Object Oriented Programming Using Java 1.8 (Spider)

Contents:

1. Data Types and Conversion
2. Wrapper Classes
3. Classes and Objects

Data Types

- Java is a strongly typed language.
- Every variable and expression has a type.
- All types are well defined.
- Java strictly checks type compatibility while doing assignments.

Data Types

- Java defines eight primitive types - byte, short, int, long, float, double, char, boolean
- These can be put in four groups:

  Integers : byte, short, int, long Floating

  points numbers : float, double

  Characters : char

  Boolean : boolean

Data Types

- The primitive type represents single values and not complex object.
- Unlike c/c++ (which depends upon the execution environment), Java has strictly defined range for all primitive types. For example size of an int type data in C/C++ may vary but not in Java to achieve portability.

Data Types

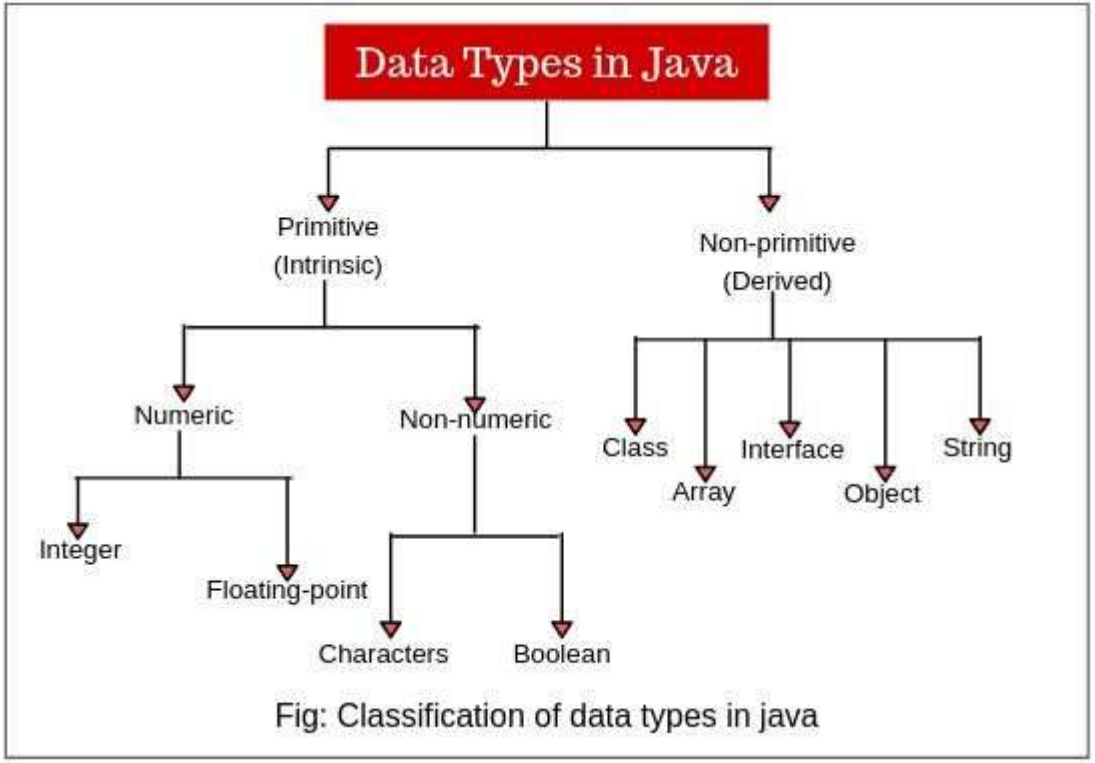

Fig: Classification of data types in java

Data Types

Data Type	Size	Range of values that can be stored	Default value
byte	1 byte	−128 to 127	0
short	2 bytes	−32768 to 32767	0
int	4 bytes	−2,147,483,648 to 2,147,483,647	0
long	8 bytes	-9,223,372,036,854,775,808 to 9223372036854750000	0
float	4 bytes	3.4e−038 to 3.4e+038	0.0f
double	8 bytes	1.7e−308 to 1.7e+038	0.0d
boolean	1 bit	true or false	false
char	2 bytes		\u0000

Data Types

- Java does not support unsigned positive-only integers as many other languages do. Ex. in C signed int, unsigned int etc.
- The java run-time environment is free to use whatever size it wants, as long as the types behave as you declared them. Infact one implementation stores bytes and shorts as 32 bit (rather than 8-bit and 16-bit) values to improve performance because that is the word size of most computers currently.

Data Types

Use of byte

- Stream of data over network
- Working with raw binary data not compatible with other built-in types.

Use of short

- It is probably the least-used java type, mostly useful in 16 bit computers.

Data Types

Use of int

- It may seem that using short or byte will save space, but there is no guarantee that Java won’t promote those types to int internally anyway. Remember, type determines behavior, not size.
- The only exception is arrays, where byte is guaranteed to use only one byte per array element, short will use two bytes, and int will use four.

Data Types

Use of long

- When range is out of int type, use long. Ex. Calculate distance travelled by light in a given number of days (e.g. 10000). Please note that light travels 186000 miles approx in a second.

Use of float

- To store the real numbers that require fractional precision. Type float specifies a single-precision value that uses 32 bits of storage.
- Single precision will become imprecise when the values are either very large or very small.
- Use float when you don’t require a large degree of precision.

Data Types

Use of double

- It specifies a double-precision value.
- Use double when you need many iterative calculations (like sqrt, sin, cos etc.) or manipulate large valued numbers.
- The extra bits in double, increase not only the precision but also the range of magnitudes that can be represented.

Data Types

Use of char

- Java uses Unichar to represent characters which is an international character set including all the characters found in all human languages.
- Standard set of characters ASCII range : 0 to 127
- Extended 8-bit character set, ISO-Latin-1 range : 0 to 255
- First 127 values in the Unicode character set is same as ASCII set.

Data Types

- Java allows to add two characters and increment the value of character.

  Ex: char ch = ‘J’;

  ch ++;

  S.o.p(ch); //Will print K

Data Types

Use of boolean

- To store logical values (true or false)

  Ex. boolean b = false;

- Boolean is also the type required by the conditional expressions that govern the control statements used in if and for.
- Outcome of a relational operator is a boolean value.

  Ex. boolean res = (a > b);

Variables

Variables

- Basic unit of storage in a program.
- Basic form of variable

  type identifier [= value];

  Where the identifier is the name of the variable.

- Ex. int num = 100, char ch, String name etc.
- We can not use Java keywords as a variable name.

Java Keywords

abstarct	continue	for	new	switch
assert	default	goto	package	synchronized
boolean	do	if	private	this
break	double	implements	protected	throw
byte	else	import	public	throws
case	enum	instanceof	return	transient
catch	extends	int	short	try
char	final	interface	static	void
class	finally	long	strictfp	volatile
const	float	native	super	while

Variables

Scope of a variable

- Scope - Begins with opening curly bracket and ends with closing curly bracket.
- The variables declared in the outer scope will be visible to code within inner scope. However the reverse is not true.
- Within a block, variables can be declared at any point.
- Variables are created when their scope is entered and destroyed when their scope is left.

Variables

Scope of a variable

Although blocks can be nested, we can not declare a variable to have the same name as one in an outer scope.

```
Class ScopeDemo
{
P.s.v.m. ( String args [ ] )
{       int num = 10;
        {
                int num = 20; // Compile-time error, num is already defined.
        }
}
}
```

Variables

Scope of a variable

- A variable must be initialized (not necessarily at the time of declaration) before its use in the code otherwise the compiler will give an error “variable might not have been initialized”.

Literals

Literals

- Literals are numbers, text, or anything that represent a value. In other words, Literals in Java are the constant values assigned to the variable. Ex. 235, “Hello”, ‘Y’ etc.

Literals

Integer literals

- Integer literals in java can be represented in any of the following form.

  Decimal (Base 10) - Ex . 93

  Octal (Base 8) - Ex. **0**135 (Leading with zero)

  Hexadecimal (Base 16) - Ex. **0x**5D (Leading with zero and small/capital X)

Literals

Integer literals Contd..

- To assign a long value you need to explicitly tell the compiler that it is a long value by appending lowercase or uppercase L. ex. long p = 0xffffffffL or long p = 23654777788L
- By **default** each integer literal type is treated as **int**.

Literals

Floating point literals

Floating point literal in java default to double precision.

To specify a float literal, you must append f or F to the constant.

Boolean literals:

In java true does not mean 1 and false does not mean 0.

Literals

Character literals:

- A character literal is represented by using single quote. Ex. ‘A’, ‘a’ etc.
- Using escape sequence to represent special characters like ‘\’’ to represent single quote.
- \n for newline character
- Representing character using octal or hexadecimal values. Ex. ‘\141’ (three digit Octal for ‘a’), ‘\u0061 (4 digit hexadecimal for ‘a’, see \u for hexadecimal).

Literals

String literals

- Enclosing sequence of characters between pair of double quotes.
- Ex. “Hello World”, “Line\nBreak” , “\”Quotes\”” etc.In java String must begin and end on the same line.
- There is no line-continuation escape sequence.

  Ex.

  “Hello

  World”

  Is wrong.

Literals

Escape Sequences

\ddd - Octal character

\uxxxx - Hexadecimal unicode character

\’ - single quote

\” - double quote

\\ - backslash

\n - new line (line feed)

Literals

Escape Sequences

\t - Tab

\b - backspace

Type Conversion

Type Conversion and Casting

- It is common to assign value of one type to another type. Ex. assigning int value to a long variable.
- If the two types are compatible, java will perform automatic conversion. Ex. int to long (Automatic type conversion by java - implicit)
- It is possible to obtain conversion between incompatible types. Ex. double to byte (Typecasting - explicit)

Type Conversion

Automatic type conversion (Implicit Conversion) conditions

- The two types are compatible.
- Destination type is larger than the source type. (Also called, Widening Conversion). Ex. Assigning byte to int as int type is always large enough to hold byte values.
- All numeric types (int, float, double) are type compatible with each other for widening conversion.
- Ex.
  int a ; byte b = 10 ; a = b;

Type Conversion

Explicit Conversion (Typecasting)

Syntax:
(Target-Type) value

Assigning larger type to smaller type. (Also called, Narrowing Conversion). Ex. Assigning int to byte, float to int.

Ex. int a = 10;

byte b;

b = (byte) a;

Type Conversion

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

  Ex.

  int i = 258;

  byte b ;

  b = (byte) i ;

  s.o.p(i); // Prints 258

  s.o.p(b); // Prints 2 (remainder) i.e. result of modulo operation 258 % 256 = 2

Type Conversion

- When a double value is converted to int, its fractional component is lost.

  Ex.

  int i ;

  double d = 258.73;

  i = d;

  S.o.p (i); // Prints 258

Type Conversion

- When a double value is converted to byte, its fractional component is lost and value is reduced modulo 256.

  Ex.

  byte b ;

  double d = 258.73;

  b = (byte) d;

  S.o.p (b); // Prints 2 (remainder) i.e. result of modulo operation 258 % 256 = 2

Type Conversion

- When a double value is converted to float

```
float f = 85.0;
```

Error: incompatible types: possible lossy conversion from double to float.

```
float f = 85.0f;
```

Type Conversion

- In the widening conversion, high order bits of target type is filled up with the highest bit of the given value. This way, the value's sign is retained.

  byte (2) => 00000010
  Short (2) => 00000000 000000010

- Exception is widening of char. As char are unsigned, high order bits of target type is filled up with 0.

  (char) \uFFFF => 11111111 11111111
  (int) 65535 => 00000000 00000000 11111111 11111111

Type Conversion

- In the narrowing conversion, where the original type has more bits than the target type, the additional bits are merely cut off.

  Short (2) => 0000001 00000100 // 260
  byte (1) => 00000100 // 4

  (int) 65535 => 00000000 00000000 11111111 11111111
  (char) \uFFFF => 11111111 11111111

Type Promotion

- If an expression contains multiple type variables, the type of each variable is first promoted to the largest type variable in the expression and also evaluates the result to that largest type.

- Java promotes each byte or short operand to int when evaluating the expression.

  Ex. 1.

  byte a = 20, b = 30, c = 100;

  int res = a * b * c; // a, b, c promoted to int during evaluation

Identifiers

Identifiers

A name in the program is an identifier it may be class name or method name, variable name or label name.

Example

```
Class Demo{

public static void main(String[] args) {

int a = 5;

System.out.println(a);

}

}
```

Identifiers

Rules for defining Identifiers

- Identifiers cannot start with a number.
- Identifiers must start with a letter, a currency character ($), or a connecting character such as the underscore (_)
- After the first character, identifiers can contain any combination of letters, currency characters, connecting characters, or numbers.
- In practice, there is no limit to the number of characters an identifier can contain.
- You can't use a Java keyword as an identifier.
- Identifiers in Java are case-sensitive; num and NUM are two different identifiers.
- Upper camel case ex: MyClass
- Lower camel case ex: myClassObj

Identifiers

Name	Convention
class name	should start with uppercase letter and be a noun e.g. String, Color, Button, System, Thread etc.
interface name	should start with uppercase letter and be an adjective e.g. Runnable, Remote, ActionListener etc.
method name	should start with lowercase letter and be a verb e.g. actionPerformed(), main(), print(), println() etc.
variable name	should start with lowercase letter e.g. firstName, orderNumber etc.
package name	should be in lowercase letter e.g. java, lang, sql, util etc.
constants name	should be in uppercase letter. e.g. RED, YELLOW, MAX_PRIORITY etc.

Wrapper Classes

Wrapper Classes

- Java uses primitive types, such as int and char for performance reasons.
- The primitive data are passed by value and can not be directly passed by reference.
- There is no way for two methods to refer to the same instance of an int.
- There are some places where only objects are required to work with like collection classes.
- In that case we need to wrap the primitive type in a class.
- Java provides classes that correspond to each of the primitive types. These classes wrap, or encapsulate the primitive types within a class. So, they are known as wrapper classes.

Wrapper Classes

Superclass: Number

- This is an abstract class and is implemented by all the wrapper classes defined for numeric types byte, short, int, long, float and double.
- Following are the abstract methods inside Number

  byte byteValue() : Returns the value as byte
  float floatValue() : Returns the value as float
  double doubleValue() : Returns the value as double
  int intValue() : Returns the value as int
  short shortValue() : Returns the value as short
  long longValue() : Returns the value as long

- The calling object can be an object of Byte, Short, Integer, Long, Float or Double class
- The values returned by these methods can be rounded.

Wrapper Classes

Double and Float

Constructors (Float)

Float (double num)

Float (float num)

Float (String str) throws NumberFormatException

Constructors (Double)

Double (double num)

Double (String str) throws NumberFormatException

Wrapper Classes

Float: Some Important Methods

int compareTo(Float obj)

Compares the numeric value of two Float class objects and return 0, -ve value, or +ve value

static float parseFloat(String str)

Return float equivalent of the string str

String toString()

Converts Float object into String Object

static Float valueOf(String str)

Converts a string str that contains some float number into Float object

*** Similar methods for other Wrapper classes for different numeric types**

Wrapper Classes

Float: Some Important Methods

Using Constructor for creating Float object

Float f=12.122f;

Float obj=new Float(f);

String str=“12.122f”;

Float obj=new Float(str);

Wrapper Classes

Boolean class

The Boolean class wraps a value of the primitive type boolean in an object. The Boolean class object contains a boolean type field that stores a primitive boolean

Constructor

Boolean(boolean value);

Boolean (String str)

Wrapper Classes

Boolean: Some Important methods

int compareTo(Boolean obj)

Compares the value of two Boolean class objects and return 0, -ve value, or +ve value

static boolean parseBoolean(String str)

Return boolean equivalent of the string str

String toString()

Converts Boolean object into String Object

static Boolean valueOf(String str)

Converts a string str that contains some boolean value into Boolean object

Wrapper Classes

Character Class

The Character class wraps a value of the primitive type char in an object.

Character class has only one constructor which accepts primitive data type

Character obj=new Character('A');

Wrapper Classes

Important methods of Character class

char charValue ( ):used to convert character object into character primitive

Character obj=new Character ('A');

char ch=obj.charValue();

int compareTo (Character obj):useful to compare two character objects

int x =obj1.compare (obj2);

if obj1==obj2,returns 0

if obj1<obj2,returns negative value

if obj1>obj2,returns positive value

String toString( ):converts character object into string object

Wrapper Classes

Important methods of Character class

static character value of (char ch)

convert a single character ch into character object

static boolean isDigit (char ch)

returns true if ch is a digit otherwise return false

static boolean isLetter(char ch)

returns true if ch is a letter

static boolean is UpperCase(char ch)

returns true if ch is a uppercase letter

static boolean is LowerCase(char ch)

returns true if ch is lower case letter

Wrapper Classes

Important methods of Character class

static boolean is SpaceChar(char ch)

returns true if ch is represents a white space

static boolean is LetterorDigit(char ch)

returns true if ch either a letter or digit

static char toUpperCase(char ch)

converts ch into uppercase

static char toLowerCase(char ch)

converts ch into lowercase

Wrapper Classes

Important methods of Character class

static boolean is SpaceChar(char ch)

returns true if ch is represents a white space

static boolean is LetterorDigit(char ch)

returns true if ch either a letter or digit

static char toUpperCase(char ch)

converts ch into uppercase

static char toLowerCase(char ch)

converts ch into lowercase

Wrapper Classes

- To convert a whole number into a decimal string, we can use versions of toString defined in various wrapper classes like Byte, Short, Integer or Long.

- The Integer and Long classes also provide the static methods toBinaryString(), to HexString(), and toOctalString(), which convert a vlaue into a binary, hexadecimal, or octal string, resepectively.

- Ex. Integer.toBinaryString(intWrapperObj);

  Integer.toBinaryString(integerPrimitiveData);

Wrapper Classes

Conversion Between Type Wrapper and Primitive Type (Boxing and Unboxing)

The process of encapsulating a value within an object is called **boxing**.

Integer iob = new Integer(10);

The process of extracting a value from a type wrapper is called **unboxing**.

int i = iob.intValue( );

Wrapper Classes

Autoboxing & Auto-unboxing

Autoboxing is the process by which a primitive type is automatically encapsulated (boxed) into its equivalent type wrapper whenever an object of that type is needed. There is no need to explicitly construct an object.

Auto-unboxing is the process by which the value of a boxed object is automatically extracted (unboxed) from a type wrapper when its value is needed. There is no need to call a method such as intValue( ) or doubleValue( ).

Wrapper Classes

Autoboxing & Auto-unboxing

```
Integer iob = 100;  //autobox an int

int i = iob;       //auto-unbox

System.out.println( i + ”  “ + iob); // displays 100 100
```

Wrapper Classes

Autoboxing & Auto-unboxing

Autoboxing/unboxing might occur when an argument is passed to a method, or when a value is returned by a method.

```
class BoxingDemo {

int fun(Integer v) {

        return v;

}

public static void main(String args[ ]) {

        Integer iob = fun(100);

        System.out.println(iob);

}

}
```

Wrapper Classes

Autoboxing & Auto-unboxing

Autoboxing/unboxing applies to expressions also.

Ex. Integer iob = 100;

++iob

Autoboxing and unboxing allows to mix different type of numeric objects in an expression.

Integer iob = 100;

Double dob = 98.6;

dob = iob + dob

System.out.println(dob) ; // 198.6

Wrapper Classes

Autoboxing & Auto-unboxing

Because of auto-unboxing, we can use Integer numeric objects to control a swtich statement.

Ex. Integer iob = 2;

Switch (iob) { …. } // iob is unboxed and its int value is obtained

Wrapper Classes

Autoboxing & Auto-unboxing

It applies to Boolean and Character wrapper types also.

Character ch = ‘x’ ; // box a char

char ch2 = ch; // unbox a char

Boolean b = true; // box a boolean

boolean b2 = b;

Because of unboxing a Boolean object can be used to check conditions wherever boolean values are used like in if statement and while conditions.

Ex. Boolean b = true; if (b) { …. }

Wrapper Classes

Autoboxing & Auto-unboxing

Autobox and auto-unbox **adds overhead** which is not present when the primitive type is used.

Bad use of autoboxing/unboxing:

```
Double a, b, c;

a = 5.0; b = 10.0;

c = Math.sqrt(a * a + b*b) ;        // This could be done using primitive types only.
```

In general, **we should restrict use of the type wrappers** to only those cases in which an object representation of a primitive type is required. Autoboxing / unboxing was not added to java as a “back door” way of eliminating primitive types.

Classes & Objects

Classes & Objects

- What is Class ?
- What is Object ?
- A simple class
- Creating Objects
- Assigning Object Reference Variables (Emp e1 = e2)
- Methods (With paratmeters and return)
- Constructors
- finalize

Classes & Objects

- Parameterized Constructors
- “this” keyword
- Instance variable hiding / Variable Shadowing
- Garbage Collection
- finalize method
- method overloading
- Constructor overloading

Classes & Objects

- Using Objects as Parameters
- Argument passing (Call by value and call by reference)
- Returning objects
- Static member of class
- Introducing Access Control
- final keyword and its use

Classes & Objects

Object

- An entity which does exist, has state and behavior is known as an object e.g. chair, bike, marker, pen, table, car etc.
- If something does not really exist, then it is not an object e.g. our thoughts, imagination, plans, ideas etc.,
- According to System existence means contains memory. So a software object represent a memory.
- Word object and instance used interchangeably.

Classes & Objects

Class

- It is possible that some objects may have similar properties and actions. Such objects belong to same category called a ‘class’
- It is only a logical component and not the physical entity e.g. If we have class of “Expensive Cars” it could have objects like Mercedes, BMW, Toyota, etc.
- Properties of car are price, speed etc.
- Class defines a new data type. Class is a template for an object and an object is an instance of a class.

Classes & Objects

Class

- It is possible that some objects may have similar properties and actions. Such objects belong to same category called a ‘class’
- It is only a logical component and not the physical entity e.g. If we have class of “Expensive Cars” it could have objects like Mercedes, BMW, Toyota, etc.
- Properties of car are price, speed etc.

Classes & Objects

Class declaration syntax

<modifier> class <ClassName>{

//Class body with variables and methods

}

<modifier> => public, default (not a keyword), final, strictfp (For all classes)
<modifier> => public, default (not a keyword), private, protected, final, strictfp, static (For inner classes)

Classes & Objects

Class modifiers

- A class may be Access modifiers: public, protected, and private
- Modifier requiring override: abstract
- Modifier restricting to one instance: static
- Modifier prohibiting value modification: final
- Modifier forcing strict floating point behavior: strictfp
- declared with one or more modifiers which affect its runtime behavior.

Classes & Objects

Class Creation: Example.

```
class Cricketer
{
        //Data/Properties
        String name;
        String country;
        int totalMatches;
        // Methods/Behaviours/functions
        void getDetails();
        void displayDetails();
}
```

Classes & Objects

Constructor

- A constructor initializes an object immediately upon creation. Once defined, the constructor is automatically called immediately after the object is created before the new operator completes.
- The implicit return type of a class' constructor is the class type itself.
- It constructs the values i.e. provides data for the object that is why it is known as constructor.
- Name of the constructor must be same as that of a class name.
  Constructors must not have a return type. If we keep return type for the constructor, it will be treated as another method.
- If we don't write any constructor, compiler gives default constructor.
- Constructor can be declared private. In this case it can not be accessed from outside the class.

Classes & Objects

Constructor

- There can be multiple overloaded constructor inside a class.
- Only public, private, protected keyword are allowed before any constructor name. Applying any other keywords (like static, final etc.) will give compilation error.
  Ex. **static** Cricketer() { } // **Error**

Classes & Objects

‘this’ keyword

- Reference to the current object. That is, this is always a reference to an object on which method was invoked.
- ‘this’ can be used to call the overloaded constructor from the other constructors within the same class.
  But it should always be the first statement within the constructor.
  Ex.

```
Cricketer(int totRun){
        this();                 //Calling no argument constructor of Cricketer class.
        //Other code
```

Classes & Objects

- this ( ) can not be used within any other methods other than constructors

```
void updateTotalRuns(int totRuns)
{
        //this();       //Compilation Error,  this must be first statement in constructor
        totalRuns = totRuns;
}
```

Classes & Objects

Variable Shadowing

- When a local variable has the same name as an instance variable, the local variable hides the instance variable. We can use ‘this’ to resolve this name space collisions.

```
void updateTotalRuns(int totalRuns)
{
    //this();                  //Compilation Error,  this must be first statement in constructor
    this.totalRuns = totalRuns; // Resolving variable shadowing
}
```

Classes & Objects

Method Overloading

- Two or more methods within the same class that share the same name with different parameter list.
- The overloaded methods must differ in the type and/ or number of their parameters.
- Overloaded methods can have different return types.
- Constructors can also be overloaded.

Classes & Objects

Different Type of Variables used inside Class

- static variable
- Instance variable
- local variable

Classes & Objects

Instance variable

- Variables that are part of each object or we can say each instance of class contains its own copy of these variables.
- Instance variables have default values. For numbers, the default value is 0, for Booleans it is false, and for object references it is null.
- Values can be assigned **during the declaration** or within the **constructor**.

Local variable

- A variable which is declared inside the methods, constructors, or blocks is called local variable.

Classes & Objects

Static variable

- Class variables also known as static variables are declared with the static keyword in a class.
- There would only be one copy of each class , regardless of how many objects are created from it
- Default values are same as instance variables. Values can be assigned **during the declaration** or within the **constructor**. Additionally, values can be assigned in special **static initializer blocks**.
- Static variables can be accessed by calling the class name ClassName.VariableName.

Classes & Objects

Static block

- Static block is used for initializing the static variables.
- A static keyword is prefixed before the start of the block.
- This block gets executes only once when the class is loaded in the memory.
- A class can have multiple Static blocks, which will execute in the same sequence in which they have been written into the program.
- Inside static block all static variables can be accessed freely
- Instance variables can also be accessed but only through object reference after object creation.
- Syntax:

```
Class ABC{
        static
        {
                //static block
        }
}
```

Classes & Objects

Non-static block

- A non-static block executes when the object is created, before the constructor.
- Unlike static block, no keyword is prefixed before the start of the block.
- This block gets executed every time when any object of the class is created.
- A class can have multiple non-static blocks, which will execute in the same sequence in which they have been written into the program.
- Inside non-static block all static and non-static variables can be accessed freely
- Syntax:

```
Class ABC{
        {
            //non-static block
        }
}
```

Classes & Objects

Use of ‘static’ keyword within class

- static keyword can be applied on data member as well as member functions.
- When a class member is declared static, it can be accessed before any objects of its class are created, and without reference to any object using class-name.static-member

Static data member

- When objects of a class are declared/created, no copy of a static variable is made. Instead all instances of the class share the same static variable.

static member functions

Methods declared as static have several restrictions

- *They can only call other static methods*
- *They can only access static data*

Classes & Objects

Call by value & Call by reference

- Primitive type data are passed by value.
- Object are passed by using reference but that reference is passed by value.