UNIT I

The History and Evolution of Java:

Java's Lineage:

Java is related to C++, which is a direct descendant of C. Much of the character of Java is inherited from these two languages. From C, Java derives its syntax. Many of Java's object oriented features were influenced by C++. In fact, several of Java's defining characteristics come from—or are responses to—its predecessors. The creation of Java was deeply rooted in the process of refinement and adaptation that has been occurring in computer programming languages for the past several decades. For these reasons, this section reviews the sequence of events and forces that led to Java. Each innovation in language design was driven by the need to solve a fundamental problem that the preceding languages could not solve. Java is no exception.

The Creation of java:

Java was conceived by James Gosling, Patrick Naughton, Chris Warth, Ed Frank, and Mike Sheridan at Sun Microsystems, Inc. in 1991. It took 18 months to develop the first working version. This language was initially called "Oak," but was renamed "Java" in 1995. Between the initial implementation of Oak in the fall of 1992 and the public announcement of Java in the spring of 1995, many more people contributed to the design and evolution of the language. Bill Joy, Arthur van Hoff, Jonathan Payne, Frank Yellin, and Tim Lindholm were key contributors to the maturing of the original prototype.

The trouble with C and C++ (and most other languages) is that they are designed to be compiled for a specific target.

Although it is possible to compile a C++ program for just about any type of CPU, to do so requires a full C++ compiler targeted for that CPU. The problem is that compilers are expensive and time-consuming to create. An easier and more cost-efficient solution was needed. In an attempt to find such a solution, Gosling and others began work on a portable, platform- independent language that could be used to produce code that would run on a variety of CPUs under differing environments. This effort ultimately led to the creation of Java.

About the time that the details of Java were being worked out, a second, and ultimately more important, factor was emerging that would

play a crucial role in the future of Java. This second force was, of course, the World Wide Web.

Most programmers learn early in their careers that portable programs are as elusive as they are desirable. The Internet consists of a diverse, distributed universe populated with various types of computers, operating systems, and CPUs. Even though many kinds of platforms are attached to the Internet, users would like them all to be able to run the same program.

By 1993, it became obvious to members of the Java design team that the problems of portability frequently encountered when creating code for embedded controllers are also found when attempting to create code for the Internet. In fact, the same problem that Java was initially designed to solve on a small scale could also be applied to the Internet on a large scale. This realization caused the focus of Java to switch from consumer electronics to Internet programming. So, while the desire for an architecture-neutral programming language provided the initial spark, the Internet ultimately led to Java's large-scale success.

As mentioned earlier, Java derives much of its character from C and C++. This is by intent. The Java designers knew that using the familiar syntax of C and echoing the object-oriented features of C++ would make their language appealing to the legions of experienced C/C++ programmers. In addition to the surface similarities, Java shares some of the other attributes that helped make C and C++ successful. First, Java was designed, tested, and refined by real, working programmers. It is a language grounded in the needs and experiences of the people who devised it. Thus, Java is a programmer's language. Second, Java is cohesive and logically consistent. Third, except for those constraints imposed by the Internet environment, Java gives you, the programmer, full control. If you program well, your programs reflect it. If you program poorly, your programs reflect that, too. Put differently, Java is not a language with training wheels. It is a language for professional programmers.

Because of the similarities between Java and C++, it is tempting to think of Java as simply the "Internet version of C++." However, to do so would be a large mistake. Java has significant practical and philosophical differences. While it is true that Java was influenced by C++, it is not an enhanced version of C++. For example, Java is neither upwardly nor downwardly compatible with C++. Of course, the similarities with C++ are significant, and if you are a C++ programmer, then you will feel right at home with Java. One other point: Java was not designed to replace C++. Java was designed to solve a certain set of problems. C++ was

designed to solve a different set of problems. Both will coexist for many years to come.
Computer languages evolve for two reasons:
To adapt to changes in environment and to implement advances in the art of programming. The environmental change that prompted Java was the need for platform-independent programs destined for distribution on the Internet. Java enhanced and refined the object-oriented paradigm used by C++, added integrated support for multithreading, and provided a library that simplified Internet access. In the final analysis, though, it was not the individual features of Java that made it so remarkable. Java was the perfect response to the demands of the then newly emerging, highly distributed computing universe. Java was to Internet programming what C was to system programming: a revolutionary force that changed the world.

How Java Changed the Internet

The Internet helped catapult Java to the forefront of programming, and Java, in turn, had a profound effect on the Internet. In addition to simplifying web programming in general, Java innovated a new type of networked program called the applet that changed the way the online world thought about content. Java also addressed some of the thorniest issues associated with the Internet: portability and security. Let’s look more closely at each of these.

Java Applets

An *applet* is a special kind of Java program that is designed to be transmitted over the Internet and automatically executed by a Java-compatible web browser. An applet is downloaded on demand, without further interaction with the user. If the user clicks a link that contains an applet, the applet will be automatically downloaded and run in the browser. Applets are intended to be small programs. They are typically used to display data provided by the server, handle user input, or provide simple functions, such as a loan calculator, that execute locally, rather than on the server. In essence, the applet allows some functionality to be moved from the server to the client.
The creation of the applet changed Internet programming because it expanded the universe of objects that can move about freely in cyberspace. In general, there are two very broad categories of objects that are transmitted between the server and the client: passive

information and dynamic, active programs. For example, when you read your e-mail, you are viewing passive data. Even when you download a program, the program's code is still only passive data until you execute it. By contrast, the applet is a dynamic, self-executing program. Such a program is an active agent on the client computer, yet it is initiated by the server.
As desirable as dynamic, networked programs are, they also present serious problems in the areas of security and portability. Obviously, a program that downloads and executes automatically on the client computer must be prevented from doing harm. It must also be able to run in a variety of different environments and under different operating systems.

Security

Every time we download a "normal" program, you are taking a risk, because the code you are downloading might contain a virus, Trojan horse, or other harmful code. At the core of the problem is the fact that malicious code can cause its damage because it has gained unauthorized access to system resources. For example, a virus program might gather private information, such as credit card numbers, bank account balances, and passwords, by searching the contents of your computer's local file system. In order for Java to enable applets to be downloaded and executed on the client computer safely, it was necessary to prevent an applet from launching such an attack.
Java achieved this protection by confining an applet to the Java execution environment and not allowing it access to other parts of the computer. The ability to download applets with confidence that no harm will be done and that no security will be breached may have been the single most innovative aspect of Java.

Portability

Portability is a major aspect of the Internet because there are many different types of computers and operating systems connected to it. If a Java program were to be run on virtually any computer connected to the Internet, there needed to be some way to enable that program to execute on different systems. For example, in the case of an applet, the same applet must be able to be downloaded and executed by the wide variety of CPUs, operating systems, and browsers connected to the Internet. It is

not practical to have different versions of the applet for different computers. The *same* code must work on *all* computers. Therefore, some means of generating portable executable code was needed.

Java's Magic: The Bytecode

The key that allows Java to solve both the security and the portability. *Bytecode* is a highly optimized set of instructions designed to be executed by the Java run-time system, which is called the *Java Virtual Machine (JVM)*. In essence, the original JVM was designed as an *interpreter for bytecode*. Java program is executed by the JVM helps solve the major problems associated with web-based programs. Translating a Java program into bytecode makes it much easier to run a program in a wide variety of environments because only the JVM needs to be implemented for each platform. Once the run-time package exists for a given system, any Java program can run on it.

If a Java program were compiled to native code, then different versions of the same program would have to exist for each type of CPU connected to the Internet. This is, of course, not a feasible solution. Thus, the execution of bytecode by the JVM is the easiest way to create truly portable programs.

The fact that a Java program is executed by the JVM also helps to make it secure. Because the JVM is in control, it can contain the program and prevent it from generating side effects outside of the system. As you will see, safety is also enhanced by certain restrictions that exist in the Java language.

In general, when a program is compiled to an intermediate form and then interpreted by a virtual machine, it runs slower than it would run if compiled to executable code. However, with Java, the differential between the two is not so great. Because bytecode has been highly optimized, the use of bytecode enables the JVM to execute programs much faster than you might expect.

Although Java was designed as an interpreted language, there is nothing about Java that prevents on-the-fly compilation of bytecode into native code in order to boost performance. For this reason, the HotSpot technology was introduced not long after Java's initial release. HotSpot provides a Just-In-Time (JIT) compiler for bytecode. When a JIT compiler is part of the JVM, selected portions of bytecode are compiled into executable code in real time, on a piece-by-piece, demand basis. It is important to understand that it is not practical to compile an entire Java program into executable code all at once, because Java performs various

run-time checks that can be done only at run time. Instead, a JIT compiler compiles code as it is needed, during execution. Furthermore, not all sequences of bytecode are compiled—only those that will benefit from compilation. The remaining code is simply interpreted. However, the just-in-time approach still yields a significant performance boost. Even when dynamic compilation is applied to bytecode, the portability and safety features still apply, because the JVM is still in charge of the execution environment.

Servlets: Java on the Server Side

Java would also be useful on the server side. The result was the *servlet*. A servlet is a small program that executes on the server. Just as applets dynamically extend the functionality of a web browser, servlets dynamically extend the functionality of a web server. Thus, with the advent of the servlet,

Java spanned both sides of the client/server connection. Servlets are used to create dynamically generated content that is then served to the client. For example, an online store might use a servlet to look up the price for an item in a database. The price information is then used to dynamically generate a web page that is sent to the browser. Although dynamically generated content is available through mechanisms such as CGI (Common Gateway Interface), the servlet offers several advantages, including increased performance.

Because servlets (like all Java programs) are compiled into bytecode and executed by the JVM, they are highly portable. Thus, the same servlet can be used in a variety of different server environments. The only requirements are that the server support the JVM and a servlet container.

JAVA BUZZWORDS

Following are the features or buzzwords of Java language which made it popular:

1.Simple
2.Secure
3.Portable
4.Object-Oriented
5.Robust
6.Multithreaded
7.Architecture neutral

8.Interpreted
9.High Performance
10.Distributed
11.Dynamic

Simple:

•Java easy to learn
•It is easy to write programs using Java
•Expressiveness is more in Java.
•Most of the complex or confusing features in C++ are removed in Java like pointers etc..

Secure:

•Java provides data security through encapsulation.
•Also we can write applets in Java which provides security.
•An applet is a small program which can be downloaded from one computer to another automatically.
•There is no need to worry about applets accessing the system resources which may compromise security.
•Applets are run within the JVM which protects from unauthorized or illegal access to system resources.

Portable:

•Applications written using Java are portable in the sense that they can be executed on any kind of computer containing any CPU or any operating system.
•When an application written in Java is compiled, it generates an intermediate code file called as "bytecode".
•Bytecode helps Java to achieve portability.
•This bytecode can be taken to any computer and executed directly.

Object - Oriented:

•Java follows object oriented model.
•So, it supports all the features of object oriented model like:

Encapsulation
Inheritance
Polymorphism
Abstraction

Robust:

•A program or an application is said to be robust(reliable) when it is able to give some response in any kind of context.
•Java's features help to make the programs robust. Some of those features are:

1. Type checking
2. Exception handling

Multithreaded:

•Java supports multithreading which is not supported by C and C++.
•A thread is a light weight process.
•Multithreading increases CPU efficiency.
•A program can be divided into several threads and each thread can be executed concurrently or in parallel with the other threads.
•Real world example for multithreading is computer. While we are listening to music, at the same time we can write in a word document or play a game.

Architecture - Neutral:

•Byte code helps Java to achieve portability.
•Byte code can be executed on computers having any kind of operating system or any kind of CPU.
•Since Java applications can run on any kind of CPU, Java is architecture – neutral.

Interpreted and High Performance:

•In Java 1.0 version there is an interpreter for executing the byte code. As interpreter is quite slow when compared to a compiler, java programs used to execute slowly.
•After Java 1.0 version the interpreter was replaced with JIT(Just-In-Time) compiler.
•JIT compiler uses Sun Micro system's Hot Spot technology.
•JIT compiler converts the byte code into machine code piece by piece and caches them for future use.
•This enhances the program performance means it executes rapidly.

Distributed:

•Java supports distributed computation using Remote Method Invocation (RMI) concept.
•The server and client(s) can communicate with another and the computations can be divided among several computers which makes the programs to execute rapidly.
•In distributed systems, resources are shared.

Dynamic:

•The Java Virtual Machine(JVM) maintains a lot of runtime information about the program and the objects in the program.
•Libraries are dynamically linked during runtime.
•So, even if you make dynamic changes to pieces of code, the program is not effected.

The Evolution of Java

The initial release of Java was nothing short of revolutionary, but it did not mark the end of Java's era of rapid innovation. Unlike most other software systems that usually settle into a pattern of small, incremental improvements, Java continued to evolve at an explosive pace. Soon after the release of Java 1.0, the designers of Java had already created Java 1.1. The features added by Java 1.1 were more significant and substantial than the increase in the minor revision number would have you think. Java 1.1 added many new library elements, redefined the way events are handled, and reconfigured many features of the 1.0 library. It also deprecated (rendered obsolete) several features originally defined by Java 1.0. Thus, Java 1.1 both added to and subtracted from attributes of its original specification.

The next major release of Java was Java 2, where the "2" indicates "second generation." The creation of Java 2 was a watershed event, marking the beginning of Java's "modern age." The first release of Java 2 carried the version number 1.2. It may seem odd that the first release of Java 2 used the 1.2 version number. The reason is that it originally referred to the internal version number of the Java libraries, but then was generalized to refer to the entire release. With Java 2, Sun repackaged the Java product as J2SE (Java 2 Platform Standard Edition), and the version numbers began to be applied to that product.

Java 2 added support for a number of new features, such as Swing and the Collections Framework, and it enhanced the Java Virtual Machine and various programming tools. Java 2 also contained a few deprecations. The most important affected the **Thread** class in which the methods **suspend( )**, **resume( )**, and **stop( )** were deprecated.

J2SE 1.3 was the first major upgrade to the original Java 2 release. For the most part, it added to existing functionality and "tightened up" the development environment. In general, programs written for version 1.2 and those written for version 1.3 are source-code compatible. Although version 1.3 contained a smaller set of changes than the preceding three major releases, it was nevertheless important.

The release of J2SE 1.4 further enhanced Java. This release contained several important upgrades, enhancements, and additions. For example, it added the new keyword **assert**, chained exceptions, and a channel-based I/O subsystem. It also made changes to the Collections Framework and the networking classes. In addition, numerous small changes were made throughout. Despite the significant number of new features, version 1.4 maintained nearly 100 percent source-code

compatibility with prior versions. The next release of Java was J2SE 5, and it was revolutionary. Unlike most of the previous Java upgrades, which offered important, but measured improvements, J2SE 5 fundamentally expanded the scope, power, and range of the language. To grasp the magnitude of the changes that J2SE 5 made to Java, consider the following list of its major new features:

- Generics
- Annotations
- Autoboxing and auto-unboxing
- Enumerations
- Enhanced, for-each style **for** loop
- Variable-length arguments (varargs)
- Static import
- Formatted I/O
- Concurrency utilities

Each item in the list represented a significant addition to the Java language. Some, such as generics, the enhanced **for**, and varargs, introduced new syntax elements. Others, such as autoboxing and auto-unboxing, altered the semantics of the language. Annotations added an entirely new dimension to programming. In all cases, the impact of these additions went beyond their direct effects. They changed the very character of Java itself. The importance of these new features is reflected in the use of the version number "5." The next version number for Java would normally have been 1.5. However, the new features were so significant that a shift from 1.4 to 1.5 just didn't seem to express the magnitude of the change. Instead, Sun elected to increase the version number to 5 as a way of emphasizing that a major event was taking place. Thus, it was named J2SE 5, and the developer's kit was called JDK 5. However, in order to maintain consistency, Sun decided to use 1.5 as its internal version number, which is also referred to as the *developer version* number.

The "5" in J2SE 5 is called the *product version* number. The next release of Java was called Java SE 6. Sun once again decided to change the name of the Java platform. First, notice that the "2" was dropped. Thus, the platform was now named *Java SE*, and the official product name was *Java Platform, Standard Edition 6*.

The Java Development Kit was called JDK 6. As with J2SE 5, the 6 in Java SE 6 is the product version number. The internal, developer version number is 1.6.Java SE 6 built on the base of J2SE 5, adding incremental improvements. Java SE 6 added no major features to the Java language proper, but it did enhance the API libraries, added several

new packages, and offered improvements to the runtime. It also went through several updates during its (in Java terms) long life cycle, with several upgrades added along the way.

In general, Java SE 6 served to further solidify the advances made by J2SE 5.Java SE 7 was the next release of Java, with the Java Development Kit being called JDK 7,and an internal version number of 1.7. Java SE 7 was the first major release of Java since Sun Microsystems was acquired by Oracle. Java SE 7 contained many new features, including significant additions to the language and the API libraries. Upgrades to the Java run-time system that support non-Java languages were also included, but it is the language and library additions that were of most interest to Java programmers.

The new language features were developed as part of *Project Coin*. The purpose of Project Coin was to identify a number of small changes to the Java language that would be incorporated into JDK 7. Although these features were collectively referred to as "small," the effects of these changes have been quite large in terms of the code they impact. In fact, for many programmers, these changes may well have been the most important new features in Java SE 7. Here is a list of the language features added by JDK 7:

• A **String** can now control a **switch** statement.
• Binary integer literals.
• Underscores in numeric literals.
• An expanded **try** statement, called ***try****-with-resources*, that supports automatic resource management. (For example, streams can be closed automatically when they are no longer needed.)
• Type inference (via the *diamond* operator) when constructing a generic instance.
• Enhanced exception handling in which two or more exceptions can be caught by a single **catch** (multi-catch) and better type checking for exceptions that are rethrown.
• Although not a syntax change, the compiler warnings associated with some types of varargs methods were improved, and you have more control over the warnings. As you can see, even though the Project Coin features were considered small changes to the language, their benefits were much larger than the qualifier "small" would suggest. In particular, the **try**-*with-resources* statement has profoundly affected the way that stream-based code is written. Also, the ability to use a **String** to control a **switch** statement was a long desired improvement that simplified coding in many situations.

Java SE 7 made several additions to the Java API library. Two of the most important were the enhancements to the NIO Framework and the addition of the Fork/Join Framework. NIO (which originally stood for *New I/O*) was added to Java in version 1.4. However, the changes added by Java SE 7 fundamentally expanded its capabilities. So significant were the changes, that the term *NIO.2* is often used.

The Fork/Join Framework provides important support for *parallel programming*. Parallel programming is the name commonly given to the techniques that make effective use of computers that contain more than one processor, including multicore systems. The advantage that multicore environments offer is the prospect of significantly increased program performance. The Fork/Join Framework addressed parallel programming by

- Simplifying the creation and use of tasks that can execute concurrently
- Automatically making use of multiple processors

Therefore, by using the Fork/Join Framework, you can easily create scalable applications that automatically take advantage of the processors available in the execution environment. Of course, not all algorithms lend themselves to parallelization, but for those that do, a significant improvement in execution speed can be obtained.

An Overview of Java:

OBJECT ORIENTED PROGRAMMING

THE KEY ATTRIBUTES OF OBJECT ORIENTED PROGRAMMING:

- Object
- Class
- Data Abstraction and Encapsulation
- Dynamic Binding
- Message Passing
- Inheritance
- Polymorphism

Object:

Object is a collection of number of entities. Object take up space in the memory. Objects are instances of classes. When a program is executed, the objects interact by sending messages to one another. Each object contains data and code to manipulate the data. Objects can interact without having no details of each other's data or code.

Class:

Class is a collection of objects of similar type. Objects are variables of the type class. Once a class has been defined, we can create any number of objects belonging to that class.

Ex: banana, orange are the objects of class Fruit.

Fruit mango;

By above mango object is created for class Fruit.

Data Abstraction and Encapsulation:

Combining data and functions into a single unit called class and the process is known as Encapsulation. Data encapsulation is important feature of a class. Class contains both data and functions. Data is not accessible from the outside world and only those functions which are present in the class can access the data. The insulation of the data from direct access by the program is called data hiding or data abstraction. Hiding the complexity of program is called abstraction only essential features are represented. In short we can say that internal working is hidden.

Dynamic Binding:

Refers to linking of function call with function defination is called binding and when it is take place at run time called dynamic binding.

Message Passing:

The process by which one object can interact with other object is called message passing.

Inheritance:

It is the process by which object of one class aquire the properties or features of objects of another class. The concept of inheritance provide the idea of reusability means we can add additional features to an existing class without Modifying it. This is possible by driving a new class from the existing one. The new class will have the combined features of both the classes.

Example: **Parrot** is a part of the class flying bird which is again a part of the class bird.

Polymorphism:

A Greek term means ability to take more than one form. An operation may exhibit different behaviors in different instances. The behavior depends upon the types of data used in the operation.

Example:

- Operator Overloading
  You can redefine or overload most of the built in operators.
- Function Overloading
  You can have multiple definitions for the same function name in the same scope. The definition of the function must differ from each other by the type and/or the number of arguments in the argument list but not only by return type.

SIMPLE JAVA PROGRAM

```
import java.io.*;

class Example

{

 public static void main(String args[])

{

 System.out.println("This is a simple java program");

}

}
```

Two Control Statements

The if Statement

The Java **if** statement works much like the IF statement in any other language. Further, it is syntactically identical to the **if** statements in C, C++, and C#. Its simplest form is shown here: if*(condition) statement;*
Here, *condition* is a Boolean expression. If *condition* is true, then the statement is executed.
If *condition* is false, then the statement is bypassed. Here is an example:
if(num < 100) System.out.println("num is less than 100");

In this case, if **num** contains a value that is less than 100, the conditional expression is true, and **println( )** will execute. If **num** contains a value greater than or equal to 100, then the **println( )** method is bypassed.

```
/*
Demonstrate the if.
Call this file "IfSample.java".
*/
class IfSample {
public static void main(String args[]) {
int x, y;
x = 10;
y = 20;
if(x < y) System.out.println("x is less than y");
x = x * 2;
if(x == y) System.out.println("x now equal to y");
if(x > y) System.out.println("x now greater than y");
// this won't display anything
if(x == y) System.out.println("you won't see this");
}
}
```

The output generated by this program is shown here:

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

Notice one other thing in this program. The line

```
int x, y;
```

declares two variables, **x** and **y**, by use of a comma-separated list.

The for Loop

The simplest form of the **for** loop is shown here:

for*(initialization; condition; iteration) statement;*

In its most common form, the *initialization* portion of the loop sets a loop control variable to an initial value. The *condition* is a Boolean expression that tests the loop control variable. If the outcome of that test is true, the **for** loop continues to iterate. If it is false, the loop terminates. The *iteration* expression determines how the loop control variable is changed each time the loop iterates. Here is a short program that illustrates the **for** loop:

```
/*
Demonstrate the for loop.
Call this file "ForTest.java".
*/
class ForTest {
public static void main(String args[]) {
int x;
for(x = 0; x<10; x = x+1)
System.out.println("This is x: " + x);
}
}
```

This program generates the following output:

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

In this example, **x** is the loop control variable. It is initialized to zero in the initialization portion of the **for**. At the start of each iteration (including the first one), the conditional test **x < 10** is performed. If the outcome of this test is true, the **println( )** statement is executed, and then the iteration portion of the loop is executed, which increases **x** by 1.This process continues until the conditional test is false.

Using Blocks of Code

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

Consider this **if** statement:

```
if(x < y) { // begin a block
x = y;
```

```
y = 0;
} // end of block
```
Here, if **x** is less than **y**, then both statements inside the block will be executed. Thus, the two statements inside the block form a logical unit, and one statement cannot execute without the other also executing. The key point here is that whenever you need to logically link two or more statements, you do so by creating a block.

The following program uses a block of code as the target of a **for** loop.

```
/*
Demonstrate a block of code.
Call this file "BlockTest.java"
*/
class BlockTest {
public static void main(String args[]) {
int x, y;
y = 20;
// the target of this loop is a block
for(x = 0; x<10; x++) {
System.out.println("This is x: " + x);
System.out.println("This is y: " + y);
y = y - 2;
}
}
}
```

Lexical Issues

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

Whitespace

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

Identifiers

Identifiers are used to name things, such as classes, variables, and methods. An identifier may be any descriptive sequence of uppercase and lowercase letters, numbers, or the underscore and dollar-sign characters.

(The dollar-sign character is not intended for general use.) They must not begin with a number, lest they be confused with a numeric literal. Again, Java is case-sensitive, so **VALUE** is a different identifier than **Value**.

Some examples of valid identifiers are
AvgTemp count a4 $test this_is_ok
Invalid identifier names include these:
2count high-temp Not/ok

Literals
A constant value in Java is created by using a *literal* representation of it. For example, here are some literals:
100 98.6 'X' "This is a test"
Left to right, the first literal specifies an integer, the next is a floating-point value, the third is a character constant, and the last is a string. A literal can be used anywhere a value of its type is allowed.

Comments
As mentioned, there are three types of comments defined by Java. You have already seen two: single-line and multiline. The third type is called a *documentation comment*. This type of comment is used to produce an HTML file that documents your program. The documentation comment begins with a /** and ends with a */.

Separators
In Java, there are a few characters that are used as separators. The most commonly used separator in Java is the semicolon. As you have seen, it is used to terminate statements. The separators are shown in the following table:

Symbol	Name	Purpose
( )	Parentheses	Used to contain lists of parameters in method definition and invocation. Also used for defining precedence in expressions, containing expressions in control statements, and surrounding cast types.
{ }	Braces	Used to contain the values of automatically initialized arrays. Also used to define a block of code, for classes, methods, and local scopes.
[ ]	Brackets	Used to declare array types. Also used when dereferencing array values.
;	Semicolon	Terminates statements.
,	Comma	Separates consecutive identifiers in a variable declaration. Also used to chain statements together inside a **for** statement.
.	Period	Used to separate package names from subpackages and classes. Also used to separate a variable or method from a reference variable.
::	Colons	Used to create a method or constructor reference. (Added by JDK 8.)

The Java Class Libraries

The sample programs shown in this chapter make use of two of Java's built-in methods:
println( ) and **print( )**. As mentioned, these methods are available through **System.out**. **System** is a class predefined by Java that is automatically included in your programs. In the larger view, the Java environment relies on several built-in class libraries that contain many built-in methods that provide support for such things as I/O, string handling, networking, and graphics. The standard classes also provide support for a graphical user interface (GUI). Thus, Java as a totality is a combination of the Java language itself, plus its standard classes.

Data Types, Arrays and Variables:

The Primitive Types:

Java defines eight primitive types of data: byte, short, int, long, char, float, double, and boolean.

The primitive types are also commonly referred to as simple types, and both terms will be used in this book. These can be put in four groups:

• **Integers**: This group includes byte, short, int, and long, which are for whole-valued signed numbers.

• **Floating-point numbers**: This group includes float and double, which represent numbers with fractional precision.

• **Characters:** This group includes char, which represents symbols in a character set, like letters and numbers.

• **Boolean:** This group includes boolean, which is a special type for representing true/false values.

You can use these types as-is, or to construct arrays or your own class types. Thus, they form the basis for all other types of data that you can create.

The primitive types represent single values—not complex objects. Although Java isotherwise completely object-oriented, the primitive types are not. They are analogous to the simple types found in most other non–object-oriented languages. The reason for this is efficiency.Making the primitive types into objects would have degraded performance too much.

The primitive types are defined to have an explicit range and mathematical behavior. Languages such as C and C++ allow the size of an integer to vary based upon the dictates of the execution environment. However, Java is different. Because of Java's portability requirement, all data types have a strictly defined range.

For example, an int is always 32 bits,

Regardless of the particular platform. This allows programs to be written that are guaranteed to run without porting on any machine architecture. While strictly specifying the size of an integer may cause a small loss of performance in some environments, it is necessary in order to achieve portability.

Let's look at each type of data in turn.

Integers

Java defines four integer types: byte, short, int, and long.

All of these are signed, positive and negative values. Java does not support unsigned, positive-only integers. Many other computer languages support both signed and unsigned integers. However, Java's designers felt that unsigned integers were unnecessary. Specifically, they felt that the concept of unsigned was used mostly to specify the behavior of the high-order bit, which defines the sign of an integer value. Java manages the meaning of the high-order bit differently, by adding a special "unsigned right shift" operator. Thus, the need for an unsigned integer type was eliminated.

The width of an integer type should not be thought of as the amount of storage it consumes, but rather as the behavior it defines for variables and expressions of that type. The Java run-time environment is free to use whatever size it wants, as long as the types behave as you declared them. 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

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

short s;

short t;

int:

The most commonly used integer type is int. It is a signed 32-bit type that has a range from 2,147,483,648 to 2,147,483,647. In addition to other uses, variables of type int are commonly employed to control loops and to index arrays. Although you might think that using a byte or short would be more efficient than using an int in situations in which the larger range of an int is not needed, this may not be the case. The reason is that when byte and short values are used in an expression they are promoted to int when the expression is evaluated. Therefore, int is often the best choice when an integer is needed.

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.

Floating-Point Types:

Floating-point numbers, also known as real numbers, are used when evaluating expressions that require fractional precision. For example, calculations such as square root, or transcendental such as sine and cosine, result in a value whose precision requires a floating-point type.

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

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 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);

}

}
```

Characters

In Java, the data type used to store characters is char.

Note: char in Java is not the same as char in C or C++. In C/C++, char is 8 bits wide. This is not the case in Java. Instead, Java uses Unicode to represent characters. Unicode defines a fully international character set that can represent all of the characters found in all human languages. It is a unification of dozens of character sets, such as Latin, Greek, Arabic, Cyrillic, Hebrew, Katakana, Hangul, and many more. For this purpose, it requires 16 bits.

Thus, in **Java char is a 16-bit type**. The range of a char is 0 to 65,536. There are no negative chars.

The standard set of characters known as ASCII still ranges from 0 to 127 as always, and the extended 8-bit character set, ISO-Latin-1, ranges from 0 to 255. Since Java is designed to allow programs to be written for worldwide use, it makes sense that it would use Unicode to represent characters. Of course, the use of Unicode is somewhat inefficient for languages such as English, German, Spanish, or French, whose characters can easily be contained within 8 bits.

But such is the price that must be paid for global portability.

Here is a program that demonstrates char variables:

```
// Demonstrate char data type.

class CharDemo {

public static void main(String args[]) {

char ch1, ch2;

ch1 = 88;  // code for X

ch2 = 'Y';

System.out.print("ch1 and ch2: ");

System.out.println(ch1 + " " + ch2);
```

```
}

}
```

This program displays the following

output:

```
ch1 and ch2: X Y
```

Notice that ch1 is assigned the value 88, which is the ASCII (and Unicode) value that corresponds to the letter X. As mentioned, the ASCII character set occupies the first 127 values in the Unicode character set. For this reason, all the "old tricks" that you may have used with characters in other languages will work in Java, too.

Although char is designed to hold Unicode characters, it can also be thought of as an integer type on which you can perform arithmetic operations. For example, you can add two characters together, or increment the value of a character variable. Consider the following program:

```
// char variables behave like integers.

class CharDemo2 {

public static void main(String args[]) {

char ch1;

ch1 = 'X';

System.out.println("ch1 contains " + ch1);

ch1++; // increment ch1

System.out.println("ch1 is now " + ch1);

}

}
```

The **output** generated by this program is shown here:

```
ch1 contains X

ch1 is now Y
```

In the program, ch1 is first given the value X. Next, ch1 is incremented. This results in ch1 containing Y, the next character in the ASCII (and Unicode) sequence.

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.

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 statementChapter 3: Data Types, Variables, and Arrays 39

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 <,isa 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 >.

Escape Sequence: Java supports all escape sequence which is supported by C/ C++. A character preceded by a backslash (\) is an escape sequence and has special meaning to the compiler. When an escape sequence is encountered in a print statement, the compiler interprets it accordingly.

Escape Sequence	Description
\t	Insert a tab in the text at this point.
\b	Insert a backspace in the text at this point.
\n	Insert a newline in the text at this point.
\r	Insert a carriage return in the text at this point.
\f	Insert a form feed in the text at this point.
\'	Insert a single quote character in the text at this point.
\"	Insert a double quote character in the text at this point.
\\	Insert a backslash character in the text at this point.

VARIABLES:

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

Declaring a Variable

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

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

The type is one of Java's atomic types, or the name of a class or interface. (Class and interface types are discussed later in Part I of this book.) The identifier is the name of the variable. You can initialize the variable by specifying an equal sign and a value. Keep in mind that the initialization expression must result in a value of the same (or compatible) type as that specified for the variable. To declare more than one variable of the specified type, use a comma-separated list.

```
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 identifiers that you choose have nothing intrinsic in their names that indicates their type. Java allows any properly formed identifier to have any declared type.

Dynamic Initialization

Although the preceding examples have used only constants as initializers, Java allows variables to be initialized dynamically, using any expression valid at the time the variable is declared.

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

```
// Demonstrate dynamic initialization.

class DynInit {

public static void main(String args[]) {

double a = 3.0, b = 4.0;

// c is dynamically initialized

double c = Math.sqrt(a * a + b * b);

System.out.println("Hypotenuse is " + c);
```

```
}

}
```

Here, three local variables—a, b, and c—are declared. The first two, a and b, are initialized by constants. However, c is initialized dynamically to the length of the hypotenuse (using the Pythagorean theorem). The program uses another of Java's built-in methods, sqrt( ), which is a member of theMath class, to compute the square root of its argument. The key point here is that the initialization expression may use any element valid at the time of the initialization, including calls to methods, other variables, or literals.

The Scope and Lifetime of Variables

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

A block is begun with an opening curly brace and ended by a closing curly brace. A block defines a scope. Thus, each time you start a new block, you are creating a new scope. A scope determines what objects are visible to other parts of your program. It also determines the lifetime of those objects.

Many other computer languages define two general categories of scopes: **global and local.**

However, these traditional scopes do not fit well with Java's strict, object-oriented model.

In Java, the two major scopes are those defined by a class and those defined by a method. Even this distinction is somewhat artificial. However, since the class scope has several unique properties and attributes that do not apply to the scope defined by a method, this distinction makes some sense. Because of the differences, a discussion of class scope (and variables declared within it) is deferred, when classes are described.

For now, we will only examine the scopes defined by or within a method.

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.

As a general rule, variables declared inside a scope are not visible (that is, accessible) to code that is defined outside that scope. Thus, when you declare a variable within a scope, you are localizing that variable and protecting it from unauthorized access and/or modification.

Indeed, the **scope rules provide the foundation for encapsulation.**

Scopes can be nested.

For example, each time you create a block of code, you are creating a new, nested scope. When this occurs, the outer scope encloses the inner scope. This means that objects declared in the outer scope will be visible to code within the inner scope. However, the reverse is not true. Objects declared within the inner scope will not be visible outside it.

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

```
// Demonstrate block scope.

class Scope {

public static void main(String args[]) {

int x; // known to all code within main

x = 10;

if(x == 10) { // start new scope

int y = 20; // known only to this block

// x and y both known here.

System.out.println("x and y: " + x + " " + y);

x = y * 2;

}

// y = 100; // Error! y not known here

// x is still known here.

System.out.println("x is " + x);

}

}
```

As the comments indicate, the variable x is declared at the start of main( )'s scope and is accessible to all subsequent code within main( ). Within the if block, y is declared. Since a block defines a scope, y is only visible to other code within its block. This is why outside of its block, the line y = 100; is commented out.

Within a block, variables can be declared at any point, but are valid only after they are declared. Thus, if you define a variable at the start of a method, it is available to all of the code within that method. Conversely, if you declare a variable at the end of a block, it is effectively useless, because no code will have access to it.

For example, this fragment is invalid because count cannot be used prior to its declaration:

```
// This fragment is wrong!

count = 100; // oops!  cannot use count before it is declared!

int count;
```

```
// Demonstrate lifetime of a variable.

class LifeTime {

public static void main(String args[]) {

int x;

for(x = 0; x < 3; x++) {

int y = -1; // y is initialized each time block is entered

System.out.println("y is: " + y); // this always prints -1

y = 100;

System.out.println("y is now: " + y);

}

}

}
```

The output generated by this program is shown here:

```
y is: -1
```

```
y is now: 100

y is: -1

y is now: 100

y is: -1

y is now: 100
```

As you can see, y is reinitialized to –1 each time the inner for loop is entered. Even though it is subsequently assigned the value 100, this value is lost.

One last point: Although blocks can be nested, you cannot declare a variable to have the same name as one in an outer scope. For example, the following program is illegal:

```
// This program will not compile

class ScopeErr {

public static void main(String args[]) {

 int bar = 1;

{              // creates a new scope

int bar = 2; // Compile-time error – bar already defined!

}

}

}
```

Type Conversion and Casting:

If you have previous programming experience, then you already know that it is fairly common to assign a value of one type to a variable of another type. If the two types are compatible, then Java will perform the conversion automatically.

For example, it is always possible to assign an int value to a long variable. However, not all types are compatible, and thus, not all type conversions are implicitly allowed. For instance, there is no automatic conversion defined from double to byte. Fortunately, it is still possible to obtain a conversion between in compatible types. To do so, you must use

a cast, which performs **an explicit conversion** between incompatible types.

Let's look at both automatic type conversions and casting.

Java's Automatic Conversions

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

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

When these two conditions are met, a widening conversion takes place. For example, the int type is always large enough to hold all valid byte values, so no explicit cast statement is required.

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

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

Casting Incompatible Types

Although the automatic type conversions are helpful, they will not fulfill all needs. For example, what 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. For example, the following fragment casts an int to a byte. If the

integer’s value is larger than the range of a byte, it will be reduced modulo (the remainder of an integer division by the) byte’s range.

```
int a;

byte b;

// ...

b = (byte) a;
```

Adifferent type of conversion will occur when a floating-point value is assigned to an integer type: truncation. As you know, integers do not have fractional components. Thus, when a floating-point value is assigned to an integer type, the fractional component is lost.

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. Of course, if the size of the whole number component is too large to fit into the target integer type, then that value will be reduced modulo the target type’s range.

The following program demonstrates some **type conversions that require casts:**

```
// Demonstrate casts.

class Conversion {

public static void main(String args[]) {

byte b;

int i = 257;

double d = 323.142;

System.out.println("\nConversion of int to byte.");

b = (byte) i;

System.out.println("i and b " + i + " " + b);

System.out.println("\nConversion of double to int.");

i = (int) d;

System.out.println("d and i " + d + " " + i);
```

```
System.out.println("\nConversion of double to byte.");

b = (byte) d;

System.out.println("d and b " + d + " " + b);

}

}
```

This program generates the following **output:**

Conversion of int to byte.

i and b 257 1

Conversion of double to int.

d and i 323.142 323

Conversion of double to byte.

d and b 323.142 67Let's look at each conversion. When the value 257 is cast into a byte variable, the result is the remainder of the division of 257 by 256 (the range of a byte), which is 1 in this case.

When the d is converted to an int, its fractional component is lost. When d is converted to a byte, its fractional component is lost, and the value is reduced modulo 256, which in this case is 67.

ARRAYS:

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

On which memory, arrays are created in java?

Arrays are created on dynamic memory by JVM. There is no question of static memory in Java; everything (variable, array, object etc.) is created on dynamic memory only.

Arrays: An array represents a group of elements of same data type. Arrays are generally categorized into two types:

- Single Dimensional arrays (or 1 Dimensional arrays)

- Multi-Dimensional arrays (or 2 Dimensional arrays, 3 Dimensional arrays, …)

Single Dimensional Arrays:

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

type var-name[ ];

Here, type declares the base type of the array. The base type determines the data type of each element that comprises the array. Thus, the base type for the array determines what type of data the array will hold.

A one dimensional array or single dimensional array represents a row or a column of elements. For example, the marks obtained by a student in 5 different subjects can be represented by a 1D array.

- We can declare a one dimensional array and directly store elements at the time of its declaration, as*: int marks[] = {50, 60, 55, 67, 70};*
- We can create a 1D array by declaring the array first and then allocate memory for it by using new operator, as:

array-var = new type[size];

Here, type specifies the type of data being allocated, size specifies the number of elements in the array, and array-var is the array variable that is linked to the array. That is, to use new to allocate an array, you must specify the type and number of elements to allocate. The elements in the array allocated by new will automatically be initialized to zero.

int marks[]; //declare marks array

marks = new int[5]; //allot memory for storing 5 elements

These two statements also can be written as:

int marks [] = new int [5];

- We can pass the values from keyboard to the array by using a loop, as given here

```
for(int i=0;i<5;i++)
{
//read integer values from keyboard and store into marks[i]
```

BufferedReader br = new BufferedReader (new InputStreamReader (System.in));
Marks[i]=Integer.parseInt(br.readLine());

}

Let us examine some more examples for 1D array:

float salary[]={5670.55f,12000f};

float[] salary={5670.55f,12000f};

string names[]=new string[10];

string[] names={'v','r'};

Let's review: Obtaining an array is a two-step process.

First, you must declare a variable of the desired array type.

Second, you must allocate the memory that will hold the array, using new, and assign it to the array variable. Thus, in Java all arrays are dynamically allocated.

Once you have allocated an array, you can access a specific element in the array by specifying its index within square brackets. All array indexes start at zero.

The advantage of using arrays is that they simplify programming by replacing a lot of statements by just one or two statements. In C/C++, by default, arrays are created on static memory unless pointers are used to create them. In java, arrays are created on dynamic memory i.e., allotted at runtime by JVM.

Program : Write a program to accept elements into an array and display the same.

```
// program to accept elements into an array and display the same.

import java.io.*;

class ArrayDemo1

{

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

  {  //Create a BufferedReader class object (br)
```

```
    BufferedReader br = new BufferedReader (new InputStreamReader (System.in));
    System.out.println (“How many elements: “ );
int n = Integer.parseInt (br.readLine ());
//create a 1D array with size n
    int a[] = new int[n];
    System.out.print ("Enter elements into array : ");
for (int i = 0; i<n;i++)
      a [i] = Integer.parseInt ( br.readLine ());
System.out.print (“The entered elements in the array are: “);
    for (int i =0; i < n; i++)
      System.out.print (a[i] + “\t”);
  }
}
```

Output:

```
D:/kumar>javac ArrayDemo1.java
D:/kumar>java ArrayDemo1
How many elements:5
Enter elements into arry:10 20 30 40      50
The entered elements in the array are:10 20     30     40     50
D:/kumar>
```

Multi-Dimensional Arrays (2D, 3D … arrays):

A two dimensional array is a combination of two or more (1D) one dimensional arrays. A three dimensional array is a combination of two or more (2D) two dimensional arrays.

- **Two Dimensional Arrays (2d array):**

A two dimensional array represents several rows and columns of data. To represent a two dimensional array, we should use two pairs of square braces [ ] [ ] after the array name. For example, the marks obtained by a group of students in five different subjects can be represented by a 2D array.

o We can declare a two dimensional array and directly store elements at the time of its declaration, as:

int marks[] [] = {{50, 60, 55, 67, 70},{62, 65, 70, 70, 81}, {72, 66, 77, 80, 69} };

o We can create a two dimensional array by declaring the array first and then we can allot

memory for it by using new operator as:

int marks[ ] [ ]; //declare marks array

marks = new int[3][5]; //allot memory for storing 15 elements.

These two statements also can be written as*: int marks [ ][ ] = new int[3][5];*

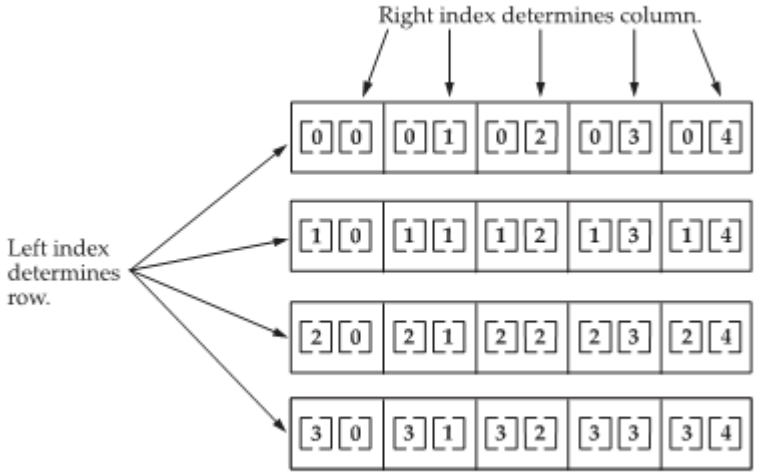
Program : Write a program to take a 2D array and display its elements in the form of a matrix.

```
//Displaying a 2D array as a matrix

class Matrix

{  public static void main(String args[])

  {  //take a 2D array

    int x[ ][ ] = {{1, 2, 3}, {4, 5, 6} };

    // display the array elements

    for (int i = 0 ; i < 2 ; i++)

    {  System.out.println ();

      for (int j = 0 ; j < 3 ; j++)

        System.out.print(x[i][j] + "\t");

    }
```

```
  }
}
```

Output:

```
D:/kumar>javac Matrix.java

D:/kumar>java Matrix
1       2       3
4       5       6
```



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

- **<u>Three Dimensional arrays (3D arrays):</u>**

We can consider a three dimensional array as a combination of several two dimensional arrays. To represent a three dimensional array, we should use three pairs of square braces [ ] [ ] after the array name.

o We can declare a three dimensional array and directly store elements at the time of its

declaration, as:

int arr[ ] [ ] [ ] = {{{50, 51, 52},{60, 61, 62}}, {{70, 71, 72}, {80, 81, 82}}};

o We can create a three dimensional array by declaring the array first and then we can allot memory for it by using new operator as:

int arr[ ] [ ] = new int[2][2][3]; //allot memory for storing 15 elements.

```
//example for 3-D array
class ThreeD
{
	public static void main(String args[])
	{
		int dept, student, marks, tot=0;
int
arr[][][]={{{50,51,52},{60,61,62}},{{70,71,72},{80,81,82}},{{65,66,
67},{75,76,77}}};
		for(dept=0;dept<3;dept++)
		{
			System.out.println("dept"+(dept+1)+":");
			for(student=0;student<2;student++)
			{

	System.out.print("student"+(student+1)+"marks:");
				for(marks=0;marks<3;marks++)
				{

	System.out.print(arr[dept][student][marks]+" ");
					tot+=arr[dept][student][marks];
				}
				System.out.println("total:"+tot);
				tot=0;
			}
			System.out.println();
		}
	}
```

}

arrayname.length:

If we want to know the size of any array, we can use the property 'length' of an array. In case of 2D, 3D length property gives the number of rows of the array.

Alternative Array Declaration Syntax

There is a second form that may be used to declare an array:

type[ ] var-name;

Here, the square brackets follow the type specifier, and not the name of the array variable.

For example, the following two declarations are equivalent:

int al[] = new int[3];

int[] a2 = new int[3];

The following declarations are also equivalent:

char twod1[][] = new char[3][4];

char[][] twod2 = new char[3][4];

This alternative declaration form offers convenience when declaring several arrays at the same time. For example,

int[] nums, nums2, nums3; // create three arrays

creates three array variables of type int. It is the same as writing

int nums[], nums2[], nums3[]; // create three arrays

The alternative declaration form is also useful when specifying an array as a return type for a method.

Using Command-Line Arguments:

Sometimes you will want to pass information into a program when you run it. This is accomplished by passing command-line arguments to main( ).

A command-line argument is the information that directly follows the program's name on the command line when it is executed. To access the command-line arguments inside a Java program is quite easy they are stored as strings in a String array passed to the args parameter of main( ).

The first command-line argument is stored at args[0], the second at args[1], and so on.

For example, the following program displays all of the command-line arguments that it is called with:

// Display all command-line arguments.

```
class CommandLine {

public static void main(String args[]) {

for(int i=0; i<args.length; i++)

System.out.println("args[" + i + "]: " +args[i]);

}

}
```

Try executing this program, as shown here:

java CommandLine

this is a test 100 -1

When you do, you will see the following **output:**

args[0]: this

args[1]: is

args[2]: a

args[3]: test

args[4]: 100

args[5]: -1

REMEMBER All command-line arguments are passed as strings. You must convert numeric values to their internal forms manually.

STRINGS

Strings:

A String represents group of characters. Strings are represented as String objects in java.

The String class is defined in the java.lang package and hence is implicitly available to all the programs in Java. The String class is declared as final, which means that it cannot be subclassed. It extends the Object class and implements the Serializable, Comparable, and CharSequence interfaces.

Java implements strings as objects of type String. A string is a sequence of characters. Unlike most of the other languages, Java treats a string as a single value rather than as an array of characters.

Creating Strings:

- We can declare a String variable and directly store a String literal using assignment operator.

  String str = "Hello";

- We can create String object using new operator with some data.

  String s1 = new String ("Java");

- We can create a String by using character array also.

  char arr[] = { 'p','r','o',’g’,’r’,’a’,’m’};

- We can create a String by passing array name to it, as:

  String s2 = new String (arr);

- We can create a String by passing array name and specifying which characters we need:

  String s3 = new String (str, 2, 3);

Here starting from 2nd character a total of 3 characters are copied into String s3.

STRING HANDLING IN JAVA :

String Class Methods:

Method	Description
String concat (String str)	Concatenates calling String with str. **Note:** + also used to do the same
int length ()	Returns length of a String
char charAt (int index)	Returns the character at specified location (from 0)
int compareTo (String str)	Returns a negative value if calling String is less than str, a positive value if calling String is greater than str or 0 if Strings are equal.
boolean equals (String str)	Returns true if calling String equals str. **Note:** == operator compares the references of the string objects. It does not compare the contents of the objects. equals () method compares the contents. While comparing the strings, equals () method should be used as it yields the correct result.
boolean equalsIgnoreCase (String str)	Same as above but ignores the case
boolean startsWith (String prefix)	Returns true if calling String starts with prefix
boolean endsWith (String suffix)	Returns true if calling String ends with suffix
int indexOf (String str)	Returns first occurrence of str in String.
int lastIndexOf(String str)	Returns last occurrence of str in the String. **Note:** Both the above methods return negative value, if str not found in calling String. Counting starts from 0.
String replace (char oldchar, char newchar)	returns a new String that is obtained by replacing all characters oldchar in String with newchar.
String substring (int beginIndex)	returns a new String consisting of all characters from beginIndex until the end of the String
String substring (int beginIndex, int endIndex)	returns a new String consisting of all characters from beginIndex until the endIndex.
String toLowerCase ()	converts all characters into lowercase
String toUpperCase ()	converts all characters into uppercase
String trim ()	eliminates all leading and trailing spaces

String represents a sequence of characters. It has fixed length of character sequence. Once a string object has been created than we can't change the character that comprise that string. It is immutable. This allows String to be shared. String object can be instantiated like any other object

String str = new String ("Stanford ");
str += "Lost!!";

Accessor methods:

length(), charAt(i), getBytes(), getChars(istart,iend,gtarget[],itargstart), split(string,delim), toCharArray(), valueOf(g,iradix), substring(iStart [,iEndIndex)]) [returns up to but not including iEndIndex]

Modifier methods:

concat(g), replace(cWhich, cReplacement), toLowerCase(), toUpperCase(), trim().

Boolean test methods:

contentEquals(g), endsWith(g), equals(g), equalsIgnoreCase(g), matches(g), regionMatches(i1,g2,i3,i4), regionMatches(bIgnoreCase,i1,g2,i3,i4), startsWith(g)

Integer test methods:

compareTo(g) [returns 0 if object equals parameter, -1 if object is before parameter in sort order, +1 if otherwise], indexOf(g) [returns position of first occurrence of substring g in the string, -1 if not found], lastIndexOf(g) [returns position of last occurrence of substring g in the string, -1 if not found], length().

Constructors defined in the String class

The String class defines several constructors. The most common constructor of the String class is the one given below:

public String(String value)

This constructor constructs a new String object initialized with the same sequence of the characters passed as the argument. In other words, the newly created String object is the copy of the string passed as an argument to the constructor.
Other constructors defined in the String class are as follows:

public String()

This constructor creates an empty String object. However, the use of this constructor is unnecessary because String objects are immutable.

public String(char[] value)

This constructor creates a new String object initialized with the same sequence of characters currently contained in the array that is passed as the argument to it.

public String(char[] value, int startindex, int len)

This constructor creates a new String object initialized with the same sequence of characters currently contained in the subarray. This subarray is derived from the character array and the two integer values that are

passed as arguments to the constructor. The int variable startindex represents the index value of the starting character of the subarray, and the int variable len represents the number of characters to be used to form the new String object.

public String(StringBuffer sbf)

This constructor creates a new String object that contains the same sequence of characters currently contained in the string buffer argument.

public String(byte[] asciichars)

The array of bytes that is passed as an argument to the constructor contains the ASCII character set. Therefore, this array of bytes is first decoded using the default charset of the platform. Then the constructor creates a new String object initialized with same sequence of characters obtained after decoding the array.

public String(byte[] asciiChars, int startindex, int len)

This constructor creates the String object after decoding the array of bytes and by using the subarray of bytes.

Special String Operations:

Finding the length of string

The String class defines the length() method that determines the length of a string. The length of a string is the number of characters contained in the string. The signature of the length() method is given below:

public int length()

String Concatenation using the + operator

The + operator is used to concatenate two strings, producing a new String object as the result. For example,

```
String sale = "500";
String s = "Our daily sale is" + sale + "dollars";
System.out.println(s);
```

This code will display the string "Our daily sale is 500 dollars".

The + operator may also be used to concatenate a string with other data types. For example,

```
int sale = 500;
String s = "Our daily sale is" + sale + "dollars";
```

System.out.println(s);
This code will display the string "Our daily sale is 500 dollars". In this case, the variable sale is declared as int rather than String, but the output produced is the same. This is because the int value contained in the variable sale is automatically converted to String type, and then the + operator concatenates the two strings.

String Comparison

The String class defines various methods that are used to compare strings or substrings within strings. Each of them is discussed in the following sections:

Note: Since strings are stored as a memory address, the == operator can't be used for comparisons. Use equals() and equalsIgnoreCase() to do comparisons. A simple example is:

equals()

The equals() method is used to check whether the Object that is passed as the argument to the method is equal to the String object that invokes the method. It returns true if and only if the argument is a String object that represents the same sequence of characters as represented by the invoking object. The signature of the equals() method is as follows:

boolean i=str1.equals(str2)

equalsIgnoreCase()

The equalsIgnoreCase() method is used to check the equality of the two String objects without taking into consideration the case of the characters contained in the two strings. It returns true if the two strings are of the same length and if the corresponding characters in the two strings are the same ignoring case. The signature of the equalsIgnoreCase() method is:

boolean i=str1.equalsIgnoreCase(str2)

compareTo()

The compareTo() method is used in conditions where a Programmer wants to sort a list of strings in a predetermined order. The compareTo() method checks whether the string passed as an argument to the method is less than, greater than, or equal to the invoking string. A string is considered less than another string if it comes before it in alphabetical order.

The signature of the compareTo() method is as follows:

int i=str1. compareTo(str2)

where, str is the String being compared to the invoking String. The compareTo() method returns an int value as the result of String comparison.

compareToIgnoreCase()

The String class also has the compareToIgnoreCase() method that compares two strings without taking into consideration their case difference. The signature of the method is given below:

int i=str1.compareToIgnoreCase(str2)

regionMatches()

The regionMatches() method is used to check the equality of two string regions where the two string regions belong to two different strings. The signature of the method is given below:

public boolean regionMatches(int startindex, String str2, int startindex2, int len)

There is also an overloaded version of the method that tests the equality of the substring ignoring the case of characters in the substring. Its signature is given below:

public boolean regionMatches(boolean ignoreCase, int startindex, String str2, int startindex2, int len)

In both signatures of the method, startindex specifies the starting index of the substring within the invoking string. The str2 argument specifies the string to be compared. The startindex2 specifies the starting index of the substring within the string to be compared. The len argument specifies the length of the substring being compared. However, in the latter signature of the method, the comparison is done ignoring the case of the characters in the substring only if the ignoreCase argument is true.

startsWith()

The startsWith() method is used to check whether the invoking string starts with the same sequence of characters as the substring passed as an argument to the method. The signature of the method is given below:
public boolean startsWith(String prefix)
There is also an overloaded version of the startsWith() method with the following signature:
public boolean startsWith(String prefix, int startindex)

In both signatures of the method given above, the prefix denotes the substring to be matched within the invoking string. However, in the second version, the startindex denotes the starting index into the invoking string at which the search operation will commence.

endsWith()

The endsWith() method is used to check whether the invoking string ends with the same sequence of characters as the substring passed as an argument to the method. The signature of the method is given below:

public boolean endsWith(String prefix)

Modifying a String

The String objects are immutable. Therefore, it is not possible to change the original contents of a string. However, the following String methods can be used to create a new copy of the string with the required modification:

substring()

The substring() method creates a new string that is the substring of the string that invokes the method. The method has two forms:
public String substring(int startindex)
public String substring(int startindex, int endindex)

where, startindex specifies the index at which the substring will begin and endindex specifies the index at which the substring will end. In the first form where the endindex is not present, the substring begins at startindex and runs till the end of the invoking string.

Concat()

The concat() method creates a new string after concatenating the argument string to the end of the invoking string. The signature of the method is given below:

public String concat(String str)

replace()

The replace() method creates a new string after replacing all the occurrences of a particular character in the string with another character. The string that invokes this method remains unchanged. The general form of the method is given below:

public String replace(char old_char, char new_char)

trim()

The trim() method creates a new copy of the string after removing any leading and trailing whitespace. The signature of the method is given below:

public String trim(String str)

toUpperCase()

The toUpperCase() method creates a new copy of a string after converting all the lowercase letters in the invoking string to uppercase. The signature of the method is given below:

public String toUpperCase()

toLowerCase()

The toLowerCase() method creates a new copy of a string after converting all the uppercase letters in the invoking string to lowercase. The signature of the method is given below:

public String toLowerCase()

Searching Strings

The String class defines two methods that facilitate in searching a particular character or sequence of characters in a string. They are as follows:

IndexOf()

The indexOf() method searches for the first occurrence of a character or a

substring in the invoking string. If a match is found, then the method returns the index at which the character or the substring first appears. Otherwise, it returns -1.
The indexOf() method has the following signatures:

public int indexOf(int ch)
public int indexOf(int ch, int startindex)
public int indexOf(String str)
public int indexOf(String str, int startindex)

lastIndexOf()

The lastIndexOf() method searches for the last occurrence of a character or a substring in the invoking string. If a match is found, then the method returns the index at which the character or the substring last appears. Otherwise, it returns –1.

The lastIndexOf() method has the following signatures:

public int lastIndexOf(int ch)
public int lastIndexOf (int ch, int startindex)
public int lastIndexOf (String str)
public int lastIndexOf (String str, int startindex)

Program : Write a program using some important methods of String class.

```
// program using String class methods

class StrOps

{   public static void main(String args [])

{  String str1 = "When it comes to Web programming, Java is #1.";

String str2 = new String (str1);

String str3 = "Java strings are powerful.";

int result, idx;    char ch;

System.out.println ("Length of str1: " + str1.length ());

 // display str1, one char at a time.

for(int i=0; i < str1.length(); i++)

System.out.print (str1.charAt (i));
```

```
System.out.println ();
if (str1.equals (str2) )
System.out.println ("str1 equals str2");
else
System.out.println ("str1 does not equal str2");
if (str1.equals (str3) )
 System.out.println ("str1 equals str3");
else
System.out.println ("str1 does not equal str3");
result = str1.compareTo (str3);
if(result == 0)
System.out.println ("str1 and str3 are equal");
else if(result < 0)
System.out.println ("str1 is less than str3");
else
System.out.println ("str1 is greater than str3");
str2 = "One Two Three One";     // assign a new string to str2
idx = str2.indexOf ("One");
System.out.println ("Index of first occurrence of One: " + idx);
idx = str2.lastIndexOf("One");
System.out.println ("Index of last occurrence of One: " + idx);
}
}
```

Output:

```
D:/kumar>javac StrOPs.java
D:/kumar>java StrOps
```

Length of str1:46

When it comes to web programming, Java is #1.

Str1 equals str2

Str1 does not equal str3

Str1 is greater than str3

Index of first occurrence of one:0

Index of last occurrence of one:14

D:/kumar>

StringBuffer:

StringBuffer:

StringBuffer objects are mutable, so they can be modified. The methods that directly manipulate data of the object are available in StringBuffer class.

Creating StringBuffer:

- We can create a StringBuffer object by using new operator and pass the string to the object, as: StringBuffer sb = new StringBuffer ("Kiran");
- We can create a StringBuffer object by first allotting memory to the StringBuffer object using new operator and later storing the String into it as:

StringBuffer sb = new StringBuffer (30);

In general a StringBuffer object will be created with a default capacity of 16 characters. Here, StringBuffer object is created as an empty object with a capacity for storing 30 characters. Even if we declare the capacity as 30, it is possible to store more than 30 characters into StringBuffer.

To store characters, we can use append () method as:

Sb.append ("Kiran");

This represents growable and writeable character sequence. It is mutable in nature. StringBuffer are safe to be used by multiple thread as they are synchronized but this brings performance penalty.
It defines 3-constructor:
• StringBuffer(); //initial capacity of 16 characters

- StringBuffer(int size); //The initial size
- StringBuffer(String str);

StringBuffer str = new StringBuffer ("Stanford ");
str.append("Lost!!");

StringBuffer Class Methods:

Method	Description
StringBuffer append (x)	x may be int, float, double, char, String or StringBuffer. It will be appended to calling StringBuffer
StringBuffer insert (int offset, x)	x may be int, float, double, char, String or StringBuffer. It will be inserted into the StringBuffer at offset.
StringBuffer delete (int start, int end)	Removes characters from start to end
StringBuffer reverse ()	Reverses character sequence in the StringBuffer
String toString ()	Converts StringBuffer into a String
int length ()	Returns length of the StringBuffer

Program : Write a program using some important methods of StringBuffer class.

```
// program using StringBuffer class methods

import java.io.*;

class Mutable

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

  {  // to accept data from keyboard

  BufferedReader br=new BufferedReader (new InputStreamReader (System.in));

    System.out.print ("Enter sur name : ");

    String sur=br.readLine ( );

    System.out.print ("Enter mid name : ");

    String mid=br.readLine ( );

    System.out.print ("Enter last name : ");

    String last=br.readLine ( );

    // create String Buffer object

    StringBuffer sb=new StringBuffer ( );
```

```
    // append sur, last to sb
    sb.append (sur);
    sb.append (last);
    // insert mid after sur
    int n=sur.length ( );
    sb.insert (n, mid);
    // display full name
    System.out.println ("Full name = "+sb);
    System.out.println ("In reverse ="+sb.reverse ( ));   }}
```

Output**:**

```
D:/kumar>javac Mutable.java
D:/kumar>java Mutable
Enter sur name:kranthi
Enter mid name:kumar
Enter last name:A
Full name=kranthi kumar A
In reverse=A ramuk ihtnark
```

Accessor methods:

capacity(), charAt(i), length(), substring(iStart [,iEndIndex])

Modifier methods:

append(g), delete(i1, i2), deleteCharAt(i), ensureCapacity(), getChars(srcBeg, srcEnd, target[], targetBeg), insert(iPosn, g), replace(i1,i2,gvalue), reverse(), setCharAt(iposn, c), setLength(),toString(g)

So the basic differences are…..

1. String is immutable but StringBuffer is not.
2. String is not threadsafe but StringBuffer is thread safe
3. String has concat() for append character but StringBuffer has append() method
4. while you create String like String str = new String(); it create 2 object

1 on heap and 1 on String Constant pool and that refered by str but in StringBuffer it Create 1 object on heap

StringBuilder

StringBuilder class is introduced in Java 5.0 version. This class is an alternative to the existing StringBuffer class. If you look into the operations of the both the classes, there is no difference. The only difference between StringBuilder and StringBuffer is that StringBuilder class is not synchronized so it gives better performance. Whenever there are no threading issues, its preferable to use StringBuilder. StringBuffer class can be replaced by StringBuilder with a simple search and replace with no compilation issue.

Accessor methods: capacity(), length(), charAt(i), indexOf(g), lastIndexOf(g)

Modifier methods: append(g), delete(i1, i2), insert(iPosn, g), getChars(i), setCharAt(iposn, c), substring(), replace(i1,i2,gvalue), reverse(), trimToSize(g), toString(g)

Pointers

If you are an experienced C/C++ programmer, then you know that these languages provide support for pointers. Java does not support or allow pointers. Java cannot allow pointers, because doing so would allow Java programs to breach the firewall between the Java execution environment and the host computer. Since C/C++ make extensive use of pointers, you might be thinking that their loss is a significant disadvantage to Java. However, this is not true. Java is designed in such a way that as long as you stay within the confines of the execution environment, you will never need to use a pointer, nor would there be any benefit in using one