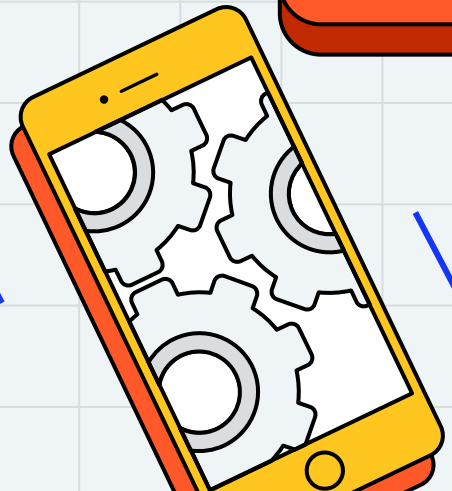
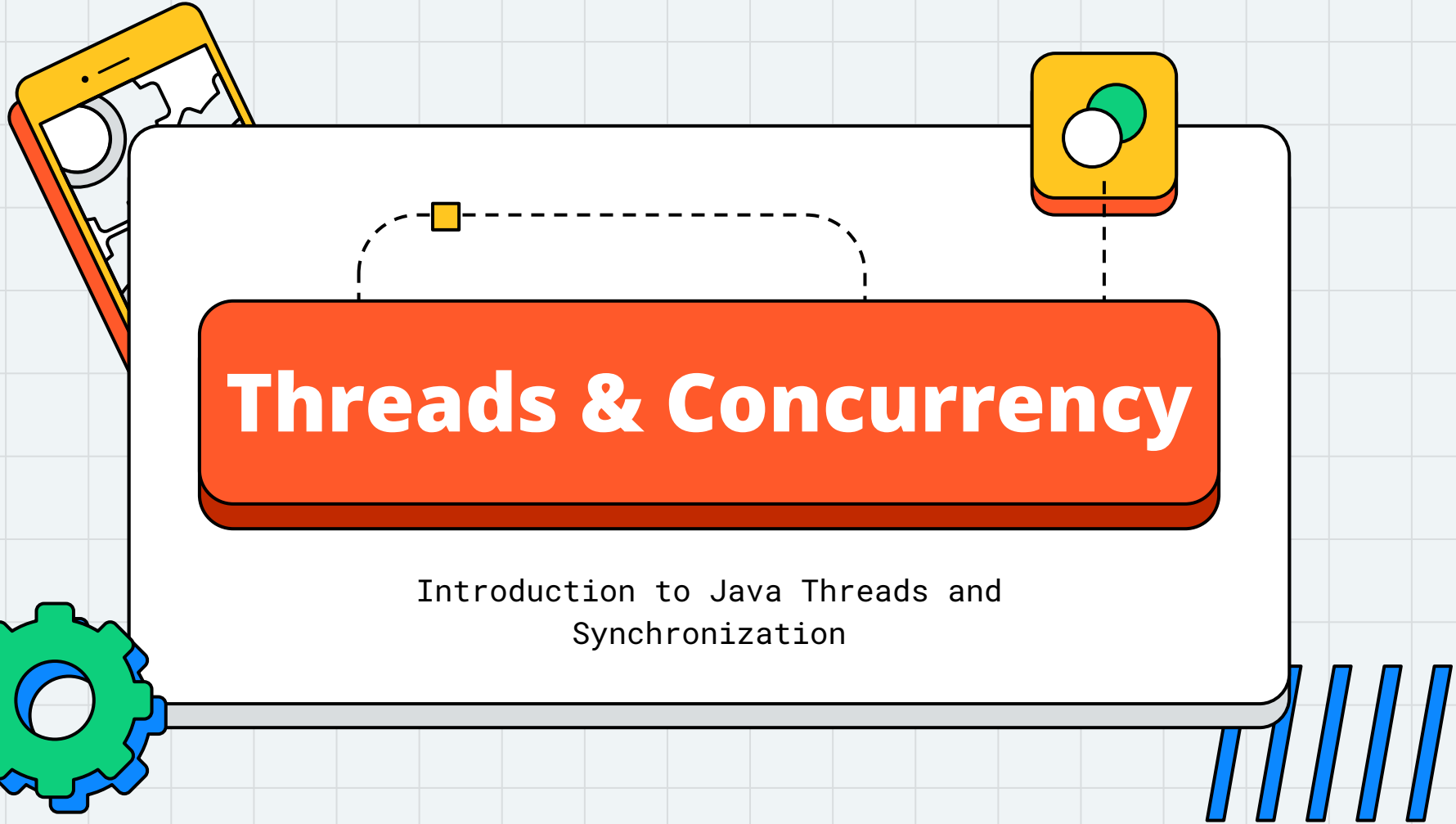
Application Programming



Hend Alkittawi

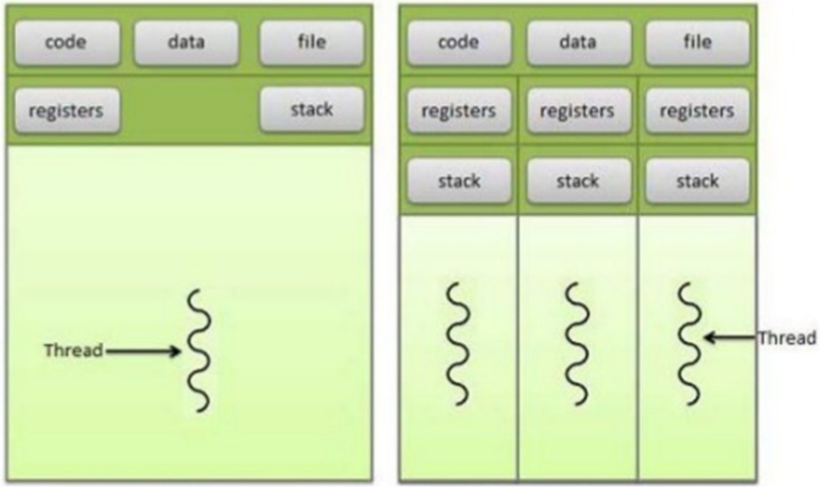

Threads & Concurrency
Introduction to Java Threads and Synchronization

INTRODUCTION

- All modern operating systems support concurrency, via **processes** and **threads**.
- A **process** is an instance of a program running in a computer
  - example: if you start a java program, the OS spawns a new process, which runs in parallel to other programs.
- A **thread** is a program unit that is executed concurrently with other parts of the program.
- One or more threads run in the context of the process.
- **Multiple threads** can collaborate and work efficiently within a single program.

THREADS

- Multi-threaded applications have **multiple threads within a single process**
  - each thread have its own program counter, stack and set of registers,
  - All threads share common code, data, and certain structures such as open files.

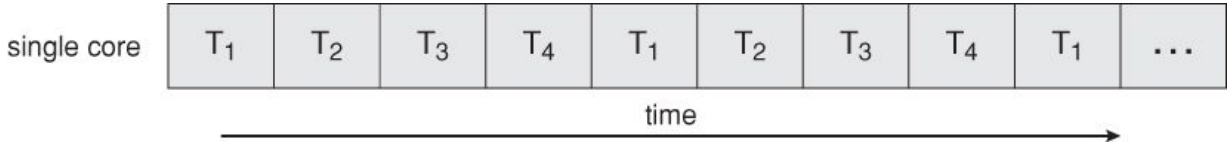


THREADS

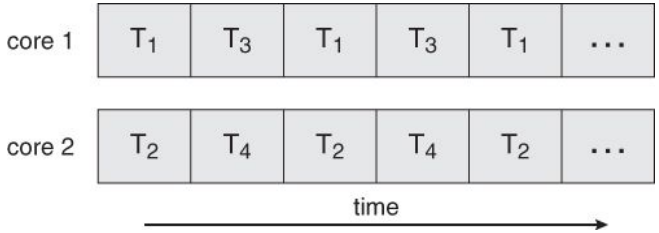
- Threads are very useful in modern programming whenever a process has multiple tasks to perform independently of the others.
- This is particularly true when one of the tasks may block, and it is desired to allow the other tasks to proceed without blocking.
  - For example in a word processor, a **background thread** may check spelling and grammar while a **foreground thread** processes user input (keystrokes), while yet a **third thread** loads images from the hard drive, and a **fourth** does periodic automatic backups of the file being edited.

THREADS

- A multi-threaded application running on a single-core chip would have to interleave the threads



- A multi-threaded application running on a multi-core chip, the threads could be spread across the available cores, allowing true parallel processing



THREADS

- The **thread scheduler** gives no guarantee about the order in which threads are executed.
- Each thread runs for a short amount of time, called a **time slice**. Then the schedule activates another thread. However, there will always be slight variations in running times. Thus, you should expect that **the order in which each thread gains controls is somewhat random**.
- It is important to observe that the order and the timing of operations performed by the threads are controlled by the runtime system, and **cannot be controlled by the programmer**.

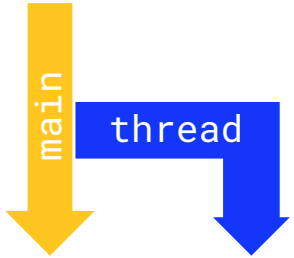
JAVA THREADS

- The JVM executes each thread for a short amount of time and then switches to another thread.
- In a multithreaded environment, threads can be: **created**, **scheduled** to run, **paused**, **resumed**, and **terminated**.
- In Java, we can create threads within that process two different ways
  - Create a new class of type **Thread**
    - java.lang.Thread
  - Create a new class that implements the **Runnable interface**
    - java.lang.Runnable
    - the Runnable interface has a single method called **run()**.

JAVA THREADS

- To create threads by creating a new class of type **Thread**
  1. Create a class that **extends the Thread** class.
  2. Override the **run()** method by placing the task code **into the run() method** of your class.
  3. Create an object of the subclass
  4. Call the start method to start the thread

```java
public class MyThread extends Thread {
    @Override
    public void run() {
        // your code here!
    }
    public static void main( String[] args ){
        MyThread thread = new MyThread();
        thread.start();
    }
}
```

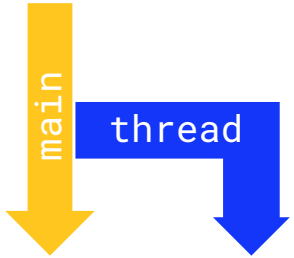


JAVA THREADS

- To create threads by creating a new class that implements the **Runnable interface:**

1. Create a class that **implements the Runnable** interface.
2. Place the task code into **the run() method** of your class.
3. Create an object of the subclass
4. Construct a thread object from the Runnable object.
5. Call the start method to start the thread

```
public class MyRunnable implements Runnable {
    public void run() {
        // your code here!
    }
    public static void main( String[] args ){
        Runnable runnable = new MyRunnable();
        Thread thread = new Thread(runnable);
        thread.start();
    }
}
```



JAVA THREADS

- Main Thread

```
public class MainThreadDemo {
      public static void main(String[] args) {
            Thread t = Thread.currentThread();
            System.out.println("Current thread: " + t);
            t.setName("My Thread"); // set the thread name
            System.out.println("After name change: " + t);
            try {
                  for(int i = 5; i > 0; i--) {
                        System.out.println(i);
                        Thread.sleep(500);
                  }
            }
            catch(InterruptedException e) {
            }
      }
}
```

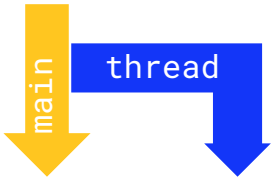
The **sleep()** method puts the current thread to sleep for a given number of milliseconds

```
Current thread: Thread[main,5,main]
After name change: Thread[My Thread,5,main]
5
4
3
2
1
```

Creating a thread by extending the Thread class

```
public class MyThread extends Thread{
    public MyThread() {
      super("Demo Thread");
      System.out.println("Child Thread: " + this);
    }
    // This is the entry point for the second thread.
    @Override
    public void run() {
      try {
         for(int i = 5; i > 0; i--) {
           System.out.printf("%-15s: %d\n", "Child Thread", i);
           Thread.sleep(500);
         }
      }
      catch(InterruptedException e) {
      }
      System.out.println("Exiting Child Thread ...");
    }
}
```

```
public class MyThreadDemo {
    public static void main(String[] args) {
        MyThread nt = new MyThread();
         nt.start();
         try {
             for(int i = 5; i > 0; i--) {
                  System.out.printf("%-15s: %d\n", "Main Thread", i);
                  Thread.sleep(1000);
              }
         }
         catch(InterruptedException e) {
         }
         System.out.println("Exiting Main Thread ...");
    }
}
```



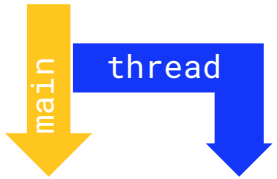

```
Child Thread: Thread[Demo Thread,5,main]
Main Thread: 5
Child Thread: 5
Child Thread: 4
Main Thread: 4
Child Thread: 3
Child Thread: 2
Main Thread: 3
Child Thread: 1
Exiting Child Thread ...
Main Thread: 2
Main Thread: 1
Exiting Main Thread ...
```

Creating a thread by implementing the Runnable interface

```java
public class MyRunnable implements Runnable{

    // This is the entry point for the second thread.
    @Override
    public void run() {
       try {
           for(int i = 5; i > 0; i--) {
              System.out.printf("%-15s: %d\n", "Child Thread", i);
              Thread.sleep(500);
           }
       }
       catch(InterruptedException e) {
       }
       System.out.println("Exiting Child Thread ...");
    }
}
```

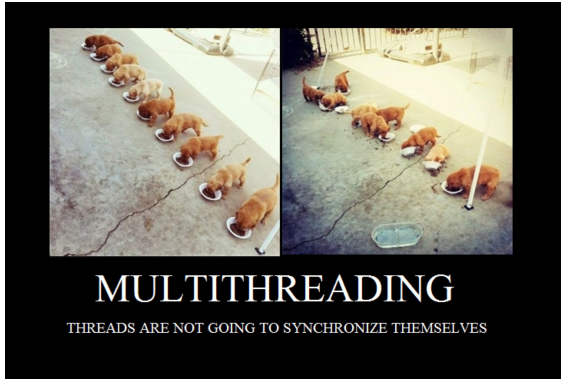
```java
public class MyRunnableDemo {
    public static void main(String[] args) {
       MyRunnable runnable = new MyRunnable();
       Thread thread = new Thread(runnable);
       thread.start();
       try {
           for(int i = 5; i > 0; i--) {
              System.out.printf("%-15s: %d\n", "Main Thread", i);
              Thread.sleep(1000);
           }
       }
       catch(InterruptedException e) {
       }
       System.out.println("Exiting Main Thread ...");
    }
}
```




```
Main Thread    : 5
Child Thread   : 5
Child Thread   : 4
Main Thread    : 4
Child Thread   : 3
Child Thread   : 2
Main Thread    : 3
Child Thread   : 1
Exiting Child Thread ...
Main Thread    : 2
Main Thread    : 1
Exiting Main Thread ...
```

THREAD SYNCHRONIZATION

- When threads share access to a common object, they can conflict with each other. The shared access creates a problem. This problem is often called a **race condition**.
- To solve the problem use a lock mechanism. The lock mechanism is used to control the threads that want to manipulate a shared object.



THREAD SYNCHRONIZATION

- To acquire the lock the code calls a **synchronized method**.
- Methods that contain threed sensitive code are tagged with the **synchronized** keyword.
- When a thread calls a synchronized method on a shared object, it owns that object's lock until it returns from the method and thereby unlocks the object.
- When an object is locked by one thread, **no other thread can enter a synchronized method** for that object, the other thread is automatically deactivated, and it needs to wait until the first thread has unlocked the object.

THREAD SYNCHRONIZATION

- When multiple threads need to update information stored in a shared object, some ordering has to be enforced to avoid unintended consequences.
- Java provides a locking mechanism for this purpose!
- When one thread wants to access the shared object, it has to
  - Lock the object
  - Complete its operations on the object
  - Unlock the object
- This way, each thread will have exclusive access to the object when the thread needs the object

Thread Synchronization - No Threads!

```
public class CallMe {
    public void call(String message) {
     System.out.print("[ ");
     System.out.print(message);
     System.out.print(" ]");
    }
}
```

```
public class SyncDemo {
    public static void main(String[] args) {
     CallMe target = new CallMe();
     target.call("No threads!");
    }
}
```

```
[ No threads! ]
```

Thread Synchronization - No Synchronization

```java
public class CallMe {
    public void call(String message) {
       System.out.print("[ ");
       System.out.print(message);
       System.out.print(" ]");
    }
}
```

```java
public class Caller implements Runnable {
    String msg;
    CallMe target;

    public Caller(CallMe targ, String msg) {
       this.target = target;
       this.msg = msg;
    }

    @Override
    public void run() {
       target.call(msg);
    }
}
```

```java
public class SyncDemo {
    public static void main(String[] args) {
       CallMe target = new CallMe();

       Runnable callerA = new Caller(target, "Hello");
       Runnable callerB = new Caller(target, "World");
       Runnable callerC = new Caller(target, "Howdy Y'all");

       Thread threadA = new Thread(callerA);
       Thread threadB = new Thread(callerB);
       Thread threadC = new Thread(callerC);

       threadA.start();
       threadB.start();
       threadC.start();
       try {
           threadA.join();
           threadB.join();
           threadC.join();
       } catch (InterruptedException e) {
       }
    }
}
```

The **join()** method allows one thread to wait until another thread completes its execution.

```
[ Hello[ Howdy Y'all ][ World ] ]
```

```
[ Hello ][ Howdy Y'all ][ World ]
```

```
[ Hello[ World ][ Howdy Y'all ] ]
```

Thread Synchronization - Method Synchronization

```java
public class CallMe {
    public synchronized void call(String message) {
       System.out.print("[ ");
       System.out.print(message);
       System.out.print(" ]");
    }
}
```

```java
public class Caller implements Runnable {
    String msg;
    CallMe target;

    public Caller(CallMe targ, String msg) {
       this.target = target;
       this.msg = msg;
    }

    @Override
    public void run() {
       target.call(msg);
    }
}
```

```java
public class SyncDemo {
    public static void main(String[] args) {
       CallMe target = new CallMe();

       Runnable callerA = new Caller(target, "Hello");
       Runnable callerB = new Caller(target, "World");
       Runnable callerC = new Caller(target, "Howdy Y'all");

       Thread threadA = new Thread(callerA);
       Thread threadB = new Thread(callerB);
       Thread threadC = new Thread(callerC);

       threadA.start();
       threadB.start();
       threadC.start();
       try {
           threadA.join();
           threadB.join();
           threadC.join();
       } catch (InterruptedException e) {
       }
    }
}
```

```
[ Hello ][ Howdy Y'all ][ World ]
```

```
[ Hello ][ World ][ Howdy Y'all ]
```

Thread Synchronization - Block Synchronization

```java
public class CallMe {
    public void call(String message) {
       System.out.print("[ ");
       System.out.print(message);
       System.out.print(" ]");
    }
}
```

```java
public class Caller implements Runnable {
    String msg;
    CallMe target;

    public Caller(CallMe targ, String msg) {
       this.target = target;
       this.msg = msg;
    }

    @Override
    public void run() {
       synchronized(target){
           target.call(msg);
       }
    }
}
```

```java
public class SyncDemo {
    public static void main(String[] args) {
       CallMe target = new CallMe();

       Runnable callerA = new Caller(target, "Hello");
       Runnable callerB = new Caller(target, "World");
       Runnable callerC = new Caller(target, "Howdy Y'all");

       Thread threadA = new Thread(callerA);
       Thread threadB = new Thread(callerB);
       Thread threadC = new Thread(callerC);

       threadA.start();
       threadB.start();
       threadC.start();
       try {
           threadA.join();
           threadB.join();
           threadC.join();
       } catch (InterruptedException e) {
       }
    }
}
```

```
[ Hello ][ Howdy Y'all ][ World ]
```

```
[ Hello ][ World ][ Howdy Y'all ]
```

THANK YOU!

DO YOU HAVE ANY QUESTIONS?

hend.alkittawi@utsa.edu

By Appointment

Online