Threads

Slide 1

- When writing games you need to do more than one thing at once.
- Threads offer a way of automatically allowing more than one thing to happen at the same time. Java has threads as a fundamental feature in the language. This will make your life much easier when you write games in Java.
- Threads are related to processes and multitasking.

Example

Slide 2

- Suppose that you are a poor student and you need to earn money. To earn money you get a job answering technical support calls for a company specialising in software to derive new weaving patterns for Turkish carpets. As it turns out there is not a very large user base and you only get 4 or 5 calls a day (but the pay is good).
- You have to revise for an exam. You hit upon the idea of revising while you are waiting for calls. When a call interrupts you go away from you books and deal with the customer.

Example Continued

Slide 3

- The last example was easy. You had plenty of time to go back and forward between the books and the exam.
- Now consider a more important situation. You are going to play in a counter strike tournament and at the same time you need to watch an important Hockey match between Sweden and Russia while you are doing your revision.
- You can again multi-task. You spend a little bit of time on each activity swapping between them. If you are fast enough you might be able to do all three (almost at the same time).

Context Switching

Slide 4

- Going back to the first example. Suppose that you have spent too much of your time on fast living and your memory is not as good as it should be.
- When you answer the phone you put a bookmark in the book so you can remember where you are.
- When you go back to the book you use the bookmark to work out where you are.
- This is an example of a *context switch*. You want to go from one task to another you save information to remember where to go back to.

Context Switching

Slide 5

- The computer does the same thing with running programs. If it wants to switch tasks. It saves all the information needed (program counter, values of variables), so the program can be returned to.
- This can happen without the program knowing it. Think of it as cryogenetics for programs.
- The operating system forcefully stops the current task saves that state and starts another task from a previously saved state.

Threads and Processes

Slide 6

- There is a technical distinction between threads and processes. That need not worry us here. You can think of a process as a whole program (such as emacs, word, minesweeper, mozilla). The operating systems allows more than one process to run at the same time.
- A thread is a unit of execution with in a process. That has less overhead and quicker context switch time.
- Threads can share data while processes can not (not always true but true enough).
- Threads are lightweight processes.

Multitasking

Slide 7

- A computer can give the illusion of doing more than one thing at once by time slicing.
- Each task is given a small amount of time to run before the next task is switched in.
- Context switches take care of all this.
- Of course every thing runs slower, but you have more flexibility.

Threads and Java

Slide 8

- In Java there is a special thread class that manages all the above for you.
- To create a new thread you could do the following :

```
Thread myThread = new Thread();
mythread.start();
```

This would not do much since the default thread class does not do anything (apart from managing all the context switching stuff).

Threads and Java

You have three choices:

Slide 9

- Extend the Thread class
- Implement a runnable interface
- Use anonymous classes.

Simple example

Slide 10

```
public class MyThread extends Thread {
   public void run() {
    System.out.println("Do Something.");
  }
}
```

Then later on when you actually want to do something.

```
  MyThread myThread = new MyThread();
```

You can even use constructors (see next example).

Slide 11

```
public class MyThread2 extends Thread {
    private int theadid;
    public  MyThread2(int id) { threadid = id;  }
    public void run() {
        for(int i=0; i<10000 ; i++) {
            System.out.println(threadid + " : " + i);
        }
    }
    public static void main(String[] args) {
        Thread thread1 = new MyThread2(1);
        Thread thread2 = new MyThread2(2);
        thread1.start();
        thread2.start();
    }
}
```

Slide 12

Interfaces

- Inheritance (`extends`) has the problem you can only extend from one class.
- This would make your program design quite hard.
- To get over this we can use interfaces.

  In general, an interface is a device or a system that unrelated entities use to interact. According to this definition, a remote control is an interface between you and a television set, the English language is an interface between two people.

Interfaces

Slide 13

- A class can only extend another class while it can have many interfaces.
- Interfaces are declared in a similar way to classes. You probably won't need to define any yourself.
- But you will need to use some of the system provided ones for example `MouseListener` , `Runnable`, `MouseEvent` , ...

Slide 14

```
public class MyThread3 implements Runnable {
    public MyThread3() {
        Thread thread = new Thread(this);
        thread.start();
    }
    public void run() {
        System.out.println("Hello.");
    }

    public static void main(String[] args) {
        MyThread3 thread  = new MyThread3();
    }
}
```

Exercise rewrite the first two examples using runnable.

Detour - Static Members

.

Slide 15

- A static member of a class has only one instance no matter how many times the class is instantiated.
- Question what is printed on the following example and why?
- What happens if we change `static int counter` to `int counter`
- Static members are best avoided (they are error prone and it is hard to debug what is going on). But they are sometimes useful.

Slide 16

```
public class StaticExample {
    static int counter = 0;

    public  StaticExample() { counter++;}
    public  int HowManyTimes() { return(counter);}

    public static void main(String[] args) {
        StaticExample x = new StaticExample();
        StaticExample y = new StaticExample();
        StaticExample z = new StaticExample();
        System.out.println(z.HowManyTimes());
    }
}
```

When does the context switching happen?

Slide 17

- It is up to the runtime system when the context switch happens. This means it could happen while you are doing something.
- In particular you could be updating a piece of shared data and the runtime system swaps tasks during this. The system is then left in an inconsistent state.
- In the following example we require (for illustration only) that `xpos` and `ypos` are always equal.
- We put a test to see if they are equal.
- The thread itself is very simple.

Slide 18

```
public class BadSynch extends Thread {

    static private int xpos;
    static private int ypos;

    public void run() {
        for(int i=0; i<1000; i++) {
            xpos++;
            ypos++;
            if(xpos != ypos) {
                System.out.println(":-( " + xpos + " " + ypos);
            }
        }
    }
```

Slide 19

If we create two threads and run them in parallel.

```
    public static void main(String[] args) {
        Thread thread1 = new BadSynch();
        Thread thread2 = new BadSynch();
        thread1.start();
        thread2.start();
    }
}
```

We will see lots of unhappiness. (Try it and see).

Slide 20

Be Warned!!!

- With threads you do not know which order things will be executed or when the context switched.
- It might be luck that that the program works.

Version without static members

Slide 21

- Instead of using static members, we should use objects and object references to communicate.
- The general idea is that you give the reference of an object to multiple threads to communicate with.
- To redo the previous example we use a class `DoubleCounter` to hold the values of `x` and `y`.

Slide 22

```
public class DoubleCounter {

    private int x;
    private int y;

    public DoubleCounter() {
        x = 0; y = 0;
    }
    public void add_one() { x++; y++;}
    public boolean happy() {
      if (x==y) { return(true);}
       else { return(false); }
    }
}
```

We can then redo the example by passing a double counter object via the constructor for the class.

Slide 23

```
public class BadSynch extends Thread {
    private DoubleCounter counter;
    public BadSynch(DoubleCounter newcounter) {
        counter = newcounter;
    }
    public  void run() {
        for(int i=0; i<1000000; i++) {
            counter.add_one();
            if(!counter.happy()) { System.out.print(" !-( "); }
        }

    }
```

Then we create one double counter and pass this to both threads.

Slide 24

```
    public static void main(String[] args) {

        DoubleCounter sharedcounter = new DoubleCounter();
        Thread thread1 = new BadSynch(sharedcounter);
        Thread thread2 = new BadSynch(sharedcounter);
        thread1.start();
        thread2.start();
    }

}
```

Run it and see, you will still see lots of unhappiness.

Synchronize

Slide 25

- The whole problem can be avoided by putting locks on data.
- You can put a piece of data in a room with a door. When a thread wants to modify the data it goes in through the door locks it so nobody else can get in and modify it.
- When it has finished it unlocks the door and leaves.
- This is achieved in Java with the `Syncrhonized` keyword.

Example

Slide 26

Replace with the new DoubleCounter and expect no unhappiness.

```
public class DoubleCounter {
    private int x;
    private int y;
    public  DoubleCounter() {
        x = 0; y = 0;
    }
    public synchronized  void add_one() { x++; y++; }
    public synchronized boolean happy() {
        if (x==y) {return(true);}
        else {return(false);}}
```

Object Synchronisation

You can also objects as synchronisation points. You could rewrite the

above as:

Slide 27

```
public  void add_one() {
      synchronized(this)
                     {x++; y++;}
}
```

Remember `this` is a reference to the current object.

When to synchronise

Slide 28

- When two or more threads access the same piece of data.
- Don’t over synchronise. Synchronisation points force other threads to wait.
- Use the `syncrhonized(this)` construction to only synchronise at the critical points.

Anonymous Inner Classes

Slide 29

```
new Thread() {
   public void run() {
       System.out.println("Hello.");
   }
}.start();
```

Should only be used for short fragments of code.

Deadlock

Slide 30

- The synchronise statement forces other threads to wait if they are accessing shared data.
- You have the situation where a thread has locked a piece of data that another thread wants which is locking a piece of data that the first thread wants.

Deadlock example

Slide 31

- We will use *semaphores* to illustrate deadlock.
- Semaphores are shared variables (and in Java guarded by getters and setters with `syncrhonized` statements). Where one value means locked and another unlocked.

Semaphore Variables

Slide 32

```
public class Deadlock  {
    static int lockA=0; /* 0 if unlock 1 if locked. */
    static int lockB=0;

    public static synchronized int getLockA() {
        return lockA;
    }
    public static synchronized void setLockA(int newLockA) {
        lockA = newLockA;
    }
```

Slide 33

```
public static synchronized int getLockB() {
    return lockB;
}

public static synchronized void setLockB(int newLockB) {
    lockB = newLockB;
}
```

Slide 34

Threads

In this example we will use anonymous classes. The first thread locks A then locks B, then unlocks B and unlocks A.

First lock A and then do some work.

```
    new Thread() {
        public void run() {
            System.out.println("P1: Waiting for A.");
            while(getLockA()==1) {;}
            System.out.println("P1: Got A"); setLockA(1);
            try { Thread.sleep(5000); }
              catch (InterruptedException x) {}
```

First thread continued

Slide 35

Now try and lock B then unlock.

```
            System.out.println("P1: Waiting for B.");
            while(getLockB()==1) {;}
            setLockB(1);
            System.out.println("P1: Got B.");

            setLockB(0);
            setLockA(0);
        }
    }.start();
```

Code for the second thread

Slide 36

```
    new Thread() {
        public void run() {
            System.out.println("P2: Waiting for B.");
            while(getLockB()==1) {;}
            System.out.println("P2: Got B"); setLockB(1);
            System.out.println("P2: Waiting for A.");
            while(getLockA()==1) {;}
            System.out.println("P2: Got A"); setLockA(1);
            setLockA(0); setLockB(0);
        }
    }.start();
```

Output

Slide 37

```
P1: Waiting for A.
P1: Got A
P2: Waiting for B.
P2: Got B
P2: Waiting for A.
P1: Waiting for B.
```

How to avoid Deadlock

Slide 38

- There are no hard and fast rules to avoid deadlock.
- Essentially you have to look for cycles.
- There are tools that help, but they don't help that much.
- The interleaving of threads is not defined, so sometimes that code might deadlock sometimes it might not. In the previous example if thread 1 completely finishes before thread 2 starts then there should be no problem.

Wait and notify

Slide 39

Statements such as

```
while(getLockB()==1) {;}
```

Are not so efficient. This is called busy waiting, it has to check each time around the loop . Java provides and pair of statements `wait` and notify.

The behaviour is a bit complicated so I'll just give an example.

pair of statements

Without wait and notify

Slide 40

```
// Thread A
public void waitForMessage() {
  while (hasMessage == false) { ; }
}

// Thread B
  public void setMessage() {
    ....
    hasMessage = true;
}
```

With wait and notify

Slide 41

```
 // Thread A
  public sychrnoized void waitForMessage() {
   try {
     wait();
   }  catch (InterruptedException ex) {}
}
// Thread B
public synhronized void setMessage() {
   ...
   notify();
}
```

wait and notify

Slide 42

- Essentially `wait` releases current locks and gives control back to other threads.
- `notify` wakes up a thread that is waiting.
- If there is more than one `wait` a random thread is woken up.
- `notifyAll();` wakes all the waiting threads.

Threads summary

Slide 43

- Threads allow you to do more than one thing at a time.
- Problems can occur with corrupted shared data.
- `syncrhonized` can solve this problem.
- Don't syncrhonize too much.
- Don't have too many threads (JavaVM can't cope).
- Threads are non-deterministic. That is if you one the program once and it works doesn't mean it is going to work next time. Be careful.