1. Java *InterruptedException*

1.1. What are Interrupts?

In concurrency, an interrupt is a signal to *Thread* to stop itself and figure out what to do next. Generally, it asks the *Thread* to terminate itself gracefully.

The *InterruptedException* is thrown when a thread is waiting, or sleeping, or otherwise occupied, and another thread interrupts it by using the *interrupt()* method present in the Thread class.

Internally, the interrupt mechanism is implemented using an internal flag known as interrupt status. When a *Thread* invokes `interrupt()` method then this flag is set, and we can check the status of this flag by using `isInterrupted()` method. Another method present in Thread class, `Thread.interrupted()` can clear out this flag again to respond to other interrupt requests.

Note that *Thread* class contains the following methods used for creating and checking interrupts:

- *`void interrupt()`*: interrupts a *Thread*
- *`boolean isInterrupted()`*: checks whether the current *Thread* has been interrupted or not. The interrupted status of the *Thread* is unaffected by this method.

- `boolean interrupted()`: checks whether the current *Thread* has been interrupted. The interrupted status of the *Thread* is cleared by this method.

1.2. *Thread.interrupt()* Method

The *interrupt()* method interrupts the thread on which it is invoked.

- If this thread is blocked in an invocation of the *sleep()*, *wait()* or *join()* methods then its interrupt status will be cleared, and it will receive an *InterruptedException*.
- If this thread is blocked in an I/O operation upon an *InterruptibleChannel* then the channel will be closed, the thread's interrupt status will be set, and the thread will receive a *ClosedByInterruptException*.
- If this thread is blocked in a *java.nio.channels.Selector* then the thread's interrupt status will be set, and it will return immediately from the selection operation.
- If none of the previous conditions hold then this thread's interrupt status will be set, and we can check it using `isInterrupted()` or `interrupted()` methods.

2. How to Interrupt a *Thread*

A *Thread* can interrupt another waiting or sleeping *Thread* by using the `interrupt()` method of *Thread* class.

In the following example, *TestThread* works periodically after every 2 seconds. It checks if it has been interrupted or not. If not, it continues working, finishes the processing and returns. If it is interrupted in between by another *Thread*, it can terminate gracefully by throwing *InterruptedException* to the caller *Thread*.

```
public class TestThread extends Thread {

 public void run() {
     try{

          while(true) {
                                  // Check if it is interrupted, if so then throw InterruptedException
               if(Thread.interrupted()) {
                    throw new InterruptedException();
               }
                                  // else continue working
               else {
                    System.out.println("Continue working");
               }
```

```
                                Thread.sleep(2000L);
          }
     } catch (InterruptedException e) {
                         // Handling InterruptedException and Graceful shutdown of the Thread
          System.out.println("Graceful shutdown");
     }
 }
}
```

Another *Thread* (Eg: main *Thread*) that started it, sometimes later decides that the task performed by *TestThread* is not necessary anymore, so it interrupts it.

```
// TestThread Instantiation
TestThread t1 = new TestThread();

// Starting TestThread
t1.start();

// main Thread enters into sleeping state
Thread.sleep(5000);

// main Thread decided that TestThread is no longer needed, so interrupting it

t1.interrupt();
```

When we execute this program, we will get the following output,

Output

```
Continue working

Continue working

Continue working


Graceful Termination
```

3. Handling *InterruptedException*

When working in a multithreaded application, it is important to handle *InterruptedException* gracefully, and the threads should respond to interrupts promptly to avoid the application from entering into a *deadlock* situation.

- As *InterruptedException* is a *checked exception* so we have to handle it either by using try-catch or *throws* keyword.
- We can propagate it up in the stack to the caller method by using *throws* keyword. In this case, the caller method needs to handle this exception.
- There are some scenarios where throwing an exception is not possible like in case of *Runnable* interface `run()` method, it won't allow throwing an exception, so for these cases we can use *try-catch* block

and handle this exception explicitly in the calling Thread itself instead of throwing it.

Let’s consider the first case when an Interrupted *Thread* throws *InterruptedException* and ask the caller method to handle it instead of handling it by itself.

```
// method throwing InterruptedException so that it is propagated to the caller method
public static void throwInterruptedException() throws InterruptedException
{
    // Thread enters into sleeping state
    Thread.sleep(1000);

    // Thread interrupting itself
    Thread.currentThread().interrupt();

    // check if the Thread is interrupted
    if (Thread.interrupted()) {
        // Throwing InterruptedException
        throw new InterruptedException();
    }
}

// main method
public static void main(String[] args)
```

```
{
    try{
             // calling a method that throws InterruptedException
             throwInterruptedException();
         } catch(InterruptedException e){
             System.out.println("Thread interrupted by throwing the exception")
         }
}
```

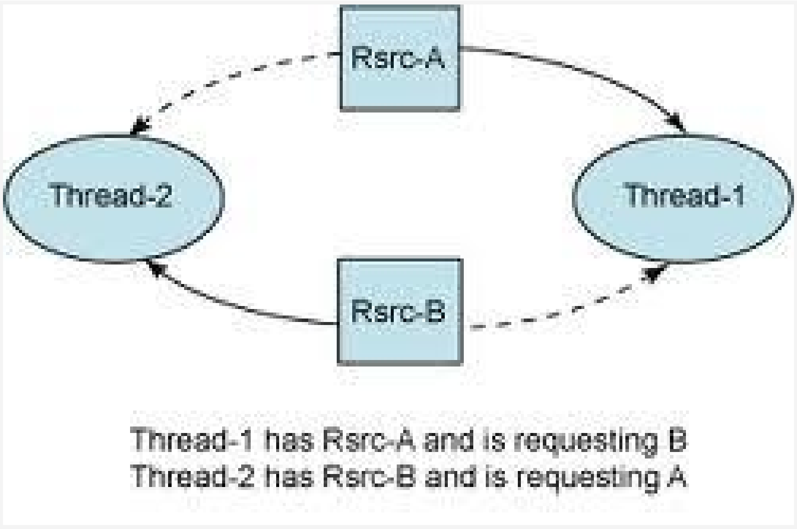
Let’s now consider the case where an interrupted *Thread* itself handling the *InterruptedException* using *try-catch* block instead of propagating it up to the caller method using *throws* keyword.

```
// method handling InterruptedException using try-catch block
public void run()
{
     try{
             // Thread enters into sleeping state
             Thread.sleep(1000);
             // Thread interrupting itself
             Thread.currentThread().interrupt();
          } catch(InterruptedException e){
               System.out.println("Thread interrupted and handling exception by itself")
          }
}
```

```
// main method
public static void main(String[] args)
{
    // Creating new Thread
    Thread thread = new Thread();
    // Starting a Thread
     thread.start();
}
```

4. Simulating a Deadlock

In Java, a deadlock is a situation where a minimum of two threads are holding the lock on some different resource, and both are waiting for the other's resource to complete its task. And, none is able to leave the lock on the resource it is holding.


Rsrc-A
Thread-2
Thread-1
Rsrc-B
Thread-1 has Rsrc-A and is requesting B
Thread-2 has Rsrc-B and is requesting A

Deadlock Scenario

In the above case, Thread-1 has A but need B to complete processing and similarly Thread-2 has resource B but need A first.

```
public class ResolveDeadLockTest {

  public static void main(String[] args) {
    ResolveDeadLockTest test = new ResolveDeadLockTest();

    final A a = test.new A();
    final B b = test.new B();

    // Thread-1
    Runnable block1 = new Runnable() {
      public void run() {
        synchronized (a) {
          try {
            // Adding delay so that both threads can start trying
 to
            // lock resources
            Thread.sleep(100);
          } catch (InterruptedException e) {
            e.printStackTrace();
          }
          // Thread-1 have A but need B also
          synchronized (b) {
            System.out.println("In block 1");
          }
        }
      }
    };

    // Thread-2
    Runnable block2 = new Runnable() {
      public void run() {
        synchronized (b) {
          // Thread-2 have B but need A also
          synchronized (a) {
            System.out.println("In block 2");
          }
        }
      }
    };

    new Thread(block1).start();
    new Thread(block2).start();
  }

  // Resource A
  private class A {
    private int i = 10;

    public int getI() {
      return i;
    }

    public void setI(int i) {
      this.i = i;
    }
  }

  // Resource B
  private class B {
    private int i = 20;

    public int getI() {
      return i;
    }

    public void setI(int i) {
      this.i = i;
    }
  }
}
```

Running the above code will result in a deadlock for very obvious reasons

(explained above). Now we have to solve this issue.

4.1 How to Solve a Deadlock?

I believe that the solution to any problem lies in identifying the root of the problem. In our case, it is the pattern of accessing the resources A and B, is main issue. So, to solve it, we will simply re-order the statements where the code is accessing shared resources.

```
// Thread-1
Runnable block1 = new Runnable() {
  public void run() {
    synchronized (b) {
      try {
        // Adding delay so that both threads can start trying to
        // lock resources
        Thread.sleep(100);
      } catch (InterruptedException e) {
        e.printStackTrace();
      }
      // Thread-1 have A but need B also
      synchronized (a) {
        System.out.println("In block 1");
      }
    }
  }
};

// Thread-2
Runnable block2 = new Runnable() {
  public void run() {
    synchronized (b) {
      // Thread-2 have B but need A also
      synchronized (a) {
        System.out.println("In block 2");
      }
    }
  }
};
```

Run again above class, and you will not see any deadlock kind of situation. I hope it will help you avoid deadlocks and, if encountered, resolve them.

5. Conclusion

This article taught us about *InterruptedException*, its constructors, methods and causes. We have also seen various ways by which we can handle *InterruptedException* and Deadlocks and how to resolve them in our code