Object Oriented Concepts (17CS42)

Module – II

Question and Answers:

1. Explain the Java Bytecode

Java's Magic: the Byte Code:

The means that allows Java to solve both the security and the portability problems is that the output of a Java compiler is not executable code but it is the Bytecode. *Bytecode* is a highly optimized set of instructions designed to be executed by the Java run-time system, which is called the *lava Virtual Machine (JVM).* JVM is an *interpreter for bytecode.* The fact that a Java program is executed by JVM helps solve the major problems associated with downloading programs over the Internet. Translating a Java program into bytecode helps makes it much easier to run a program in a wide variety of environments. This is because only the JVM needs to be implemented for each platform.

Once the run-time package exists for a given system, any Java program can run on it. If a Java program is compiled to its subject code, then different versions of the same program would exist for each type of CPU connected to the Internet. This is not a practical solution. Thus, the understanding of bytecode is the most effective way to create strictly portable programs. Now, if a Java program is interpreted, it also helps to make it secure. Since the execution of every Java program is under the control of the JVM, the JVM can contain the program, and prevent it from generating side effects outside of the system. The use of byte code enables the Java run-time system to execute programs much faster.

Sun provides a facility called Just In Time (JIT) compiler for bytecode. It is not possible to compile an entire Java program into executable code all at once, because Java performs various run-time checks that can be done only at run time. JIT compiles the code as and when needed.

2. List and Explain the Java Buzz Words

3. Explain a simple java program with suitable examples

Example:

```
class Sampleone
{
        public static void main(String args[])
        {
          System.out.println("Welcome to JAVA");
        }
}
```

Description:

(1) **Class declaration:** "class sampleone" declares a class, which is an object-oriented construct. Sampleone is a Java identifier that specifies the name of the class to be defined.
(2) **Opening braces:** Every class definition of Java starts with opening braces and ends with matching one.
(3) **The main line:** the line " public static void main(String args[]) " defines a method name main. Java application program must include this main. This is the starting point of the interpreter from where it starts executing. A Java program can have any number of classes but only one class will have the main method.
(4) **Public:** This key word is an access specifier that declares the main method as unprotected and therefore making it accessible to the all other classes.
(5) **Static:** Static keyword defines the method as one that belongs to the entire class and not for a particular object of the class. The main must always be declared as static.
(6) **Void:** the type modifier void specifies that the method main does not return any value.
(7) **The println:** It is a method of the object out of system class. It is similar to the printf or cout of c or c++.

Java Program structure: Java program structure contains six stages.
They are:

(1) Documentation section: The documentation section contains a set of comment lines describing about the program.

(2) Package statement: The first statement allowed in a Java file is a package statement. This statement declares a package name and informs the compiler that the class defined here belong to the package.

(3) Import statements: Import statements instruct the compiler to load the specific class belongs to the mentioned package.

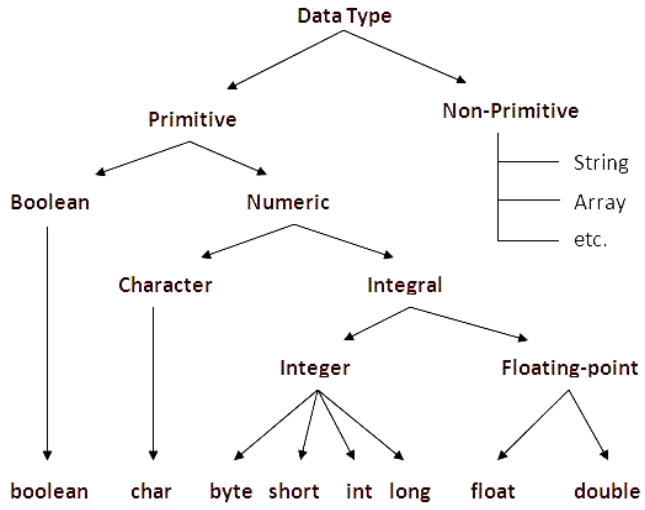
(4) Interface statements: An interface is like a class but includes a group of method deceleration. This is an optional statement.

(5) Class definition: A Java program may contain multiple class definition The class are used to map the real world object.

(6) Main method class: The main method creates objects of various classes and establish communication between them. On reaching to the end of main the program terminates and the control goes back to operating system.

---

4. List and explain the java primitive data types with suitable examples



Primitive Data Types

A primitive data type specifies the size and type of variable values, and it has no additional methods.

There are eight primitive data types in Java:

Data Type	Size	Description
byte	1 byte	Stores whole numbers from -128 to 127
short	2 bytes	Stores whole numbers from -32,768 to 32,767
int	4 bytes	Stores whole numbers from -2,147,483,648 to 2,147,483,647
long	8 bytes	Stores whole numbers from -9,223,372,036,854,775,808 to 9,223.372,036,854,775,808
float	4 bytes	Stores fractional numbers from 3.4e−038 to 3.4e+038. Sufficient for storing 6 to 7 decimal digits
double	8 bytes	Stores fractional numbers from 1.7e−308 to 1.7e+038. Sufficient for storing 15 decimal digits
boolean	1 byte	Stores true or false values
char	2 bytes	Stores a single character/letter

Numbers

Primitive number types are divided into two groups:

Integer types stores whole numbers, positive or negative (such as 123 or -456), without decimals. Valid types are `byte`, `short`, `int` and `long`. Which type you should use, depends on the numeric value.

Floating point types represents numbers with a fractional part, containing one or more decimals. There are two types: `float` and `double`.

Even though there are many numeric types in Java, the most used for numbers are `int` (for whole numbers) and `double` (for floating point numbers). However, we will describe them all as you continue to read.

Integer Types

Byte

The `byte` data type can store whole numbers from -128 and 127. This can be used instead of `int` or other integer types to save memory when you are certain that the value will be within -128 and 127:

Example

```
byte myNum = 100;
System.out.println(myNum);
```

Short

The `short` data type can store whole numbers from -32768 to 32767:

Example

```
short myNum = 5000;
System.out.println(myNum);
```

Int

The `int` data type can store whole numbers from -2147483648 to 2147483647. In general, and in our tutorial, the `int` data type is the preferred data type when we create variables with a numeric value.

Example

```
int myNum = 100000;
System.out.println(myNum);
```

Long

The `long` data type can store whole numbers from -9223372036854775808 to 9223372036854775808. This is used when int is not large enough to store the value. Note that you should end the value with an "L":

Example

```
long myNum = 15000000000L;
System.out.println(myNum);
```

Floating Point Types

You should use a floating point type whenever you need a number with a decimal, such as 9.99 or 3.14515.

Float

The `float` data type can store fractional numbers from 3.4e−038 to 3.4e+038. Note that you

should end the value with an "f":

Example

```
float myNum = 5.75f;
System.out.println(myNum);
```

Double

The `double` data type can store fractional numbers from 1.7e−308 to 1.7e+038. Note that you should end the value with a "d":

Example

```
double myNum = 19.99d;
System.out.println(myNum);
```

Use `float` or `double`?

The **precision** of a floating point value indicates how many digits the value can have after the decimal point. The precision of `float` is only six or seven decimal digits, while `double` variables have a precision of about 15 digits. Therefore it is safer to use `double` for most calculations.

Scientific Numbers

A floating point number can also be a scientific number with an "e" to indicate the power of 10:

Example

```
float f1 = 35e3f;
double d1 = 12E4d;
System.out.println(f1);
System.out.println(d1);
```

Booleans

A boolean data type is declared with the `boolean` keyword and can only take the values `true` or `false`:

Example

```
boolean isJavaFun = true;
boolean isFishTasty = false;
System.out.println(isJavaFun);     // Outputs true
System.out.println(isFishTasty);   // Outputs false
```

Boolean values are mostly used for conditional testing, which you will learn more about in a later chapter.

Characters

The `char` data type is used to store a **single** character. A char value must be surrounded by

single quotes, like 'A' or 'c':

Example

```
char myGrade = 'B';
System.out.println(myGrade);
```

Strings

The `String` data type is used to store a sequence of characters (text). String values must be surrounded by double quotes:

Example

```
String greeting = "Hello World";
System.out.println(greeting);
```

The String data type is so much used and integrated in Java, that some call it "the special **ninth** type".

A String in Java is actually a **non-primitive** data type, because it refers to an object. The String object has methods that is used to perform certain operations on strings. **Don't worry if you don't understand the term "object" just yet**. We will learn more about strings and objects in a later chapter.

5. Define a variable and explain how variables are used in Java

6. Explain Type casting in Java

Type Casting: It is often necessary to store a value of one type into the variable of another type. In these situations the value that to be stored should be casted to destination type. Type casting can be done in two ways.

Type Casting

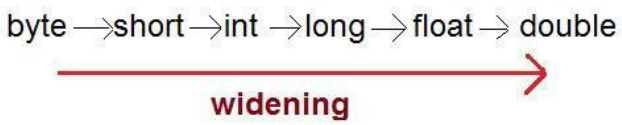
Assigning a value of one type to a variable of another type is known as **Type Casting**.

Example :

```
int x = 10;
byte y = (byte)x;
```

In Java, type casting is classified into two types,

- Widening Casting(Implicit)



- Narrowing Casting(Explicitly done)

Widening or Automatic type converion

Automatic Type casting take place when,

- the two types are compatible
- the target type is larger than the source type

Example :

```
public class Test
{
      public static void main(String[] args)
      {
         int i = 100;
         long l = i;              //no explicit type casting required
         float f = l; //no explicit type casting required
         System.out.println("Int value "+i); System.out.println("Long
         value "+l); System.out.println("Float value "+f);
      }

}
```

Output :

```
Int value 100
Long value 100
Float value 100.0
```

Narrowing or Explicit type conversion

When you are assigning a larger type value to a variable of smaller type, then you need to perform explicit type casting.

Example :

```
public class Test
{
      public static void main(String[] args)
      {
         double d = 100.04;
```

```
        long l = (long)d; //explicit type casting required int i =
        (int)l; //explicit type casting required

        System.out.println("Double value "+d);
        System.out.println("Long value "+l);
        System.out.println("Int value "+i);

    }

}
```

Output :

```
Double value 100.04
Long value 100
Int value 100
```

7. Explain switch with suitable examples

Java Switch Statements

Use the `switch` statement to select one of many code blocks to be executed.

Syntax

```
switch(expression) {
  case x:
    // code block
    break;
  case y:
    // code block
    break;
  default:
    // code block
}
```

This is how it works:

- The `switch` expression is evaluated once.
- The value of the expression is compared with the values of each case.
- If there is a match, the associated block of code is executed.
- The `break` and `default` keywords are optional, and will be described later in this chapter

The example below uses the weekday number to calculate the weekday name:

Example

```
int day = 4;
switch (day) {
  case 1:
    System.out.println("Monday");
    break;
  case 2:
    System.out.println("Tuesday");
    break;
  case 3:
    System.out.println("Wednesday");
    break;
  case 4:
    System.out.println("Thursday");
    break;
  case 5:
    System.out.println("Friday");
    break;
  case 6:
    System.out.println("Saturday");
    break;
  case 7:
    System.out.println("Sunday");
    break;
}
// Outputs "Thursday" (day 4)
```

8. Explain ternary Operators with suitable examples

If...Else (Ternary Operator)

If you have only one statement to execute, one for `if`, and one for `else`, you can put it all on the same line:

Syntax

variable = (*condition*) ? *expressionTrue* : *expressionFalse*;

Instead of writing:

Example

```
int time = 20;
if (time < 18) {
  System.out.println("Good day.");
} else {
```

```
  System.out.println("Good evening.");
}
```

You can simply write:

Example

```
int time = 20;
String result = (time < 18) ? "Good day." : "Good evening.";
System.out.println(result);
```

---

9. Explain for-each with suitable examples

For-Each Loop

There is also a "**for-each**" loop, which is used exclusively to loop through elements in an **array**:

Syntax

```
for (type variable : arrayname) {
  // code block to be executed
}
```

The following example outputs all elements in the **cars** array, using a "**for-each**" loop:

Example

```
String[] cars = {"Volvo", "BMW", "Ford", "Mazda"};
for (String i : cars) {
  System.out.println(i);
}
```

10. List and explain the java operators

Java Operators

Operators are used to perform operations on variables and values.

The value is called an operand, while the operation (to be performed between the two operands) is defined by an **operator**:

Operand	Operator	Operand
100	+	50

In the example below, the numbers 100 and 50 are **operands**, and the + sign is an **operator**:

Example

```
int x = 100 + 50;
```

Although the + operator is often used to add together two values, like in the example above, it can also be used to add together a variable and a value, or a variable and a variable:

Example

```
int sum1 = 100 + 50;        // 150 (100 + 50)
int sum2 = sum1 + 250;      // 400 (150 + 250)
int sum3 = sum2 + sum2;     // 800 (400 + 400)
```

Java divides the operators into the following groups:

1. Arithmetic operators
2. Assignment operators
3. Relational operators
4. Logical operators
5. Bitwise operators

---

1. Arithmetic Operators

Arithmetic operators are used to perform common mathematical operations.

Operator	Name	Description	Example
+	Addition	Adds together two values	x + y
-	Subtraction	Subtracts one value from another	x - y
*	Multiplication	Multiplies two values	x * y
/	Division	Divides one value from another	x / y
%	Modulus	Returns the division remainder	x % y
++	Increment	Increases the value of a variable by 1	++x
--	Decrement	Decreases the value of a variable by 1	--x

---

2. Java Assignment Operators

Assignment operators are used to assign values to variables.

In the example below, we use the **assignment** operator (=) to assign the value **10** to a variable called **x**:

Example

```
int x = 10;
```

The **addition assignment** operator (+=) adds a value to a variable:

Example

```
int x = 10;
x += 5;
```

A list of all assignment operators:

Operator	Example	Same As	Try it
=	x = 5	x = 5	
+=	x += 3	x = x + 3	
-=	x -= 3	x = x - 3	
*=	x *= 3	x = x * 3	
/=	x /= 3	x = x / 3	
%=	x %= 3	x = x % 3	
&=	x &= 3	x = x & 3	
|=	x |= 3	x = x | 3	
^=	x ^= 3	x = x ^ 3	
>>=	x >>= 3	x = x >> 3	
<<=	x <<= 3	x = x << 3	

---

3. Java Relational Operators

Relational operators are used to compare two values:

Operator	Name	Example	Try it
==	Equal to	x == y	
!=	Not equal	x != y	
>	Greater than	x > y	
<	Less than	x < y	
>=	Greater than or equal to	x >= y	
<=	Less than or equal to	x <= y	

---

4. Java Logical Operators

Logical operators are used to determine the logic between variables or values:

Boolean Logical Operators

The Boolean logical operators shown here operate only on **boolean** operands. All of the binary logical operators combine two **boolean** values to form a resultant **boolean** value.

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

The logical Boolean operators, &, |, and ^, operate on **boolean** values in the same way that they operate on the bits of an integer. The logical ! operator inverts the Boolean state: **!true == false** and **!false == true**. The following table shows the effect of each logical operation:

A	B	A | B	A & B	A ^ B	!A
False	False	False	False	False	True
True	False	True	False	True	False
False	True	True	False	True	True
True	True	True	True	False	False

|| Logical or Returns true if one of the statements is true x < 5 || x < 4 ! Logical not Reverse the result, returns false if the result is true !(x < 5 && x < 10)

```
public class MyClass
{
  public static void main(String[] args)
     {
    int x = 5;
    System.out.println(x > 3 || x < 4); // returns true because
one of the conditions are true (5 is greater than 3, but 5 is not
less than 4)
  }
}
```

```
public class MyClass {
  public static void main(String[] args) {
    int x = 5;
    System.out.println(!(x > 3 && x < 10)); // returns false
because ! (not) is used to reverse the result
```

```
  }
}
```

```
public class MyClass {
  public static void main(String[] args) {
    int x = 5;
    System.out.println(x > 3 || x < 4); // returns true because
one of the conditions are true (5 is greater than 3, but 5 is not
less than 4)
  }
}
```

Bitwise Operator

/* refer Class Notes*/

11. Explain single dimensional Arrays in Java with suitable examples

Arrays in Java

Array which stores a fixed-size sequential collection of elements of the same type. An array is used to store a collection of data, but it is often more useful to think of an array as a collection of variables of the same type.

Declaring Array 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; or dataType arrayRefVar[];

Example:

The following code snippets are examples of this syntax:

int[] myList; or int myList[];

Creating Arrays:

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};

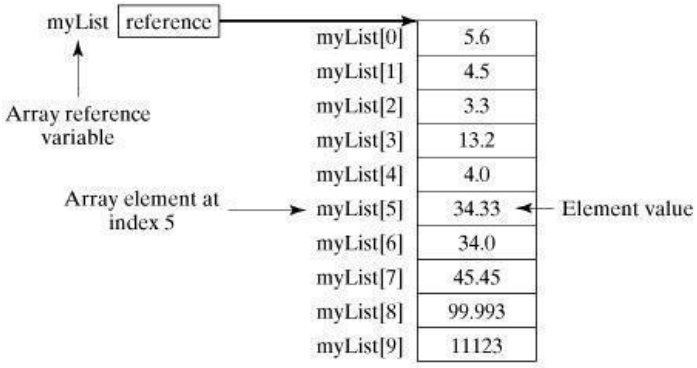
The array elements are accessed through the **index**. Array indices are 0-based; that is, they start from 0 to **arrayRefVar.length-1**.

Example:

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.



Processing Arrays:

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.

Example:

Here is a complete example of showing how to create, initialize and process arrays:

```
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] + " ");
}
```

---

12. Explain Arrays using for-each

There is also a "**for-each**" loop, which is used exclusively to loop through elements in arrays:

Syntax

for (*type variable* : *arrayname*)

{

...

}

The following example outputs all elements in the **cars** array, using a "**for-each**" loop:

Example

```
String[] cars = {"Volvo", "BMW", "Ford", "Mazda"};
for (String i : cars) {
  System.out.println(i);
}
```

The example above can be read like this: **for each** `String` element (called **i** - as in **i**ndex) in **cars**, print out the value of **i**.

If you compare the `for` loop and **for-each** loop, you will see that the **for-each** method is easier to write, it does not require a counter (using the length property), and it is more readable.

Question Bank

IA – 1, March 2019

1. List the difference between procedure oriented programming and object oriented programming
2. Define function overloading.
3. Define reference variable. Explain returning by reference in C++ using suitable examples.
4. Write short notes on encapsulation and inheritance.
5. Briefly explain the concept of bytecode in java.
6. Explain the structure of a simple Java program with the necessary keywords using suitable examples
7. Explain the following operators with suitable examples: &, &&, >>>, >>, ^
8. Explain the concepts of arrays in Java with examples.
9. **E**xplain type casting in Java using suitable examples.

** Java Programs*