Sri Sai Vidya Vikas Shikshana Samithi ®

SAI VIDYA INSTITUTE OF TECHNOLOGY

Approved by AICTE, New Delhi, Affiliated to VTU, Recognized by Govt. of Karnataka Accredited by NBA

RAJANUKUNTE, BENGALURU 560 064, KARNATAKA

Phone: 080-28468191/96/97/98 ,Email: info@saividya.ac.in, URLwww.saividya.ac.in

DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING (CSE)

Module -1

Introduction to Java

Java is a high level, robust, object-oriented, platform independent and secured programming language.

The term WORA, write once and run everywhere is often associated with Java language. It means whenever we compile a Java code, we get the byte code (.class file), and that can be executed (without compiling it again) on different platforms provided they support Java.

In the year **1995,** Java language was developed. It is mainly used to develop web, desktop, and mobile devices.

The Terminologies in Java

- **JVM (Java Virtual Machine):** JVM is the specification that facilitates the runtime environment in which the execution of the Java bytecode takes place
- **Byte Code:** Java compiler compiles the Java code to generate the .class file or the byte code. One has to use the ***javac*** command to invoke the Java compiler.
- **Java Development Kit (JDK):** It is the complete Java Development Kit that encompasses everything, including JRE(Java Runtime Environment), compiler, java docs, debuggers, etc. JDK must be installed on the computer for the creation, compilation, and execution of a Java program.

- **Java Runtime Environment (JRE):** JRE is part of the JDK. If a system has only JRE installed, then the user can only run the program. In other words, only the *java* command works. The compilation of a Java program will not be possible (the *javac* command will not work).

- **Garbage Collector:** Garbage Collectors recollect or delete unreferenced objects. Garbage Collector makes the life of a developer/ programmer easy as they do not have to worry about memory management.

Object-Oriented Programming

Object-oriented programming (OOP) is at the core of Java. In fact, all Java programs are to at least some extent object-oriented. Object-oriented concepts form the heart of Java just as they form the basis for human understanding. object-oriented programming is a powerful and natural paradigm for creating programs.

Two Paradigms

All computer programs consist of two elements: **code and data.**

These are the two paradigms that govern how a program is constructed.

1. **process- oriented model:** This approach characterizes a program as a series of linear steps (that is, code). The process-oriented model can be thought of as code acting on data.
2. **object-oriented programming**: To manage increasing complexity, the second approach, called object-oriented programming, was conceived. Object-oriented programming organizes a program around its data (that is, objects) and a set of well-defined interfaces to that data. An object-oriented program can be characterized as data controlling access to code.

The Three OOP Principles

object-oriented programming is a powerful and natural paradigm for creating programs.

All object-oriented programming languages provide mechanisms that help you implement the object-oriented model. They are **encapsulation, inheritance, and polymorphism**.

➢ *Encapsulation*

- *Encapsulation* is the mechanism that binds together code and the data it manipulates, and keeps both safe from outside interference and misuse.
- Encapsulation is as a protective wrapper that prevents the code and data from being arbitrarily accessed by other code defined outside the wrapper. Access to the code and data inside the wrapper is tightly controlled through a well-defined interface.
- In Java, the basis of encapsulation is the class.
- When you create a class, you will specify the code and data that constitute that class.
- Collectively, these elements are called *members* of the class.
- Specifically, the data defined by the class are referred to as *member variables* or *instance variables*. In properly written Java programs, the methods define how the member variables can be used. This means that the behavior and interface of a class are defined by the methods that operate on its instance data.
- Since the purpose of a class is to encapsulate complexity, there are mechanisms for hiding the complexity of the implementation inside the class. Each method or variable in a class may be marked private or public.
- The *public* interface of a class represents everything that external users of the class need to know, or may know.
- The *private* methods and data can only be accessed by code that is a member of the class. Therefore, any other code that is not a member of the class cannot access a private method or variable. Since the private members of a class may only be accessed by other parts of your program through the class' public methods, you can ensure that no improper actions take place. this means that the public interface should be carefully designed not to expose too much of the inner workings of a class.

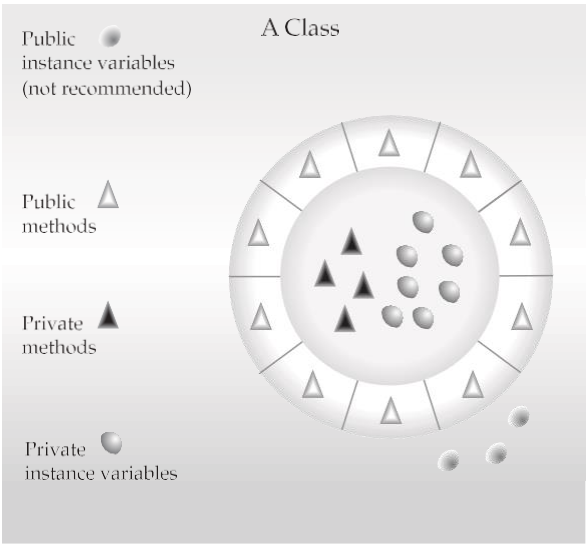
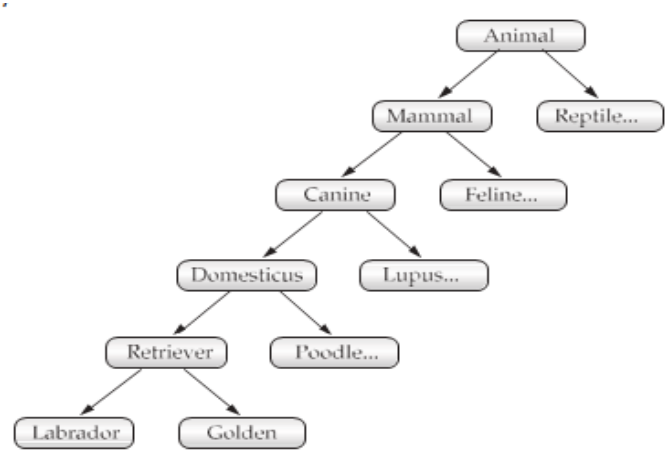


Figure 2-1 Encapsulation: public methods can be used to protect private data.

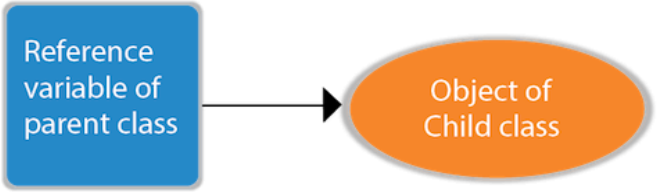
➢ Inheritance

- *Inheritance* is the process by which one object acquires the properties of another object.
- For example, a Golden Retriever is part of the classification *dog*, which in turn is part of the *mammal* class, which is under the larger class *animal*. Without the use of hierarchies, each object would need to define all of its characteristics explicitly. It can inherit its general attributes from its parent.
- Most people naturally view the world as made up of objects that are related to each other in a hierarchical way, such as animals, mammals, and dogs. If you wanted to describe animals in an abstract way, you would say they have some attributes, such as size, intelligence, and type of skeletal system. Animals also have certain behavioral aspects; they eat, breathe, and sleep.



➢ Polymorphism

- *Polymorphism* (from Greek, meaning "many forms") is a feature that allows one interface to be used for a general class of actions. The specific action is determined by the exact nature of the situation.
- This helps reduce complexity by allowing the same interface to be used to specify a *general class of action*. It is the compiler's job to select the *specific action* (that is, method) as it applies to each situation. You, the programmer, do not need to make this selection manually.
- Extending the dog analogy, a dog's sense of smell is polymorphic. If the dog smells a cat, it will bark and run after it. If the dog smells its food, it will salivate and run to its bowl.
- The same sense of smell is at work in both situations. The difference is what is being smelled, that is, the type of data being operated upon by the dog's nose! This same general concept can be implemented in Java as it applies to methods within a Java program.



Two Control Statements

The if Statement

- It determines the flow of execution based on whether some condition is true or false. Its simplest form is shown here:
- if*(condition) statement;*
- Here, *condition* is a Boolean expression. (A Boolean expression is one that evaluates to either true or false.) If *condition* is true, then the statement is executed. If *condition* is false, then the statement is bypassed.
- Relational operators which may be used in a conditional expression. Here are a few:

Operator	Meaning
<	Less than
>	Greater than
==	Equal to

```
class IfSample {
  public static void main(String[] args) {
    int x, y;
            x = 10;
            y = 20;
            if(x < y) System.out.println("x is less than y"); x = x * 2;
            if(x == y) System.out.println("x now equal to y");
            x = x * 2;
            if(x > y) System.out.println("x now greater than y");

// this won't display anything
if(x == y) System.out.println("you won't see this");
}
}
```

The output generated by this program is shown here:

```
x is less than y
x now equal to y
x now greater than y
```

The for Loop

- Loop statements are an important part of nearly any programming language because they provide a way to repeatedly execute some task.
- The simplest form of the **for** loop is shown here:
- for(initialization; condition; iteration) statement;

Here is a short program that illustrates the **for** loop:

```
class ForTest {
 public static void main(String[] args) { int x;
   for(x = 0; x<10; x = x+1) System.out.println("This is x: " + x);
}
}
```

This program generates the following output:

```
This is x: 0
 This is x: 1
This is x: 2
 This is x:3
 This is x: 4
This is x: 5
This is x: 6
This is x: 7
 This is x: 8
This is x: 9
```

Using Blocks of Code

- Java allows two or more statements to be grouped into *blocks of code*, also called *code blocks*.
- This is done by enclosing the statements between opening and closing curly braces.
- Once a block of code has been created, it becomes a logical unit that can be used any place that a single statement can.

For example,

```
class BlockTest {
  public static void main(String[] args) {
   int x, y;
   y = 20;
   for(x = 0; x<10; x++) {
   System.out.println("This is x: " + x);
System.out.println("This is y: " + y); y = y - 2;
    }
  }
 }
```

The output generated by this program is shown here:

```
This is x: 0
This is y: 20
This is x: 1
```

```
This is y: 18
This is x: 2
This is y: 16
This is x: 3
This is y: 14
This is x: 4
This is y: 12
This is x: 5
This is y: 10
This is x: 6
This is y: 8
This is x: 7
This is y: 6
This is x: 8
This is y: 4
This is x: 9
This is y: 2
```

Lexical Issues

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

➢ Whitespace

- Java is a free-form language.
- This means that you do not need to follow any special indentation rules.
- For instance, the **Example** program could have been written all on one line or in any other strange way you felt like typing it, as long as there was at least one whitespace character between each token that was not already delineated by an operator or separator. In Java, whitespace includes a space, tab, newline, or form feed.

➢ Identifiers

- Identifiers are used to name things, such as classes, variables, and methods.
- An identifier may be any descriptive sequence of uppercase and lowercase letters, numbers, or the underscore and dollar-sign characters.
- They must not begin with a number, lest they be confused with a numeric literal.
- Again, Java is case- sensitive, so **VALUE** is a different identifier than **Value**.
- Some examples of valid identifiers are

AvgTemp	count	a4	$test	this_is_ok

Invalid identifier names include these:

2count	high-temp	Not/ok

➢ Literals

- A constant value in Java is created by using a *literal* representation of it.
- For example, here are some literals:

100	98.6	'X'	"This is a test"

Left to right, the first literal specifies an integer, the next is a floating-point value, the third is a character constant, and the last is a string. A literal can be used anywhere a value of its type is allowed.

➢ Comments

There are three types of comments defined by Java: **single-line and multiline and *documentation comment***. This type of comment is used to produce an HTML file that documents your program. The documentation comment begins with a /** and ends with a */. Documentation comments are explained in Appendix A.

➢ Separators

In Java, there are a few characters that are used as separators. The most commonly used separator in Java is the semicolon. The separators are shown in the following table:

Symbol	Name	Purpose
( )	Parentheses	Used to contain lists of parameters in method definition and invocation. Also used for defining precedence in expressions, containing expressions in control statements, and surrounding cast types.
{ }	Braces	Used to contain the values of automatically initialized arrays. Also used to define a block of code, for classes, methods, and local scopes.
[ ]	Brackets	Used to declare array types. Also used when dereferencing array values.
;	Semicolon	Terminates statements.
,	Comma	Separates consecutive identifiers in a variable declaration. Also used to chain statements together inside a **for** statement.
.	Period	Used to separate package names from subpackages and classes. Also used to separate a variable or method from a reference variable.
::	Colons	Used to create a method or constructor reference.
...	Ellipsis	Indicates a variable-arity parameter.
@	At-sign	Begins an annotation.

➢ The Java Keywords

- There are 67 keywords currently defined in the Java language
- These keywords, combined with the syntax of the operators and separators, form the foundation of the Java language.
- In general, keywords cannot be used as identifiers, meaning that they cannot be used as names for a variable, class, or method.

abstract	assert	boolean	break	byte	case
catch	char	class	const	continue	default
do	double	else	enum	exports	extends
final	finally	float	for	goto	if
implements	import	instanceof	int	interface	long
module	native	new	non-sealed	open	opens
package	permits	private	protected	provides	public
record	requires	return	sealed	short	static
strictfp	super	switch	synchronized	this	throw
throws	to	transient	transitive	try	uses
var	void	volatile	while	with	yield
_					

Data Types, Variables, and Arrays

The Primitive Types

Java defines eight *primitive* types of data: **byte**, **short**, **int**, **long**, **char**, **float**, **double**, and **boolean**. The primitive types are also commonly referred to as *simple* types, and both terms will be used in this book. These can be put in four groups:

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

Integers

- Java defines four integer types: **byte**, **short**, **int**, and **long**.
- All of these are signed, positive and negative values.
- The *width* of an integer type should not be thought of as the amount of storage it consumes, but rather as the *behavior* it defines for variables and expressions of that type.
- The width and ranges of these integer types vary widely, as shown in this table.

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.
- They are also useful when you're working with raw binary data that may not be directly compatible with Java's other built-in types.
- 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 the least- used Java type.
- Here are some examples of **short** variable declarations:

```
short s; short t;
```

int

- The most commonly used integer type is **int**.
- It is a signed 32-bit type that has a range from –2,147,483,648 to 2,147,483,647.
- The reason is that when **byte** and **short** values are used in an expression, they are *promoted* to **int** when the expression is evaluated.

long

- **long** is a signed 64-bit type and is useful for those occasions where an **int** type is not large enough to hold the desired value.
- The range of a **long** is quite large.
- This makes it useful when big, whole numbers are needed

For example, here is a program that computes the number of miles that light will travel in a specified number of days:

```
// Compute distance light travels using long variables.
class Light {
 public static void main(String[] args) {
 int lightspeed;
long days;
long seconds;
long distance;
 lightspeed = 186000;
days = 1000;
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.");
}
}
```

output:

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

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 transcendentals 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.

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

float

- The type **float** specifies a *single-precision* value that uses 32 bits of storage.
- Single precision is faster on some processors and takes half as much space as double precision, but will become imprecise when the values are either very large or very small.
- Variables of type **float** are useful when you need a fractional component,
- Here are some example **float** variable declarations:

```
float hightemp, lowtemp;
```

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.
- All transcendental math functions, such as **sin( )**, **cos( )**, and **sqrt( )**, return **double** values. When you need to maintain accuracy over many iterative calculations, or are manipulating large-valued numbers, **double** is the best choice.

Here is a short program that uses **double** variables to compute the area of a circle:

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

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

Characters

- In Java, the data type used to store characters is **char**.
- A key point to understand is that 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
- In Java **char** is a 16-bit type. The range of a **char** is 0 to 65,535.
- There are no negative **char**s.
- The standard set of characters known as ASCII still ranges from 0 to 127 as always

```
// 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
```

Booleans

- Java has a primitive type, called **boolean**, for logical values.
- It can have only one of two possible values, **true** or **false**.
- This is the type returned by all relational operators, as in the case of **a < b**. **boolean** is also the type *required* by the conditional expressions that govern the control statements such as **if** and **for**.

```
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);
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
```

A Closer Look at Literals

➢ Integer Literals

- Integers are probably the most commonly used type in the typical program.
- Any whole number value is an integer literal.
- Examples are 1, 2, 3, and 42. These are all decimal values, meaning they are describing a base 10 number.
- Two other bases that can be used in integer literals are *octal* (base eight) and *hexadecimal* (base 16).
- Octal values are denoted in Java by a leading zero.
- Integer literals create an **int** value, which in Java is a 32-bit integer value.
- When a literal value is assigned to a **byte** or **short** variable, no error is generated if the literal value is within the range of the target type. An integer literal can always be assigned to a **long** variable.
- You can also specify integer literals using binary. To do so, prefix the value with **0b** or **0B**
- For example, this specifies the decimal value 10 using a binary literal:

```
int x = 0b1010;
```

You can embed one or more underscores in an integer literal. Doing so makes it easier to read large integer literals. When the literal is compiled, the underscores are discarded.

For example, given

```
int x = 123_456_789;
```

the value given to **x** will be 123,456,789. The underscores will be ignored. Underscores can only be used to separate digits. They cannot come at the beginning or the end of a literal. It is, however, permissible for more than one underscore to be used between two digits.

For example, this is valid:

```
int x = 123__456__789;
```

The use of underscores in an integer literal is especially useful when encoding such things as telephone numbers, customer ID numbers, part numbers, and so on. For example, binary values are often visually grouped in four-digits units, as shown here:

```
int x = 0b1101_0101_0001_1010;
```

➢ Floating-Point Literals

- Floating-point numbers represent decimal values with a fractional component.
- They can be expressed in either standard or scientific notation.
- *Standard notation* consists of a whole number component followed by a decimal point followed by a fractional component.
- For example, 2.0, 3.14159, and 0.6667 represent valid standard-notation floating-point numbers.
- Examples include 6.022E23, 314159E–05, and 2e+100.
- Floating-point literals in Java default to **double** precision.
- To specify a **float** literal, you must append an *F* or *f* to the constant.
- You can also explicitly specify a **double** literal by appending a *D* or *d*.
- Hexadecimal floating-point literals are also supported, but they are rarely used.
- They must be in a form similar to scientific notation, but a **P** or **p**, rather than an **E** or **e**, is used.
- For example, 0x12.2P2 is a valid floating-point literal.
- The value following the **P**, called the *binary exponent*, indicates the power-of-two by which the number is multiplied. Therefore, **0x12.2P2** represents 72.5.

➢ Boolean Literals

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

➢ Character Literals

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

➢ String Literals

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

```
"Hello World" "two\nlines"
" \"This is in quotes\""
```

Escape Sequence	Description
\ddd	Octal character (ddd)
\uxxxx	Hexadecimal Unicode character (xxxx)
\'	Single quote
\"	Double quote
\\	Backslash
\r	Carriage return
\n	New line (also known as line feed)
\f	Form feed
\t	Tab
\b	Backspace
\s	Space (added by JDK 15)
endofline	Continue line (applies only to text blocks; added by JDK 15)

Table 3-1 Character Escape Sequences

Variables

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

Declaring a Variable

In Java, all variables must be declared before they can be used.

The basic form of a variable declaration is

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

Here are several examples of variable declarations of various types. Note that some include an initialization.

```
int a, b, c;                                    // declares three ints, a, b, and c.
int d = 3, e, f = 5;                            // declares three more ints, initializing
// d and f.
byte z = 22;                                    // initializes z.
double pi = 3.14159;                            // declares an approximation of pi.
char x = 'x';the variable x has the value 'x'.
```

Dynamic Initialization

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

For example, here is a short program that computes the length of the hypotenuse of a right triangle given the lengths of its two opposing sides:

```
class DynInit {
public static void main(String[] args) {
double a = 3.0, b = 4.0;
// c is dynamically initialized
 double c = Math.sqrt(a * a + b * b);
System.out.println("Hypotenuse is " + c);
}
}
```

Here, three local variables—**a**, **b**, and **c**—are declared. The first two, **a** and **b**, are initialized by constants. However, **c** is initialized dynamically to the length of the hypotenuse The program uses another of Java's built-in methods, **sqrt( )**, which is a member of the **Math** class, to compute the square root of its argument.

The Scope and Lifetime of Variables

So far, all of the variables used have been declared at the start of the **main( )** method. However, Java allows variables to be declared within any block.

It is not uncommon to think in terms of two general categories of scopes: global and local.

The scope defined by a method begins with its opening curly brace. However, if that method has parameters, they too are included within the method's scope.

A method's scope ends with its closing curly brace. This block of code is called the *method body*.

To understand the effect of nested scopes, consider the following program:

```
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);
}
}
```

Type Conversion and Casting

Java's Automatic Conversions

When one type of data is assigned to another type of variable, an *automatic type conversion* will take place if the following two conditions are met:

- The two types are compatible.
- The destination type is larger than the source type.

When these two conditions are met, a *widening conversion* takes place.

For example, the **int** type is always large enough to hold all valid **byte** values, so no explicit cast statement is required.

For widening conversions, the numeric types, including integer and floating-point types, are compatible with each other. However, there are no automatic conversions from the numeric types to **char** or **boolean**. Also, **char** and **boolean** are not compatible with each other.

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

Casting Incompatible Types

To create a conversion between two incompatible types, you must use a cast. A *cast* is simply an explicit type conversion. It has this general form:

(*target-type*) *value*

If a variable declaration includes an initializer, then that variable will be reinitialized each time the block in which it is declared is entered. For example

```
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
```

Arrays

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

One-Dimensional Arrays

A *one-dimensional array* is, essentially, a list of like-typed variables. To create an array, you first must create an array variable of the desired type. The general form of a one-dimensional array declaration is

type[ ] *var-name*;

```
class Array {

public static void main(String[] args)
{
     int[] month_days;
     month_days = new int[12];
     month_days[0] = 31;
     month_days[1] = 28;
     month_days[2] = 31;
     month_days[3] = 30;
     month_days[4] = 31;
     month_days[5] = 30;
     month_days[6] = 31;
     month_days[7] = 31;
     month_days[8] = 30;
     month_days[9] = 31;
     month_days[10] = 30;
     month_days[11] = 31;
```

```
System.out.println("April has " + month_days[3] + " days.");
    }
  }
```

Multidimensional Arrays

In Java, *multidimensional arrays* are implemented as arrays of arrays. To declare a multidimensional array variable, specify each additional index using another set of square brackets. For example, the following declares a two-dimensional array variable called **twoD**:

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

This allocates a 4 by 5 array and assigns it to **twoD**. Internally, this matrix is implemented as an *array* of *arrays* of **int**. Conceptually, this array will look like the one shown in Figure 3-1.

The following program numbers each element in the array from left to right, top to bottom, and then displays these values:

```
// Demonstrate a two-dimensional array.

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

for(i=0; i<4; i++) { for(j=0; j<5; j++)
   System.out.print(twoD[i][j] + " "); System.out.println();
}
}
}
```

This program generates the following output:

```
0 1 2   3 4
5 6 7   8 9
10 11   12 13 14
15 16   17 18 19
```

Operators

Arithmetic Operators

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

Operator	Result
+	Addition (also unary plus)
–	Subtraction (also unary minus)
*	Multiplication
/	Division
%	Modulus
++	Increment
+=	Addition assignment
– =	Subtraction assignment
*=	Multiplication assignment
/=	Division assignment
%=	Modulus assignment
– –	Decrement

The operands of the arithmetic operators must be of a numeric type. You cannot use them on **boolean** types, but you can use them on **char** types, since the **char** type in Java is, essentially, a subset of **int**.

The Basic Arithmetic Operators

The basic arithmetic operations—addition, subtraction, multiplication, and division—all behave as you would expect for all numeric types.

The following simple example program demonstrates the arithmetic operators. It also illustrates the difference between floating-point division and integer division.

```
 class BasicMath {
public static void main(String[] args) {
// arithmetic using integers
System.out.println("Integer Arithmetic");
int a = 1 + 1;
int b = a * 3;
int c = b / 4;
int d = c - a;
int e = -d;
System.out.println("a = " + a);
```

```
System.out.println("b = " + b);
System.out.println("c = " + c);
System.out.println("d = " + d);
System.out.println("e = " + e);

// arithmetic using doubles
 System.out.println("\nFloating Point Arithmetic");
double da = 1 + 1;
double db = da * 3;
double dc = db / 4;
 double dd = dc - a;
 double de = -dd;
System.out.println("da = " + da);
System.out.println("db = " + db);
System.out.println("dc = " + dc);
 System.out.println("dd = " + dd);
 System.out.println("de = " + de);
}
}
```

output:

```
Integer Arithmetic
a = 2
b = 6
c = 1
d = -1
e = 1

Floating Point Arithmetic da = 2.0
        db = 6.0
        dc = 1.5
        dd = -0.5
        de = 0.5
```

The Modulus Operator

The modulus operator, **%**, returns the remainder of a division operation. It can be applied to floating-point types as well as integer types. The following example program demonstrates the **%**:

```
class Modulus {
 public static void main(String[] args) { int x = 42;
double y = 42.25;
```

```
System.out.println("x mod 10 = " + x % 10);
System.out.println("y mod 10 = " + y % 10);
}
}
```

output:

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

Arithmetic Compound Assignment Operators

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

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

The output of this program is shown here:

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

Increment and Decrement

The ++ and the – – are Java's increment and decrement operators

```
class IncDec {
 public static void main(String[] args) {
 int a = 1;
int b = 2;
int c;
int d;
c = ++b;
d = a++;
```

```
c++;
System.out.println("a = " + a);
System.out.println("b = " + b);
 System.out.println("c = " + c);
System.out.println("d = " + d);
}
}
```

The output of this program follows:

a = 2

b = 3

c = 4

d = 1

The Bitwise Operators

Java defines several *bitwise operators* that can be applied to the integer types: **long**, **int**, **short**, **char**, and **byte**. These operators act upon the individual bits of their operands. They are summarized in the following table:

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

The Bitwise Logical Operators

The bitwise logical operators are **&**, **|**, **^**, and **~**. The following table shows the outcome of each operation. In the discussion that follows, keep in mind that the bitwise operators are applied to each individual bit within each operand.

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

Using the Bitwise Logical Operators

```
// Demonstrate the bitwise logical operators

class BitLogic {
public static void main(String[] args) {
  String[] binary = {"0000", "0001", "0010", "0011", "0100", "0101", "0110", "0111","1000", "1001", "1010", "1011", "1100", "1101",
"1110", "1111"};
int a = 3; // 0 + 2 + 1 or 0011 in binary
int b = 6; // 4 + 2 + 0 or 0110 in binary int c = a | b;
int d = a & b; int e = a ^ b;
int f = (~a & b)|(a & ~b); int g = ~a & 0x0f;
System.out.println("    a = " + binary[a]);
System.out.println("    b = " + binary[b]);
 System.out.println("   a|b = " + binary[c]);
 System.out.println("   a&b = " + binary[d]);
System.out.println("   a^b = " + binary[e]);
System.out.println("~a&b|a&~b = " + binary[f]);
System.out.println("         ~a = " + binary[g]);
}
}
```

output from this program:

```
a              = 0011
b              = 0110
a|b            = 0111
a&b            = 0010
a^b            = 0101
~a&b|a&~b      = 0101
~a             = 1100
```

The Left Shift

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

value << num

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

The output generated by this program is shown here:

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

The Right Shift

The right shift operator, **>>**, shifts all of the bits in a value to the right a specified number of times.

Its general form is shown here:

value >> num

```
class HexByte {
 public static void main(String[] args) {
 char[] hex = {'0', '1', '2', '3', '4', '5', '6', '7','8', '9', 'a', 'b', 'c', 'd', 'e', 'f'};
 byte b = (byte) 0xf1;
System.out.println("b = 0x" + hex[(b >> 4) & 0x0f] + hex[b & 0x0f]);
}
}
```

Here is the output of this program:

```
b = 0xf1
```

Relational Operators

The *relational operators* determine the relationship that one operand has to the other.

Specifically, they determine equality and ordering. The relational operators are shown here:

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

Control Statements

The if-else-if Ladder

A common programming construct that is based upon a sequence of nested **if**s 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.

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

Here is the output produced by the program:

```
April is in the Spring.
```

The Traditional switch

The **switch** statement is Java's multiway branch statement. It provides an easy way to dispatch execution to different parts of your code based on the value of an expression
Here is a simple example that uses a **switch** statement:

```
// A simple example of the switch.
class SampleSwitch {
  public static void main(String[] args) {
    for(int i=0; i<6; i++)
      switch(i) {
        case 0:
          System.out.println("i is zero.");
          break;
        case 1:
          System.out.println("i is one.");
          break;
        case 2:
          System.out.println("i is two.");
          break;
        case 3:
          System.out.println("i is three.");
          break;
        default:
          System.out.println("i is greater than 3.");
      }
  }
}
```

The output produced by this program is shown here:

```
i is zero.
i is one.
i is twO
```

Iteration Statements

Java's iteration statements are **for**, **while**, and **do-while**. These statements create what we commonly call *loops*. As you probably know, a loop repeatedly executes the same set of instructions until a termination condition is met.

➢ while

- The **while** loop is Java's most fundamental loop statement.
- It repeats a statement or block while its controlling expression is true.

Here is its general form:

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

The *condition* can be any Boolean expression. The body of the loop will be executed as long as the conditional expression is true. When *condition* becomes false, control passes to the next line of code immediately following the loop. The curly braces are unnecessary if only a single statement is being repeated.
Here is a **while** loop that counts down from 10, printing exactly ten lines of "tick":

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

When you run this program, it will "tick" ten times:

```
tick    10
tick    9
tick    8
tick    7
tick    6
tick    5
tick    4
tick    3
tick    2
tick    1
```

➢ do-while

The **do-while** loop always executes its body at least once, because its conditional expression is at the bottom of the loop.

Its general form is

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

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

```
// Demonstrate the do-while loop.

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

Declaring Loop Control Variables Inside the for Loop

Often the variable that controls a **for** loop is needed only 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**. For example, here is the preceding program recoded so that the loop control variable **n** is declared as an **int** inside the **for**:

```
// Declare a loop control variable inside the for.

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

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. (That is, the scope of the variable is limited to the **for** loop.) Outside the **for** loop, the variable will cease to exist. If you need to use the loop control variable elsewhere in your program, you will not be able to declare it inside the **for** loop.

The For-Each Version of the for Loop

- A second form of **for** implements a "for-each" style loop. As you may know, contemporary language theory has embraced the for-each concept, and it has become a standard feature that programmers have come to expect.
- A for-each style loop is designed to cycle through a collection of objects, such as an array, in strictly sequential fashion, from start to finish. In Java, the for-each style of **for** is also referred to as the *enhanced* **for** loop.
- The general form of the for-each version of the **for** is shown here:

for(*type itr-var* : *collection*) *statement-block*

```
// Use a for-each style for loop.

 class ForEach {
public static void main(String[] args) {
int[] nums = { 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 };
int sum = 0;
// use for-each style for to display and sum the values for(int x : nums) {
System.out.println("Value is: " + x); sum += x;
}
System.out.println("Summation: " + sum);
}
}
```

The output from the program is shown here:

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

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:

```
// Loops may be nested.
 class Nested {
    public static void main(String[] args) { int i, j;
    for(i=0; i<10; i++) { for(j=i; j<10; j++)
            System.out.print("."); System.out.println();
}
}
}
```

The output produced by this program is shown here:

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

Jump Statements

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

➢ Using break

- In Java, the **break** statement has three uses.
- First, as you have seen, it terminates a statement sequence in a **switch** statement.
- Second, it can be used to exit a loop.
- Third, it can be used as a "civilized" form of goto.

Using break to Exit a Loop

By using **break**, you can force immediate termination of a loop, bypassing the conditional expression and any remaining code in the body of the loop. When a **break** statement is encountered inside a loop, the loop is terminated and program control resumes at the next statement following the loop. Here is a simple example:

```
// Using break to exit a loop.
class BreakLoop {
    public static void main(String[] args) {
     for(int i=0; i<100; i++) {
        if(i == 10) break; // terminate loop if i is 10
        System.out.println("i: " + i);
        }
        System.out.println("Loop complete.");
        }
        }
```

This program generates the following output:

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

Using break as a Form of Goto

In addition to its uses with the **switch** statement and loops, the **break** statement can also be employed by itself to provide a "civilized" form of the goto statement.

```
// Using break as a civilized form of goto.

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

Running this program generates the following output:

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

➢ Using continue

- In **while** and **do-while** loops, a **continue** statement causes control to be transferred directly to the conditional expression that controls the loop.
- In a **for** loop, control goes first to the iteration portion of the **for** statement and then to the conditional expression. For all three loops, any intermediate code is bypassed.
- Here is an example program that uses **continue** to cause two numbers to be printed on each line:

```
// Demonstrate continue.

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

Here is the output from this program:

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

➢ return

- The last control statement is **return**. The **return** statement is used to explicitly return from a method. That is, it causes program control to transfer back to the caller of the method.
- At any time in a method, the **return** statement can be used to cause execution to branch back to the caller of the method. Thus, the **return** statement immediately terminates the method in which it is executed.
- The following example illustrates this point. Here, **return** causes execution to return to the Java run-time system, since it is the run-time system that calls **main( )**:

```
// Demonstrate return.
class Return {
public static void main(String[] args) { boolean t = true;
    System.out.println("Before the return.");

    if(t) return; // return to caller

    System.out.println("This won't execute.");
    }
    }
```

The output from this program is shown here:

```
Before the return.
```