1495 UNIVERSITY OF ABERDEEN

JC2002 Java Programming

Lecture 27: Basics of concurrency

References and learning objectives

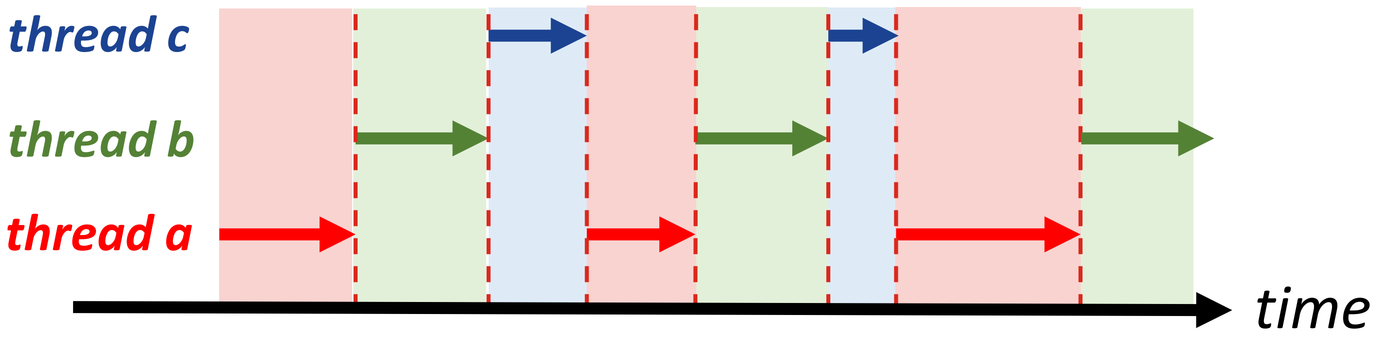
- Today's sessions are mostly based on:
  - Deitel, H., ***Java How to Program, Early Objects***, Chapter 23, 2018
  - https://docs.oracle.com/javase/tutorial/uiswing
- After today's session, you should be able to:
  - Explain the concepts of concurrency and multithreading
  - Define and use threads in Java using `Thread` superclass
  - Implement multithreading in Swing applications using Swing API
  - Implement the basic techniques to avoid thread interference and deadlocks in your multithreading applications

Concurrent programming

- In concurrent programming, blocks of program code (e.g., methods) are executed *concurrently* during overlapping time periods
- There are two basic units of execution in concurrent programming:
  - ***Processes***: each process has a self-contained execution environment (complete, private set of run-time resources, i.e., its own memory space)
  - ***Threads***: each thread exists within a process (every process has at least one thread) and therefore threads share the process's resources, including memory and open files
  - In Java programming, we are mostly concerned with threads

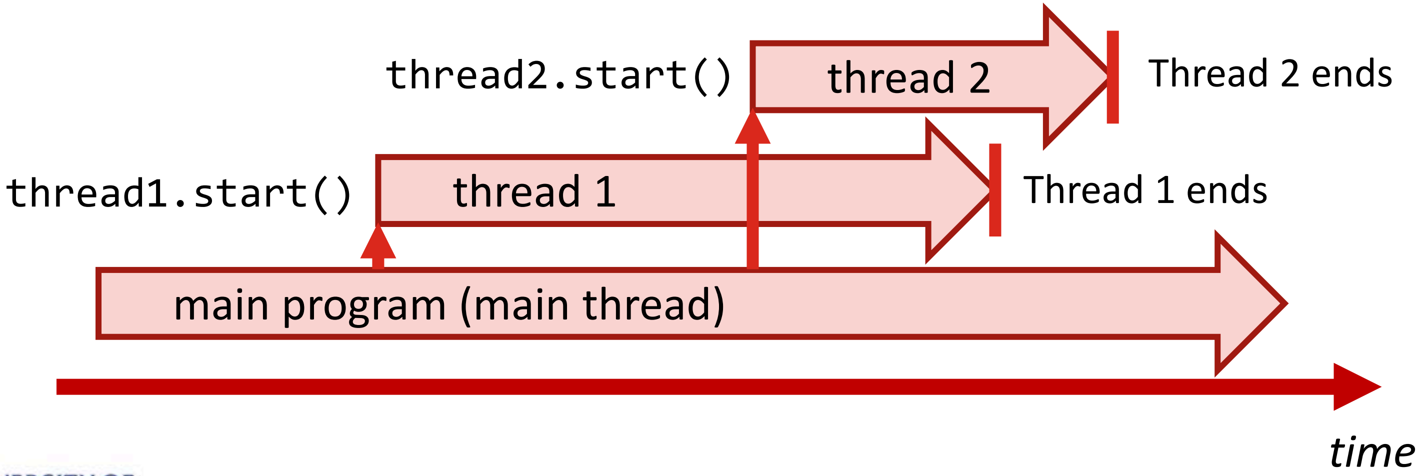
Context switching

- Typically, multithreading is implemented in operating systems by using *context switching*
  - Threads are run using short time slots in round robin fashion (each thread gets its turn alternatingly), creating illusion of CPU multitasking



Multithreading in Java

- In Java, threads can be used by extending class **`Thread`**
  - The code to be executed is implemented in overridden method **`run()`**
  - The thread is started using its method **`start()`**



Simple multithreading example (1)

```java
public class TestThreads {
  static void printList(int n) {
    for(int i=1; i<=5; i++) {
      System.out.print(i*n + " ");
    }
    System.out.println();
  }
  public static void main(String args[]){
    Thread thread1 = new Thread() {
      public void run() {
        TestThreads.printList(1);
 }
    };
    Thread thread2 = new Thread() {
      public void run() {
        TestThreads.printList(10);
      }
    };
    thread1.start();
    thread2.start();
  }
}
```

Simple multithreading example (2)

```java
public class TestThreads {
 static void printList(int n) {
  for(int i=1; i<=5; i++) {
   System.out.print(i*n + " ");
  }
  System.out.println();
 }
 public static void main(String args[]){
  Thread thread1 = new Thread() {
   public void run() {
    TestThreads.printList(1);
   }
  };
  Thread thread2 = new Thread() {
   public void run() {
    TestThreads.printList(10);
   }
  };
  thread1.start();
  thread2.start();
 }
}
```

Implement threads by overriding method **run()** in class Thread

Start threads by using method **start()**

UNIVERSITY OF ABERDEEN

Simple multithreading example (3)

```java
public class TestThreads {
  static void printList(int n) {
    for(int i=1; i<=5; i++) {
      System.out.print(i*n + " ");
    }
    System.out.println();
  }
  public static void main(String args[]){
    Thread thread1 = new Thread() {
      public void run() {
        TestThreads.printList(1);
      }
    };
    Thread thread2 = new Thread() {
      public void run() {
        TestThreads.printList(10);
      }
    };
    thread1.start();
    thread2.start();
  }
```

```
$ java TestThreads
10 20 1 30 2 40 3 50
4 5
$
```

Prints numbers 1, 2, 3, 4, 5

Prints numbers 10, 20, 30, 40, 50

Both threads will run their own instance of method `printList()` in parallel

Thread interference (1)

- Threads may interfere with each other when they access the same data simultaneously, leading to memory inconsistency

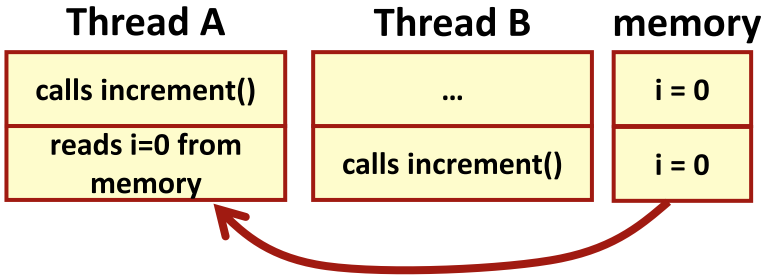
```java
class Counter {
  private int i = 0;
  public void increment() {
    i++;
  }
  ...
}
```

Thread A	Thread B	memory
calls increment()	…	i = 0

Thread interference (2)

- Threads may interfere with each other when they access the same data simultaneously, leading to memory inconsistency

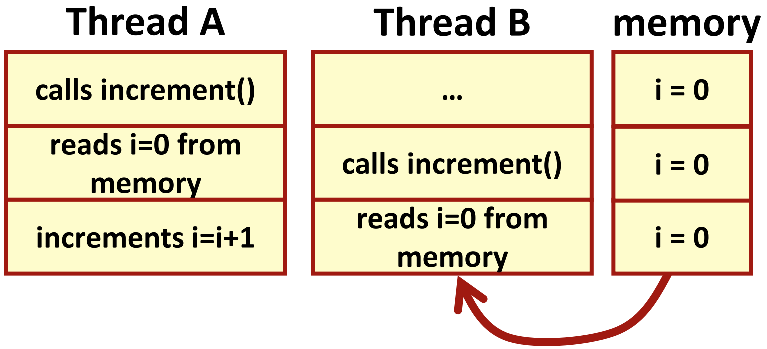
```
class Counter {
  private int i = 0;
  public void increment() {
    i++;
  }
  ...
}
```



Thread interference (3)

- Threads may interfere with each other when they access the same data simultaneously, leading to memory inconsistency

```
class Counter {
  private int i = 0;
  public void increment() {
    i++;
  }
  ...
}
```

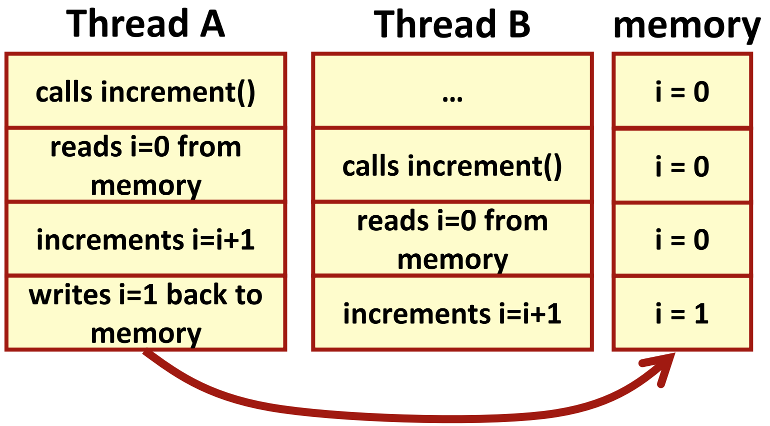


Thread A	Thread B	memory
calls increment()	…	i = 0
reads i=0 from memory	calls increment()	i = 0
increments i=i+1	reads i=0 from memory	i = 0

Thread interference (4)

- Threads may interfere with each other when they access the same data simultaneously, leading to memory inconsistency

```
class Counter {
  private int i = 0;
  public void increment() {
    i++;
  }
  ...
}
```

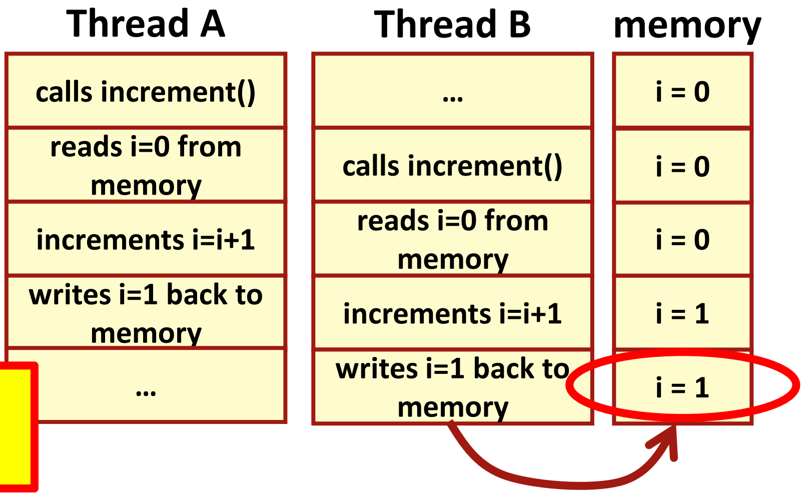


Thread A	Thread B	memory
calls increment()	…	i = 0
reads i=0 from memory	calls increment()	i = 0
increments i=i+1	reads i=0 from memory	i = 0
writes i=1 back to memory	increments i=i+1	i = 1

Thread interference (5)

- Threads may interfere with each other when they access the same data simultaneously, leading to memory inconsistency

```
class Counter {
  private int i = 0;
  public void increment() {
    i++;
  }
  ...
}
```



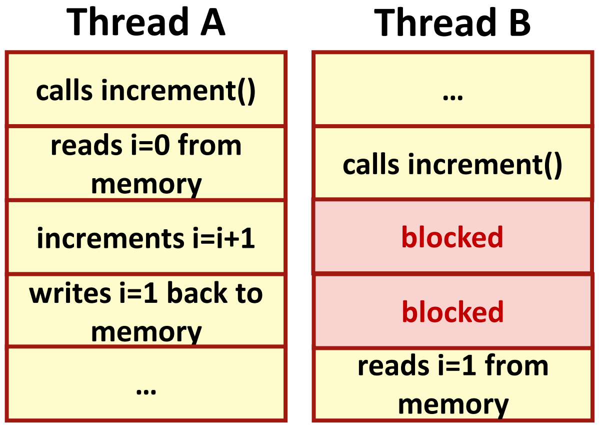
Thread A	Thread B	memory
calls increment()	…	i = 0
reads i=0 from memory	calls increment()	i = 0
increments i=i+1	reads i=0 from memory	i = 0
writes i=1 back to memory	increments i=i+1	i = 1
…	writes i=1 back to memory	i = 1

Two threads invoked `increment()`, but `i` is incremented only once!

Solution: synchronisation

- Synchronisation is one solution to thread interference

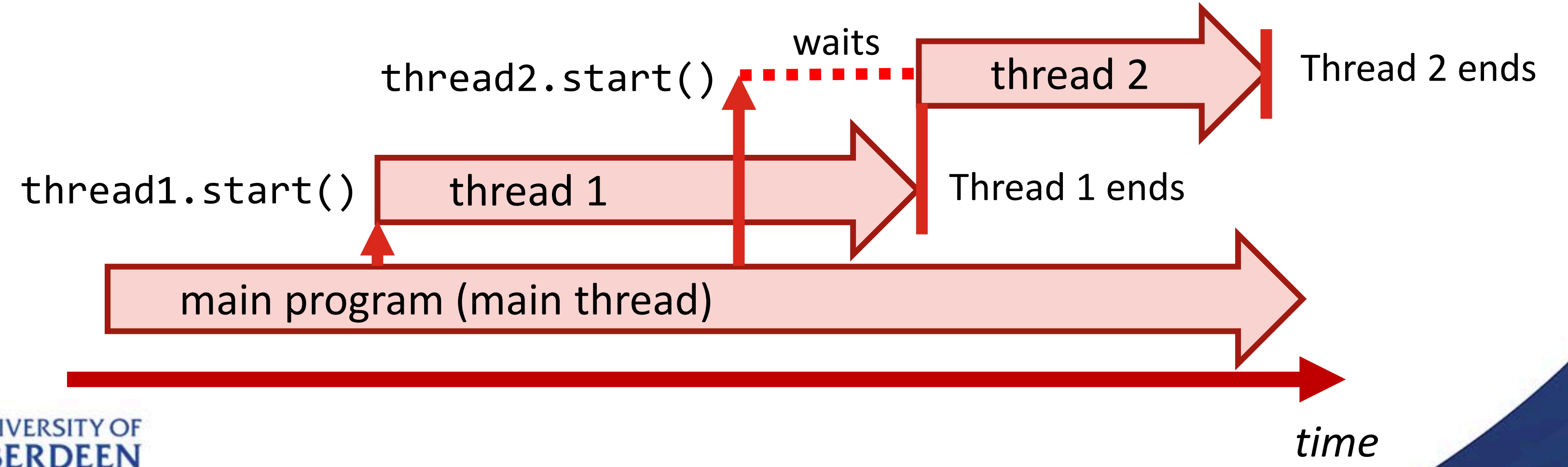
```
class Counter {
  private int i = 0;
  public synchronized void increment() {
    i++;
  }
  ...
}
```



Thread A	Thread B
calls increment()	…
reads i=0 from memory	calls increment()
increments i=i+1	blocked
writes i=1 back to memory	blocked
…	reads i=1 from memory

Synchronising threads

- When multiple threads are running independently, things can happen in an unexpected order
  - Using keyword **`synchronized`** will “lock” the method execution and force other threads to wait until the execution completes



Multithreading with synchronized

```java
public class TestThreads2 {
  synchronized static void printList(int n) {
    for(int i=1; i<=5; i++) {
      System.out.print(i*n + " ");
    }
    System.out.println();
  }
  public static void main(String args[]){
    Thread thread1 = new Thread() {
      public void run() {
        TestThreads2.printList(1);
      }
    };
    Thread thread2 = new Thread() {
      public void run() {
        TestThreads2.printList(10);
      }
    };
    thread1.start();
    thread2.start();
  }
}
```

Waits for thread1 to finish before starting

```
$ java TestThreads2
1 2 3 4 5
10 20 30 40 50
$
```

Apart from keyword synchronized, this example is the same as the previous one!

Threads with sleep()

```java
public class TestThreads3 {
  static void countDown(){
    System.out.print("Seconds to launch: ");
    for(int i=10; i>0; i--) {
      System.out.print(i + " ");
      try {
        Thread.sleep(1000);
      } catch(Exception e) {}
    }
    System.out.println("WHOOOSSH!");
  }
  public static void main(String args[]){
    Thread thread1 = new Thread() {
      public void run() {
        TestThreads3.countDown();
      }
    };
    thread1.start();
  }
}
```

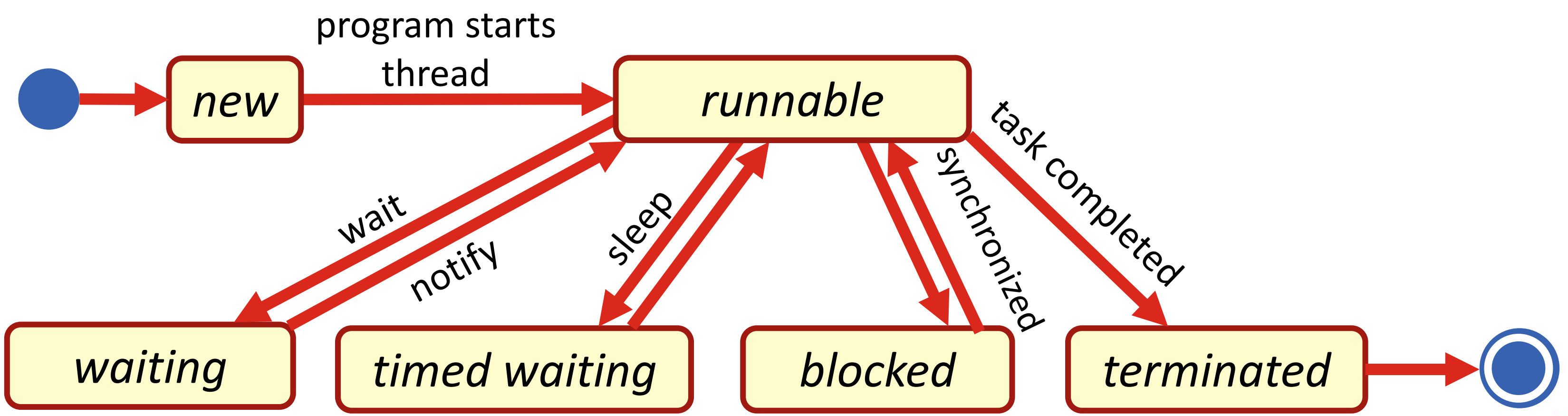
Static method `sleep()` of class Thread stops the thread temporarily and continues after the time given as parameter (in milliseconds) has passed

The numbers will appear with one second intervals!

```
$ java TestThreads3
Seconds to launch: 10 9 8 7 6 5 4 3 2 1 WHOOOSSH!
$
```

Thread life cycle

- Threads can be in different states; after termination, the thread cannot be started again (however, you can create a new thread)



Interruptions in multithreading

- When a Java thread is on a waiting state, e.g. after invoking `sleep()`, another thread can try to interrupt it by invoking its **`interrupt()`** method: in this case, ***`InterruptedException`*** is thrown
  - *`InterruptedException`* is a checked exception, so exception handler (`try`…`catch` structure) is required when `sleep()` is invoked

Example of InterruptedException (1)

```java
public class TestThreads4 {
  static void countDown(){
    System.out.print("Seconds to launch: ");
    for(int i=10; i>0; i--) {
      System.out.print(i + " ");
      try {
        Thread.sleep(1000);
      } catch(InterruptedException e) {
        System.out.print("interrupt ");
      }
      System.out.println("WHOOOSSH!");
    }
  }
  public static void main(String args[]){
    Thread thread1 = new Thread() {
      public void run() {
        TestThreads4.countDown();
      }
    };
    thread1.start();
    thread1.interrupt();
  }
}
```

In this example, the thread continues running normally after `InterruptedException` is handled

```
$ java TestThreads3
Seconds to launch: 10 interrupt 9 8 7 6 5 4 3 2 1 WHOOOSSH!
$
```

Example of InterruptedException (2)

```java
public class TestThreads4 {
  static void countDown(){
    System.out.print("Seconds to launch: ");
    for(int i=10; i>0; i--) {
      System.out.print(i + " ");
      try {
        Thread.sleep(1000);
      } catch(InterruptedException e) {
        System.out.print("interrupt ");
        return;
      }
      System.out.println("WHOOOSSH!");
    }
  }
  public static void main(String args[]){
    Thread thread1 = new Thread() {
      public void run() {
        TestThreads4.countDown();
      }
    };
    thread1.start();
    thread1.interrupt();
  }
}
```

In this example, the thread ends when `InterruptedException` is catched

```
$ java TestThreads3
Seconds to launch: 10 interrupt
$
```

Questions, comments?