Lecture 1

PRESENTATION BY KHUSHWANT SINGH

Features of Java

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

- **Simple**
- Java was designed to be easy for the professional programmer to learn and use effectively. Assuming that you have some programming experience, you will not find Java hard to master. If you already understand the basic concepts of object-oriented programming, learning Java will be even easier.
- **Object-Oriented**
- Although influenced by its predecessors, Java was not designed to be source-code compatible with any other language. This allowed the Java team the freedom to design with a blank slate. One outcome of this was a clean, usable, pragmatic approach to objects. Borrowing liberally from many seminal object-software environments of the last few decades, Java manages to strike a balance between the purist's "everything is an object" paradigm and the pragmatist's "stay out of my way" model. The object model in Java is simple and easy to extend, while primitive types, such as integers, are kept as high-performance nonobjects.

- **Robust**
- The multiplatformed environment of the Web places extraordinary demands on a program, because the program must execute reliably in a variety of systems. Thus, the ability to create robust programs was given a high priority in the design of Java. To gain reliability, Java restricts you in a few key areas to force you to find your mistakes early in program development. At the same time, Java frees you from having to worry about many of the most common causes of programming errors. Because Java is a strictly typed language, it checks your code at compile time. However, it also checks your code at run time. Many hard-to-track-down bugs that often turn up in hard-to-reproduce run-time situations are simply impossible to create in Java. Knowing that what you have written will behave in a

predictable way under diverse conditions is a key feature of Java.

- **Multithreaded**
- Java was designed to meet the real-world requirement of creating interactive, networked programs. To accomplish this, Java supports multithreaded programming, which allows you to write programs that do many things simultaneously. The Java run-time system comes with an elegant yet sophisticated solution for multiprocess synchronization that enables you to construct smoothly running interactive systems. Java's easy-to-use approach to multithreading allows you to think about the specific behavior of your program, not the multitasking subsystem.
- **Architecture-Neutral**
- Acentral issue for the Java designers was that of code longevity and portability. One of the main problems facing programmers is that no guarantee exists that if you write a program today, it will run tomorrow—even on the same machine. Operating system upgrades, processor upgrades, and changes in core system resources can all combine to make a program malfunction. The Java designers made several hard decisions in the Java language and the Java Virtual Machine in an attempt to alter this situation. Their goal was "write once; run anywhere, any time, forever."To a great extent, this goal was accomplished.

- **Interpreted and High Performance**
- As described earlier, 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. Most previous attempts at
- cross-platform solutions have done so at the expense of performance. As explained earlier,
- the 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 compiler. Java run-time
- systems that provide this feature lose none of the benefits of the platform-independent code.

- **Distributed**
- Java is designed for the distributed environment of the Internet because it handles TCP/IP
- protocols. In fact, accessing a resource using a URL is not much different from accessing a
- file. Java also supports *Remote Method Invocation (RMI). This feature enables a program to*
- invoke methods across a network.

- **Dynamic**
- Java programs carry with them substantial amounts of run-time type information that is used to verify and resolve accesses to objects at run time. This makes it possible to dynamically link code in a safe and expedient manner. This is crucial to the robustness of the Java environment, in which small fragments of bytecode may be dynamically updated on a running system.

Type Conversion and Casting

Java's Automatic Conversions

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

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

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

Casting Incompatible Types

- 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*
- int a;
- byte b;
- // …
- b = (byte) a;

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.

One-Dimensional Arrays

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

- *type var-name[ ];*
- Here, *type declares the base type of the array.*
- *array-var = new type[size];*
- We 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.**

- Example: int month_days[]; month_days = new int[12];
- // Demonstrate a one-dimensional array.
- class Array {
- public static void main(String args[]) {
- int month_days[];
- month_days = new int[12];
- month_days[0] = 31;
- month_days[1] = 28;
- month_days[2] = 31;
- month_days[3] = 30;
- month_days[4] = 31;
- month_days[5] = 30;
- month_days[6] = 31;
- month_days[7] = 31;
- month_days[8] = 30;
- month_days[9] = 31;
- month_days[10] = 30;
- month_days[11] = 31;
- System.out.println("April has " + month_days[3] + " days.");
- }
- }

```
// An improved version of the previous program.
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.");
}
}
```

Multidimensional Arrays

- In Java, *multidimensional arrays are actually arrays of arrays.*
- To declare a multidimensional array variable, specify each additional index using another set of square brackets.
- For example, the following declares a two dimensional array variable called **twoD.**
- int twoD[][] = new int[4][5];
- This allocates a 4 by 5 array and assigns it to **twoD.**

// Demonstrate a two-dimensional array.

- class TwoDArray {
- public static void main(String args[]) {
- int twoD[][]= new int[4][5];
- int i, j, k = 0;
- for(i=0; i<4; i++)
- for(j=0; j<5; j++) {
- twoD[i][j] = k;
- k++;
- }
- for(i=0; i<4; i++) {
- for(j=0; j<5; j++)
- System.out.print(twoD[i][j] + " ");
- System.out.println();
- }
- }
- }
- This program generates the following output:
- 0 1 2 3 4
- 5 6 7 8 9
- 10 11 12 13 14
- 15 16 17 18 19

// 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];
- int i, j, k = 0;
- for(i=0; i<4; i++)
- for(j=0; j<i+1; j++) {
- twoD[i][j] = k;
- k++;
- }
- for(i=0; i<4; i++) {
- for(j=0; j<i+1; j++)
- System.out.print(twoD[i][j] + " ");
- System.out.println();
- }
- }
- }
- This program generates the following output:
- 0
- 1 2
- 3 4 5
- 6 7 8 9

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

Exploring the String Class

- The first thing to understand about strings is that every string you create is actually an object of type **String. Even string constants are actually String objects. For example, in the** statement
- System.out.println("This is a String, too");
- the string "This is a String, too" is a **String constant.**

- The second thing to understand about strings is that objects of type **String are immutable;** once a **String object is created, its contents cannot be altered. While this may seem like a** serious restriction, it is not, for two reasons:

1. If you need to change a string, you can always create a new one that contains the modifications.
2. Java defines a peer class of **String, called StringBuffer, which allows strings** to be altered, so all of the normal string manipulations are still available in Java.

- Strings can be constructed in a variety of ways. The easiest is to use a statement like this:
- String myString = "this is a test";
- System.out.println(myString);

- Java defines one operator for **String objects: +. It is used to concatenate two strings.**
- For example, this statement
- String myString = "I" + " like " + "Java.";
- results in **myString containing "I like Java."**

// Demonstrating Strings.

- class StringDemo {
- public static void main(String args[]) {
- String strOb1 = "First String";
- String strOb2 = "Second String";
- String strOb3 = strOb1 + " and " + strOb2;
- System.out.println(strOb1);
- System.out.println(strOb2);
- System.out.println(strOb3);
- }
- }
- The output produced by this program is shown here:
- First String
- Second String
- First String and Second String

- The **String class contains several methods that you can use. Here are a few. You can test** two strings for equality by using **equals( ).**
- **You can obtain the length of a string by calling** the **length( ) method.**
- **You can obtain the character at a specified index within a string by**
- calling **charAt( ).**
- **The general forms of these three methods are shown here:**
- boolean equals(String *object)*
- int length( )
- char charAt(int *index)*

// Demonstrating some String methods.

- class StringDemo2 {
- public static void main(String args[]) {
- String strOb1 = "First String";
- String strOb2 = "Second String";
- String strOb3 = strOb1;
- System.out.println("Length of strOb1: " +
- strOb1.length());
- System.out.println("Char at index 3 in strOb1: " +
- strOb1.charAt(3));
- if(strOb1.equals(strOb2))
- System.out.println("strOb1 == strOb2");
- else
- System.out.println("strOb1 != strOb2");
- if(strOb1.equals(strOb3))
- System.out.println("strOb1 == strOb3");
- Else
- System.out.println("strOb1 != strOb3");
- }
- }
- This program generates the following output:
- Length of strOb1: 12
- Char at index 3 in strOb1: s
- strOb1 != strOb2
- strOb1 == strOb3