

✚ We cannot **directly** use *non-static* variables inside *static* methods.

Reason – The non-static variables are object members. Each object will have their own copy of non-static variables. But, static methods are common for every object. So, during compilation, the compiler cannot understand – which object I am referring to.

Indirectly, we can use non-static variables inside static methods – by, passing the object reference variable as argument to the static method. Inside the static method we have to associate the non-static variables with the object reference variable using dot operator.

```
StaticMethod.java x
StaticMethod.java
1 // Checking if we can use non-static variables in static methods.
2
3 class Mobile {
4     String brand;
5     int price;
6     static void show(Mobile motorola) {
7         System.out.println("Inside show() method (static)");
8         System.out.println("Brand: " + motorola.brand + ", Price: " + motorola.price);
9     }
10 }
11
12 public class StaticMethod {
13     public static void main(String args[]) {
14         Mobile motorola = new Mobile();
15         motorola.brand = "Motorola";
16         motorola.price = 10000;
17
18         Mobile.show(motorola);
19     }
20 }
```

✚ Why **main** method is static?

Reason – If main method won't be static, then we will need to create an object of the class which contains the main method. But, we know that the main method is the entry point of execution. So, its not possible to create object when the execution itself has not started. That's why main

method is declared as static so that it can be executed without the need of any object.

What is **Encapsulation**?

Ans – Encapsulation is the process of binding data and operations(methods) together in a single unit(class). The purpose of encapsulation is to provide data privacy by preventing direct access to data from outside of the unit. It can be achieved by declaring the instance variables as private.

```
Encapse.java X
Encapse.java
1  class Info {
2      private String name;
3      private int age;
4
5      public void setName(String str) {
6          name = str;
7      }
8      public void setAge(int num) {
9          age = num;
10     }
11
12     public String getName() {
13         return name;
14     }
15     public int getAge() {
16         return age;
17     }
18 }
19
20 public class Encapse {
21     public static void main(String args[]) {
22         Info obj = new Info();
23
24         obj.setName("Suman Maji");
25         obj.setAge(21);
26
27         String name = obj.getName();
28         int age = obj.getAge();
29
30         System.out.println("Name: " + name);
31         System.out.println("Age: " + age);
32     }
33 }
```

“this” keyword –

When the names of instance variable and local variable are same, by default the preference goes to local variable. “this” keyword is used to specify the instance variable of the current object.

```
Encapse.java X
Encapse.java
1  class Info {
2      private String name;
3      private int age;
4
5      public void setName(String name) {
6          this.name = name;
7      }
8      public void setAge(int age) {
9          this.age = age;
10     }
11
12     public String getName() {
13         return name;
14     }
15     public int getAge() {
16         return age;
17     }
18 }
19
20 public class Encapse {
21     public static void main(String args[]) {
22         Info obj = new Info();
23
24         obj.setName("Suman Maji");
25         obj.setAge((int) (Math.random() * 100));
26
27         String name = obj.getName();
28         int age = obj.getAge();
29
30         System.out.println("Name: " + name);
31         System.out.println("Age: " + age);
32     }
33 }
```

+ Packages:

- Same package – No need to import
- Different package – Needs to import

+ Access Modifiers: Default, Public, Private, Protected

	Private	Protected	Public	Default
Same class	Yes	Yes	Yes	Yes
Same package subclass	NO	Yes	Yes	Yes
Same package non-subclass	NO	Yes	Yes	Yes
Different package subclass	NO	Yes	Yes	NO
Different package non-subclass	NO	NO	Yes	NO

○ When to use which one?

○ **Class – Public**

- You cannot make more than one class Public in a file.
You can have only one Public class in a file.
- The general idea is to have only one class in a file.

○ **Instance variables – Private**

○ **Methods – Public**

- If there is a method or a variable which should be accessed only in the subclasses in different packages, make it **Protected**.

- Try to avoid Default.


Polymorphism:

○ Compile-time: Method Overloading

- Happens within a single class
- **Same name with Different parameters**
- Return type can be same or different

○ Runtime: Method Overriding

- Happens between parent-child classes
- **Same name with Same parameter**
- Return type must be same or [covariant](#)

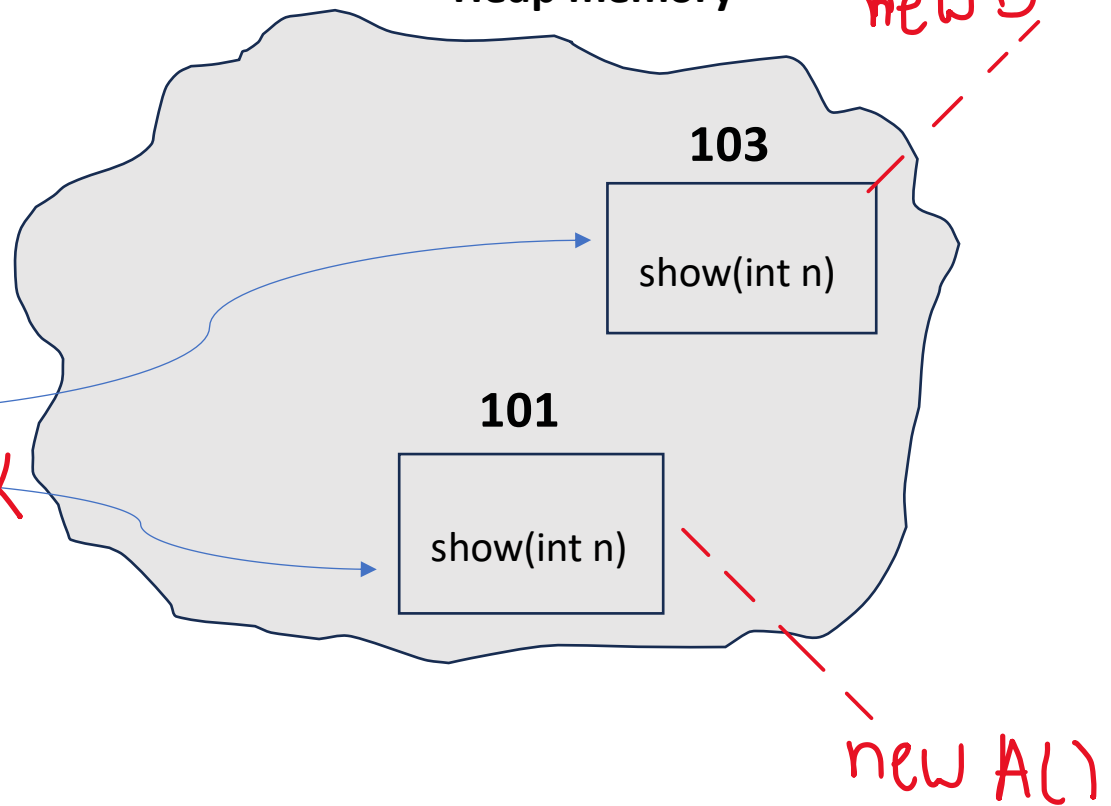
 Dynamic Method Dispatch: It is a mechanism in which, the reference variable of the parent class, refers to the, object of child class. During runtime, the overridden method in the child class gets called by the reference variable of parent class.

```
DMD.java
1  class A {
2      public void show(int n) {
3          System.out.println("In A show = " + ++n);
4      }
5  }
6
7  class B extends A {
8      public void show(int n) {
9          System.out.println("In B show = " + --n);
10     }
11 }
12
13 public class DMD {
14     public static void main(String args[]) {
15         A obj = new A();
16         obj.show(5); // ++n = ++5 = 6
17
18         obj = new B();
19         obj.show(5); // --n = --5 = 4
20     }
21 }
```

Stack memory

Obj	101 103

Heap memory



Time complexity of Array Insertion: $O(1)$

```
int arr[] = new int[5];
```

```
arr[3] = 99; → arr + (3 * 4)
```

Size of int = 4 bytes

= 100 + 12 = 112 {arr holds the starting address of the

array = 100}

Since, for insertion, only calculation is needed, no loop is needed, so time complexity is $O(1)$.

❖ **SAME FOR Array Fetch**

 **“final” keyword:** used for variable, method and class.

- **final variable** – constant, the assigned value cannot be changed.
 - **final class** – stops the inheritance, no class can extend it.
 - **final method** – the method cannot be overridden.
-

 **“abstract ” keyword:** used for methods and classes.

- **abstract method:** has only declaration, but no definition.
e.g – public void show();
 - **abstract class:** If I want to make a method abstract, its class also must be declared as **abstract**.
 - An abstract method needs to be in an abstract class.
 - An abstract class **does not** necessarily need any abstract method. It can have only regular methods also.
 - We cannot create object of abstract class. We have to create a subclass which will contain the definitions of the abstract methods. This class is known as concrete class. We can create object of this concrete class.
-

Wrapper class: In-built classes of java that represent primitive data types.


```
Wrapper.java X
Wrapper.java > Run | Debug main(String[])
1 // Wrapper classes are needed in scenarios when we want to work with primitives but also we have to use them as objects only. e.g: Collections
2
3 public class Wrapper {
4     public static void main(String args[]) {
5         A obj = new A();
6         obj.show1();
7         obj.show2();
8         obj.show3();
9     }
10 }
11
12 class A {
13     int num = 5; // primitive variable
14
15     // Integer num1 = new Integer(num); // → Boxing (storing primitive value into object) (deprecated since version 9)
16     Integer num1 = num; // → Auto-boxing
17     public void show1() {
18         System.out.println("Auto-boxing: " + num1);
19     }
20
21     // int num2 = num1.intValue(); // → Unboxing (extracting the primitive value from object)
22     int num2 = num1; // → Auto-unboxing
23     public void show2() {
24         System.out.println("Auto-unboxing: " + num2);
25     }
26
27     // Wrapper classes are also useful when performing arithmetic operations on string
28     String str = "15";
29     int num3 = Integer.parseInt(str);
30     public void show3() {
31         System.out.println("In show3(): " + num3*2);
32     }
33 }
```

Inner Class:

```
Inner.java X
Inner.java > Run | Debug Inner
1 // If there is a class that is used only to extend another class, it needs not to be used independently, then it can
  be treated as inner class.
2
3 public class Inner {
4     public static void main(String args[]) {
5         A obj = new A();
6         obj.show();
7
8         A.B obj1 = obj.new B(); ←
9         obj1.show1();
10    }
11 }
12
13 class A {
14     public void show() {
15         System.out.println(x:"In A show()");
16     }
17
18     class B {
19         public void show1() {
20             System.out.println(x:"In B show()");
21         }
22     }
23 }
```


Inner class can be static, Outer class cannot be static.

```
Inner.java X
Inner.java > A
1 // If there is a class that is used only to extend another class, it needs not to be used independently, then it can
  be treated as inner class.
2
3 public class Inner {
4     Run | Debug
5     public static void main(String args[]) {
6         A obj = new A();
7         obj.show();
8
9         A.B obj1 = new A.B(); ←
10        obj1.show1();
11    }
12 }
13 class A {
14     public void show() {
15         System.out.println(x:"In A show()");
16     }
17
18     static class B {
19         public void show1() {
20             System.out.println(x:"In B show()");
21         }
22     }
23 }
```

 **Anonymous inner class:** Inner class without a name. It is used when we know that the class will be used just once.

```
Anonymous.java X
Anonymous.java > A
1 public class Anonymous {
2     Run | Debug
3     public static void main(String[] args) {
4         A obj1 = new A();
5         obj1.show();
6
7         A obj2 = new A() { // → Anonymous inner class
8             public void show() {
9                 System.out.println(x:"In anonymous show()");
10            }
11        };
12        obj2.show();
13    }
14 }
15
16 class A {
17     public void show() {
18         System.out.println(x:"In A show()");
19     }
20 }
```

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL PORTS

```
PS D:\Programming\Development\Core Java - J2SE\Inner class\Anonymous inner class> java Anonymous
In A show()
In anonymous show()
PS D:\Programming\Development\Core Java - J2SE\Inner class\Anonymous inner class>
```

Enums: Named constants


```
Status.java X
Status.java > Status > main(String[])
1  enum PageStatus {
2      Pending, Running, Stopped, Crashed;
3  }
4
5  public class Status {
6      public static void main(String[] args) {
7          PageStatus stat = PageStatus.Running;
8          switch(stat) {
9              case Pending:
10             System.out.println(x:"Trying to connect ...");
11             break;
12             case Running:
13             System.out.println(x:"Server is running");
14             break;
15             case Stopped:
16             System.out.println(x:"Server is stopped");
17             break;
18             default:
19             System.out.println(x:"Server has crashed!");
20         }
21
22         PageStatus stat1[] = PageStatus.values(); // → returns the enum objects as an array
23
24         for(PageStatus s: stat1) {
25             System.out.println(s + " : " + s.ordinal()); // → returns the order numbers of the objects from enum array
26         }
27
28         System.out.println(stat.getClass()); // → returns the class name of the enum object
29         System.out.println(stat.getClass().getSuperclass()); // → return the super class (java.lang.Enum)
30     }
31 }
```

- Creating our own enum class:

```
OurEnum.java X
OurEnum.java > OurEnum > main(String[])
1  enum Laptop {
2      HP(price:58000), Dell(price:60000), Lenovo(price:50000), Asus(price:55000), Acer(price:40000), Macbook(price:0);
3
4      private int price;
5
6      // Generating constructors
7      private Laptop(int price) {
8          this.price = price;
9      }
10
11     // Generating getter
12     public int getPrice() {
13         return price;
14     }
15
16     //Generating setter
17     public void setPrice(int price) {
18         this.price = price;
19     }
20 }
21
22 public class OurEnum {
23     public static void main(String[] args) {
24         Laptop mac = Laptop.Macbook;
25         mac.setPrice(price:5);
26         for(Laptop lap: Laptop.values()) {
27             System.out.println(lap + " : " + lap.getPrice());
28         }
29     }
30 }
```

Types of Interface:

- Normal interface → More than one method.
 - Functional interface / SAM (Single Abstract Method) → Only one method.
 - Marker interface → Allows serialization and de-serialization.
-

 Lambda expression: It is used to shorten the code for implementing an interface to a class. It can be used only with **Functional interface**.

```
Lambda.java X
Lambda.java > Lambda > main(String[])
1 @FunctionalInterface // → Annotation
2 interface Design {
3     void show(int x, int y);
4 }
5
6 > // class A implements Design {...
11
12 public class Lambda {
13     public static void main(String[] args) {
14         Design obj = (x, y) → System.out.println("In lambda " + x + " " + y);
15     }
16     obj.show(x:5, y:8);
17 }
18 }
```



Exception Handling:

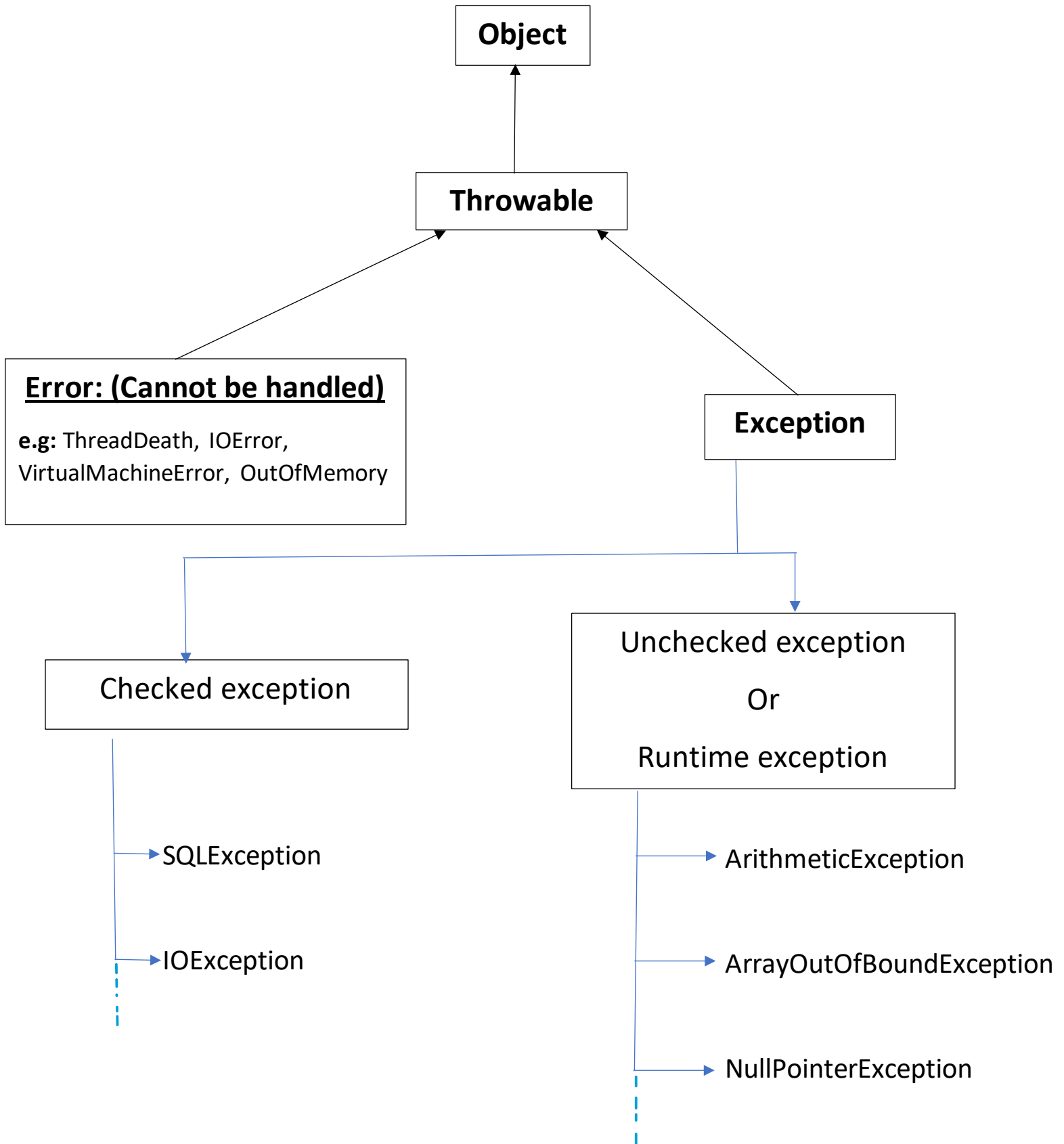
- Exception = Runtime error

```
Excep.java X
Excep.java > Excep > main(String[])
1 public class Excep {
2     public static void main(String[] args) {
3         int x = 0;
4         int y = 0;
5
6         try {
7             y = 5 / x;
8         } catch (Exception e) {
9             System.out.println(x: "Cannot divide by 0");
10        }
11
12        System.out.println(y);
13        System.out.println(x: "Hello");
14    }
15 }
```

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL PORTS

```
PS D:\Programming\Development\Core Java - J2SE\Exception handling> javac Excep.java
PS D:\Programming\Development\Core Java - J2SE\Exception handling> java Excep
Cannot divide by 0
0
Hello
```

Exception diagram:



Threads:

- **Priority range:** 1 – 10 (Default is 5)

➤ Creating threads by Implementing Runnable interface:

```
class A implements Runnable {
    public void run() {
        for (int i = 1; i <= 5; i++) {
            System.out.println("Hi");
            try {
                Thread.sleep(10);
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
        }
    }
}
```

```
class B implements Runnable {
    public void run() {
        for (int i = 1; i <= 5; i++) {
            System.out.println("Hello");
            try {
                Thread.sleep(10);
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
        }
    }
}
```

```

public class ThreadRun {
    public static void main(String[] args) {
        Runnable r1 = new A();
        Runnable r2 = new B();

        Thread t1 = new Thread(r1);
        Thread t2 = new Thread(r2);

        t1.start();
        t2.start();
    }
}

```

➤ Using Lambda expression:

```

public class ThreadRun {
    public static void main(String[] args) {
        Runnable r1 = () -> {
            for(int i=1; i<=5; i++) {
                System.out.println("Hi");
                try {
                    Thread.sleep(10);
                }
                catch(InterruptedException e) {
                    System.out.println(e);
                }
            }
        }
    }
}

```

```
};
```

```
Runnable r2 = () -> {  
    for(int i=1; i<=5; i++) {  
        System.out.println("Hello");  
        try {  
            Thread.sleep(10);  
        }  
        catch(InterruptedException e) {  
            System.out.println(e);  
        }  
    }  
};
```

```
Thread t1 = new Thread(r1);  
Thread t2 = new Thread(r2);
```

```
t1.start();  
t2.start();
```

```
}
```

```
}
```


Race conditions:

```
class Counter {
    int count;
    public synchronized void increment() { //
synchronized -> allows only one thread to call this
method at a time
        count++;
    }
}

public class ThreadRace {
    public static void main(String[] args) {
        Counter c = new Counter();

        Runnable r1 = () -> {
            for(int i=1; i<=10000; i++) {
                c.increment();
            }
        };

        Runnable r2 = () -> {
            for(int i=1; i<=10000; i++) {
                c.increment();
            }
        };

        Thread t1 = new Thread(r1);
        Thread t2 = new Thread(r2);

        t1.start();
        t2.start();
    }
}
```

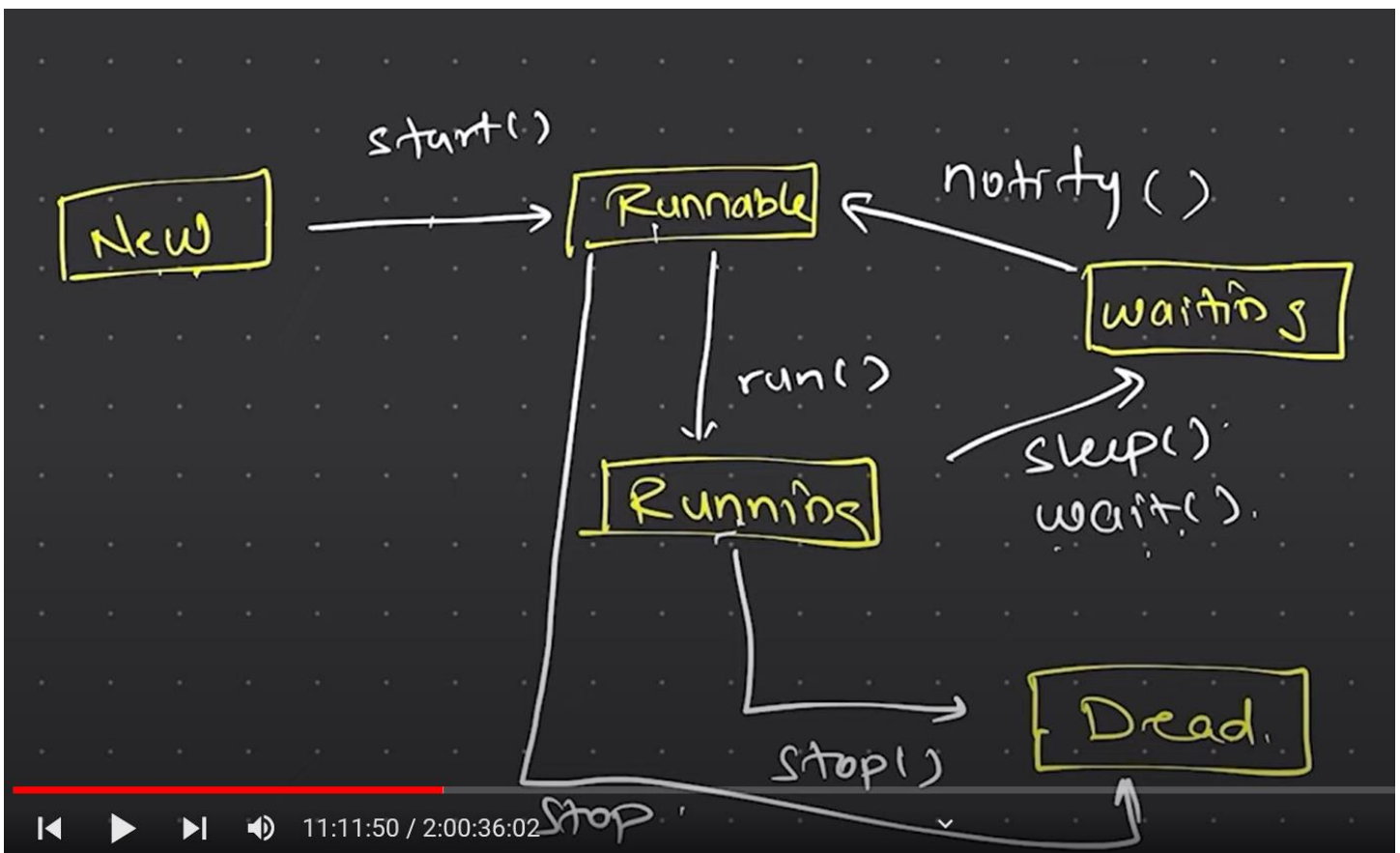
```

try {
    t1.join(); // join() -> tells main thread
to wait until t1 & t2 complete their task
    t2.join();
}
catch(InterruptedException e) {
    System.out.println(e);
}

System.out.println(c.count);
}
}

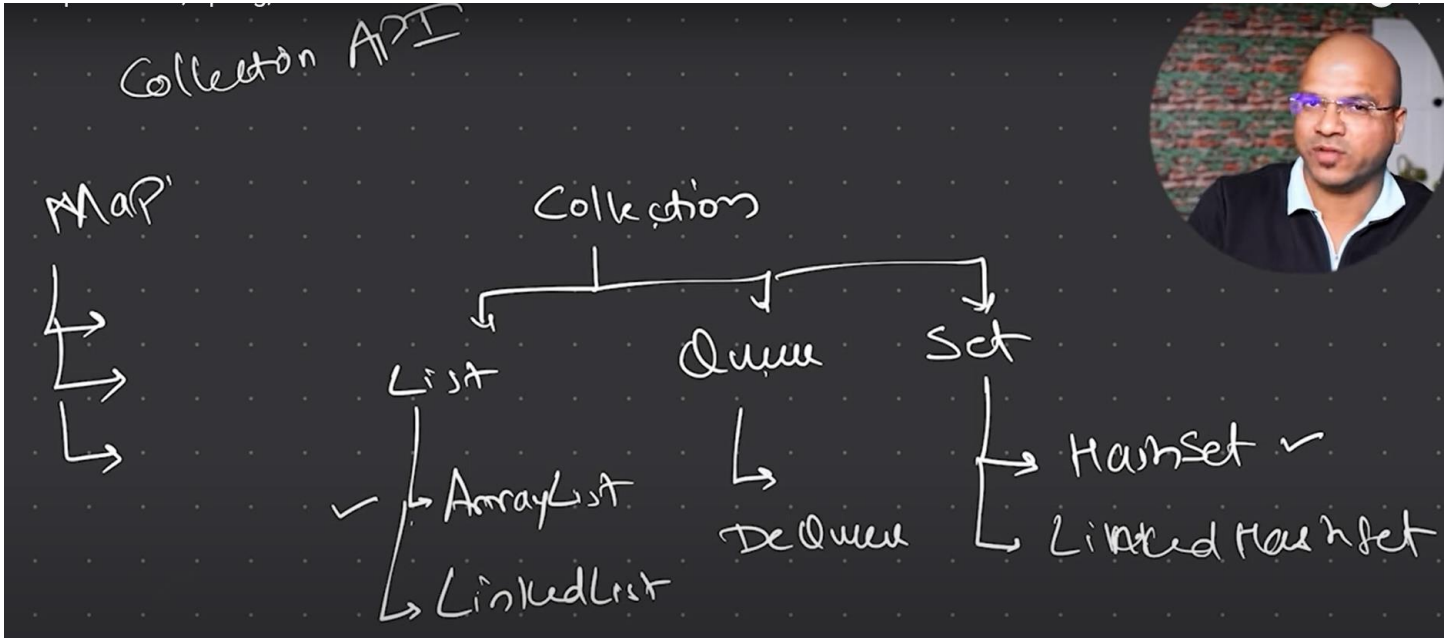
```

➤ Thread states:



Collection Framework:

- **Collection API** – it is a concept
- **Collection** – interface
- **Collections** – Class



➤ List:

```
import java.util.ArrayList;
import java.util.List;

public class Demo {
    public static void main(String[] args) {
        List<Integer> nums = new ArrayList<Integer>(); //
        // class ArrayList implements the interface List

        nums.add(5);
        nums.add(6);
        nums.add(100);
        nums.add(-8);
        nums.add(0);
```

```
        System.out.println(nums);  
    }  
}
```