Object-Oriented Programming

Concurrent Programming

Dr. Seán Russell
sean.russell@ucd.ie

School of Computer Science,
University College Dublin

November 26, 2018

Learning outcomes

After this lecture and the related practical students should...

- understand the concept of concurrent programming in Java
- be able to create multi threaded programs in Java
- understand the use of join, yield, sleep and interrupt to control the execution of threads

Table of Contents

Concurrent Programming

- Concurrent programming is where several computations are executed at the same time
- Concurrent programming can be achieved using multiple **processes** or multiple **threads** or both
- This is a concept that you will study in your Operating Systems class
- In Java, concurrent programming is implemented using multiple threads

Concurrent Programming

- Multiple programs - same processor - one program at a time
- One program performs IO another program uses the CPU
- Modern processors allow multiple calculations at the same time
- We can write programs that are able to perform many calculations at the same time and complete their tasks quicker

What is a Thread?

- A thread is a sequence of instructions that are executed in order
- Java programs are executed in a thread created based on the main method
- Every program we write in Java will have at least one thread
- If we wish to have our programs complete more than one action at a time we can add more threads to our programs

Thread Names

- Threads have a name
- Remember every exception begins with the words `"Exception in thread"`
- The name can be passed as a string to the constructor of the thread class, and returned using the method `getName`
- If a name is not supplied by the programmer, the system will automatically generate a name such as `"thread-1"`

Other Threads

- In the programs we have written we have only created a single thread
- However, the JVM may have created threads
- User interfaces require multiple different threads to manage things like listening for key presses and mouse actions or redrawing the screen
- Additionally, features of the language such as memory management require some threads
- When we add more threads to our programs, they are known as multithreaded
- To do this we must use the `Thread` class

Table of Contents

Creating Threads

- There are two ways to create a thread
  - Extend the Thread class
  - Implement the Runnable interface and pass a Runnable object to a new Thread object
- When we have created a thread, we must start its execution in the correct way
- If this is not done correctly, the code will be executed sequentially rather than concurrently

Table of Contents

Extending Thread

- To define a new thread using the Thread class, we override a method called `run`
- The signature of run is `public void run()`
- In the run method, we add the code to be executed
- When the thread object is started, these actions will be performed

Extending Thread

- We will write a simple example class
- This class is called `AThread`, and will simply print the letter `'a'` to the screen 30 times

```java
public class AThread extends Thread {
  public void run() {
    for (int i = 0; i < 30; i++) {
      System.out.print('a');
    }
  }
}
```

- The code for printing `'a'` to the screen is exactly the same as if it was in the main method

Table of Contents

Implementing Runnable

- To define a new thread using the Runnable interface, we have to implement a method called `run`
- The signature of run is `public void run()`
- In the run method, we add the code to be executed
- When the runnable object is passed to a thread object, and the thread object is started, these actions will be performed

Implementing Runnable

- We will write a simple example class
- This class is called `BRunnable`, and will simply print the letter `'b'` to the screen 30 times

```java
public class BRunnable implements Runnable {
  public void run() {
    for (int i = 0; i < 30; i++) {
      System.out.print('b');
    }
  }
}
```

How To Choose

- When we need to create a thread, which of these techniques should we choose?
- Generally, it is easier to manage when we extend the Thread class
- This is the technique we should use whenever possible
- Because each class can only have a single superclass, if the class needs to inherit functionality it is not possible to extend the Thread class
- In these cases, we should implement the runnable interface instead

Table of Contents

Executing Threads

- The execution of a thread is done by calling the method start and not by calling the method run
- If you call the run method instead of the start method, the code will be executed sequentially
- When the start method is called, the new thread is set up and then the run method is executed in this new thread

Starting a Thread

- If we have extended the Thread class, we only have to complete two steps;
  - Create an object based on the class that we have defined
    `AThread at = new AThread();`
  - Call the start method
    `at.start();`
- If we have executed this code in the main method, then there will be two threads
  - The first thread executes the main method and
  - the second thread executes the run method of the `AThread` object

Executing a Runnable Object

- If we implemented the Runnable interface, then we are required to execute three steps;
  - Create an object based on the class that we have defined
    ```
    BRunnable br = new BRunnable();
    ```
  - Create a Thread object, and pass the object top its constructor
    ```
    Thread t = new Thread(br);
    ```
  - Call the start method of the thread object
    ```
    t.start();
    ```
- If we have executed this code in the main method, then there will be two threads
  - The first thread executes the main method and
  - the second thread executes the run method of the `BRunnable` object

What Happens?

- Consider the following example, what order will the characters be printed in?

```
public static void main(String[] args) {
  AThread at = new AThread();
  at.start();

  BRunnable br = new BRunnable();
  Thread t = new Thread(br);
  t.start();

  for (int i = 0; i < 30; i++) {
    System.out.print('c');
  }
}
```

What Happens?

- We do not know what order the letters a, b and c will be printed in
- An example of the output is

```
aaaaaaaaaaaaaaaaaaaaaaaccccccccccccc
cccccbaaaabbbcbbbcbbbbbbbbbcbbbbbbbb
bbbcccccc
```

- The order that the different threads are chosen for execution, and the amount of statements that a thread will execute are based on many different factors out of our control
- Such as other programs and operating system functions being executed at the same time

Order of Execution

- Sometimes we need to impose order on when certain actions are carried out
- If we have a calculation performed in one thread, and another thread that is supposed to perform some operation with the result, then we need some way to make sure that the calculation is complete before we can use the result
- This means that we need some mechanisms to allow us to control the execution of threads

Table of Contents

Controlling Execution

- There are a number of methods in the Thread class that can be used to control execution
- The following are the most useful
  - join
  - sleep
  - interrupt

Table of Contents

Join API

```
public final void join() throws InterruptedException
```

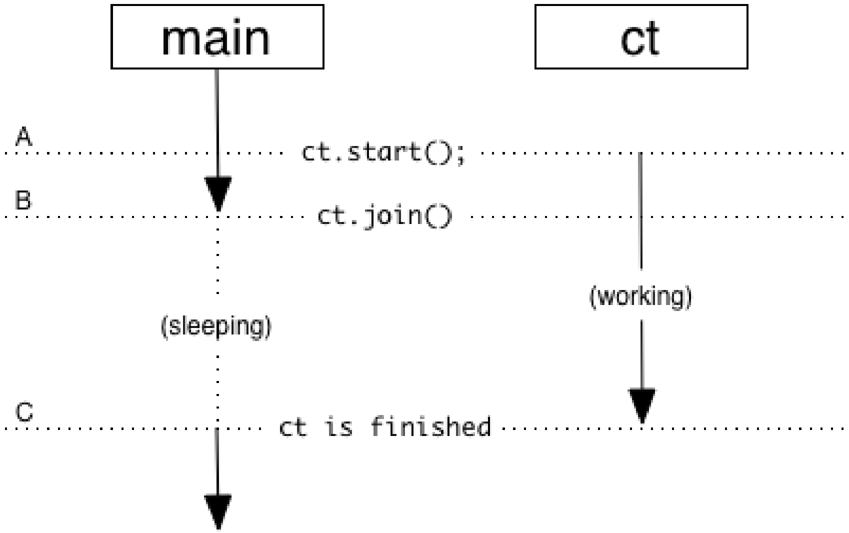
Waits for this thread to die.

Throws: `InterruptedException` - if any thread has interrupted the current thread. The interrupted status of the current thread is cleared when this exception is thrown.

Join

- The join method allows us to **wait** for another thread to finish
- This ensures that the chosen thread is finished before we do something
- If we wish to wait for another thread to finish, we must have a **reference** to that thread

Join



Join Example

```java
public static void main(String[] args) {
  AThread ct = new AThread();
  ct.start();
  // do some other calculations as needed
  try{
    ct.join();
  }catch(InterruptedException e){
    System.out.println("main was
   interrupted");
  }
  for (int i = 0; i < 30; i++) {
    System.out.print('c');
  }
}
```

Join

- The image shows how execution is changed in the example code
- In this example, at point `A` the main thread creates and starts the thread `ct`
- At point `B`, the main thread executes the join method for the `ct` thread object
- This causes the execution in the main thread to stop, and remain stopped
- This changes at point `C` when the thread `ct` has completed its assigned work and finished
- At this point the main thread can continue executing

One After Another

- This means that whatever order the actions are performed while both threads are finished, the thread `ct` must be complete before the main thread begins printing characters
- In this example, this would result in the following output: `a...ac...c` (all a's followed by all c's)
- If we look at another example, where the main thread is performing some work at the same time as the `ct` thread we will see a different output

Another Example

```java
public static void main(String[] args) {
  AThread ct = new AThread();
  ct.start();
  for (int i = 0; i < 30; i++) {
    System.out.print('d');
  }
  try{
    ct.join();
  }catch(InterruptedException e){
    System.out.println("main was
   interrupted");
  }
  for (int i = 0; i < 30; i++) {
    System.out.print('c');
  }
}
```

Another Example

- In this example, both threads are still performing work at the same time
- So the main thread is printing the character `'d'`, and the `ct` thread is printing the character `'a'`
- Because both of these are happening at the same time, we cannot know what order the result will be in
- However, because the join method is called before we begin printing `'c'` we can be sure that all c's will be printed last (mix of a's and d's followed by only c's)
- This is the level of control that can be achieved using the join method

Table of Contents

Sleep

- Another way to control the execution of threads is using the sleep method
- `sleep` is a class method in `Thread`
- We do not need a reference to use `sleep` and we can only sleep our own thread
- It takes a long as a parameter, this represents a number of milliseconds that the thread should sleep
- During this time no code will be executed by this thread, but all other threads will continue as normal

Sleep API

`public static void sleep(long millis) throws InterruptedException`

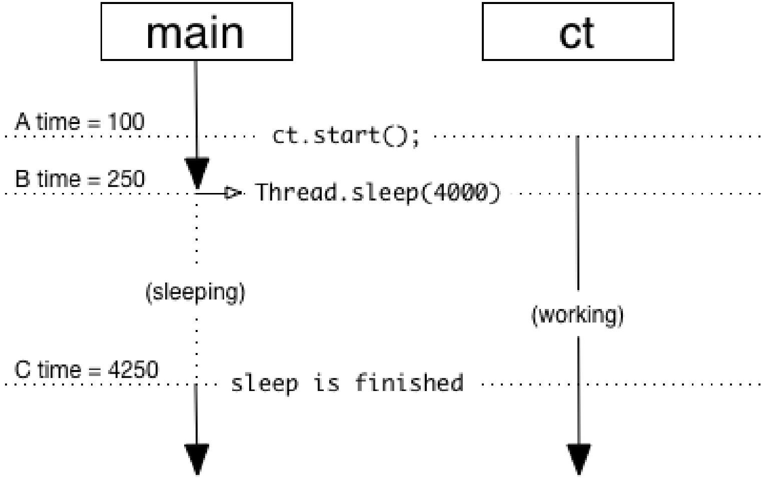
Causes the currently executing thread to sleep (temporarily cease execution) for the specified number of milliseconds, subject to the precision and accuracy of system timers and schedulers.

Parameters: millis - the length of time to sleep in milliseconds

Throws: `IllegalArgumentException` - if the value of millis is negative

`InterruptedException` - if any thread has interrupted the current thread. The interrupted status of the current thread is cleared when this exception is thrown.

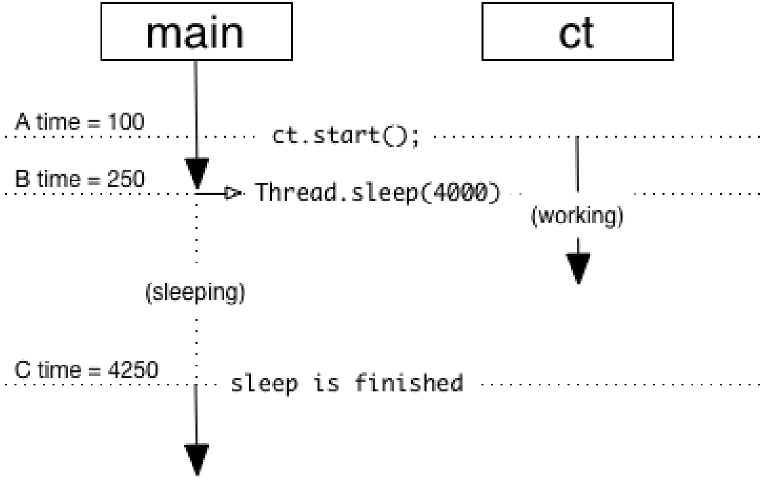
Sleeping Threads



Sleeping Threads

- In this image, the main method calls sleep at point B
- This means that the main method will not execute any code until this time has elapsed
- Once the 4000 milliseconds have passed, the main thread continues executing
- The use of `Thread.sleep()` in the main thread has no effect on the execution of the thread `ct`, which continues executing as normal and is still executing when the main thread continues

Sleeping Threads



Sleeping Threads

- In this image, the main thread also calls sleep at point B
- But the `ct` thread completes its work and finishes before it continues
- Again, the use of `Thread.sleep()` in the main thread has no effect on the execution of the thread `ct`

Sleep Example

```java
public static void main(String[] args) {
  AThread ct = new AThread();
  ct.start();
  try{
    Thread.sleep(4000);
  }catch(InterruptedException e){
    System.out.println("main was interrupted
 while sleeping");
  }
  for (int i = 0; i < 30; i++) {
    System.out.print('c');
  }
}
```

Sleep Example

- Once sleep is called, the main thread will sleep for 4000 milliseconds
- 4 seconds is a long time and `ct` is finished long before the main thread continues executing
- This might lead to confusion between sleep and join,
  - `join` makes this thread wait until another thread is finished (no matter how long or short)
  - `sleep` makes this thread wait for a defined amount of time

Table of Contents

Interrupt

- The interrupt method sends a signal to a thread to `ask` it to stop executing
- Similar to join, we must have a reference to a thread if we wish to interrupt it
- One problem with using the interrupt method is that the thread may simply ignore the signal and continue executing

Interrupt API

`public void interrupt()`

Interrupts this thread.

If this thread is blocked in an invocation of the wait(), wait(long), or wait(long, int) methods of the Object class, or of the join(), join(long), join(long, int), sleep(long), or sleep(long, int), methods of this class, then its interrupt status will be cleared and it will receive an InterruptedException.

If none of the previous conditions hold then this thread's interrupt status will be set. Interrupting a thread that is not alive need not have any effect.

Throws: `SecurityException` - if the current thread cannot modify this thread

Handling Interrupts

- Handling an interrupt means checking if we have been sent the interrupt signal
- This is usually done as the condition of a loop
- Every time we complete a task, before we begin the next task we check the condition
- If the interrupt is received, the loop is not executed
- To do this we use the class method `interrupted`
- This returns a boolean, true if we have been interrupted

Checking for Interrupt Signal

```java
public class ItThread extends Thread {
  public void run(){
    while(!Thread.interrupted()){
      System.out.println("doing work");
    }
  }
}
```

- The condition requires the binary operator **not**
- The value returned by the operator is not `true` when the thread has not been interrupted
- This allows the while loop to execute as long as the thread has not been interrupted

Interrupting a Thread

```java
public static void main(String[] args)
    throws InterruptedException {
  ItThread it = new ItThread();
  it.start();

  Thread.sleep(1000);

  it.interrupt();
}
```

Interrupting a Thread

- The example shows how a thread can be interrupted
- Here the thread `it` is created and started
- The thread `it` has no finishing condition
- Once the interrupt is received, the thread will continue performing its current task
- However, once the task is completed it will check the condition of the loop and discover that it has been interrupted
- This allows the thread to finish

Table of Contents

Thread Safety

- Threads share the same data address space
- This means that they can share objects and variables
- This means that one thread may change an object in a way that the other is not expecting, it can be difficult to know what the result of this will be

Sharing Variables

- When variables are shared between threads, we must make sure that they are used in a way that is safe
- The idea of safety in threads is explained easily with an analogy, consider two trains that must cross the same bridge with only one set of tracks
- Without some organisation, a disaster would surely happen the next time that both trains wanted to cross the bridge at nearly the same time

Table of Contents

Synchronisation

- Synchronisation is the process of ensuring that these events cannot happen
- In computer programming, synchronisation the idea that multiple processes reach an agreement or commit to a certain **sequence of actions**
- Synchronisation is achieved through the use of mutual exclusion

Mutual Exclusion

- Mutual exclusion is making sure that a resource can only be used by thread at a time
- In the train analogy, it means that only one train is allowed to use the bridge at a time
- Synchronisation generally uses mutual exclusion to decide the sequence of actions
- For example, the easiest solution is that whichever train arrives at the bridge first gets to use it, and the other train must wait

Adder

```java
public class Adder {
  int x, y;
  public void setXY(int x1, int y1){
    x = x1;
    y = y1;
  }
  public int sumXY(){
    return x + y;
  }
}
```

- This class only performs a very simple function, but it is designed to represent the how shared objects are used in much more complicated classes

Sharing the Adder Object

- The basic function of the methods are to set some state in the object (`setXY`) and then to perform some calculation based on the state of that object
- If we assume that this same object is being used by two separate threads, what will be the result of the operations?
- There are different situations to look at so lets have a closer look at the order that the statements might be executed in.

Thread 1	Thread 2
`a.setXY(1,2);`	`a.setXY(3,4);`
`int w = a.sumXY();`	`int z = a.sumXY();`
`System.out.println(w);`	`System.out.println(z);`

Different Situations

1. What happens if thread 1 completes all statements before thread 2 calls `setXY`?
2. What happens if thread 1 calls `setXY` between thread 2 calling `setXY` and `sumXY`?
3. What happens if thread 1 is calling `setXY` while thread 2 is calling `sumXY`?

1

- What happens if thread 1 completes all statements before thread 2 calls `setXY`?
- Here, the answer is pretty easy, thread 1 will print the number 3 and thread 2 will print the number 7
- Both threads correctly calculating the result
- This is the way that the object is supposed to be used

2

- What happens if thread 1 calls `setXY` between thread 2 calling `setXY` and `sumXY`?
- Again, the answer is again pretty easy, thread 1 will print the correct result of 3
- However, the values were set by thread 1 before thread 2 performed its calculations, thread 2 will also print the incorrect result of 3

3

- What happens if thread 1 is calling `setXY` while thread 2 is calling `sumXY`?
- The answer will be determined by the order that the individual statements are executed
- If the addition is performed first, then thread 2 may get the correct answer
- However, it is also possible that both assignments will be completed first and that it will get the answer thread 1 was expecting
- Lastly, if the addition is performed between the two assignments, we may get an entirely different value based on the first value from thread 1 and the second value from thread 2 giving us the result of 5

Table of Contents

Synchronised Methods

- Java allows us to prevent the problem 3 by defining methods as synchronised
- Only one synchronised method can be used in an object at any time
- Another method that tries to use a synchronised method while we are executing one, will have to wait
- To declare a method as synchronised, we add the keyword `synchronized` before the return type
- To correctly prevent methods unknown behaviour by method sharing data, we must define all methods that can influence each other as synchronised
- Any method that is not synchronised may be used at any time

Synchronised Adder

```java
public class Adder {
  int x, y;

 public synchronized void setXY(int x,int y){
    this.x = x;
    this.y = y;
  }

  public synchronized int sumXY() {
    return x + y;
  }
}
```

Synchronised Adder Example

- As both methods are defined as synchronised, it is impossible to execute the methods at the same time
- We had three possible outcomes based on the order of execution of the two threads
- First, the correct answer
- Second, an incorrect but predictable answer
- Finally, a unknown result
- It should be noted that now that the methods are synchronised, we can no longer get an unknown result
- However, it is still possible that we get the incorrect result