UNIT - I

Object Oriented Concepts

&

Java

Object Oriented Programming using Java

- The History and Evolution
- Overview and basic concepts
- Features, Advantages and Applications
- Java Development Kit (JDK) and JVM
- Simple Java programs
- Data types and Variables, dynamic initialization , the scope and lifetime of variables
- Type conversion and casting
- Operators and Expressions: Operator Precedence, Logical expression, Access specifiers
- Control statements & Loops and
- Arrays

Object Oriented Programming using Java

The History and Evolution:

- Java is a blend of the best elements of its rich heritage combined with the innovative concepts required by its unique mission.
- Although Java has become inseparably linked with the online environment of the Internet, it is important to remember that Java is first and foremost a programming language.
- Computer language innovation and development occurs for two fundamental reasons:
  - To adapt to changing environments and uses
  - To implement refinements and improvements in the art of programming

Object Oriented Programming using Java

The History and Evolution: ...

- 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.
  - Object-oriented features were influenced by C++.

By the end of the 1980s and the early 1990s, object-oriented programming using C++ took hold. Indeed, for a brief moment it seemed as if programmers had finally found the perfect language. Because C++ blended the high efficiency and stylistic elements of C with the object-oriented paradigm, it was a language that could be used to create a wide range of programs.

Object Oriented Programming using Java

The History and Evolution: ...

- However, just as in the past, forces were brewing that would, once again, drive computer language evolution forward. ***Within a few years, the World Wide Web and the Internet would reach critical mass. This event would bring about abruptly another revolution in programming.***
- Java was conceived by ***James Gosling***, Patrick Naughton, Chris Warth, Ed Frank, and Mike Sheridan ***at Sun Microsystems, Inc. in 1991***. It was initially called "Oak" but was renamed "Java" in 1995.
- Bill Joy, Arthur van Hoff, Jonathan Payne, Frank Yellin, and Tim Lindholm were key contributors to the maturing of the original prototype.

Object Oriented Programming using Java

The History and Evolution: ...

- Original impetus for Java was not the Internet! Instead, the primary motivation was the need for a platform-independent (i.e., architecture-neutral) language that could be used to create software to be embedded in various consumer electronic devices, such as microwave ovens and remote controls.
- 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.
- 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.

Object Oriented Programming using Java

The History and Evolution: ...

- Today, with technology such a part of our daily lives, we take it for granted that we can be connected and access applications and content anywhere, anytime. Because of Java, we expect digital devices to be smarter, more functional, and way more entertaining.
- Today, Java not only permeates the Internet, but also is the invisible force behind many of the applications and devices that power our day-to-day lives.
- From mobile phones to handheld devices, games and navigation systems to e-business solutions, ***Java is everywhere!***

Object Oriented Programming using Java

The History and Evolution: ...

- The first version of Java Development Kit, The JDK 1.0 was released on January 23, 1996. From the few hundred class files the JDK 1.0 had, it has now grown dramatically into more than 3000 classes in J2SE 5.0, J2SE6, . . .
- Now Java is extensively used to program stand alone applications to Multi tier web applications.
  - ***JDK 1.0 (January 23, 1996)***
  - ***JDK 1.1 (February 19, 1997)***
  - ***J2SE 1.2 (December 8, 1998)***
  - ***J2SE 1.3 (May 8, 2000)***
  - ***J2SE 1.4 (February 6, 2002)***
  - ***J2SE 5.0 (September 30, 2004)***
  - ***Java SE 6 (December 11, 2006)***
  - ***Java SE 7 (July 28, 2011)***
  - ***Java SE 8 (March 18, 2014)***

Object Oriented Programming using Java

The History and Evolution: ...*Brief Summary*

- In 1990, Sun Microsystems initiated a team to develop software for consumer electronics devices headed by ***James Gosling.***
- In 1991, the team announced the ***"Oak"*** from C++
- In 1992, GreenProject team by Sun demonstrated the application to home appliances.
- In 1993, Transformation of text-based internet into a graphical-rich environment. (Applets)
- In 1994, HotJava – web browser to locate & run applets
- In 1995, Oak renamed as Java.
- In 1996, Leader as General Purpose Programming Language / OOPL.

Object Oriented Programming using Java

Overview and Basic Concepts:

- Originally developed by Sun Microsystems, initiated by James Gosling. With the advancement of Java and its widespread popularity, multiple configurations were built to suite various types of platforms. ***Ex:*** J2EE for Enterprise Applications, J2ME for Mobile Applications. Sun Microsystems has renamed the new J2 versions as Java SE, Java EE and Java ME, respectively.
- Java is guaranteed to be ***Write Once, Run Anywhere***.
- Java is easy to learn. Java was designed to be easy to use and is therefore easy to write, compile, debug, and learn than other programming languages.
- Java is object-oriented. This allows you to create modular programs and reusable code.
- Java is platform-independent.

Object Oriented Programming using Java

Features: - *The Java Buzzwords*

The key considerations were summed up by the Java team in the following list of buzzwords:

Simple, Secure, Portable, Object-oriented, Robust, Multithreaded, Architecture-neutral, Interpreted, High performance, Distributed and Dynamic

Object Oriented Programming using Java

Features: - *The Java Buzzwords...*

- **Simple**: Java was designed to be easy for the professional programmer to learn and use effectively. It contains many features of other Languages like c and C++ and Java Removes Complexity because it doesn't use pointers.
- **Secure:** Java achieved protection by confining an applet to the Java execution environment and not allowing it access to other parts of the computer.
- **Portable:** The same code must work on all computers. Being architectural-neutral and having no implementation dependent aspects of the specification makes Java portable.
- **Object-oriented**: In Java, everything is an Object. Java can be easily extended since it is based on the Object model.

Object Oriented Programming using Java

Features: - *The Java Buzzwords...*

- **Robust:** Java makes an effort to eliminate error prone situations by emphasizing mainly on compile time error checking and runtime checking. Also auto garbage collection mechanism.
- **Multithreaded:** Enables us to write programs that can do many tasks simultaneously/concurrently. This design feature allows developers to construct smoothly running interactive applications.
- **Architecture-neutral**: A central issue for the Java designers was that of code longevity and portability. It follows 'Write-once-run-anywhere' ***WORA*** approach. To a great extent, this goal was accomplished.
- **Interpreted**: Java enables the creation of cross-platform programs by compiling into an intermediate representation called Java bytecode. This code can be executed on any system that implements the Java Virtual Machine (JVM).

Object Oriented Programming using Java

Features: - *The Java Buzzwords...*

- **High performance:** Java bytecode was carefully designed so that it would be easy to translate directly into native machine code for very high performance by using a just-in-time (JIT) compiler.
- **Distributed:** Java is designed for the distributed environment of the Internet because it handles TCP/IP protocols. Also supports Remote Method Invocation (RMI) feature it enables a program to invoke methods across a network.
- **Dynamic:** It is designed to adapt to an evolving environment. Java programs can carry extensive amount of run-time information that can be used to verify and resolve accesses to objects on run-time.

Object Oriented Programming using Java

Advantages:

- ***Write Once, Run Anywhere*** - You only have to write your application once--for the Java platform--and then you'll be able to run it anywhere. Java support is becoming ubiquitous.
- ***Security*** **-** Allows users to download untrusted code over a network and run it in a secure environment in which it cannot do any harm: it cannot infect the host system with a virus, cannot read or write files from the hard drive, and so forth. This capability alone makes the Java platform unique.
- ***Network-centric Programming*** **-** Java makes it unbelievably easy to work with resources across a network and to create network-based applications using client/server or multitier architectures. ***"The network is the computer"***

Object Oriented Programming using Java

Advantages: ...

- ***Dynamic, Extensible Programs*** **-** Java application can dynamically extend itself by loading new classes over a network.
- ***Internationalization*** **-** Java uses 16-bit Unicode characters that represent the phonetic alphabets and ideographic character sets of the entire world.
- ***Performance*** **-** The VM has been highly tuned and optimized in many significant ways. Using sophisticated JIT compilers, Java programs can execute at speeds comparable to the speeds of native C and C++ applications.
- ***Programmer Efficiency and Time-to-Market*** **-** Java is an elegant language combined with a powerful and well-designed set of APIs.

Object Oriented Programming using Java

Applications:

- ✓ Standalone/Desktop/Console –based Applications
  - Stand alone CUI/GUI based applications
- ✓ Web Applications
  - Applets and Servlets
- ✓ Distributed Applications
  - Database applications
- ✓ Client/Server Applications

Object Oriented Programming using Java

DIFFERENCES BETWEEN C , C++ AND JAVA

- C Uses header Files but java uses Packages
- C Uses Pointers but java doesn't supports pointers .
- Java doesn't supports storage classes like auto, external etc.
- The Code of C Language is Converted into the Machine code after Compilation But in Java Code First Converted into the Bytes Codes then after it is converted into the Machine Code.
- C++ supports Operator Overloading but java doesn't Supports Operator Overloading .
- In C++ Multiple Inheritance is Possible but in java A Class Can not Inherit the features from the two classes in other words java doesn't supports Multiple Inheritance The Concept of Multiple Inheritances is Introduced in the Form of Interfaces.
- Java Uses import statement for including the contents of screen instead of #include.
- Java Doesn't uses goto.
- Java Doesn't have Destructor like C++ Instead Java Has finalize Method.
- Java Doesn't have Structure Union , enum data types.

Object Oriented Programming using Java

DIFFERENCES BETWEEN C++ and JAVA

C++ and Java both are Object Oriented Languages but some of the features of both languages are different from each other. Some of these features are:

- All stand-alone C++ programs require a function named main and can have numerous other functions, including both stand-alone functions and functions, which are members of a class. There are no stand-alone functions in Java. Instead, there are only functions that are members of a class, usually called methods. Global functions and global data are not allowed in Java.
- All classes in Java ultimately inherit from the Object class. This is significantly different from C++ where it is possible to create inheritance trees that are completely unrelated to one another.
- Java does not support multiple inheritance. To some extent, the interface feature provides the desirable features of multiple inheritance to a Java program without some of the underlying problems.
- Java does not support operator overloading.

Object Oriented Programming using Java

Java Runtime Environment (JRE):

- The smallest set of executables and files that constitute the standard java plotform.
- It provides the libraries, the JVM, and other components to run applets and applications written in the Java programming language.
- In addition, two key deployment technologies are part of the JRE: ***Java Plug-in***, which enables applets to run in popular browsers; and ***Java Web Start***, which deploys standalone applications over a network.

Object Oriented Programming using Java

Java Development Kit (JDK):

- The Java Development Kit (JDK) is a package that includes a large number of development tools and hundreds of classes, Java Standard library APIs and methods.
- The JDK is developed by Sun Microsystem's. it contain JRE and development tools.
- JDK provides tools for users to integrate and execute applications and Applets.
- **Eg.** javac, java, javap, jdb, javah, javadoc, appletviwer

Object Oriented Programming using Java

Java Development Kit (JDK): ...

- **javac -** The Java Compiler, it compiles Java source code into Java bytecodes.

  javac filename.java

- **java -** The Java Interpreter, it executes Java class files created by a Java compiler.

  java classfilename

- **javap -** The Java Class File Disassembler, Disassembles class files. Its output depends on the options used.

  javap [options] classfilename

Object Oriented Programming using Java

Java Development Kit (JDK): ...

- **jdb -** The Java Debugger, helps you find and fix bugs in Java language programs.

  jdb [options]

- **javah -** C Header and Stub File Generator, produces C header files and C source files from a Java class. These files provide the connective glue that allow your Java and C code to interact.

  javah [options] classfilename

Object Oriented Programming using Java

Java Development Kit (JDK): ...

- **javadoc -** The Java API Documentation Generator Generates HTML pages of API documentation from Java source files, parses the declarations and documentation comments in a set of Java source files and produces a set of HTML pages.

  javadoc [options] [package | source.java]*

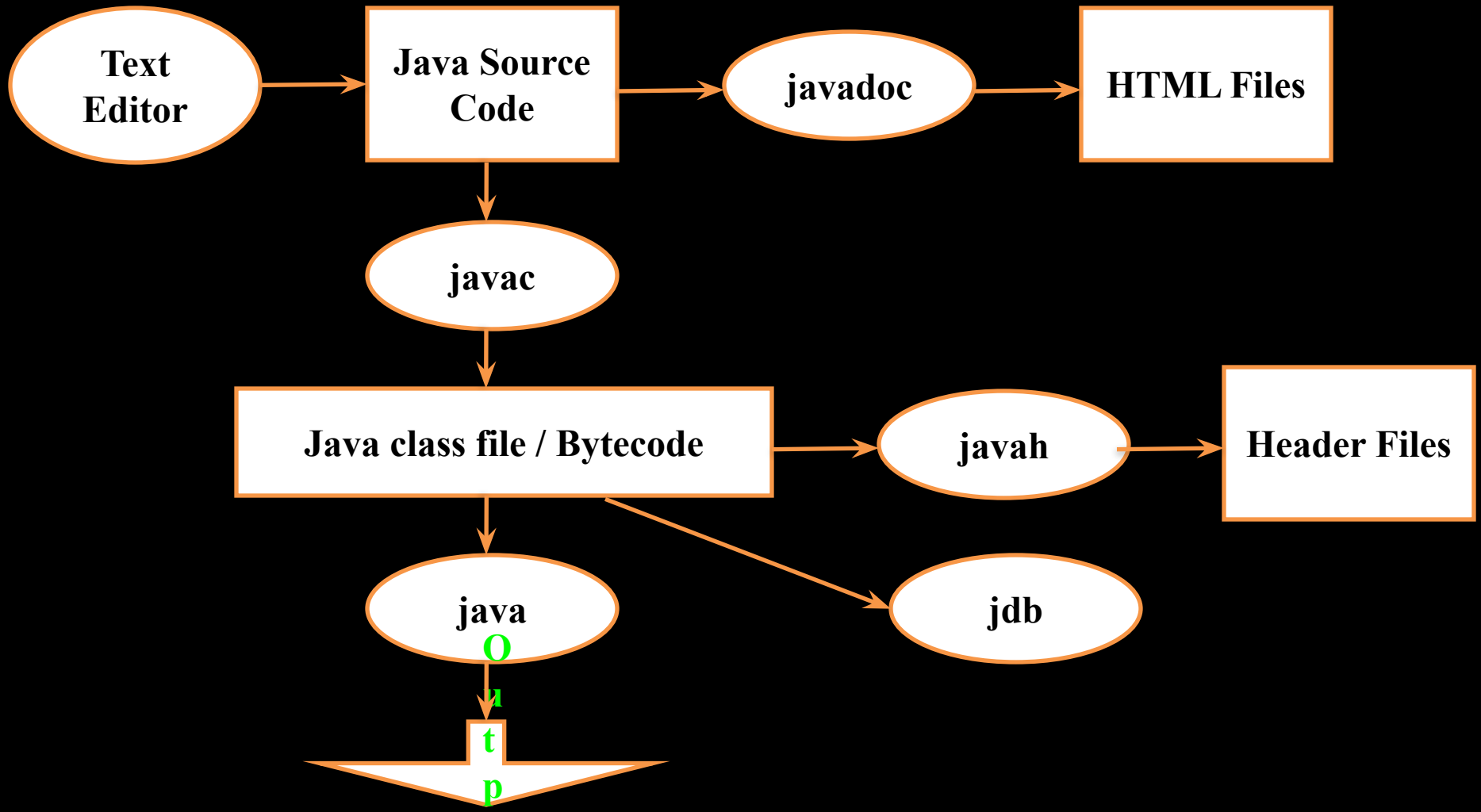
- **appletviewer -** The appletviewer command allows you to run applets outside of a web browser.

  appletviewer [options] filename.html [URLs]

Object Oriented Programming using Java

Java Development Kit (JDK): ...

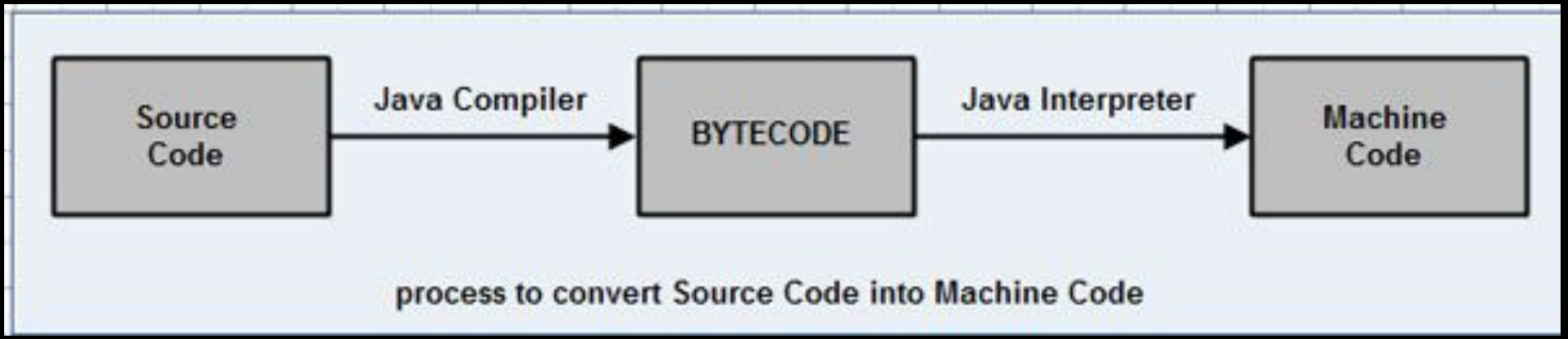
The Java Programming Environment



Object Oriented Programming using Java

Java Virtual Machine(JVM): - 'Simulated computer within the computer'

- JVM is the virtual machine that run the Java byte code. It's also the entity that allows Java to be a "portable language" (write once, run anywhere). Indeed there are specific implementations of the JVM for different systems (Windows, Linux, MacOS,..), the aim is that with the same byte code they all give the same results (***i.e. JVM is platform dependent***).



Object Oriented Programming using Java

Basic Structure of Java programs:

Documentation Section
Package Statements
Import Statements
Interface Statements
Class Definitions

main method class{
 	
//Definitions
 	
}

Object Oriented Programming using Java

Simple Java programs:

```
/*
     This is a simple Java program.
     Call this file "Example.java".
*/
class Example {
    // Your program begins with a call to main().
    public static void main(String args[]) {
     System.out.println("This is a simple Java program.");
    }
}
```

Object Oriented Programming using Java

Simple Java programs: ...

Entering the Program:

- For most computer languages, the name of the file that holds the source code to a program is immaterial. For this example, the name of the source file should be Example.java.
- In Java, a source file is officially called a compilation unit. It is a text file that contains one or more class definitions. The Java compiler requires that a source file use the .java filename extension.
- ***In Java, all code must reside inside a class. By convention, the name of that class should match the name of the file that holds the program.***
- You should also make sure that the capitalization of the filename matches the class name.
- The Java is case-sensitive. The convention that filenames correspond to

Object Oriented Programming using Java

Simple Java programs: ...

Compiling the Program:

- To compile the Example program, execute the compiler, **javac**, specifying the name of the source file on the command line, as shown here:

C:\>javac Example.java

- *The **javac** compiler creates a file called Example.class that contains the bytecode version of the program. The Java bytecode is the intermediate representation of your program that contains instructions the Java Virtual Machine will execute. Thus, the output of **javac** is not code that can be directly executed.*
- To run the program, you must use the Java application launcher, called **java**

Simple Java programs: ...

Compiling the Program: ...

- When the program is run, the following output is displayed:

  This is a simple Java program.

- ***When Java source code is compiled, each individual class is put into its own output file named after the class and using the .class extension.*** This is why it is a good idea to give your Java source files the same name as the class they contain-the name of the source file will match the name of the .class file. When you execute java as just shown, you are actually specifying the name of the class that you want to execute. It will automatically search for a file by that name that has the .class extension. If it finds the file, it will execute the code contained in the specified class.

Object Oriented Programming using Java

Simple Java programs: ...

A Closer Look at the First Sample Program:

Example.java includes several key features that are common to all Java programs.
The program begins with the following lines:

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

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

Object Oriented Programming using Java

Simple Java programs: ...

A Closer Look at the First Sample Program: ...

- Java supports three styles of comments.
- // Single line comment
- /* *and end with* */ A multiline comment
- /** *documentation* */ called doc comment, the JDK **javadoc** tool uses doc comments when preparing automatically generated documentation.
- Any comments are ignored by the compiler.

- **class Example {**
- This line uses the keyword **class** to declare that a new class is being defined. The entire class definition, including all of its members, will be between the opening curly brace ({) and the closing curly brace

Object Oriented Programming using Java

Simple Java programs: ...

A Closer Look at the First Sample Program: ...

public static void main(String args[]) {

- The **main( )** method, at which the program will begin executing. All Java applications begin execution by calling main( ).

- The **public** keyword is an access specifier, which allows the programmer to control the visibility of class members. When a class member is preceded by **public**, then that member may be accessed by code outside the class in which it is declared.
- The opposite of **public** is **private**, which prevents a member from being used by code defined outside of its class.

Object Oriented Programming using Java

Simple Java programs: ...

A Closer Look at the First Sample Program: ...

public static void main(String args[]) {

- In this case, **main( )** must be declared as **public**, since it must be called by code outside of its class when the program is started.
- The keyword **static** allows main( ) to be called without having to instantiate a particular instance of the class. This is necessary since main( ) is called by the JVM before any objects are made.
- The keyword **void** simply tells the compiler that **main( )** does not return a value.

Object Oriented Programming using Java

Simple Java programs: ...

```
/*
   Here is another short example.
   Call this file "Example2.java".
*/
class Example2 {
   public static void main(String args[]) {
      int num; // this declares a variable called num
      num = 100; // this assigns num the value 100
      System.out.println("This is num: " + num);
      num = num * 2;
      System.out.print("The value of num * 2 is ");
      System.out.println(num);
   }
}
```

Object Oriented Programming using Java

Two Control Statements:

- **The if Statement:** 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.

  Example: if(num < 100) System.out.println("num is less than 100");

- Java defines a full complement of relational operators which may be used in a conditional expression. Here are a few: **>, < and ==**

Object Oriented Programming using Java

Two Control Statements:

- **The if Statement: ...**

```
/*
  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");

    x = x * 2;
    if(x > y) System.out.println("x now greater than y");

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

Object Oriented Programming using Java

Two Control Statements: ...

- **The for Loop:** Java supplies a powerful assortment of loop constructs. Perhaps the most versatile is 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 con

```
/*
  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);
  }
}
```

Object Oriented Programming using Java

Two Control Statements: ...

- **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, it becomes a logical unit that can be used any place that a single statement can.
- **Eg:**

```
if(x < y) { // begin a block
  x = y;
  y = 0;
} // end of block
```

Two Control Statements: ...

- Using Blocks of Code: ...

```
/*
  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;
    }
  }
}
```

Object Oriented Programming using Java

Lexical Issues :

- Java programs are a collection of whitespace, identifiers, literals, comments, operators, separators, and keywords - *The atomic elements of Java*
- **Whitespace:** Java is a free-form language. In Java, whitespace is a space, tab, or newline.
- **Identifiers:** Used for class names, method names, and variable names. An identifier may be any descriptive sequence of uppercase and lowercase letters, numbers, or the underscore and dollar-sign characters. They must not begin with a number, again, Java is case-sensitive.

Object Oriented Programming using Java

Lexical Issues : ...

- **Identifiers:...**
  - ✔ AvgTemp, count, a4, $test, this_is_ok are Valid
  - ✔ 2count, high-temp, Not/ok are Invalid
- **Literals:** A constant value in Java is created by using a literal representation of it. It can be used anywhere a value of its type is allowed.
  - ✔ 100, 98.6, 'X', "This is a test"
- **Comments:** As mentioned, there are three types of comments defined by Java.

Object Oriented Programming using Java

Lexical Issues : ...

- **Separators:**

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.

Object Oriented Programming using Java

Lexical Issues : ...

- **The Java Keywords:** 50 keywords currently defined, these combined with the syntax of the operators and separators, form the foundation of the Java language. These keywords cannot be used as names for a variable, class, or method.
- The keywords **const** and **goto** are reserved but not used. In the early days of Java, several other keywords were reserved for possible future use.
- In addition to the keywords, Java reserves the following: **true, false,** and **null**. These are values defined by Java.
- You may not use these words for the names of variables, classes, and so on.

Object Oriented Programming using Java

Lexical Issues : ...

- **The Java Keywords: ...**

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

Object Oriented Programming using Java

- The History and Evolution
- Overview and basic concepts
- Features, Advantages and Applications
- Java Development Kit (JDK) and JVM
- Simple Java programs
- Data types and Variables, dynamic initialization , the scope and lifetime of variables
- Type conversion and casting
- Operators and Expressions: Operator Precedence, Logical expression, Access specifiers
- Control statements & Loops and
- Arrays

Object Oriented Programming using Java

Data types and Variables

- The Java is a strongly typed language, part of Java's safety and robustness.
- First, every variable has a type, every expression has a type, and every type is strictly defined.
- Second, all assignments, whether explicit or via parameter passing in method calls, are checked for type compatibility.
- There are no automatic coercions or conversions of conflicting types as in some languages.
- The Java compiler checks all expressions and parameters to ensure that the types are compatible. Any type mismatches are errors that must be corrected before the

Object Oriented Programming using Java

Data types and Variables ...

The domains which determine what type of contents can be stored in a variable. In Java, there are two types of data types:

- ***Primitive /Simple data types:*** Defines eight types of data: *byte, short, int, long, char, float, double, and boolean.*
- ***Reference data types: or Abstaract dataypes***

  arrays, objects, interfaces, enum, Srting etc.

Object Oriented Programming using Java

Data types and Variables ...

Type	Length	Min. Value	Max. Value
byte	8 bits	-128	127
short	16 bits	-32768	32767
int	32 bits	-2,147,483,648	2,147,483,647
long	64 bits	-9,223,372,036,854,775,808	9,223,372,036,854,775,80
float	32 bits	-3.4E+38	+3.4E+38
double	64 bits	-1.7E+308	+1.7E+308
boolean	1 bit	Possible values are ***true*** or ***false***	
char	16 bits	Unicode / International character set	

Object Oriented Programming using Java

Data types and Variables:...

- **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. **Eg:** ***int num1, num2,...***
- However, the concept of unsigned was used 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 by adding a special *"unsigned right shift"* operator. Thus, the need for an unsigned integer type was eliminated.

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

Object Oriented Programming using Java

Data types and Variables:...

- **byte:** The smallest integer type, a signed 8-bit type, has a range from –128 to 127.
- Useful when you're working with a stream of data from a network or file. **Eg:** ***byte Byte1, Byte2,...***
- **short:** A signed 16-bit type, has a range from –32,768 to 32,767. It is probably the least-used type. **Eg:** ***short s1, s2;***
- **int:** Most commonly used integer type, a signed 32-bit type that has a range from –2,147,483,648 to 2,147,483,647.
- Variables of type *int* are commonly employed to control loops and to index arrays. *byte* and *short* values are used in an expression are promoted to *int* when the expression is

Object Oriented Programming using Java

Data types and Variables:...

- **long:** A signed 64-bit type, useful for those occasions where an **int** type is not large enough to hold the desired value.
- This makes it useful when big, whole numbers are

```
// 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.
```

Object Oriented Programming using Java

Data types and Variables:...

- **Floating-Point Types:** Also known as real numbers, are used when evaluating expressions that require fractional precision. There are two kinds of floating-point types, float and double, which represent single- and double-precision numbers.
- **float**: Specifies a single-precision value that uses 32 bits of storage, Single precision is faster on some processors, Useful when you need a fractional component, but don't require a large degree of precision. **Eg:** float hightemp, lowtemp;

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

Object Oriented Programming using Java

Data types and Variables:...

- **double**: 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.

- **char**: A 16-bit type, the range of a char is 0 to 65,536. 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.

Object Oriented Programming using Java

Data types and Variables:...

- **char:...** An *Eg*

```
// 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);
    }
}
```

- **boolean:** a primitive type, for logical values. It can have only one of two possible values, true or false. This is the type returned by all relational operators.

Object Oriented Programming using Java

Data types and Variables:...

- **boolean:...**

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
```

Object Oriented Programming using Java

Data types and Variables:...

A Closer Look at Literals

- **Integer Literals:** The most commonly used type in the typical program. Any whole number value is an integer literal. Eg: 1, 2, 3, and 42. all decimal values, 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.
- 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.

Object Oriented Programming using Java

Data types and Variables:...

A Closer Look at Literals...

- **Integer Literals:...**
- Integer literals create an **int** value, which in Java is a 32-bit integer value.
- 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.
- When a literal value is assigned to a **byte** or **short** variable, no error is generated if the literal value is within the range of the target type.
- An integer literal can always be assigned to a **long** variable. 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,

Object Oriented Programming using Java

Data types and Variables:...

A Closer Look at Literals...

- **Floating-Point Literals:** 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.
- 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.

Object Oriented Programming using Java

Data types and Variables:...

A Closer Look at Literals...

- **Boolean Literals:** Simple, only two logical values ***true*** and ***false***. These values 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:** Indices into the Unicode character set, 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***.

Object Oriented Programming using Java

Data types and Variables:...

A Closer Look at Literals...

- **String Literals:** Specified by enclosing a sequence of characters between a pair of double quotes. **Eg:** “Hello World” , “two\nlines” , “\”This is in quotes\”“

Escape Sequence	Description
\ddd	Octal character (ddd)
\uxxxx	Hexadecimal Unicode character (xxxx)
\'	Single quote
\"	Double quote
\\	Backslash
\r	Carriage return
\n	New line (also known as line feed)
\f	Form feed
\t	Tab
\b	Backspace

Object Oriented Programming using Java

Data types and Variables:...

- **Variables:** The basic unit of storage, 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.

The basic form of a variable declaration:

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

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

Dynamic initialization:

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

```
// 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);
    }
}
```

Note: 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:

- 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.

Object Oriented Programming using Java

The Scope and Lifetime of Variables:...

```
// 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);
  }
}
```

Object Oriented Programming using Java

The Scope and Lifetime of Variables:...

```
// 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);
    }
  }
}
```

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!
                }
        }
}
```

The Scope and Lifetime of Variables:...

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

Object Oriented Programming using Java

Type Conversion and Casting:

- 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.
- Fortunately, it is still possible to obtain a conversion between incompatible types. To do so, you must use a cast, which performs an explicit conversion between incompatible types.
- Let's look at both automatic type conversions and casting.

Type Conversion and Casting:...

Java's Automatic Conversions - *Widening Conversion*

- 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 widening conversions, the numeric types, including integer and floating-point types, are compatible with each other.
- Java also performs an automatic type conversion when storing a literal integer constant into variables of type byte, short, long, or char.
- ***However, there are no automatic conversions from the numeric types to char or boolean. Also, char and boolean are not compatible with***

Object Oriented Programming using Java

Type Conversion and Casting:...

Casting Incompatible Types - *Narrowing Conversion*

- To create a conversion between two incompatible types, you must use a cast. A *cast* is simply an explicit type conversion.

(target-type) value

- **Eg:**
```
int a;
byte b;
// ...
b = (byte) a;
```
- 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)

Object Oriented Programming using Java

Type Conversion and Casting:...

Casting Incompatible Types - *Narrowing Conversion…*

- A different type of conversion will occur when a floating-point value is assigned to an integer type: ***truncation*.**

Automatic Type Promotion in Expressions

```
byte a = 40;
byte b = 50;
byte c = 100;
int d = a * b / c;
```

- To handle this kind of problem, Java automatically promotes each ***byte, short,*** *or* ***char*** operand to **int** when evaluating an expression.
- As useful as the automatic promotions are, they can cause confusing compile-time errors. For example, this seemingly correct code causes a problem:

```
byte b = 50;
b = b * 2; // Error! Cannot assign an int to a byte!
```

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

Object Oriented Programming using Java

Type Conversion and Casting:...

Casting Incompatible Types - *Narrowing Conversion…*

```
// 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);
  }
}
```

output:
```
Conversion of int to byte.
i and b 257 1
Conversion of double to int.
d and i 323.142 323
Conversion of double to byte.
d and b 323.142 67
```

Type Conversion and Casting:...

The Type Promotion Rules

- Java defines several type promotion rules that apply to expressions.
- ***They are as follows:*** First, all **byte**, **short**, and **char** values are promoted to **int**, as just described. Then, if one operand is a **long**, the whole expression is promoted to **long**. If one operand is a **float**, the entire expression is promoted to **float**. If any of the operands is **double**, the result is **double**.

```
class Promote {
  public static void main(String args[]) {
    byte b = 42;
    char c = 'a';
    short s = 1024;
    int i = 50000;
    float f = 5.67f;
    double d = .1234;
    double result = (f * b) + (i / c) - (d * s);
    System.out.println((f * b) + " + " + (i / c) + " - " + (d * s));
    System.out.println("result = " + result);
  }
}
```

Object Oriented Programming using Java

Operators and Expressions:

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

Classification of Operators

(1) Arithmetic Operators
(2) Relational Operators
(3) Logical Operators
(4) Assignment Operators
(5) Increments and Decrement Operators
(6) Conditional Operators
(7) Bitwise Operators
(8) Special Operators

Object Oriented Programming using Java

Operators and Expressions:...

- **Arithmetic Operators:**

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

The Basic Arithmetic Operators: addition, subtraction, multiplication, and division

The Modulus Operator: It can be applied to floating-point types as well as integer types.

Arithmetic Compound Assignment Operators: Used to combine an arithmetic operation with an assignment.

var = var op expression; can be written as var op= expression;

Increment and Decrement: ++ and --

Operators and Expressions:...

- **The Bitwise Operators:** Java defines several bitwise operators that can be applied to the integer types, long, int, short, char, and byte. These operators act upon the individual bits of their operands.

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

The Bitwise Logical Operators: &, |, ^, and ~

The Bit Shift: <<, >> , >>>
value << num, value >> num,

Bitwise Operator Compound Assignments:

a = a >> 4;

a >>= 4;

Object Oriented Programming using Java

Operators and Expressions:...

- **The Bitwise Operators: *The Unsigned Right Shift***

The Left Shift: For each shift left, the high-order bit is shifted out (and lost), and a zero is brought in on the right., multiply by 2.

value << num

The Right Shift: Causes the two low-order bits to be lost, divide by 2.

value >> num

The Unsigned Right Shift/Shift Right Zero Fill: Always shifts zeros into the high-order bit. For example, if you are shifting something that does not represent a numeric value, you may not want sign extension to take place. ***This situation is common when you are working with pixel-based values and graphics.*** In these cases, you will generally want to shift a zero into the high-order bit no matter what its initial value was.

value >> num

int a = -1;
a = a >>> 24;
Here is the same operation in binary form to further illustrate what is happening:
11111111 11111111 11111111 11111111 –1 in binary as an int
>>>24
00000000 00000000 00000000 11111111 255 in binary as an int

Object Oriented Programming using Java

Operators and Expressions:...

Relational Operators: The outcome of these operations is a ***boolean*** value. Any type in Java, including integers, floating-point numbers, characters, and Booleans can be compared using the equality test, ==, and the inequality test, !=. Only integer, floating-point, and character operands compared to see which is greater or less than

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

Eg:

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

Operators and Expressions:...

- **Boolean Logical Operators:** operate only on boolean operands.

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

Object Oriented Programming using Java

Operators and Expressions:...

- **Short-Circuit Logical Operators:** **&&** and || secondary versions of the Boolean AND and OR operators (& and |).

- **The Assignment Operator:** ***var = expression;***

- **The ? Operator:** Java includes a special ternary (three-way) operator that can replace certain types of *if-then-else* statements. **expression1 ? expression2 : expression3**
  Eg: ratio = denom == 0 ? 0 : num / denom;

Object Oriented Programming using Java

Operator Precedence:

Category	Operator	Associativity
Postfix	() [] . (dot operator)	Left to right
Unary	++ - - ! ~	Right to left
Multiplicative	* / %	Left to right
Additive	+ -	Left to right
Shift	>> >>> <<	Left to right
Relational	> >= < <=	Left to right
Equality	== !=	Left to right
Bitwise AND	&	Left to right
Bitwise XOR	^	Left to right
Bitwise OR	|	Left to right
Logical AND	&&	Left to right
Logical OR	||	Left to right
Conditional	?:	Right to left
Assignment	= += -= *= /= %= >>= <<= &= ^= |=	Right to left
Comma	,	Left to right

Object Oriented Programming using Java

Access specifiers:

- Encapsulation links data with the code that manipulates it. However, it provides another important attribute: ***access control***.
- Through encapsulation, you can control what parts of a program can access the members of a class. By controlling access, you can prevent misuse.
- **Eg:** Allowing access to data only through a well defined set of methods, you can prevent the misuse of that data.
- How a member can be accessed is determined by the access specifier that modifies its declaration. Java supplies a rich set of access specifiers.

Object Oriented Programming using Java

Access specifiers: ...

- Java's access specifiers are ***public***, ***private***, ***protected*** and *default*.
- ***public:*** Member can be accessed by any other code.
- ***private:*** Member can only be accessed by other members of its class.
- ***protected:*** Applies only when inheritance is involved.
- ***When no access specifier is used, then by default the member of a class is public within its own package, but cannot be accessed outside of its package.***

Object Oriented Programming using Java

Control Statements & Loops:

- Programming language uses control statements to Cause the flow of execution to advance and branch based on changes to the state of a program.

Three Categories

- **Selection:** Allows program to choose different paths of execution based upon the outcome of an expression or the state of a variable. **Eg:** ***if*** *and* ***switch***
- **Iteration:** Enables program execution to repeat one or more statements (form loops). **Eg:** ***for, while*** *and* ***do-while***
- **Jump:** Allows program to execute in a nonlinear fashion. **Eg:** ***break, continue,*** *and* ***return.***

Object Oriented Programming using Java

Java's Selection Statements:

- ***The if statement:*** Java's conditional branch statement. It can be used to route program execution through two different paths.
- The general form: ***if (condition) statement1;***
  else statement2;
- Here, each *statement* may be a single statement or a compound statement enclosed in curly braces (that is, a block). The *condition* is any expression that returns a *boolean* value. The *else* clause is optional.

```
int a, b;
// ...
if(a < b) a = 0;
else b = 0;
```

```
boolean dataAvailable;
// ...
if (dataAvailable)
  ProcessData();
else
  waitForMoreData();
```

```
int bytesAvailable;
// ...
if (bytesAvailable > 0) {
     ProcessData();
     bytesAvailable -= n;
} else
     waitForMoreData();
```

Java's Selection Statements: ...

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

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

Object Oriented Programming using Java

Java's Selection Statements: ...

- ***The if-else-if Ladder:*** A common programming construct that is based upon a sequence of nested **ifs** is the ***if-else-if ladder***.

```
if(condition)
      statement;
else if(condition)
      statement;
else if(condition)
      statement;
.
.
.
else
      statement;
```

```
// Demonstrate if-else-if statements.
class IfElse {
  public static void main(String args[]) {
    int month = 4; // April
    String season;

    if(month == 12 || month == 1 || month == 2)
      season = "Winter";
    else if(month == 3 || month == 4 || month == 5)
      season = "Spring";
    else if(month == 6 || month == 7 || month == 8)
      season = "Summer";
    else if(month == 9 || month == 10 || month == 11)
      season = "Autumn";
    else
      season = "Bogus Month";

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

Java's Selection Statements: ...

- ***switch:*** Java's *multiway* branch statement, provides an easy way to dispatch execution to different parts of your code based on the value of an expression, provides a better alternative than a large series of ***if-else-if*** statements.
- ***The general form:***

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

The *expression must be of type byte, short, int, or char; each of the values specified in the* case statements must be of a type compatible with the expression. (An enumeration value can also be used to control a switch statement. Each case value must be a unique literal (that is, it must be a constant, not a variable). Duplicate case values are not allowed. the default statement is optional. If no case matches and no default is present, then no further action is taken.

Object Oriented Programming using Java

Java's Selection Statements: ...

- *switch:* ...

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

```
// In a switch, break statements are optional.
class MissingBreak {
  public static void main(String args[]) {
    for(int i=0; i<12; i++)
      switch(i) {
        case 0:
        case 1:
        case 2:
        case 3:
        case 4:
          System.out.println("i is less than 5");
          break;
        case 5:
        case 6:
        case 7:
        case 8:
        case 9:
          System.out.println("i is less than 10");
          break;
        default:
          System.out.println("i is 10 or more");
      }
  }
}
```

Java's Selection Statements: ...

- *switch: ...*

```
// An improved version of the season program.
class Switch {
    public static void main(String args[]) {
        int month = 4;
        String season;
        switch (month) {
          case 12:
          case 1:
          case 2:
            season = "Winter";
            break;
          case 3:
          case 4:
          case 5:
            season = "Spring";
            break;
          case 6:
          case 7:
          case 8:
            season = "Summer";
            break;
          case 9:
          case 10:
          case 11:
            season = "Autumn";
            break;
          default:
            season = "Bogus Month";
        }
        System.out.println("April is in the " + season + ".");
    }
}
```

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

```
switch(count) {
  case 1:
    switch(target) { // nested switch
      case 0:
        System.out.println("target is zero");
        break;
      case 1: // no conflicts with outer switch
        System.out.println("target is one");
        break;
    }
    break;
  case 2: // ...
```

Object Oriented Programming using Java

Java's Iteration Statements: ***for, while,*** and ***do-while,*** these statements create loops. A loop repeatedly executes the same set of instructions until a termination condition is met.

while	*do-while*	*for*
while(*condition*) { ** // body of loop** **}**	**do {** ** // body of loop** **} while (*condition*);**	**for(initialization; *condition*; iteration) {** ** // body** **}**

The *condition can be any Boolean expression. The body of the loop will be executed as long* as the conditional expression is true. When *condition becomes false, control passes to the* next line of code immediately following the loop. The curly braces are unnecessary if only a single statement is being repeated. The body of the ***while*** (or any other of Java's loops) can be empty. This is because a ***null*** *statement (one that consists only of a semicolon) is syntactically valid in Java.*

Object Oriented Programming using Java

Java's Iteration Statements: ...

```
// Demonstrate the while loop.
class While {
  public static void main(String args[]) {
    int n = 10;

    while(n > 0) {
      System.out.println("tick " + n);
      n--;
    }
  }
}
```

```
// Demonstrate the do-while loop.
class DoWhile {
  public static void main(String args[]) {
    int n = 10;

    do {
      System.out.println("tick " + n);
      n--;
    } while(n > 0);
  }
}
```

```
// Demonstrate the for loop.
class ForTick {
  public static void main(String args[]) {
    int n;

    for(n=10; n>0; n--)
      System.out.println("tick " + n);
  }
}
```

```
// Declare a loop control variable inside the for
class ForTick {
  public static void main(String args[]) {

    // here, n is declared inside of the for loop
    for(int n=10; n>0; n--)
      System.out.println("tick " + n);
  }
}
```

Object Oriented Programming using Java

Java's Iteration Statements: ... *for*

- Beginning with JDK 5, there are two forms of the **for** loop. The first is the traditional form that has been in use since the original version of Java. The second is the new ***"for-each"*** form.

- **Declaring Loop Control Variables Inside the for Loop:**

```
for(int n=10; n>0; n--)
```

- **Using the Comma:**

```
for(a=1, b=4; a<b; a++, b--)
```

- **Some for Loop Variations:**

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

Java's Iteration Statements: ... *for*...

- Some for Loop Variations:...

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

// Parts of the for loop can be empty.
for( ; !done; )

for( ; ; ) {
    // ...
}
```

Object Oriented Programming using Java

Java's Iteration Statements: ... *for*...

- **The For-Each Version of the for Loop:**

Beginning with JDK 5, a second form of **for** was defined that implements a **"for-each"** style loop.

- A f***oreach*** style loop is designed to cycle through a collection of objects, such as an array, in strictly sequential fashion, from start to finish. Unlike some languages, such as C#, that implement a for-each loop by using the keyword **foreach,** Java adds the for-each capability by enhancing the **for** statement.
- The advantage of this approach is that no new keyword is required, and no pre-existing code is broken. The for-each style of **for** is also referred to as the ***enhanced for loop.***
- General form:

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

Object Oriented Programming using Java

Java's Iteration Statements: ... *for*...

- **The For-Each Version of the for Loop:...*Examples***

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

Can be expressed as:

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

```
// Use break with a for-each style for.
class ForEach2 {
    public static void main(String args[]) {
        int sum = 0;
        int nums[] = { 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 };
        // use for to display and sum the values
        for(int x : nums) {
            System.out.println("Value is: " + x);
            sum += x;
            if(x == 5) break; // stop the loop when 5 is obtained
        }
        System.out.println("Summation of first 5 elements: " + su
    }
}
```

Object Oriented Programming using Java

Java's Iteration Statements: ... *for*...

- **The For-Each Version of the for Loop:...**
- ***Iterating Over Multidimensional Arrays... Examples***

```
// Use for-each style for on a two-dimensional array.
class ForEach3 {
  public static void main(String args[]) {
    int sum = 0;
    int nums[][] = new int[3][5];

    // give nums some values
    for(int i = 0; i < 3; i++)
      for(int j=0; j < 5; j++)
        nums[i][j] = (i+1)*(j+1);

    // use for-each for to display and sum the values
    for(int x[] : nums) {
      for(int y : x) {
        System.out.println("Value is: " + y);
        sum += y;
      }
    }
    System.out.println("Summation: " + sum);
  }
}
```

Object Oriented Programming using Java

Java's Iteration Statements: ... *for*...

- **The For-Each Version of the for Loop:...**
- ***Applying the Enhanced for... Examples***

```
// Search an array using for-each style for.
class Search {
  public static void main(String args[]) {
    int nums[] = { 6, 8, 3, 7, 5, 6, 1, 4 };
    int val = 5;
    boolean found = false;

    // use for-each style for to search nums for val
    for(int x : nums) {
      if(x == val) {
        found = true;
        break;
      }
    }

    if(found)
      System.out.println("Value found!");
  }
}
```

Object Oriented Programming using Java

Java's Iteration Statements: ... *for*...

- **The For-Each Version of the for Loop:...**
- ***Nested Loops... Examples***

```
// Loops may be nested.
class Nested {
  public static void main(String args[]) {
    int i, j;

    for(i=0; i<10; i++) {
      for(j=i; j<10; j++)
        System.out.print(".");
      System.out.println();
    }
  }
}
```

Java's Jump Statements: *break, continue, and return.*

- These statements transfer control to another part of your program.
- ***Using break:*** Statement has three uses:
  - Terminates a statement sequence in a switch statement.
  - Used to exit a loop
  - Used as a "civilized" form of ***goto.***

break was not designed to provide the normal means by which a loop is terminated. *The loop's conditional expression serves this purpose. The* ***break statement should be used to*** *cancel a loop only when some sort of special situation occurs.*

Object Oriented Programming using Java

Java's Jump Statements:...*Using break: ...*

•***Using break to Exit a Loop:*** force immediate termination of a loop, bypassing the conditional expression and any remaining code in the body of the loop.

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

- ***Using break as a form of goto:*** **break *label;***

Java does not have a goto statement because it provides a way to branch in an arbitrary and unstructured manner. This usually makes goto-ridden code hard to understand and hard to maintain. It also prohibits certain compiler optimizations.

```
// Using break to exit from nested loops
class BreakLoop4 {
  public static void main(String args[]) {
    outer: for(int i=0; i<3; i++) {
      System.out.print("Pass " + i + ": ");
      for(int j=0; j<100; j++) {
        if(j == 10) break outer; // exit both loops
        System.out.print(j + " ");
      }
      System.out.println("This will not print");
    }
    System.out.println("Loops complete.");
  }
}
```

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

  one: for(int i=0; i<3; i++) {
    System.out.print("Pass " + i + ": ");
  }

  for(int j=0; j<100; j++) {
    if(j == 10) break one; // WRONG
    System.out.print(j + " ");
  }
}
```

Java's Jump Statements:...

Using continue: *C*auses control to be transferred directly to the conditional expression that controls the loop. Sometimes it is useful to force an early iteration of a loop.

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

```
// Using continue with a label.
class ContinueLabel {
  public static void main(String args[]) {
outer: for (int i=0; i<10; i++) {
         for(int j=0; j<10; j++) {
           if(j > i) {
             System.out.println();
             continue outer;
           }
           System.out.print(" " + (i * j));
         }
       }
       System.out.println();
  }
}
```

As with the ***break*** statement, ***continue*** may specify a label to describe which enclosing loop to continue.

Java's Jump Statements:... *Return*

- The last control statement is **return.** The return statement is used to explicitly return from a method. That is, it causes program control to transfer back to the caller of the method.

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

    if(t) return; // return to caller

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

Object Oriented Programming using Java

Arrays:

- A group of ***like-typed*** variables that are referred to by a common name.
- Arrays of any type can be created and may have one or more dimensions. A specific element in an array is accessed by its index.
- Arrays offer a convenient means of grouping related information.
- **Eg:**
  - ***One-Dimensional Arrays:*** Essentially, a list of like-typed variables.
  - ***Multidimensional Arrays:*** Arrays of arrays.

Object Oriented Programming using Java

One-Dimensional Arrays: A list of like-typed variables. To create an array, you first must create an array variable of the desired type.

- General form: ***type var-name[ ];***
- **Eg:** int month_days[];

month_days is an array variable, no array actually exists. In fact, the value of **month_days** is set to **null,** which represents ***an array with no value.***

- To allocate memory for arrays, the general form:

array-var = new type[size]; or

type array-var[] = new type[size];

- **Eg:** month_days = new int[12];

- It is possible to combine the declaration of the array variable with the allocation of the array itself, as shown here: ***int month_days[] = new int[12];***

Object Oriented Programming using Java

One-Dimensional Arrays: ...

```
// An improved version.
class AutoArray {
    public static void main(String args[]) {
        int month_days[] = { 31, 28, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31 };
        System.out.println("April has " + month_days[3] + "
        days.");
    }
}
```

Object Oriented Programming using Java

Multidimensional Arrays: *arrays of arrays*

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

```
// Demonstrate a two-dimensional array.
class TwoDArray {
    public static void main(String args[]) {
        int twoD[][]= new int[4][5];
        int i, j, k = 0;
        for(i=0; i<4; i++)
        for(j=0; j<5; j++) {
            twoD[i][j] = k;
            k++;
        }
        for(i=0; i<4; i++) {
            for(j=0; j<5; j++)
            System.out.print(twoD[i][j] + " ");
            System.out.println();
        }
    }
}
```

Object Oriented Programming using Java

Multidimensional Arrays: …

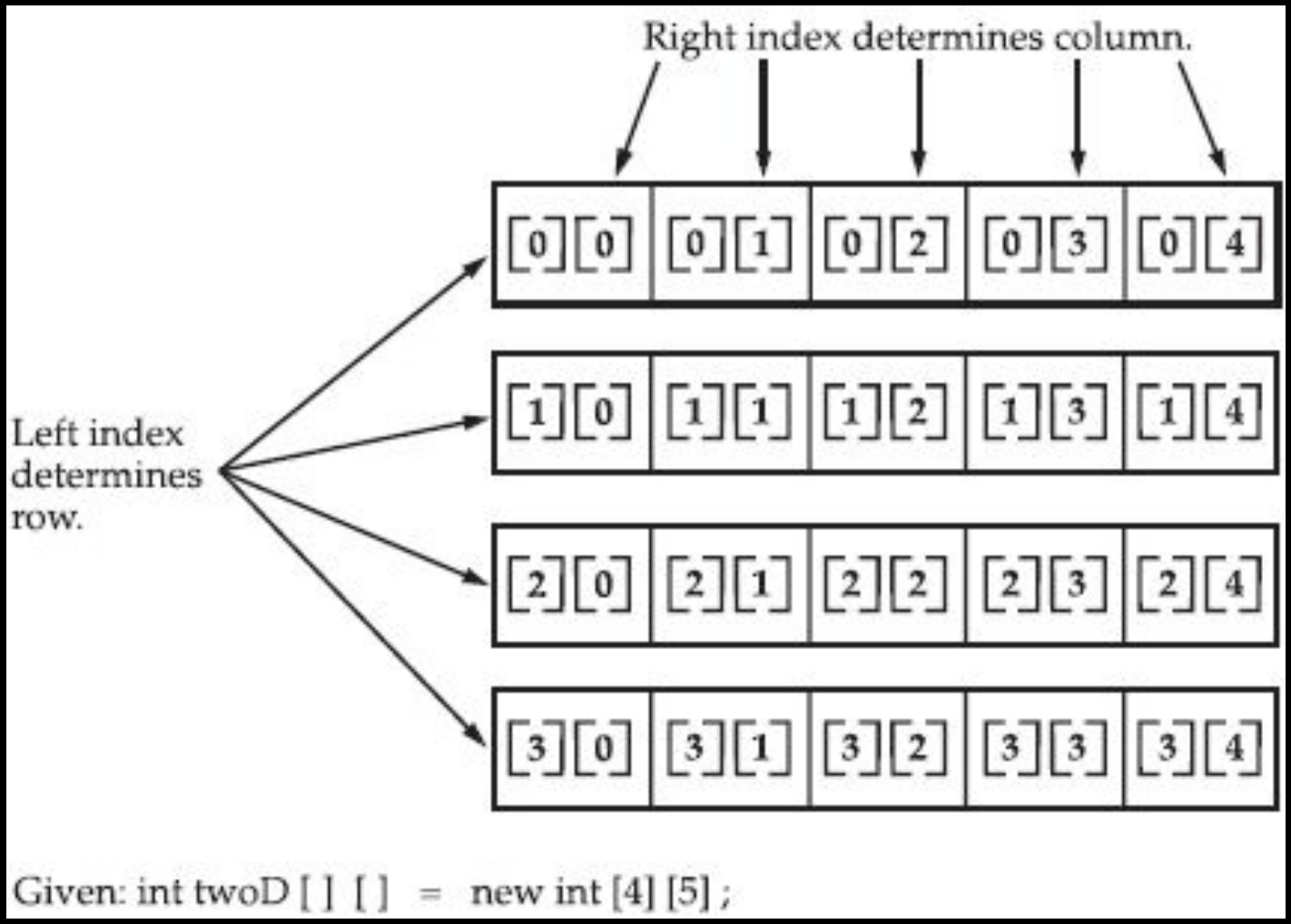
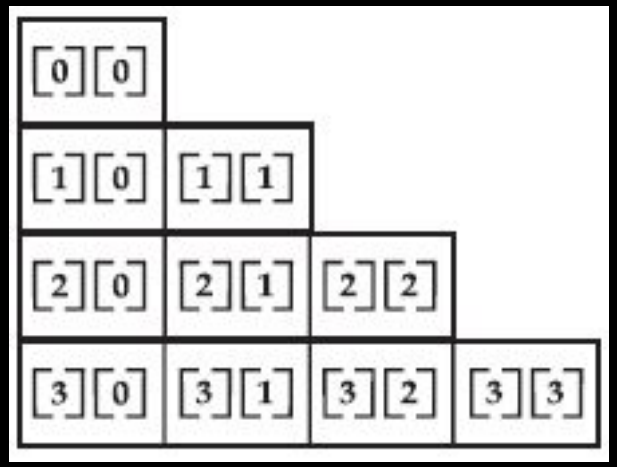


Figure : ***A conceptual view of a 4 by 5, two-dimensional array***

Multidimensional Arrays: … *uneven/irregular*

```
// Manually allocate differing size second dimensions.
class TwoDAgain {
    public static void main(String args[]) {
    int twoD[][] = new int[4][];
        twoD[0] = new int[1];
        twoD[1] = new int[2];
        twoD[2] = new int[3];
        twoD[3] = new int[4];
        . . .
```



- The use of uneven (or, irregular) multidimensional arrays may not be appropriate for many applications, because it runs contrary to what people expect to find when a multidimensional array is encountered. However, irregular arrays can be used effectively in some situations. For example, if you need a very large two-dimensional array that is sparsely populated (that is, one in which not all of the elements will be used), then an irregular array might be a perfect solution.

Object Oriented Programming using Java

- **Alternative Array Declaration Syntax:**

type[ ] var-name;

int al[] = new int[3]; **OR** int[] a2 = new int[3];

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

OR

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

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

OR

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.***