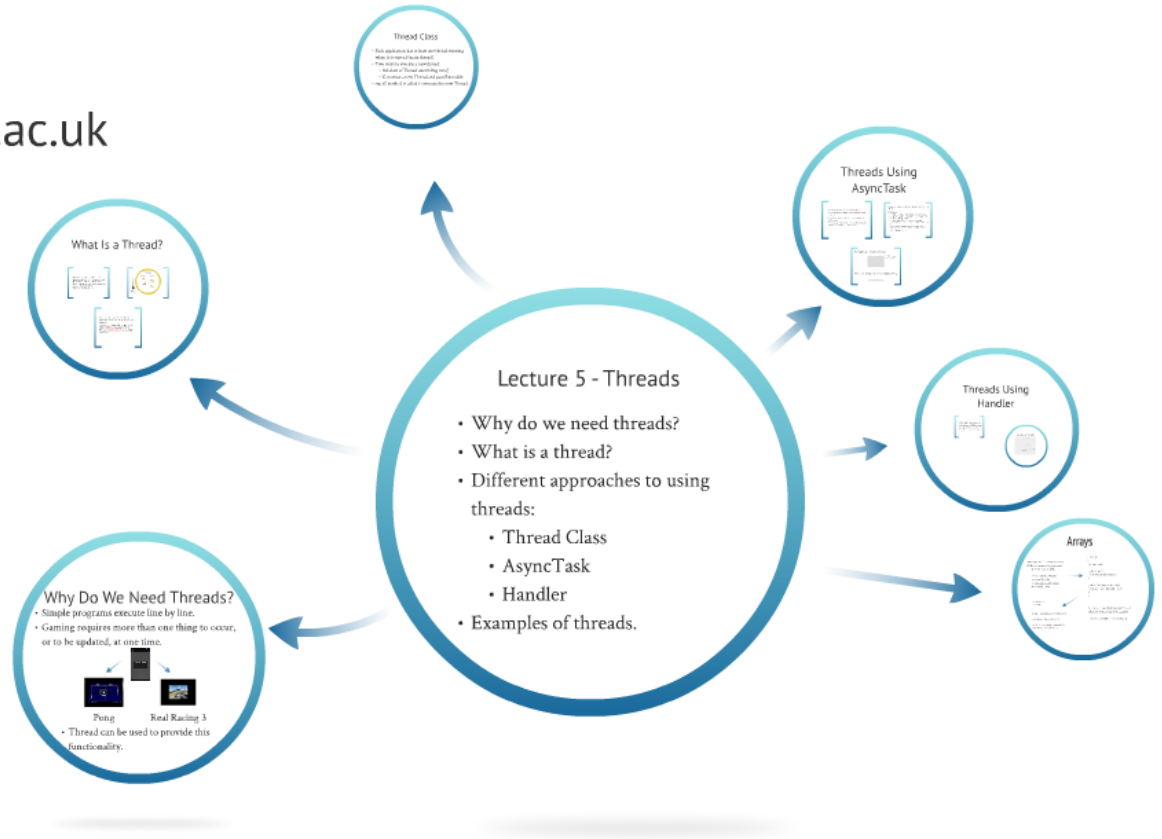
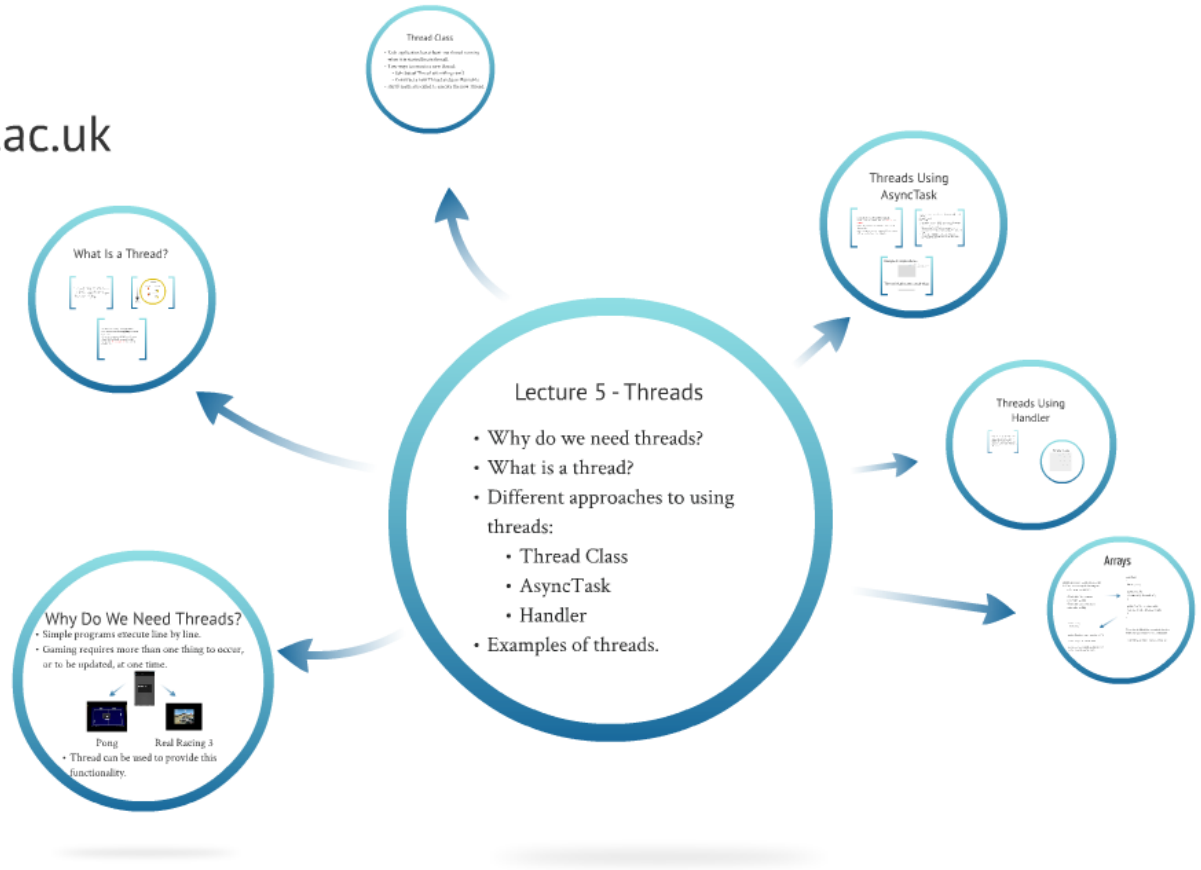
Mobile Games Development (IMAT2608/3608)

Dr Jethro Shell
jethros_at_dmu.ac.uk



Mobile Games Development (IMAT2608/3608)

Dr Jethro Shell
jethros_at_dmu.ac.uk

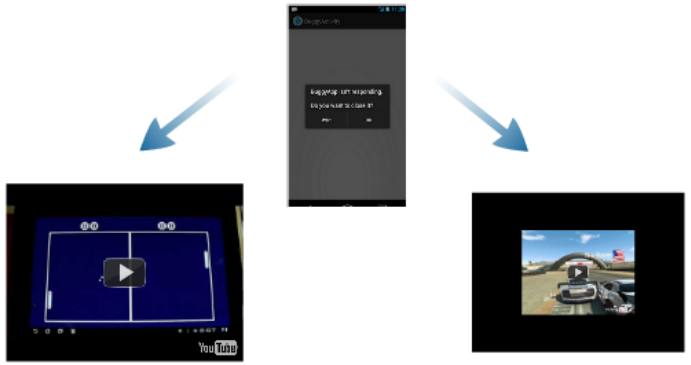


Lecture 5 - Threads

- Why do we need threads?
- What is a thread?
- Different approaches to using threads:
  - Thread Class
  - AsyncTask
  - Handler
- Examples of threads.

Why Do We Need Threads?

- Simple programs execute line by line.
- Gaming requires more than one thing to occur, or to be updated, at one time.



Pong Real Racing 3

- Thread can be used to provide this functionality.

- Line by line, waiting for each instruction to complete.

```
//Get the next line in the file
pParserT.nextLine();

//Initialise the process
pParserT.translate();

//Get the tags of the words in the title.
String [] strTags = pParserT.getTag();
int iCount = strTags.length;

ArrayList<String> arrayCollection = pParserT.getCollection();
```

- Ideal for simple tasks.

truction to complete.

```
//Get the next line in the file
pParserT.nextLine();

//Initialise the process
pParserT.translate();

//Get the tags of the words in the title.
String [] strTags = pParserT.getTag();
int iCount = strTags.length;

ArrayList<String> arrayCollection = pParserT.getCollection();
```

deal for simple tasks.

- Line by line, waiting for each instruction to complete.

```
//Get the next line in the file
pParserT.nextLine();

//Initialise the process
pParserT.translate();

//Get the tags of the words in the title.
String [] strTags = pParserT.getTag();
int iCount = strTags.length;

ArrayList<String> arrayCollection = pParserT.getCollection();
```

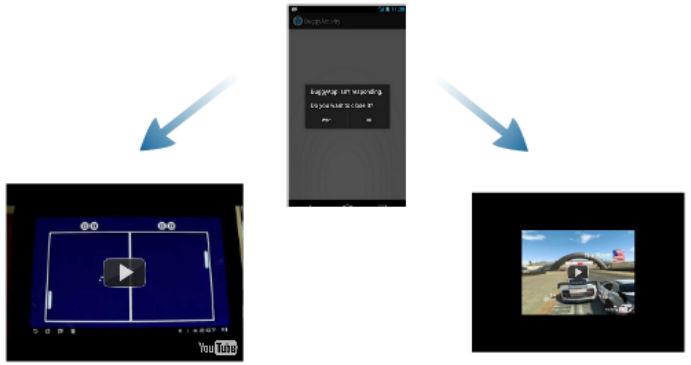
- Ideal for simple tasks.

- Android UI works on a single thread.
- As a result nontrivial applications need to be multithreaded.
- E.G. UI responds to input despite updating from the internet, not hanging.
- UI can not be ordered by long running processes.

- Main thread is in charge of dispatching events (draw events, widgets etc).
- The system does not create a separate thread for each instance of a component.
- Consequently, methods that respond to system callbacks (such as onKeyDown() to report user actions or a lifecycle callback method) always run in the UI thread of the process.
- If everything is happening in the UI thread, performing long operations such as network access or database queries will block the whole UI.
- Even worse, if the UI thread is blocked for more than a few seconds (about 5 seconds currently) the user is presented with the infamous "application not responding" (ANR) dialog.
- KEY POINTS: Do not block the UI thread and Do not access the Android UI toolkit from outside the UI thread.

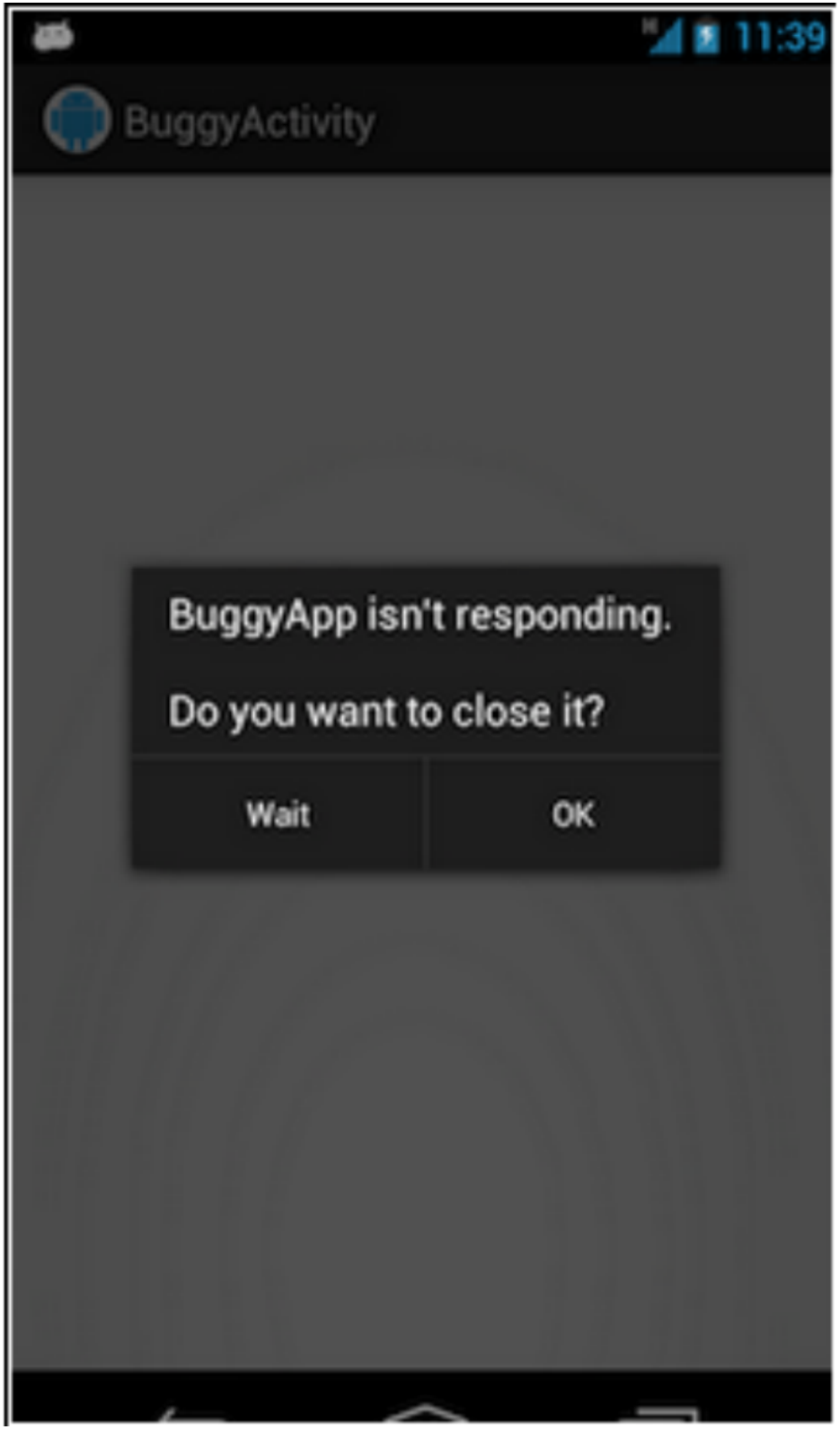
Why Do We Need Threads?

- Simple programs execute line by line.
- Gaming requires more than one thing to occur, or to be updated, at one time.



Pong Real Racing 3

- Thread can be used to provide this functionality.

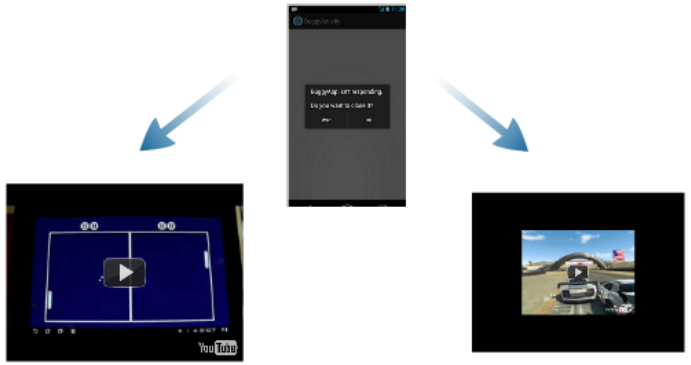

11:39
BuggyActivity
BuggyApp isn't responding.
Do you want to close it?
Wait
OK


0 0
0 0
2:07
YouTube


The Raven
lap 1/5
30.086
12th
166 kph
YouTube

Why Do We Need Threads?

- Simple programs execute line by line.
- Gaming requires more than one thing to occur, or to be updated, at one time.

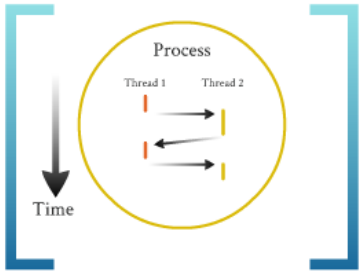


Pong Real Racing 3

- Thread can be used to provide this functionality.

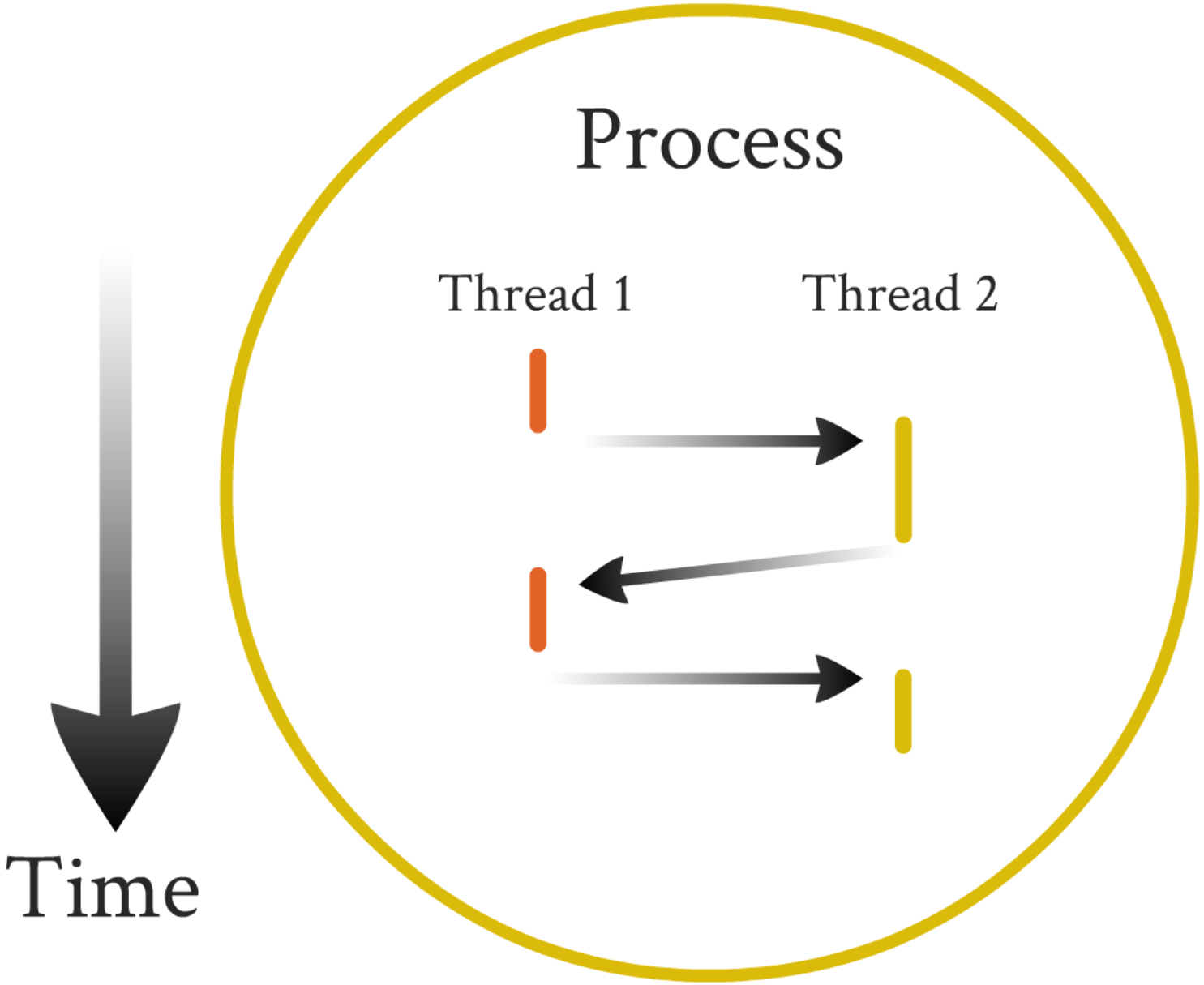
What Is a Thread?

- Two concurrent programs are known as processes.
- Separate tasks performed by a single program are threads.
- Threads can be referred to as lightweight processes.
- Threads do not completely separate areas of memory, they share code, data areas and context.
- They work on time-slicing principles.



- For data intensive, long running operations improvements can be made by splitting into smaller operations.
- For example, decoding multiple image files to be displayed as thumbnails on separate threads.
- We can use ThreadPoolExecutor to achieve this (optimisation).

- Two concurrent programs are known as processes.
- Separate tasks performed by a single program are threads.
- Threads can be referred to as lightweight processes.
- Threads do not completely separate areas of memory, they share code, data areas and context.
- They work on time-slicing principles.


Process
Thread 1
Thread 2
Time

- For data intensive, long running operations improvements can be made by splitting into smaller operations.
- For example, decoding multiple image files to be displayed as thumbnails on separate threads.
- We can use ThreadPoolExecutor to achieve this (optimisation).

Thread Class

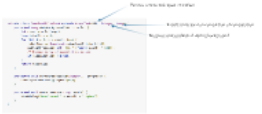
- Each application has at least one thread running when it is started (main thread).
- Two ways to execute a new thread:
  - Subclass of Thread overriding run()
  - Construct a new Thread and pass Runnable
- start() method is called to execute the new Thread.

Threads Using AsyncTask

- AsyncTask is part of the android.os package.
- It is designed to be a helper class around Thread and Handler.
- AsyncTasks should be used for short operations, just a few seconds.
- For longer operations, it is recommended to use other API's such as handler or thread directly.

- An asynchronous task runs on a background thread and is published on the UI thread.
- This makes it thread safe.
- It has four steps:
  - onPreExecute - invoked on the UI thread before the task is executed.
  - doInBckground- invoked on the background thread immediately after onPreExecute() finishes executing.
  - onProgressUpdate- This method is used to display any form of progress in the user interface while the background computation is still executing.
  - onPostExecute - invoked on the UI thread after the background computation finishes. The result of the background computation is passed to this step as a parameter.

Example of a Async subclass.

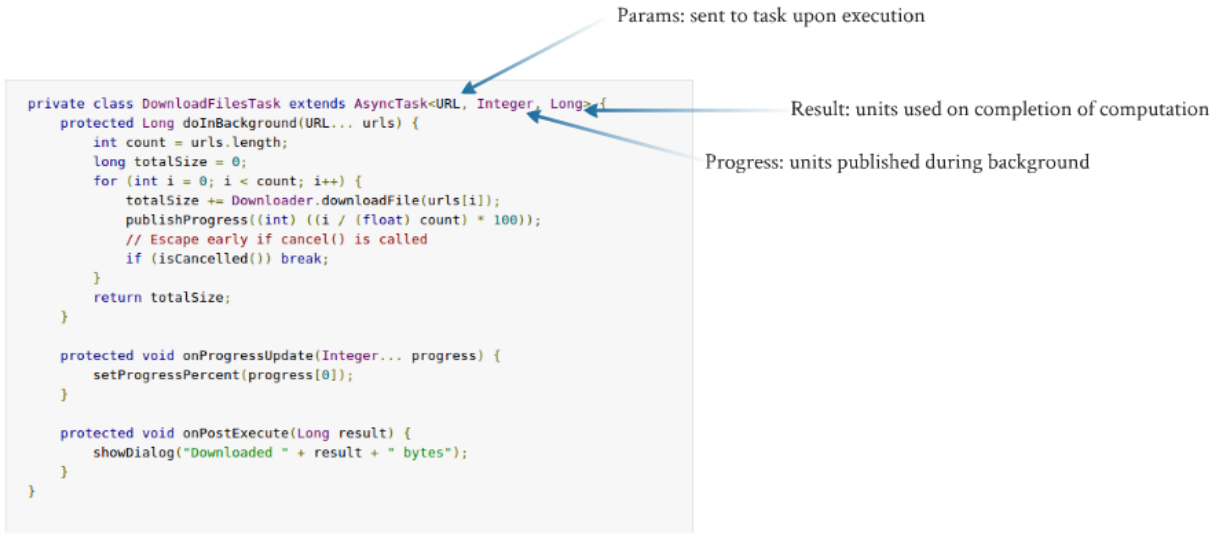


The task is implemented simply using:

- AsyncTask is part of the android.os package.
- It is designed to be a helper class around Thread and Handler.
- AsyncTasks should be used for short operations, just a few seconds.
- For longer operations, it is recommended to use other API's such as handler or thread directly.

- An asynchronous task runs on a background thread and is published on the UI thread.
- This makes it thread safe.
- It has four steps:
  - onPreExecute - invoked on the UI thread before the task is executed.
  - doInBckground- invoked on the background thread immediately after onPreExecute() finishes executing.
  - onProgressUpdate- This method is used to display any form of progress in the user interface while the background computation is still executing.
  - onPostExecute - invoked on the UI thread after the background computation finishes. The result of the background computation is passed to this step as a parameter.

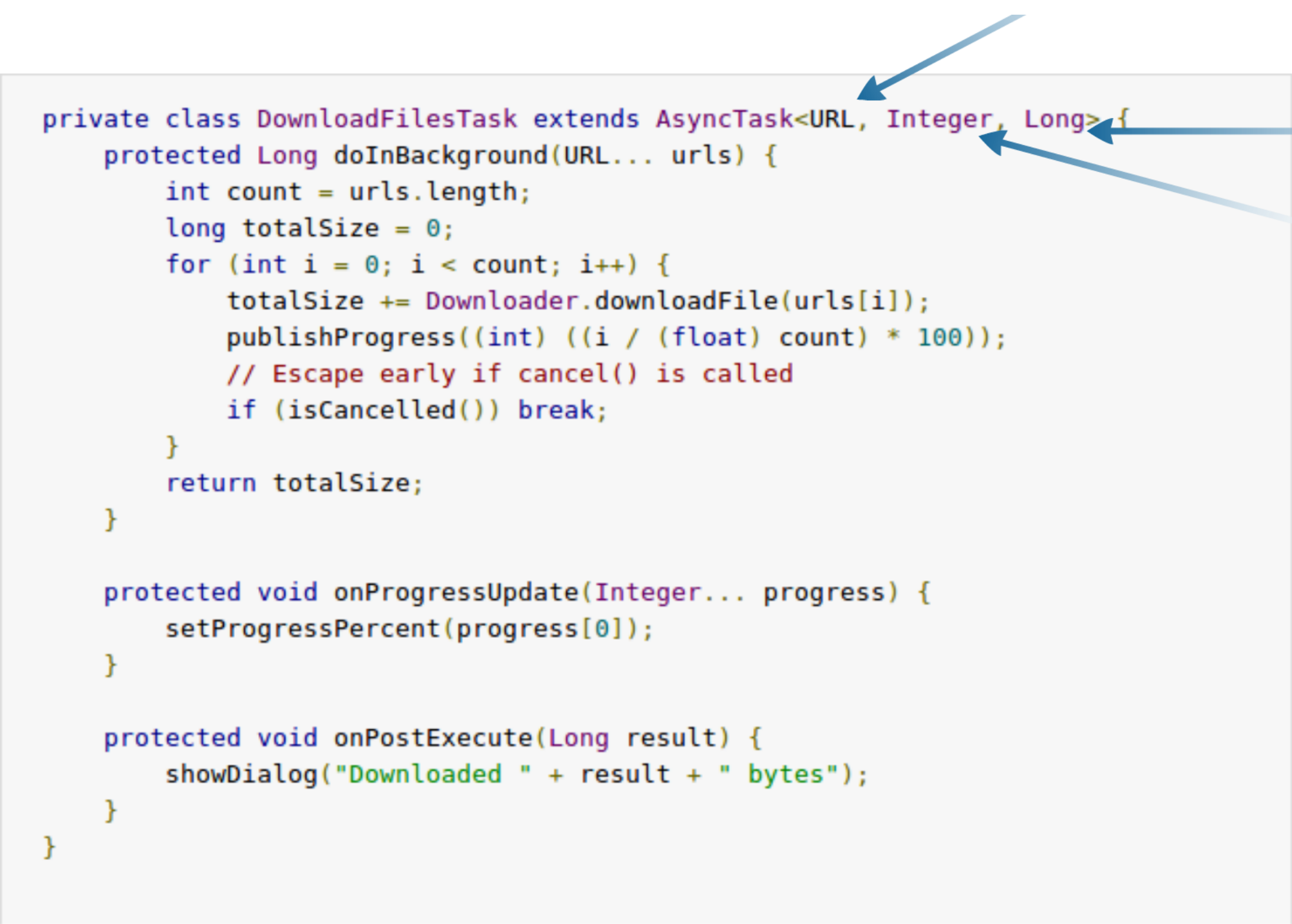
Example of a Async subclass.



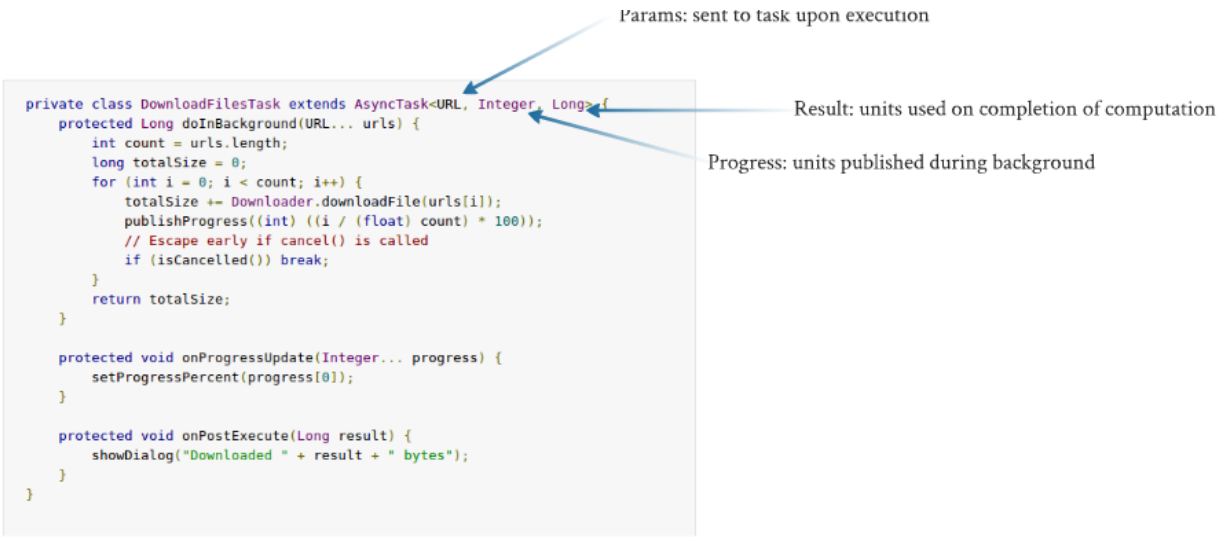

```java
private class DownloadFilesTask extends AsyncTask<URL, Integer, Long> {
    protected Long doInBackground(URL... urls) {
        int count = urls.length;
        long totalSize = 0;
        for (int i = 0; i < count; i++) {
            totalSize += Downloader.downloadFile(urls[i]);
            publishProgress((int) ((i / (float) count) * 100));
            // Escape early if cancel() is called
            if (isCancelled()) break;
        }
        return totalSize;
    }

    protected void onProgressUpdate(Integer... progress) {
        setProgressPercent(progress[0]);
    }

    protected void onPostExecute(Long result) {
        showDialog("Downloaded " + result + " bytes");
    }
}
```

The task is implemented simply using:

```java
new DownloadFilesTask().execute(url1, url2, url3);
```



```
private class DownloadFilesTask extends AsyncTask<URL, Integer, Long> {
    protected Long doInBackground(URL... urls) {
        int count = urls.length;
        long totalSize = 0;
        for (int i = 0; i < count; i++) {
            totalSize += Downloader.downloadFile(urls[i]);
            publishProgress((int) ((i / (float) count) * 100));
            // Escape early if cancel() is called
            if (isCancelled()) break;
        }
        return totalSize;
    }

    protected void onProgressUpdate(Integer... progress) {
        setProgressPercent(progress[0]);
    }

    protected void onPostExecute(Long result) {
        showDialog("Downloaded " + result + " bytes");
    }
}
```




```
private class DownloadFilesTask extends AsyncTask<URL, Integer, Long> {
    protected Long doInBackground(URL... urls) {
        int count = urls.length;
        long totalSize = 0;
        for (int i = 0; i < count; i++) {
            totalSize += Downloader.downloadFile(urls[i]);
            publishProgress((int) ((i / (float) count) * 100));
            // Escape early if cancel() is called
            if (isCancelled()) break;
        }
        return totalSize;
    }

    protected void onProgressUpdate(Integer... progress) {
        setProgressPercent(progress[0]);
    }

    protected void onPostExecute(Long result) {
        showDialog("Downloaded " + result + " bytes");
    }
}
```

The task is implemented simply using:

```
new DownloadFilesTask().execute(url1, url2, url3);
```

mplemented s

```
new DownloadFilesTask().execute(url1, url2, url3);
```

Threads Using Handler

- The Handler class can be used to register to a thread and provides a channel to send data to this thread.
- It registers to the thread it was created in.
- If created in onCreate() of your activity, the Handler object can be used to post data to the main thread.
- Handler class can be an instance of the Message or the Runnable class.

Handler Class

- The Handler class can be used to register to a thread and provides a channel to send data to this thread.
- It registers to the thread it was created in.
- If created in onCreate() of your activity, the Handler object can be used to post data to the main thread.
- Handler class can be an instance of the Message or the Runnable class.

Handler Class

```
public class MyActivity extends Activity {

    [ . . . ]
    // Need handler for callbacks to the UI thread
    final Handler mHandler = new Handler();

    // Create runnable for posting
    final Runnable mUpdateResults = new Runnable() {
        public void run() {
            updateResultsInUi();
        }
    };

    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);

        [ . . . ]
    }

    protected void startLongRunningOperation() {

        // Fire off a thread to do some work that we shouldn't do directly in the UI thread
        Thread t = new Thread() {
            public void run() {
                mResults = doSomethingExpensive();
                mHandler.post(mUpdateResults);
            }
        };
        t.start();
    }

    private void updateResultsInUi() {

        // Back in the UI thread -- update our UI elements based on the data in mResults
        [ . . . ]
    }
}
```

Create a Handler object in your UI thread.

Update the views on the UI thread as needed.

Spawn off worker threads to perform any required expensive operations.

Post results from a worker thread back to the UI thread's handler.

```
public class MyActivity extends Activity {

    [ . . . ]
    // Need handler for callbacks to the UI thread
    final Handler mHandler = new Handler();

    // Create runnable for posting
    final Runnable mUpdateResults = new Runnable() {
        public void run() {
            updateResultsInUi();
        }
    };

    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);

        [ . . . ]
    }

    protected void startLongRunningOperation() {

        // Fire off a thread to do some work that we shouldn't do directly in the UI thread
        Thread t = new Thread() {
            public void run() {
                mResults = doSomethingExpensive();
                mHandler.post(mUpdateResults);
            }
        };
        t.start();
    }

    private void updateResultsInUi() {

        // Back in the UI thread -- update our UI elements based on the data in mResults
        [ . . . ]
    }
}
```

Create a Handler object in your UI thread.

Update the views on the UI thread as needed.

Spawn off worker threads to perform any required expensive operations.

Post results from a worker thread back to the UI thread's handler.

Arrays

Simple approach - construct an array.

```
// allocates memory for 10 integers
    anArray = new int[10];

    // initialize first element
    anArray[0] = 100;
    // initialize second element
    anArray[1] = 200;
```

```
class Card
{
 int suit, rank;

 public Card () {
  this.suit = 0;  this.rank = 0;
 }

 public Card (int suit, int rank) {
  this.suit = suit;  this.rank = rank;
 }
}
```

```
class Student {
  int marks;
}

Student[] studentArray = new Student[7];

studentArray[0] = new Student();

for ( int i=0; i<studentArray.length; i++) {
studentArray[i]=new Student();
}
```

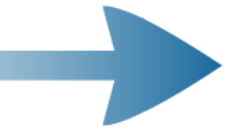
To create an object that represents the 3 of Clubs, we would use the new command:

```
Card threeOfClubs = new Card (0, 3);
```

Simple approach - construct an array.

```
// allocates memory for 10 integers
    anArray = new int[10];

    // initialize first element
    anArray[0] = 100;
    // initialize second element
    anArray[1] = 200;
```

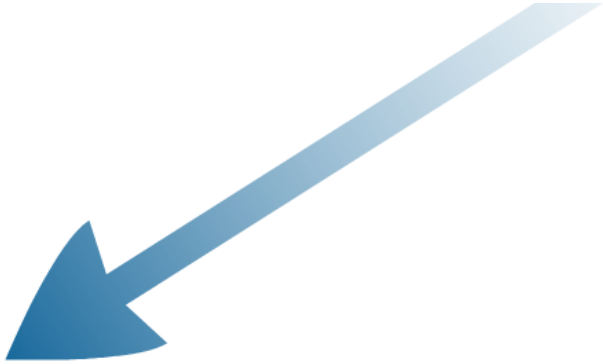
```
class Card
{
  int suit, rank;

  public Card () {
    this.suit = 0;  this.rank = 0;
  }

  public Card (int suit, int rank) {
    this.suit = suit;  this.rank = rank;
  }
}
```



}

To create an object that represents the 3 of Clubs, we would use the new command:

```
Card threeOfClubs = new Card (0, 3);
```



```
class Student {
   int marks;
}

Student[] studentArray = new Student[7];

studentArray[0] = new Student();

for ( int i=0; i<studentArray.length; i++) {
studentArray[i]=new Student();
}
```

Mobile Games Development (IMAT2608/3608)

Dr Jethro Shell
jethros_at_dmu.ac.uk