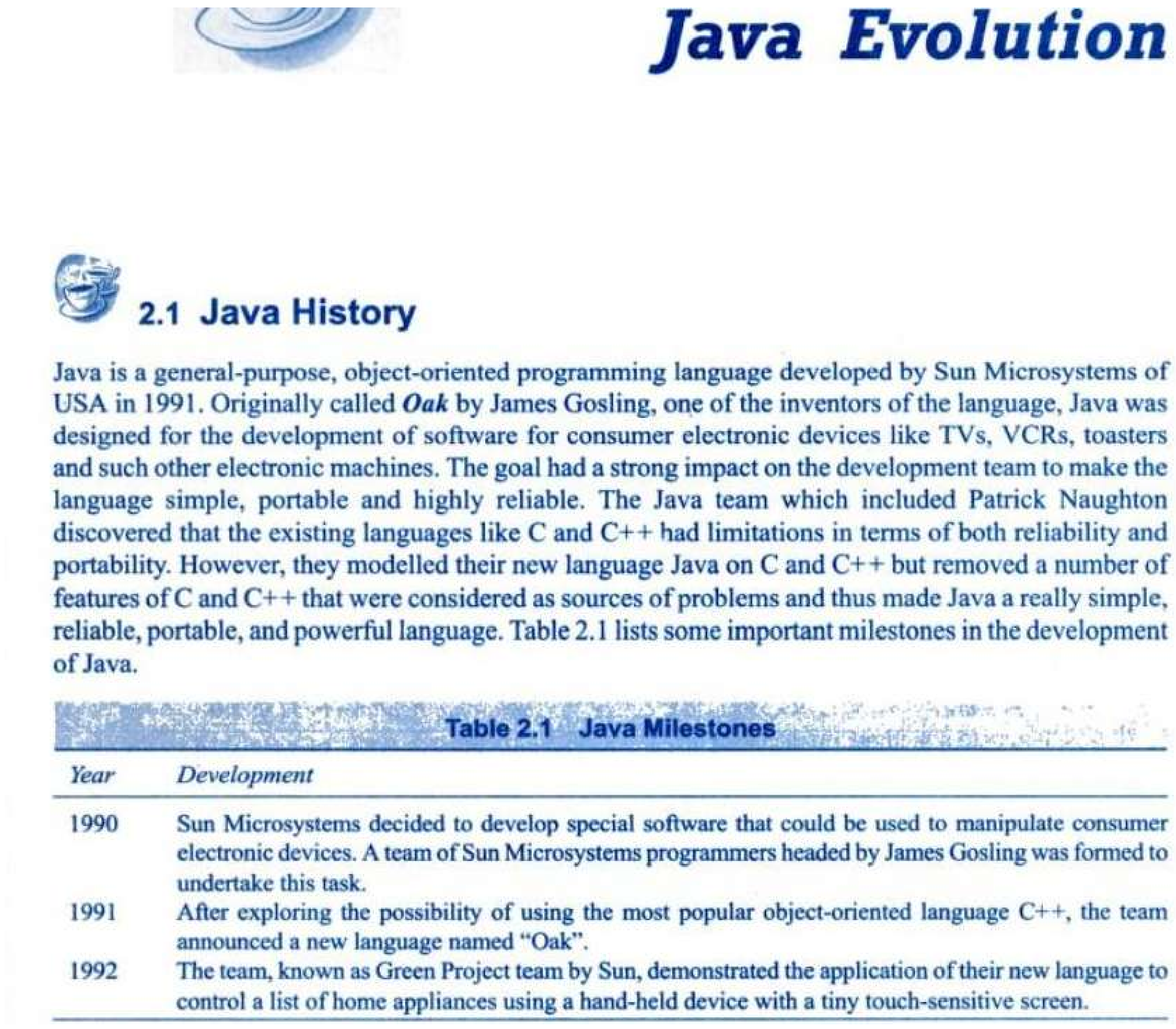


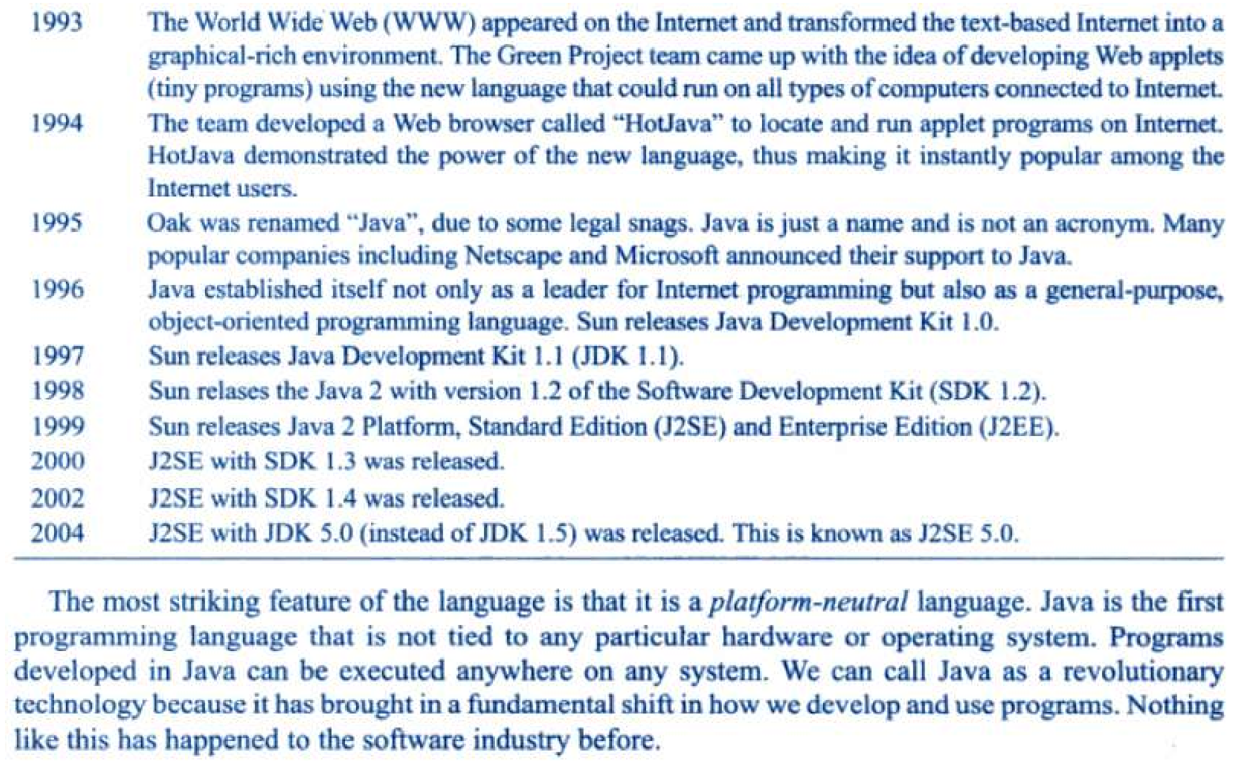
Java Evolution

2.1 Java History

Java is a general-purpose, object-oriented programming language developed by Sun Microsystems of USA in 1991. Originally called ***Oak*** by James Gosling, one of the inventors of the language, Java was designed for the development of software for consumer electronic devices like TVs, VCRs, toasters and such other electronic machines. The goal had a strong impact on the development team to make the language simple, portable and highly reliable. The Java team which included Patrick Naughton discovered that the existing languages like C and C++ had limitations in terms of both reliability and portability. However, they modelled their new language Java on C and C++ but removed a number of features of C and C++ that were considered as sources of problems and thus made Java a really simple, reliable, portable, and powerful language. Table 2.1 lists some important milestones in the development of Java.

Table 2.1 Java Milestones

Year	*Development*
1990	Sun Microsystems decided to develop special software that could be used to manipulate consumer electronic devices. A team of Sun Microsystems programmers headed by James Gosling was formed to undertake this task.
1991	After exploring the possibility of using the most popular object-oriented language C++, the team announced a new language named "Oak".
1992	The team, known as Green Project team by Sun, demonstrated the application of their new language to control a list of home appliances using a hand-held device with a tiny touch-sensitive screen.



1993	The World Wide Web (WWW) appeared on the Internet and transformed the text-based Internet into a graphical-rich environment. The Green Project team came up with the idea of developing Web applets (tiny programs) using the new language that could run on all types of computers connected to Internet.
1994	The team developed a Web browser called "HotJava" to locate and run applet programs on Internet. HotJava demonstrated the power of the new language, thus making it instantly popular among the Internet users.
1995	Oak was renamed "Java", due to some legal snags. Java is just a name and is not an acronym. Many popular companies including Netscape and Microsoft announced their support to Java.
1996	Java established itself not only as a leader for Internet programming but also as a general-purpose, object-oriented programming language. Sun releases Java Development Kit 1.0.
1997	Sun releases Java Development Kit 1.1 (JDK 1.1).
1998	Sun relases the Java 2 with version 1.2 of the Software Development Kit (SDK 1.2).
1999	Sun releases Java 2 Platform, Standard Edition (J2SE) and Enterprise Edition (J2EE).
2000	J2SE with SDK 1.3 was released.
2002	J2SE with SDK 1.4 was released.
2004	J2SE with JDK 5.0 (instead of JDK 1.5) was released. This is known as J2SE 5.0.

The most striking feature of the language is that it is a *platform-neutral* language. Java is the first programming language that is not tied to any particular hardware or operating system. Programs developed in Java can be executed anywhere on any system. We can call Java as a revolutionary technology because it has brought in a fundamental shift in how we develop and use programs. Nothing like this has happened to the software industry before.

An Overview of JAVA

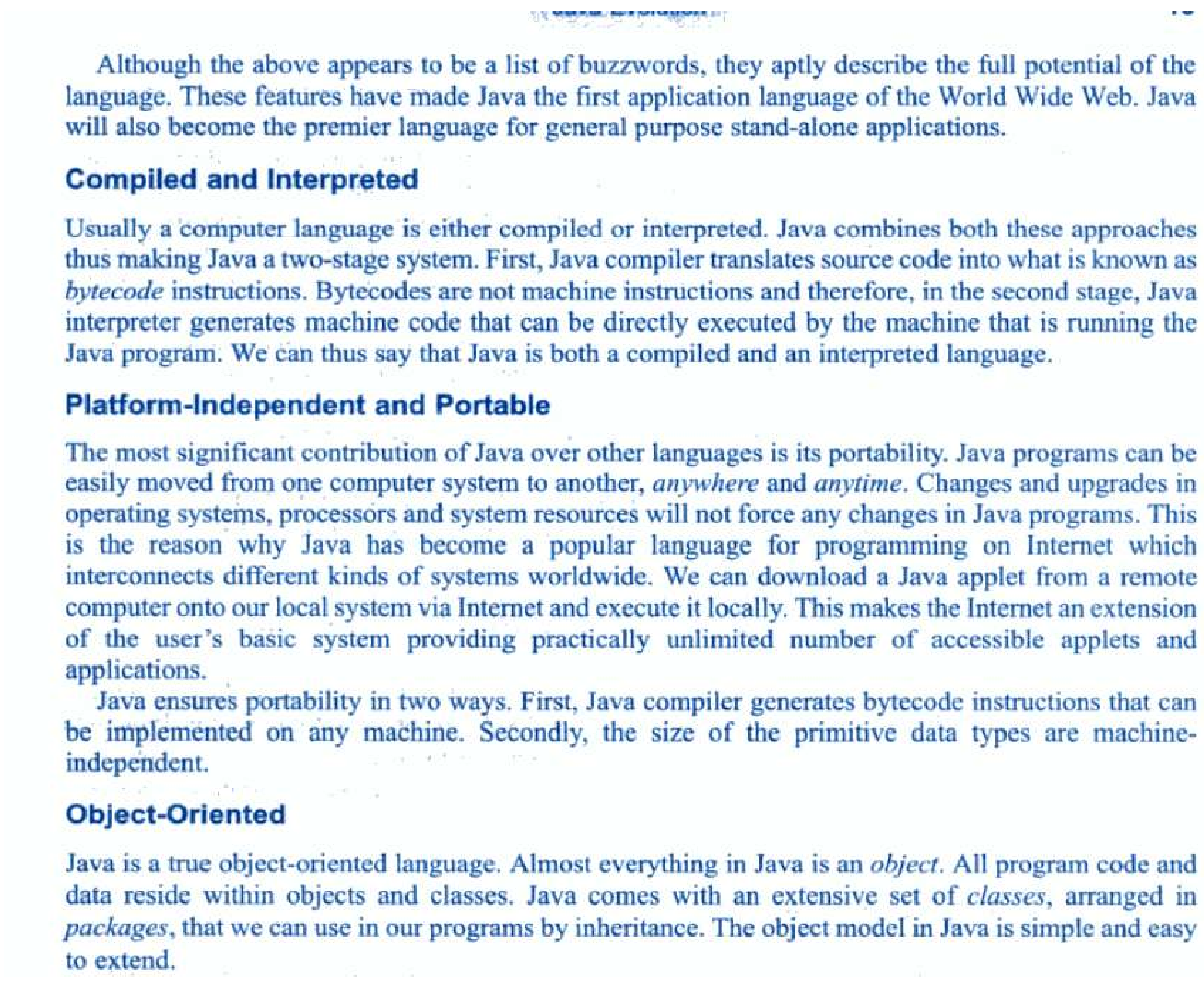
What is Java

- Java is an object-oriented programming language developed by Sun Microsystems that plays to the strengths of the Internet.
- Java program is created as a text file with the file extension .java. It is compiled into one or more files of bytecodes with the extension .class. Bytecodes are sets of instructions similar to the machine code instructions created when a computer program is compiled. The difference is that machine code must run on the computer system it was compiled for, and bytecodes can run on any computer system equipped to handle Java programs.
- Java is just a small, simple, safe, object-oriented, interpreted or dynamically optimized, byte-coded, architecture-neutral, garbage-collected, multithreaded programming language with a strongly typed exception-handling mechanism for writing distributed, dynamically extensible programs.

2.2 Java Features

The inventors of Java wanted to design a language which could offer solutions to some of the problems encountered in modern programming. They wanted the language to be not only reliable, portable and distributed but also simple, compact and interactive. Sun Microsystems officially describes Java with the following attributes:

Java 2 Features	*Additional Features of J2SE 5.0*
• Compiled and Interpreted	• Ease of Development
• Platform-Independent and Portable	• Scalability and Performance
• Object-Oriented	• Monitoring and Manageability
• Robust and Secure	• Desktop Client
• Distributed	• Core XML Support
• Familiar, Simple and Small	• Supplementary character support
• Multithreaded and Interactive	• JDBC RowSet
• High Performance	
• Dynamic and Extensible	



Although the above appears to be a list of buzzwords, they aptly describe the full potential of the language. These features have made Java the first application language of the World Wide Web. Java will also become the premier language for general purpose stand-alone applications.

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.

Ease of Development

Java 2 Standard Edition (J2SE) 5.0 supports features, such as Generics, Enhanced for Loop, Autoboxing or unboxing, Typesafe Enums, Varargs, Static import and Annotation. These features reduce the work of the programmer by shifting the responsibility of creating the reusable code to the compiler. The resulting source code is free from bugs because the errors made by the compiler are less

Monitoring and Manageability

Java supports a number of APIs, such as JVM Monitoring and Management API, Sun Management Platform Extension, Logging, Monitoring and Management Interface, and Java Management Extension (JMX) to monitor and manage Java applications. For example, Java provides JVM Monitoring and Management API to track the information at the application level and JVM level when deploying a large application. Java provides tools, such as jconsole, jps, jstat, and jstatd to make use of monitoring and management facilities. For example, GUI based tool called jconsole is used to monitor the JVM.

Desktop Client

J2SE 5.0 provides enhanced features to meet the requirements and challenges of the Java desktop users. It provides an improved Swing look and feel called Ocean. This feature is mainly used for developing graphics applications that require OpenGL hardware acceleration.

Miscellaneous Features

In addition to the above features, J2SE 5.0 supports the features such as:

Core XML Support J2SE 5.0 adds a powerful XML feature to the Java platform. Java contains some special packages for interface, to instantiate Simple API for XML (SAX) and Document Object Model (DOM) parsers to parse an XML document, transform the content of an XML document, and validate an XML document against the schema.

Supplementary Character Support Java adds the 32-bit supplementary character support as part of the Unicode 4.0 support. The supplementary characters are encoded with UTF-16 values to generate a different character called, *surrogate codepoint*.

JDBC RowSet Java supports JDBC RowSet to send data in a tabular format between the remote components of a distributed enterprise application. JDBC RowSet contains CachedRowSet and WebRowSet objects. The CachedRowSet object is a JavaBean component which acts like a container. This object contains a number of rows of data, which are retrieved from the database. The data stored in the CachedRowSet can be directly accessed without connecting to the database or any other data source. The rows of data that are retrieved from the database can be synchronized later. The WebRowSet object can operate without being connected to the database or data source. The WebRowSet object uses XML format to read and write the rowset.

Java Evolution

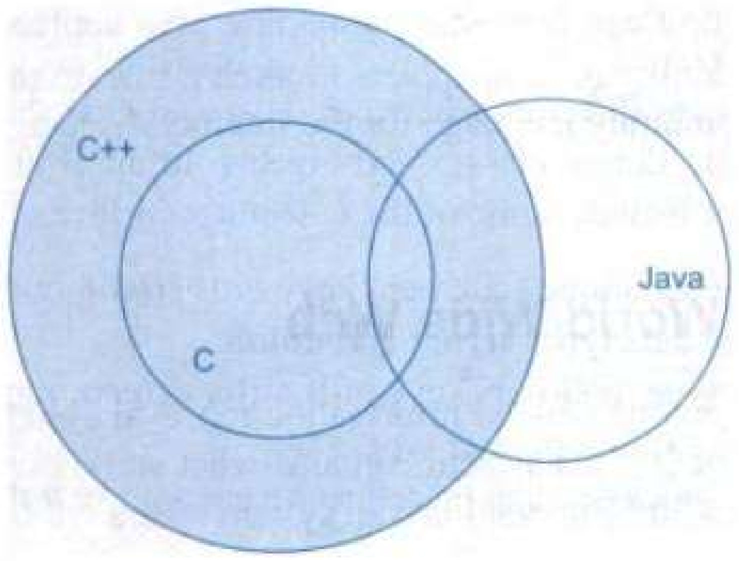


Fig. 2.1 *Overlapping of C, C++, and Java*

2.4 Java and Internet

Java is strongly associated with the Internet because of the fact that the first application program written in Java was HotJava, a Web browser to run applets on Internet. Internet users can use Java to create applet programs and run them locally using a "Java-enabled browser" such as HotJava. They can also use a Java-enabled browser to download an applet located on a computer anywhere in the Internet and run it on his local computer (see Fig. 2.2). In fact, Java applets have made the Internet a true extension of the storage system of the local computer.

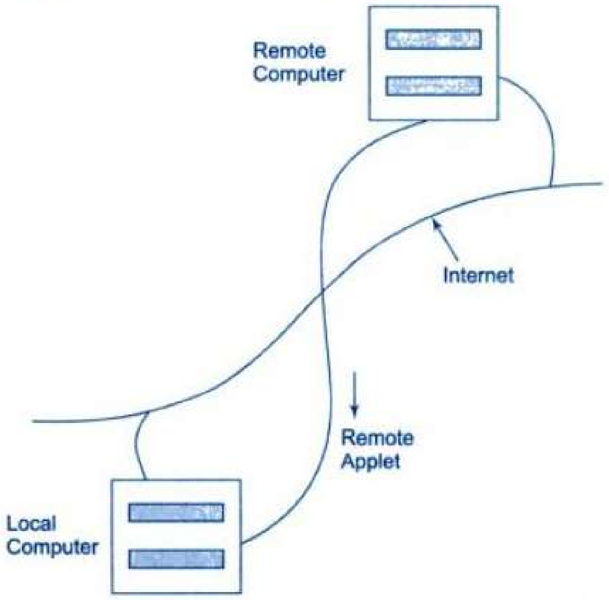


Fig. 2.2 *Downloading of applets via Internet*

Internet users can also set up their Web sites containing Java applets that could be used by o remote users of Internet. The ability of Java applets to hitch a ride on the Information Superhigh has made Java a unique programming language for the Internet. In fact, due to this, Java is popul known as *Internet language*.

2.5 Java and World Wide Web

World Wide Web (WWW) is an open-ended information retrieval system designed to be used in Internet's distributed environment. This system contains what are known as Web pages that pro both information and controls. Unlike a menu-driven system where we are guided through a partic direction using a decision tree structure, the Web system is open-ended and we can navigate to a document in any direction as shown in Fig. 2.3. This is made possible with the help of a langu called *Hypertext Markup Language* (HTML). Web pages contain HTML tags that enable us to f retrieve, manipulate and display documents worldwide.

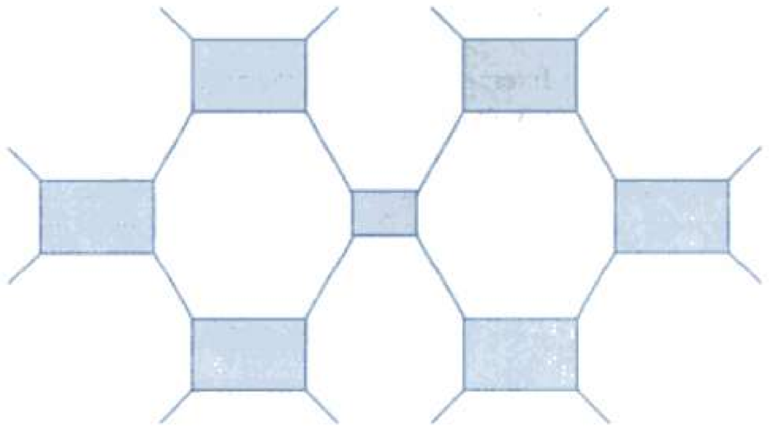


Fig. 2.3 *Web structure of information search*

Java was meant to be used in distributed environments such as Internet. Since, both the Web and Java share the same philosophy, Java could be easily incorporated into the Web system. Before Java, the World Wide Web was limited to the display of still images and texts. However, the incorporation of Java into Web pages has made it capable of supporting animation, graphics, games, and a wide range of special effects. With the support of Java, the Web has become more interactive and dynamic. On the other hand, with the support of Web, we can run a Java program on someone else's computer across the Internet.

Java communicates with a Web page through a special tag called <APPLET>. Figure 2.4 illustrates this process. The figure shows the following communication steps:

1. The user sends a request for an HTML document to the remote computer's Web server. The Web server is a program that accepts a request, processes the request, and sends the required document.
2. The HTML document is returned to the user's browser. The document contains the APPLET tag, which identifies the applet.

3. The corresponding applet bytecode is transferred to the user's computer. This bytecode had been previously created by the Java compiler using the Java source code file for that applet.
4. The Java-enabled browser on the user's computer interprets the bytecodes and provides output.
5. The user may have further interaction with the applet but with no further downloading from the provider's Web server. This is because the bytecode contains all the information necessary to interpret the applet.

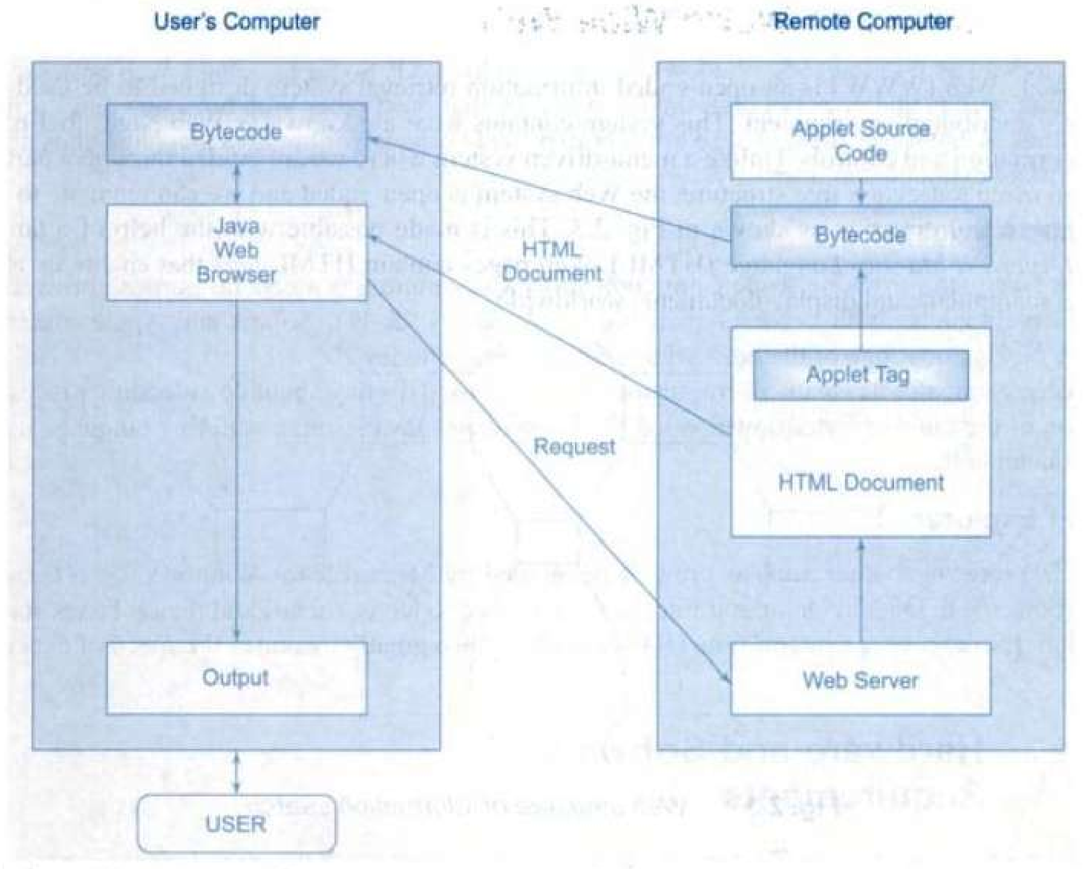


Fig. 2.4 *Java's interaction with the web*

2.6 Web Browsers

As pointed out earlier, the Internet is a vast sea of information represented in many formats and stored on many computers. A large portion of the Internet is organized as the World Wide Web which uses hypertext. Web browsers are used to navigate through the information found on the net. They allow us to retrieve the information spread across the Internet and display it using the hypertext markup language (HTML). Examples of Web browsers, among others, include:

- HotJava
- Netscape Navigator
- Internet Explorer

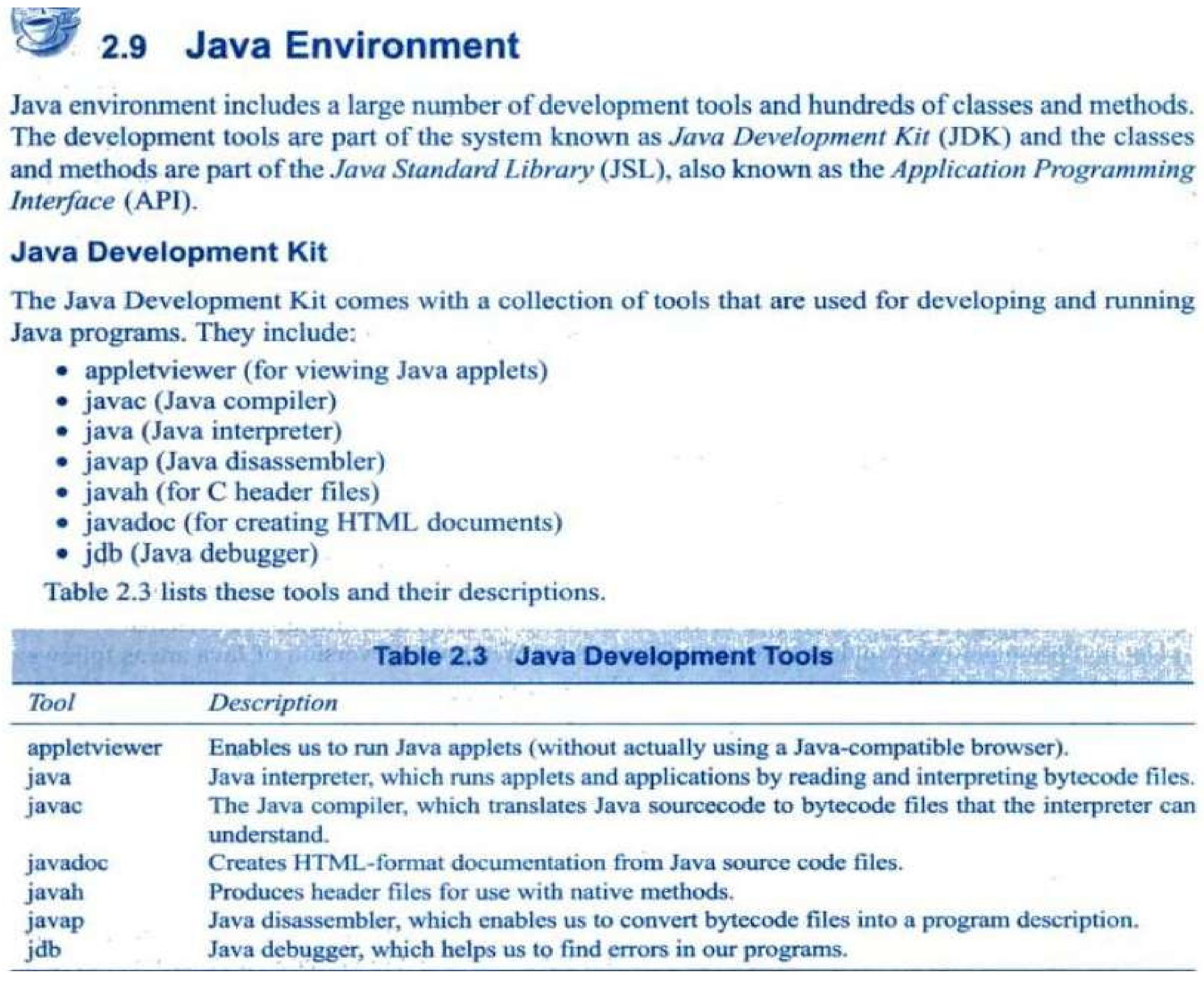
HTML documents and <APPLET> tags are discussed in detail in Chapter 14.

2.8 Java Support Systems

It is clear from the discussion we had up to now that the operation of Java and Java-enabled browsers on the Internet requires a variety of support systems. Table 2.2 lists the systems necessary to support Java for delivering information on the Internet.

Table 2.2 Java Support Systems

Support System	*Description*
Internet Connection	Local computer should be connected to the Internet.
Web Server	A program that accepts requests for information and sends the required documents.
Web Browser	A program that provides access to WWW and runs Java applets.
HTML	A language for creating hypertext for the Web.
APPLET Tag	For placing Java applets in HTML document.
Java Code	Java code is used for defining Java applets.
Bytecode	Compiled Java code that is referred to in the APPLET tag and transferred to the user computer.



2.9 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.

Table 2.3 Java Development Tools

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.

Overview of Java Language

3.1 Introduction

Java is a general-purpose, object-oriented programming language. We can develop two types of Java programs:

- Stand-alone applications
- Web applets

They are implemented as shown in Fig. 3.1. Stand-alone applications are programs written in Java to carry out certain tasks on a stand-alone local computer. In fact, Java can be used to develop programs for all kinds of applications, which earlier, were developed using languages like C and C++. As pointed out earlier, HotJava itself is a Java application program. Executing a stand-alone Java program involves two steps:

1. Compiling source code into bytecode using **javac** compiler
2. Executing the bytecode program using **java** interpreter.

Applets are small Java programs developed for Internet applications. An applet located on a distant computer (Server) can be downloaded via Internet and executed on a local computer (Client) using a Java-capable browser. We can develop applets for doing everything from simple animated graphics to complex games and utilities. Since applets are embedded in an HTML (Hypertext Markup Language) document and run inside a Web page, creating and running applets are more complex than creating an application.

Stand-alone programs can read and write files and perform certain operations that applets cannot do. An applet can only run within a Web browser.

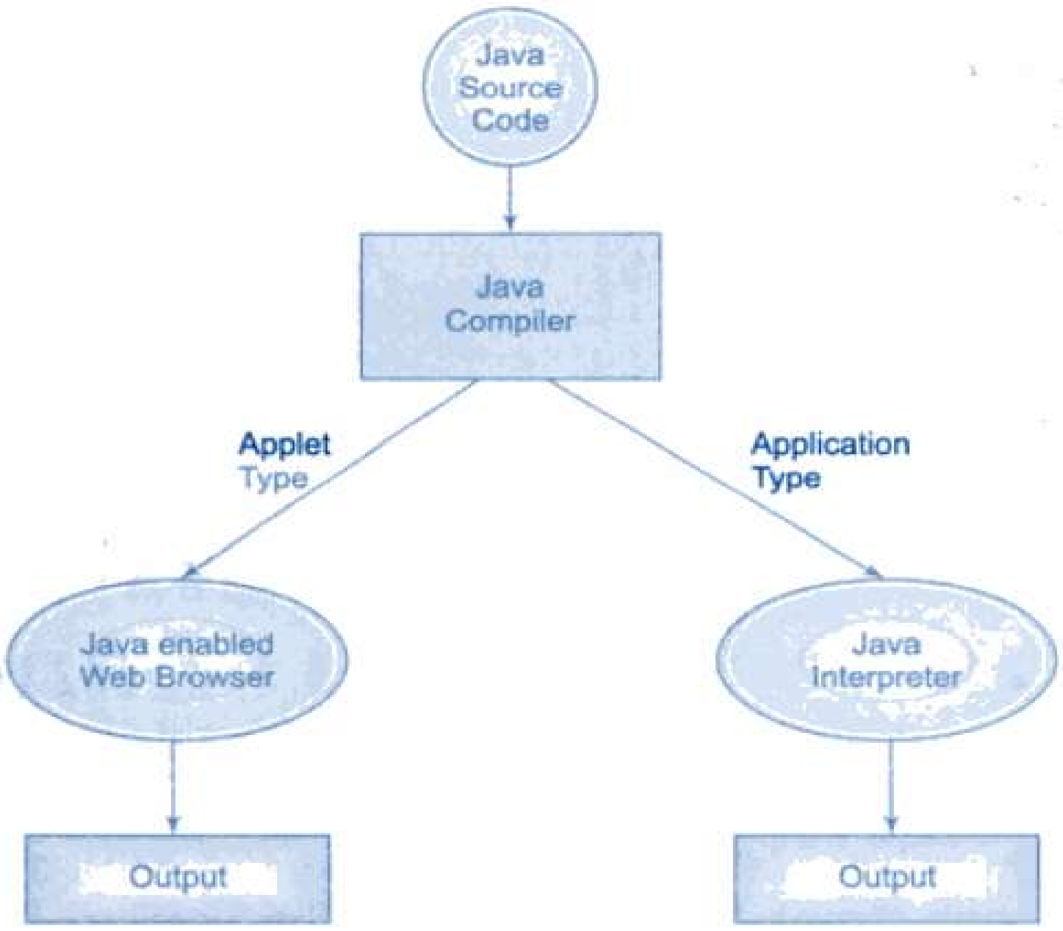


Fig. 3.1 *Two ways of using Java*

3.10 Command Line Arguments

There may be occasions when we may like our program to act in a particular way depending on the input provided at the time of execution. This is achieved in Java programs by using what are known as command line arguments. Command line arguments are parameters that are supplied to the application program at the time of invoking it for execution. It may be recalled that Program 3.4 was invokėd for execution of the command line as follows:

```
java Test
```

Here, we have not supplied any command line arguments. Even if we supply arguments, the program does not know what to do with them.

We can write Java programs that can receive and use the arguments provided in the command line. Recall the signature of the **main( )** method used in our earlier example programs:

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

As pointed out earlier, **args** is declared as an array of strings (known as string objects). Any arguments provided in the command line (at the time of execution) are passed to the array **args** as its elements. We can simply access the array elements and use them in the program as we wish. For example, consider the command line

```
java Test BASIC FORTRAN C++ Java
```

This command line contains four arguments. These are assigned to **the** array **args** as follows:

```
BASIC      ———→  args [0]
FORTRAN    ———→  args [1]
C++        ———→  args [2]
Java       ———→  args [3]
```

The individual elements of an array are accessed by using an index or subscript like **args[i]**. The value of i denotes the position of the elements inside the array. For example, **args[2]** denotes the third element and represents C++. Note that Java subscripts begin with 0 and not 1. (Arrays and strings are discussed in detail in Chapter 9.)

Program 3.5 ***Use of command line arguments***

```
/*
 * This program uses command line
 * arguments as input.
 */
Class ComLineTest
{
    public static void main (String args[ ])
    {
        int count, i=0;
        String string;
        count = args.length;
        System.out.println("Number of arguments = " + count);
        while (i < count)
        {
            string = args[i];
            i = i + 1;
            System.out.println(i+ " : " + "Java is" + string+ "!");
        }
    }
}
```

Program 3.5 illustrates the use of command line arguments. Compile and run the program with the command line as follows:

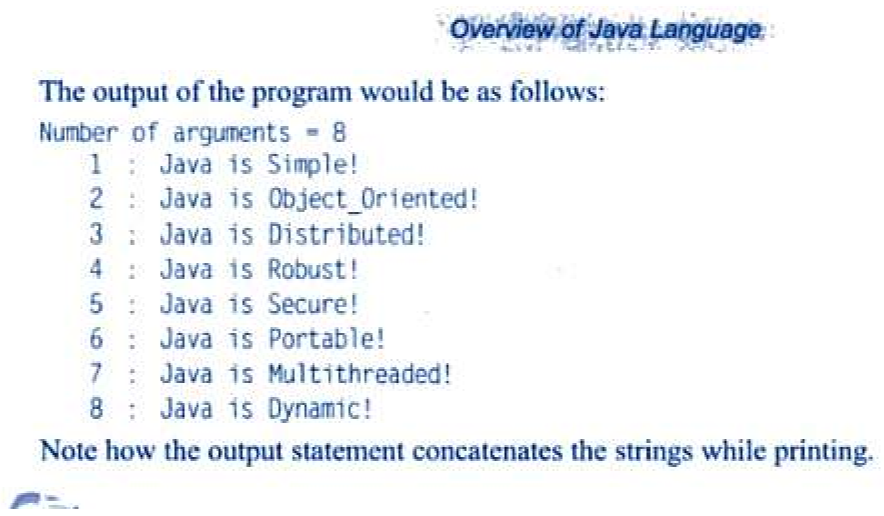
```
java ComLineTest Simple Object Oriented Distributed Robust
Secure Portable Multithreaded Dynamic
```

Upon execution, the command line arguments Simple, Object_Oriented, etc. are passed to the program through the array **args** as discussed earlier. That is the element **args[0]** contains Simple, **args[1]** contains Object_Oriented, and so on. These elements are accessed using the loop variable i as an index like

```
name = args[i]
```

The index **i** is incremented using a **while** loop until all the arguments are accessed. The number of arguments is obtained by statement

```
count = args.length;
```



Overview of Java Language

The output of the program would be as follows:

```
Number of arguments = 8
    1  :  Java is Simple!
    2  :  Java is Object_Oriented!
    3  :  Java is Distributed!
    4  :  Java is Robust!
    5  :  Java is Secure!
    6  :  Java is Portable!
    7  :  Java is Multithreaded!
    8  :  Java is Dynamic!
```

Note how the output statement concatenates the strings while printing.

FUNDAMENTAL OF JAVA

3) Creating a Java Application

```
class HelloWorld
{
 public static void main (String args[])
{
System.out.println("Hello World!");
 }
}
```

This program has some points to understand:

- The entire program is enclosed in a class definition. here, a class called HelloWorld.
- The body of the program is contained in a method called main(). In Java applications main() is the first method that is run when the program is executed.
- The main() method is defined as being public static with a void return type. public means that the method can be called from anywhere inside or outside the

class. static means that the method is the same for all instances of the class. The void return type means that main() does not return a value.

- The main() method is defined as taking a single parameter, String args[]. args is an array of String objects that represent command-line arguments passed to the class at execution.
- main() consists of a single statement that prints the message Hello, World! to the standard output stream, as follows:

System.out.println("Hello, World!");

The println() method is similar to the printf() method in C, except that it automatically appends a newline character (\n) at the end of the string. The out object is a member variable of the System object that represents the standard output stream. Finally, the System object is a global object in the Java environment that encapsulates system functionality.

- After compiling the program with the Java compiler (javac), you are ready to run it in the Java interpreter. The Java compiler places the executable output in a file called HelloWorld.class.

Statement in JAVA (Comment)

Unlike C& C++ java supports two types of comments. Java programmers (i.e., you) are familiar with the /* . . */ style of commenting. Anything lying within the /* . . */ pair is ignored by the compiler also supports the // notation for commenting.

Here, everything following // till the end of line is treated as comment. Usually */ style is used for commenting out a block of code, whereas, //is used for single-line comments.

Example of single-tine comment:

// This is one line comment

Example of multiple-line comment:

/* This is multiple line comment */

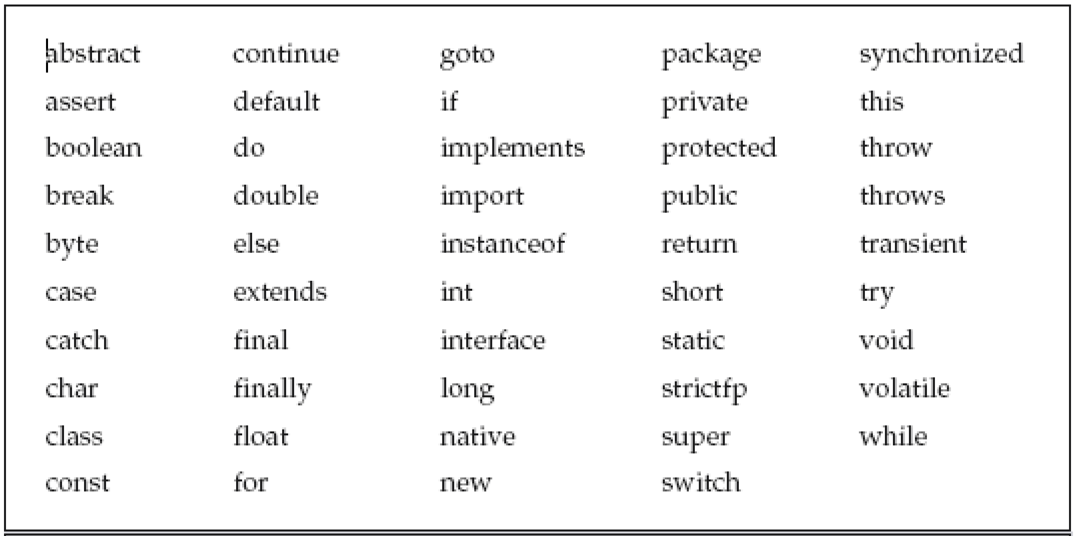
Java Tokens

1) Identifiers

Identifiers are tokens that represent names. These names can be assigned to variables ,methods, and classes to uniquely identify them for the compiler and give them meaningful names for the programmer.

2) Keywords

Keywords are predefined identifiers reserved by Java for a specific purpose



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

Table 2-1. *Java Reserved Keywords*

3) Literals

Program elements used in an invariant manner are called literals, or constants. Literals can be numbers, characters, or strings. Numeric literals include integers, floating-point numbers, and boolean.

<u>Variables & Declarations in java</u>

<u>Variables in java:</u>

I **variable** can be defined as "a quantity that varies during program **execution*.**

A variable is a symbol that represents a storage location in the computer's memory **The information that is stored** in that location is called the value of the variable. One **~~common~~** ***way for a variable*** **to obtain** a value is by an assignment. This has the syntax :

variable = **express**

First, the **expression is** evaluated and then the resulting value is assigned to *the variable. The equal* "=" is the assignment operator in java .

Declarations In java

A declaration associates a group of variables with a specific data-type. All variables **must** be declared before they can appear in executable statements.

A declaration consists of a data-type, followed by one or more variable names, **ending with a** semicolon.

Example :

Int a.b,c;

float rootl, root2;

Here and c are declared to be integer variables, rootl and root2 are floating-point **variables** and flag is a char-type variable.

Where Variables Are Declared

Variables will be declared in three basic places: inside functions, in the definition of function parameters, and outside of all functions.

1) Instance variables, 2) class variables, 3) local variables.

- **Instance variables:** - Variable defined within a class are called instance variable because each instance of the class contains its own copy of these instances.
- **Class variables: -** These are similar to instance variables, except their values apply to all that class's instances (and to the class itself) rather than having different values for each object.
- **Local variables: -** These are declared and used inside method definitions

Data Types in java

Data Types

Data types define the storage methods available for representing information, along with how the information is interpreted. Java data types can be divided into four categories:- Integer, floating-point, boolean, and Character

(1) int data-type

(2) char data-type

(3) float data-type

(4) double data-type

(5) bool (boolean) data-type

(6) enum (enumeration) data-type

1)Integer Data Types

Integer data types are used to represent signed integer numbers. There are four integer types: byte, short, int, and long. Each of these types takes up a different amount of space in memory

The width and ranges of these integer types vary widely, as shown in this table:

Name Width Range

Type Size

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

Let's look at each type of integer.

byte

The smallest integer type is **byte**. This is a signed 8-bit type that has a range from –128 to 127. Variables of type **byte** are especially useful when you're working with a stream of data from a network or file. 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 probably the least-used Java type, since it is defined as having its high byte first (called *big-endian* format). This type is mostly applicable to 16-bit computers, which are becoming increasingly scarce.
Here are some examples of **short** variable declarations:
short s;
short t;
*"Endianness" describes how multibyte data types, such as **short**, **int**, and **long**, are stored in memory. If it takes 2 bytes to represent a **short**, then which one comes first, the most significant or the least significant? To say that a machine is big-endian, means that the most significant byte is first, followed by the least significant one. Machines such as the SPARC and PowerPC are big-endian, while the Intel x86 series is little-endian.*

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. In addition to other uses, variables of type **int** are commonly employed to control loops and to index arrays. Any time you have an

integer expression involving **byte**s, **short**s, **int**s, and literal numbers, the entire expression is *promoted* to **int** before the calculation is done.
The **int** type is the most versatile and efficient type, and it should be used most of the time when you want to create a number for counting or indexing arrays or doing integer math. It may seem that using **short** or **byte** will save space, but there is no guarantee that Java won't promote those types to **int** internally anyway. Remember, type determines behavior, not size. (The only exception is arrays, where **byte** is guaranteed to use only one byte per array element, **short** will use two bytes, and **int** will use four.)

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;
// approximate speed of light in miles per second
lightspeed = 186000;
days = 1000; // specify number of days here
seconds = days * 24 * 60 * 60; // convert to seconds
distance = lightspeed * seconds; // compute distance
System.out.print("In " + days);
System.out.print(" days light will travel about ");
System.out.println(distance + " miles.");
}
}
```

This program generates the following output:

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

Clearly, the result could not have been held in an **int** variable.

2)Floating-Point Data 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. Java implements the standard (IEEE–754) set of floating-point types and operators. 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**

Each of these floating-point types is examined next.

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, but don't require a large degree of precision. For example, **float** can be useful when representing dollars
and cents.

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; // radius of circle
pi = 3.1416; // pi, approximately
a = pi * r * r; // compute area

System.out.println("Area of circle is " + a);
}
}
```

4)Character Data Type

The character data type (char) is used to store single Unicode characters. Because the Unicode character set is composed of 16-bit values, the char data type is stored as a 16- bit unsigned integer

Booleans

Java has a simple 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, such as **a < b**. **boolean** is also the type *required* by the conditional expressions that govern the
control statements such as **if** and **for**.
Here is a program that demonstrates the **boolean** type:

```
// Demonstrate boolean values.
class BoolTest {
public static void main(String args[]) {
boolean b;
b = false;
System.out.println("b is " + b);
b = true;
System.out.println("b is " + b);
// a boolean value can control the if statement
if(b) System.out.println("This is executed.");
b = false;
if(b) System.out.println("This is not executed.");
// outcome of a relational operator is a boolean value
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
```

There are three interesting things to notice about this program. First, as you can see, when a **boolean** value is output by **println( )**, “true” or “false” is displayed. Second, the value of a **boolean** variable is sufficient, by itself, to control the **if** statement. There is no need to write an **if** statement like this:

```
if(b == true) ...
```

Third, the outcome of a relational operator, such as **<**, is a **boolean** value. This is why the expression **10 > 9** displays the value “true.” Further, the extra set of parentheses around **10 > 9** is necessary because the + operator has a higher precedence than the >.

Literals in java

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. There are two other bases which can be used in integer literals, *octal* (base eight) and *hexadecimal* (base 16). Octal values are denoted in Java by a leading zero. Normal decimal numbers cannot have a leading zero. Thus, the seemingly valid value 09 will produce an error from the compiler, since 9 is outside of octal's 0 to 7 range. A more common base for numbers used by programmers is hexadecimal, which matches cleanly with modulo 8 word sizes, such as 8, 16, 32, and 64 bits. You signify a hexadecimal constant with a leading zero-x, (**0x** or **0X**). The range of a hexadecimal digit is 0 to 15, so *A* through *F* (or *a* through *f)* are substituted for 10 through 15.
Integer literals create an **int** value, which in Java is a 32-bit integer value. Since Java is strongly typed, you might be wondering how it is possible to assign an integer literal to one of Java's other integer types, such as **byte** or **long**, without causing a type mismatch error. Fortunately, such situations are easily handled. 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. Also, an integer literal can always be assigned to a **long** variable. However, to specify a **long** literal, you will need to explicitly tell the compiler that the literal value is of type **long**. You do this by appending an upper- or lowercase *L* to the literal. For example, 0x7fffffffffffffffL or 9223372036854775807L is the largest **long**.

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. *Scientific notation* uses a standard-notation, floating-point number plus a suffix that specifies a power of 10 by which the number is to be multiplied. The exponent is indicated by an *E* or *e* followed by a decimal number, which can be positive

or negative. 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.* Doing so is, of course, redundant. The default **double** type consumes 64 bits of storage, while the less-accurate **float** type requires only 32 bits.

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. In Java, they can only be assigned to variables declared as **boolean**, or used in expressions with Boolean operators.

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 *'@'.* For characters that are impossible to enter directly, there are several escape sequences, which allow you to enter the character you need, such as '\'' for the single-quote character itself, and **'\n'** for the newline character. There is also a mechanism for directly entering the value of a character in octal or hexadecimal. For octal notation use the backslash followed by the three-digit number. For example, *'\141'* is the letter *'a'.* For hexadecimal, you enter a backslash-u (**\u**), then exactly four hexadecimal digits. For example, *'\u0061'* is the ISO-Latin-1 *'a'* because the top byte is zero. *'\ua432'* is a Japanese Katakana character. Table shows the character escape sequences.

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

Table *Character Escape Sequences*

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\””

The escape sequences and octal/hexadecimal notations that were defined for character literals work the same way inside of string literals. One important thing to note about Java strings is that they must begin and end on the same line. There is no line-continuation escape sequence as there is in other languages.

As you may know, in some other languages, including C/C++, strings are implemented as arrays of characters. However, this is not the case in Java. Strings are actually object types. As you will see later in this book, because Java implements strings as objects, Java includes extensive string-handling capabilities that are both powerful and easy to use.

Operators:

Operators

An operator is a symbol that operates certain data type. The set of values, on which the operations are called as the operands. The operator thus operates on an operand. Operators could be classified as “unary”, “binary” or “ternary” depending on the number of operands i.e, one, two or three respectively.

Basic Operators in java :

Arithmetic operators.

Relational operators

Boolean Logical operators.

Increment and Decrement operators.

Ternary/Conditional Operator.

Bitwise operators.

Special Operator.

1) Arithmetic operators:

The five arithmetical operations :

Operator	Description
+	Addition
-	subtraction
*	multiplication
/	division

The binary arithmetic operators are +, -, *, / and the modulus operator %. Integer division truncates any fractional part. Modulus operator returns the remainder of the integer division.This operator is applicable only for integers and cannot be applied to float or double.

Example of the use of arithmetic operator :

The operators *, / and % all have the same priority, which is higher than the priority of binary addition (+) and subtraction (-). In case of an expression containing the operators having the same precedence it gets evaluated from left to right. This default precedence can be overridden by using a set of parentheses. If there is more than one set of parentheses, the innermost parentheses will be performed first, followed by the operations with-in the second innermost pair and so on. E.g.:

34 + 5 = 39

12 – 7 = 5

15 * 5 = 75

14 / 8 = 1

17 % 6 = 5

```
/* Program to demonstrate the addition and subtraction of double
numbers*/
class AddSubDoubles
{   public static void main (String args[])
   {
        double x = 7.5;
        double y = 5.4;
        System.out.println("x is : " + x);
        System.out.println("y is : " + y);
        double z = x + y;
        System.out.println("x + y is : " + z);
        z = x - y;
        System.out.println("x - y is : " + z);
}  }
```

/* Program to illustrate Associative law for Arithmetic Operators */

```
 # include <iostream.h>
 #include<conio.h>
Int main( )
{
        int a, b, c, d, e, f;
        clrscr();
        a = 5, b = 2 ;
        c = 3,  d = 7;
        e = 13;
        f = a * b - d % c + e ;
        cout<<"\n value  =  "<<f ;
        getch();
 }
      /************output*****************
      value=22
```

mixed mode Arithmetic Operators

when one Operators is real and other is integer,the expression is called a mixed mode Arithmetic expression.if either operator is real type and,then the other operand is converted to real and real arithmetic is performed.the result will be a real

e.g

25/10.0=2.5

25/10 =2

Unary Operator

Java includes a class of operator that act upon a single operand to produce a new value . such operator are known as Unary operators .

Unary operators usually precede their single operands, though some Unary operators are written after their operands.

Example :

-743 -0x7fff -0.2 -E-8

-root 1 -(x + y) -3*(x + y)

Increment and Decrement operators :

Thes operators also fall under the broad category of unary operators but are quite distinct than unary minus

The increment and decrement operators are very useful in c language . They are extensively used in for and while loops.

The ***SYNTAX*** of these operators is given below:

++<variable name >

<variable name>++

--<variable name>

<variable name>--

Operator	Operation
+ +	increment the value of the variable by 1
--	decrements the value variable by 1

The operator ++ adds 1 to the operand and - - subtracts 1 from the operand.

Increment operator:

Example:

++m and m++

Pre/Post Increment Decrement Operator :

Each of these operators has two version : a 'pre' and 'post' version.

'pre' version:

The 'pre' version performs the operation(either adding 1 or subtracting 1)

On the object before the resulting value in its surrounding context.

'post' version:

The 'post' version performs the operation after the objects current value has been used .

So consider a variable say i, then see following table.

Operator	**Representation**
Pre-increment	**+ + i**
Post increment	**i + +**
Pre decrement	**-- i**
Post decrement	**i - -**

E.g:

// Now p will have a value 11. (Postfixing)

x = ++p; -----------> Result : x = 12

// Now p will have a value 12. (Prefixing)

p = 11 -----------> p = 11

/*Program to illustrate Associativity of Increment/Decrement Operator */

```
 Class incdec
{
  Public static void main(String args[])
{ int x = 4, y = 9;
           int z;
           z = (x++) + (--y) + y ;
           System .out.println( "\n Value  =  "+z);
           z =(--x) + x + (y--) ;
system.out.print ( "\n Value  =  " +z);
        return 0;
}}
```

```
***************output*********
  Value=20
  Value=16
```

<u>Precedence Associativity of Increment/Decrement Operators:</u>

till now we have seen 3 types of operators , namely

*Arithmetic operators

*Assignment operator

*Increment /Decrement operators

The associativity of these operators is listed below

Associativity

Operator	Associativity

Icrement/Decrements	**Right to Left**
Arithmetic	**Left to right**
ASSIGNMENT	**Right to Left**

Precedence Increment/Decrement Operators:

Increment/Decrements	++ , - -
Arithmetic operators	*, / ,% + , -
ASSIGNMENT	=

Precedence and associativity of Unary operator

Operator	**operations**	**Associativity**
-	Unary minus	Right to Left
++ --	Increment Decrement	Right to Left
* / %	Multiplication Division Modulus	Left to Right
+	Addition	Left to Right

=	Assignment	Right to Left

Relational operators:

Relational operator are binary operators.

Operator	Description
>	**Greater than**
>=	**Greater than or equals to**
<	**Less than**

These operators all fall within the same precedence group, which is lower than the arithmetic and unary operators. The associativity of these operators is left to right.

Closely associated with the above-mentioned relational operators are the following two equality operators:

Operator	Description
= =	**equals to**
!=	**not equals to**

The equality operators fall precedence group beneath the relational operators. These operators also have a left to right associativity .

The value of the relational expression is of integer type and is 1, if the result of comparison is true and 0 if it is false.

E.g.:

14 > 8	**has the value 1, as it is true**
34 <= 19	**has the value 0, as it is false**

/* Program to illustrate Relational Operator */

```
class RelationalOp
{
            public static void main(String args[])
            {
                float x = 15.0F, y = 10.65F, z = 25.0F;
                System.out.println("x = "+x);
                System.out.println("y = "+y);
                System.out.println("z = "+z);
                System.out.println("x < y is : "+(x<y));
                System.out.println("x > y is : "+(x>y));
                System.out.println("x == z is : "+(x==z));
                System.out.println("x <= z is : "+(x<=z));
                System.out.println("x >= y is : "+(x>=y));
                System.out.println("y != z is : "+(y!=z));
                System.out.println("y == x+z is : "+(y==x+z));
            }
```

Logical operators

The logical operators && (AND), || (OR) allow two or more expressions to be combined to form a single expression. The expressions involving these operators are evaluated left to right, and evaluation stops as soon as the truth or the falsehood of the result is known.

Operator	Usage
&&	Logical AND. Returns 1 if both the expressions are non-zero.
||	Logical OR. Returns 1 if either of the expression is non-zero.
!	Unary negation. It converts a non-zero operand into 0 and a zero operand into 1.

Example of && operator :

consider the following expression :

a > b && x== 10

The expression on left is a > b and that on the right is x= =10. the above-stated whole expression evaluates to true(1) only if both .

example of || operator :

a < m || a < n

The expression is true if one of the expression(eithr a<m or a<n) is true. or both of them are true. that is, if the value of a is less thane that of m or n then whole expression evaluates to true. needless to say , it evaluates to true

in case a less than both m and n.

example of != operator :

the not operator takes single expression and evaluates to true(1) if the expression is false (0), and evaluates to false(0), and evaluates to false (0) if the expression is true(1). in other words it just reverses the value of the expression.

for example

!(x >= y)

Note: All the expressions, which are part of a compound expression, may not be evaluated, when they are connected by && or || operators.

Expr1	Expr2	Expr1 && Expr2	Expr1 || Expr2
0	0	0	0
0	non-zero	0	1
non-zero non-zero	0	0	1

Table : Operation of logical && and || operators

<u>Assignment operators</u>

there are several different assignment operators in c . all of them are used to form assignment expression ,which assign the value of an expression to an identifier

Most commonly assignment operator are = .

precedence of assignment operator (Right to left)

syntax:

variable = expression

example:

a =3

x=y

also java contains the following five additional assignment operator

(+ =, - =,* =, % =)

syntax:

expression1+= expression

is equivalent to:

expression1=expression1+expression2

simllarly, the assignment expression

expression 1 -= expression2

is equivalent to:

expression1=expression1-expression

Operator	Expression	Result
i + = 3	i = i + 3	i = 18
i - = 2	i = i – 2	i = 13
i * = 4	i = i * 4	i = 60
i / = 3	i = i / 3	i = 5
i % = 4	i = i % 4	i = 3

Table 2.6: Examples for Compound Assignment operators

Ternary/Conditional Operator :

The conditional expressions written with the ternary operator “ ? : ” provides an alternate way to write the if conditional construct. This operator takes three arguments.

The syntax is:

condition ? expression1 : expression2

(true) (false)

If condition is true (i.e. Value is non-zero), then the value returned would be expression1

otherwise condition is false(value is zero), then value returned would be expression2.

/* Program to demonstrate the ternary Operator */

```
class Ternary
{

          public static void main(String args[])
          {
            int value1 = 1;
                  int value2 = 2;
            int result;
                  boolean Condition = true;

            result = Condition ? value1 : value2;
```

```
                System.out.println("Result is : "+result);

        }
}
```

Bitwise operators

The bitwise operators provided by C may only be applied to operands of type char, short, int and long, whether signed or unsigned.

Operator	Meaning
&	*BITWISE AND*
|	*BITWISE OR*
^	*BITWISE XOR(EXCLUSIVE OR)*
~	*One's complement*
>>	*Right SHIFT*
<<	*LEFT SHIFT*
>>>	*SHIFT righit zero fill assiggnment*

1) BITWISE AND Operator :

TRUE TABLE OF BITWISE AND Operator

X	Y	OUTPUT(X&Y)
0	1	0
0	1	0
1	0	0
1	1	1

2) Bit-wise OR OPERATOR :

TRUE TABLE OF BITWISE OR Operator

X	Y	OUTPUT(X | Y)
0	1	0

0	1	1
1	0	1
1	1	1

3)Bitwise EXOR Operator :

TRUE TABLE OF BITWISE EX-OR Operator

X	Y	OUTPUT(X^Y)
0	1	0
0	1	0
1	0	0
1	1	1

4) *One's complement*

TRUE TABLE *One's complement* Operator

X	OUTPUT(~X)
0	1
1	0

5) Bitwise Right SHIFT OPERATOR :

6) Bitwise LEFT SHIFT OPERATOR :

/* Program to demonstrate Bitwise Operator Assignments */

```
class BitAssign
{
        public static void main(String args[])
        {
                int A=1,B=2,C=3;

                A |=4;
                B >>=1;
                C <<=1;
                A ^=C;
                System.out.println(" A = "+A);
                System.out.println(" B = "+B);
                System.out.println(" C = "+C);
        }
}
```

Output

```
A=3
B=1
C=6
```

/* Program to demonstrate Bitwise Shift Operators */

```
class BitShift
{
    public static void main(String args[])
    {
        int A = 6, B=-6;
        System.out.println("A : "+A);
        System.out.println("B : "+B);
        System.out.println("A >> 2 : "+(A>>2));
        System.out.println("A << 1 : "+(A<<1));
        System.out.println("A >>> 1 : "+(A>>>1));
        System.out.println("B >> 1 : "+(B>>1));
        System.out.println("B >>> 1 : "+(B>>>1));
    }
}
```

Output

```
A=6
B=-6
A>>2=1
```

/* Program to demonstrate the boolean logical operators */

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

/* Program to demonstrate Bitwise Logical operators */

```
class BitLogic
{
        public static void main(String args[])
        {
                int A=35,B=15,C=3;
                System.out.println("A is : "+A);
                System.out.println("B is : "+B);
                System.out.println("A | B  is : "+(A|B));
                System.out.println("A & B  is : "+(A&B));
                System.out.println("A ^ B  is : "+(A^B));
            int D = ~C & 0x0f;
                System.out.println(" ~ C is : "+D);

}}
```

TYPE CAST OPERATOR:

Primitive Type Casting ....

- ***Casting between primitive types enables you to convert the value of one data from one type to another primitive type.***
- ***Commonly occurs between numeric types.(integer , floating point)***
- ***There is one primitive data type that we cannot do casting though, and that is the boolean data type.***
- ***Types of Casting:***

- ***Implicit Casting***
- ***Explicit Casting***

Implicit Casting ....

C++ permits explicit type conversion of variable or expression using the Type cast operator

- Suppose we want to store a value of int data type to a variable of data type double.

  int numInt = 10;

  double numDouble = numInt; //implicit cast
- In this example, since the destination variable's data type (double) holds a larger value than the value's data type (int),

  the data is implicitly casted to the destination variable's data type double.
- Example:

  int numInt1 = 1;

  int numInt2 = 2;

  //result is implicitly casted to type double

  double numDouble = numInt1/num

●

Explicit Casting ....

- When we convert a data that has a large type to a smaller type, we must use an explicit cast.
- Explicit casts take the following form:

(Type)value

- where,
- **Type (target type)-** is the name of the type you're converting to
- **value** -is an expression that results in the value of the source type

- Example

```
double valDouble = 10.12;
int valInt = (int)valDouble;
//convert valDouble to int type
double x = 10.2;
int y = 2;
int result = (int)(x/y); //typecast result of operation to int

/* Program to demonstrate casting Incompatible Types */
class Casting
{
        public static void main(String args[])
        {
                byte b;
                int i = 258;
                double d = 340.142;

                System.out.println("\n Conversion of int to byte.....");
                b = (byte) i;
System.out.println(" integer i : "+i+" is converted to byte b : "+b);
System.out.println("\n Conversion of double to int.....");
                i = (int) d;
        System.out.println(" double d : "+d+" is converted to integer i : "+i);

                System.out.println("\n Conversion of double to byte.....");
                b = (byte) d;
        System.out.println(" double d : "+d+" is converted to byte b : "+b);
        }
}
```

Special operator

Type comparison operator **instanceof**

e.g

person instanceof employee;

Dot Operator

e.g

personob.work; //reference to the variablr work

personob.getdata// reference to the method getdata

Operator Precedence

Table shows the order of precedence for Java operators, from highest to lowest.

Notice that the first row shows items that you may not normally think of as operators: parentheses, square brackets, and the dot operator. Parentheses are used to alter the precedence of an operation. As you know from the previous chapter, the square brackets provide array indexing. The dot operator is used to dereference objects and will be discussed later in this book.

Operator	operations	Associativity
( ) [ ]	Member selection Function call Array element reference	Left to Right
- ++ -- ! ~ (type)	Unary minus Increment Decrement Logical negation Ones complement Casting	Right to Left
* / %	Multiplication Division Modulus	Left to Right
>> << >>>	*Right SHIFT* *LEFT SHIFT* *SHIFT righit zero fill assiggnment*	Left to Right

< <= > >= instanceof	**Less than** **Greater than or equals to** **Greater than** **Greater than or equals to** **Type comparison**	**Left to Right**
= !=	Equality in Equality	Left to Right
& ^ | && || ?: = Op=	BITWISE AND BITWISE XOR(EXCLUSIVE OR) BITWISE OR Logical AND Logical OR. Conditional Operator Assignment Shortcut assignment	Left to Right Left to Right Left to Right Left to Right Left to Right Left to Right Left to Right Right to Left Right to Left