack
```

```
int pop() {
    if(top < 0) {
        System.out.println("Stack underflow.");
        return 0;
    }
    else
        return stck[top--];
}
}

class TestStack {
    public static void main(String args[]) {
        Stack mystack1 = new Stack();
        Stack mystack2 = new Stack();
        // push some numbers onto the stack
        for(int i=0; i<10; i++) mystack1.push(i);
        for(int i=10; i<20; i++) mystack2.push(i);
        // pop those numbers off the stack
        System.out.println("Stack in mystack1:");
        for(int i=0; i<10; i++)
            System.out.println(mystack1.pop());
        System.out.println("Stack in mystack2:");
        for(int i=0; i<10; i++)
            System.out.println(mystack2.pop());
    }
}
```

This program generates the following output:

Stack in mystack1:

9  
8  
7  
6  
5  
4  
3

2

1

0

Stack in mystack2:

19

18

17

16

15

14

13

12

11

10

### **Overloading Methods**

- In Java it is possible to define two or more methods within the same class that share the same name, as long as their parameter declarations are different.
- When this is the case, 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 an overloaded method is invoked, Java uses the type and/or number of arguments as its guide to determine which version of the overloaded method to actually call.
- Thus, overloaded methods must differ in the type and/or number of their parameters.
- While overloaded methods may have different return types, the return type alone is insufficient to distinguish two versions of a method.

```
class OverloadDemo {  
    void test() {  
        System.out.println("No parameters");  
    }  
    // Overload test for one integer parameter.
```



```
void test(int a) {
    System.out.println("a: " + a);
}
// Overload test for two integer parameters.
void test(int a, int b) {
    System.out.println("a and b: " + a + " " + b);
}
// overload test for a double parameter
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;
        // call all versions of test()
        ob.test();
        ob.test(10);
        ob.test(10, 20);
        result = ob.test(123.25);
        System.out.println("Result of ob.test(123.25): " + result);
    }
}
```

This program generates the following output:

No parameters

a: 10

a and b: 10 20

double a: 123.25

Result of ob.test(123.25): 15190.5625

- When an overloaded method is called, Java looks for a match between the arguments used to call the method and the method's parameters.

- However, this match need not always be exact. In some cases, Java's automatic type conversions can play a role in overload resolution.

For example, consider the following program:

// Automatic type conversions apply to overloading.

```
class OverloadDemo {
    void test() {
        System.out.println("No parameters");
    }
    // Overload test for two integer parameters.
    void test(int a, int b) {
        System.out.println("a and b: " + a + " " + b);
    }
    // overload test for a double parameter
    void test(double a) {
        System.out.println("Inside test(double) a: " + a);
    }
}

class Overload {
    public static void main(String args[]) {
        OverloadDemo ob = new OverloadDemo();
        int i = 88;
        ob.test();
        ob.test(10, 20);
        ob.test(i); // this will invoke test(double)
        ob.test(123.2); // this will invoke test(double)
    }
}
```

This program generates the following output:

No parameters

a and b: 10 20

Inside test(double) a: 88

Inside test(double) a: 123.2

- When **test()** is called with an integer argument inside **Overload**, no matching method is found.
- However, Java can automatically convert an integer into a **double**, and this conversion can be used to resolve the call.
- Therefore, after **test(int)** is not found, Java elevates **i** to **double** and then calls **test(double)**.
- Of course, if **test(int)** had been defined, it would have been called instead. Java will employ its automatic type conversions only if no exact match is found.
- Method overloading supports polymorphism because it is one way that Java implements the “one interface, multiple methods” paradigm.
- When you overload a method, each version of that method can perform any activity you desire.
- There is no rule stating that overloaded methods must relate to one another.
- However, from a stylistic point of view, method overloading implies a relationship. Thus, while you can use the same name to overload unrelated methods, you should not.

### Overloading Constructors

- In addition to overloading normal methods, you can also overload constructor methods. In fact, for most real-world classes that you create, overloaded constructors will be the norm, not the exception.
- To understand why, let's return to the **Box** class developed in the preceding chapter. Following is the latest version of **Box**:

```
class Box {  
    double width;  
    double height;  
    double depth;  
    // constructor used when all dimensions specified  
    Box(double w, double h, double d) {  
        width = w;  
        height = h;  
        depth = d;  
    }  
}
```

```
// constructor used when no dimensions specified
Box() {
    width = -1; // use -1 to indicate
    height = -1; // an uninitialized
    depth = -1; // box
}

// constructor used when cube is created
Box(double len) {
    width = height = depth = len;
}

// compute and return volume
double volume() {
    return width * height * depth;
}
}

class OverloadCons {
    public static void main(String args[]) {
        // create boxes using the various constructors
        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);
        // get volume of second box
        vol = mybox2.volume();
        System.out.println("Volume of mybox2 is " + vol);
        // get volume of cube
        vol = mycube.volume();
        System.out.println("Volume of mycube is " + vol);
    }
}
```

The output produced by this program is shown here:

Volume of mybox1 is 3000.0

Volume of mybox2 is -1.0

Volume of mycube is 343.0

### Using Objects as Parameters

- So far, we have only been using simple types as parameters to methods. However, it is both correct and common to pass objects to methods.
- For example, consider the following short program:

// Objects may be passed to methods.

```
class Test {
    int a, b;
    Test(int i, int j) {
        a = i;
        b = j;
    }
    // return true if o is equal to the invoking object
    boolean equals(Test o) {
        if(o.a == a && o.b == b) return true;
        else return false;
    }
}

class PassOb {
    public static void main(String args[]) {
        Test ob1 = new Test(100, 22);
        Test ob2 = new Test(100, 22);
        Test ob3 = new Test(-1, -1);
        System.out.println("ob1 == ob2: " + ob1.equals(ob2));
        System.out.println("ob1 == ob3: " + ob1.equals(ob3));
    }
}
```

This program generates the following output:

ob1 == ob2: true

ob1 == ob3: false

- As you can see, the **equals( )** method inside **Test** compares two objects for equality and returns the result.
- That is, it compares the invoking object with the one that it is passed.
- If they contain the same values, then the method returns **true**. Otherwise, it returns **false**. Notice that the parameter **o** in **equals( )** specifies **Test** as its type.
- Although **Test** is a class type created by the program, it is used in just the same way as Java's built-in types.
- One of the most common uses of object parameters involves constructors. Frequently, you will want to construct a new object so that it is initially the same as some existing object.
- To do this, you must define a constructor that takes an object of its class as a parameter

### A Closer Look at Argument Passing

- In general, there are two ways that a computer language can pass an argument to a subroutine.
- The first way is *call-by-value*. This approach copies the *value* of an argument into the formal parameter of the subroutine. Therefore, changes made to the parameter of the subroutine have no 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 parameter.
- Inside the subroutine, this reference is used to access the actual argument specified in the call. This means that changes made to the parameter will affect the argument used to call the subroutine.
- As you will see, Java uses both approaches, depending upon what is passed.
- In Java, when you pass a primitive type to a method, it is passed by value. Thus, what occurs to the parameter that receives the argument has no effect outside the method.

For example, consider the following program:

// Primitive types are passed by value.

```
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);
    }
}

```

The output from this program is shown here:

a and b before call: 15 20

a and b after call: 15 20

- When you pass an object to a method, the situation changes dramatically, because objects are passed by what is effectively call-by-reference.
- Thus, when you pass 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 to the object inside the method *do* affect the object used as an argument.
- For example, consider the following program:

// Objects are passed by reference.

```

class Test {
    int a, b;
    Test(int i, int j) {
        a = i;
        b = j;
    }
}
// pass an object
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);
    }
}

```

This program generates the following output:

ob.a and ob.b before call: 15 20

ob.a and ob.b after call: 30 10

### **Returning Objects**

- A method can return any type of data, including class types that you create. For example, in the following program, the **incrByTen( )** method returns an object in which the value of **a** is ten greater than it is in the invoking object.

// Returning an object.

```

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);  
}  
}
```

The output generated by this program is shown here:

ob1.a: 2

ob2.a: 12

ob2.a after second increase: 22

### Recursion

- Java supports *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. The factorial of a number  $N$  is the product of all the whole numbers between 1 and  $N$ .
- For example, 3 factorial is  $1 \times 2 \times 3$ , or 6. Here is how a factorial can be computed by use of a recursive method:

// A simple example of recursion.

```
class Factorial {  
    // this is a recursive method  
    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));  
    }  
}
```

The output from this program is shown here:

Factorial of 3 is 6

Factorial of 4 is 24

Factorial of 5 is 120

- When a method calls itself, new local variables and parameters are allocated storage on the stack, and the method code is executed with these new variables from the start.
- As each recursive call returns, the old local variables and parameters are removed from the stack, and execution resumes at the point of the call inside the method.
- Recursive methods could be said to “telescope” out and back.
- Recursive versions of many routines may execute a bit more slowly than the iterative equivalent because of the added overhead of the additional function calls.
- Many recursive calls to a method could cause a stack overrun. Because storage for parameters and local variables is on the stack and each new call creates a new copy of these variables, it is possible that the stack could be exhausted.
- If this occurs, the Java run-time system will cause an exception.

**The main advantage to recursive methods is that they can be used to create clearer and simpler versions of several algorithms than can their iterative relatives.**

Here is one more example of recursion. The recursive method **printArray( )** prints the first **i** elements in the array **values**.

// Another example that uses recursion.

```
class RecTest {  
    int values[];
```

```
        RecTest(int i) {
            values = new int[i];
        }
// display array -- recursively
    void printArray(int i) {
        if(i==0) return;
        else printArray(i-1);
        System.out.println "[" + (i-1) + " ] " + values[i-1]);
    }
}
class Recursion2 {
    public static void main(String args[]) {
        RecTest ob = new RecTest(10);
        int i;
        for(i=0; i<10; i++) ob.values[i] = i;
        ob.printArray(10);
    }
}
```

This program generates the following output:

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

### **Introducing Access Control**

- encapsulation provides another important attribute: *access control*.

- Through encapsulation, you can control what parts of a program can access the members of a class.
- By controlling access, you can prevent misuse. For example, allowing access to data only through a welldefined set of methods, you can prevent the misuse of that data.
- Thus, when correctly implemented, a class creates a “black box” which may be used, but the inner workings of which are not open to tampering
- Java’s access specifiers are **public**, **private**, and **protected**.
- Java also defines a default access level. **protected** applies only when inheritance is involved 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.
- Now you can understand why **main( )** has always been preceded by the **public** specifier. It is called by code that is outside the program—that is, by the Java run-time system.
- When no access specifier is used, then by default the member of a class is public within its own package, but cannot be accessed outside of its package.
- An access specifier precedes the rest of a member’s type specification. That is, it must begin a member’s declaration statement.
- Here is an example:

```
        public int i;  
        private double j;  
        private int myMethod(int a, char b) { // ...  
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());
    }
}

```

- As you can see, inside the **Test** class, **a** uses default access, which for this example is the same as specifying **public**. **b** is explicitly specified as **public**.
- Member **c** is given private access. This means that it cannot be accessed by code outside of its class.
- So, inside the **AccessTest** class, **c** cannot be used directly. It must be accessed through its public methods: **setc( )** and **getc( )**.
- If you were to remove the comment symbol from the beginning of the following line,
 

```
// ob.c = 100; // Error!
```

### Understanding static

- There will be times when you will want to define a class member that will be used independently of any object of that class.
- Normally, a class member must be accessed only in conjunction with an object of its class.
- However, it is possible to create a member that can be used by itself, without reference to a specific instance.
- To create such a member, precede its declaration with the keyword **static**.

- When a member is declared **static**, it can be accessed before any objects of its class are created, and without reference to any object.
- You can declare both methods and variables to be **static**.
- The most common example of a **static** member is **main( )**. **main( )** is declared as **static** because it must be called before any objects exist.
- **Instance variables declared as static are, essentially, global variables.**
- **When objects of its class are declared, no copy of a static variable is made. Instead, all instances of the class share the same static variable.**
- Methods declared as **static** have several restrictions:
  - They can only call other **static** methods.
  - They must only access **static** data.
  - They cannot refer to **this** or **super** in any way

The following example shows a class that has a **static** method, some **static** variables, and a **static** initialization block:

// Demonstrate static variables, methods, and blocks.

```
class UseStatic {  
    static int a = 3;  
    static int b;  
    static void meth(int x) {  
        System.out.println("x = " + x);  
        System.out.println("a = " + a);  
        System.out.println("b = " + b);  
    }  
    static {  
        System.out.println("Static block initialized.");  
        b = a * 4;  
    }  
    public static void main(String args[]) {  
        meth(42);  
    }  
}
```

- As soon as the **UseStatic** class is loaded, all of the **static** statements are run.

- First, **a** is set to **3**,
- then the **static** block executes, which prints a message and then initializes **b** to **a \* 4** or **12**.
- Then **main()** is called, which calls **meth()**, passing **42** to **x**.
- The three **println()** statements refer to the two **static** variables **a** and **b**, as well as to the local variable **x**.

Here is the output of the program:

**Static block initialized.**

**x = 42**

**a = 3**

**b = 12**

- Outside of the class in which they are defined, **static** methods and variables can be used independently of any object.
- To do so, you need only specify the name of their class followed by the dot operator.
- For example, if you wish to call a **static** method from outside its class, you can do so using the following general form:

*classname.method()*

- Here, *classname* is the name of the class in which the **static** method is declared. As you can see, this format is similar to that used to call non-**static** methods through object-reference variables.
- A **static** variable can be accessed in the same way—by use of the dot operator on the name of the class. This is how Java implements a controlled version of global methods and global variables.
- Here is an example. Inside **main()**, the **static** method **callme()** and the **static** variable **b** are accessed through their class name **StaticDemo**.

```
class StaticDemo {
    static int a = 42;
    static int b = 99;
    static void callme() {
        System.out.println("a = " + a);
    }
}
```

```
class StaticByName {  
    public static void main(String args[]) {  
        StaticDemo.callme();  
        System.out.println("b = " + StaticDemo.b);  
    }  
}
```

Here is the output of this program:

a = 42

b = 99

### Introducing final

- A variable can be declared as **final**. Doing so prevents its contents from being modified. This means that you must initialize a **final** variable when it is declared.
- For example:

```
final int FILE_NEW = 1;  
final int FILE_OPEN = 2;  
final int FILE_SAVE = 3;  
final int FILE_SAVEAS = 4;  
final int FILE_QUIT = 5;
```

- Subsequent parts of your program can now use **FILE\_OPEN**, etc., as if they were constants, without fear that a value has been changed.
- It is a common coding convention to choose all uppercase identifiers for **final** variables.
- Variables declared as **final** do not occupy memory on a per-instance basis. Thus, a **final** variable is essentially a constant.
- The keyword **final** can also be applied to methods, but its meaning is substantially different than when it is applied to variables.

### Arrays Revisited

- Now that you know about classes, an important point can be made about arrays: they are implemented as objects.



- Because of this, there is a special array attribute that you will want to take advantage of. Specifically, the size of an array—that is, the number of elements that an array can hold—is found in its **length** instance variable.
- All arrays have this variable, and it will always hold the size of the array.
- Here is a program that demonstrates this property:

// This program demonstrates the length array member.

```
class Length {  
    public static void main(String args[]) {  
        int a1[] = new int[10];  
        int a2[] = {3, 5, 7, 1, 8, 99, 44, -10};  
        int a3[] = {4, 3, 2, 1};  
        System.out.println("length of a1 is " + a1.length);  
        System.out.println("length of a2 is " + a2.length);  
        System.out.println("length of a3 is " + a3.length);  
    }  
}
```

This program displays the following output:

length of a1 is 10

length of a2 is 8

length of a3 is 4

- You can put the **length** member to good use in many situations. For example, here is an improved version of the **Stack** class. As you might recall, the earlier versions of this class always created a ten-element stack.
- The following version lets you create stacks of any size. The value of **stck.length** is used to prevent the stack from overflowing.

// Improved Stack class that uses the length array member.

```
class Stack {  
    private int stck[];  
    private int tos;  
    // allocate and initialize stack  
    Stack(int size) {  
        stck = new int[size];  
    }  
}
```

```
        tos = -1;
    }
    // Push an item onto the stack
    void push(int item) {
        if(tos==stck.length-1) // use length member
            System.out.println("Stack is full.");
        else
            stck[++tos] = item;
    }
    // Pop an item from the stack
    int pop() {
        if(tos < 0) {
            System.out.println("Stack underflow.");
            return 0;
        }
        else
            return stck[tos--];
    }
}

class TestStack2 {
    public static void main(String args[]) {
        Stack mystack1 = new Stack(5);
        Stack mystack2 = new Stack(8);
        // push some numbers onto the stack
        for(int i=0; i<5; i++) mystack1.push(i);
        for(int i=0; i<8; i++) mystack2.push(i);
        // pop those numbers off the stack
        System.out.println("Stack in mystack1:");
        for(int i=0; i<5; i++)
            System.out.println(mystack1.pop());
        System.out.println("Stack in mystack2:");
        for(int i=0; i<8; i++)
            System.out.println(mystack2.pop());
    }
}
```

}

## **Module 4**

### **Packages and Interfaces, Exception Handling**

#### **Packages and Interfaces:**

- Packages,
- Access Protection,
- Importing Packages,
- Interfaces,

#### **Exception Handling:**

- Exception-Handling Fundamentals,
- Exception Types,
- Uncaught Exceptions,
- Using try and catch,
- Multiple catch Clauses,
- Nested try Statements,
- throw, throws, finally,
- Java's Built-in Exceptions,
- Creating Your Own Exception Subclasses,
- Chained Exceptions,
- Using Exceptions.

## Packages and Interfaces:

### Packages:

- The name of each example class was taken from the same name space. This means that a unique name had to be used for each class to avoid name collisions.
- **Java provides a mechanism for partitioning the class name space into more manageable chunks. This mechanism is the package.**
- The package is both a naming and a visibility control mechanism. You can define classes inside a package that are not accessible by code outside that package. You can also define class members that are only exposed to other members of the same package.

### Defining a Package:

- To create a package is quite easy: simply include a **package** command as the first statement in a Java source file.
- Any classes declared within that file will belong to the specified package.
- The **package** statement defines a name space in which classes are stored. If you omit the **package** statement, the class names are put into the default package, which has no name.

This is the general form of the **package** statement:

**package *pkg*;**

Here, *pkg* is the name of the package.

For example, the following statement creates a package called **MyPackage**.

```
package MyPackage;
```

- Java uses file system directories to store packages. For example, the **.class** files for any classes you declare to be part of **MyPackage** must be stored in a directory called **MyPackage**.
- Remember that case is significant, and the directory name must match the package name exactly.
- More than one file can include the same **package** statement. The **package** statement simply specifies to which package the classes defined in a file belong.

- You can create a hierarchy of packages. To do so, simply separate each package name from the one above it by use of a period.

The general form of a multileveled package statement is shown here:

**package *pkg1*[.*pkg2*[.*pkg3*]];**

- A package hierarchy must be reflected in the file system of your Java development system.
- For example, a package declared as

**package java.awt.image;**

### Finding Packages and CLASSPATH:

- Packages are mirrored by directories.
- The Java run-time system know where to look for packages that you create
  - ➔ First, by default, the Java run-time system uses the current working directory as its starting point. Thus, if your package is in a subdirectory of the current directory, it will be found.
  - ➔ Second, you can specify a directory path or paths by setting the **CLASSPATH** environmental variable.
  - ➔ Third, you can use the **-classpath** option with **java** and **javac** to specify the path to your classes.
- For example, consider the following package specification:

**package MyPack**

can be executed from a directory immediately above **MyPack**, or the **CLASSPATH** must be set to include the path to **MyPack**, or the **-classpath** option must specify the path to **MyPack** when the program is run via **java**.

- When the second two options are used, the class path *must not* include **MyPack**, itself.
- It must simply specify the *path to MyPack*. For example, in a Windows environment, if the path to **MyPack** is

**C:\MyPrograms\Java\MyPack**

- Then the class path to **MyPack** is

**C:\MyPrograms\Java**

**A Short Package Example:**

```
package MyPack;

class Balance {
    String name;
    double bal;
    Balance(String n, double b) {
        name = n;
        bal = b;
    }
    void show() {
        if(bal<0)
            System.out.print("--> ");
        System.out.println(name + ": $" + bal);
    }
}

class AccountBalance {
    public static void main(String args[]) {
        Balance current[] = new Balance[3];
        current[1] = new Balance("Will Tell", 157.02);
        current[2] = new Balance("Tom Jackson", -12.33);
        for(int i=0; i<3; i++) current[i].show();
    }
}
```

- Call this file **AccountBalance.java** and put it in a directory called **MyPack**.
- Next, compile the file. Make sure that the resulting **.class** file is also in the **MyPack** directory.
- Then, try executing the **AccountBalance** class, using the following command line:

**java MyPack.AccountBalance**

**Access Protection:**

- Packages add another dimension to access control.

- Java provides many levels of protection to allow fine-grained control over the visibility of variables and methods within classes, subclasses, and packages. Classes and packages are both means of encapsulating and containing the name space and scope of variables and methods.
- Packages act as containers for classes and other subordinate packages.
- Classes act as containers for data and code. The class is Java's smallest unit of abstraction.
- Because of the interplay between classes and packages, Java addresses four categories of visibility for class members:
  - Subclasses in the same package
  - Non-subclasses in the same package
  - Subclasses in different packages
  - Classes that are neither in the same package nor subclasses
- The three access specifiers, **private**, **public**, and **protected**, provide a variety of ways to produce the many levels of access required by these categories.
- Table sums up the interactions.

	<b>Private</b>	<b>No Modifier</b>	<b>Protected</b>	<b>Public</b>
Same class	Yes	Yes	Yes	Yes
Same package subclass	No	Yes	Yes	Yes
Same package non-subclass	No	Yes	Yes	Yes
Different package subclass	No	No	Yes	Yes
Different package non-subclass	No	No	No	Yes

### Importing Packages:

- Given that packages exist and are a good mechanism for compartmentalizing diverse classes from each other, it is easy to see why all of the built-in Java classes are stored in packages.
- There are no core Java classes in the unnamed default package; all of the standard classes are stored in some named package.



- Java includes the **import** statement to bring certain classes, or entire packages, into visibility.
- Once imported, a class can be referred to directly, using only its name. The **import** statement is a convenience to the programmer and is not technically needed to write a complete Java program.
- If you are going to refer to a few dozen classes in your application, however, the **import** statement will save a lot of typing.

This is the general form of the **import** statement:

**import *pkg1*[*.pkg2*].(*classname*|\*);**

Here, *pkg1* is the name of a top-level package, and *pkg2* is the name of a subordinate package inside the outer package separated by a dot (.).

- There is no practical limit on the depth of a package hierarchy, except that imposed by the file system.
- Finally, you specify either an explicit *classname* or a star (\*), which indicates that the Java compiler should import the entire package.
- This code fragment shows both forms in use:

```
import java.util.Date;  
import java.io.*;
```

- All of the standard Java classes included with Java are stored in a package called **java**.
- The basic language functions are stored in a package inside of the **java** package called **java.lang**. This is equivalent to the following line being at the top of all of your programs:

**import java.lang.\*;**

- If a class with the same name exists in two different packages that you import using the star form, the compiler will remain silent, unless you try to use one of the classes.
- In that case, you will get a compile-time error and have to explicitly name the class specifying its package.
- It must be emphasized that the **import** statement is optional. Any place you use a class name, you can use its *fully qualified name*, which includes its full package hierarchy.

- For example, this fragment uses an import statement:

```
import java.util.*;  
class MyDate extends Date {  
}
```

The same example without the **import** statement looks like this:

```
class MyDate extends java.util.Date {  
}
```

In this version, **Date** is fully-qualified.

### Interfaces:

- Using the keyword **interface**, you can fully abstract a class' interface from its implementation.
- That is, using **interface**, you can specify what a class must do, but not how it does it. Interfaces are syntactically similar to classes, but they lack instance variables, and their methods are declared without any body.
- In practice, this means that you can define interfaces that don't make assumptions about how they are implemented.
- Once it is defined, any number of classes can implement an **interface**. Also, one class can implement any number of interfaces.
- To implement an interface, a class must create the complete set of methods defined by the interface. However, each class is free to determine the details of its own implementation.
- By providing the **interface** keyword, Java allows you to fully utilize the "one interface, multiple methods" aspect of polymorphism.
- Interfaces are designed to support dynamic method resolution at run time.
- Normally, in order for a method to be called from one class to another, both classes need to be present at compile time so the Java compiler can check to ensure that the method signatures are compatible.

### Defining an Interface:

- An interface is defined much like a class.
- This is the general form of an interface:

```
access interface name {  
    return-type method-name1(parameter-list);  
    return-type method-name2(parameter-list);  
    type final-varname1 = value;  
    type final-varname2 = value;  
    // ...  
    return-type method-nameN(parameter-list);  
    type final-varnameN = value;  
}
```

- When no access specifier is included, then default access results, and the interface is only available to other members of the package in which it is declared.
- When it is declared as **public**, the interface can be used by any other code.
- In this case, the interface must be the only public interface declared in the file, and the file must have the same name as the interface.
- *name* is the name of the interface, and can be any valid identifier.
- Notice that the methods that are declared have no bodies. They end with a semicolon after the parameter list.
- They are, essentially, abstract methods; there can be no default implementation of any method specified within an interface.
- Each class that includes an interface must implement all of the methods.
- Variables can be declared inside of interface declarations.
- They are implicitly **final** and **static**, meaning they cannot be changed by the implementing class. They must also be initialized.
- All methods and variables are implicitly **public**.

Here is an example of an interface definition. It declares a simple interface that contains one method called **callback()** that takes a single integer parameter.

```
interface Callback {  
    void callback(int param);  
}
```

### Implementing Interfaces:

- Once an **interface** has been defined, one or more classes can implement that interface.

- To implement an interface, include the **implements** clause in a class definition, and then create the methods defined by the interface.
- The general form of a class that includes the **implements** clause looks like this:

```
class classname [extends superclass] [implements interface [,interface...]] {  
    // class-body  
}
```

- The methods that implement an interface must be declared **public**.
- Also, the type signature of the implementing method must match exactly the type signature specified in the **interface** definition

Here is a small example class that implements the **Callback** interface shown earlier.

```
class Client implements Callback {  
    // Implement Callback's interface  
    public void callback(int p) {  
        System.out.println("callback called with " + p);  
    }  
}
```

- Notice that **callback( )** is declared using the **public** access specifier.
- It is both permissible and common for classes that implement interfaces to define additional members of their own.
- For example, the following version of **Client** implements **callback( )** and adds the method **nonInterfaceMeth( )**:

```
class Client implements Callback {  
    // Implement Callback's interface  
    public void callback(int p) {  
        System.out.println("callback called with " + p);  
    }  
    void nonInterfaceMeth() {  
        System.out.println("Classes that implement interfaces " +  
            "may also define other members, too.");  
    }  
}
```

**Accessing Implementations Through Interface References:**

- You can declare variables as object references that use an interface rather than a class type.
- Any instance of any class that implements the declared interface can be referred to by such a variable.
- When you call a method through one of these references, the correct version will be called based on the actual instance of the interface being referred to. This is one of the key features of interfaces
- The method to be executed is looked up dynamically at run time, allowing classes to be created later than the code which calls methods on them.
- The calling code can dispatch through an interface without having to know anything about the “callee.”
- The following example calls the **callback( )** method via an interface reference variable:

```
class TestIface {  
    public static void main(String args[]) {  
        Callback c = new Client();  
        c.callback(42);  
    }  
}
```

The output of this program is shown here:

callback called with 42

- While the preceding example shows, mechanically, how an interface reference variable can access an implementation object, it does not demonstrate the polymorphic power of such a reference.
- To sample this usage, first create the second implementation of **Callback**, shown here:

// Another implementation of Callback.

```
class AnotherClient implements Callback {  
    // Implement Callback's interface  
    public void callback(int p) {  
        System.out.println("Another version of callback");  
        System.out.println("p squared is " + (p*p));  
    }  
}
```

```

    }
}

```

Now, try the following class:

```

class TestIface2 {
    public static void main(String args[]) {
        Callback c = new Client();
        AnotherClient ob = new AnotherClient();
        c.callback(42);
        c = ob; // c now refers to AnotherClient object
        c.callback(42);
    }
}

```

The output from this program is shown here:

callback called with 42

Another version of callback

p squared is 1764

### Partial Implementations:

- If a class includes an interface but does not fully implement the methods defined by that interface, then that class must be declared as **abstract**.
- For example:

```

abstract class Incomplete implements Callback {
    int a, b;
    void show() {
        System.out.println(a + " " + b);
    }
    // ...
}

```

- Here, the class **Incomplete** does not implement **callback( )** and must be declared as **abstract**.
- Any class that inherits **Incomplete** must implement **callback( )** or be declared **abstract** itself.

**Nested Interfaces:**

- An interface can be declared a member of a class or another interface. Such an interface is called a *member interface* or a *nested interface*.
- A nested interface can be declared as **public**, **private**, or **protected**.
- This differs from a top-level interface, which must either be declared as **public** or use the default access level, as previously described.
- When a nested interface is used outside of its enclosing scope, it must be qualified by the name of the class or interface of which it is a member.
- Thus, outside of the class or interface in which a nested interface is declared, its name must be fully qualified.

Here is an example that demonstrates a nested interface:

```
class A {  
    // this is a nested interface  
    public interface NestedIF {  
        boolean isNotNegative(int x);  
    }  
}  
  
// B implements the nested interface.  
class B implements A.NestedIF {  
    public boolean isNotNegative(int x) {  
        return x < 0 ? false : true;  
    }  
}  
  
class NestedIFDemo {  
    public static void main(String args[]) {  
        // use a nested interface reference  
        A.NestedIF nif = new B();  
        if(nif.isNotNegative(10))  
            System.out.println("10 is not negative");  
        if(nif.isNotNegative(-12))  
            System.out.println("this won't be displayed");  
    }  
}
```

```
}
```

- Notice that **A** defines a member interface called **NestedIF** and that it is declared **public**.
- Next, **B** implements the nested interface by specifying implements **A.NestedIF**
- Notice that the name is fully qualified by the enclosing class' name.
- Inside the **main( )** method, an **A.NestedIF** reference called **nif** is created, and it is assigned a reference to a **B** object. Because **B** implements **A.NestedIF**, this is legal.

### Applying Interfaces:

- To understand the power of interfaces, let's look at a more practical example. In earlier chapters, you developed a class called **Stack** that implemented a simple fixed-size stack. However, there are many ways to implement a stack.
- First, here is the interface that defines an integer stack. Put this in a file called **IntStack.java**.

This interface will be used by both stack implementations.

// Define an integer stack interface.

```
interface IntStack {
    void push(int item); // store an item
    int pop(); // retrieve an item
}

class FixedStack implements IntStack {
    private int stck[];
    private int tos;
    // allocate and initialize stack
    FixedStack(int size) {
        stck = new int[size];
        tos = -1;
    }
    // Push an item onto the stack
    public void push(int item) {
        if(tos==stck.length-1) // use length member
            System.out.println("Stack is full.");
    }
}
```



```
        else
            stck[++tos] = item;
    }
    // Pop an item from the stack
    public int pop() {
        if(tos < 0) {
            System.out.println("Stack underflow.");
            return 0;
        }
        else
            return stck[tos--];
    }
}

class IFTest {
    public static void main(String args[]) {
        FixedStack mystack1 = new FixedStack(5);
        FixedStack mystack2 = new FixedStack(8);
        // push some numbers onto the stack
        for(int i=0; i<5; i++) mystack1.push(i);
        for(int i=0; i<8; i++) mystack2.push(i);
        // pop those numbers off the stack
        System.out.println("Stack in mystack1:");
        for(int i=0; i<5; i++)
            System.out.println(mystack1.pop());
        System.out.println("Stack in mystack2:");
        for(int i=0; i<8; i++)
            System.out.println(mystack2.pop());
    }
}
```

**Variables in Interfaces:**

- You can use interfaces to import shared constants into multiple classes by simply declaring an interface that contains variables that are initialized to the desired values.
- It is as if that class were importing the constant fields into the class name space as **final** variables.
- The next example uses this technique to implement an automated “decision maker”:

```
import java.util.Random;
```

```
interface SharedConstants {
```

```
    int NO = 0;
```

```
    int YES = 1;
```

```
    int MAYBE = 2;
```

```
    int LATER = 3;
```

```
    int SOON = 4;
```

```
    int NEVER = 5;
```

```
}
```

```
class Question implements SharedConstants {
```

```
    Random rand = new Random();
```

```
    int ask() {
```

```
        int prob = (int) (100 * rand.nextDouble());
```

```
        if (prob < 30)
```

```
            return NO; // 30%
```

```
        else if (prob < 60)
```

```
            return YES; // 30%
```

```
        else if (prob < 75)
```

```
            return LATER; // 15%
```

```
        else if (prob < 98)
```

```
            return SOON; // 13%
```

```
        else
```

```
            return NEVER; // 2%
```

```
    }
```

```
}
```

```
class AskMe implements SharedConstants {
```

```
    static void answer(int result) {
```

```
        switch(result) {
            case NO:
                System.out.println("No");
                break;
            case YES:
                System.out.println("Yes");
                break;
            case MAYBE:
                System.out.println("Maybe");
                break;
            case LATER:
                System.out.println("Later");
                break;
            case SOON:
                System.out.println("Soon");
                break;
            case NEVER:
                System.out.println("Never");
                break;
        }
    }

    public static void main(String args[]) {
        Question q = new Question();
        answer(q.ask());
        answer(q.ask());
        answer(q.ask());
        answer(q.ask());
    }
}
```

**Interfaces Can Be Extended:**

- One interface can inherit another by use of the keyword **extends**. The syntax is the same as for inheriting classes.

- When a class implements an interface that inherits another interface, it must provide implementations for all methods defined within the interface inheritance chain.

Following is an example:

// One interface can extend another.

```
interface A {  
    void meth1();  
    void meth2();  
}
```

// B now includes meth1() and meth2() -- it adds meth3().

```
interface B extends A {  
    void meth3();  
}
```

// This class must implement all of A and B

```
class MyClass implements B {  
    public void meth1() {  
        System.out.println("Implement meth1().");  
    }  
    public void meth2() {  
        System.out.println("Implement meth2().");  
    }  
    public void meth3() {  
        System.out.println("Implement meth3().");  
    }  
}
```

```
class IFExtend {  
    public static void main(String arg[]) {  
        MyClass ob = new MyClass();  
        ob.meth1();  
        ob.meth2();  
        ob.meth3();  
    }  
}
```

## **MODULE 5**

- **Enumerations, Type Wrappers**
- **I/O, Applets, and Other Topics:** I/O Basics, Reading Console Input, Writing Console Output, The PrintWriter Class, Reading and Writing Files, Applet Fundamentals, The transient and volatile Modifiers, Using instanceof, strictfp, Native Methods, Using assert, Static Import, Invoking Overloaded Constructors Through this( ),
- **String Handling:** The String Constructors, String Length, Special String Operations, Character Extraction, String Comparison, Searching Strings, Modifying a String, Data Conversion Using valueOf( ), Changing the Case of Characters Within a String , Additional String Methods, StringBuffer, StringBuilder.

## ENUMERATIONS

- “**Enumeration** means a list of named constant, enum in java is a data type that contains fixed set of constants.
- It can be used for days of the week (SUNDAY, MONDAY, TUESDAY, WEDNESDAY, THURSDAY, FRIDAY and SATURDAY) , directions (NORTH, SOUTH, EAST and WEST) etc.
- It is available from JDK 1.5.
- An Enumeration can have constructors, methods and instance variables. It is created using **enum** keyword.
- Each enumeration constant is *public*, *static* and *final* by default.
- Even though enumeration defines a class type and have constructors, you do not instantiate an **enum** using **new**.
- Enumeration variables are used and declared in much a same way as you do a primitive variable.

## ENUMERATION FUNDAMENTALS

### *How to Define and Use an Enumeration*

- An enumeration can be defined simply by creating a list of enum variable. Let us take an example for list of Subject variable, with different subjects in the list.

```
enum Subject //Enumeration defined
{
    JAVA, CPP, C, DBMS
}
```

- Identifiers **JAVA**, **CPP**, **C** and **DBMS** are called **enumeration constants**. These are public, static and final by default.
- Variables of Enumeration can be defined directly without any **new** keyword.

*Ex: Subject* **sub**;

- Variables of Enumeration type can have only enumeration constants as value.

- We define an enum variable as: `enum_variable = enum_type.enum_constant;`  
Ex: `sub = Subject.Java;`
- Two enumeration constants can be compared for equality by using the `==` relational operator.

**Example:**

```
if(sub == Subject.Java)
{
    ...
}
```

**Program 1: Example of Enumeration**

```
enum WeekDays
{
    sun, mon, tues, wed, thurs, fri, sat
}
class Test
{
    public static void main(String args[])
    {
        WeekDays wk; //wk is an enumeration variable of type WeekDays
        wk = WeekDays.sun;
        //wk can be assigned only the constants defined under enum type Weekdays
        System.out.println("Today is "+wk);
    }
}
```

**Output :** Today is sun

**Program 2: Example of Enumeration using switch statement**

```
enum SpecialStud
{
    ABI, GAUTHAM, NANDA, SOURABH, SUHAS, SHARATH
}
```

```
class Test
{
    public static void main(String args[])
    {
        SpecailStud s;
        s = SpecailStud.SHARATH;
        switch(s)
        {
            case ABI:
                System.out.println("Cricket champion " + s.ABI);
                break;
            case GAUTHAM:
                System.out.println("Football Champion " + s.GAUTHAM);
                break;
            case NANDA:
                System.out.println("KLE student" + s.NANDA);
                break;
            case SOURABH:
                System.out.println("Mechanical Hero " + s.SOURABH);
                break;
            case SUHAS:
                System.out.println("Talkitive Hero " + s.SUHAS);
                break;
            case SHARATH:
                System.out.println("Bunking Hero " + s.SHARATH);
                break;
        }
    }
}
```

**Output:**

Bunking Hero SHARATH



### values() and valueOf() Methods

- The java compiler internally adds the values() method when it creates an enum.
- The values() method returns an array containing all the values of the enum.
- Its general form is,

`public static enum-type[ ] values()`

- valueOf() method is used to return the enumeration constant whose value is equal to the string passed in as argument while calling this method.
- It's general form is,

`public static enum-type valueOf (String str)`

#### **Program 3: Example of enumeration using values() and valueOf() methods:**

```
enum SpecialStud
```

```
{
```

```
    ABI, GAUTHAM, NANDA, SOURABH, SUHAS, SHARATH
```

```
}
```

```
class Test
```

```
{
```

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

```
    {
```

```
        SpecialStud s;
```

```
        System.out.println("All constants of enum type Students are:");
```

```
        SpecialStud sArray[] = SpecialStud.values();
```

```
        //returns an array of constants of type Restaurants
```

```
        for(SpecialStud a : sArray) //using foreach loop
```

```
            System.out.println(a);
```

```
        s = SpecialStud.valueOf("SUHAS");
```

```
        System.out.println("TALKATIVE BOY " + s);
```

```
    }
```

```
}
```

#### **Output:**

All constants of enum type students are:

ABI

GAUTHAM

NANDA

SOURABH

SUHAS

SHARATH

**TALKITIVE BOY SUHAS**

### ***Points to remember about Enumerations***

1. Enumerations are of class type, and have all the capabilities that a Java class has.
2. Enumerations can have Constructors, instance Variables, methods and can even implement Interfaces.
3. Enumerations are not instantiated using **new** keyword.
4. All Enumerations by default inherit **java.lang.Enum** class.
5. enum may implement many interfaces but cannot extend any class because it internally extends Enum class

## **JAVA ENUMERATIONS ARE CLASS TYPES**

### **Enumeration with Constructor, instance variable and Method**

- Java Enumerations Are Class Type.
- It is important to understand that each enum constant is an object of its enumeration type.
- When you define a constructor for an enum, the constructor is called when each enumeration constant is created.
- Also, each enumeration constant has its own copy of any instance variables defined by the enumeration

### **Program 4: Example of Enumeration with Constructor, instance variable and Method**

**enum Apple2**

```
{  
    Jonathan(10), GoldenDel(9), RedDel(12), Winesap(15), Cortland(8);  
    // variable  
    int price;  
    // Constructor
```

```
        Apple2(int p)
        {
            price = p;
        }
//method
    int getPrice()
    {
        return price;
    }
}
public class EnumConstructor
{
    public static void main(String[] args)
    {
        Apple2 ap;
        // Display price of Winesap.
        System.out.println("Winesap costs " + Apple2.Winesap.getPrice() + "
cents.\n");
        System.out.println(Apple2.GoldenDel.price);
        // Display all apples and prices.
        System.out.println("All apple prices:");
        for(Apple2 a : Apple2.values())
            System.out.println(a + " costs " + a.getPrice() + " cents.");
    }
}
```

**Output:**

Winesap costs 15 cents.

9

All apple prices:

Jonathan costs 10 cents.

GoldenDel costs 9 cents.

RedDel costs 12 cents.

Winesap costs 15 cents.

Cortland costs 8 cents.

- In this example as soon as we declare an enum variable(Apple2 ap ) the constructor is called once, and it initializes value for every enumeration constant with values specified with them in parenthesis.

### ENUMERATIONS INHERITS ENUM

- All enumerations automatically inherit one: **java.lang.Enum**. This class defines several methods that are available for use by all enumerations.
- You can obtain a value that indicates an enumeration constant's position in the list of constants. This is called its ordinal value, and it is retrieved by calling the **ordinal( )** method.
- It has this general form: `final int ordinal( )`
- It returns the ordinal value of the invoking constant. Ordinal values begin at zero.
- Thus, in the Apple enumeration, Jonathan has an ordinal value of zero, GoldenDel has an ordinal value of 1, RedDel has an ordinal value of 2, and so on.
- You can compare the ordinal value of two constants of the same enumeration by using the **compareTo( )** method.
- It has this general form: `final int compareTo(enum-type e)`, Here, enum-type is the type of the enumeration, and e is the constant being compared to the invoking constant
- If the invoking constant has an ordinal value less than e's, then `compareTo( )` returns a negative value.
- If the two ordinal values are the same, then zero is returned.
- If the invoking constant has an ordinal value greater than e's, then a positive value is returned.

#### **Program 5: Example with ordinal(), comapareTo and equals() methods**

```
enum Apple5
{
    Jonathan, GoldenDel, RedDel, Winesap, Cortland
}

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

```
Apple5 ap, ap2, ap3;
System.out.println("Here are all apple constants and their ordinal values: ");
for(Apple5 a : Apple5.values())
    System.out.println(a + " " + a.ordinal());
ap = Apple5.RedDel;
ap2 = Apple5.GoldenDel;
ap3 = Apple5.RedDel;
System.out.println();
if(ap.compareTo(ap2) < 0)
    System.out.println(ap + " comes before " + ap2);
if(ap.compareTo(ap2) > 0)
    System.out.println(ap2 + " comes before " + ap);
if(ap.compareTo(ap3) == 0)
    System.out.println(ap + " equals " + ap3);
System.out.println();
if(ap.equals(ap2))
    System.out.println("Error!");
if(ap.equals(ap3))
    System.out.println(ap + " equals " + ap3);
if(ap == ap3)
    System.out.println(ap + " == " + ap3);
    }
}
```

**Output:**

Here are all apple constants and their ordinal values:

Jonathan 0

GoldenDel 1

RedDel 2

Winesap 3

Cortland 4

GoldenDel comes before RedDel

RedDel equals RedDel

RedDel equals RedDel

RedDel == RedDel

**Program 6: Example of Decisions Makers**

```
import java.util.Random;
enum Answers
{
    NO, YES, MAYBE, LATER, SOON, NEVER
}
class Question
{
    Random rand = new Random();
    Answers ask()
    {
        int prob = (int) (100 * rand.nextDouble());
        if (prob < 15)
            return Answers.MAYBE; // 15%
        else if (prob < 30)
            return Answers.NO; // 15%
        else if (prob < 60)
            return Answers.YES; // 30%
        else if (prob < 75)
            return Answers.LATER; // 15%
        else if (prob < 98)
            return Answers.SOON; // 13%
        else
            return Answers.NEVER; // 2%
    }
}
public class DescisionMakers
{
    static void answer(Answers result)
    {
        switch(result)
        {
            case NO: System.out.println("No"); break;
```

```
        case YES: System.out.println("Yes"); break;
        case MAYBE: System.out.println("Maybe"); break;
        case LATER: System.out.println("Later"); break;
        case SOON: System.out.println("Soon"); break;
        case NEVER: System.out.println("Never"); break;
    }
}

public static void main(String[] args)
{
    Question q = new Question();
    answer(q.ask());
    answer(q.ask());
    answer(q.ask());
    answer(q.ask());
}
}
```

**Output:**

No  
Soon  
Soon  
Yes

**TYPE WRAPPERS**

- Java uses primitive data types such as int, double, float etc. to hold the basic data types for the sake of performance.
- Despite the performance benefits offered by the primitive data types, there are situations when you will need an object representation of the primitive data type.
- For example, many data structures in Java operate on objects. So you cannot use primitivedata types with those data structures.
- To handle such type of situations, Java provides type Wrappers which provide classes that encapsulate a primitive type within an object.
- Character : It encapsulates primitive type char within object.

**Character (char *ch*)**

- To obtain the **char** value contained in a **Character** object, call **charValue()**, shown here:

**char charValue()**: It returns the encapsulated character.

- Boolean : It encapsulates primitive type boolean within object.

**Boolean (boolean boolValue)** : *boolValue* must be either **true** or **false**

**Boolean(String boolString)**: if *boolString* contains the string “true” (in uppercase or lowercase), then the new **Boolean** object will be true. Otherwise, it will be false.

- Numeric type wrappers : It is the most commonly used type wrapper.

byte byteValue()  
double doubleValue()  
float floatValue()  
int intValue()  
long longValue()  
short shortValue()

### Example:

The following program demonstrates how to use a numeric type wrapper to encapsulate a value and then extract that value.

```
class Wrap {  
    public static void main(String args[]) {  
        Integer iOb = new Integer(100);  
        int i = iOb.intValue();  
        System.out.println(i + " " + iOb); // displays 100 100  
    }  
}
```

This program wraps the integer value 100 inside an **Integer** object called **iOb**. The program then obtains this value by calling **intValue()** and stores the result in **i**.



## I/O, Applets, and Other Topics

### I/O Basics:

- Till now in the programs **print( )** and **println( )**, none of the I/O methods have been used significantly.
- The reason is simple: most real applications of Java are not text-based, console programs. Rather, they are graphically oriented programs that rely upon Java's Abstract Window Toolkit (AWT) or Swing for interaction with the user.
- Text-based programs are excellent as teaching examples, they do not constitute an important use for Java in the real world.
- Java's support for console I/O is limited and somewhat awkward to use—even in simple example programs
- Java does provide strong, flexible support for I/O as it relates to files and networks. Java's I/O system is cohesive and consistent

### Streams:

- Java programs perform I/O through streams.
- A *stream* is an abstraction that either produces or consumes information. A stream is linked to a physical device by the Java I/O system.
- All streams behave in the same manner, even if the actual physical devices to which they are linked differ.
- An input stream can abstract many different kinds of input: from a disk file, a keyboard, or a network socket.
- An output stream may refer to the console, a disk file, or a network connection. Streams are a clean way to deal with input/output without having every part of your code understand the difference between a keyboard and a network.
- Java defines **two types of streams: byte and character**

## Byte Streams and Character Streams

### *Byte Streams:*

- *Byte streams* provide a convenient means for handling input and output of bytes.
- Byte streams are used, for example, when reading or writing binary data.
- The original version of Java (Java 1.0) was I/O was byte-oriented.

- At the lowest level, all I/O is still byte-oriented.

### *Character Streams:*

- *Character streams* provide a convenient means for handling input and output of characters.
- They use Unicode and, therefore, can be internationalized.
- Character streams were added by Java 1.1, and certain byte-oriented classes and methods were deprecated.
- The character-based streams simply provide a convenient and efficient means for handling characters.

### *The Byte Stream Classes*

- Byte streams are defined by using two class hierarchies. At the top are two abstract classes: **InputStream** and **OutputStream**.
- Each of these abstract classes has several concrete subclasses that handle the differences between various devices, such as disk files, network connections, and even memory buffers.
- The abstract classes **InputStream** and **OutputStream** define several key methods that the other stream classes implement. Two of the most important are **read( )** and **write( )**, which, respectively, read and write bytes of data
- The byte stream classes are shown in Table

Stream Class	Meaning
BufferedInputStream	Buffered input stream
BufferedOutputStream	Buffered output stream
ByteArrayInputStream	Input stream that reads from a byte array
ByteArrayOutputStream	Output stream that writes to a byte array
DataInputStream	An input stream that contains methods for reading the Java standard data types
DataOutputStream	An output stream that contains methods for writing the Java standard data types
FileInputStream	Input stream that reads from a file
FileOutputStream	Output stream that writes to a file
FilterInputStream	Implements InputStream
FilterOutputStream	Implements OutputStream

InputStream	Abstract class that describes stream input
OutputStream	Abstract class that describes stream output

### *The Character Stream Classes*

- Character streams are defined by using two class hierarchies. At the top are two abstract classes, Reader and Writer.
- These abstract classes handle Unicode character streams.
- The abstract classes Reader and Writer define several key methods that the other stream classes implement. Two of the most important methods are read( ) and write( ), which read and write characters of data, respectively.
- The character stream classes are shown in Table

Stream Class	Meaning
BufferedReader	Buffered input character stream
BufferedWriter	Buffered output character stream
CharArrayReader	Input stream that reads from a character array
CharArrayWriter	Output stream that writes to a character array
FileReader	Input stream that reads from a file
FileWriter	Output stream that writes to a file
FilterReader	Filtered reader
FilterWriter	Filtered writer
InputStreamReader	Input stream that translates bytes to characters
LineNumberReader	Input stream that counts lines
PipedReader	Input pipe
PipedWriter	Output pipe

### *The Predefined Streams:*

- All Java programs automatically import the **java.lang** package.
- This package defines a class called **System**, which encapsulates several aspects of the run-time environment.
- **System** also contains three predefined stream variables: **in**, **out**, and **err**. These fields are declared as **public**, **static**, and **final** within **System**.
- **System.out** refers to the standard output stream. By default, this is the console.
- **System.in** refers to standard input, which is the keyboard by default.

- **System.err** refers to the standard error stream, which also is the console by default.
- **System.in** is an object of type **InputStream**; **System.out** and **System.err** are objects of type **PrintStream**. These are byte streams, even though they typically are used to read and write characters from and to the console.

### Reading Console Input

- The only way to perform console input was to use a byte stream during older versions. The preferred method of reading console input now is to use a character-oriented stream, which makes your program easier to internationalize and maintain.
- In Java, console input is accomplished by reading from **System.in**.
- To obtain a character based stream that is attached to the console, wrap **System.in** in a **BufferedReader** object.
- **BufferedReader** supports a buffered input stream.

`BufferedReader(Reader inputReader)`

- Here, *inputReader* is the stream that is linked to the instance of **BufferedReader** that is being created. **Reader** is an abstract class.
- To obtain an **InputStreamReader** object that is linked to **System.in**, use the following constructor:

`InputStreamReader(InputStream inputStream)`

- Because **System.in** refers to an object of type **InputStream**, it can be used for *inputStream*.
- *Combining together we have*  
**BufferedReader br = new BufferedReader(new InputStreamReader(System.in));**

### *Reading Characters:*

- To read a character from a **BufferedReader**, use `read( )`.
- The version of `read( )` that we will be using is  
`int read( )` throws **IOException**
- Each time that `read( )` is called, it reads a character from the input stream and returns it as an integer value.
- It returns `-1` when the end of the stream is encountered.

### **Example:**

```
import java.io.*;
```

```
class BRRead {  
    public static void main(String args[]) throws IOException  
    {  
        char c;  
        BufferedReader br = new  
        BufferedReader(new InputStreamReader(System.in));  
        System.out.println("Enter characters, 'q' to quit.");  
        do {  
            c = (char) br.read();  
            System.out.println(c);  
        } while(c != 'q');  
    }  
}
```

**Output:**

Enter characters, 'q' to quit.

Pinkuq

P

i

n

k

u

q

**Reading Strings**

- To read a string from the keyboard, use the version of `readLine()` that is a member of the `BufferedReader` class.
- Its general form is shown here:

`String readLine()` throws `IOException`: it returns a `String` object.

**Example:**

```
import java.io.*;
```

```
class BRReadLines {  
    public static void main(String args[]) throws IOException  
    {
```

```
// create a BufferedReader using System.in
BufferedReader br = new BufferedReader(new
InputStreamReader(System.in));
String str;
System.out.println("Enter lines of text.");
System.out.println("Enter 'stop' to quit.");
do {
    str = br.readLine();
    System.out.println(str);
} while(!str.equals("stop"));
}
```

**Output:**

Enter lines of text.

Enter 'stop' to quit

Suhas speaks fluent english stop

Suhas

speaks

fluent

english

stop

**Writing Console Output**

- Console output is most easily accomplished with **print()** and **println()**, These methods are defined by the class **PrintStream**.
- **PrintStream** is an output stream derived from **OutputStream**, it also implements the low-level method **write()**. Thus, **write()** can be used to write to the console.
- The simplest form of **write()** defined by **PrintStream** is shown here:  

`void write(int byteval)`
- This method writes to the stream the byte specified by *byteval*.
- Although *byteval* is declared as an integer, only the low-order eight bits are written.

**Example:**

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

```

        int b;
        b = 'A';
        System.out.write(b);
        System.out.write('\n');
    }
}

```

### The PrintWriter Class:

- **PrintWriter** is one of the character-based classes.
- Using a character-based class for console output makes it easier to internationalize your program.
- **PrintWriter** defines several constructors. The one we will use is shown here:

```
PrintWriter(OutputStream outputStream, boolean flushOnNewline)
```

- Here, *outputStream* is an object of type **OutputStream**, and *flushOnNewline* controls whether Java flushes the output stream every time a **println()** method is called.
- If *flushOnNewline* is **true**, flushing automatically takes place. If **false**, flushing is not automatic.
- **PrintWriter** supports the **print()** and **println()** methods for all types including **Object**.
- For example, this line of code creates a **PrintWriter** that is connected to console output:

```
PrintWriter pw = new PrintWriter(System.out, true);
```

### **Example:**

```

import java.io.*;

public class PrintWriterDemo {
    public static void main(String args[]) {
        PrintWriter pw = new PrintWriter(System.out, true);
        pw.println("Abdul Kalamji");
        int i = -7;
        pw.println(i);
    }
}

```

### **Output:**

```

Abdul Kalamji
-7

```

## Reading and Writing Files

- Two of the most often-used stream classes are **FileInputStream** and **FileOutputStream**, which create byte streams linked to files.
- To open a file, you simply create an object of one of these classes, specifying the name of the file as an argument to the constructor.

`FileInputStream(String fileName)` throws `FileNotFoundException`

`FileOutputStream(String fileName)` throws `FileNotFoundException`

- Here, *fileName* specifies the name of the file that you want to open.
- When you create an input stream, if the file does not exist, then **FileNotFoundException** is thrown.
- For output streams, if the file cannot be created, then **FileNotFoundException** is thrown.
- When an output file is opened, any preexisting file by the same name is destroyed.
- When you are done with a file, you should close it by calling **close()**.

`void close()` throws `IOException`

- To read from a file, you can use a version of **read()**

`int read()` throws `IOException`

```
import java.io.*;
```

```
class ShowFile {
```

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

```
    {
```

```
        int i;
```

```
        FileInputStream fin;
```

```
        try {
```

```
            fin = new FileInputStream(args[0]);
```

```
        } catch(FileNotFoundException e) {
```

```
            System.out.println("File Not Found");
```

```
            return;
```

```
        } catch(ArrayIndexOutOfBoundsException e) {
```

```
            System.out.println("Usage: ShowFile File");
```

```
            return;
```



```
    }  
    do {  
        i = fin.read();  
        if(i != -1) System.out.print((char) i);  
    } while(i != -1);  
    fin.close();  
}  
}
```

- To write to a file, you can use the **write( )** method

import java.io.\*;

```
class CopyFile {  
    public static void main(String args[]) throws IOException  
    {  
        int i;  
        FileInputStream fin;  
        FileOutputStream fout;  
        try {  
            // open input file  
            try {  
                fin = new FileInputStream(args[0]);  
            } catch(FileNotFoundException e) {  
                System.out.println("Input File Not Found");  
                return;  
            }  
            try {  
                fout = new FileOutputStream(args[1]);  
            } catch(FileNotFoundException e) {  
                System.out.println("Error Opening Output File");  
                return;  
            }  
        } catch(ArrayIndexOutOfBoundsException e) {  
            System.out.println("Usage: CopyFile From To");  
            return;  
        }  
    }  
}
```

```
    }  
    // Copy File  
    try {  
        do {  
            i = fin.read();  
            if(i != -1) fout.write(i);  
        } while(i != -1);  
    } catch(IOException e) {  
        System.out.println("File Error");  
    }  
    fin.close();  
    fout.close();  
}  
}
```

### Applet Fundamentals

- *Applets* are small applications that are accessed on an Internet server, transported over the Internet, automatically installed, and run as part of a web document.
- After an applet arrives on the client, it has limited access to resources so that it can produce a graphical user interface and run complex computations without introducing the risk of viruses or breaching data integrity.

#### **Example:**

```
import java.awt.*;  
import java.applet.*;  
public class SimpleApplet extends Applet {  
    public void paint(Graphics g) {  
        g.drawString("A Simple Applet", 20, 20);  
    }  
}
```

#### **Description:**

- The first imports the Abstract Window Toolkit (AWT) classes. Applets interact with the user (either directly or indirectly) through the AWT, not through the console-

based I/O classes. The AWT contains support for a window-based, graphical user interface.

- The second **import** statement imports the **applet** package, which contains the class **Applet**. Every applet that you create must be a subclass of **Applet**.
- The next line in the program declares the class **SimpleApplet**. This class must be declared as **public**, because it will be accessed by code that is outside the program
- Inside **SimpleApplet**, **paint( )** is declared. This method is defined by the AWT and must be overridden by the applet. **paint( )** is called each time that the applet must redisplay its output. This situation can occur for several reasons. For example, the window in which the applet is running can be overwritten by another window and then uncovered. Or, the applet window can be minimized and then restored. **paint( )** is also called when the applet begins execution. Whatever the cause, whenever the applet must redraw its output, **paint( )** is called.
- The **paint( )** method has one parameter of type **Graphics**
- Inside **paint( )** is a call to **drawString( )**, which is a member of the **Graphics** class.
- This method outputs a string beginning at the specified X,Y location. It has the following general form:  

```
void drawString(String message, int x, int y)
```
- Here, *message* is the string to be output beginning at x,y.
- Unlike Java programs, applets do not begin execution at **main( )**. In fact, most applets don't even have a **main( )** method.
- However, running **SimpleApplet** involves a different process. In fact, there are two ways in which you can run an applet:
  1. Executing the applet within a Java-compatible web browser.
  2. Using an applet viewer, such as the standard tool, **appletviewer**. An applet viewer executes your applet in a window. This is generally the fastest and easiest way to test your applet.

*To execute an applet in a web browser, you need to write a short HTML text file that contains a tag that loads the applet.*

```
<applet code="SimpleApplet" width=200 height=60>  
</applet>
```

- The **width** and **height** statements specify the dimensions of the display area used by the applet.

#### *Steps for execution of applet*

1. Edit a Java source file.
2. Compile your program.
3. Execute the applet viewer, specifying the name of your applet's source file. The applet viewer will encounter the APPLET tag within the comment and execute your applet.

#### *Points that you should remember now:*

- Applets do not need a **main( )** method.
- Applets must be run under an applet viewer or a Java-compatible browser.
- User I/O is not accomplished with Java's stream I/O classes. Instead, applets use the interface provided by the AWT or Swing.

### The transient and volatile Modifiers

- Java defines two interesting type modifiers: **transient** and **volatile**
- When an instance variable is declared as **transient**, then its value need not persist when an object is stored.

#### **For example:**

```
class T {  
    transient int a; // will not persist  
    int b; // will persist  
}
```

- Here, if an object of type **T** is written to a persistent storage area, the contents of **a** would not be saved, but the contents of **b** would.
- The **volatile** modifier tells the compiler that the variable modified by **volatile** can be changed unexpectedly by other parts of your program

#### **Example:**

- In a multithreaded program, sometimes two or more threads share the same variable. For efficiency considerations, each thread can keep its own, private copy of such a shared variable.

- The real (or *master*) copy of the variable is updated at various times, such as when a **synchronized** method is entered.
- While this approach works fine, it may be inefficient at times.
- In some cases, all that really matters is that the master copy of a variable always reflects its current state.
- To ensure this, simply specify the variable as **volatile**, which tells the compiler that it must always use the master copy of a **volatile** variable

### Using instanceof

- Knowing the type of an object during run time: we use instanceof
- The **instanceof** operator has this general form:  
*objref instanceof type*
- Here, *objref* is a reference to an instance of a class, and *type* is a class type. If *objref* is of the specified type or can be cast into the specified type, then the **instanceof** operator evaluates to **true**. Otherwise, its result is **false**.

#### **Example:**

```
class A {  
    int i, j;  
}  
class B {  
    int i, j;  
}  
class C extends A {  
    int k;  
}  
class InstanceOf {  
    public static void main(String args[]) {  
        A a = new A();  
        B b = new B();  
        C c = new C();  
        if(a instanceof A)  
            System.out.println("a is instance of A");  
        if(b instanceof B)
```

```
        System.out.println("b is instance of B");
        if(c instanceof C)
            System.out.println("c is instance of C");
        if(c instanceof A)
            System.out.println("c can be cast to A");
    }
}
```

**Output**

a is instance of A  
b is instance of B  
c is instance of C  
c can be cast to A

**strictfp**

- By modifying a class or a method with strictfp, you ensure that floating-point calculations (and thus all truncations) take place precisely as they did in earlier versions of Java.
- When a class is modified by strictfp, all the methods in the class are also modified by strictfp automatically.

```
strictfp class MyClass { //...
```

- Most programmers never need to use strictfp, because it affects only a very small class of problems.

**Native Methods**

- Occasionally you may want to call a subroutine that is written in a language other than Java.
- Typically, such a subroutine exists as executable code for the CPU and environment in which you are working—that is, native code.

**For example,**

- You may want to call a native code subroutine to achieve faster execution time.
- Or, you may want to use a specialized, third-party library, such as a statistical package.

- Java provides the **native** keyword, which is used to declare native code methods. Once declared, these methods can be called from inside your Java program just as you call any other Java method

For example:

```
public native int meth() ;
public class NativeDemo {
    int i;
    public static void main(String args[]) {
        NativeDemo ob = new NativeDemo();
        ob.i = 10;
        System.out.println("This is ob.i before the native method:" +
            ob.i);
        ob.test(); // call a native method
        System.out.println("This is ob.i after the native method:" +
            ob.i);
    }
    // declare native method
    public native void test() ;
    // load DLL that contains static method
    static {
        System.loadLibrary("NativeDemo");
    }
}
```

- Notice that the **test()** method is declared as **native** and has no body. This is the method that we will implement in C shortly.
- The library is loaded by the **loadLibrary()** method, which is part of the **System** class. This is its general form:

`static void loadLibrary(String file name)`

- After you enter the program, compile it to produce **NativeDemo.class**. Next, you must use **javah.exe** to produce one file: **NativeDemo.h**.
- To produce **NativeDemo.h**, use the following command:

javah -jni NativeDemo

- This command produces a header file called **NativeDemo.h**. This file must be included in the C file that implements **test( )**.

```
#include <jni.h>
#include "NativeDemo.h"
#include <stdio.h>

JNIEXPORT void JNICALL Java_NativeDemo_test(JNIEnv *env, jobject obj)
{
    jclass cls;
    jfieldID fid;
    jint i;
    printf("Starting the native method.\n");
    cls = (*env)->GetObjectClass(env, obj);
    fid = (*env)->GetFieldID(env, cls, "i", "I");
    if(fid == 0) {
        printf("Could not get field id.\n");
        return;
    }
    i = (*env)->GetIntField(env, obj, fid);
    printf("i = %d\n", i);
    (*env)->SetIntField(env, obj, fid, 2*i);
    printf("Ending the native method.\n");
}
```

## Problems with Native Methods

### 1. Potential security risk

- Because a native method executes actual machine code, it can gain access to any part of the host system.
- That is, native code is not confined to the Java execution environment. This could allow a virus infection
- The loading of DLLs can be restricted, and their loading is subject to the approval of the security manager



## 2. Loss of portability

- Because the native code is contained in a DLL, it must be present on the machine that is executing the Java program each native method is CPU- and operating system–dependent, each DLL is inherently nonportable.
- Thus, a Java application that uses native methods will be able to run only on a machine for which a compatible DLL has been installed.

### Using assert

- It is used during program development to create an *assertion*, which is a condition that should be true during the execution of the program.
- At run time, if the condition actually is true, no other action takes place. However, if the condition is false, then an **AssertionError** is thrown
- The **assert** keyword has two forms. The first is shown here:

`assert condition;`

- Here, *condition* is an expression that must evaluate to a Boolean result.

Example:

```
class AssertDemo {
    static int val = 3;
    // Return an integer.
    static int getnum() {
        return val--;
    }
    public static void main(String args[])
    {
        int n;
        for(int i=0; i < 10; i++) {
            n = getnum();
            assert n > 0; // will fail when n is 0
            System.out.println("n is " + n);
        }
    }
}
```

- The second form of **assert** is shown here:

`assert condition : expr;`

- *expr* is a value that is passed to the **AssertionError** constructor. This value is converted to its string format and displayed if an assertion fails.

```
class AssertDemo {  
    // get a random number generator  
    static int val = 3;  
    // Return an integer.  
    static int getnum() {  
        return val--;  
    }  
    public static void main(String args[])  
    {  
        int n = 0;  
        for(int i=0; i < 10; i++) {  
            assert (n = getnum()) > 0; // This is not a good idea!  
            System.out.println("n is " + n);  
        }  
    }  
}
```

### Assertion Enabling and Disabling Options

- When executing code, you can disable or enable assertions
- To enable assertions in a package called **MyPack**, use  
-ea:MyPack
- To disable assertions in **MyPack**, use  
-da:MyPack

### Static Import

- JDK 5 added a new feature to Java called *static import* that expands the capabilities of the **import** keyword.

- By following **import** with the keyword **static**, an **import** statement can be used to import the static members of a class or interface

Example:

```
import static java.lang.Math.sqrt;
import static java.lang.Math.pow;
// Compute the hypotenuse of a right triangle.
class Hypot {
    public static void main(String args[]) {
        double side1, side2;
        double hypot;
        side1 = 3.0;
        side2 = 4.0;
        // Here, sqrt() and pow() can be called by themselves,
        // without their class name.
        hypot = sqrt(pow(side1, 2) + pow(side2, 2));
        System.out.println("Given sides of lengths " + side1 + " and " + side2 + " the
        hypotenuse is " + hypot);
    }
}
```

- There are two general forms of the **import static** statement.
- The first, which is used by the preceding example, brings into view a single name. Its general form is shown here:

```
import static pkg.type-name.static-member-name;
```

- The second form of static import imports all static members of a given class or interface. Its general form is shown here:

```
import static pkg.type-name.*;
```

For example,

- this brings the static field **System.out** into view:  

```
import static java.lang.System.out;
```
- After this statement, you can output to the console without having to qualify **out** with **System**, as shown here:

```
out.println("Welcome to ECE and Mechanical students");
```

### Invoking Overloaded Constructors Through this( )

- When working with overloaded constructors, it is sometimes useful for one constructor to invoke another.
- In Java, this is accomplished by using another form of the **this** keyword.
- The general form is shown here:

```
    this(arg-list);
```

```
class MyClass {
    int a;
    int b;
    // initialize a and b individually
    MyClass(int i, int j) {
        a = i;
        b = j;
    }
    // initialize a and b to the same value
    MyClass(int i) {
        this(i, i); // invokes MyClass(i, i)
    }
    // give a and b default values of 0
    MyClass( ) {
        this(0); // invokes MyClass(0)
    }
}
```

- One reason why invoking overloaded constructors through **this( )** can be useful is that it can prevent the unnecessary duplication of code.
- In many cases, reducing duplicate code decreases the time it takes to load your class because often the object code is smaller.
- Constructors that call **this( )** will execute a bit slower than those that contain all of their initialization code inline.
- There are two restrictions you need to keep in mind when using **this( )**.

- First, you cannot use any instance variable of the constructor's class in a call to **this()**.
- Second, you cannot use **super()** and **this()** in the same constructor because each must be the first statement in the constructor.

## String Handling

### The String Constructors

- The **String** class supports several constructors. To create an empty **String**, you call the default constructor. For example,

```
String s = new String();
```

- To create a **String** initialized by an array of characters, use the constructor shown here:

```
String(char chars[ ])
```

#### **Example:**

```
char chars[] = { 'm', 'a', 'n' };
```

```
String s = new String(chars);
```

This constructor initializes **s** with the string “man”.

- You can specify a subrange of a character array as an initializer using the following constructor:

```
String(char chars[ ], int startIndex, int numChars);
```

- *startIndex* specifies the index at which the subrange begins, and *numChars* specifies the number of characters to use.

#### **Example:**

```
char chars[] = { 'h', 'e', 'l', 'l', 'o' };
```

```
String s = new String(chars, 2, 3);
```

This initializes **s** with the characters **llo**.

- You can construct a **String** object that contains the same character sequence as another **String** object using this constructor:

```
String(String strObj)
```

#### **Example:**

```
class MakeString {  
    public static void main(String args[]) {  
        char c[] = { 'r', 'a', 'm', 'a' };  
        String s1 = new String(c);  
        String s2 = new String(s1);  
    }  
}
```

```
        System.out.println(s1);
        System.out.println(s2);
    }
}
```

**Output:**

rama

rama

- Even though Java's **char** type uses 16 bits to represent the basic Unicode character set, the typical format for strings on the Internet uses arrays of 8-bit bytes constructed from the ASCII character set.
- The byte formats are Their forms are shown here:

```
String(byte asciiChars[ ])
```

```
String(byte asciiChars[ ], int startIndex, int numChars)
```

**Example:**

```
class SubStringCons {
    public static void main(String args[]) {
        byte ascii[] = { 65, 66, 67, 68, 69, 70 };
        String s1 = new String(ascii);
        System.out.println(s1);
        String s2 = new String(ascii, 2, 3);
        System.out.println(s2);
    }
}
```

**Output**

ABCDEF

CDE

**String Constructors Added by J2SE 5**

- J2SE 5 added two constructors to **String**. The first supports the extended Unicode character set and is shown here:

```
String(int codePoints[ ], int startIndex, int numChars);
```

- The second new constructor supports the new **StringBuilder** class. It is shown here:

String(StringBuilder *strBuildObj*)

## String Length

- The length of a string is the number of characters that it contains. To obtain this value, call the **length( )** method, shown here:

int length( );

### **Example:**

```
char chars[] = {'k', 'r', 'i', 's', 'h'};  
String s = new String(chars);  
System.out.println(s.length()); // displays 5
```

## Special String Operations

### *String Literals*

- To explicitly create a **String** instance from an array of characters by using the **new** operator. an easier way to do this using a string literal.

### **Example:**

```
char chars[] = {'k', 'r', 'i', 's', 'h'};  
String s1 = new String(chars);  
String s2 = "krish"; // use string literal
```

### *String Concatenation*

- Java does not allow operators to be applied to **String** objects.
- The one exception to this rule is the + operator, which concatenates two strings, producing a **String** object

### **Example:**

```
String marks = "35";  
String s = "He scored " + marks + " in internals.";  
System.out.println(s); // He scored 35 in internals
```

### *String Concatenation with Other Data Types*

- You can concatenate strings with other types of data.

### **Example1:**

```
String s = "four: " + 2 + 2;
```



```
System.out.println(s);
```

This fragment displays

four: 22

**Example2:**

```
String s = "four: " + (2 + 2);
```

Now s contains the string “four: 4”.

***String Conversion and toString( )***

- Java converts data into its string representation during concatenation, it does so by calling one of the overloaded versions of the string conversion method **valueOf( )** defined by **String**
- Every class implements **toString( )** because it is defined by **Object**
- The **toString( )** method has this general form:

String toString( )

**Example:**

```
class Box {
    double width;
    double height;
    double depth;
    Box(double w, double h, double d) {
        width = w;
        height = h;
        depth = d;
    }
    public String toString() {
        return "Dimensions are " + width + " by " + depth + " by " + height + ".";
    }
}

class toStringDemo {
    public static void main(String args[]) {
        Box b = new Box(10, 12, 14);
        String s = "Box b: " + b; // concatenate Box object
```

```
        System.out.println(b); // convert Box to string
        System.out.println(s);
    }
}
```

Output:

Dimensions are 10.0 by 14.0 by 12.0

Box b: Dimensions are 10.0 by 14.0 by 12.0

### Character Extraction

#### *charAt( )*:

- To extract a single character from a **String**, you can refer directly to an individual character via the **charAt( )** method. It has this general form:

```
char charAt(int where);
```

#### **Example:**

```
char ch;
```

```
ch = "abc".charAt(1); // ch contains b
```

#### *getChars( )*

- If you need to extract more than one character at a time, you can use the **getChars( )** method. It has this general form:

```
void getChars(int sourceStart, int sourceEnd, char target[ ], int targetStart)
```

#### **Example:**

```
String s = "Abhimanyu is a good cricket player";
```

```
int start = 16;
```

```
int end = 20;
```

```
char buf[] = new char[end - start];
```

```
s.getChars(start, end, buf, 0);
```

```
System.out.println(buf);
```

#### **Output:**

good

***getBytes( )***

- There is an alternative to **getChars( )** that stores the characters in an array of bytes. This method is called **getBytes( )**, and it uses the default character-to-byte conversions provided by the platform.
- Here is its simplest form:

byte[ ] getBytes( )

***toCharArray( )***

- If you want to convert all the characters in a **String** object into a character array, the easiest way is to call **toCharArray( )**. It returns an array of characters for the entire string. It has this general form:

char[ ] toCharArray( )

**String Comparison*****equals( ) and equalsIgnoreCase( )***

- To compare two strings for equality, use **equals( )**. It has this general form:

boolean equals(Object *str*)

- To perform a comparison that ignores case differences, call **equalsIgnoreCase( )**. It has this general form:

boolean equalsIgnoreCase(String *str*)

**Example:**

```
class equalsDemo {
public static void main(String args[]) {
String s1 = "Sourabh";
String s2 = "Sourabh";
String s3 = "Nikil";
String s4 = "SOURABH";
System.out.println(s1 + " equals " + s2 + " -> " + s1.equals(s2));
System.out.println(s1 + " equals " + s3 + " -> " + s1.equals(s3));
System.out.println(s1 + " equals " + s4 + " -> " + s1.equals(s4));
System.out.println(s1 + " equalsIgnoreCase " + s4 + " -> " + s1.equalsIgnoreCase(s4));
}
```

```
}  
}
```

### *regionMatches( )*

- The **regionMatches( )** method compares a specific region inside a string with another specific region in another string
- Here are the general forms for these two methods:

```
boolean regionMatches(int startIndex, String str2, int str2StartIndex,  
int numChars)
```

```
boolean regionMatches(boolean ignoreCase, int startIndex, String str2,  
int str2StartIndex, int numChars)
```

- For both versions, *startIndex* specifies the index at which the region begins within the invoking **String** object.
- The index at which the comparison will start within *str2* is specified by *str2StartIndex*
- The length of the substring being compared is passed in *numChars*.
- In the second version, if *ignoreCase* is **true**, the case of the characters is ignored. Otherwise, case is significant.

### *startsWith( ) and endsWith( )*

- **String** defines two routines that are, more or less, specialized forms of **regionMatches( )**.
- The **startsWith( )** method determines whether a given **String** begins with a specified string.
- **The endsWith( )** determines whether the **String** in question ends with a specified string. They have the following general forms:

```
boolean startsWith(String str)
```

```
boolean endsWith(String str)
```

#### **Example:**

```
"Nandasagar".endsWith("gar")
```

and

```
"Nandasagar".startsWith("Nan")
```

are both **true**.

- A second form of **startsWith( )**, shown here, lets you specify a starting point:

```
boolean startsWith(String str, int startIndex);
```

"Nandasagar".startsWith("gar", 7) returns **true**.

### *equals( ) Versus ==*

- It is important to understand that the **equals( )** method and the **==** operator perform two different operations.
- The **equals( )** method compares the characters inside a **String** object
- The **==** operator compares two object references to see whether they refer to the same instance.

Example:

```
String s1 = "Sunaina";
```

```
String s2 = new String(s1);
```

```
System.out.println(s1 + " equals " + s2 + " -> " + s1.equals(s2)); // true
```

```
System.out.println(s1 + " == " + s2 + " -> " + (s1 == s2)); // false
```

### *compareTo( )*

- To know which is *less than*, *equal to*, or *greater than* **String** method **compareTo( )** serves this purpose. It has this general form:

```
int compareTo(String str)
```

Value	Meaning
Less than zero	The invoking string is less than str.
Greater than zero	The invoking string is greater than str.
Zero	The two strings are equal.

### **Example : Bubble sort**

```
class SortString {
    static String arr[] = {
        "abi", "nanda", "gautham", "saurab", "akshay", "suhas"
    };
    public static void main(String args[]) {
        for(int j = 0; j < arr.length; j++) {
            for(int i = j + 1; i < arr.length; i++) {
```

```

        if(arr[i].compareTo(arr[j]) < 0) {
            String t = arr[j];
            arr[j] = arr[i];
            arr[i] = t;
        }
    }
    System.out.println(arr[j]);
}
}
}

```

- If you want to ignore case differences when comparing two strings, use **compareToIgnoreCase( )**, as shown here:

```
int compareToIgnoreCase(String str)
```

### Searching Strings

- The **String** class provides two methods that allow you to search a string for a specified character or substring:
- **indexOf( )** Searches for the first occurrence of a character or substring.
- **lastIndexOf( )** Searches for the last occurrence of a character or substring.
- The methods return the index at which the character or substring was found, or -1 on failure.

- To search for the first occurrence of a character, use

```
int indexOf(int ch)
```

- To search for the last occurrence of a character, use

```
int lastIndexOf(int ch)
```

- To search for the first or last occurrence of a substring, use

```
int indexOf(String str)
```

```
int lastIndexOf(String str)
```

- You can specify a starting point for the search using these forms:

```
int indexOf(int ch, int startIndex)
```

```
int lastIndexOf(int ch, int startIndex)
```

```
int indexOf(String str, int startIndex)
```

```
int lastIndexOf(String str, int startIndex)
```

## Modifying a String

### *substring( )*

- You can extract a substring using **substring( )**. It has two forms.
- The first is `String substring(int startIndex)`
- The second form of **substring( )** allows you to specify both the beginning and ending index of the substring:

```
String substring(int startIndex, int endIndex)
```

### Example:

```
class StringReplace {  
    public static void main(String args[]) {  
        String org = "This is a test. This is, too.";  
        String search = "is";  
        String sub = "was";  
        String result = "";  
        int i;  
        do { // replace all matching substrings  
            System.out.println(org);  
            i = org.indexOf(search);  
            if(i != -1) {  
                result = org.substring(0, i);  
                result = result + sub;  
                result = result + org.substring(i + search.length());  
                org = result;  
            }  
        } while(i != -1);  
    }  
}
```

The output from this program is shown here:

This is a test. This is, too.

Thwas is a test. This is, too.

Thwas was a test. This is, too.

Thwas was a test. Thwas is, too.

Thwas was a test. Thwas was, too.

### *concat( )*

- You can concatenate two strings using **concat( )**, shown here:

String concat(String *str*)

For example,

```
String s1 = "Nanda";
```

```
String s2 = s1.concat("sagar");
```

puts the string “Nandasagar” into **s2**. It generates the same result as the following sequence:

```
String s1 = "Nanda";
```

```
String s2 = s1 + "sagar";
```

### *replace( )*

- The **replace( )** method has two forms.
- The first replaces all occurrences of one character in the invoking string with another character. It has the following general form:

String replace(char *original*, char *replacement*)

- Here, *original* specifies the character to be replaced by the character specified by *replacement*.
- The resulting string is returned. For example,

```
String s = "Hello".replace('l', 'w');
```

puts the string “Hewwo” into **s**.

- The second form of **replace( )** replaces one character sequence with another. It has this general form:

String replace(CharSequence *original*, CharSequence *replacement*)

### *trim( )*

- The **trim( )** method returns a copy of the invoking string from which any leading and trailing whitespace has been removed. It has this general form:

String trim( )

Example:

```
String s = " Hello World ".trim();
```



This puts the string “Hello World” into s.

### Data Conversion Using valueOf()

- The **valueOf()** method converts data from its internal format into a human-readable form Here are a few of its forms:

static String valueOf(double *num*)

static String valueOf(long *num*)

static String valueOf(Object *ob*)

static String valueOf(char *chars*[])

- There is a special version of **valueOf()** that allows you to specify a subset of a **char** array. It has this general form:

static String valueOf(char *chars*[], int *startIndex*, int *numChars*)

### *Changing the Case of Characters Within a String*

- The method **toLowerCase()** converts all the characters in a string from uppercase to lowercase.
- The **toUpperCase()** method converts all the characters in a string from lowercase to uppercase
- General forms of these methods:

String toLowerCase()

String toUpperCase()

#### **Example:**

```
class ChangeCase {  
    public static void main(String args[])  
    {  
        String s = "Lord Krishna";  
        System.out.println("Original: " + s);  
        String upper = s.toUpperCase();  
        String lower = s.toLowerCase();  
        System.out.println("Uppercase: " + upper);  
        System.out.println("Lowercase: " + lower);  
    }  
}
```

Output:

Original: Lord Krishna

Uppercase: LORD KRISHNA

Lowercase: lord krishna

### Additional String Methods

Method	Description
<code>int codePointAt(int i)</code>	Returns the Unicode code point at the location specified by <i>i</i> . Added by J2SE 5.
<code>int codePointBefore(int i)</code>	Returns the Unicode code point at the location that precedes that specified by <i>i</i> . Added by J2SE 5.
<code>int codePointCount(int start, int end)</code>	Returns the number of code points in the portion of the invoking <b>String</b> that are between <i>start</i> and <i>end</i> -1. Added by J2SE 5.
<code>boolean contains(CharSequence str)</code>	Returns <b>true</b> if the invoking object contains the string specified by <i>str</i> . Returns <b>false</b> , otherwise. Added by J2SE 5.
<code>boolean contentEquals(CharSequence str)</code>	Returns <b>true</b> if the invoking string contains the same string as <i>str</i> . Otherwise, returns <b>false</b> . Added by J2SE 5.
<code>boolean contentEquals(StringBuffer str)</code>	Returns <b>true</b> if the invoking string contains the same string as <i>str</i> . Otherwise, returns <b>false</b> .
<code>static String format(String fmtstr, Object ... args)</code>	Returns a string formatted as specified by <i>fmtstr</i> . (See Chapter 18 for details on formatting.) Added by J2SE 5.
<code>static String format(Locale loc, String fmtstr, Object ... args)</code>	Returns a string formatted as specified by <i>fmtstr</i> . Formatting is governed by the locale specified by <i>loc</i> . (See Chapter 18 for details on formatting.) Added by J2SE 5.
<code>boolean matches(String regExp)</code>	Returns <b>true</b> if the invoking string matches the regular expression passed in <i>regExp</i> . Otherwise, returns <b>false</b> .
<code>int offsetByCodePoints(int start, int num)</code>	Returns the index with the invoking string that is <i>num</i> code points beyond the starting index specified by <i>start</i> . Added by J2SE 5.
<code>String replaceFirst(String regExp, String newStr)</code>	Returns a string in which the first substring that matches the regular expression specified by <i>regExp</i> is replaced by <i>newStr</i> .
<code>String replaceAll(String regExp, String newStr)</code>	Returns a string in which all substrings that match the regular expression specified by <i>regExp</i> are replaced by <i>newStr</i> .
<code>String[] split(String regExp)</code>	Decomposes the invoking string into parts and returns an array that contains the result. Each part is delimited by the regular expression passed in <i>regExp</i> .
<code>String[] split(String regExp, int max)</code>	Decomposes the invoking string into parts and returns an array that contains the result. Each part is delimited by the regular expression passed in <i>regExp</i> . The number of pieces is specified by <i>max</i> . If <i>max</i> is negative, then the invoking string is fully decomposed. Otherwise, if <i>max</i> contains a nonzero value, the last entry in the returned array contains the remainder of the invoking string. If <i>max</i> is zero, the invoking string is fully decomposed.
<code>CharSequence subSequence(int startIndex, int stopIndex)</code>	Returns a substring of the invoking string, beginning at <i>startIndex</i> and stopping at <i>stopIndex</i> . This method is required by the <b>CharSequence</b> interface, which is now implemented by <b>String</b> .

## StringBuffer

- **StringBuffer** is a peer class of **String** that provides much of the functionality of strings.
- **StringBuffer** may have characters and substrings inserted in the middle or appended to the end.
- **StringBuffer** will automatically grow to make room for such additions and often has more characters preallocated than are actually needed, to allow room for growth

### *StringBuffer Constructors*

- **StringBuffer** defines these four constructors:
- **StringBuffer( )**: The default constructor (the one with no parameters) reserves room for 16 characters without reallocation.
- **StringBuffer(int size)**: accepts an integer argument that explicitly sets the size of the buffer.
- **StringBuffer(String str)**: accepts a **String** argument that sets the initial contents of the **StringBuffer** object and reserves room for 16 more characters without reallocation
- **StringBuffer(CharSequence chars)**: creates an object that contains the character sequence contained in *chars*.

### *length( ) and capacity( )*

- The current length of a **StringBuffer** can be found via the **length( )** method.
- The total allocated capacity can be found through the **capacity( )** method.
- They have the following general forms:

int length( )  
int capacity( )

```
class StringBufferDemo {  
    public static void main(String args[]) {  
        StringBuffer sb = new StringBuffer("Krishna");  
        System.out.println("buffer = " + sb); // displays Krishna  
        System.out.println("length = " + sb.length()); // displays 7  
        System.out.println("capacity = " + sb.capacity()); // displays 23  
    }  
}
```

```

    }
}

```

### *ensureCapacity( )*

- If you want to preallocate room for a certain number of characters after a **StringBuffer** has been constructed, you can use **ensureCapacity( )** to set the size of the buffer **ensureCapacity( )** has this general form:

```
void ensureCapacity(int capacity)
```

Here, *capacity* specifies the size of the buffer

### *setLength( )*

- To set the length of the buffer within a **StringBuffer** object, use **setLength( )**. Its general form is shown here:

```
void setLength(int len)
```

### *charAt( ) and setCharAt( )*

- The value of a single character can be obtained from a **StringBuffer** via the **charAt( )** method. You can set the value of a character within a **StringBuffer** using **setCharAt( )**. Their general forms are shown here:

```
char charAt(int where)
```

```
void setCharAt(int where, char ch)
```

### **Example:**

```
StringBuffer sb = new StringBuffer("Krishna");
```

```
sb.charAt(2)// selects i
```

```
sb.setCharAt(2, 'u')// sets second index value to u means krushna
```

### *getChars( )*

- To copy a substring of a **StringBuffer** into an array, use the **getChars( )** method. It has this general form:

```
void getChars(int sourceStart, int sourceEnd, char target[ ], int targetStart)
```

- Here, *sourceStart* specifies the index of the beginning of the substring, and *sourceEnd* specifies an index that is one past the end of the desired substring.
- The array that will receive the characters is specified by *target*.

***append( )***

- The **append( )** method concatenates the string representation of any other type of data to the end of the invoking **StringBuffer** object. It has several overloaded versions.

Here are a few of its forms:

`StringBuffer append(String str)`

`StringBuffer append(int num)`

`StringBuffer append(Object obj)`

**Example:**

```
class appendDemo {  
    public static void main(String args[]) {  
        String s;  
        int a = 42;  
        StringBuffer sb = new StringBuffer(40);  
        s = sb.append("a = ").append(a).append("!").toString();  
        System.out.println(s);  
    }  
}
```

The output of this example is shown here:

a = 42!

- The **append( )** method is most often called when the + operator is used on **String** objects.
- Java automatically changes modifications to a **String** instance into similar operations on a **StringBuffer** instance.
- There are many optimizations that the Java run time can make knowing that **String** objects are immutable

***insert( )***

- The **insert( )** method inserts one string into another. It is overloaded to accept values of all the simple types, plus **Strings**, **Objects**, and **CharSequences**. These are a few of its forms:

`StringBuffer insert(int index, String str)`

```
StringBuffer insert(int index, char ch)
StringBuffer insert(int index, Object obj)
```

**Example:**

```
class insertDemo {
    public static void main(String args[]) {
        StringBuffer sb = new StringBuffer("I Java!");
        sb.insert(2, "like ");
        System.out.println(sb);
    }
}
```

The output of this example is shown here:

I like Java!

***reverse( )***

- You can reverse the characters within a **StringBuffer** object using **reverse( )**, shown here:

```
StringBuffer reverse( )
```

**Example:**

```
class ReverseDemo {
    public static void main(String args[]) {
        StringBuffer s = new StringBuffer("abcdef");
        System.out.println(s);
        s.reverse();
        System.out.println(s);
    }
}
```

Here is the output produced by the program:

abcdef

fedcba

***delete( ) and deleteCharAt( )***

- You can delete characters within a **StringBuffer** by using the methods **delete( )** and **deleteCharAt( )**. These methods are shown here:

```
StringBuffer delete(int startIndex, int endIndex)
```

```
StringBuffer deleteCharAt(int loc)
```

**Example:**

```
class deleteDemo {  
    public static void main(String args[]) {  
        StringBuffer sb = new StringBuffer("This is a test.");  
        sb.delete(4, 7);  
        System.out.println("After delete: " + sb);  
        sb.deleteCharAt(0);  
        System.out.println("After deleteCharAt: " + sb);  
    }  
}
```

The following output is produced:

After delete: This a test.

After deleteCharAt: his a test.

***replace( )***

- You can replace one set of characters with another set inside a **StringBuffer** object by calling **replace( )**. Its signature is shown here:

```
StringBuffer replace(int startIndex, int endIndex, String str)
```

**Example:**

```
class replaceDemo {  
    public static void main(String args[]) {  
        StringBuffer sb = new StringBuffer("This is a test.");  
        sb.replace(5, 7, "was");  
        System.out.println("After replace: " + sb);  
    }  
}
```

Here is the output:

After replace: This was a test.

**substring( )**

- You can obtain a portion of a **StringBuffer** by calling **substring( )**. It has the following two forms:

String substring(int *startIndex*)

String substring(int *startIndex*, int *endIndex*)

- The first form returns the substring that starts at *startIndex* and runs to the end of the invoking **StringBuffer** object.
- The second form returns the substring that starts at *startIndex* and runs through *endIndex*-1.

**Additional StringBuffer Methods**

Method	Description
StringBuffer appendCodePoint(int <i>ch</i> )	Appends a Unicode code point to the end of the invoking object. A reference to the object is returned. Added by J2SE 5.
int codePointAt(int <i>i</i> )	Returns the Unicode code point at the location specified by <i>i</i> . Added by J2SE 5.
int codePointBefore(int <i>i</i> )	Returns the Unicode code point at the location that precedes that specified by <i>i</i> . Added by J2SE 5.
int codePointCount(int <i>start</i> , int <i>end</i> )	Returns the number of code points in the portion of the invoking <b>String</b> that are between <i>start</i> and <i>end</i> -1. Added by J2SE 5.
int indexOf(String <i>str</i> )	Searches the invoking <b>StringBuffer</b> for the first occurrence of <i>str</i> . Returns the index of the match, or -1 if no match is found.
int indexOf(String <i>str</i> , int <i>startIndex</i> )	Searches the invoking <b>StringBuffer</b> for the first occurrence of <i>str</i> , beginning at <i>startIndex</i> . Returns the index of the match, or -1 if no match is found.
int lastIndexOf(String <i>str</i> )	Searches the invoking <b>StringBuffer</b> for the last occurrence of <i>str</i> . Returns the index of the match, or -1 if no match is found.
int lastIndexOf(String <i>str</i> , int <i>startIndex</i> )	Searches the invoking <b>StringBuffer</b> for the last occurrence of <i>str</i> , beginning at <i>startIndex</i> . Returns the index of the match, or -1 if no match is found.
int offsetByCodePoints(int <i>start</i> , int <i>num</i> )	Returns the index with the invoking string that is <i>num</i> code points beyond the starting index specified by <i>start</i> . Added by J2SE 5.
CharSequence subSequence(int <i>startIndex</i> , int <i>stopIndex</i> )	Returns a substring of the invoking string, beginning at <i>startIndex</i> and stopping at <i>stopIndex</i> . This method is required by the <b>CharSequence</b> interface, which is now implemented by <b>StringBuffer</b> .
void trimToSize( )	Reduces the size of the character buffer for the invoking object to exactly fit the current contents. Added by J2SE 5.

**Example:**

The following program demonstrates **indexOf( )** and **lastIndexOf( )**:

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



```
        StringBuffer sb = new StringBuffer("one two one");
        int i;
        i = sb.indexOf("one");
        System.out.println("First index: " + i);
        i = sb.lastIndexOf("one");
        System.out.println("Last index: " + i);
    }
}
```

The output is shown here:

First index: 0

Last index: 8

### StringBuilder

- J2SE 5 adds a new string class to Java's already powerful string handling capabilities. This new class is called **StringBuilder**.
- It is identical to **StringBuffer** except for one important difference: it is not synchronized, which means that it is not thread-safe.
- The advantage of **StringBuilder** is faster performance.
- However, in cases in which you are using multithreading, you must use **StringBuffer** rather than **StringBuilder**.