**Java Lectures Summary**

A computer program is a sequence of instructions and decisions. A program is stored in secondary storage, typically a hard disk, when it is not currently running. The central processing unit of the computer carries out arithmetic operations, such as addition and multiplication.

Java is a Programming Language. It is Safe, Portable, Platform-independent and has a Vast set of library packages. Safety and portability are the two most important benefits of the Java language. Java is case sensitive. Java compiler translates source code into class files (byte code). Class files contain instructions for the Java virtual machine.

Classes are the fundamental building blocks of Java programs. In Java, every source file can contain, at most one public class. The name of the public class must match the name of the file containing the class. Each class contains declarations of methods. Each method contains a sequence of instructions. A method contains a collection of programming instructions that describe how to carry out a particular task. A method is called by specifying the method and its arguments.

A Java application usually contains many classes. But every Java application contains a class with a main method. When the application starts, the instructions in the main method are executed.

In Java, programs are build for objects. Objects are the entities that you can manipulate by calling its methods. Objects are created from classes. You can create many objects from a class. For example, "String" is class of Java library. In your program you can many objects from this class. They will have different names and different contents that means different objects but all belong to the same class "String". A class describes a set of objects with the same behavior.

To hold any value/data in java, variables are used. In Java, all variables must be declared before they can be used. When declaring a variable in Java, its type (data type) and name must be specified. There are several rules for declaring the name of the variable like it should not start with digit, no space in variable name etc (See Book for more details). There are three kinds of variables in Java: Local variables, Instance variables and Class/static variables. Local variables are declared in methods, constructors, or blocks. Instance variables are declared in a class, but outside a method, constructor or any block. Class variables also known as static variables are declared with the static keyword in a class, but outside a method, constructor or a block.

In Java you can add comments using // (comment from here to the end of line)

 and /\* multi

 line

 comments

\*/.

Assignment operator (=) is used to update the value of a variable. e.g.

variableToBeUpdated = value;

A class can declare two or more methods with the same name and different argument types. This is called as method overloading as in the case of System.out.println() method. You can print different type of data using println() method, this is because in java library there are many form of println() method.

Encapsulation is the process of hiding the implementation details and providing methods for data access. Methods are also called as the behavior of the class/objects.

Inheritance is the relationship between a superclass (parent class or a more general class) and a subclass (child class or a more specialized class). The subclass inherits data and behavior from the super class. Inheritance lets you can reuse the code instead of duplicating it. We use 'extends' keyword to inherit a class in Java. When you inherit a class its data and behavior (methods) are also inherited (depending upon the accessibility protected, public etc.). If you write a new definition or implementation of a superclass method in the subclass, then this is called as overriding. If a method in the superclass is "final" you cannot override it (see Lecture#4 Slide#14-15).

Operators in Java follow the following order of evaluation or precedence:



++ is an increment operator (adds 1 to the variables value) and -- is a decrement operator (subtracts 1 from the variables' current value). So if x =5, x++ will be 6, and if y = 5, y-- will be 4.

% is remainder operator. so 7%4 will be 3 while 8%4 will be 0.

To compute xn, you have to use pow() method from Math class. So, Math.pow(x, n) will compute xn.

To compute square root, you can use Math.sqrt(x).

You have to use a cast (typeName) to convert a value to a different type. For example, if you want double d = 4.0 to convert into an integer.

int i = (int) d;

To read user input, you can use Scanner class. To use Scanner class, you have to import it first. So you have to write.

import java.util;

and to read input in the program, you have to write:

Scanner input = new Scanner(System.in);

String s = input.next(); // to read String

new is a keyword in Java and is use to create an object of the class.

You can use + operator to concatenate two strings also.

Some important methods (see more details this isn't a complete list) from String class.

|  |  |
| --- | --- |
| length() | Returns the length of this string. |
| substring(...) | Returns a new string that is a substring of this string. |
| toLowerCase() | Converts all of the characters in this String to lower case |
| toUpperCase() | Converts all of the characters in this String to upper case |
| charAt(...) | Returns the character at the specified index. |
| String trim() | Returns a copy of the string, with leading and trailing whitespace omitted. |
| equals(...) | Compares this string to the specified object. |
| indexOf(...)  | Returns the index within this string of the first occurrence of the specified character/string. |

If statement allows a program to carry out different actions depending on the nature of the data to be processed. An if statement consists of a Boolean expression followed by one or more statements. The syntax of an if statement is:

if(Boolean\_expression)

{

 //Statements will execute if the Boolean expression is true

}

If the Boolean expression evaluates to true then the block of code inside the if statement will be executed. If not the first set of code after the end of the if statement (after the closing curly brace) will be executed

An if statement can be followed by an optional else statement, which executes when the Boolean expression is false.

if(Boolean\_expression){

 //Executes when the Boolean expression is true

}else{

 //Executes when the Boolean expression is false

}

An if statement can be followed by an optional else if...else statement, which is very useful to test various conditions using single if...else if statement. When using if , else if , else statements there are few points to keep in mind.

* An if can have zero or one else's and it must come after any else if's.
* An if can have zero to many else if's and they must come before the else.
* Once an else if succeeds, none of the remaining else if's or else's will be tested.

if(Boolean\_expression 1){

 //Executes when the Boolean expression 1 is true

}else if(Boolean\_expression 2){

 //Executes when the Boolean expression 2 is true

}else if(Boolean\_expression 3){

 //Executes when the Boolean expression 3 is true

}else {

 //Executes when the none of the above condition is true.

}

It is always legal to nest if-else statements which means you can use one if or else if statement inside another if or else if statement.

if(Boolean\_expression 1){

 //Executes when the Boolean expression 1 is true

 if(Boolean\_expression 2){

 //Executes when the Boolean expression 2 is true

 }

}

You can nest else if...else in the similar way as we have nested if statement.

There may be a situation when we need to execute a block of code several number of times, and is often referred to as a loop.

Java has very flexible three looping mechanisms. You can use one of the following three loops:

* while Loop
* do...while Loop
* for Loop

As of Java 5, the *enhanced for loop* (also called as *foreach* loop) was introduced. This is mainly used for accessing contents of Arrays or other data structures.

**The while Loop:**

A while loop is a control structure that allows you to repeat a task a certain number of times.

Syntax:

The syntax of a while loop is:

while(Boolean\_expression)

{

 //Statements

}

When executing, if the *boolean\_expression* result is true, then the actions inside the loop will be executed. This will continue as long as the expression result is true.

Here, key point of the *while* loop is that the loop might not ever run. When the expression is tested and the result is false, the loop body will be skipped and the first statement after the while loop will be executed.

Example:

public class Test {

 public static void main(String args[]) {

 int x = 10;

 while( x < 20 ) {

 System.out.print("value of x : " + x );

 x++;

 System.out.print("\n");

 }

 }

}

This would produce the following result:

value of x : 10

value of x : 11

value of x : 12

value of x : 13

value of x : 14

value of x : 15

value of x : 16

value of x : 17

value of x : 18

value of x : 19

**The do...while Loop:**

A do...while loop is similar to a while loop, except that a do...while loop is guaranteed to execute at least one time.

Syntax:

The syntax of a do...while loop is:

do

{

 //Statements

}while(Boolean\_expression);

Notice that the Boolean expression appears at the end of the loop, so the statements in the loop execute once before the Boolean is tested.

If the Boolean expression is true, the flow of control jumps back up to do, and the statements in the loop execute again. This process repeats until the Boolean expression is false.

Example:

public class Test {

 public static void main(String args[]){

 int x = 10;

 do{

 System.out.print("value of x : " + x );

 x++;

 System.out.print("\n");

 }while( x < 20 );

 }

}

This would produce the following result:

value of x : 10

value of x : 11

value of x : 12

value of x : 13

value of x : 14

value of x : 15

value of x : 16

value of x : 17

value of x : 18

value of x : 19

**The for Loop:**

A for loop is a repetition control structure that allows you to efficiently write a loop that needs to execute a specific number of times.

A for loop is useful when you know how many times a task is to be repeated.

Syntax:

The syntax of a for loop is:

for(initialization; Boolean\_expression; update)

{

 //Statements

}

Here is the flow of control in a for loop:

* The initialization step is executed first, and only once. This step allows you to declare and initialize any loop control variables. You are not required to put a statement here, as long as a semicolon appears.
* Next, the Boolean expression is evaluated. If it is true, the body of the loop is executed. If it is false, the body of the loop does not execute and flow of control jumps to the next statement past the for loop.
* After the body of the for loop executes, the flow of control jumps back up to the update statement. This statement allows you to update any loop control variables. This statement can be left blank, as long as a semicolon appears after the Boolean expression.
* The Boolean expression is now evaluated again. If it is true, the loop executes and the process repeats itself (body of loop, then update step, then Boolean expression). After the Boolean expression is false, the for loop terminates.

Example:

public class Test {

 public static void main(String args[]) {

 for(int x = 10; x < 20; x = x+1) {

 System.out.print("value of x : " + x );

 System.out.print("\n");

 }

 }

}

This would produce the following result:

value of x : 10

value of x : 11

value of x : 12

value of x : 13

value of x : 14

value of x : 15

value of x : 16

value of x : 17

value of x : 18

value of x : 19

Enhanced for loop in Java:

As of Java 5, the enhanced for loop was introduced. This is mainly used for Arrays.

Syntax:

The syntax of enhanced for loop is:

for(declaration : expression)

{

 //Statements

}

* **Declaration:** The newly declared block variable, which is of a type compatible with the elements of the array you are accessing. The variable will be available within the for block and its value would be the same as the current array element.
* **Expression:** This evaluates to the array you need to loop through. The expression can be an array variable or method call that returns an array.

Example:

public class Test {

 public static void main(String args[]){

 int [] numbers = {10, 20, 30, 40, 50};

 for(int x : numbers ){

 System.out.print( x );

 System.out.print(",");

 }

 System.out.print("\n");

 String [] names ={"James", "Larry", "Tom", "Lacy"};

 for( String name : names ) {

 System.out.print( name );

 System.out.print(",");

 }

 }

}

This would produce the following result:

10,20,30,40,50,

James,Larry,Tom,Lacy,

**The break Keyword:**

The *break* keyword is used to stop the entire loop. The break keyword must be used inside any loop or a switch statement.

The break keyword will stop the execution of the innermost loop and start executing the next line of code after the block.

Syntax:

The syntax of a break is a single statement inside any loop:

break;

Example:

public class Test {

 public static void main(String args[]) {

 int [] numbers = {10, 20, 30, 40, 50};

 for(int x : numbers ) {

 if( x == 30 ) {

 break;

 }

 System.out.print( x );

 System.out.print("\n");

 }

 }

}

This would produce the following result:

10

20

**The continue Keyword:**

The *continue* keyword can be used in any of the loop control structures. It causes the loop to immediately jump to the next iteration of the loop.

* In a for loop, the continue keyword causes flow of control to immediately jump to the update statement.
* In a while loop or do/while loop, flow of control immediately jumps to the Boolean expression.

Syntax:

The syntax of a continue is a single statement inside any loop:

continue;

Example:

public class Test {

 public static void main(String args[]) {

 int [] numbers = {10, 20, 30, 40, 50};

 for(int x : numbers ) {

 if( x == 30 ) {

 continue;

 }

 System.out.print( x );

 System.out.print("\n");

 }

 }

}

This would produce the following result:

10

20

40

50

**Arrays**

An array is used to store a collection of variables of the same type. Instead of declaring individual variables, such as number0, number1, ..., and number99, you declare one array variable such as numbers and use numbers[0], numbers[1], and ..., numbers[99] to represent individual variables.

To use an array in a program, you must declare a variable to reference the array, and you must specify the type of array the variable can reference. Here is the syntax for declaring an array variable:

dataType[] arrayRefVar; // preferred way.

or

dataType arrayRefVar[]; // works but not preferred way.

You can create an array by using the new operator with the following syntax:

arrayRefVar = new dataType[arraySize];

The above statement does two things:

* It creates an array using new dataType[arraySize];
* It assigns the reference of the newly created array to the variable arrayRefVar.

Declaring an array variable, creating an array, and assigning the reference of the array to the variable can be combined in one statement, as shown below:

dataType[] arrayRefVar = new dataType[arraySize];

Alternatively you can create arrays as follows:

dataType[] arrayRefVar = {value0, value1, ..., valuek};

The array elements are accessed through the **index**. Array indices are 0-based; that is, they start from 0 to **arrayRefVar.length-1**.

Following statement declares an array variable, myList, creates an array of 10 elements of double type and assigns its reference to myList:

double[] myList = new double[10];

Following picture represents array myList. Here, myList holds ten double values and the indices are from 0 to 9.



When processing array elements, we often use either for loop or foreach loop because all of the elements in an array are of the same type and the size of the array is known.

public class TestArray {

 public static void main(String[] args) {

 double[] myList = {1.9, 2.9, 3.4, 3.5};

 // Print all the array elements

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

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

 }

 // Summing all elements

 double total = 0;

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

 total += myList[i];

 }

 System.out.println("Total is " + total);

 // Finding the largest element

 double max = myList[0];

 for (int i = 1; i < myList.length; i++) {

 if (myList[i] > max) max = myList[i];

 }

 System.out.println("Max is " + max);

 }

}

This would produce the following result:

1.9

2.9

3.4

3.5

Total is 11.7

Max is 3.5

The foreach Loops:

JDK 1.5 introduced a new for loop known as foreach loop or enhanced for loop, which enables you to traverse the complete array sequentially without using an index variable.

The following code displays all the elements in the array myList:

public class TestArray {

 public static void main(String[] args) {

 double[] myList = {1.9, 2.9, 3.4, 3.5};

 // Print all the array elements

 for (double element: myList) {

 System.out.println(element);

 }

 }

}

This would produce the following result:

1.9

2.9

3.4

3.5

**Passing Arrays to Methods:**

Just as you can pass primitive type values to methods, you can also pass arrays to methods. For example, the following method displays the elements in an int array:

public static void printArray(int[] array) {

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

 System.out.print(array[i] + " ");

 }

}

You can invoke it by passing an array. For example, the following statement invokes the printArray method to display 3, 1, 2, 6, 4, and 2:

printArray(new int[]{3, 1, 2, 6, 4, 2});

**Returning an Array from a Method:**

A method may also return an array. For example, the method shown below returns an array that is the reversal of another array:

public static int[] reverse(int[] list) {

 int[] result = new int[list.length];

 for (int i = 0, j = result.length - 1; i < list.length; i++, j--) {

 result[j] = list[i];

 }

 return result;

}

**Two-dimensional arrays**

To declare a two-dimensional array, you simply list two sets of empty brackets, like this:

int numbers[][];

Here, numbers is a two-dimensional array of type int. To put it another way, numbers is an array of int arrays. To create the array, you use the new keyword and provide lengths for each set of brackets, as in this example:

numbers = new int[10][8];

Here, the first dimension specifies that the numbers array has 10 elements. The second dimension specifies that each of those elements is itself an array with 8 elements.

To access the elements of a two-dimensional array, you use two indexes. For example:

int[5][7] = 23853;

Often, nested for loops are used to process the elements of a two-dimensional array, as in this example:

for (int x = 0; x < 10; x++)

{

 for (int y = 0; y < 8; y++)

 {

 numbers[x][y] = (int)(Math.random() \* 100) + 1

 }

}

//Math.random() method returns a random double number

You can use an array initializer with a two-dimensional array, as in this example:

String members[][] =

 {

 {"Larry", "Curly", "Moe" },

 {"Manny", "Moe", "Jack"},

 {"Huey", "Dewey", "Louie"}

 };

When you create an array with an expression — such as new int[5][3] — you’re specifying that each element of the main array is actually an array of type int with three elements~~. Java, however, lets you create two-dimensional arrays in which the length of each element of the main array is different. Sometimes, this is called a~~*~~jagged array~~*~~because the array doesn’t form a nice rectangle. Instead, its edges are jagged~~.

**ArrayList**

Standard Java arrays are of a fixed length. After arrays are created, they cannot grow or shrink, which means that you must know in advance how many elements an array will hold.

Array lists are created with an initial size. When this size is exceeded, the collection is automatically enlarged. When objects are removed, the array may be shrunk.

The ArrayList class supports three constructors. The first constructor builds an empty array list.

ArrayList( )

The following constructor builds an array list that has the specified initial capacity. The capacity is the size of the underlying array that is used to store the elements.

The capacity grows automatically as elements are added to an array list.

ArrayList(int capacity)

Apart from the methods inherited from its parent classes, ArrayList defines following methods:

|  |  |
| --- | --- |
| **SN** | **Methods with Description** |
| 1 | **void add(int index, Object element)**Inserts the specified element at the specified position index in this list. Throws IndexOutOfBoundsException if the specified index is is out of range (index < 0 || index > size()). |
| 2 | **boolean add(Object o)** Appends the specified element to the end of this list. |
| 3 | **boolean addAll(Collection c)**Appends all of the elements in the specified collection to the end of this list, in the order that they are returned by the specified collection's iterator. Throws NullPointerException if the specified collection is null. |
| 4 | **boolean addAll(int index, Collection c)** Inserts all of the elements in the specified collection into this list, starting at the specified position. Throws NullPointerException if the specified collection is null. |
| 5 | **void clear()** Removes all of the elements from this list. |
| 6 | **Object clone()** Returns a shallow copy of this ArrayList. |
| 7 | **boolean contains(Object o)** Returns true if this list contains the specified element. More formally, returns true if and only if this list contains at least one element e such that (o==null ? e==null : o.equals(e)).  |
| 8 | **void ensureCapacity(int minCapacity)** Increases the capacity of this ArrayList instance, if necessary, to ensure that it can hold at least the number of elements specified by the minimum capacity argument. |
| 9 | **Object get(int index)** Returns the element at the specified position in this list. Throws IndexOutOfBoundsException if the specified index is is out of range (index < 0 || index >= size()). |
| 10 | **int indexOf(Object o)** Returns the index in this list of the first occurrence of the specified element, or -1 if the List does not contain this element. |
| 11 | **int lastIndexOf(Object o)**Returns the index in this list of the last occurrence of the specified element, or -1 if the list does not contain this element. |
| 12 | **Object remove(int index)** Removes the element at the specified position in this list. Throws IndexOutOfBoundsException if index out of range (index < 0 || index >= size()). |
| 13 | **protected void removeRange(int fromIndex, int toIndex)** Removes from this List all of the elements whose index is between fromIndex, inclusive and toIndex, exclusive. |
| 14 | **Object set(int index, Object element)** Replaces the element at the specified position in this list with the specified element. Throws IndexOutOfBoundsException if the specified index is is out of range (index < 0 || index >= size()). |
| 15 | **int size()** Returns the number of elements in this list. |
| 16 | **Object[] toArray()** Returns an array containing all of the elements in this list in the correct order. Throws NullPointerException if the specified array is null. |
| 17 | **Object[] toArray(Object[] a)** Returns an array containing all of the elements in this list in the correct order; the runtime type of the returned array is that of the specified array. |
| 18 | **void trimToSize()** Trims the capacity of this ArrayList instance to be the list's current size. |

The following program illustrates several of the methods supported by ArrayList:

import java.util.\*;

public class ArrayListDemo {

 public static void main(String args[]) {

 // create an array list

 ArrayList al = new ArrayList();

 System.out.println("Initial size of al: " + al.size());

 // add elements to the array list

 al.add("C");

 al.add("A");

 al.add("E");

 al.add("B");

 al.add("D");

 al.add("F");

 al.add(1, "A2");

 System.out.println("Size of al after additions: " + al.size());

 // display the array list

 System.out.println("Contents of al: " + al);

 // Remove elements from the array list

 al.remove("F");

 al.remove(2);

 System.out.println("Size of al after deletions: " + al.size());

 System.out.println("Contents of al: " + al);

 }

}

This would produce the following result:

Initial size of al: 0

Size of al after additions: 7

Contents of al: [C, A2, A, E, B, D, F]

Size of al after deletions: 5

Contents of al: [C, A2, E, B, D]

**Discovering Classes**

A class represents a single concept from the problem domain

Name for a class should be a noun that describes concept

Concepts from mathematics: Point, Rectangle, Ellipse

Concepts from real life: BankAccount, CashRegister

Actors (end in -er, -or) – objects do some kinds of work for you: Scanner
Random // better name: RandomNumberGenerator

Utility classes – no objects, only static methods and constants: Math ,

Program starters: only have a main method

Don’t turn actions into classes, Paycheck is a better name than ComputePaycheck

**Cohesion**: A class should represent a single concept. The public interface of a class is cohesive if all of its features are related to the concept that the class represents

**Coupling:** Howa class depends on another if it uses objects of that class. Minimize coupling to minimize the impact of interface changes

**Accessor:** Does not change the state of the implicit parameter:

double balance = account.getBalance();

**Mutator:** Modifies the object on which it is invoked:

account.deposit(1000);

**Immutable class:** Has no mutator methods (e.g., String):

String name = "John Q. Public";

String uppercased = name.toUpperCase(); // name is not changed

It is safe to give out references to objects of immutable classes; no code can modify the object at an unexpected time

**Call by value**: Method parameters are copied into the parameter variables when a method starts

**Call by reference**: Methods can modify parameters

Java has call by value. A method can change state of object reference parameters, but cannot replace an object reference with another

**Precondition:** Requirement that the caller of a method must meet. If precondition is violated, method is not responsible for computing the correct result. It is free to do *anything*

Typical use: To restrict the parameters of a method. To require that a method is only called when the object is in an appropriate state

**Post-condition:** requirement that is true after a method has completed. If method call is in accordance with preconditions, it must ensure that postconditions are valid. There are two kinds of postconditions: 1) The return value is computed correctly. 2) The object is in a certain state after the method call is completed

**Method Side effects**

In computer science, a function or expression is said to have a side effect if, in addition to producing a value, it also modifies some state or has an observable interaction with calling functions or the outside world.

In the presence of side effects, a program's behavior depends on past history; that is, the order of evaluation matters.

**Scope of variable:** Region of program in which the variable can be accessed. Scope of a local variable extends from its declaration to end of the block that encloses it

If variables are independent from each other they can use the same variable name.

A local variable can shadow a variable with the same name, Local scope wins over class scope.

A **package** allows a developer to group classes (and interfaces) together. These classes will all be related in some way – they might all have to do with a specific application or perform a specific set of tasks. The Java API is a collection of packages – for example, the javax.xml package. The javax.xml package and its subpackages contain classes to handle XML.

A package provides a unique namespace for the types it contains. Classes in the same package can access each other's package-access members.

**Static**

Sometimes, you want to have variables that are common to all objects. This is accomplished with the static modifier. Fields that have the static modifier in their declaration are called static fields or class variables. They are associated with the class, rather than with any object. Every instance of the class shares a class variable, which is in one fixed location in memory. Any object can change the value of a class variable, but class variables can also be manipulated without creating an instance of the class.

The Java programming language supports static methods as well as static variables. Static methods, which have the static modifier in their declarations, should be invoked with the class name, without the need for creating an instance of the class

The static modifier, in combination with the final modifier, is also used to define constants. The final modifier indicates that the value of this field cannot change.