Object Oriented Programming & Methodology

Unit-3 Notes

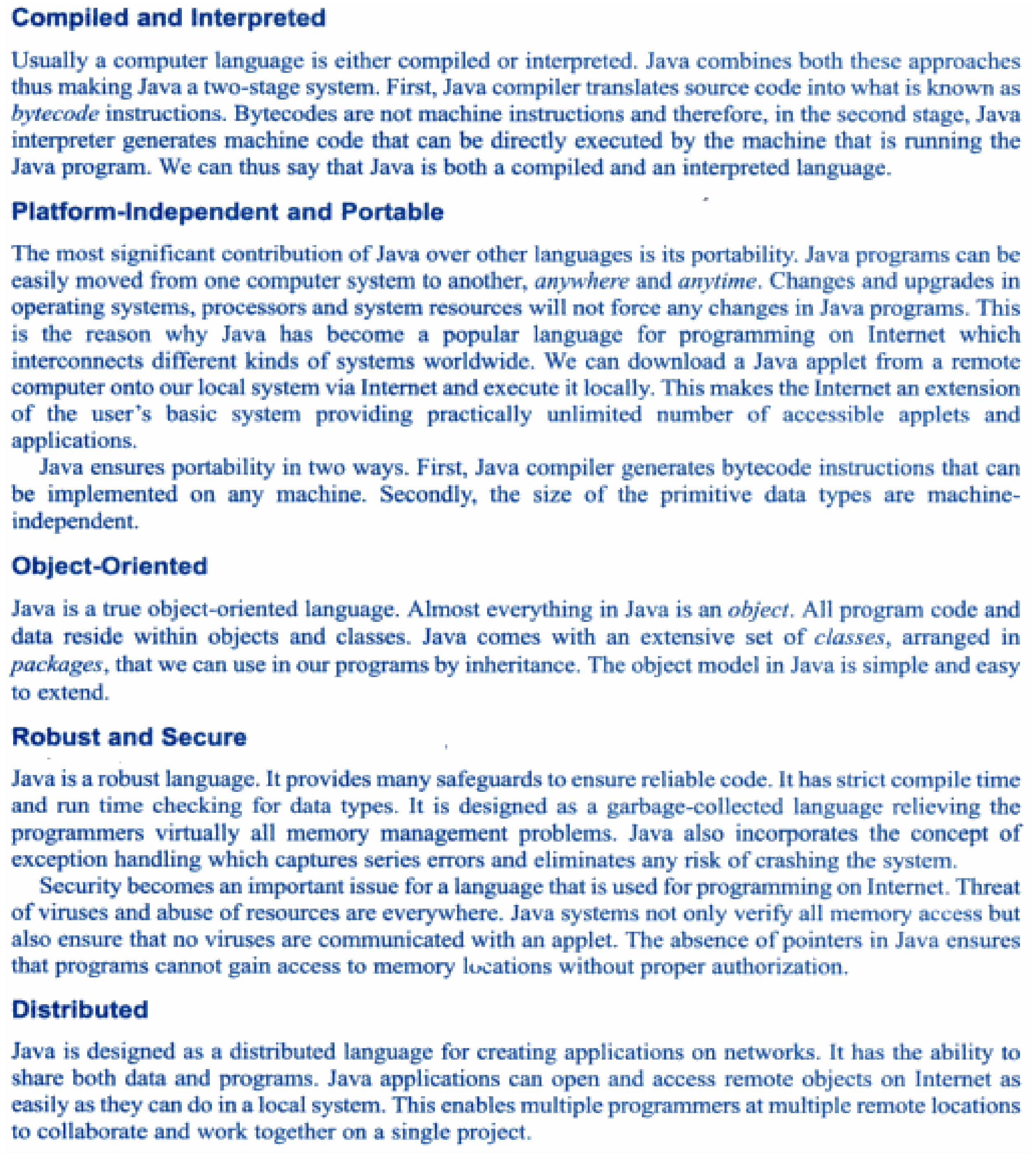
Textbooks to study from:

- Herbert Schieldt, “The Complete Reference: Java”, TMH, 7th Edition.
- E. Balagurusamy, “Programming in JAVA”, TMH, 4th Edition.

Introduction

- **Java Features**

Java is a general purpose, object oriented programming language developed by Sun Microsystems of USA in 1991. Java invertors wanted the language to be not only reliable, portable and distributed but also simple, compact and interactive. java describes following features:



Compiled and Interpreted

Usually a computer language is either compiled or interpreted. Java combines both these approaches thus making Java a two-stage system. First, Java compiler translates source code into what is known as *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. We can thus say that Java is both a compiled and an interpreted language.

Platform-Independent and Portable

The most significant contribution of Java over other languages is its portability. Java programs can be easily moved from one computer system to another, *anywhere* and *anytime*. Changes and upgrades in operating systems, processors and system resources will not force any changes in Java programs. This is the reason why Java has become a popular language for programming on Internet which interconnects different kinds of systems worldwide. We can download a Java applet from a remote computer onto our local system via Internet and execute it locally. This makes the Internet an extension of the user's basic system providing practically unlimited number of accessible applets and applications.

Java ensures portability in two ways. First, Java compiler generates bytecode instructions that can be implemented on any machine. Secondly, the size of the primitive data types are machine-independent.

Object-Oriented

Java is a true object-oriented language. Almost everything in Java is an *object*. All program code and data reside within objects and classes. Java comes with an extensive set of *classes*, arranged in *packages*, that we can use in our programs by inheritance. The object model in Java is simple and easy to extend.

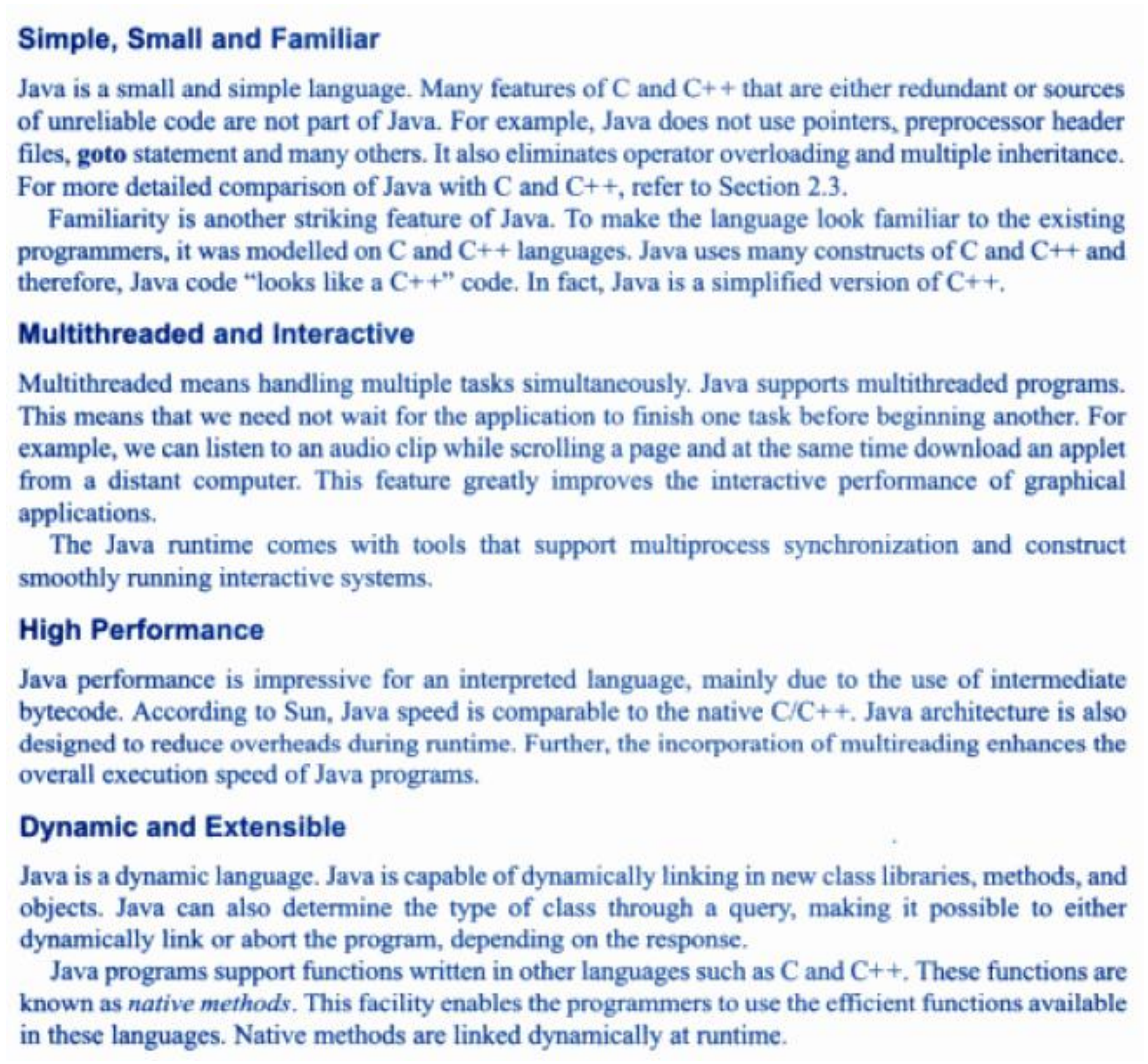
Robust and Secure

Java is a robust language. It provides many safeguards to ensure reliable code. It has strict compile time and run time checking for data types. It is designed as a garbage-collected language relieving the programmers virtually all memory management problems. Java also incorporates the concept of exception handling which captures series errors and eliminates any risk of crashing the system.

Security becomes an important issue for a language that is used for programming on Internet. Threat of viruses and abuse of resources are everywhere. Java systems not only verify all memory access but also ensure that no viruses are communicated with an applet. The absence of pointers in Java ensures that programs cannot gain access to memory locations without proper authorization.

Distributed

Java is designed as a distributed language for creating applications on networks. It has the ability to share both data and programs. Java applications can open and access remote objects on Internet as easily as they can do in a local system. This enables multiple programmers at multiple remote locations to collaborate and work together on a single project.



Simple, Small and Familiar

Java is a small and simple language. Many features of C and C++ that are either redundant or sources of unreliable code are not part of Java. For example, Java does not use pointers, preprocessor header files, **goto** statement and many others. It also eliminates operator overloading and multiple inheritance. For more detailed comparison of Java with C and C++, refer to Section 2.3.

Familiarity is another striking feature of Java. To make the language look familiar to the existing programmers, it was modelled on C and C++ languages. Java uses many constructs of C and C++ and therefore, Java code "looks like a C++" code. In fact, Java is a simplified version of C++.

Multithreaded and Interactive

Multithreaded means handling multiple tasks simultaneously. Java supports multithreaded programs. This means that we need not wait for the application to finish one task before beginning another. For example, we can listen to an audio clip while scrolling a page and at the same time download an applet from a distant computer. This feature greatly improves the interactive performance of graphical applications.

The Java runtime comes with tools that support multiprocess synchronization and construct smoothly running interactive systems.

High Performance

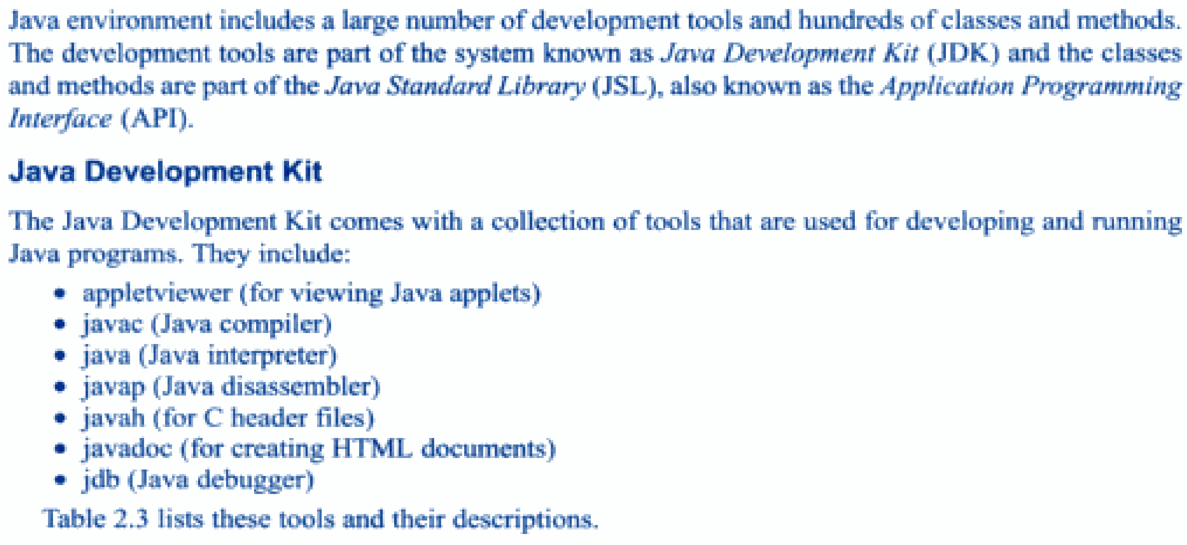
Java performance is impressive for an interpreted language, mainly due to the use of intermediate bytecode. According to Sun, Java speed is comparable to the native C/C++. Java architecture is also designed to reduce overheads during runtime. Further, the incorporation of multireading enhances the overall execution speed of Java programs.

Dynamic and Extensible

Java is a dynamic language. Java is capable of dynamically linking in new class libraries, methods, and objects. Java can also determine the type of class through a query, making it possible to either dynamically link or abort the program, depending on the response.

Java programs support functions written in other languages such as C and C++. These functions are known as *native methods*. This facility enables the programmers to use the efficient functions available in these languages. Native methods are linked dynamically at runtime.

- **Java Environment**



Java environment includes a large number of development tools and hundreds of classes and methods. The development tools are part of the system known as *Java Development Kit* (JDK) and the classes and methods are part of the *Java Standard Library* (JSL), also known as the *Application Programming Interface* (API).

Java Development Kit

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

- appletviewer (for viewing Java applets)
- javac (Java compiler)
- java (Java interpreter)
- javap (Java disassembler)
- javah (for C header files)
- javadoc (for creating HTML documents)
- jdb (Java debugger)

Table 2.3 lists these tools and their descriptions.

Tool	*Description*
appletviewer	Enables us to run Java applets (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 sourcecode to bytecode files that the interpreter can understand.
javadoc	Creates HTML-format documentation from Java source code files.
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 us to find errors in our programs.

The way these tools are applied to build and run application programs is illustrated in Fig. 2.5. To create a Java program, we need to create a source code file using a text editor. The source code is then compiled using the Java compiler **javac** and executed using the Java interpreter **java**. The Java debugger **jdb** is used to find errors, if any, in the source code. A compiled Java program can be converted into a source code with the help of Java disassembler **javap**. We learn more about these tools as we work through the book.

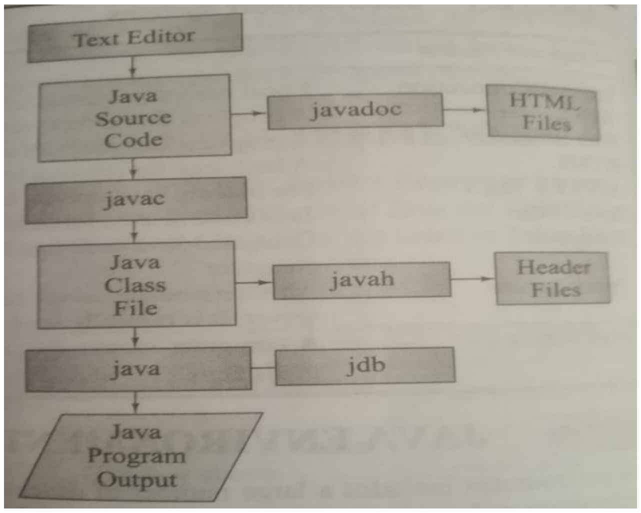


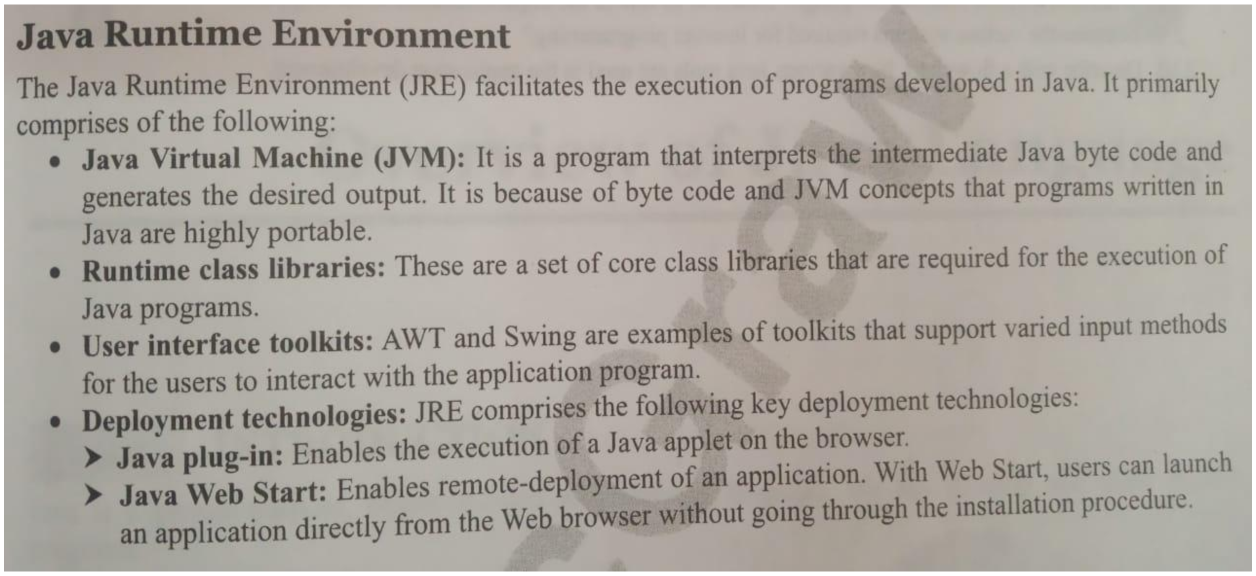
Fig 2.5 Process of building and running Java applications.

Application Programming Interface

The Java Standard Library (or API) includes hundreds of classes and methods grouped into several functional packages (see Appendix G). Most commonly used packages are:

- **Language Support Package:** A collection of classes and methods required for implementing basic features of Java.
- **Utilities Package:** A collection of classes to provide utility functions such as date and time functions.
- **Input/Output Package:** A collection of classes required for input/output manipulation.
- **Networking Package:** A collection of classes for communicating with other computers via Internet.
- **AWT Package:** The Abstract Window Tool Kit package contains classes that implements platform-independent graphical user interface.
- **Applet Package:** This includes a set of classes that allows us to create Java applets.

The use of these library classes will become evident when we start developing Java programs.

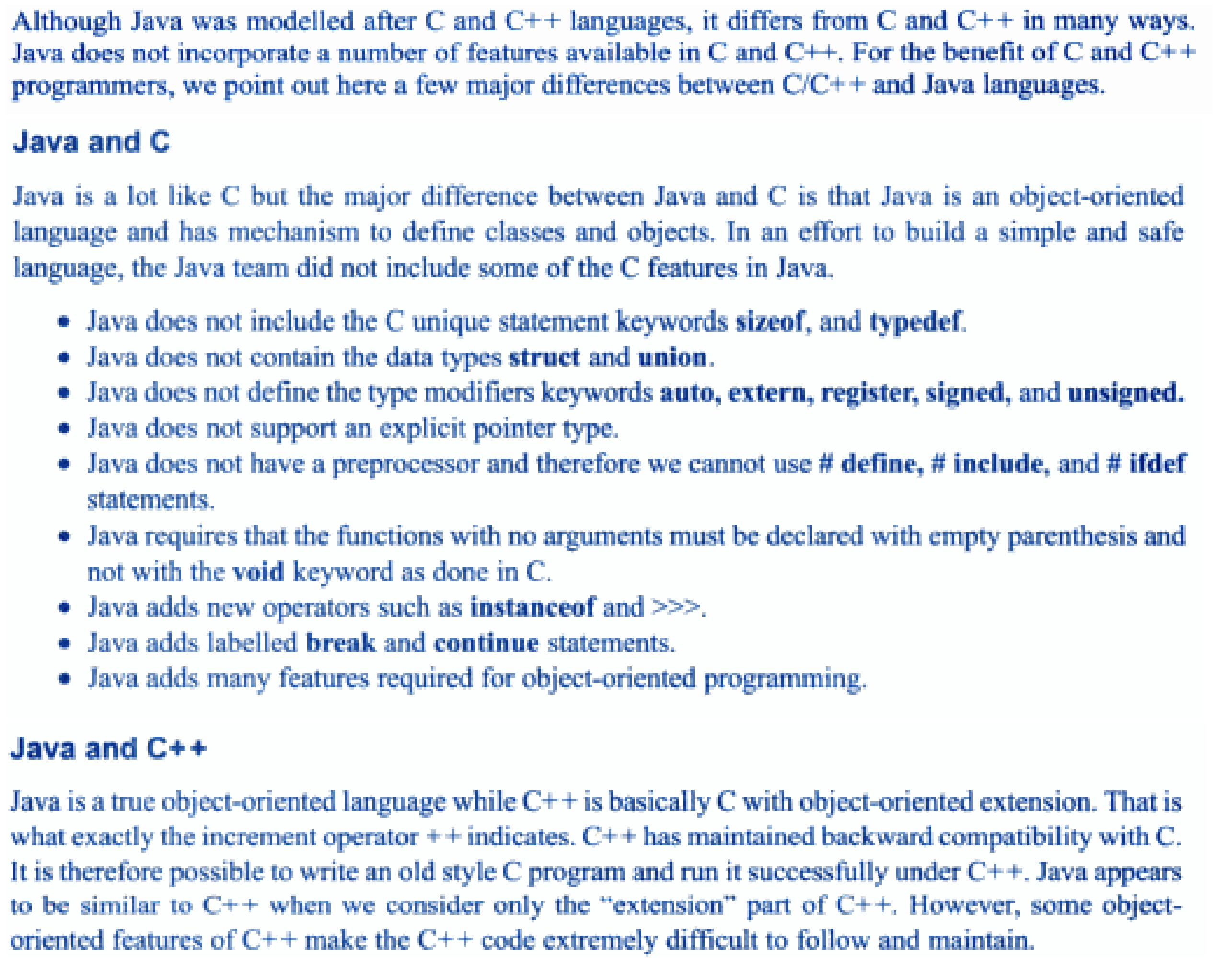


Java Runtime Environment

The Java Runtime Environment (JRE) facilitates the execution of programs developed in Java. It primarily comprises of the following:

- **Java Virtual Machine (JVM):** It is a program that interprets the intermediate Java byte code and generates the desired output. It is because of byte code and JVM concepts that programs written in Java are highly portable.
- **Runtime class libraries:** These are a set of core class libraries that are required for the execution of Java programs.
- **User interface toolkits:** AWT and Swing are examples of toolkits that support varied input methods for the users to interact with the application program.
- **Deployment technologies:** JRE comprises the following key deployment technologies:
  - **Java plug-in:** Enables the execution of a Java applet on the browser.
  - **Java Web Start:** Enables remote-deployment of an application. With Web Start, users can launch an application directly from the Web browser without going through the installation procedure.

- **How Java differs from C and C++**



Although Java was modelled after C and C++ languages, it differs from C and C++ in many ways. Java does not incorporate a number of features available in C and C++. For the benefit of C and C++ programmers, we point out here a few major differences between C/C++ and Java languages.

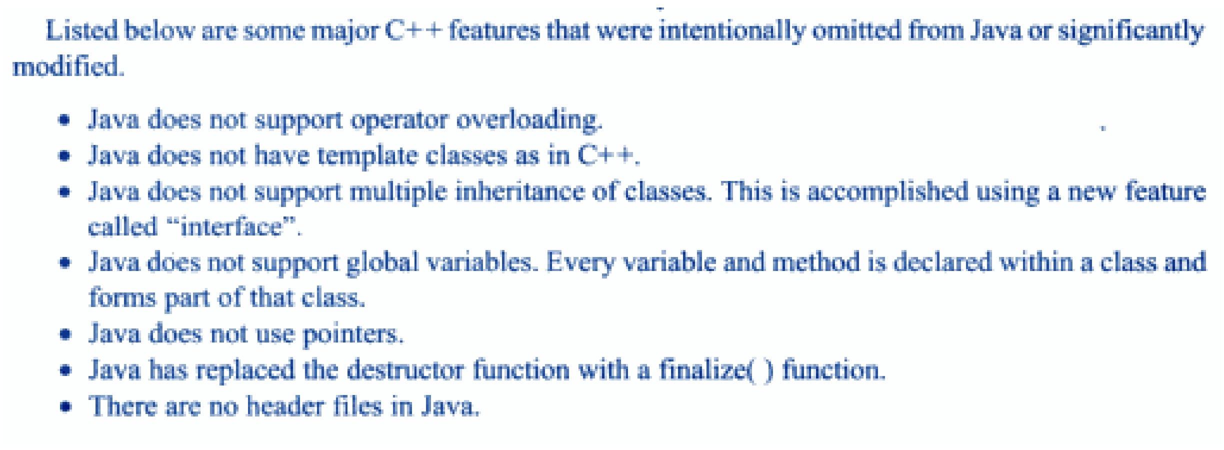
Java and C

Java is a lot like C but the major difference between Java and C is that Java is an object-oriented language and has mechanism to define classes and objects. In an effort to build a simple and safe language, the Java team did not include some of the C features in Java.

- Java does not include the C unique statement keywords **sizeof**, and **typedef**.
- Java does not contain the data types **struct** and **union**.
- Java does not define the type modifiers keywords **auto, extern, register, signed,** and **unsigned.**
- Java does not support an explicit pointer type.
- Java does not have a preprocessor and therefore we cannot use **# define, # include**, and **# ifdef** statements.
- Java requires that the functions with no arguments must be declared with empty parenthesis and not with the **void** keyword as done in C.
- Java adds new operators such as **instanceof** and >>>.
- Java adds labelled **break** and **continue** statements.
- Java adds many features required for object-oriented programming.

Java and C++

Java is a true object-oriented language while C++ is basically C with object-oriented extension. That is what exactly the increment operator ++ indicates. C++ has maintained backward compatibility with C. It is therefore possible to write an old style C program and run it successfully under C++. Java appears to be similar to C++ when we consider only the "extension" part of C++. However, some object-oriented features of C++ make the C++ code extremely difficult to follow and maintain.


Listed below are some major C++ features that were intentionally omitted from Java or significantly modified.

- Java does not support operator overloading.
- Java does not have template classes as in C++.
- Java does not support multiple inheritance of classes. This is accomplished using a new feature called "interface".
- Java does not support global variables. Every variable and method is declared within a class and forms part of that class.
- Java does not use pointers.
- Java has replaced the destructor function with a finalize( ) function.
- There are no header files in Java.

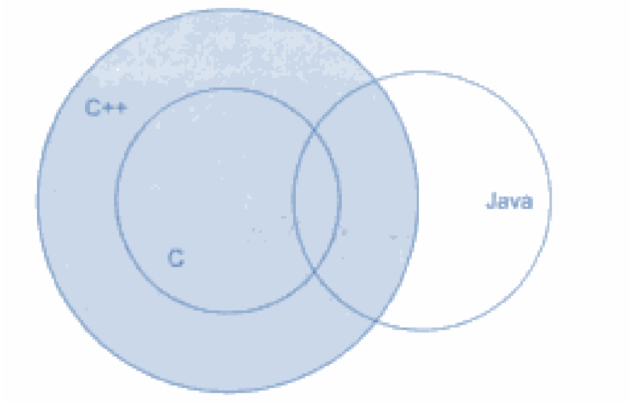


Fig: Overlapping of C, C++ and Java

- **Java Tokens**

Here we will describe the atomic elements of Java. Java programs are a collection of tokens, comments and white spaces. There are five Java types of tokens: Keywords, identifiers, literals, operators and separators. The operators are described in the next section.

Identifiers

Identifiers are used for class names, method names, and variable names. 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. Again, Java is case-sensitive, so **VALUE** is a different identifier than **Value**.

Some examples of **valid** identifiers are:

AvgTemp count a4 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.

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.

The Java Keywords
There are 50 keywords currently defined in the Java language (see Table 2-1). These keywords, combined with the syntax of the operators and separators, form the foundation of the Java language. These keywords cannot be used as names for a variable, class, or method.

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

NOTE:
you don't need to memorize all keywords for exam.

In addition to the keywords, Java reserves the following: **true**, **false**, and **null**. These are values defined by Java. You may not use these words for the names of variables, classes, and so on.

- ## Comments

Like most other programming languages, Java lets you enter a remark into a program's source file. The contents of a comment are ignored by the compiler. Instead, a comment describes or explains the operation of the program to anyone who is reading its source code. In real applications, comments generally explain how some part of the program works or what a specific feature does. There are three types of comments defined by Java.

single-line comment:

```
// Your program begins with a call to main().
```

A single-line comment begins with a **//** and ends at the end of the line.

multiline comment:

```
/*
This is a simple Java program.
Call this file "Example.java".
*/
```

This type of comment must begin with **/*** and end with ***/**. Anything between these two comment symbols is ignored by the compiler. As the name suggests, a multiline comment may be several lines long.

documentation comment:

The documentation comment begins with a **/**** and ends with a ***/**. After the beginning **/****, the first line or lines become the main description of your class, variable, or method.

```
/**
* This class draws a bar chart.
* @author Ramesh
* @version 3.2
*/
```

- ## System Class

System class is a class predefined by Java that is automatically included in your programs. Java environment relies on several built-in class libraries that contain many built-in methods that provide support for things as I/O, string handling, networking, and graphics. Two of Java's built-in methods are **println( )** and **print( )**.

- ## 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. All of Java's control statements are examined here.

Selection Statements
Java supports two selection statements: **if** and **switch**. These statements allow you to control the flow of your program's execution based upon conditions known only during run time.
Here is the general form of the **if** statement:

```
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.

Nested ifs
A *nested* **if** is an **if** statement that is the target of another **if** or **else**. When you nest **if**s, 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.**

```
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 **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. If none of the conditions is true, then the final **else** statement will be executed. The final **else** acts as a default condition; that is, if all other conditional tests fail, then the last **else** statement is performed. If there is no final **else** and all other conditions are **false**, then no action will take place.

Example:

```
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";
```

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. As such, it often provides a better alternative than a large series of **if-else-if** statements. Here is the general form of a **switch** statement:

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

Example:

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

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

Iteration Statements
Java's iteration statements are **for**, **while**, and **do-while**. These statements create what we commonly call *loops.*
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.

Example:

```
class While {
```

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

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. As with all of Java's loops, *condition* must be a Boolean expression.

Example:

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

for

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

Here is the general form of the traditional **for** statement:

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

The **for** loop operates as follows. When the loop first starts, the *initialization* portion of the loop is executed. This is an expression that sets the value of the *loop control variable* and the initialization expression is only executed once. Next, *condition* is evaluated. This must be a Boolean expression. It usually tests the loop control variable against a target value. If this expression is true, then the body of the loop is executed. If it is false, the loop terminates.

Next, the *iteration* portion of the loop is executed. This is usually an expression that increments or decrements the loop control variable.

Example:

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

The For-Each Version of the for Loop

Second form of **for** was defined that implements a “for-each” style loop. 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

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. 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.

To understand the motivation behind a for-each style loop, consider the type of **for** loop that it is designed to replace.

The following fragment uses a traditional **for** loop to compute the sum of the values in an array:

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

here is the preceding fragment rewritten using a for-each version of the **for**:

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

Operators

Java provides a rich operator environment. Most of its operators can be divided into the following four groups: arithmetic, bitwise, relational, and logical.

- ## Arithmetic Operators

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

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

arithmetic using integers

```
int a = 1 + 1;
int b = a * 3;
int c = b / 4;
int d = c - a;
int e = -d;
int x = 42;
y= x % 10;
```

```
a = 2
b = 6
c = 1
d = -1
e = 1
y=2
```

arithmetic using doubles

```
double da = 1 + 1;
double db = da * 3;
double dc = db / 4;
double dd = dc - a;
double de = -dd;
int dx = 42;
dy= dx % 10;
```

```
da = 2.0
db = 6.0
dc = 1.5
dd = -0.5
de = 0.5
dy= 2.25
```

Arithmetic Compound Assignment Operators

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

```
a = a + 4;
```

In Java, you can rewrite this statement as shown here:

```
a += 4;
```

This version uses the **+=** compound assignment operator. Both statements perform the same action: they increase the value of **a** by 4.

```
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 increment operator increases its operand by one. The decrement operator decreases its operand by one. For example, this statement:

```
x = x + 1;
```

can be rewritten like this by use of the increment operator:

```
x++;
```

Similarly, this statement:

```
x = x - 1;
```

is equivalent to

```
x--;
```

These operators are unique in that they can appear both in postfix form, where they follow the operand as just shown, and prefix form, where they precede the operand.

is modified. For example:

```
x = 42;
y = ++x;
```

In this case, **y** is set to 43 as you would expect, because the increment occurs before **x** is assigned to **y**. Thus, the line **y = ++x;** is the equivalent of these two statements:

```
x = x + 1;
y = x;
```

However, when written like this,

```
x = 42;
y = x++;
```

the value of **x** is obtained before the increment operator is executed, so the value of **y** is 42. Of course, in both cases **x** is set to 43. Here, the line **y = x++;** is the equivalent of these two statements:

```
y = x;
x = x + 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

The Bitwise NOT
Also called the bitwise complement, the unary NOT operator, **~**, inverts all of the bits of its operand. For example, the number 42, which has the following bit pattern:
00101010
becomes
11010101
after the NOT operator is applied.

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

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

The Bitwise OR
The OR operator, |, combines bits such that if either of the bits in the operands is a 1, then the resultant bit is a 1, as shown here:

```
        00101010 (42)
|       00001111 (15)
------------------------------------
        00101111 47
```

The Bitwise XOR
The XOR operator, ^, combines bits such that if exactly one operand is 1, then the result is 1. Otherwise, the result is zero. The following example shows the effect of the ^.
Notice how the bit pattern of 42 is inverted wherever the second operand has a 1 bit. Wherever the second operand has a 0 bit, the first operand is unchanged.

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

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
Here, num specifies the number of positions to left-shift the value in value. That is, the **<<** moves all of the bits in the specified value to the left by the number of bit positions specified by num.

a = 00 0100 0000 (64)

i = a << 2

i = 01 0000 0000 (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

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

a = 00100011 (35)
i = a >> 2
i= 00001000 (8)

When you are shifting right, the top (leftmost) bits exposed by the right shift are filled in with the previous contents of the top bit. This is called sign extension and serves to preserve the sign of negative numbers when you shift them right. For example, –8 >> 1 is –4, which, in binary, is

a = 11111000 (–8)
i = a >>1
i = 11111100 (–4)

The Unsigned Right Shift
As you have just seen, the **>>** operator automatically fills the high-order bit with its previous contents each time a shift occurs. This preserves the sign of the value. However, sometimes this is undesirable. When we shift a zero into the high-order bit no matter what its initial value was. This is known as an unsigned shift. To accomplish this, you will use Java's unsigned, shift-right operator, **>>>**, which always shifts zeros into the high-order bit.

The following code fragment demonstrates the **>>>**. Here, **a** is set to –1, which sets all 32 bits to 1 in binary. This value is then shifted right 24 bits, filling the top 24 bits with zeros, ignoring normal sign extension. This sets **a** to 255.
```
int a = -1;
a = a >>> 24;
```

11111111 11111111 11111111 11111111 –1 in binary as an int

```
>>>24
00000000 00000000 00000000 11111111 255 in binary as an int
```

Bitwise Operator Compound Assignments

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

```
int a = 1;
int b = 2;
int c = 3;
a |= 4;
b >>= 1;
c <<= 1;
a ^= c;
```

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

- ## Relational Operators

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

The outcome of these operations is a **boolean** value. Only numeric types can be compared using the ordering operators. That is, only integer, floating-point, and character operands may be compared to see which is greater or less than the other.
For example,

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

In this case, the result of **a<b** (which is **false**) is stored in **c**.

If you are coming from a C/C++ background, please note the following. In C/C++, these types of statements are very common:

```
int done;
// ...
if(!done) ... // Valid in C/C++
if(done) ... // but not in Java.
```

In Java, these statements must be written like this:

```
if(done == 0) ... // This is Java-style.
if(done != 0) ...
```

• Boolean Logical Operators

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

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

The logical Boolean operators, **&**, **|**, and **^**, operate on **boolean** values in the same way that they operate on the bits of an integer.

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

Example:

```
boolean a = true;
boolean b = false;
boolean c = a | b;
boolean d = a & b;
boolean e = a ^ b;
boolean f = (!a & b) | (a & !b);
boolean g = !a;

a = true
```

```
b = false
a|b = true
a&b = false
a^b = true
a&b|a&!b = true
!a = false
```

Short-Circuit Logical Operators

These are secondary versions of the Boolean AND and OR operators, and are known as short-circuit logical operators. As you can see from the preceding table, the OR operator results in **true** when **A** is **true**, no matter what **B** is. Similarly, the AND operator results in **false** when **A** is **false**, no matter what **B** is. If you use the **||** and **&&** forms, rather than the **|** and **&** forms of these operators, Java will not bother to evaluate the right-hand operand when the outcome of the expression can be determined by the left operand alone. This is very useful when the right-hand operand depends on the value of the left one in order to function properly.

Example:

We can take advantage of short-circuit logical evaluation to be sure that a division operation will be valid before evaluating it:

```
if (denom != 0 && num / denom > 10)
```

Since the short-circuit form of AND (**&&**) is used, there is no risk of causing a divide by zero error when **denom** is zero. If this line of code were written using the single **&** version of AND, both sides would be evaluated, causing a divide by zero error when **denom** is zero.

It is standard practice to use the short-circuit forms of AND and OR in cases involving Boolean logic, leaving the single-character versions exclusively for bitwise operations.

The Assignment Operator

The assignment operator is the single equal sign, **=**. The assignment operator works in Java much as it does in any other computer language. It has this general form:

```
var = expression;
```

The assignment allows you to create a chain of assignments. For example, consider this fragment:

```
int x, y, z;
x = y = z = 100; // set x, y, and z to 100
```

This fragment sets the variables **x**, **y**, and **z** to 100 using a single statement. This works because the **=** is an operator that yields the value of the right-hand expression Thus, the value of **z = 100** is 100, which is then assigned to **y**, which in turn is assigned to **x**. Using a "chain of assignment" is an easy way to set a group of variables to a common value.

The ? Operator

Java includes a special ternary (three-way) operator that can replace certain types of if-then-else statements.

The **?** has this general form:
expression1 **?** expression2 **:** expression3
Here, expression1 can be any expression that evaluates to a **boolean** value. If expression1 is **true**, then expression2 is evaluated; otherwise, expression3 is evaluated.

Both expression2 and expression3 are required to return the same type, which **can't** be **void**. Here is an example of the way that the **?** is employed:

```
demon =0,num=50;
ratio = denom == 0 ? 0 : num / denom;
ratio=0;
demon=10,num=50;
ratio = denom == 0 ? 0 : num / denom;
ratio = 5;
```

- ## Operator Precedence

The Precedence of the Java Operators
Highest

( )	[ ]	.	
++	– –	~	!
*	/	%	
+	–		
>>	>>>	<<	
>	>=	<	<=
==	!=		
&			
^			
|			
&&			
||			
?:			
=	op=		

Lowest

the first row shows items: parentheses, square brackets, and the dot operator. Parentheses are used to alter the precedence of an operation, the square brackets provide array indexing, the dot operator is used to dereference objects.

Data Types

As with all modern programming languages, Java supports several types of data. Data types specify size and type of values that can be stored. Data types are divided into two groups:

- Primitive data types - includes byte, short, int, long, float, double, boolean and char
- Non-primitive data types - such as String, Arrays and Classes

• Primitive Data Types

Java defines eight primitive types of data: **byte**, **short**, **int**, **long**, **char**, **float**, **double**, and **boolean**. These can be put in four groups:
• Integers This group includes **byte**, **short**, **int**, and **long**, which are for whole-valued signed numbers.
• Floating-point numbers This group includes **float** and **double**, which represent numbers with fractional precision.
• Characters This group includes **char**, which represents symbols in a character set, like letters and numbers.
• Boolean This group includes **boolean**, which is a special type for representing true/false values.

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

Floating-Point Types

Floating-point numbers, also known as real numbers, are used when evaluating expressions that require fractional precision. There are two kinds of floating-point types, **float** and **double**, which represent single- and double-precision numbers, respectively. Their width and ranges are shown here:

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

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. Doing so is, of course, redundant.

Characters

In Java, the data type used to store characters is **char.** Java **char** is a 16-bit type. The range of a **char** is 0 to 65,536. The standard set of characters known as ASCII still ranges from 0 to 127 as always, and the
extended 8-bit character set, ISO-Latin-1, ranges from 0 to 255. 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 '@'. There are several escape sequences that allow you to enter the character you need, such as '\'' for the single-quote character itself and **'\n'** for the newline character.

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**. 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.

Type Conversion and Casting
Type Conversion is to assign a value of one type to a variable of another type. If the two types are compatible,
then Java will perform the conversion automatically. it is still possible to obtain a conversion between
incompatible types. To do so, you must use a cast, which performs an explicit conversion between incompatible types.

Java's Automatic Conversions
When one type of data is assigned to another type of variable, an automatic type conversion will take place if the following two conditions are met:
• 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.

Casting Incompatible Types
if you want to assign an **int** value to a **byte** variable? This conversion will not be performed automatically, because a **byte** is smaller than an **int**. This kind of conversion is sometimes called a narrowing conversion, since you are explicitly making the value narrower so that it will fit into the target type.
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
```

Here, target-type specifies the desired type to convert the specified value to.

```
int a;
byte b;
// ...
b = (byte) a;
```

the fragment above casts an **int** to a **byte**.

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

• Wrapper classes in Java

Java uses primitive types (also called simple types), such as **int** or **double**, to hold the basic data types supported by the language. The **wrapper class in Java** provides the mechanism to convert primitive into object and object into primitive. Despite the performance benefit offered by the primitive types, there are times when you will need an object representation. For example, many of the standard data structures implemented by Java operate on objects, which means that you can't use these data structures to store primitive types. To handle these (and other) situations, Java provides type wrappers, which are classes that encapsulate a primitive type within an object. The type wrappers are **Double**, **Float**, **Long**, **Integer**, **Short**, **Byte**, **Character**, and **Boolean**.
The following code snippet demonstrates how to use a numeric type wrapper to encapsulate a value and then extract that value.

```
Integer iOb = new Integer(100); //boxing
int i = iOb.intValue(); //unboxing
```

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**. The process of encapsulating a value within an object is called boxing. The process of extracting a value from a type wrapper is called unboxing.

Autoboxing
Autoboxing is the process by which a primitive type is automatically encapsulated (boxed) into its equivalent type wrapper whenever an object of that type is needed. There is no need to explicitly construct an object

```
Integer iOb = 100; // autobox an int
```

Auto-unboxing is the process by which the value of a boxed object is automatically extracted (unboxed) from a type wrapper when its value is needed.

```
int i = iOb; // auto-unbox;
```

- **Non-Primitive Data Types**

Arrays
An array is a group of contiguous or related data item that share a common name. A specific element in an array is accessed by its index. Arrays offer a convenient means of grouping related information.

For example, the following declares an array named **month_days** with the type "array of int":
int month_days[];

This declaration establishes the fact that **month_days** is an array variable, but no array actually exists. To link **month_days** with an actual, physical array of integers, you must allocate one using **new** and assign it to **month_days**. **new** is a special operator that allocates memory.

This example allocates a 12-element array of integers and links them to **month_days**.
month_days = new int[12];

After this statement executes, **month_days** will refer to an array of 12 integers. Further, all elements in the array will be initialized to zero. Once you have allocated an array, you can access a specific element in the array by specifying its index within square brackets.
month_days[1] = 28;

Arrays can also be initialized when they are declared. An array initializer is a list of comma-separated expressions surrounded by curly braces.

int month_days[] = { 31, 28, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31 };

String
The String type is used to declare string variables. You can also declare arrays of strings.
A quoted string constant can be assigned to a **String** variable

For example, consider the following fragment:
String str = "this is a test";

Here, **str** is an object of type **String**. It is assigned the string "this is a test".

Variables

A variable is an identifier that denotes a storage location used to store a data value. A variable may take different values at different times during the execution of program. Variable names may consists of alphabets, digits, the underscore and dollar characters, subject to following conditions.

- They must not begin with a digit.
- Uppercase and lowercase variables are distinct. This means that variable **VALUE** is different from **value**.
- It should not be keyword.
- Whitespaces are not allowed.
- Variables name can be any length.

Declaration of Variables:

Variable declaration does three things.

- It tells compiler what the variable name is.
- It specifies what type data variable will hold.
- The path of declaration decides the scope of variable.

A variable must be declared before it is used in the program.

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'.
```

The Scope and Lifetime of Variables

In Java, the two major scopes are those defined by a class and those defined by a method.

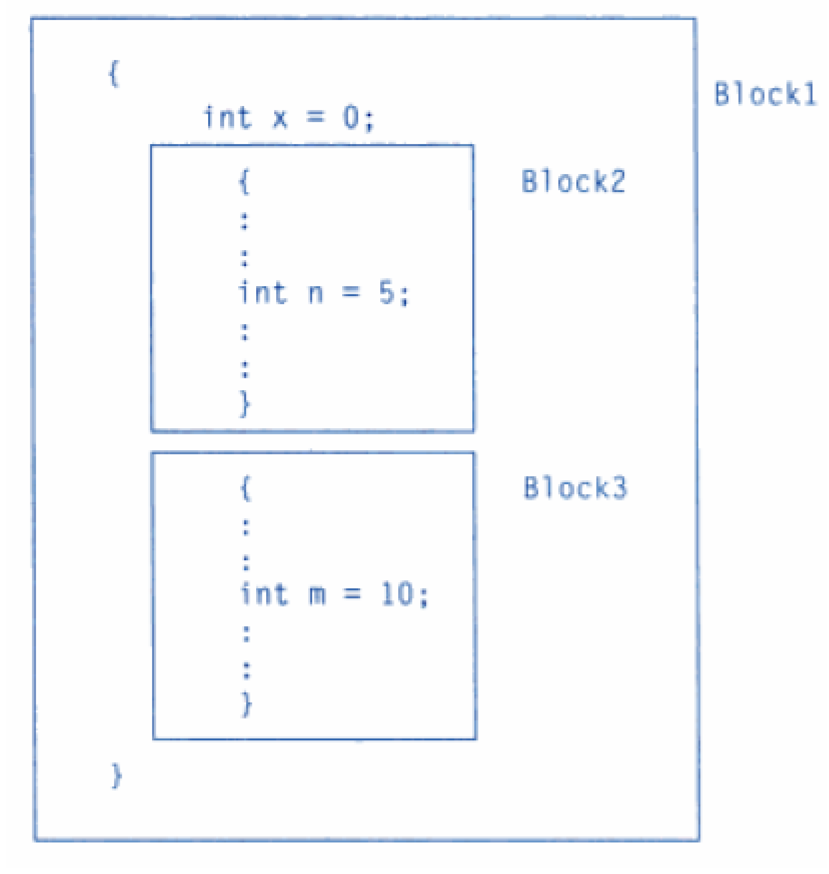
The data, or variables, defined within a **class** are called **instance variables**. Variables defined within a class are called instance variables because each instance of the class (that is, each object of the class) contains its own copy of these variables. Thus, the data for one object is separate and unique from the data for another.

An instance variable must be accessed only through an object of its class. However, it is possible to create a variable that can be used by itself, without reference to a specific object. To create such a variable, we precede its declaration with the keyword **static**. When a variable is declared **static**, it can be accessed before any objects of its class are created, and without reference to any object.

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**.

The variables declared and used inside **methods** are called **local variables**. 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. These variables are not available for use outside method definition.

Local variables can also be declared inside program blocks that are defined between an opening brace"{" and closing brace "}". These variables are visible to program only from beginning of its program block and the end of program block. The following code segment show nested program block i.e. program block within program block.



```
{
      int x = 0;
        {
        :
        :
        int n = 5;
        :
        :
        }
        {
        :
        :
        int m = 10;
        :
        :
        }
}
```

The scope of variable in the above code snippet is nested. The outer scope encloses the inner scope. This means that variables declared in the outer scope will be visible to code within the inner scope. However, the reverse is not true. Variables declared within the inner scope will not be visible outside it.

Methods & Classes

• Class Fundamentals

A class creates a logical framework that defines the relationship between its members. a class is defined by specifying the data that it contains and the code that operates on that data. While very simple classes may contain only code or only data, most real-world classes contain both.
A class is declared by use of the **class** keyword.

A simplified general form of a **class** definition is shown here:

```
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);
}
}
```

The data, or variables, defined within a **class** are called *instance variables.* 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. In the example above width, height and depth are instance variables of class Box.

A class defines a new type of data. In this case, the new data type is called **Box**. Class name is used 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.

Methods are used to access the instance variables defined by the class. If the method does not return a value, its return type must be **void**. 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.

Declaring Objects

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.

In the preceding class example, the following is used to declare an object of type **Box**:
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**.

Example of a class creating object of Box class described below:

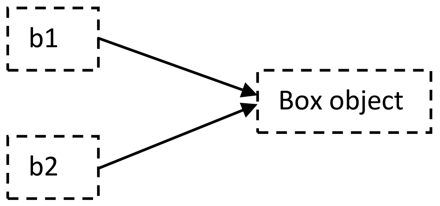
```
class BoxDemo3 {
public static void main(String args[]) {
Box mybox1 = new Box();

// assign values to mybox1's instance variables
mybox1.width = 10;
mybox1.height = 20;
mybox1.depth = 15;

// display volume of first box
mybox1.volume();
}
}
```

Assigning Object Reference Variables

Consider the following code segment:

```
Box b1 = new Box();  //statement 1
Box b2 = b1;         //statement 2
```

In this code fragment, **b1** and **b2** will both refer to the *same* object created at statement 1. 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. This situation is depicted here:



- ## Static keyword

Static keyword is used to define a class member that will be used independently of any object of that class. Normally, a class member must be accessed only through 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. Both methods and variables can be declared **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/variables.

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

```
class UseStatic {

static int a = 3;
static int b = 4;

static void meth(int x)
{
System.out.println("x = " + x);
System.out.println("a = " + a);
System.out.println("b = " + b);
}

public static void main(String args[]) {
meth(42);
}
}
```

output of the program is:

```
x=42
a=3
b=4
```

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.

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
```

- ## Constructors

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 when the object is created. Constructors have no return type, not even **void.**

The following example define a simple constructor that simply sets the dimensions of each box to the same values.

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

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

```
Constructing Box
Volume is 1000.0
```

when creating an object of Box class:

```
Box mybox1 = new Box();
```

new Box( ) is calling the **Box( )** constructor. When you do not explicitly define a constructor for a class, then Java creates a default constructor for the class. The default constructor automatically initializes all instance variables to zero.

Parameterized Constructors

The following version of **Box** defines a parameterized constructor that sets the dimensions of a box as specified by those parameters.

```
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);
double vol;

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

The output from this program is shown here:

```
Volume is 3000.0
```

As you can see, each object is initialized as specified in the parameters to its constructor.
For example, in the following line,

```
Box mybox1 = new Box(10, 20, 15);
```

the values 10, 20, and 15 are passed to the **Box( )** constructor when **new** creates the object.

• Overloading Methods

Method overloading is a way to define two or more methods within the same class that share the same name, as long as the type and/or number of their parameters are different. Method overloading is one of the ways that Java supports polymorphism.
When an overloaded method is called, Java uses the type and/or number of arguments to determine which version of the overloaded method to actually call.
While overloaded methods may have different return types, the return type alone is insufficient to distinguish two versions of a method.

Here is a simple example that illustrates method overloading:

```
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);
System.out.println("Result of ob.test:" + (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);
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:15190.5625
```

As you can see, **test( )** is overloaded four times. The first version takes no parameters, the second takes one integer parameter, the third takes two integer parameters, and the fourth takes one **double** parameter.

- ## Overloading Constructors

Like method overloading, we can also overload constructors. Example of overloaded constructor is described below:

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

// 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;
}

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

// 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;

// get volume of first box
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
```

The proper overloaded constructor is called based upon the parameters specified when **new** is executed.

- ## 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. That is, **this** is always a reference to the object on which the method was invoked.

Instance Variable Hiding

When a local variable has the same name as an instance variable, the local variable *hides* the instance variable. Because **this** lets you refer directly to the object, you can use it to resolve any name space collisions that might occur between instance variables and local variables.

For example, here is a version of **Box( )**, which uses **width**, **height**, and **depth** for parameter names and then uses **this** to access the instance variables by the same name:

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

// Use this to resolve name-space collisions.
Box(double width, double height, double depth) {
this.width = width;
this.height = height;
this.depth = depth;
}
}
```

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. When **this( )** is executed, the overloaded constructor that matches the parameter list specified by *argumentst* is executed first. Then, if there are any statements inside the original constructor, they are executed. The call to **this( )** must be the first statement within the constructor.

To understand how **this( )** can be used, let’s work through a short example.

```
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)
}
}
```

In this version of **MyClass**, the only constructor that actually assigns values to the **a** and **b** fields is **MyClass(int, int)**. The other two constructors simply invoke that constructor (either directly or indirectly) through **this( )**.

For example, consider what happens when this statement executes:
MyClass mc = new MyClass(8);

The call to **MyClass(8)** causes **this(8, 8)** to be executed, which translates into a call to **MyClass(8, 8)**, because this is the version of the **MyClass** constructor whose parameter list matches the arguments passed via **this( )**.

Now, consider the following statement, which uses the default constructor:
MyClass mc2 = new MyClass();

In this case, **this(0)** is called. This causes **MyClass(0)** to be invoked because it is the constructor with the matching parameter list. Of course, **MyClass(0)** then calls **MyClass(0, 0)** as just described.
One reason why invoking overloaded constructors through **this( )** can be useful is that it can prevent the unnecessary duplication of code.

• Inheritance

Inheritance is the process by which one object acquires the properties of another object. It supports the concept of hierarchical classification. 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.* y use of inheritance, an object need only define those qualities that make it unique within its class. It can inherit its general attributes from its parent.
Using inheritance, you can create a general class that defines traits common to a set of related items. This class can then be inherited by other, more specific classes, each adding those things

that are unique to it. In Java, a class that is inherited is called a *superclass*. The class that does the inheriting is called a *subclass*. Therefore, a subclass is a specialized version of a superclass. It inherits all of the instance variables and methods defined by the superclass and adds its own, unique elements.

Inheritance may take different forms:

- Single Inheritance (only one super class)
- Multilevel Inheritance (Derived from a derived class)
- Hierarchical Inheritance(one super class, many sub-classes)
- Multiple Inheritance(several super classes)

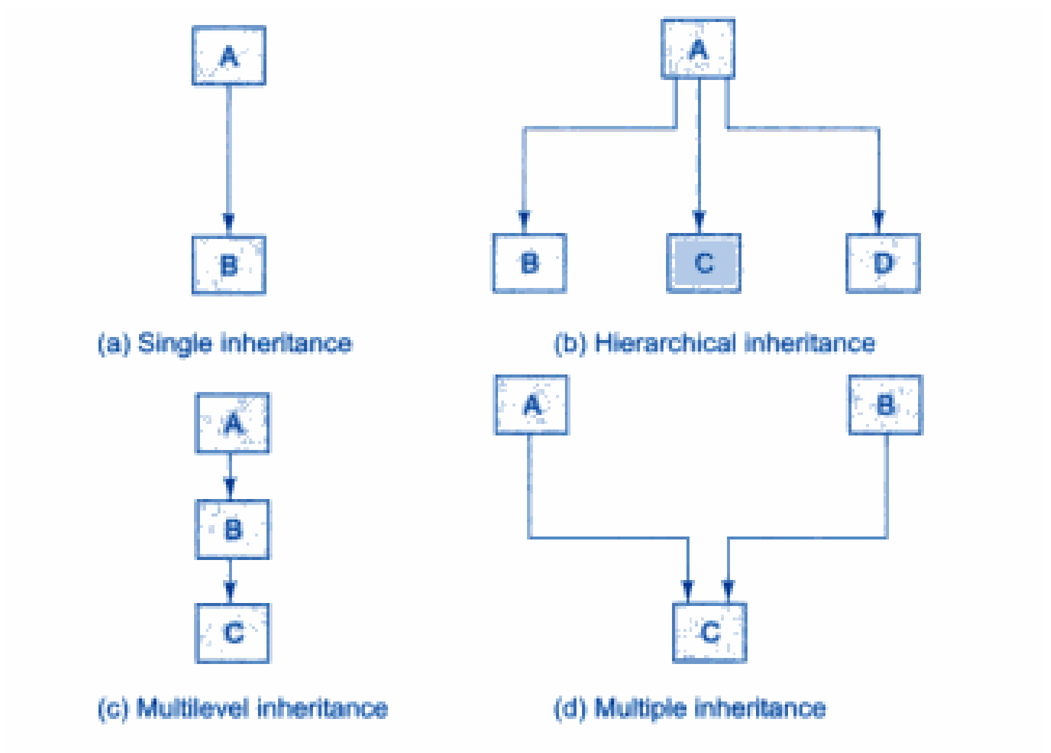


fig: Forms of Inheritance

Single Inheritance
To inherit a class, you simply incorporate the definition of one class into another by using the **extends** keyword.
Following example describe single inheritance:

```
class Animal{
void eat()
{System.out.println("eating...");
}
}

class Dog extends Animal
```

```
{
void bark()
{
System.out.println("barking...");}
}

class TestInheritance{

public static void main(String args[]){
Dog d=new Dog();
d.bark();
d.eat();
}}
```

Member Access and Inheritance

Although a subclass includes all of the members of its superclass, it cannot access those members of the superclass that have been declared as **private**. For example, consider the following simple class hierarchy:

```
// Create a superclass.
class A {
int i;                  //  default
private int j;          // private to A

void setij(int x, int y) {
i = x;
j = y;
}
}

// A's j is not accessible here.
class B extends A {
int total;

void sum() {
total = i + j;          // ERROR, j is not accessible here
}
}
class Access {
public static void main(String args[]) {
B subOb = new B();
subOb.setij(10, 12);
subOb.sum();
System.out.println("Total is " + subOb.total);
}
}
```

This program will not compile because the reference to **j** inside the **sum( )** method of **B** causes an access violation. Since **j** is declared as **private**, it is only accessible by other members of its own class. Subclasses have no access to it.

A Superclass Variable Can Reference a Subclass Object
Areference variable of a superclass can be assigned a reference to any subclass derived from that superclass. You will find this aspect of inheritance quite useful in a variety of situations.

For example, consider the following:

```
 class Animal{
 void eat()
 {System.out.println("eating...");
 }
 }

 class Dog extends Animal
 {
 void bark()
 {
 System.out.println("barking...");}
 }

 class TestInheritance{

 public static void main(String args[]){
 Animal a = new Animal();
 a.eat();

 Dog d=new Dog();
 d.bark();
 d.eat();

 Animal ad = new Dog();
 ad.eat();
}}
```

Here, **a** is a reference to **Animal** objects, and **b** is a reference to **Dog** objects.
Since **Dog** is a subclass of **Animal**, it is permissible to assign **Dog** object to **ad,** a reference variable of Animal class. It is important to understand that it is the type of the reference variable that determines what members can be accessed. That is, when a reference to a subclass object is assigned to a superclass reference variable, you will have access only to those parts of the object defined by the superclass.

Multilevel Inheritance

Simple class hierarchies consist of only a superclass and a subclass. However, you can build hierarchies that contain as many layers of inheritance as you like. As mentioned, it is perfectly acceptable to use a subclass as a superclass of another.
For example, given three classes called **A**, **B**, and **C**, **C** can be a subclass of **B**, which is a subclass of **A**. When this type of situation occurs, each subclass inherits all of the traits found in all of its superclasses. In this case, **C** inherits all aspects of **B** and **A.** Let us describe an example below:

```
class Animal{
void eat(){System.out.println("eating...");}
}
class Dog extends Animal{
void bark(){System.out.println("barking...");}
}
class BabyDog extends Dog{
void weep(){System.out.println("weeping...");}
}
class TestInheritance2{
public static void main(String args[]){
BabyDog d=new BabyDog();
d.weep();
d.bark();
d.eat();
}}
```

When Constructors Are Called

When a class hierarchy is created, in what order are the constructors for the classes that make up the hierarchy called? The answer is that in a class hierarchy, constructors are called in order of derivation, from superclass to subclass. Further, since **super( )** must be the first statement executed in a subclass' constructor, this order is the same whether or not **super( )** is used. If **super( )** is not used, then the default or parameterless constructor of each superclass will be executed.

Example:

```
// Create a super class.
class A {
A() {
System.out.println("Inside A's constructor.");
}
}

// Create a subclass by extending class A.
class B extends A {
B() {
System.out.println("Inside B's constructor.");
}
}

// Create another subclass by extending B.
class C extends B {
C() {
System.out.println("Inside C's constructor.");
}
}

class CallingCons {
public static void main(String args[]) {
C c = new C();
}
}
```

The output from this program is shown here:

```
Inside A’s constructor
Inside B’s constructor
Inside C’s constructor
```

As you can see, the constructors are called in order of derivation.

Hierarchical Inheritance

Many programming problems can be cast into hierarchy where certain features of one level are shared by many other below the level. This is known as Hierarchical Inheritance.

Example:

```
class Animal{
void eat(){System.out.println("eating...");}
}
class Dog extends Animal{
void bark(){System.out.println("barking...");}
}
class Cat extends Animal{
void meow(){System.out.println("meowing...");}
}
class TestInheritance3{
public static void main(String args[]){
Cat c=new Cat();
c.meow();
c.eat();

Dog d=new Dog();
d.bark();
d.eat();
}}
```

• Method Overriding

In a class hierarchy, when a method in a subclass has the same name and type signature as a method in its superclass, then the method in the subclass is said to *override* the method in the superclass. When an overridden method is called from within a subclass, it will always refer to the version of that method defined by the subclass. The version of the method defined by the superclass will be hidden.

Consider the following:

```
// Method overriding.
class A {
int i, j;
A(int a, int b) {
i = a;
j = b;
}
// display i and j
void show() {
System.out.println("i and j: " + i + " " + j);
}
}
class B extends A {
int k;
B(int a, int b, int c) {
super(a, b);
k = c;
}
// display k – this overrides show() in A
void show() {
System.out.println("k: " + k);
}
}
class Override {
public static void main(String args[]) {
B subOb = new B(1, 2, 3);
subOb.show(); // this calls show() in B
}
}
```

The output produced by this program is shown here:

```
k: 3
```

When **show( )** is invoked on an object of type **B**, the version of **show( )** defined within **B** is used. That is, the version of **show( )** inside **B** overrides the version declared in **A**.

Overridden methods allow Java to support run-time polymorphism. Polymorphism is essential to object-oriented programming for one reason: it allows a general class to specify methods that will be common to all of its derivatives, while allowing subclasses to define the specific implementation of some or all of those methods.

- ## The super keyword

Whenever a subclass needs to refer to its immediate superclass, it can do so by use of the keyword **super**.
super has two general forms. The first calls the superclass' constructor. The second is used to access a member of the superclass that has been hidden by a member of a subclass.
Each use is examined here.

Using super to Call Superclass Constructors
A subclass can call a constructor defined by its superclass by use of the following form of **super**:
Let's review the key concepts behind **super( )**. When a subclass calls **super( )**, it is calling the constructor of its immediate superclass. Thus, **super( )** always refers to the superclass immediately above the calling class. This is true even in a multileveled hierarchy.
Also, **super( )** must always be the first statement executed inside a subclass constructor.

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

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

class BoxWeight extends Box
{
double weight;                  // weight of box

// constructor when all parameters are specified
BoxWeight(double w, double h, double d, double m)
{
super(w, h, d);                 // call superclass constructor
weight = m;
}

// default constructor
BoxWeight() {
super();
weight = -1;
}
}

class DemoSuper {
public static void main(String args[])
{
BoxWeight mybox1 = new BoxWeight(10, 20, 15, 34.3);
BoxWeight mybox2 = new BoxWeight(); // default
double vol;

vol = mybox1.volume();
System.out.println("Volume of mybox1 is " + vol);
System.out.println("Weight of mybox1 is " + mybox1.weight);
System.out.println();

vol = mybox2.volume();
System.out.println("Volume of mybox2 is " + vol);
System.out.println("Weight of mybox2 is " + mybox2.weight);
System.out.println();
}
}
```

A Second Use for super
The second form of **super** acts somewhat like **this**, except that it always refers to the superclass of the subclass in which it is used. This usage has the following general form:

super.*member*

Here, *member* can be either a method or an instance variable.
This second form of **super** is most applicable to situations in which member names of a subclass hide members by the same name in the superclass.

```
// Using super to overcome name hiding.
class A {
int i;
}
// Create a subclass by extending class A.
class B extends A {
int i; // this i hides the i in A
B(int a, int b) {
super.i = a; // i in A
i = b; // i in B
}
void show() {
System.out.println("i in superclass: " + super.i);
System.out.println("i in subclass: " + i);
}
}
class UseSuper {
public static void main(String args[]) {
B subOb = new B(1, 2);
subOb.show();
}
}
```

This program displays the following:

```
i in superclass: 1
i in subclass: 2
```

Although the instance variable **i** in **B** hides the **i** in **A**, **super** allows access to the **i** defined in the superclass.

super can also be used to call methods that are hidden by a subclass. If you wish to access the superclass version of an overridden method, you can do so by using **super.**

Example:

```
class A {
int i, j;
A(int a, int b) {
i = a;
j = b;
}
class B extends A {
int k;
B(int a, int b, int c) {
super(a, b);
k = c;
}
void show() {
super.show(); // this calls A's show()
System.out.println("k: " + k);
}
}
```

```
i and j: 1 2
k: 3
```

Here, **super.show( )** calls the superclass version of **show( )**.

- ## The final keyword

The keyword **final** has three uses.

- It can be used to create the equivalent of a named constant.
- Using final to Prevent Overriding.
- Using final to Prevent Inheritance.

final variables

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.

```
final int FILE_NEW = 1;
final int FILE_OPEN = 2;
```

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.

Using final to Prevent Overriding

To disallow a method from being overridden, specify **final** as a modifier at the start of its declaration. Methods declared as **final** cannot be overridden.

The following fragment illustrates **final**:

```
class A {
final void meth() {
System.out.println("This is a final method.");
}
}
class B extends A {
void meth() { // ERROR! Can't override.
System.out.println("Illegal!");
}
}
```

Because **meth( )** is declared as **final**, it cannot be overridden in **B**. If you attempt to do so, a compile-time error will result.

Using final to Prevent Inheritance

Sometimes you will want to prevent a class from being inherited. To do this, precede the class declaration with **final**. Declaring a class as **final** implicitly declares all of its methods as **final**, too.

Here is an example of a **final** class:

```
final class A {
// ...
}
// The following class is illegal.
class B extends A { // ERROR! Can't subclass A
// ...
}
```
As the comments imply, it is illegal for **B** to inherit **A** since **A** is declared as **final**.

• Abstract Classes

Sometimes you will want to create a superclass that only defines a generalized form that will be shared by all of its subclasses, leaving it to each subclass to fill in the details. Such a class determines the nature of the methods that the subclasses must implement. To accomplish this it is required that certain methods be overridden by subclasses by specifying the **abstract** type modifier. These methods have no implementation specified in the superclass. Thus, a subclass must override them.

Any class that contains one or more abstract methods must also be declared abstract. To declare a class abstract, you simply use the **abstract** keyword in front of the **class** keyword at the beginning of the class declaration.

There can be no objects of an abstract class. That is, an abstract class cannot be directly instantiated with the **new** operator. Such objects would be useless, because an abstract class is not fully defined.

Any subclass of an abstract class must either implement all of the abstract methods in the superclass, or be itself declared **abstract**.

```
abstract class A {
abstract void callme();
// concrete methods are still allowed in abstract classes
void callmetoo() {
System.out.println("This is a concrete method.");
}
}
class B extends A {
void callme() {
System.out.println("B's implementation of callme.");
}
}
class AbstractDemo {
public static void main(String args[]) {
B b = new B();
b.callme();
b.callmetoo();
}
}
```
Notice that no objects of class **A** are declared in the program. As mentioned, it is not possible to instantiate an abstract class.
One other point: class **A** implements a concrete method called **callmetoo( )**. This is perfectly acceptable.

Although abstract classes cannot be used to instantiate objects, they can be used to create object references, because Java’s approach to run-time polymorphism is implemented through the use of superclass references. Thus, it must be possible to create a reference to an abstract class so that it can be used to point to a subclass object.