CST205: OBJECT ORIENTED PROGRAMMING USING JAVA

MODULE 2

B.Tech CSE

Semester III

Viswajyothi College of Engineering and Technology

Syllabus of Module 2

- **Primitive Data types –**

- Operators –

- Control Statements –

- Object Oriented Programming in Java –

- Inheritance -

T1. Herbert Schildt, Java: The Complete Reference, 8/e, Tata McGraw Hill, 2011.

Data Types

Data type specify size and type of value that can be stored. Data types are classified as:

- Integer type
- Floating point type
- Character type
- Boolean type

Data Types : Integer Type

- Java supports 4 types of integers

Type	Size	Minimum value	Maximum value
byte	1 byte	-128	127
short	2 byte	-32768	32767
int	4 byte	-2147483648	2147483647
long	8 byte	-9223372036854775808	9223372036854775807

Data Types : Floating point Type

- Java supports 2 kinds of floating point storage

Type	Size	Minimum value	Maximum value
Float	4 byte	1.4e-45	3.4e+38
double	8 byte	4.9e-324	1.8e+308

- Floating point numbers are treated as double-precision quantities. To force them to be in single-precision mode, f or F is appended to numbers.
- There are two kinds of floating-point types, **float and double,** which represent single- and double-precision numbers, respectively.
- **Width of double :** 64
- **Width of float :** 32

```
// FLOATING TYPE EG: Compute the area of a circle.
class Area
{
        public static void main(String args[])
        {
                double pi, r, a;
                r = 10.8; // radius of circle
                pi = 3.1416; // pi, approximately
                a = pi * r * r; // compute area
                System.out.println("Area of circle is " + a);
        }
}      o/p Area of circle is 366.436224
```

Data Types : Character Type

Type	Size	Minimum value	Maximum value
Char	2 byte	0	65535

- 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. Thus, in Java char is a 16-bit type.
  - ASCII character set occupies the first 127 values in the Unicode character set.
- There are no negative chars.
- Even though **char**s are not integers, in many cases you can operate on them as if they were integers. This allows you to add two characters together, or to increment the value of a character variable.

CHARACTER TYPE EG

```
class CharDemo
{
        public static void main(String args[])
        {
            char ch1, ch2;
            ch1 = 88; // code for X
            ch2 = 'Y';
            System.out.print("ch1 and ch2: ");
            System.out.println(ch1 + " " + ch2);
         }
}   o/p  ch1 and ch2: X  Y
```

```
class CharDemo2
{
    public static void main(String args[])
    {
        char ch1;
        ch1 = 'X';
        System.out.println("ch1 contains " + ch1);
        // increment ch1, the next character in the unicode sequence
        ch1++;
        System.out.println("ch1 is now " + ch1);
    }
}   o/p ch1 contains X
     ch1 is now Y
```

Data Types : Boolean Type

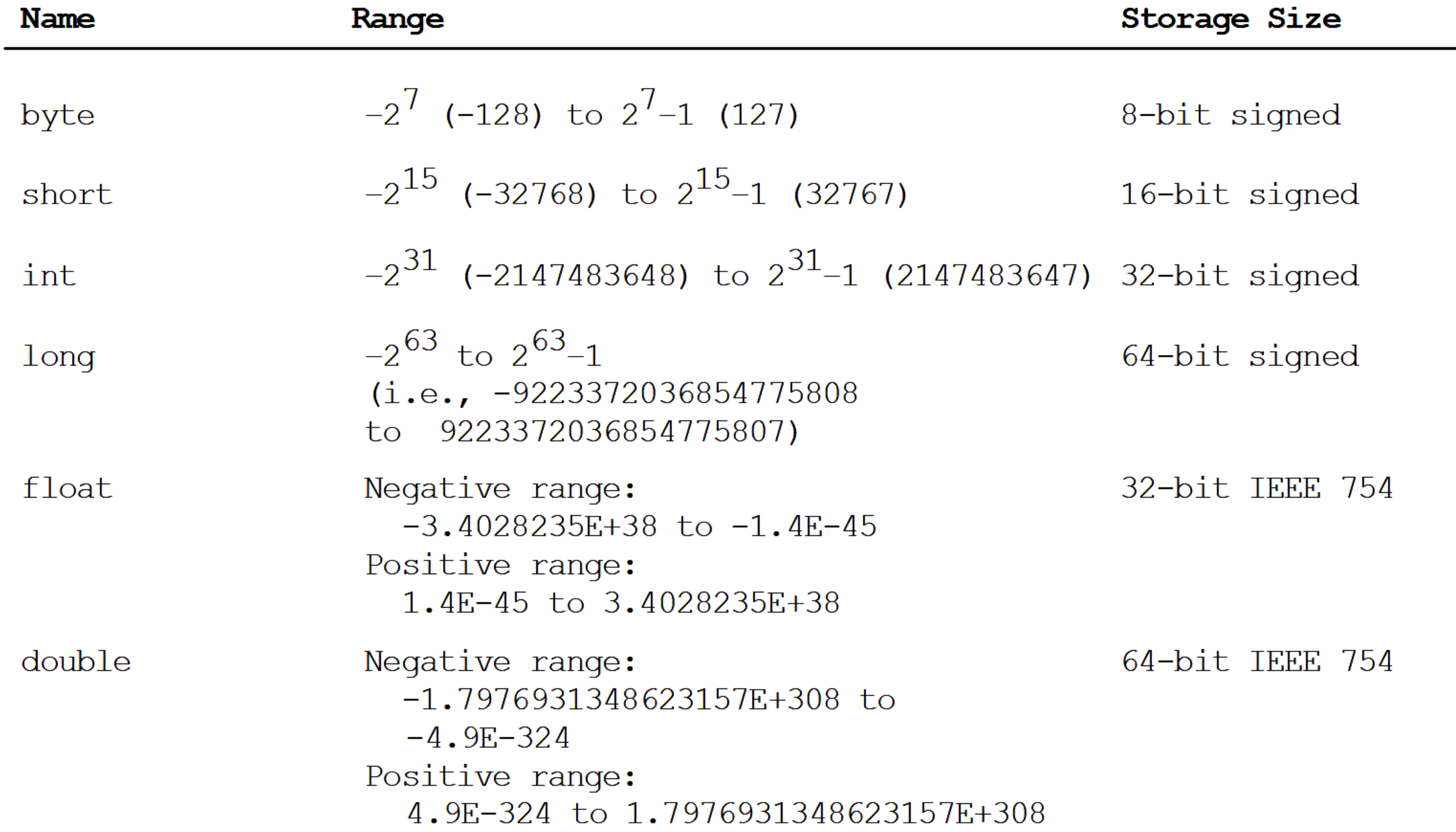
- **boolean** type is used for logical values.
- It can have only one of two possible values, **true** or **false**.
- This is the type returned by all relational operators

Data Types : Boolean Type

```
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);
if(b)
        System.out.println("This is executed.");
b = false;
if(b)
        System.out.println("This is not executed.");
System.out.println("10 > 9 is " + (10 > 9));
}
}
```

o/p

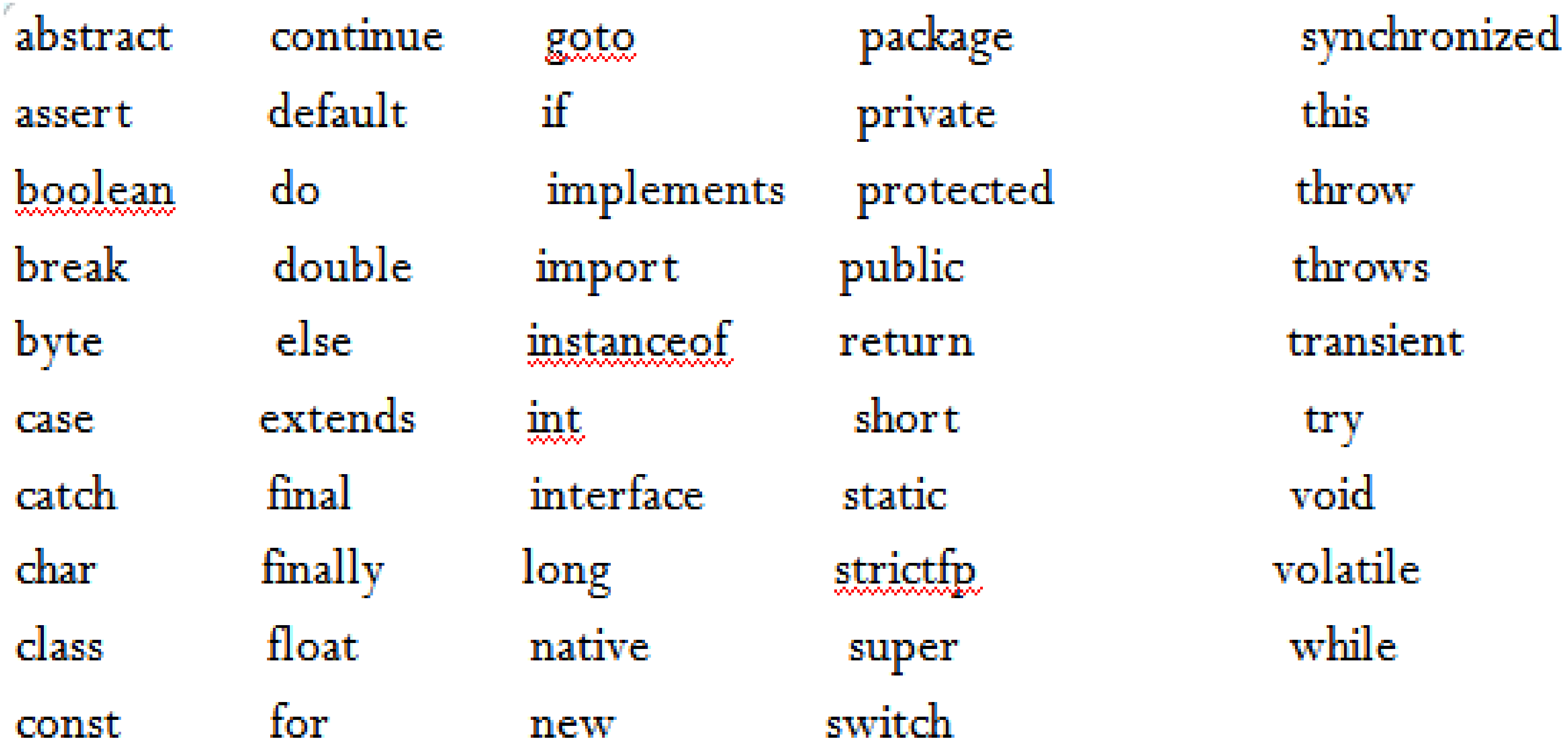
```
b is false
b is true
This is exexuted
10>9 is true
```



Name	Range	Storage Size
byte	-2^7 (-128) to 2^7-1 (127)	8-bit signed
short	-2^{15} (-32768) to $2^{15}-1$ (32767)	16-bit signed
int	-2^{31} (-2147483648) to $2^{31}-1$ (2147483647)	32-bit signed
long	-2^{63} to $2^{63}-1$ (i.e., -9223372036854775808 to 9223372036854775807)	64-bit signed
float	Negative range: -3.4028235E+38 to -1.4E-45 Positive range: 1.4E-45 to 3.4028235E+38	32-bit IEEE 754
double	Negative range: -1.7976931348623157E+308 to -4.9E-324 Positive range: 4.9E-324 to 1.7976931348623157E+308	64-bit IEEE 754

Keywords

- There are 49 reserved keywords currently defined in the Java language.
- In addition to the keywords, Java reserves the following: **true**, **false**, and **null**.
- We may not use these words for the names of variables, classes, and so on.



abstract
assert
boolean
break
byte
case
catch
char
class
const

continue
default
do
double
else
extends
final
finally
float
for

goto
if
implements
import
instanceof
int
interface
long
native
new

package
private
protected
public
return
short
static
strictfp
super
switch

synchronized
this
throw
throws
transient
try
void
volatile
while

Variables

- Basic unit of storage in a Java program.

- It is defined by the combination of an identifier, a type, and an optional initializer.

- **Variable declaration** Syntax :

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

- **Dynamic Initialization**
  - Variables are initialized dynamically(at run time)

Variables : Scope and Life time

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*.
- Within a block, variables can be declared at any point, but are valid only after they are declared.
- 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.
- If a method has parameters, they too are included within the method’s scope.

Variables : Scope and Life time

- Scopes can be nested.
  - The objects declared in the outer scope will be visible to code within the inner scope.
  - The reverse is not true.

- We cannot declare a variable to have the same name as one in an outer scope.

Example:

```
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
            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);
    }
}
```

Variables : Scope and Life time

```
// 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
          }
  }
}
```

Operators

- Operators can be divided into four groups:
  - Arithmetic
  - Bitwise
  - Relational
  - Logical
  - Assignment

Operators : Arithmetic Operators

Operator	Result
+	Addition
-	Subtraction(Also Unary Minus)
*	Multiplications
/	Division
%	Modulus
++	Increment
--	Decrement
+=	Addition Assignment
-=	Subtraction “
*=	Multiplication ,,
/=	Division ,,
%=	Modulus ,,

Operators : Arithmetic Operators

- Used in arithmetic expressions.

- The operands of the arithmetic operators must be of a numeric type or **char** types (since the **char** type in Java is a subset of **int**)

- Arithmetic operators are classified as:
  - Basic Arithmetic Operators
  - Modulus Operators
  - Arithmetic Assignment Operators
  - Increment and Decrement Operators

Arithmetic Operators : Basic Arithmetic

- Basic arithmetic operators are
  - Addition : +
  - Subtraction : -
  - Multiplication : *
  - Division : /
- When the division operator is applied to an integer type, there will be no fractional component attached to the result.

Arithmetic Operators : Modulus Operator(%)

- Returns the remainder of a division operation.
- It can be applied to floating-point types as well as integer types.
- Example:

```
 int x = 42;
double y = 42.25;
System.out.println("x mod 10 = " + x % 10);
System.out.println("y mod 10 = " + y % 10);
```

Output:

```
x mod 10 = 2
y mod 10 = 2.25
```

Arithmetic Operators : Arithmetic Compound Assignment Operators

- Arithmetic operator is combined with assignment operator.
- += , -=, *=, /=, %=
- Any statement of the form

  var = var op expression;

  can be rewritten as

  var op=expression;
- Benefits :
  - Save a bit of typing time.
  - They are implemented more efficiently by the Java run-time system than their equivalent long forms.

Arithmetic Operators : Increment and Decrement Operators

- ++, --
- Example :

```
x = x + 1;        can be rewritten as:     x++;
x = x - 1;        can be rewritten as:     x--;
```

- These operators are appear in
  - *postfix* form :

    Example:

    ```
    x = 42;
    y = x++;    //output of y : 42
    ```
  - *prefix* form :

    Example:

    ```
    x = 42;
    y = ++x;   //output of  y : 43
    ```

Operators : Bitwise Operators

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 Excusive OR Assignment
>>=	Shift right Assignment
>>>=	Shift right zero fill Assignment
<<=	Shift left Assignment

Operators : Bitwise Operators

- These operators act upon the individual bits of their operands.

- Bitwise operators can be classified as:
  - Bitwise Logical Operators
  - Left Shift Operator
  - Right Shift Operator
  - Unsigned Right Shift Operator
  - Bitwise Assignment Operators

Binary Number Representation in Java

- byte b=42;

2^7	2^6	2^5	2^4	2^3	2^2	2^1	2^0
0	0	1	0	1	0	1	0

2^1+2^3+2^5

2+8+32=42

- left most bit
  - 0 means number is positive
  - 1 means number is negative

- Negative Number Representation in Java
  - Java uses 2's complement encoding

    a. inverting 1 to 0 and vice versa

    b. add 1 to the result

Eg: -

byte b=-42;

42 is represented as 00101010

a. Inver 1s and zeros =11010101
b. Add 1 to the result 11010101+

```
                                    1
             Result= 11010110
```

- To Decode a Negative Number

a. Invert all 1s and 0s

b. Add 1 to the result

Eg:-

-42= 11010110

a. Inver all 1s and zeros= 00101001 +

b. Add 1 1

Result=00101010

Bitwise Operators : Bitwise Logical Operators

A	B	A|B	A&B	A^B	~A
0	0	0	0	0	1
1	0	1	0	1	0
0	1	1	0	1	1
1	1	1	1	0	0

```
int a = 3;
int b = 6;
int c = a | b;
int d = a & b;
int e = a ^ b;
int f = (~a & b) | (a & ~b);
System.out.println(" c : "+c+" d : "+d+" e : "+e+" f : "+f);
```

Output: ?

Bitwise Operators : Left Shift Operators

- The left shift operator, <<, shifts all of the bits in a value to the left a specified number of times.

- Syntax:

  value << num

- For each shift left, the high-order bit is shifted out and a zero is brought in on the right.

- Byte and short values are promoted to int when an expression is evaluated. Furthermore, the result of such an expression is also an int.

Example of left shift operator:

```
    byte a = 64,b;
    int i;
    i = a << 2;
    b = (byte) (a << 2);
    System.out.println("Original value of a: " + a);
    System.out.println("i and b: " + i + " " + b);
```

Output:

```
Original value of a: 64
i and b: 256 0
```

Bitwise Operators : Right Shift Operators

- The right shift operator, >>, shifts all of the bits in a value to the right a specified number of times.
- Syntax :

  value >> num
- Example : int a = 32;

  a = a >> 2; // a now contains 8
- When a number is shifting right, the top (leftmost) bits exposed by the right shift are filled in with the previous contents of the top bit. This is called *sign extension* and serves to preserve the sign of negative numbers when you shift them right.
- Example : –8 >> 1 is –4

```
11111000        –8
>>1
11111100        –4
```

Bitwise Operators : Unsigned Right Shift Operators

- Unsigned shift-right operator, >>>, always shifts zeros into the high-order bit.

Example :

```
int a=-1;
a=a>>>24;
System.out.println(a);

11111111 11111111 11111111 11111111        –1
>>>24
00000000 00000000 00000000 11111111     255
```

Bitwise Operators : Bitwise Assignment Operator

- All of the binary bitwise operators have a shorthand form
- Example :

```
a = a >> 4;      equivalent to   a >>= 4;
a = a | b;       equivalent to   a |= b;
```

Operators : Boolean Logical Operator

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

- Operate only on boolean operands.

- All of the binary logical operators combine two boolean values to form a resultant boolean value.

Boolean Logical Operator : Basic Boolean Logical Operator

- 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.

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

Boolean Logical Operator : Short circuit Operator

- The & and | operators, when used as logical operators, always evaluate both sides.

- The **&& and || operators "short-circuit",** meaning they don't evaluate the right hand side if it isn't necessary.

- Example :

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

- Example : situation in which && and || can not be used

  ```
  if(c==1 & e++ < 100) d = 100;
  ```

Here, using a single **&** ensures that the increment operation will be applied to **e** whether **c** is equal to 1 or not

- Boolean Assignment Operators
  - &=
  - |=
  - ^=
- Boolean Comparison Operators
  - ==
  - !=
- Boolean Ternary Operator ? :
- Syntax :

  expression1 **?** *expression2* **:** *expression3*

  expression2 and *expression3* are required to return the same type, which can't be **void**.
- Example :

```
int a=2,b=3,c=4,d;
d=a>b?a:b;
System.out.println(d); //output : 3
```

Operators : Relational Operator

- The *relational operators* determine the relationship that one operand has to the other.
- The outcome of these operations is a **boolean** value i.e. true or false.

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

Operators : Assignment Operator

- Syntax :

  var = expression;

  Variable must be compactable with expression

- It allows you to create a chain of assignments.

  Example :

```
int x, y, z;
x = y = z = 100; // set x, y, and z to 100
```

Operators Precedence

DOT OPERATOR

Precedence				
1	()	[]	.	
2	++	--	~	!
3	*	/	%	
4	+	-		
5	>>	>>>	<<	
6	>	>=	<	<=
7	==	!=		
8	&			
9	^			
10	|			
11	&&			
12	||			
13	?:			
14	=			

Tutorial 2:

- Q1: Write a java program to do operations addition, subtraction, multiplication and division on any two numbers specified.

- Q2 a: Predict Output of the following code segment:

```
int a=2,b=1,c=1,d;
d=a | 4 + c >> b & 7;
System.out.println(d);
```

Q2 b:

```
int xa=2;
int ya=xa++;
int za=xa;
System.out.println("xa : "+(++xa)+" ya: "+ya+" za : "+za);
```

Type Conversion and Casting

- Assign a value of one type to a variable of another type is called type conversion.

- Type conversion is classified as :
  - Automatic Type Conversion
  - Explicit Type Conversion

- 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.

- Example :
  - Integer and floating-point types are compatible with each other
  - **int** type is always large enough to hold all valid **byte** values

- Type Conversion Rules :
  - All byte and short values are promoted to **int**
  - If one operand is long then the whole expression is promoted to **long**
  - If one operand is float then the whole expression is promoted to **float**
  - If one operand is double then the whole expression is promoted to **double**

- Example :

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

- To handle this situation rewrite the above code as :

```
byte b = 50;               OR            byte b = 50;
b = (byte)(b * 2);                        int c;
                                          c = b * 2;
```

Q)

```
byte b = 42;
char c = 'a';
short s = 1024;
int i = 50000;
float f = 5.67f;
double d = .1234;
System.out.println((f * b) + (i / c) - (d * s));
```

What is the type of data displayed by the println()? Why?

Explicit Type Conversion

- To create a conversion between two incompatible types, we must use a cast.
- Syntax:

  (*target-type*) *value*

  target-type specifies the desired type to convert the specified value

- ***narrowing conversion*** *:* Explicitly making the value narrower so that it will fit into the target type
- Ex: Assign an **int** value to a **byte** variable

```
int a;
byte b;
// ...
b = (byte) a;
```

- *Truncation* : the fractional component is lost
  - Ex : floating-point value is assigned to an integer type

- If the size of the whole number component is too large to fit into the target integer type, then that value will be reduced to modulo the target type's range.

Q)

```
 byte b,c;
int i = 257,j;
double d = 323.142;
b = (byte) i;
c = (byte) d;
j = (int) d;
```

What is the value of b, c and j after executing above java codes?

Control Statements

- Control statements are used to alter the execution of statements based on certain conditions.

- Java’s program control statements can be put into the following categories:
  - Selection statements/Branching Statements
  - Iteration/Looping statements
  - Jump statements

Selection Statement

- Selection statements allows to control the flow of program's execution based upon conditions known only during run time.

- Java supports two selection statements:
  - if
    - nested if
    - if-else-if ladder
  - switch
    - nested switch

Selection Statement : if

- It can be used to route program execution through two different paths.
- Syntax :

```
if (condition)
        statement1;
else
        statement2;
```

Each *statement* may be a single statement or a compound statement enclosed in curly braces.

The *condition* is any expression that returns a **boolean** value.

The **else** clause is optional.

Selection Statement : nested if

- A *nested if* is an **if** statement that is the target of another **if** or **else**.

- Example :

```
        if(i == 10)
        {
                if(j < 20)          a = b;
                if(k > 100)         c = d;
                else                a = c;
        }
        else        a = d;
```

Selection Statement : if-else-if ladder

- Syntax :

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

Selection Statement : switch

- Multiway branch statement. Used in menu driven programming.
- Syntax :

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

- Each **case** value must be a unique literal (that is, it must be a constant, not a variable).

- Duplicate **case** values are not allowed.

- If we omit the **break**, execution will continue on into the next **case**.

Selection Statement : nested switch

- A **switch** statement inside another **switch**
- Example :

```
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: // ...
```

- Important features of the **switch** statements :
  - The **switch** differs from the **if** in that **switch** can only test for equality, whereas **if** can evaluate any type of Boolean expression.
  - No two **case** constants in the same **switch** can have identical values.
  - A **switch** statement is usually more efficient than a set of nested **if**s and is faster when lot of cases are to be considered.

- Write a java program to print integers 1,2,3,4 and 5 in words using switch

Iteration Statement

- A loop repeatedly executes the same set of instructions until a termination condition is met.

- Java’s iteration statements(loops) are
  - **while**
  - **do-while**
  - **for**

Iteration Statement : while (entry-controlled loop)

- Syntax :

```
while(condition)
{
        // body of loop
}
```

- The *condition* can be any Boolean expression.
- The body of the loop will be executed as long as the conditional expression is true.
- The body of the **while** can be empty.

- Write a program to print 10 to 0 using while loop.
- Write a program to find the mid point between i and j

Iteration Statement : do-while (exit-control loop)

- Syntax:

```
        do
        {
                // body of loop
        } while (condition);
```

- **do-while** loop always executes its body at least once

```
class DoWhile
{     public static void main(String args[])
{     int n = 10;
do {
      System.out.println( n);      n--;
      } while(n > 0);
}
}
```

Iteration Statement : for

- Syntax :

```
        for(initialization; condition; iteration)
        {
                // body
        }
```

- It is possible to declare the variable inside the initialization portion of the **for**. When we declare a variable inside a **for** loop, the scope of that variable ends when the **for** statement does.

- It is possible to include more than one statement in the initialization and iteration portions of the **for** loop.

- Write a program to find the factorial of a number.

- **for Loop Variations:**
  - The condition controlling the **for** can be any Boolean expression.
    - Example : boolean done = false;

```
for(int i=1; !done; i++)
{ ----------
}
```

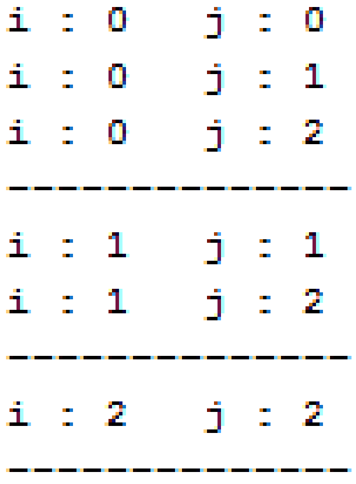
- Either the initialization or the iteration expression or both may be absent
- We can create an infinite loop by leaving all three parts of the **for** empty.

- **Nested Loops:**
  - one loop may be inside another

```
class Nested
{     public static void main(String args[])
      {
            for(i=0; i<3; i++)
            {
                  for(j=i; j<3; j++)
                        System.out.println("i : "+i+"\tj : "+j);
                  System.out.println("--------------");


             }
}
```

//output



```
i : 0    j : 0
i : 0    j : 1
i : 0    j : 2
----------------
i : 1    j : 1
i : 1    j : 2
----------------
i : 2    j : 2
----------------
```

Jump Statement

- Transfer control to another part of the program.

- Java supports three jump statements:
  - break
  - continue
  - return

Jump Statement : break

- **break** statement has three uses.
  - It terminates a statement sequence in a **switch** statement
  - It can be used to exit a loop
    - When used inside a set of nested loops, the **break** statement will only break out of the innermost loop
  - It can be used as a “civilized” form of goto.
    - Syntax : break *label*;
    - *label* is the name of a label that identifies a block of code
    - When executing a **break** statement, control is transferred out of the named block of code. The labeled block of code must enclose the **break** statement, but it does not need to be the immediately enclosing block.

```
class Break
{     public static void main(String args[])
      {     boolean t = true;
            first:
            {     second:
                  {     third:
                        {
                              System.out.println("Before the break.");
                              if(t) break second; // break out of second block
                              System.out.println("This won't execute");
                        }
                        System.out.println("This won't execute");
                  }
                  System.out.println("This is after second block.");
            }
      }
}
//output:
Before the break.
This is after second block.
```

Jump Statement : continue

- Continue running the loop, but stop processing the remainder of the code in its body for this particular iteration.
  - In **while** and **do-while** loops, a **continue** statement causes control to be transferred directly to the conditional expression that controls the loop.
  - In a **for** loop, control goes first to the iteration portion of the **for** statement and then to the conditional expression.

- Q) Write a program to print all even numbers between 0 and 20 using for loop and continue statement

- **continue** may specify a label to describe which enclosing loop to continue

```
class ContinueLabel
{     public static void main(String args[])
{
    outer:
            for(int i=0;i<4;i++)
            {
                for(int j=0;j<4;j++)
                {
                    if(j==2)
                        continue outer;
                    System.out.println("i : "+i+" j : "+j);
                }
            }
}
}
```

Jump Statement : return

- Used to explicitly return from a method.
- Program control to transfer back to the caller of the method.
- Immediately terminates the method in which it is executed

```
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.");
}
}
```

Reading Input from user

- A Java program can obtain input from the console through the keyboard.
- The Java system variable System.in represents the keyboard.
- The Java programming language provides a collection of methods stored in the Scanner class that perform read operations.
- The Java program must first import the containing class using

  import java.util.Scanner;
- Then a Scanner object is constructed using the following statement:

  Scanner in = new Scanner(System.in);
- Different methods that can be invoked using scanner object are:

  in.nextByte(), in.nextShort(), in.nextInt(), in.nextLong(), in.nextFloat(), in.nextDouble(), in.nextLine()

Sample program

```
import java.util.Scanner;
Import java.io.*;
class Sampleapp {
public static void main(String[] args) {
      int num;
      float fnum;
      String str;
      Scanner in = new Scanner(System.in);
      System.out.println("Enter a string: ");
      str = in.nextLine();      //read i/p string
      System.out.println("Input String is: "+str);
       System.out.println("Enter an integer: ");
      num = in.nextInt();    //read i/p integer no
      System.out.println("Input Integer is: "+num);
      System.out.println("Enter a float number: ");
      fnum = in.nextFloat();  //read  i/p float number
      System.out.println("Input Float number is: "+fnum);
      }
 }
```

Assignment 1(Date of submission :27/2/17)

- **Set 1(Roll 1-20)**

1. Design a use case diagram for a Hospital management system.
2. Write a program to display Armstrong numbers in an interval in java.

- **Set 2 (Roll:21-40)**

1. Design a class diagram for a Railway reservation system.
2. Write a program to display prime numbers in an interval in java.

- **Set 3 (Roll:41-59)**

1. Design a class diagram for Course registration system.
2. Write a menu driven program to implement a calculator in java.