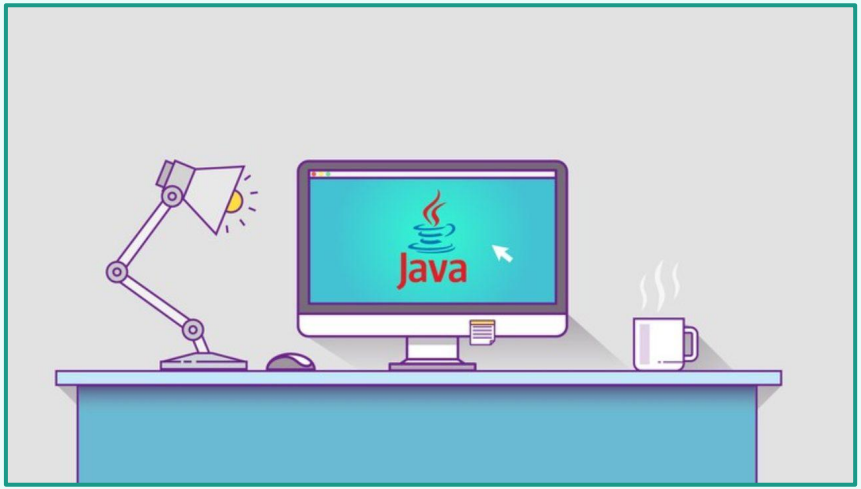
# Java SE & Spring

## Module 1: Java SE

# 15.Concurrency in Java



# What is Concurrency?

**Concurrency** is the ability to run several programs or several parts of a program in parallel. Concurrency enable a program to achieve high performance and throughput by utilizing the untapped capabilities of underlying operating system and machine hardware. e.g. modern computers has several CPU's or several cores within one CPU, program can utilize all cores for some part of processing; thus completing task much before in time in comparison to sequential processing.

The backbone of java concurrency are **threads**. A thread is a lightweight process which has its own call stack, but can access shared data of other threads in the same process. A Java application runs by default in one process. Within a Java application you can work with many threads to achieve parallel processing or concurrency.

---

Further reading: https://howtodoinjava.com/java-concurrency-tutorial/

# What is Concurrency?

Concurrent applications usually have more complex design in comparison to single threaded application. Code executed by multiple threads accessing shared data need special attention. Errors arising from incorrect thread synchronization are very hard to detect, reproduce and fix. They usually shows up in higher environments like production, and replicating the error is sometimes not possible in lower environments.

Apart from complex defects, concurrency requires more resources to run the application.

---

Further reading: https://howtodoinjava.com/java-concurrency-tutorial/

# Making a Java Application Concurrent

The very first class, you will need to make a java class concurrent, is **java.lang.Thread** class. This class is the basis of all concurrency concepts in java. Then you have **java.lang.Runnable** or **java.lang.Callable** interface to abstract the thread behavior out of thread class.

Other classes you will need to build advanced applications can be found at **java.util.concurrent** package added in Java 1.5.

# Multithreading in Java

The very first class, you will need to make a java class concurrent, is **java.lang.Thread** class. This class is the basis of all concurrency concepts in java. Then you have **java.lang.Runnable** or **java.lang.Callable** interface to abstract the thread behavior out of thread class.

Other classes you will need to build advanced applications can be found at **java.util.concurrent** package added in Java 1.5. Threads can be created by using mechanisms:

- extending Thread class
- implementing Runnable or Callable interfaces

# Extending Thread Class

We create a class that extends the java.lang.Thread class. This class overrides the run() method available in the Thread class. A thread begins its life inside run() method. We create an object of our new class and call start() method to start the execution of a thread. Start() invokes the run() method on the Thread object.

**Example**:

```
1  public class MyExtendedThread extends Thread {
2      @Override
3      public void run() {
4          try {
5              System.out.println("This is a thread, and has ID: "
6                      + Thread.currentThread().getId()
7                      + " and with priority: "
8                      + Thread.currentThread().getPriority());
9          } catch (Exception e) {
10             System.out.println("An error occurred while running thread: " +
11 e.getMessage());
12         }
13     }
14 }
```

# Lifecycle and States of a Thread

A thread in Java at any point of time exists in any one of the following states. A thread lies only in one of the shown states at any instant:

- **New** : When a new thread is created, it is in the new state.
- **Runnable** : A thread that is ready to run is moved to runnable state.
- **Blocked** : When a thread is waiting for a task to complete, it lies in the blocked state.
- **Waiting** : A thread is in the waiting state when it waits for another thread on a condition.
- **Timed Waiting** : A thread lies in this state until the timeout is completed or until a notification is received.
- **Terminated** : A thread terminates because of either exists normally or some unusual erroneous event, like segmentation fault or an unhandled exception.

# Implementing Runnable Interface

We create a new class which implements java.lang.Runnable interface and override run() method. Then we instantiate a Thread object and call start() method on this object.

**Example** :

```
1  public class CounterRunnable  implements Runnable{
2      @Override
3      public void run() {
4          long sum = 0;
5          for (long i = 0; i < 2000; i++) {
6              sum+= i;
7              try {
8                  Thread.sleep(1);
9              } catch (InterruptedException e) {
10                 e.printStackTrace();
11             }
12         }
13         System.out.println("(Calculated in child thread) Sum of numbers between 0
14 and 2000 is: " + sum);
15     }
16 }
```

# Thread Class vs Runnable Interface

- If we extend the Thread class, our class cannot extend any other class because Java doesn't support multiple inheritance. But, if we implement the Runnable interface, our class can still extend other base classes.
- We can achieve basic functionality of a thread by extending Thread class because it provides some inbuilt methods like yield(), interrupt() etc. that are not available in Runnable interface.
- Using runnable will give you an object that can be shared amongst multiple threads.

# Implementing Callable Interface

Sometimes we wish that a thread could return some value that we can use. Java 5 introduced java.util.concurrent.Callable interface in concurrency package that is similar to Runnable interface but it can return any Object and able to throw Exception.

**Example** :

```
1  public class FactorialCallable implements Callable<Long> {
2
3      @Override
4      public Long call(){
5          System.out.println("This is call method of call method");
6      }
7  }
```

# Implementing Callable Interface

Here we sent a **Callable** object to be executed in an executor using the **submit()** method. This method receives a **Callable** object as a parameter and returns a **Future** object that we can use with two main objectives

- We can control the status of the task – we can cancel the task and check if it has finished. For this purpose, we have used the **isDone()** method to check if the tasks had finished.
- We can get the result returned by the **call()** method. For this purpose, we have used the **get()** method. This method waits until the **Callable** object has finished the execution of the **call()** method and has returned its result.

# Runnable vs Callable Interface

- For implementing Runnable, the run() method needs to be implemented which does not return anything, while for a Callable, the call() method needs to be implemented which returns a result on completion. Note that a thread can’t be created with a Callable, it can only be created with a Runnable.
- Another difference is that the call() method can throw an exception whereas run() cannot.

# Questions ?

# Next:Java Database Connectivity(JDBC)