CST 205
OOP USING JAVA
MODULE 2

SMITHA JACOB,AP,CSE

SJCET,PALA

MODULE 2-Syllabus

Primitive Data types - Integers, Floating Point Types, Characters, Boolean. Literals, Type Conversion and Casting, Variables, Arrays, Strings, Vector class.

Operators – Arithmetic Operators, Bitwise Operators, Relational Operators, Boolean Logical Operators, Assignment Operator, Conditional (Ternary) Operator, Operator Precedence.

Control Statements - Selection Statements, Iteration Statements and Jump Statements.

MODULE 2-Syllabus

Object Oriented Programming in Java –

Class Fundamentals, Declaring Objects, Object Reference, Introduction to Methods, **Constructors,** this Keyword, **Method Overloading**, Using Objects as Parameters, Returning Objects, **Recursion**, Access Control, Static Members, Final Variables, Inner Classes, Command Line Arguments, Variable Length Arguments.

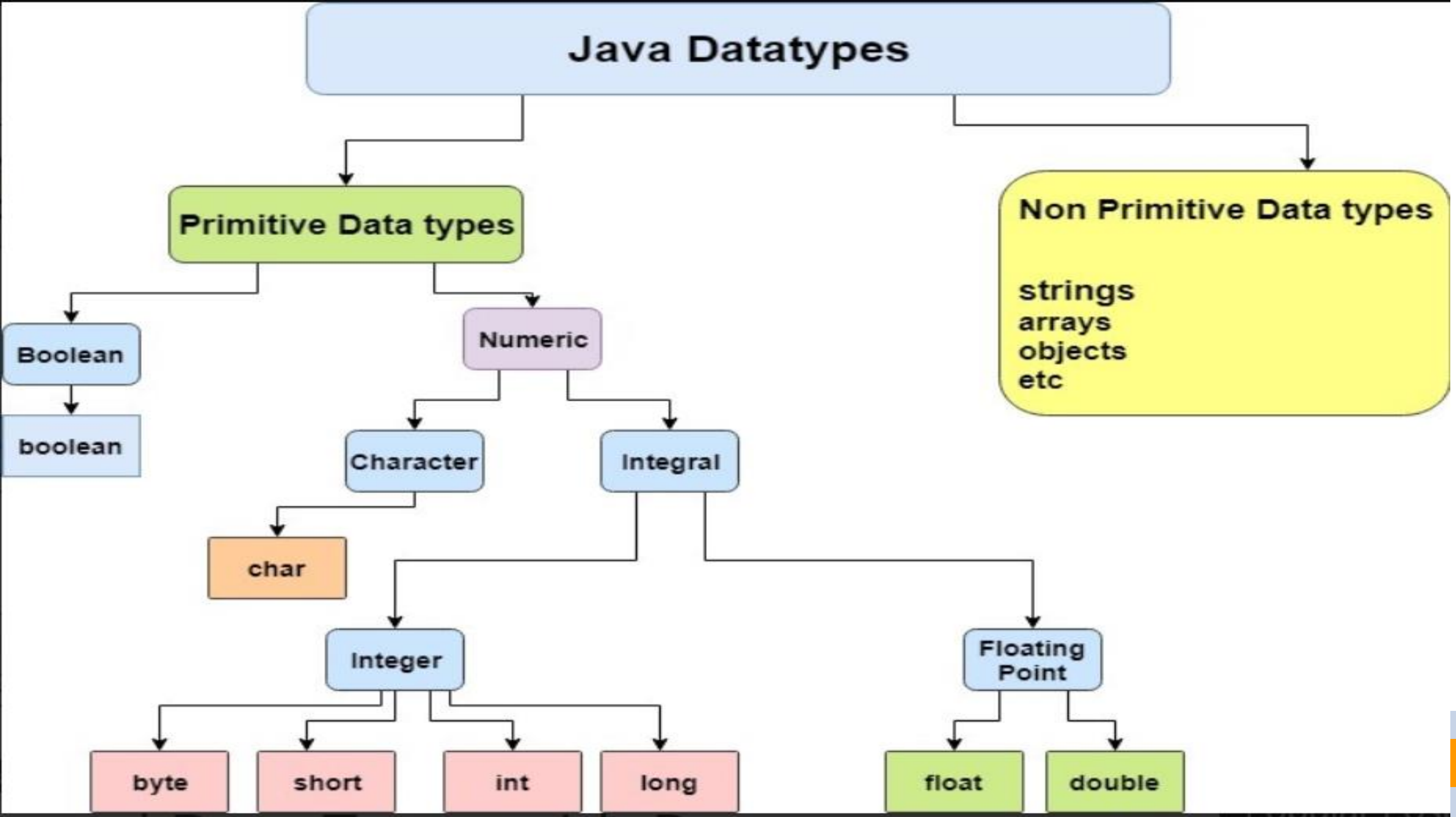
Inheritance –

Super Class, Sub Class, The Keyword super, protected Members, Calling Order of Constructors, **Method Overriding**, the Object class, Abstract Classes and Methods, using final with Inheritance

DATA TYPES

Data Types

- **Variables are the reserved memory** locations to store values.
- Based on the data type of a variable, the operating system allocates memory and decides what can be stored in the reserved memory.
- Data types specify the different sizes and values that can be stored in the variable.
- The Java programming language is statically-typed, which means that all variables must first be declared before they can be used.
- There are two types of data types in Java:
  - Primitive data types
  - Non-primitive data types


Java Datatypes
Primitive Data types
Non Primitive Data types
strings
arrays
objects
etc
Boolean
boolean
Numeric
Character
Integral
char
Integer
Floating Point
byte
short
int
long
float
double

Data Types

Primitive Data Types:

- A primitive data type is pre-defined by the programming language.
- The size and type of variable values are specified, and it has no additional methods.
- It stores the values

Non-Primitive Data Types:

- These data types are not actually defined by the programming language but are created by the programmer.
- They are also called “reference variables” or “object references” since they reference a memory location which stores the data.
- Refers to a m/y location where data is stored

Primitive Data Types

- Primitive Data Types are the Fundamental Data Types
- Primitive Data Types are predefined and available within the Java language.
- A primitive type is predefined by the language and is named by a reserved keyword.
- Java defines eight primitive types of data:
  - Integers- **byte, short, int**, and **long**
  - Floating Point Numbers- **float** and **double**
  - **Character**
  - **Boolean**

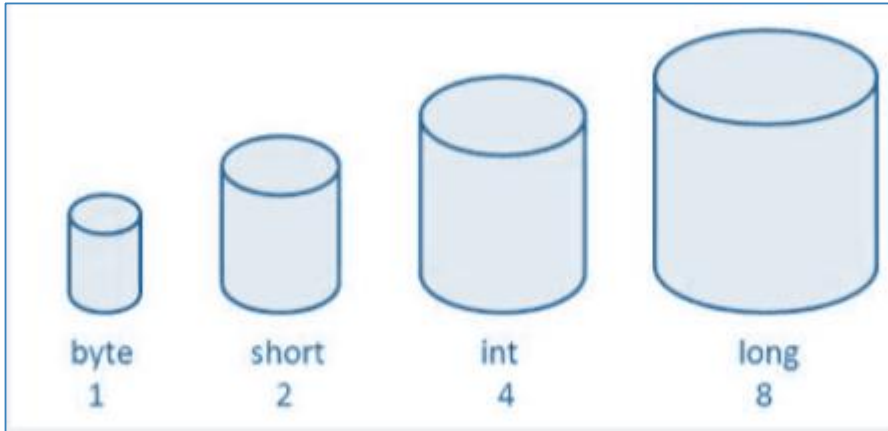
PRIMITIVE DATA TYPES

- Java defines eight primitive types of data: **byte, short, int, long, char, float, double**, and **boolean.**
  - **Integers** : **byte, short, int**, and **long**, which are for whole-valued signed numbers.
  - **Floating-point numbers : float** and **double**, which represent numbers with fractional precision.
  - **Characters** :**char**, which represents symbols in a character set, like letters .
  - **Boolean** : **boolean**, which is a special type for representing true/false values.

INTEGERS

- Java defines four integer types:
- **byte, short, int,** and **long**.
- All of these are signed, positive and negative values.
- **Signed numbers** use **sign** flag or can be distinguish between negative values and positive values..
- Java does not support unsigned, positive -only integers.

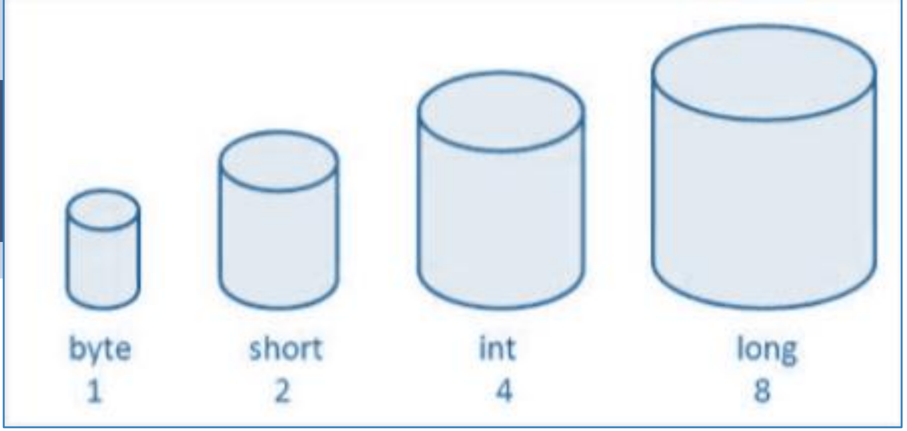
Name	Width
long	64 bit
int	32 bit
short	16 bit
Byte	8 bit



INTEGER - byte

- **byte**: The smallest integer type is byte.
- This is a **signed 8-bit**
- Variables of type byte are especially useful
  - when you’re working with a stream of data from a network or file.
  - when you’re working with raw binary data that may not be directly compatible with Java’s other built-in types.
- Byte variables are declared by use of the **byte** keyword.
- For example, the following declares two byte variables called b and c:
  - **byte** b, c;

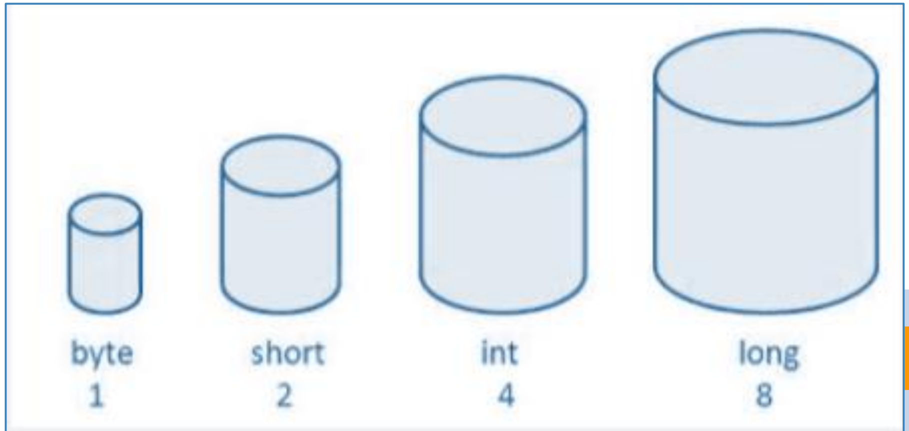
INTEGER - short



- short is a signed 16-bit type.
- It is probably **the least used Java type**.
- you can use a short to save memory in large arrays, in situations where the memory savings actually matters.
- Example : **short** s;

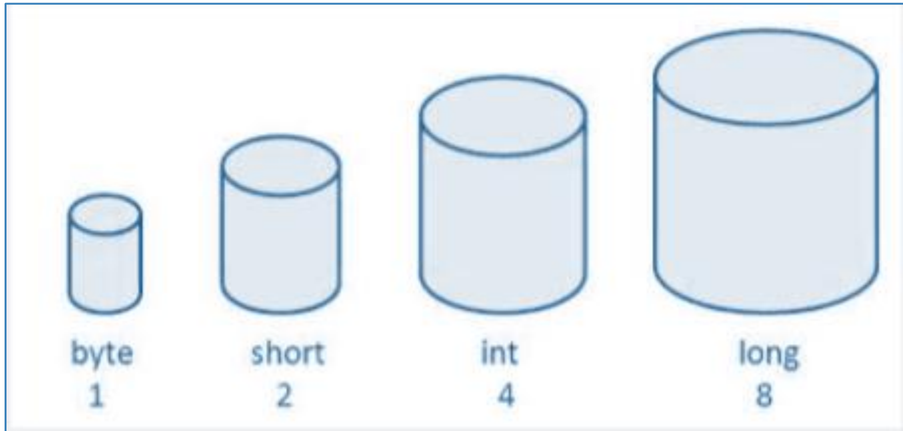
INTEGER - int

- The most commonly used integer type is **int**.
- variables of type int are commonly employed to control loops and to index arrays.
- Example: **int** a;

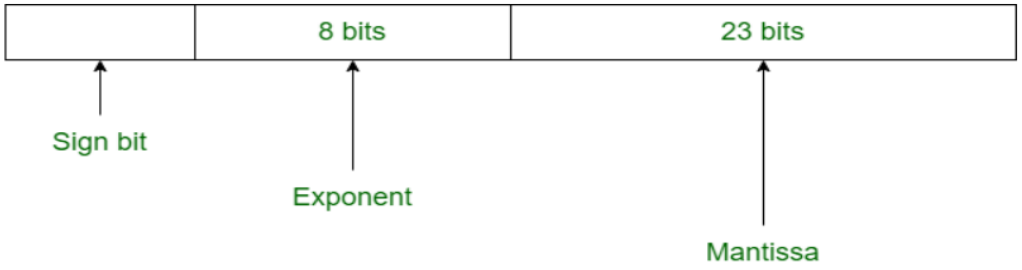


INTEGER - long

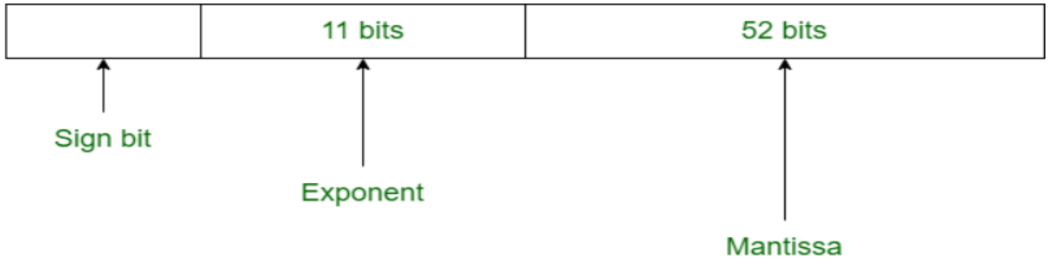
- **long** is a signed 64-bit type
- It is useful for those occasions where an **int type is not large enough to hold the desired value**.
- Example: **long** a;



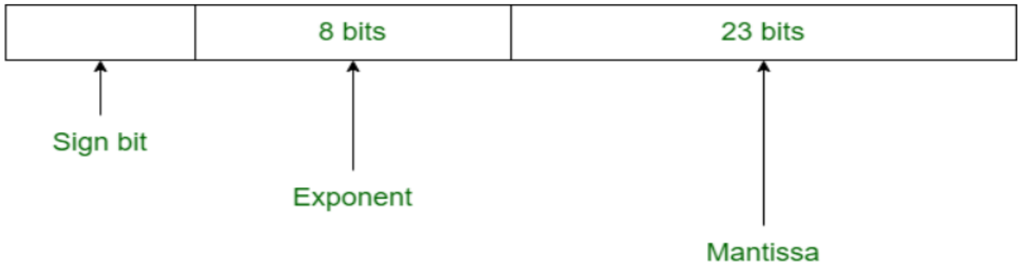
FLOATING-POINT



- Floating-point numbers, also known as real numbers, are used when evaluating expressions that **require fractional precision**.
- There are two kinds of floating-point types, **float** and **double**, which represent single(32 bit)- and double-precision numbers(64 bit), respectively.
- Ex: all mathematical functions like sin(),cos(),sqrt() return double values



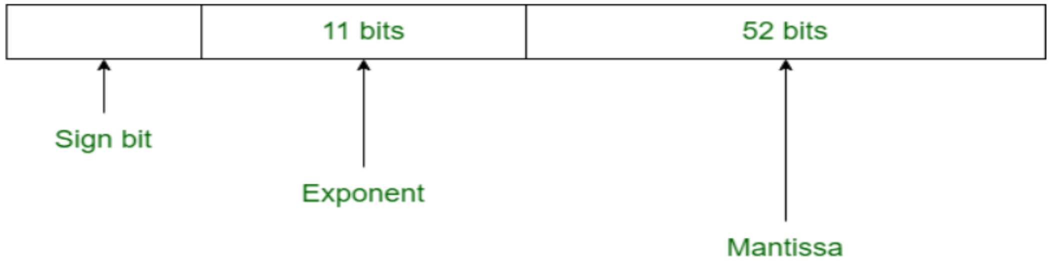
FLOATING-POINT : float



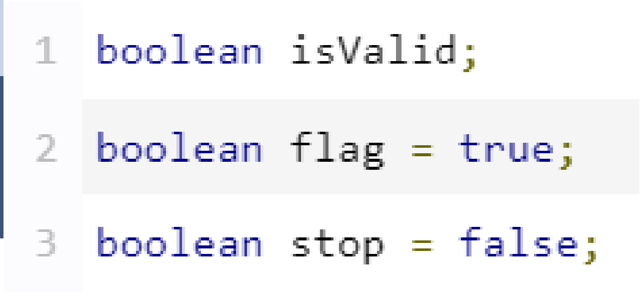
- The type **float** specifies a single-precision value that uses 32 bits of storage.
- 32 bit floats Format contains a 1 sign bit,8 exponent bits(range),24 fraction bits
- Variables of type **float** are useful when you need a fractional component, but don't require a large degree of precision.
- Example: **float** hightemp, lowtemp;

FLOATING-POINT : double

- Double precision, as denoted by the **double** keyword, uses 64 bits to store a value.
- When you need to maintain accuracy over many iterative calculations, or are manipulating large-valued numbers, **double** is the best choice.
- 64 bit floats Format contains a 1 sign bit,11 exponent bits(range),53 fraction bits
- Example : **double** pi, r, a;



BOOLEANS



```java
boolean isValid;
boolean flag = true;
boolean stop = false;
```

- Java has a primitive type, called **boolean**, for logical values.
- It can have only one of two possible values, **true** or **false**.
- they cannot be used for variable, function, class, or object names!
- the result of a Boolean operation is only true or false (lowercase in Java code).
- This is the type returned by all relational operators, as in the case of **a < b**.
- Example: **boolean** b;

CHARACTERS

Data Type	Memory Size	Default value	Minimum Value	Maximum Value	Example
char	2 bytes	'\u0000'	0	65535	Char c='t'

- In Java, the data type used to store characters is **char**. a char could be any value from A to Z, and all the numbers.
- Java uses *Unicode* to represent characters.At the time of Java's creation, Unicode required 16 bits.
- Unicode is a computer encoding methodology that assigns a unique number for every character. It doesn't matter what language, or computer platform it's on.
- in Java char is a **16-bit type**.
- In Unicode, all of the characters are represented by numeric values.
- Example: **char** letterA = 'A‘;

DATA TYPES IN JAVA

Data Type	Default Value	Default size
boolean	false	1 bit
char	‘\u0000’	2 byte
byte	0	1 byte
short	0	2 byte
int	0	4 byte
long	0L	8 byte
float	0.0f	4 byte
double	0.0d	8 byte

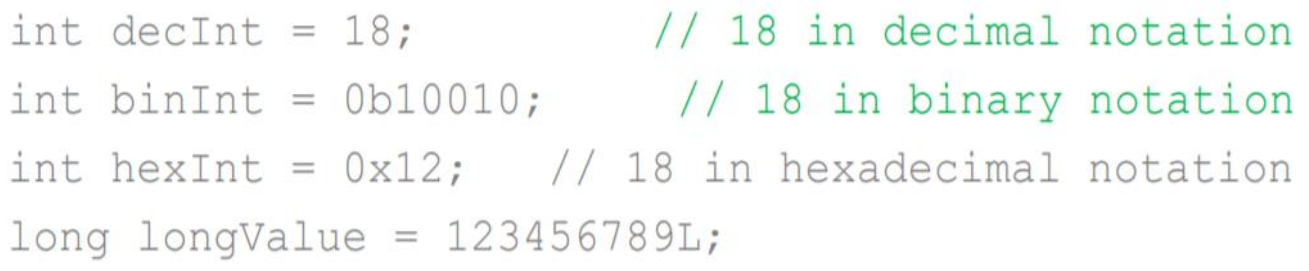
Literals

Literals

```
boolean result = true;
char capitalC = 'C';
byte b = 100;
short s = 10000;
int i = 100000;
```

- when initializing a variable of a primitive type, new keyword isn't used
- Primitive types are special data types built into the language; they are not objects created from a class.
- A literal is the source code representation of a fixed value;
- Represented directly in your code without requiring computation.
- We can assign a literal to a variable of a primitive type:

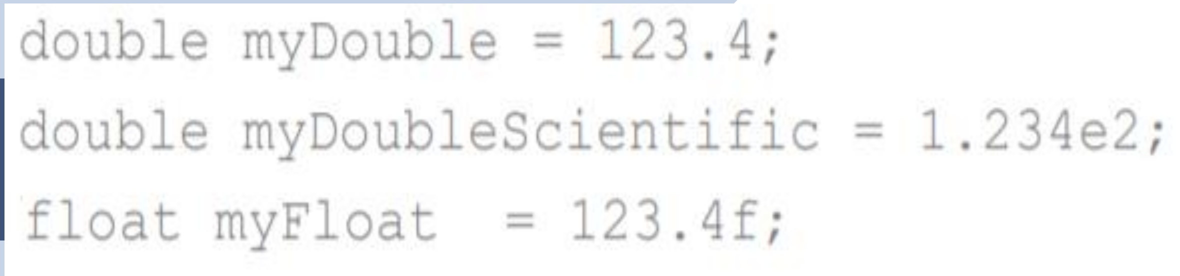
a) Integer Literals



```
int decInt = 18;                // 18 in decimal notation
int binInt = 0b10010;           // 18 in binary notation
int hexInt = 0x12;    // 18 in hexadecimal notation
long longValue = 123456789L;
```

- Integers can be expressed in decimal (base 10),Binary(base 2) hexadecimal (base 16), or octal (base 8) format.
- In Java, a binary literals starts with 0b and a decimal integer literal consists of a sequence of digits without a leading 0 (zero).
- A leading 0 (zero) on an integer literal means it is in octal; Octal integers can include only the digits 0-7.
- a leading 0x (or 0X) means hexadecimal. Hexadecimal integers can include digits (0-9) and the letters a-f and A-F.
- Integer literal that ends with l or L is of type long.

b) Floating Point Literals

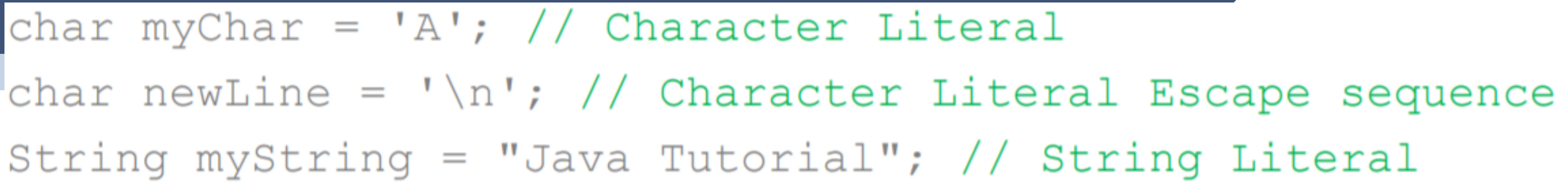


```
double myDouble = 123.4;
double myDoubleScientific = 1.234e2;
float myFloat  = 123.4f;
```

A floating point literal can have the following parts:

- a decimal integer, a decimal point ("."), a fraction (another decimal number),
- an exponent, and a type suffix.
- The exponent part is an e or E followed by an integer, which can be signed.
- A floating-point literal is of type float if it ends with the letter F or f
- Otherwise its type is double and it can optionally end with the letter D or d.
- The floating point types (float and double) can also be expressed using E or e (for scientific notation)

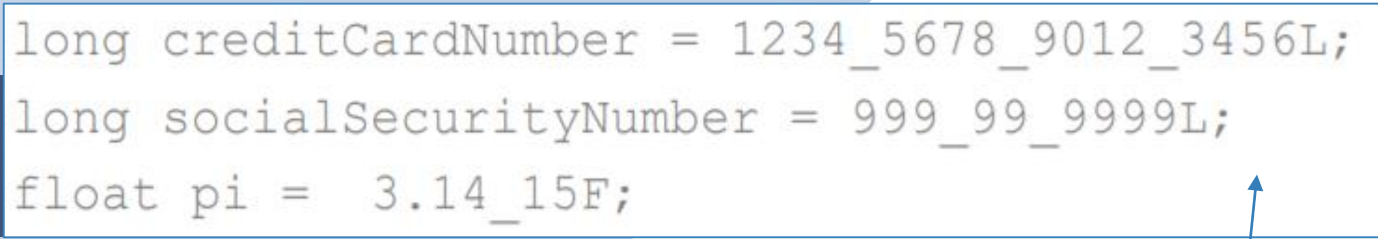
c) Character & String Literals



```
char myChar = 'A'; // Character Literal
char newLine = '\n'; // Character Literal Escape sequence
String myString = "Java Tutorial"; // String Literal
```

- A character literal is a character (or group of characters representing a single character) enclosed in single quotes.
- Characters have type char and are drawn from the Unicode character set
- String literals are represented as a sequence of characters surrounded by double quotes
- Java also allows use of escape sequences in string and character literals.
  - For example, \b (backspace), \t (tab), \n (line feed), \f (form feed), \r (carriage return), \” (double quote), \’ (single quote), and \\ (backslash).

BOOLEANS Literals



```
long creditCardNumber = 1234_5678_9012_3456L;
long socialSecurityNumber = 999_99_9999L;
float pi =  3.14_15F;
```

- Java has a primitive type, called **Boolean** and the type Boolean has two literal values: true and false.
- It can have only one of two possible values, **true** or **false**.
- ▶ boolean flag = true;
- This is the type returned by all relational operators, as in the case of **a < b**.
- In Java SE 7 and later, any number of underscore characters (_) can appear anywhere between digits in a numerical literal.

variables

Variable

- Variable in Java is a data container that stores the data values during Java program execution.
- Variable is a memory location name of the data.
- variable="vary + able" that means its value can be changed.
- all variables have a scope, which defines their visibility, and a lifetime
- In order to use a variable in a program we need to perform 2 steps
  - Variable Declaration
  - Variable Initialization(optional)

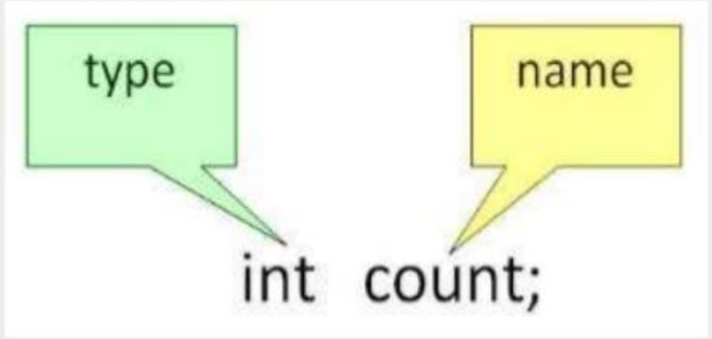
1. Variable Declaration

Syntax: data_type variable_name ;

Eg: int a,b,c;

float pi;

double d;



2. Variable Initialization

Syntax : data_type variable_name = value;

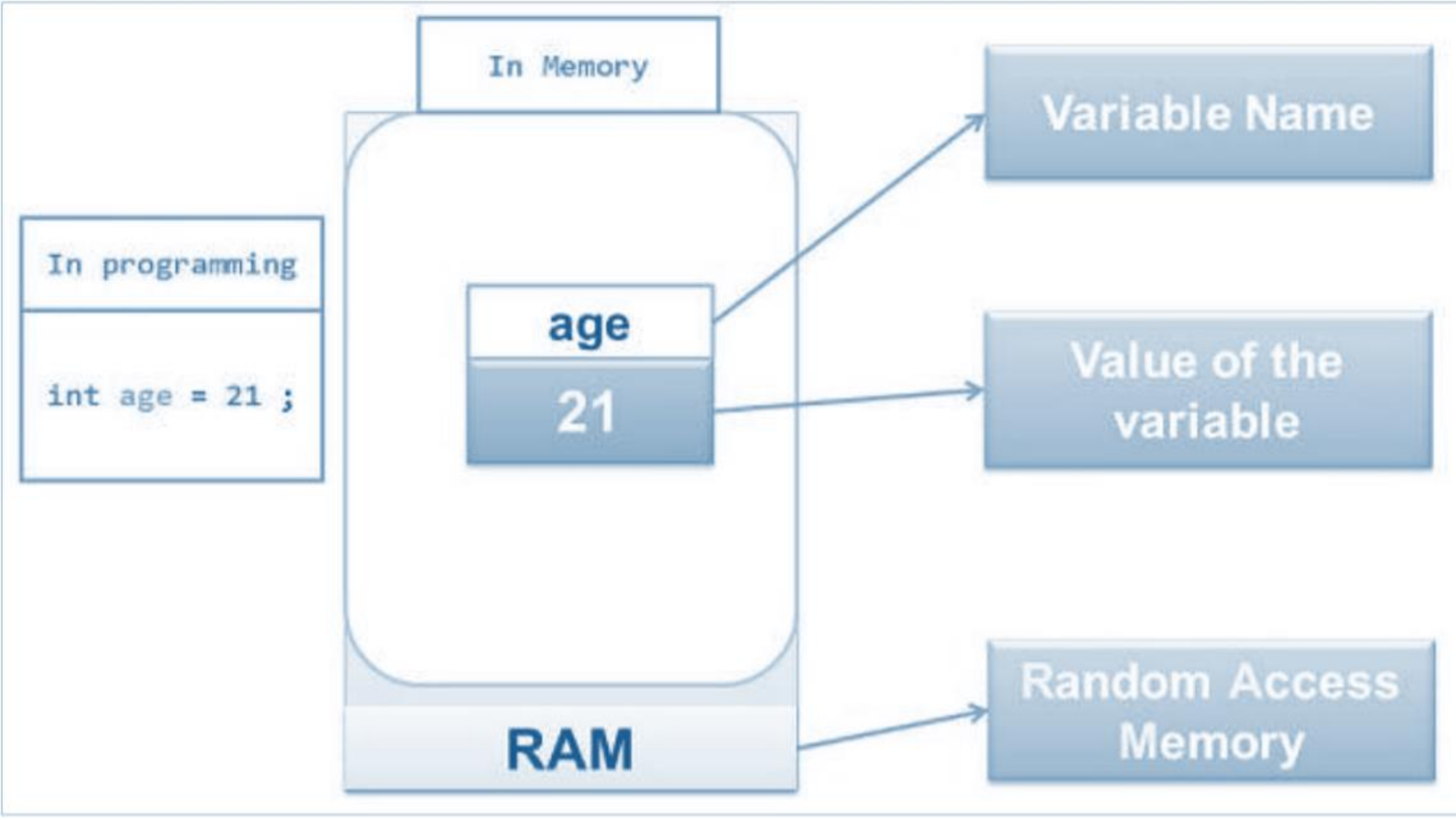
Eg: int a=2,b=4,c=6;

float pi = 3.14f;

double val = 20.22d;

char a = 'v';

int num = 45.66;

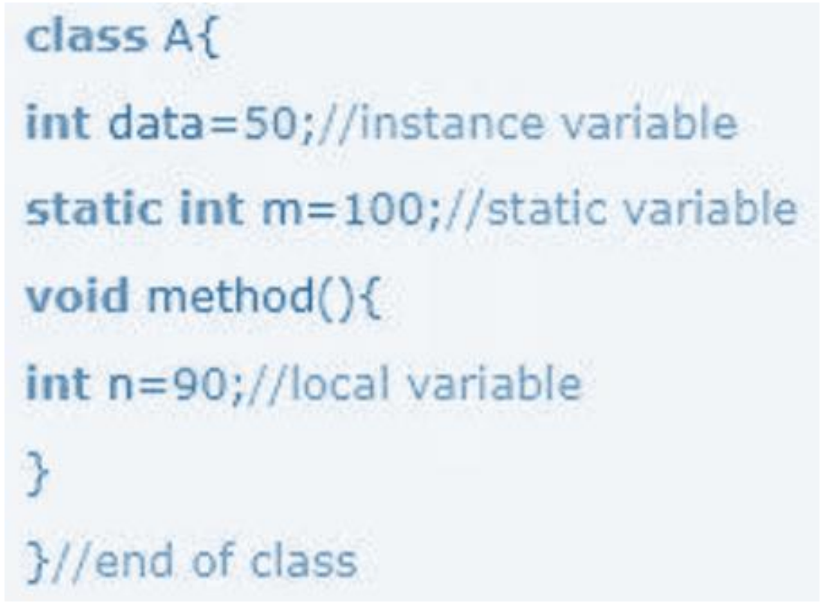

In Memory
In programming
int age = 21 ;
age
21
RAM
Variable Name
Value of the variable
Random Access Memory

Naming of Variable

- Variable names are case-sensitive . Identifiers should not start with digits([0-9]).
- The only allowed characters for identifiers are all alphanumeric characters([A-Z],[a-z],[0-9]), ‘$‘(dollar sign) and underscore.
- Reserved Words can’t be used as an identifier.
- Name→consists of only **one word**, →all lowercase letters.
- consists **of more than one word**→capitalize the first letter of each subsequent word.
  - Examples: maxValue, currentGrade (CamelCase)
- **constant value**, such as static final int MAX_MARKS = 6,→**capitalizing every letter and separating subsequent words with the underscore character**.

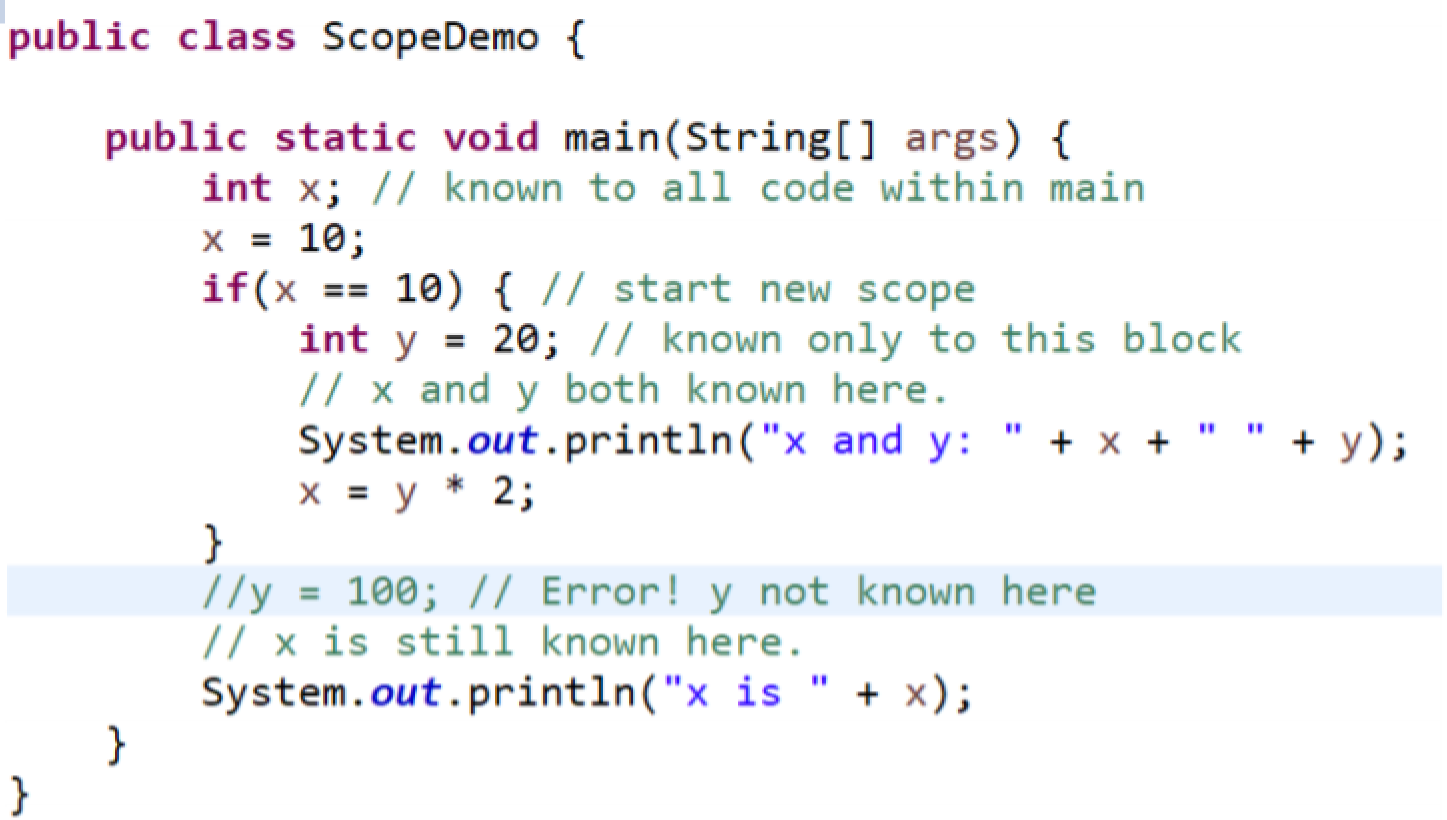
Types Of Variables

- Local variables - declared inside the method.
- Instance Variable - declared inside the class but outside the method.
- Static variable - declared as with static keyword.



```
class A{
int data=50;//instance variable
static int m=100;//static variable
void method(){
int n=90;//local variable
}
}//end of class
```

Scope & Life time of variables



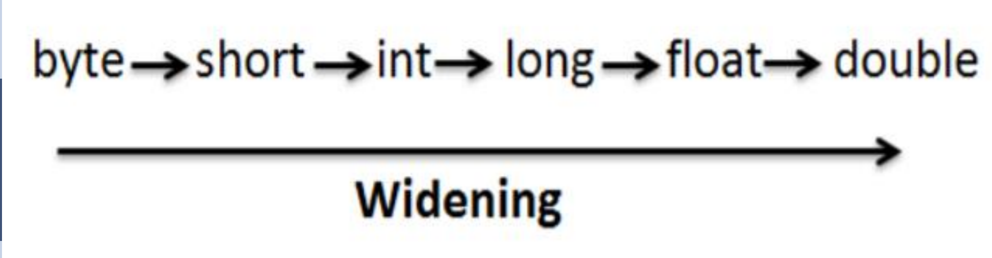
```java
public class ScopeDemo {

    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
            // x and y both known here.
            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);
    }
}
```

Type Conversion & Casting

Type Conversion

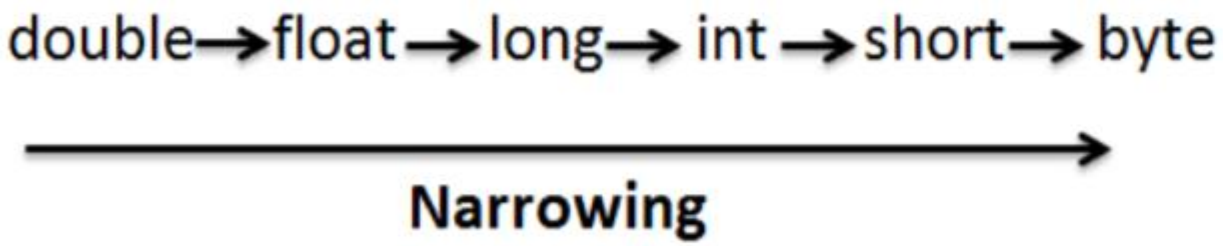
- When a data type is converted into another type is called type conversion (casting).
- When a variable is converted into a different type, the compiler basically treats the variable as of the new data type.
- Type of Type Conversion In Java
  - Implicit Conversion
  - Explicit Conversion (Type Casting)

implicit Type Conversion



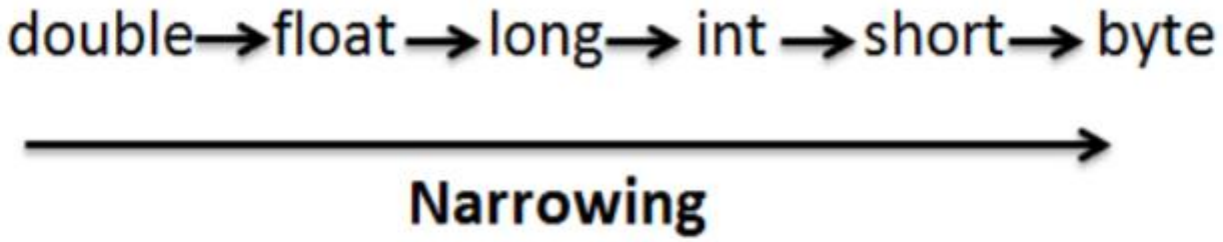
- implicit or automatic conversion compiler will automatically change one type of data into another.
- Implicit or automatic conversion can happen if both type are compatible and target type is larger than source type.
- If you assign an integer value to a floating-point variable, the compiler will insert code to convert the int to a float.
- In implicit conversion a **smaller type is converted into a larger** thus it is also known as **widening conversion**

Explicit Type Conversion



- Narrowing Casting (manually) – converting a larger type to a smaller size type (called Type Casting)
- There may be situations when we want to convert a value having larger type to a smaller type.
- In this case casting needs to be performed explicitly.
- Explicit casting allows you to make this type conversion explicit, or to force it when it wouldn't normally happen.

Explicit Type Conversion

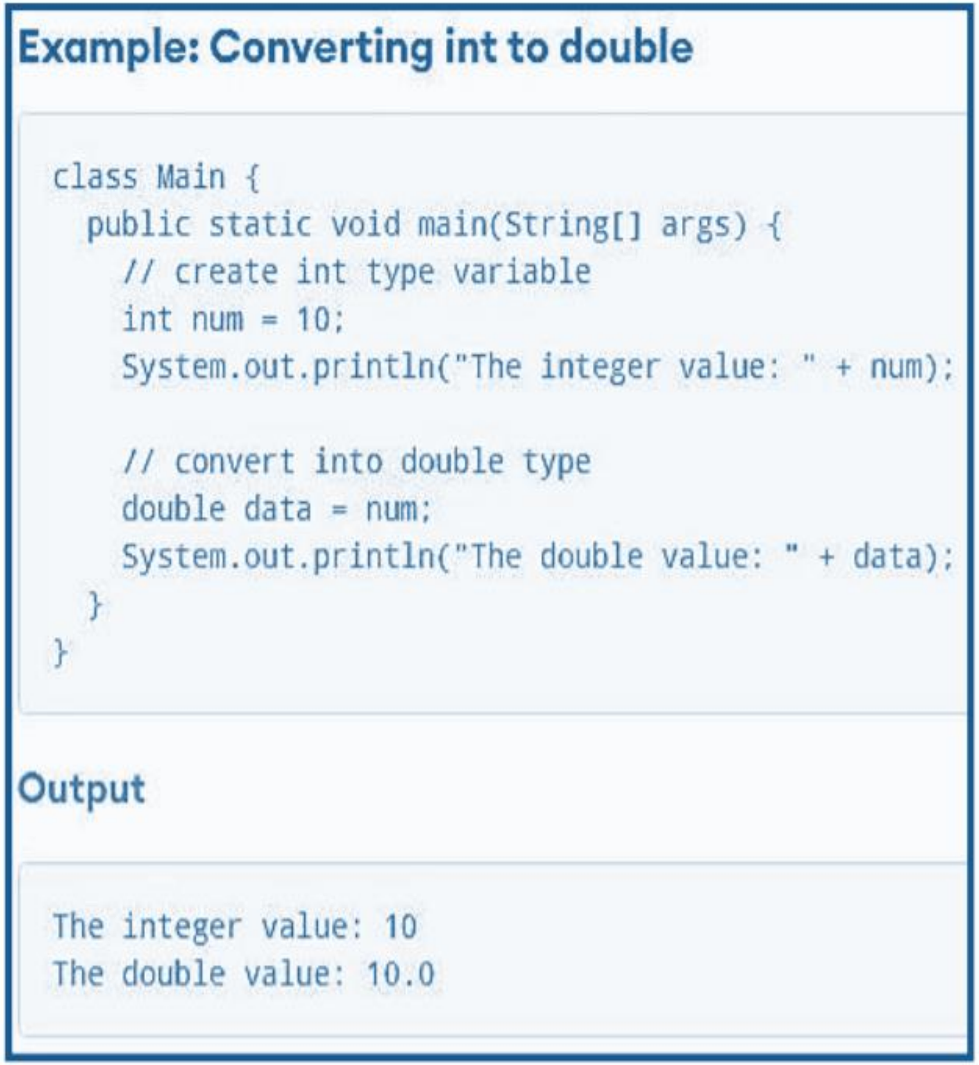


Syntax :

- *(type) expression*
- To perform type casting, put the desired type including modifiers inside parentheses to the left of the variable or constant you want to cast.
- Example int a =10;

  byte b = (byte) a;

- Here, size of source type int is 32 bits and size of d**estination type byte is 8 bits**.
- Since we are converting a source type having larger size into a destination type having less size, such conversion is known as **narrowing conversion**.

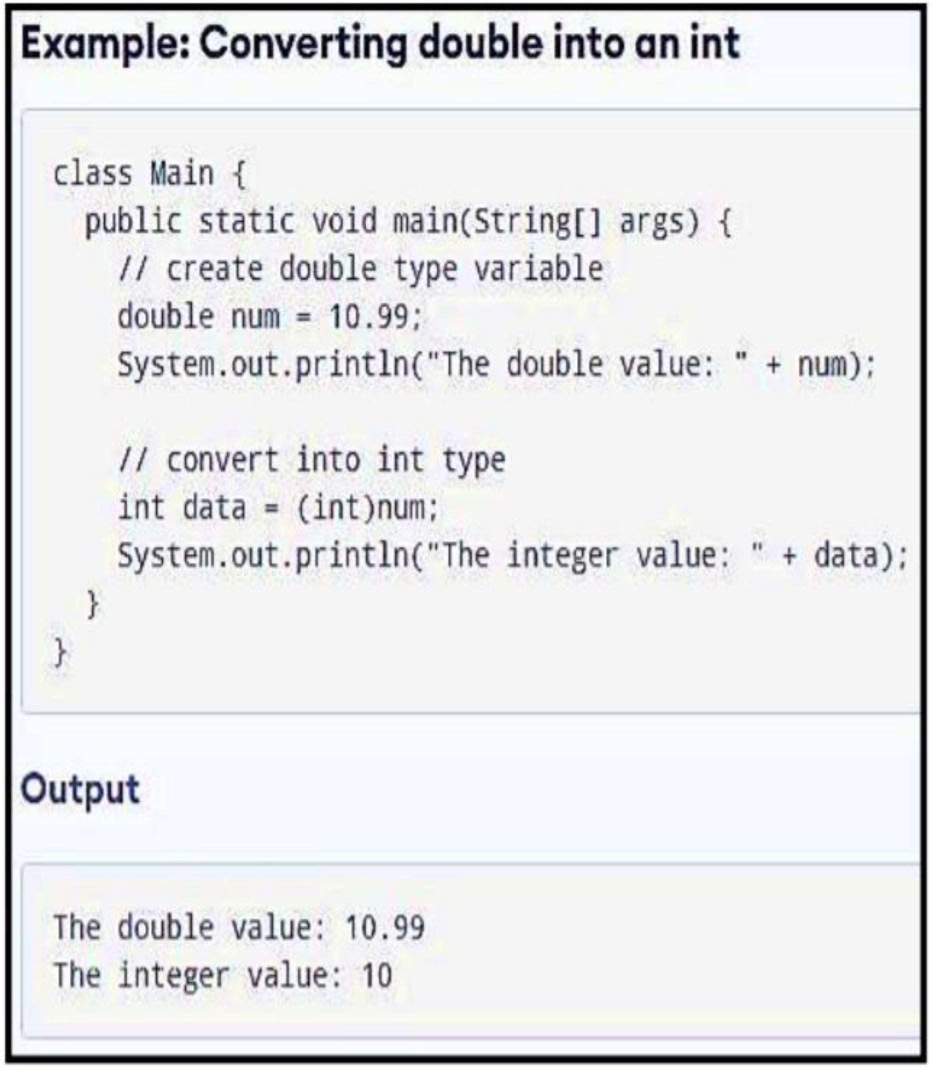


Example: Converting int to double

```
class Main {
  public static void main(String[] args) {
    // create int type variable
    int num = 10;
    System.out.println("The integer value: " + num);

    // convert into double type
    double data = num;
    System.out.println("The double value: " + data);
  }
}
```

Output

```
The integer value: 10
The double value: 10.0
```



Example: Converting double into an int

```
class Main {
  public static void main(String[] args) {
    // create double type variable
    double num = 10.99;
    System.out.println("The double value: " + num);

    // convert into int type
    int data = (int)num;
    System.out.println("The integer value: " + data);
  }
}
```

Output

```
The double value: 10.99
The integer value: 10
```

Truncation

- when a floating-point value is assigned to an integer type: truncation takes place,
- integers do not have fractional components .
- when a floating-point value is assigned to an integer type, the fractional component is lost.
- For example, if the value 45.12 is assigned to an integer, the resulting value will simply be 45. The 0.12 will have been truncated.
- **No automatic conversions from the numeric types to char or boolean. Also, char and boolean are not compatible with each other**.

Sample University Questions

What is the output of the following code? Justify.

```
class Test
{ public static void main(String a[])
                { byte b = 50;
               b = b * 2;
               System.out.print(b);
               } }
```

Answer

```
public class Test
{public static void main(String[] args)
{byte b = 50;
b = (byte)(b * 2);
System.out.println(b);
 }}
```

o/p: we will get the output 100 only if we convert int to byte explicitly.otherwise *"Type mismatch: cannot convert from int to byte"* compliation error occurs.

Sample University Questions

What is the output of the following code? Justify.

```
class Test
{ public static void main(String a[])
          { byte b = 50;
         //b = b * 2;
         System.out.print(b); //error
         int c=b*2;
         System.out.println("c="+c);//100
         System.out.println(b*2);//100} }
```

OPERATORS

OPERATORS

- Operators are special symbols used for: mathematical functions, assignment statements, logical comparisons
- Operators can be divided into the following four groups: arithmetic, bitwise, relational, and logical
- Expressions: can be combinations of variables and operators that result in a value

Operator

Java operators can be divided into following categories:

- Arithmetic Operators
- Relational Operators
- Bitwise Operators
- Logical Operators
- Assignment Operators
- conditional operator (Ternary)

ARITHMETIC OPERATORS

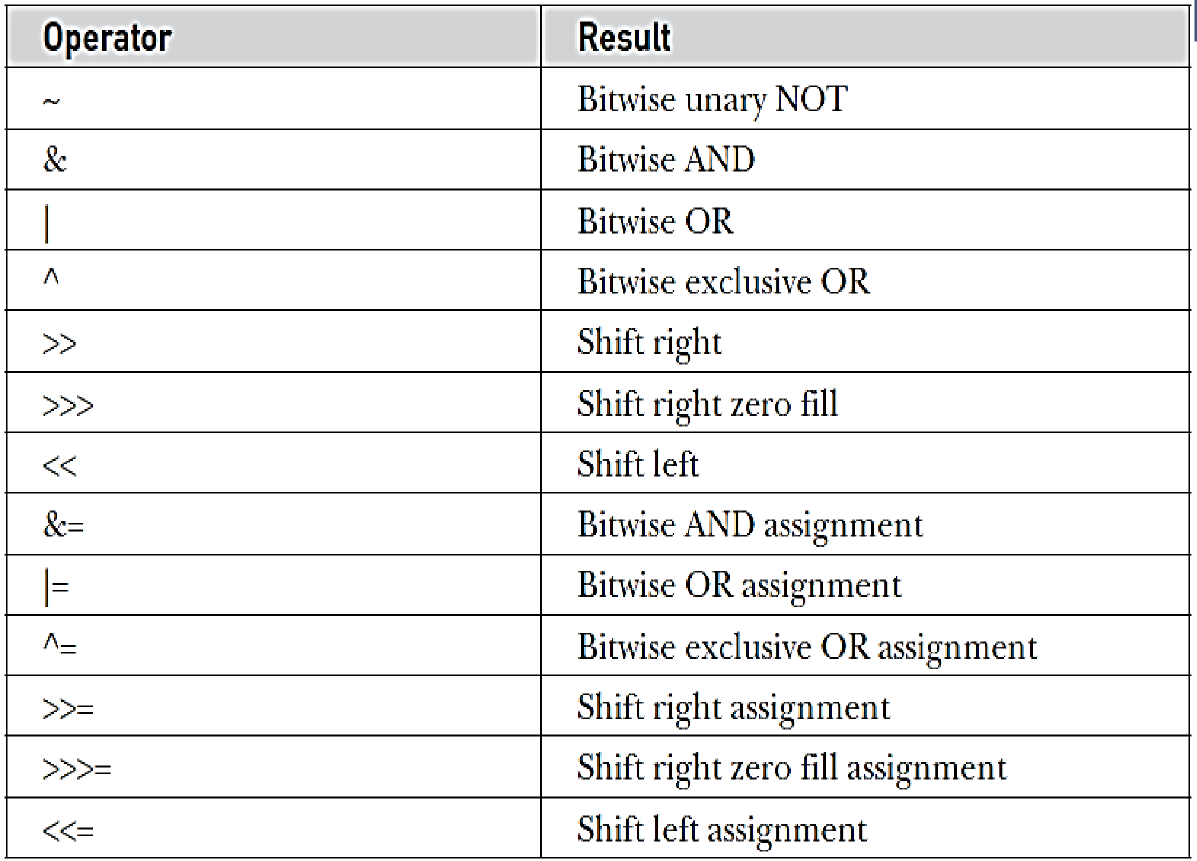
Operator	Result
+	Addition (also unary plus)
–	Subtraction (also unary minus)
*	Multiplication
/	Division
%	Modulus
++	Increment
+=	Addition assignment
– =	Subtraction assignment
*=	Multiplication assignment
/=	Division assignment
%=	Modulus assignment
– –	Decrement

```
class OperatorExample{
public static void main(String args[]){
int a=10;
int b=5;
System.out.println(a+b);//15
System.out.println(a-b);//5
System.out.println(a*b);//50
System.out.println(a/b);//2
System.out.println(a%b);//0
}}
```

Output:

```
15
5
50
2
0
```

THE 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 exclusive OR assignment
>>=	Shift right assignment
>>>=	Shift right zero fill assignment
<<=	Shift left assignment

Bitwise operators

- The Java programming language also provides operators that perform bitwise shift operations on integral types.
- The unary bitwise complement operator "~" inverts a bit pattern; it can be applied to any of the integral types, making every "0" a "1" and every "1" a “0".
- For example, a byte contains 8 bits; applying this operator to a value whose bit pattern is"00000000" would change its pattern to "11111111".
- The signed left shift operator "<<" shifts a bit pattern to the left, and
- the signed right shift operator ">>" shifts a bit pattern to the right.
- The bit pattern is given by the left-hand operand, and the number of positions to shift by the right-hand operand.

Bitwise operators

- The bitwise & operator performs a bitwise AND operation.
- The bitwise ^ operator performs a bitwise exclusive OR operation.
- The bitwise | operator performs a bitwise inclusive OR operation.

Bitwise operator works on bits and performs the bit-by-bit operation. Assume if a = 60 and b = 13; now in binary format they will be as follows −

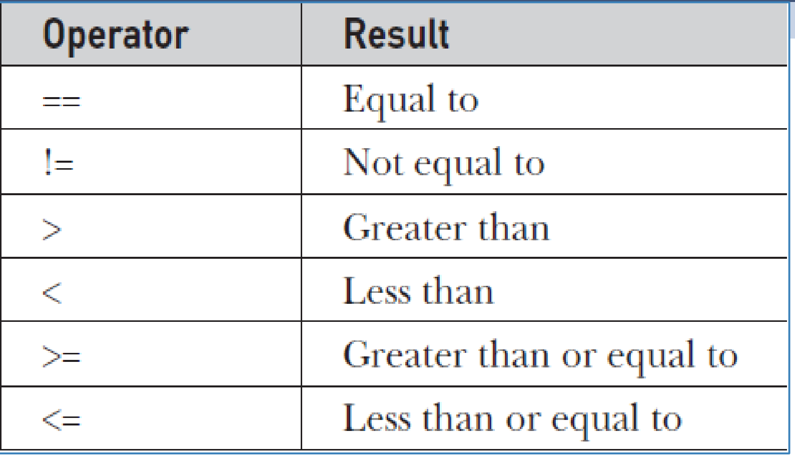
```
a  = 0011 1100
b  = 0000 1101
-----------------
a&b  = 0000 1100

a|b  = 0011 1101

a^b  = 0011 0001

~a  = 1100 0011
```

Truth Table		
A	B	Q
0	0	0
0	1	0
1	0	0
1	1	1
Read as A AND B gives Q		

Truth Table		
A	B	Q
0	0	0
0	1	1
1	0	1
1	1	1
Read as A OR B gives Q		

Operator	Description	Example
& (bitwise and)	Binary AND Operator copies a bit to the result if it exists in both operands.	(A & B) will give 12 which is 0000 1100
| (bitwise or)	Binary OR Operator copies a bit if it exists in either operand.	(A | B) will give 61 which is 0011 1101
^ (bitwise XOR)	Binary XOR Operator copies the bit if it is set in one operand but not both.	(A ^ B) will give 49 which is 0011 0001
~ (bitwise compliment)	Binary Ones Complement Operator is unary and has the effect of 'flipping' bits.	(~A) will give -61 which is 1100 0011 in 2's complement form due to a signed binary number.
<< (left shift)	Binary Left Shift Operator. The left operands value is moved left by the number of bits specified by the right operand.	A << 2 will give 240 which is 1111 0000
>> (right shift)	Binary Right Shift Operator. The left operands value is moved right by the number of bits specified by the right operand.	A >> 2 will give 15 which is 1111
>>> (zero fill right shift)	Shift right zero fill operator. The left operands value is moved right by the number of bits specified by the right operand and shifted values are filled up with zeros.	A >>>2 will give 15 which is 0000 1111

RELATIONAL OPERATORS



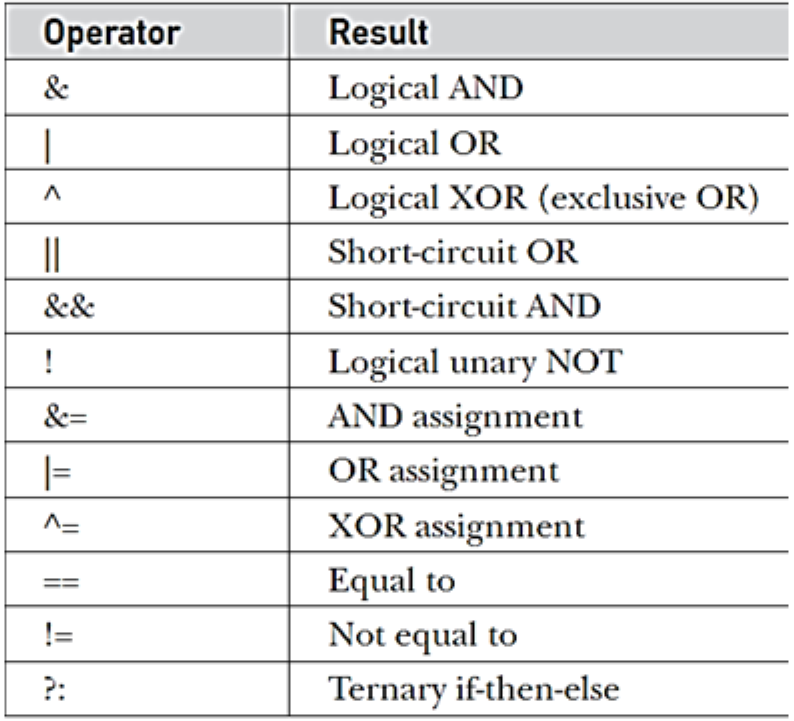
Operator	Result
==	Equal to
!=	Not equal to
>	Greater than
<	Less than
>=	Greater than or equal to
<=	Less than or equal to

```
class ComparisonDemo {

    public static void main(String[] args){
        int value1 = 1;
        int value2 = 2;
        if(value1 == value2)
            System.out.println("value1 == value2");
        if(value1 != value2)
            System.out.println("value1 != value2");
        if(value1 > value2)
            System.out.println("value1 > value2");
        if(value1 < value2)
            System.out.println("value1 < value2");
        if(value1 <= value2)
            System.out.println("value1 <= value2");
    }
}
```

Output:

```
value1 != value2
value1 <  value2
value1 <= value2
```

BOOLEAN LOGICAL OPERATORS



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 && and || operators perform **Conditional-AND and Conditional-OR operations** on two boolean expressions.
- These operators exhibit "short-circuiting" behavior, which means that the second operand is evaluated only if needed.
  && Conditional-AND
  || Conditional-OR

- You must use "==", not "=", when testing if two primitive values are equal.

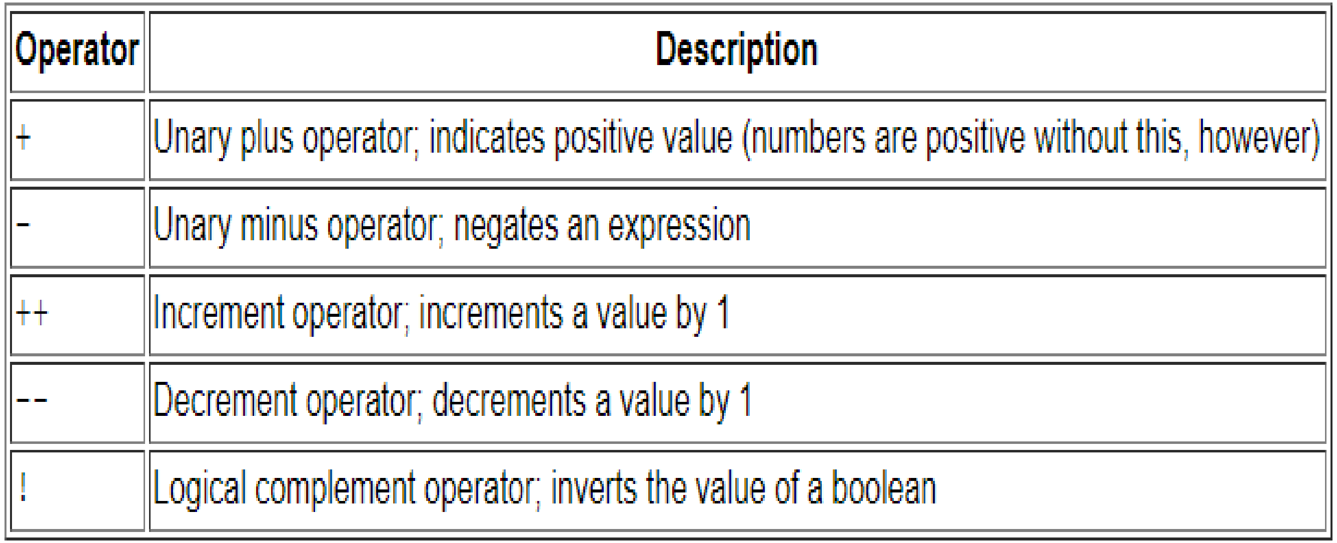
BOOLEAN LOGICAL OPERATORS

```
class ConditionalDemo1 {

    public static void main(String[] args){
        int value1 = 1;
        int value2 = 2;
        if((value1 == 1) && (value2 == 2))
            System.out.println("value1 is 1 AND value2 is 2");
        if((value1 == 1) || (value2 == 1))
            System.out.println("value1 is 1 OR value2 is 1");
    }
}
```

```
class ConditionalDemo2 {

    public static void main(String[] args){
        int value1 = 1;
        int value2 = 2;
        int result;
        boolean someCondition = true;
        result = someCondition ? value1 : value2;
        System.out.println(result);
    }
}
```

Unary operators

- The unary operators require only one operand;
- they perform various operations such as incrementing/decrementing a value by one, negating an expression, or inverting the value of a boolean.



Operator	Description
+	Unary plus operator; indicates positive value (numbers are positive without this, however)
-	Unary minus operator; negates an expression
++	Increment operator; increments a value by 1
--	Decrement operator; decrements a value by 1
!	Logical complement operator; inverts the value of a boolean

Unary operators

- result++; and ++result; will both end in result being incremented by one
- prefix version (++result) evaluates to the incremented value, whereas the postfix version (result++) evaluates to the original value.
- If you are just performing a simple increment/decrement, it doesn't really matter which version you choose.

```
class UnaryDemo {

    public static void main(String[] args)
 {
        int result = +1;
        // result is now 1
        System.out.println(result);
        result--;
        // result is now 0
        System.out.println(result);
        result++;
        // result is now 1
        System.out.println(result);
        result = -result;
        // result is now -1
        System.out.println(result);
}}
```

OPERATORS PRECEDENCE

Parentheses	(), inside-out
Increment/decrement	++, --, from left to right
Multiplicative	*, /, %, from left to right
Additive	+, -, from left to right
Relational	<, >, <=, >=, from left to right
Equality	==, !=, from left to right
AND	&&
OR	||
Assignment	=, +=, -=, *=, /=, %=

Control Statements in JAVA

CONTROL STATEMENTS

- control statements are used to control the flow of execution of a program.
- Java‘s program control statements can be put into the following categories:
  - selection,
  - iteration, and
  - jump.

CONTROL STATEMENTS

- Selection statements allow your program to choose different paths of execution based upon the outcome of an expression or the state of a variable(IF,Switch).
- Iteration statements enable program execution to repeat one or more statements .(while , do-while and for loop)
- Jump statements allow your program to execute in a nonlinear fashion.

Java’s Selection Statements

- It allow you to control the flow of your program‘s execution based upon conditions known only during run time.
- Java supports two selection statements : if and switch.
- if statement is Java‘s conditional branch statement. It can be used to route program execution through two different paths.
- The switch statement is Java‘s multiway branch statement. It provides an easy way to dispatch execution to different parts of your code based on the value of an expression.

IF statement

IF

```
if (condition)
statement1;

else

 statement2;
```

If the condition is true, then statement1 is executed. Otherwise, statement2 (if it exists) is executed

Nested IF

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

```
if(i == 10)

          {

   if(j < 20) a = b;

          }

 else a = d;
```

if-else-if ladder

```
if(condition)

statement;

else if(condition)
statement;

else if(condition)
statement;

...

else

statement;
```

Switch statement

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

- The expression must be of type byte, short, int, or char;
- The value of the expression is compared with each of the literal values in the case statements.
- If a match is found, the code sequence following that case statement is executed.
- If none of the constants matches the value of the expression , then the default statement is executed.

Switch STATEMENTS

- Duplicate case values are not allowed.
- default statement is optional.
- If no case matches and no default is present, then no further action is taken.
- The break statement is used to terminate a statement sequence.
- When a break statement is encountered, execution branches to the first line of code that follows the entire switch statement.
- The break statement is optional. If you omit the break, execution will continue on into the next case

// A simple example of the switch.

```
class SampleSwitch
 {
public static void main(String args[])
{ int i=20;
 switch(i)
  { case 10: System.out.println("num is ten.");
                  break;
     case 20: System.out.println("num is twenty.");
                  break;
     case 30: System.out.println("num is thirty.");
                  break;
 default: System.out.println("not in 10,20,or 30");
  }
}
}
```

// A simple example of the switch.

```
public class SwitchMonthExample {
public static void main(String[] args) {
   //Specifying month number
   int month=7;
   String monthString="";
   //Switch statement
   switch(month){
   //case statements within the switch block
   case 1: monthString="1 - January";
   break;
   case 2: monthString="2 - February";
   break;
   case 3: monthString="3 - March";
   break;
   case 4: monthString="4 - April";
   break;
   case 5: monthString="5 - May";
   break;
    case 6: monthString="6 - June";
   break;
   case 7: monthString="7 - July";
   break;
   case 8: monthString="8 - August";
   break;
   case 9: monthString="9 - September";
   break;
   case 10: monthString="10 - October";
   break;
   case 11: monthString="11 - November";
   break;
   case 12: monthString="12 - December";
   break;
   default:System.out.println("Invalid Month!");
   }    //Printing month of the given number
   System.out.println(monthString);
}  }
```

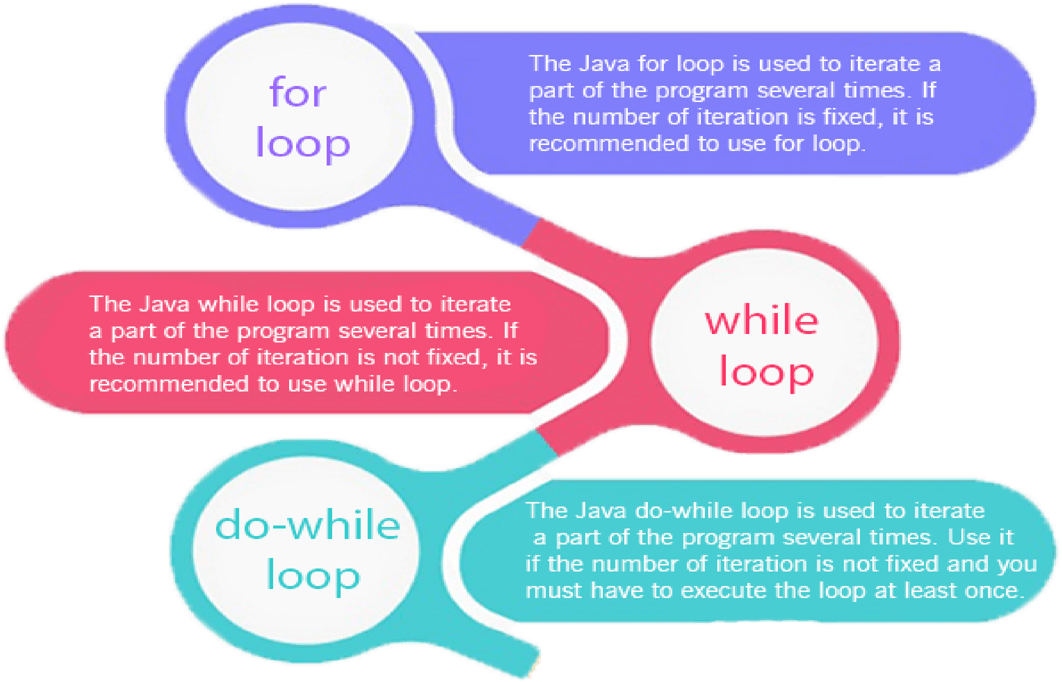
// switch using string.

- Java allows us to use strings in switch expression since Java SE 7.
- The case statement should be string literal.

Example:

```
public class SwitchStringExample
 {
public static void main(String[] args)
{
   //Declaring String variable
   String levelString="Expert";
   int level=0;
   //Using String in Switch expression
   switch(levelString)
{
//Using String Literal in Switch case
 case "Beginner": level=1;
  break;
  case "Intermediate": level=2;
  break;
  case "Expert": level=3;
  break;
  default: level=0;
  break;
  }
  System.out.println("Your Level is: "+level
);
}
}
```

loop STATEMENTS


for loop
The Java for loop is used to iterate a part of the program several times. If the number of iteration is fixed, it is recommended to use for loop.
while loop
The Java while loop is used to iterate a part of the program several times. If the number of iteration is not fixed, it is recommended to use while loop.
do-while loop
The Java do-while loop is used to iterate a part of the program several times. Use it if the number of iteration is not fixed and you must have to execute the loop at least once.

loop statements

for loop

```
for(init;condition;incr/decr)
{
// code to be executed
}
Ex:
for(int i=1;i<=10;i++)
{
System.out.println(i);
}
```

while loop

```
while(condition)
{
//code to be executed
}
Ex:int i=1;
while(i<=10){
System.out.println(i);
i++;  }
```

Do-while loop

```
do{
//code to be executed
}while(condition);
Ex: int i=1;
do{
System.out.println(i);
i++;
}while(i<=10);
```

SIMPLE FOR LOOP

- The Java *for loop* is used to iterate a part of the program several times.
- If the number of iteration is fixed, it is recommended to use for loop.
- A simple for loop is the same as C/C++.
- Initialize the variable, check condition and increment/decrement value.

```
for(initialization ; condition ; incr/decr )
{

//statement or code to be executed

}
```

SIMPLE FOR LOOP SYNTAX

- It consists of four parts:
- **Initialization**: It is the initial condition which is executed once when the loop starts. It is an optional condition.
- **Condition**: It is the second condition which is executed each time to test the condition of the loop. It continues execution until the condition is false. It must return boolean value either true or false. It is an optional condition.
- **Statement**: The statement of the loop is executed each time until the second condition is false.
- **Increment/Decrement**: It increments or decrements the variable value. It is an optional condition.

FOR EACH LOOP

- The for-each loop is used to traverse array or collection in java.
- It is easier to use than simple for loop
- It works on elements basis not index.
- It returns element one by one in the defined variable.
- **Syntax:**

```
for(Type var:array)
{
//code to be executed
}
```

Ex: Printing array using for-each loop

```
public class ForEachExample {
public static void main(String[] args) {
   //Declaring an array
   int arr[]={12,23,44,56,78};
   //Printing array using for-each loop
   for(int i:arr)
  {
      System.out.println(i);
   } } }
```

Infinite for loop

- If you use two semicolons ;; in the for loop, it will be infinitive for loop.
- Syntax:

```
for(;;)
 {
 //code to be executed
}
```

- Ex:

```
public class ForExample {
public static void main(String[] args) {
   //Using no condition in for loop
   for(;;){
      System.out.println("infinitive loop");
   }
}
}
```

Output:

```
infinitive loop
infinitive loop
infinitive loop
infinitive loop
infinitive loop
ctrl+c
```

Do While LOOP

- Java *do-while loop* is used to iterate a part of the program several times.
- If the number of iteration is not fixed and you must have to execute the loop at least once, it is recommended to use do-while loop.
- The Java *do-while loop* is executed at least once because condition is checked after loop body.
- **Syntax:**

```
do{
//code to be executed
}while(condition);
```

Infinite do While LOOP

- If you pass true in the do while loop, it will be infinitive do while loop.
- Syntax:

```
do {
//code to be executed
} while(true)
```

- Example:

```
public class WhileExample2 {
public static void main(String[] args) {
    do {
        System.out.println("infinitive while loop");
    } while(true)  }    }
```

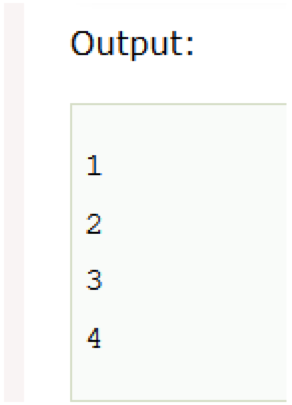
break

- When a break statement is encountered inside a loop, the loop is immediately terminated and the program control resumes at the next statement following the loop.
- The Java *break* is used to break loop or switch statement.
- In case of inner loop, it breaks only inner loop.
- We can use Java break statement in all types of loops such as for loop, while loop and do-while loop.
- **Syntax:**

```
jump-statement;
break;
```

break

```
public class BreakExample {
public static void main(String[] args) {
   //using for loop
   for(int i=1;i<=10;i++){
      if(i==5){  //breaking the loop
         break;
      }         System.out.println(i);
   }  }}
```



Output:

```
1
2
3
4
```

continue

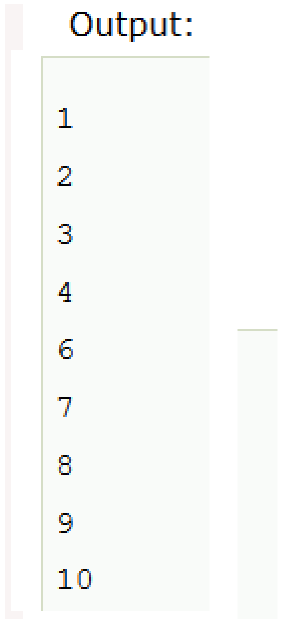
- The continue statement is used in loop control structure when you need to jump to the next iteration of the loop immediately. It can be used with for loop or while loop.
- The Java *continue statement* is used to continue the loop.
- It continues the current flow of the program and skips the remaining code at the specified condition. In case of an inner loop, it continues the inner loop only.
- **Syntax:**

jump-statement;

continue;

continue

```
public class ContinueExample {
public static void main(String[] args) {
  //for loop
  for(int i=1;i<=10;i++){
    if(i==5){
      //using continue statement
      continue;//it will skip the rest statement
    }      System.out.println(i);
  } } }
```



```
Output:

1
2
3
4
6
7
8
9
10
```

Reading i/p

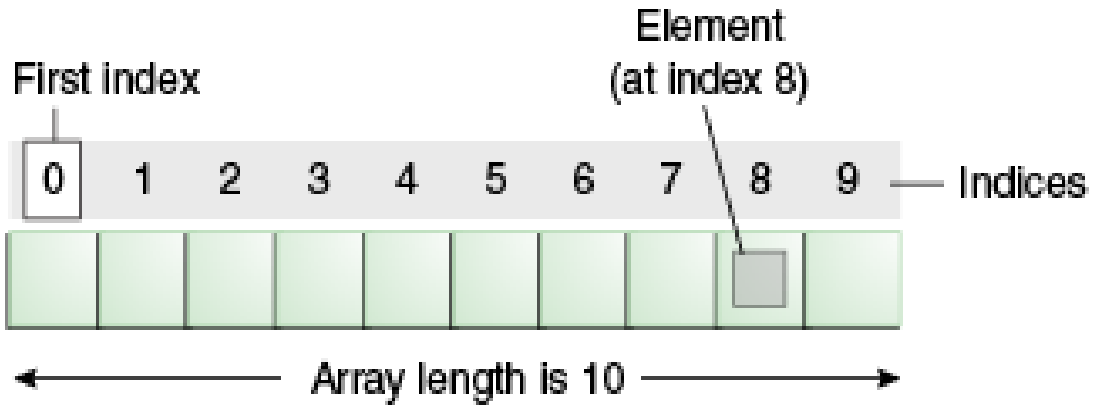
- Scanner is a class in java.util package used for obtaining the input of the primitive types like int, double etc. and strings.
- It is the easiest way to read input in a Java program
  - Scanner s=new Scanner(System.in);
  - n=s.nextInt();

```
Scanner sc = new Scanner(System.in);
  // String input
 System.out.println("Enter the Name:");
 String name = sc.nextLine();
// Numerical data input,byte, short and float can be read
   System.out.println("Enter the age:");
 int age = sc.nextInt();
 System.out.println("Enter the Mobile Number:");
 long mobileNo = sc.nextLong();
 System.out.println("Enter the cgpa:");
 double cgpa = sc.nextDouble();
```

ARRAYS

Array

- an array is a collection of similar type of elements which have a contiguous memory location
- The elements of an array are stored in a contiguous memory location.
- It is a data structure where we store similar elements.
- We can store only a fixed set of elements in a Java array.


Element
(at index 8)
First index
0 1 2 3 4 5 6 7 8 9
Indices
Array length is 10

- **Syntax to Declare an Array in Java**
  - dataType[] arr; (or)
  - dataType []arr; (or)
  - dataType arr[];
- **Instantiation of an Array in Java**
  - arrayRefVar=**new** datatype[size];