Chapter 10

Variables and Types of Variables

- In this chapter, You will learn
  - Definition of variable
  - Limitation of variable
  - Variables creation syntax
  - Types of variables
  - Local variable and its rules
  - Modifiers allowed for variables
  - 8 Types of members defined in a single class and their execution control flow
- By the end of this chapter- you will learn defining, declaring, calling variables, and its execution control flow.

Interview Questions

By the end of this chapter you answer all below interview questions

- Definition
- How can we create a variable?
- Limitation on variable
- Defining, declaring, initializing, reinitializing, and calling/using a variable
- Types of variables
    - Method Level
        - Parameter
        - Local
            - final
    - Class Level
        - static
        - non-static
            - final
            - transient
            - volatile
- Rules on local variables and Parameters?
- What are the modifiers allowed for a variable?
- Default values for variables
- Life-time and scope of a variable
- How many members we can define inside a class?

iii

Variables and Types of Variables

Definition
Variable is a named memory location used to store data temporarily. During program execution we can modify that data.

How can a variable be created?
A variable can be created by using Datatype. As we have two types of datatypes we can create two types of variables

1. **Primitive variables** - These variables are created by using primitive datatypes.
2. **Referenced variables** - These variables are created by using referenced datatypes.

The difference between primitive and referenced variables is "primitive variables stores data directly, where as referenced variables stores reference of the object, not direct values".

Note: As per compiler and JVM we do not have a separate name called "referenced variable". It means, the variables created by using referenced datatype are also considered as like normal variables only.

Below program shows creating primitive and referenced variables

```
//Example.java
class Example
{
        int x = 10;
        int y = 20;
}
```

```
//Test.java
class Test{
        public static void main(String[] args) {
                //primitive variables
                int     p = 50;
                int     q = m1();
                //referenced variables
                String  s1 = "a";
                String  s2 = new String("a");

                Example e = new Example();
        }
        static int m1() {
                return 60;
        }
}
```

Referenced variables are initialized with object reference that is created and returned by "new" keyword, as shown in the left diagram.

Limitation of variable
It can only store single value at a time. If we assign new value, old value is replaced with new value. It means, always it returns latest modified value.
So

- If we modify primitive value previous value is replaced with new value
- If we modify referenced variable previous object's reference is replaced with new object's reference and now this referenced variable is referencing to this new object.

Show memory structure from the below program and also output

```
class Sample {
        public static void main(String[] args) {
                int a = 50;
                System.out.println("a: "+a);

                a = 70;
                System.out.println("a: "+a);

                Example e1 = new Example();
                System.out.println("e1: "+e1);

                e1 = new Example();
                System.out.println("e1: "+e1);

        }
}
```

Output:

a: 50
a: 70
e1: Example@addbf1
e1: Example@42e816

Conclusion:

- The value of primitive variable is mathematical data based on its data type.
- The value of referenced variable is its class object's reference.

Defining, declaring, initializing, reinitializing, and calling/using a variable

Defining a variable

Variable creation with value is called defining a variable.

Syntax:

```
  OP                          OP          M              M          M    M    M
<Accessibility Modifier> <Modifier> <datatype> <variablename> = <value> ;
```

Ex:

```
public static int        x = 10;
public static Example  e = new Example();
```

Declaring a variable

Variable creation without value is called declaring a variable.

Syntax:

```
  OP                          OP          M              M          M
<Accessibility Modifier> <Modifier> <datatype> <variablename> ;
```

Ex:

```
public static int x;
public static Example  e ;
```

Assigning and reassigning a variable

Storing a value in a variable at the time of its creation is called initializing a variable.
Storing a value in a variable after its creation is called *assigning* a value to a variable.

Syntax:

```
    M          M   M    M
<variablename> = <value>;
```

Ex:

```
//declaring variables
int p;
Example e;

//assigning variables
p = 50;
e = new Example();

// reassignment
p = 70
e = new Example();
```

Calling a variable

Reading a value from a variable is called calling a variable.

Ex:

```
int x = 10;

//calling x variable for printing
System.out.println(x); // 10

//calling x variable for initializing y
int y = x;
```

Q) What is the difference between executing a variable and executing a method?

Executing a variable means creating variable memory location with initialized value. JVM executes variables automatically when variable creation statement (definition or declaration) is appeared in the program.

Executing a method means executing method logic. JVM does not execute method logic automatically. It executes method only when that method calling statement is appeared, not by using just definition.

Q) What is the difference between variable calling and method calling?

Calling a variable means *reading its value*.
Calling a method means *executing that method logic*.

Ex:

```
class Test{

    static void m1(){
        System.out.println("m1");
    }

    public static void main(String[] args){
        int x = 10;

        //from below statement x variable value is read, variable is not executed again.
        System.out.println(x); //10          <= variable calling statement

        //from the below statement method is executed.
        m1();          <= method calling statement
    }
}
```

Types of Variables

Based on class scopes variables are divided in two types

1. Local Variables, parameters
2. Class Level Variables

- The variables created inside a method (or) block are called local variables
- The variables created at class level are called class level variables.

2 types of class level variables

Class level variables are divided into two types, based on the time they are getting memory location. They are

- Static variables
- Non-static variables

Definitions

The class level variable which has static keyword in its creation statement is called static variable, else it is called non-static variable.

Memory location of all above three variables

- *Local variables* get memory location when method is called and their creation statement is executed. They get memory with respect to method, so they are also called *method variables*. Local variables are automatically created when method is executing and are destroyed automatically after method execution is completed, so they are also called *auto variables*.
- *Static variables* get memory location when class is loaded into JVM. They get memory with respect to class name, so they are also called "*class variables*" also called "*Fields*".
- *Non-static variables* get memory location when object is created using *new* keyword. They get memory with respect to object, so they are also called "*object variables* or *instance variables* or *properties* or *attributes*" are also called "*Fields*".

Below diagram shows different scopes in class and all above three variables creation.

```
class Example{
        //static variables
        static int a = 10;
        static int b = 20;

        //non-static variables
        int x = 30;
        int y = 40;

        public static void main(String[] args)
        {
                //local variables
                int p = 50;
                int q = 60;
        }
}
```

Q) How many variables are created in above program?

A) 5 variables. They are *a, b, args, p, q*.

Variables **x, y** are not created, because object is not created.

Local variables and its rules

While working with local variables, we must follow below 3 rules

Rule #1: Local variable cannot be accessed from another method. Because its scope is restricted only within its method, also we cannot guarantee variable creation, because that method may or may not be called. It leads to **CE: cannot find symbol**

For example:

```
class Example
{
        public static void main(String[] args)
        {
                int a = 10;
                System.out.println("a: "+a);
        }
        static void m1()
        {
                System.out.println("a: "+a);   CE: cannot find symbol
        }                                        symbol: variable a
}
```

Rule #2: Local variable should not be accessed without initialization.
It leads to **CE: "variable might not have been initialized"**

```
class Example {
        public static void main(String[] args){
                int a = 10;
                int b;

                System.out.pritnln("a: "+ a);
                //System.out.pritnln("b: "+b);   //CE: variable b might not have been initialized

                a = a + 10;
                //b = b + 10;   //CE: variable b might not have been initialized

                b = 20;
                System.out.pritnln("b: "+ b);

                b = b + 10;

                System.out.pritnln("a: "+ a);
                System.out.pritnln("b: "+ b);
        }
}
```

Rule #3: local variable must be accessed only after its creation statement; because method execution is sequential execution from top to bottom. If we access it before its creation statement it leads CE: "**cannot find symbol**"

```
class Example{
    static void m1(){
        System.out.println("a: "+a);   ✗
        int a;
        System.out.println("a: "+a);   ✗
        a = 10;
        System.out.println("a: "+a);   ✓
    }
    static void m2(){
        System.out.println("a: "+a);   ✗
    }
}
```

Q) How can we access a local variable value from other methods?

A) There are two ways

1. Pass it as an argument
2. Store it using a class level variable

- If we want to use a value in one or two methods, store it using local variable in one method and pass it as an argument to another method. This approach is recommended to save memory.
- If we want to use a value from multiple methods of a class, store it using class level variables.

Q) What is the output from the below program?

```
class Example{
    static void m1(){
        int p = 10;
        System.out.print(p);
        m2();
    }
    static void m2(){
        int q = p + 10;   ✗ CE: c f s
        System.out.print(q);
    }
    public staic void main(String[] args)
    {
        m1();
    }
}
```

1. 10 10
2. 10 20 ✓
3. CE
4. RE

Q) In the above program, if we create p variable **at class level as static, what is the output?**

A) No CE, output is: 10 20

Q) In the above program, if we create p variable **at class level as non-static, what is the output?** A) It leads to **CE**: non-static variable p cannot be referenced from static context

Class Level variables

We must create class level variable only if we want to access a value throughout the class from all its methods.

Q) What is the output from the below program?

```
class Example{
        int x = 10;

        public static void main(String[] args){
                System.out.println(x);
        }
}
```

Choose one option

1. 10
2. CE
3. RE
4. No output

Rule: Non-static variables and methods must be accessed with object from static methods; else it leads to above CE: *non-static variable cannot be referenced from static context*, because non-static variable and non-static method does not get memory location directly at the time of class loading.

Below program shows creating static, non-static, and local variables and accessing them from main method (static method).

```
class Example{
        static int a = 10;
        static int b = 20;

        int x = 30;
        int y = 40;

        public static void main(String[] args) {
                int p = 50;
                int q = 60;

                System.out.println("a: "+a);
                System.out.println("b: "+b);

                //System.out.println("x: "+x); CE:
                //System.out.println("y: "+y); CE:

                Example e = new Example();
                System.out.println("x: "+e.x);
                System.out.println("y: "+e.y);

                System.out.println("p: "+p);
                System.out.println("q: "+q);
        }
}
```

Q) Can we declare *local variable* or *parameter* as static?
No, local variables can't be declared as static it leads to CE: *illegal start of expression*. Because local variable should get memory location only if method is called. But if we declared as static, it should get memory at the time of class loading, this is violating contract, so it leads to compile time error.

For Example

```
class Example{
        static int a = 10;  ✓

        static void m1(){
                int p = 20;  ✓
                static int q = 30;  ✗ CE: illegal start of expression
        }
}
```

Q) The variables created inside a static method are static? => No, they are still local
Q) The variables created inside a non-static method are non-static? => No, they are still local

Final variables
The class level or local variable that has final keyword in its definition is called final variable.

Rule: once it is initialized by developer its value cannot be changed. If we try to change its value it leads to compile time error.

Below program shows creation final variables

```
class Example{

        static int a = 10;              <= normal static variable
        static final int b = 20;        <= final static variable

        int x = 30;                     <= normal non-static variable
        final int x = 40;               <= final non-static variable

        public static void main(String[] args) {

                int p = 50;             <= normal local variable
                final int q = 60;       <= final local variable

                //q = 70; CE: cannot assign a value to final variable q

                final int r;
                r = 70;

                //r = 80; CE: variable r might already have been assigned
        }
}
```

transient variables

The class level variable that has transient keyword in its definition is called transient variable.

Rule: local variable cannot be declared as transient. It leads to CE: illegal start expression

Find out CE in program if any?

```
class Example{
        static transient int x = 10;   ✓
        transient int y = 20;          ✓

        static void m1(){
                transient int z = 30;  ✗
        }
}
```

Note: We declare variable as transient to tell to JVM that we do not want to store variable value in a file in object serialization. Since local variable is not part object, declaring it as transient is illegal. For more details on object serialization and transient variable refer IOStreams material.

Volatile variable:

The class level variable that has volatile keyword in its definition is called volatile variable.

Rule: local variable cannot be declared as volatile. It leads to CE: illegal start expression

Find out CE in the below program if any?

```
class Example{
        static volatile int x = 10;   ✓
        volatile int y = 20;          ✓

        static void m1(){
                volatile int z = 30;  ✗
        }
}
```

Note: We declare variable as volatile to tell to JVM that we do not want to modify variable value concurrently by multiple threads. If we declare variable as volatile multiple thread are allowed to change its value in sequence one after one.

Since local variable is not directly accessible by thread, declaring it as volatile is illegal.
For more details on volatile and synchronized keywords refer Multithreading material.

Q) What are the modifiers allowed for a variables and methods?

Modifiers	private	Protected	public	static	final	abstract	native	volatile	transient	synchronized	strictfp
Local Variable	✗	✗	✗	✗	✓	✗	✗	✗	✗	✗	✗
Class Level Variable	✓	✓	✓	✓	✓	✗	✗	✓	✓	✗	✗
Method	✓	✓	✓	✓	✓	✓	✓	✗	✗	✓	✓
class	✗	✗	✓	✗	✓	✓	✗	✗	✗	✗	✓
interface	✗	✗	✓	✗	✗	✓	✗	✗	✗	✗	✓

Initial or default values of variables

- Default values are application only for class level variables.
- JVM initializes all types of class level variables with default values based its datatype even though it is assigned with explicit value.

What is the output from below program?

```
class Example{
        static int a;
        static boolean b;
        static String s;

        public static void main(String[] args){
                System.out.pritnln(a);  //0
                System.out.pritnln(b);  //false
                System.out.pritnln(s);  //null
        }
}
```

Life-time and scope of a variable

+ Lifetime is the time period between variable creation and destruction.
+ Scope is the region in which it can accessible.

Local variable life time and scope

- Local variable gets life only when its method is called and its variable creation statement is executed. It is destroyed automatically once method execution is completed.
- Its scope is only within its method after its creation statement.

How many variables are created in the below program?
Find out CE if any?

```
class Example{
        static void m1(int p){
                int q = 10;

                if (p == 10){
                        int r = 20;

                        System.out.println(p);
                        System.out.println(q);
                        System.out.println(r);
                }
                System.out.println(p);
                System.out.println(q);
                System.out.println(r);
        }
        public static void main(String[] args){
                Q) How many variables are created from the below method calls?
                m1(10);
                m1(20);
        }
}
```

Static variable life time and scope

- Static variable gets life when class is loaded. It is destroyed either if class is unloaded from JVM or if JVM is destroyed.

- Its scope is throughout class, and also in the place where class is accessible. It means where ever class is available there static variable is available provided it is public.

```
class Example{
        static int a = 10;
        static void m1(){
                System.out.pritnln(a); //10
        }
        public static void main(String[] args){
                System.out.pritnln(a);  //10
        }
}
```

Non-static variables life time and scope

- Non-Static variable gets life when object is created. It is destroyed when object is destroyed. object is destroyed when it is unreferenced or its referenced variable is destroyed.

- Its scope is the scope of object, object is available only if its referenced variable is available.

```
class Example{
        int x = 10;
        static void m1(){
                Example e1 = new Example();
                System.out.pritnln(e1.x);
        }
        public static void main(String[] args){
                Example e2 = new Example();
                System.out.pritnln(e2.x);

                System.out.pritnln(e1.x);
        }
}
```

In the above program you have one compiler time error, can you find it out?

You cannot access ***e1*** variable in main() method because it is local to m1() method. So main method ***e1.x*** statement leads to CE: cannot find symbol
symbol: variable e1

Q) How many members we can define inside a class ?
A) We can define **9** types of members in a single class

They are:

static members	non-static members
1. static variables	5. non-static variables
2. static blocks	6. non-static blocks
3. static methods	7. non-static methods
4. main method	8. constructors

Syntaxes

Static variable	Non-Static variable
Ex: static int a = 10;	*Ex:* int x = 10;

Static block
Ex:
static{
}
	Non-Static block
	Ex:
	{
	}

Static method
Ex:
static void m1(){
}
	Non-Static method
	Ex:
	void m1(){
	}

main method
Ex:
public static void main(Sting[] args){
}
	constructor
	Ex:
	Example(){
	}

Q) Why two types of members are given in OOPS?

- Static members are meant for storing data and operate that data common to all instances of an object
- Non-static members are meant for storing data and operate that data separately and specific to every instance of an object.

For more details on static and non-static members refer below two chapters
Chapter 12:- Static members and their control flow
Chapter 13:- Non-Static members and their control flow