Operators and Assignments

15) Determine the result of applying any operator including assignment operators and instanceof to operands of any type class scope or accessibility or any combination of these.

Result of applying "+" operator to any combination of variables or constants of any type:

For a + expression with 2 operands of primitive numeric type the result

- Is of a primitive numeric type
- Is atleast int, because of normal promotions.
- Is of a type atleast as wide as the wider type of the 2 operands.
- Has a value calculated by promoting the operands to the result type, then performing the addition calculation using that type. This might result in overflow or loss of precision

If you add two primitives a lower type is cast to a higher type in the following hierarchy:(implicit type casting)

```
                               −−−−−−−char
                               |
                               V
double    float    long    int                  short    byte
```

For a + expression with any operand that is not of primitive numeric type :

- one operand must be a string object, otherwise the expression is illegal
- If one operand is a String object and the other is not, the non−string object is converted to a string and result is concatenated.

```
public class concat{
public static void main(Strng s[]) {
String a="da";
int i = 20;
String b=a+i;
System.out.println(b); //prints "da20"
}
}
```

Remember : Using + operator on objects other than numbers and Strings would result in a compiler error. If either of the operand of a + expression is a string object, the meaning of the operator is changed from numeric addition to concatenation of text.

eg,

```
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);
```

- In the first subexpression, f * b, b is promoted to a float and the result of the subexpression is float.
- Next, in the subexpression i /c, c is promoted to int, and the result is of type int.
- Then, in d * s, the value of s is promoted to double, and the type of the subexpression is double.
- Finally, these three intermediate values, float, int, and double, are considered. The outcome of float plus an int is a float.
- Then the resultant float minus the last double is promoted to double, which is the type for the final result of the expression.

another eg,

```
float f=4.6;//give compilation error
float=4 ;//compiles fine
```

- The reason for the first one not compiling is 4.6 is treated as double and you should cast it explicitly to convert double to float. (4.6 is double and 4.6f is float)
- The reason for the second one getting compiled is 4 is treated as an integer, so assigning an int to float don't need an explicit casting.

another eg,

```
short s = 8;
int i = 10;
float f = 6.5f;
doubld d = 10.5;
if(++s * i >= f/d)
System.out.println("greater");
else
System.out.println("smaller");
```

- The short "s" is promoted to an int and then incremented
- Multiplying s (which is now int) by int i doesn't need any conversion and the result of multiplication is an int.
- f has to be widened to a double before dividing. The result will be a double.
- The result of multiplication gives an int and result of division gives a double. Now, the int is converted to a double and then the two operands are compared.
- The result of comparison will always be a boolean.

Result of applying "==" operator to any two objects of any type:

Note: Read also the next objective.

Remember : When the operands on the either side of == is

- an int or any primitive types, the result will be "true"
- an Object, the result will be "false"
- of different type a compiletime error occurs.

A question from one of the mock exams which usually confuses :

```
boolean flag=false;
if(flag=true) System.out.println("true");
else System.out.println("false");
```

This will print "true" because flag is assigned a boolean value and which is what if expects.

but if "=" is used with any other type

```
String s1="abc";
String s2="abc";
if(s1=s2) System.out.println("equal;");
```

gives a compiler error saying, Incompatible type for if. can't convert java.lang.String to boolean

instanceof operator to operands :

- The instanceof operator tests the class of an object at runtime.
- The left−hand argument can be any object reference expression, usually a variable or an array element, while the right−hand operand must be a class, interface or array type.
- To see whether an object is an instanceof a class or to see if an object implements an interface: object instanceof interface.

```
public class myButton extends JButton {
public static void main(String s[]) {
JButton b=new JButton("hello");
myButton mb = new myButton();
System.out.println(b instanceof JButton);//true
System.out.println(mb instanceof JButton);//true
System.out.println(mb instanceof myButton);//true
System.out.println(b instanceof myButton);//false
```

obviously b and mb are instances of JButton and myButton respectively. mb instanceof JButton returns true because myButton extends JButton.
Finally, b instanceof myButton prints false because b is not subclass of myButton.

another eg,

```
class furniture {
class table extends furniture{}
class chair extends furniture{}
public class wood {
public static void main(String arg[]) {
furniture f=new table();
if(f instanceof table) System.out.println("table");
if(f instanceof furniture) System.out.println("furniture");
if(f instanceof chair) System.out.println("chair");
else System.out.println("I am not!");
}
}
```

<u>output :</u>

```
table
furniture
I am not!
```

<u>Remember :</u> instanceof is an operator which is used with superclasses to tell if you have a particular subclass.

---

16) Determine the result of applying the boolean equals(Object) method to objects of any combination of the classes java.lang.String, java.lang.Boolean and java.lang.Object.

```
String s1 = new String("hi");
String s2 = new String("hi");
System.out.println(s1 == s2);
```

The above code will display "false".
because, the == operator performs a shallow comparison. It just checks to see if two object references are the same. It doesn't look at the contents to see if the two are essentially equal. Here s1 and s2 are pointing to two entirely
different objects, so the test has to return false.

However, the foll. code:

```
String s1 = "hi";
String s2 = "hi";
System.out.println(s1==s2);
```

would return true. Because anytime Java encounters a literal String it gives you a new String and adds it to the
literal pool. In the above code, by the time Java gets to s2, "hi" already exists as a literal, so it uses the same object
for both object references s1 and s2. So the test has to return true.

```
String s1 = "abc" + "def";
String s2 = new String(s1);
if(s1.equals(s2)) System.out.println("equals");
if(s1==s2) System.out.println("==");
```

<u>output :</u>

```
equals
```

So, for String and Boolean, equals() is overridden to provide an equality test. For any other class that is a direct extension of Object and does not override equals(), the test will return false.

```
Boolean b1 = new Boolean(true);
Boolean b2 = new Boolean(true);

System.out.println (b1==b2); //prints "false"
System.out.println(b1.equals(b2));//prints
"true"
```

b1 and b2 are different objects but the Boolean class overrides the equals() test to return true if the two objects being
compared have the same boolean value.

- The equals method in Object compares two objects for equality; returns true if the objects point to the same area of memory, and false otherwise.
- When two strings are compared, the result is true if and only if the argument is not null and the String objects represent the same sequence of characters, and false otherwise.
- If one of the compared string is null you will get "NullPointerException" at Runtime. eg, *if(a.equals(null)) System.out*
- when two instances are created and checked for equality the result of equals wills be false.

---

17) In an expression involving the operators | & || and variables of known values state which operands are evaluated and the value of the expression.

Operator Precedence :

|
||

Short−Circuit Logical Operators : ||

The Order of Evaluation of Logic Operators :

If you use the || and &forms, rather than the | and forms of these operators, Java will not bother to evaluate the right−hand operand when the outcome of the expression can be determined by the left operand alone. This is very useful when the right−hand operand depends on the left one being true or false in order to function properly.

```
if (denom != 0 &num / denom > 10)
```

Since the short−circuit form of AND (is used, there is no risk of causing a run−time exception when denom is zero. If this line of code were written using the single version of AND, both sides would have to be evaluated, causing a run−time exception when denom is zero.

It is standard practice to use the short−circuit forms of AND and OR in cases involving Boolean logic, leaving the single−character versions exclusively for bitwise operations.

Another eg,

```
boolean b, c, d;
b = (2 > 3); // b is false
c = (3 > 2); // c is true
d = b &c;
```

When Java evaluates the expression d = b &c, it first checks whether b is true. Here b is false, so b &c must

be false
regardless of whether c is or is not true, so Java doesn't bother checking the value of c.

|| is logical or. || combines two boolean variables or expressions and returns a result that is true if either or both of its
operands are true. For instance

```
boolean b;
b = 3 > 2 || 5 7; // b is true
b = 2 > 3 || 5 7; // b is still true
b = 2 > 3 || 5 > 7; // now b is false
```

Remember :

- For &operation, if one operand is false, the result is false, without regard to the other operand
- For || operation, if one operand is true, the result is true, without regard to the other operand

Operator	***Use***	***Returns true if***
	op1 &op2	op1 and op2 are both true, conditionally evaluates op2
||	op1 || op2	either op1 or op2 is true, conditionally evaluates op2
	op1 op2	op1 and op2 are both true, always evaluates op1 and op2
|	op1 | op2	either op1 or op2 is true, always evaluates op1 and op2

Bitwise Operators :,|

	0	1
0	0	0
1	0	1

eg, 12 13

- the binary representation of 12 is 1100, and
- the binary representation of 13 is 1101.

```
1100
1101
----------
1100 ie., 12 in decimal
----------

00101010 (42)
00001111 (15)
--------------
00001010 (10)
```

```
––––––––––––––
```

|	0	1
0	0	1
1	1	1

```
1100
| 1101 ie., 13 in decimal
–––––––––
1101
–––––––––

00101010 ( 42)
| 00001111 (15)
––––––––––––––
00101111 (47)
––––––––––––––
```

The following program demonstrates the bitwise logical operators:

```
// Demonstrate the bitwise logical operators.
class BitLogic {
public static void main(String args[]) {
String binary[] = {
"0000", "0001", "0010", "0011", "0100", "0101", "0110", "0111",
"1000", "1001", "1010", "1011", "1100", "1101", "1110", "1111"
};
int a = 3; // 0 + 2 + 1 or 0011 in binary
int b = 6; // 4 + 2 + 0 or 0110 in binary
int c = a | b;
int d = a b;
int e = a ^ b;
int f = (~a b) | (a ~b);
int g = ~a 0x0f;

System.out.println(" a = " + binary[a]);
System.out.println(" b = " + binary[b]);
System.out.println(" a|b = " + binary[c]);
System.out.println(" ab= " + binary[d]);
System.out.println(" a^b = " + binary[e]);
System.out.println("~a= " + binary[f]);
System.out.println(" ~a = " + binary[g]);
}
}
```

<u>output :</u>

```
a = 0011
b = 0110
```

```
a|b = 0111
ab= 0010
a^b = 0101
~a = 0101
~a = 1100
```

Boolean Logical Operator :

```
boolean a = true;
boolean b = false;
boolean c = a | b;
boolean d = a b;
boolean e = a ^ b;
boolean f = (!a b) | (a !b);
boolean g = !a;
System.out.println(" a = " + a);
System.out.println(" b = " + b);
System.out.println(" a|b = " + c);
System.out.println(" ab= " + d);
System.out.println(" a^b = " + e);
System.out.println("!a= " + f);
System.out.println(" !a = " + g);
```

After running this program, you will see that the same logical rules apply to boolean values as they did to bits. As you can see
from the following output, the string representation of a Java boolean value is one of the literal values true or false:

ouput :

```
a = true
b = false
a|b = true
ab= false
a^b = true
!a = true
!a = false
```

bitwise shift operators :

The shift operators include left shift > , and unsigned right shift >>>.

+ ve >> :

```
00000000 00000000 00000000 11000000 (192)
00000000 00000000 00000000 01100000 (192 >> 1)
00000000 00000000 00000000 00001100 (192 >> 4)
00000000 00000000 00000000 00000001 (192 >> 7)
```

−ve >> :
2's compliment :

00000000 00000000 00000000 11000000 (192)

11111111 11111111 11111111 00111111
1
−−
11111111 11111111 11111111 01000000 (−192)
11111111 11111111 11111111 10100000 (−192 >> 1) = −192 / 2^1 = −96
11111111 11111111 11111111 11110100 (−192 >> 4) = −192 / 2^4 = −12
11111111 11111111 11111111 11111110 (−192 >> 7) = −192 / 2^7 = −1

−ve >>> :

11111111 11111111 11111111 01000000 (−192)
00001111 11111111 11111111 11110100 (−192 >>> 4)
00000001 11111111 11111111 11111110 (−192 >>> 7)

+ve

00000000 00000000 00000000 11000000 (192)
00000000 00000000 00000001 10000000 (192 00000000 00000000 00001100 00000000 (192 00000000 00000000 01100000 00000000 (192 **−ve**

11111111 11111111 11111111 01000000 (−192)
11111111 11111111 11111110 10000000 (−192 11111111 11111111 11110100 00000000 (−192 11111111 11111111 10100000 00000000 (−192 **Remember :** In simple division if you divide −1 by 2, the result will be 0. But, the result of arithmetic shift right of −1 right is −1. You can think this as the shift operation rounding down, while the divisions rounds to 0.

---

18) Determine the effect upon objects and primitive values of passing variables into methods and performing assignments or other modifying operations in that method.

Variables of built−in types are passed by value, objects are passed by reference.

- "Passing by reference" means that a referenceto (ie. the address of) the argument is passed to the method. Using the reference, the method is actually directly accessing the argument, not a copy of it. Any changes to the method makes to the parameter are made to the actual object used as the argument.

```
public class passRef {
static void passMethod(passRef p) {
System.out.println("p inside passMethod() : " + p);
}
public static void main(String arg[]) {
passRef r = new passRef();
System.out.println("r inside main() : " + r);
passMethod(r);
}
}
```

output : will look something like this.
r inside main(): passRef@7e56f2e3

p inside passMethodf(): passRef@7e56f2e3
you can see that both p and r refer to the same object.

Passing a handle is the most efficient form of argument passing, but if you don't want to modify the outside object and want changes to affect only a local copy you use "pass by value".

- "Passing by value" means that the argument's value is copied, and is passed to the method. Inside the method this copy can be modified at will, and doesn't affect the original argument.

```
public class passVal{
int i=10;
void print(int i) {
i=i+5;
System.out.println("inside print : " + i);
}
public static void main(String arg[]) {
passVal p = new passVal();
p.print(p.i);
System.out.println("inside main : " + p.i);
}
}
```

output :

```
inside print : 15
inside main : 10
```

- A method call conversion happens when you pass a value of one type as an argument to a method that expects a different type. For eg, Explicit cast needed to convert double to int.

eg,

```
float f;
double d;
f = 3.45f;
d = Math.cos(f); //passing float to method that
expects double
```

Remember :

- Passing by reference − affects the object itself and not the copy of it.
- Passing by value − affect only a local copy.
- Widening conversions are permitted; narrowing conversions are forbidden.