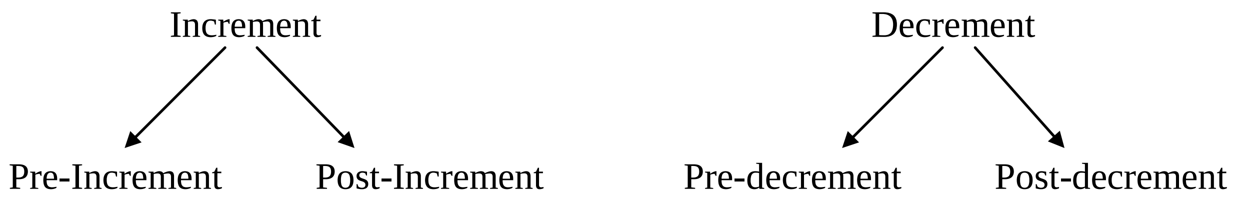
Operators And Assignments

- **Increment /Decrement**
- **Arithmetic Operators**
- **Concatenation**
- **Relational Operators**
- **Equality Operators**
- **Bitwise Operators**
- **Short-Circuit**
- **instanceOf**
- **Typecast Operator**
- **Assignment Operator**
- **Conditional Operator**
- **new Operator**
- **[ ] Operator**
- **Operator Precedence**
- **Evaluation Order of Java Operands**

Increment And Decrement Operator



Increment

Pre-Increment Post-Increment

Decrement

Pre-decrement Post-decrement

int x= ++y; int x= y++; int x= --y; int x= y--;

Expression	Initial value of x	Final value of x	Final value of y
y= ++x;	4	5	5
y= x++;	4	5	4
y= --x;	4	3	3
y= x--;	4	3	4

- We can apply increment and decrement only for variables but not for constant values.
  Example:

```
1) int x=4;
2) int y = ++4;  // C.E. unexpected type
                    required: variable
                    found: value
```

- Nesting of increment and decrement operators is not allowed otherwise we will get compile time error.
  Example:

```
1)  int x = 4;
2)   int y = ++(++x);          //C.E. unexpected type
     System.out.println(y);          required: variable
                                     found: value
```

- We can't apply increment and decrement operators for the final variables.
  Example:

```
1) final int x = 4;
   x++;                 C.E. Can,t assign a value to final
                                variable x.
2) final int x=4;
   x=5;
```

- We can apply increment and decrement operators for every primitive data type except Boolean.
  Example:

```
1) double d = 10.5;
   d++;
   System.out.println(d);  //Output= 11.5

2) char ch = 'a';
   ch++;
   System.out.println(ch); //  Output= b

3) boolean b = true;
   ++b;
   System.out.println(b);  //C.E.  operator ++ can't applied to
                                  boolean.

4) int x = 10;
   x++;
    System.out.println(x); //  Output= 11
```

- Difference between b++ and b=b+1:

  Example:

```
1) byte b = 10;
   b++;
   System.out.println(b); //Output=11

2) byte b = 10;
   b = b+1;                    //C.E. –possible loss of precision
   System.out.println(b);                required: byte
                                          found: int
3) byte b = 10;
   b = (byte)b+1;
   System.out.println(b); // Output= 11

4) byte a = 10;
   byte b = 20;
   byte c = a+b;                   // C.E. - possible loss of precision
   System.out.println(c);                 required: byte
                                           found: int
```

- Whenever we are performing any arithmetic operations (+, -, *, %, /) between two variables a and b, the result type is always,

  Max(int, type of a, type of b)

```
byte b = 10;
b = (byte)(b+1);
System.out.println(b); // output=11
```

- In the case of increment and decrement operators the required type casting (internal type casting) automatically performed by the compiler.

```
byte b++;  ⟶    b = (byte) (b+1);

b++;    ⟶ b = (typeof b) (b+1);
```

Arithmetic Operators

- The arithmetic operators are (+, -,*, /, %)
- If we are applying any arithmetic operators between two variables a and b, the result type is always,

Max(int, type of a, type of b)

byte+ byte = int
byte+short = int
int + long = long
long + float = float
double + char= double
char + char=int

Infinity:

- In case of the integral arithmetic (int, short, long, byte) there is no way to represent infinity. Hence, if the infinity is the result we will always get ArithmeticException(AE:/ by zero)
  Example:
  System.out.println(10/0); //R.E. (AE:/ by zero)

- But in case of floating point arithmetic (float and double) there is always a way to represent infinity. For this float and double classes contains the following two constants.
  Positive –Infinity = Infinity
  Negative –Infinity = -Infinity
- Hence, in the case of floating point arithmetic we won't get any ArithmeticException.

Example:

```
1)  System.out.println(10/0.0);  //Infinity
2)  System.out.println(-10/0.0);  //-Infinity
```

Not as Number(NaN):

- In integral arithmetic (int, short, long, byte) there is no way to represent undefined results. Hence, if the result is undefined we will get ArithmeticException in case of integral Arithmetic.

  Example:

  ```
  System.out.println(0/0);  // R.E.  (AE:/ by zero)
  ```

- But in case of floating point arithmetic, there is a way to represent undefined results for this float and double classes contain NaN Constance.
- Hence, even though the result is undefined we won't get any RuntimeException in floating point Arithmetic.

  Example:

  ```
  1)   System.out.println(0/0.0);  // NaN
       System.out.println(0.0/0);  // NaN
       System.out.println(-0/0.0);  // NaN
  ```

  Example:

  ```
  2)  Public static double sqrt(double d);
      System.out.println(math.sqrt(4));  //Output=2.0
      System.out.println(math.sqrt(-4)) //NaN
  ```

- For any x value including NaN the below expressions always returns false, except the (! =) expression returns "true".

```
X != NaN   // True
```

```
 at  x = 10;

System.out.println(10>Float . NaN);   // false

System.out.println(10<Float . NaN);   // false

System.out.println(10 = =Float . NaN);   // false

System.out.println(10! =Float . NaN);   // true

System.out.println(Float.NaN = = Float . NaN);   // false

System.out.println(Float . NaN! = Float. NaN);   // true
```

Conclusion about ArithmeticException:

- It is a RuntimeException but not compile time error.
- Possible only in Integral Arithmetic(int, byte, short, char) but not floating point Arithmetic(float, double).
- The only operators which cause ArithmeticException are '/ and %'.

String Concatenation Operator (+)

- The only overloaded operator in java is '+' operator.
- Sometimes it acts as arithmetic addition operator and sometimes acts as string arithmetic operator or string concatenation operator.

Example: int a = 10, b=20, c=30;

```
String  d = "Priyanka";

System.out.println (a+b+c+d);  //Output=60Priyanka

System.out.println (a+b+d+a); // Output=30Priyanka30

System.out.println (d+a+b+c); // Output=Priyanka2030
```

System.out.println (a+d+b+c); // Output=10Priyanka2030

- If at least one operand is string type then '+' operator acts as concatenation otherwise '+' acts as arithmetic operator. Here System.out.println() is evaluated from left to right.

  Example:

```
int a = 10, b=20;

String  c = "Priyanka";

a = b+c;   // C.E. –incompatible type

                      required: int

                      found:string

c=a+c;   //Valid

b=a+b;  //Valid

c = a+b;   // C.E. –incompatible type

                      required: int

                      found:string
```

Relational Operator

- The relational operators are (>, < ,>=, <=)
- We can apply relational operator for every primitive data type except boolean.

  Example:

  1) 10>20; //false
  2) 'a'< 'b'; // true
  3) 10>=10.0; //true
  4) 'a'< 125; //true

5) true<=true; } C.E. operator<=can't be applied to
6) true<false; } boolean, boolean.

- We can't apply relational operators for the object type.
  Example:
  1) "Priyanka"< "Priyanka"; } C.E. operator<=can't be
  2) "snehal" < "Snehal123"; } applied to String, string.

- Nesting of relational operators we are not allow to apply.

  Example:
  System.out.println(10<20); //Valid
  System.out.println(10<20<30) //Invalid C.E.
  Operator < can't be
  applied to boolean.

  Example:
  String s1 = new String("durga");
  String s1 = new String("durga");
  System.out.println(s1= = s2); // false
  System.out.println(s1.equals(s2)); // true

Equality Operators (= =, ! =)

- These are = =, ! =
- We can apply equality operators for every primitive type including boolean type.
  Example:
  1) 10 == 10.0 //true
  2) 'a' == 97 // true
  3) True = = false //false
  4) 10.5 = =12.3 // false
- We can apply equality operators even for object reference also.

- For the two object references and r1 and r2 and r1= = r2 returns true iff both r1 and r2 are pointing to the same object. i.e. equality operators (= =) is always meant for reference / address comparison.

  Example:

  ```
  1) Thread t1= new Thread ();
     Thread t2= new Thread ();
     Thread t3= t1;
     System.out.println(t1= =t2); // false
     System.out.println(t1==t3); //true
  ```

- To apply equality operators between the object references compulsory these should be some relationship between argument types.

  [Either parent to child or child to parent or same type] otherwise we will get C.E- Incompatible type.

  Example:

  ```
   Object o1=new object ();
    Thread t1= new Thread ();
  String s1 = new String (“Sujata”);
   System.out.println(t1= =s1); //C.E. Incompatible type
                                      java.lang.Thread and
                                       java.lang.String
    System.out.println(t1= =o1); // false
       System.out.println(s1= =o1); // false
  ```

- For any object reference r, if r is pointing to any object

  r= = null is always, false

  Otherwise r contains null value. So,

  null= = null is always, true

Note:

- In general , == operator meant for reference comparison where as .equal() method means for content comparision.

instanceOf Operators

- By using this operator we can check whether the given object is of a particular type or not.
- Syntax:

```
            r   instanceOf  x
any reference type            class/interface
```

Example:

```
        Short s=15;
        boolean b;
         b=(s instanceOf Short); // true
         b=(s instanceOf Number);// false
```

Example:

```
1)  Thread t =new Thread();
    System.out.println(t instanceOf Thread); // true
    System.out.println(t instanceOf Object); // true
    System.out.println(t instanceOf Runnable); // true
```

- To use instanceOf operator, compulsory there should be some relationship between argument type, otherwise we will get compile time error saying Inconvertable type.
  Example:

```
  Thread t =new Thread();

System.out.println(t instanceOf  String);// C.E.
                                    Inconvertable type
                                    required: String
                                    found: Thread
```

- Whenever we are checking parent object is of child type then, we will get false as output.
  Example:
  ```
          Object o = new Integer(10);
          System.out.println(o instanceOf  String);// false
  ```

- For any class or interface x, null instanceOf x always return "false".
  System.out.println(t instanceOf String);//false

  Example:
  ```
           Iterator itr = l.iterator();
            while(itr.hasnext())
         {
            Object o=itr.next();
             if(o instanceOf Student)
            {
              System.out.println("I am a student");
             }
          elseif(o instanceOf Customer)
             {
              System.out.println("I am a Customer");
             }
           }
  ```

Bitwise Operator

- & - AND → if both operands are True then result is True.
- | - AND → if at least one operands True then result is True.
- ^ - AND → if both operands are different then result is True.

  Example:

  ```
  System.out.println(T & T); //true
  System.out.println(T | T); //true
  System.out.println(T ^ T); //true
  ```

  Example:

  ```
  System.out.println(4 & 5); //4
  System.out.println(4| 5); //5
  System.out.println(4 ^ 5); //1
  ```

- We can apply these operators even for integral data type also.
  Example:

  ```
  System.out.println(4 & 5); //4
  System.out.println(4| 5); //5
  System.out.println(4 ^ 5); //1
  ```

Bitwise Complement Operator

Example:

```
System.out.println(~T);//C.E.operators ~ can’t be applied to boolean
```

- We can apply Bitwise Complement Operator only for integral types but not for Boolean type.

  Example:

  ```
  1)  System.out.println(~True); //C.E.operators ~ can’t be
                                        applied to boolean
  2) System.out.println(~4);// Output=-5
  ```

Boolean Complement Operator(!)

- We can apply these operator only for Boolean type but not for integral

Example:

1) System.out.println(!4); // C.E. Operator can't applied to int
2) System.out.println(! False); //True
3) System.out.println(! True); //False

- **Summary:**

  &, |, ^ : We can apply for both integral and Boolean types,

  ~ : We can apply only for integral types but not for Boolean types.

  ! :We can apply only for Boolean types but not for integral types.

Short Circuit Operator (&&,||)

- We can apply this operator Just to improve performance of the system.
- These are exactly same as normal bitwise operators &,| except the following difference.

&,|	&&,||
1)Both operand should be evaluated always.	1)2nd operand evaluation is optional.
2)Relatively low performance.	2)Relatively high performance.
3)Applicable for both Boolean and integral types.	3)Applicable only for Boolean types.

- X&&y - y will be evaluated iff x is true.
- X||y - y will be evaluated iff x is false.

Example1:

```
public class shortandbitwise {
    public static void main(String[] args) {

    int x=10;
    int y=15;
    if(++x>10||++y<15)
    {
    ++x;
    }
    else
    {
    ++y;
    }
    System.out.println(x+"  "+y);

    }
}
```

Output:

	X	y
&	11	17
|	12	16
||	12	15
&&	11	17

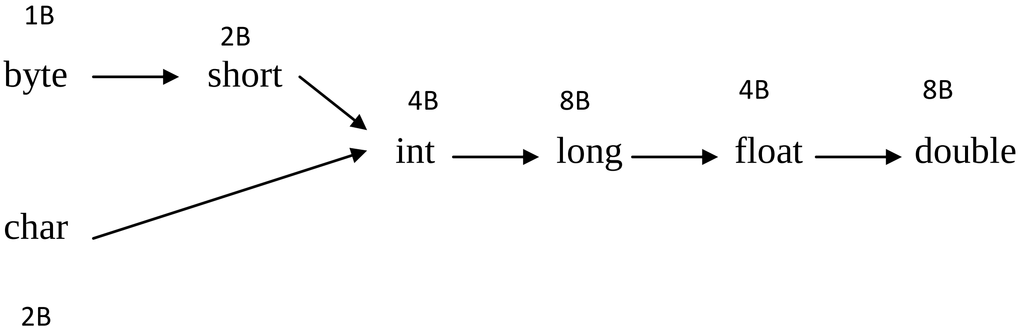
Type Casting Operator

- There are two types of primitive type casting.
  1) Implicit type casting
  2) Explicit type casting

1)Implicit type casting:

- Compiler is the responsible to perform this type casting.
- This Typecasting is required whenever we are assigning amaller datatype value to the bigger data type variable.
- It is also known as “widening (or) upcasting”.
- No loss of information in this type casting.

The following are various possible implicit type casting.



Example:

1)double d=10; [Compiler convert automatically int to double]

System.out.println(d) //10

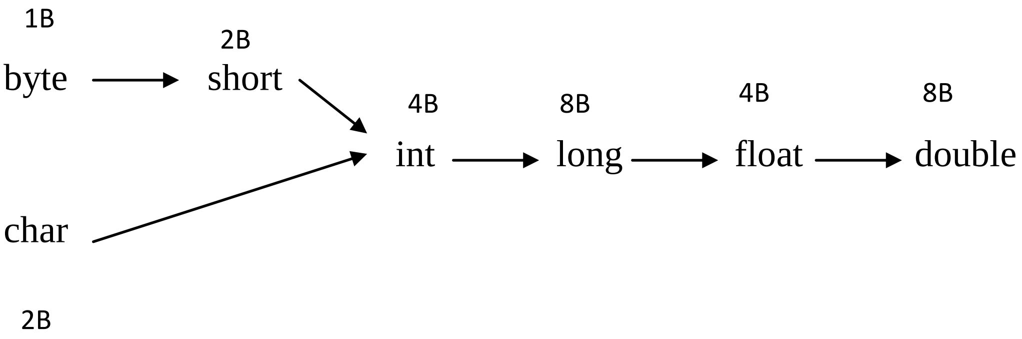
2)int x=’a’; [Compiler convert automatically int to double]

System.out.println(x) //97

1)Explicit type casting:

- Programmer is responsible to perform this typecasting.
- It is required whenever we are assigning bigger datatype value to the smaller datatype variable.
- It is also known as “Narrowing or down casting”.
- There may be a chance of loss of information in this type casting.
- The following are various possible conversion where explicite typecasting is required.
-

The following are various possible implicit type casting.



Example1:

```
        1)byte b=130;              // C.E. Possible loss of precision
                                    Required:byte
                                    Found:byte
        2)byte b=(byte)130;
          System.out .println(b);    // -126
```

Whenever we are assigning bigger datatype value to the smaller datatype variable.Then the most significant bit will be lossed.

Example1:

```
1)byte b=130;          //Invalid

byte b=(byte)130;    //-126
```

Example2:

```
1)int  x=150;          //Invalid

Short  s=(short)x;    //150
```

- Whenever we are assigning floating point datatype values to the integral datatype by explicit typecasting the digits after the decimal point will be lossed.

Example:

```
public class typecasting {

       public static void main(String[] args)

  {

        double d=130.45233;

       int a=(int) d;

     byte b=(byte) d;

     System.out.println(a);

    System.out.println(b);

  }

}
```

Output:

```
130

-126
```

Assignment Operator

- There are two types of primitive type casting.
  1) Simple Assignment operator.
  2) Chained assignment operator.
  3) Compound assignment operator

1)Simple assignment operator:

Example1: int x=10;

2)Chained assignment operator:

```
Example1: int a,b,c,d;

                    a=b=c=d=20;
```

- We can't perform chained assignment at the time of declaration

```
Example1:
                int  a=b=c=d=20;   //C.E. Can't find symbol
                                   //symbol:variable b
                                   //location:class Test
```

3)Compound assignment operator:

- Some times we can mix assignment operator with some other operator to form compound assignment operator.

```
Example1:
                int  a=20;
                    a+=30 ;
                System.out.println(a)        //40
```

- The following are various possible compound assignment operator.

Operators in java

+=	>>>=
-=	<<+
%=	&=
*=	|=
/=	^=
>>=	

- In compound assignment operators the required typecasting will be performed automatically by the compiler .

Example1:

```
    public class compoundassignment {
   public static void main(String[] args) {
        byte b=10;
        b=(byte) (b+1);
        System.out.println(b);
   }

}
```

Output: 11

Example 2:

```
public class compoundassignment {
   public static void main(String[] args) {
        byte b=127;
        b+=3;
        System.out.println(b);
   }

}
```

Output: -126

Conditional Operator

- The only ternary operator available in java is a ternary operator (or) conditional operator.

Example1:

```
    int a=10,b=20;
   int x=(a>b)?40:50;
```

- Nesting of conditional operator is possible.

  Example1:

```
     int a=10,b=20;
     int x=(a>50)?777((b>100)?888:999);
     System.out.println(x)                    //999
```

  Example2:

```
   public class conditionaloperator {
   public static void main(String[] args) {
  int a=10;
  int b=20;
  byte ch=(true)?40:50;
   byte c=(false)?40:50;
}
}
```

Example 3:

```
    public class finalconditional {
    public static void main(String[] args) {
      final  int a=10;
      final int b=10;
      byte c=(a<b)?40:50;
      byte s=(a>b)?40:50;
    }
}
```

➢ new operator:

- We can use this operator for creation of object.
- In java there is no delete operator because distraction of useless object is responsibility of garbage collector.

• [] operator

- We can use this operator for declaring and creating array.

operator Precidence

1) **Unary operators:**

   [],x++,x--;

   ++x;--x,~,!;

   New,<type> (used to type cast)

2) **Arithmetic operators:**

*,/,%
+,-

3) **shift operators:**
   >>>,>>,<<
4) **comparision operators:**
   <,<=,>,>=,instanceof
5) **equality operators:**
   =,!=
6) **bitwise operators:**
   &,^,|
7) **short circuit operators:**
   &&,||
8) **conditional operators:**
   ?:
9) **assignment operators:**
   +=,-+,%=,*=,/=,&=,|=,^=,>>=,>>>=,<<=

➢ Evaluation order of operators:

- There is no precedence for operands before applying any operator.All operands will be evaluated from left to right.

  Example1:

```
public class Evaluationprecedance {
    public static void main(String[] args) {
        System.out.println(m1(1)+m1(2)*m1(3)+m1(4)*m1(5)/m1(6));
    }
    public static int m1(int i)
    {
        System.out.println(i);
        return i;

    }
}
```

Output:

```
1
2
3
4
5
6
10
```

Explanation:$1+\underline{2*3}+\$*5/6$

$1+6+\underline{4*5}/6$

$1+6+3$

$7+3$

$= 10$