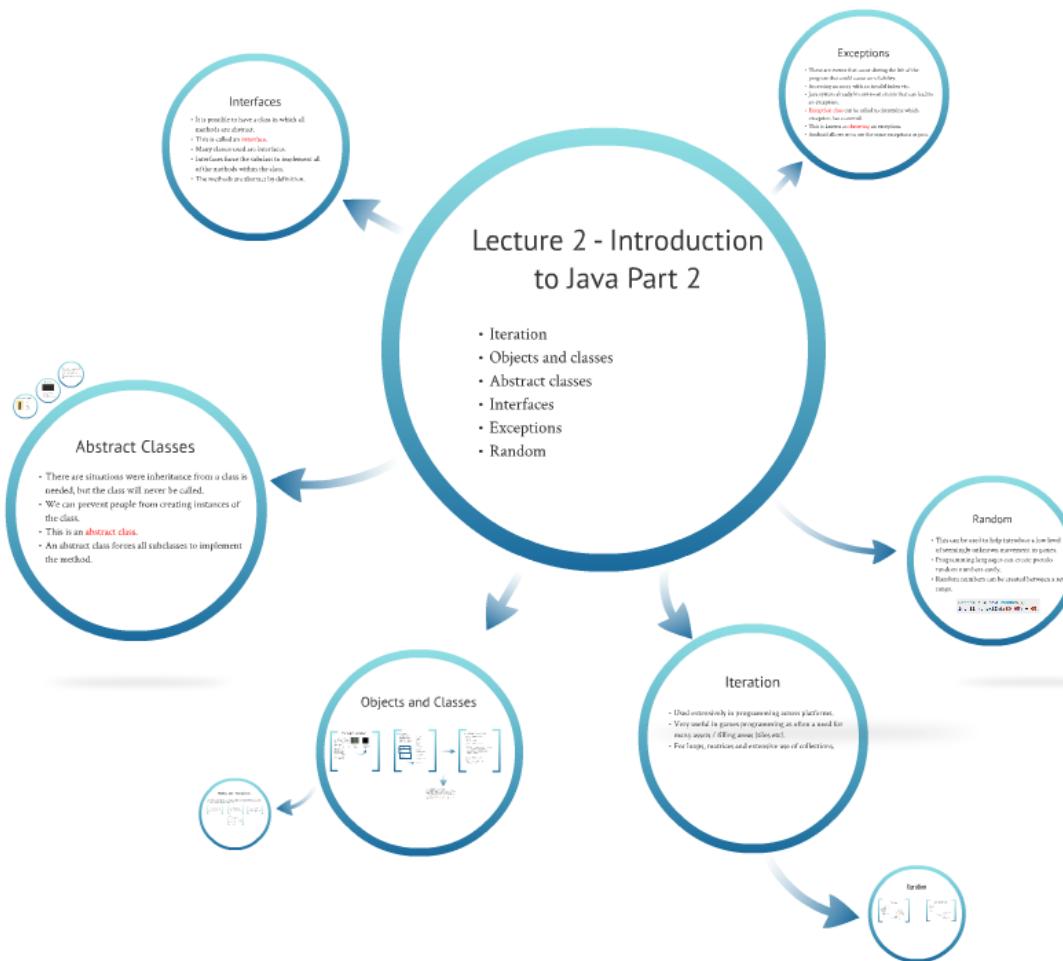


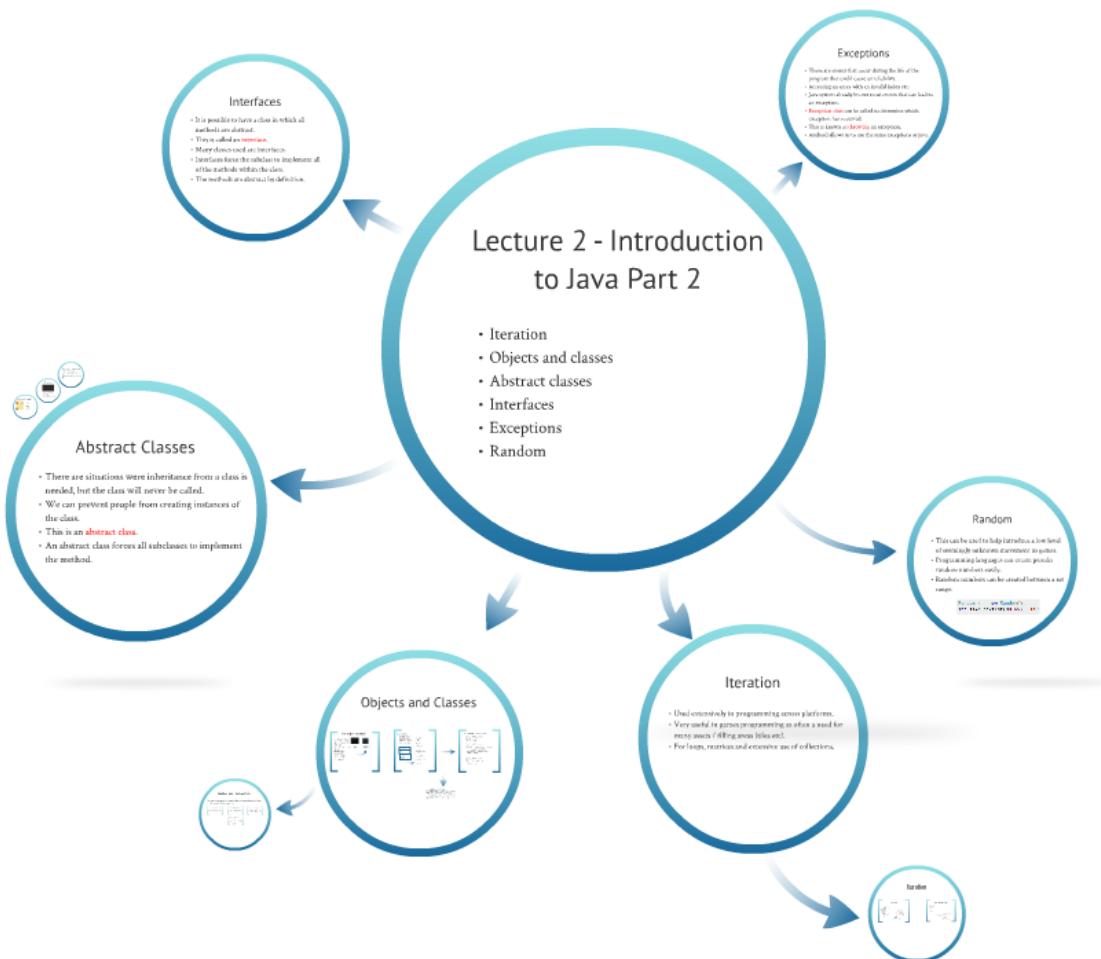
Mobile Games Development (IMAT2608/3608) - Lecture 2

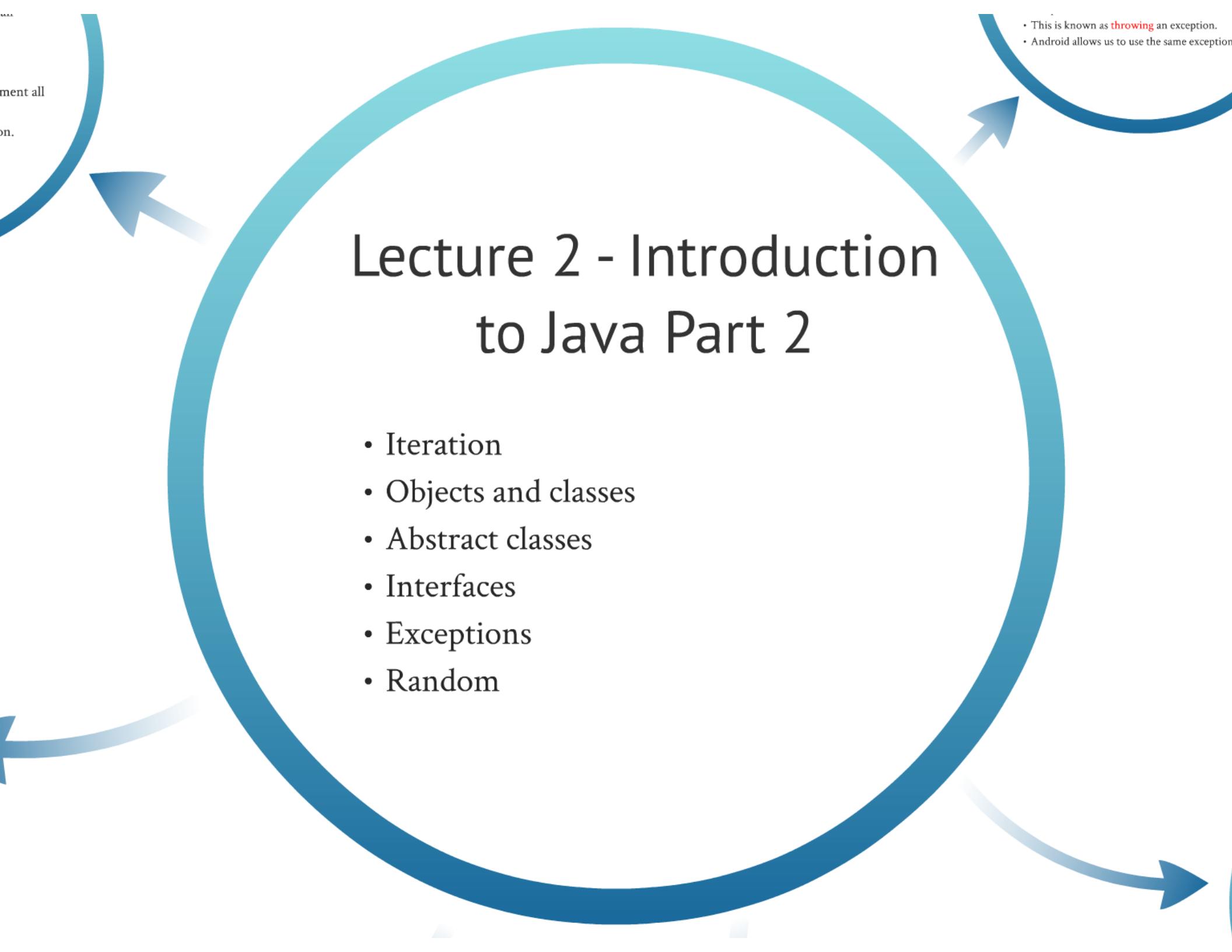
Dr Jethro Shell
jethros_at_dmu.ac.uk



Mobile Games Development (IMAT2608/3608) - Lecture 2

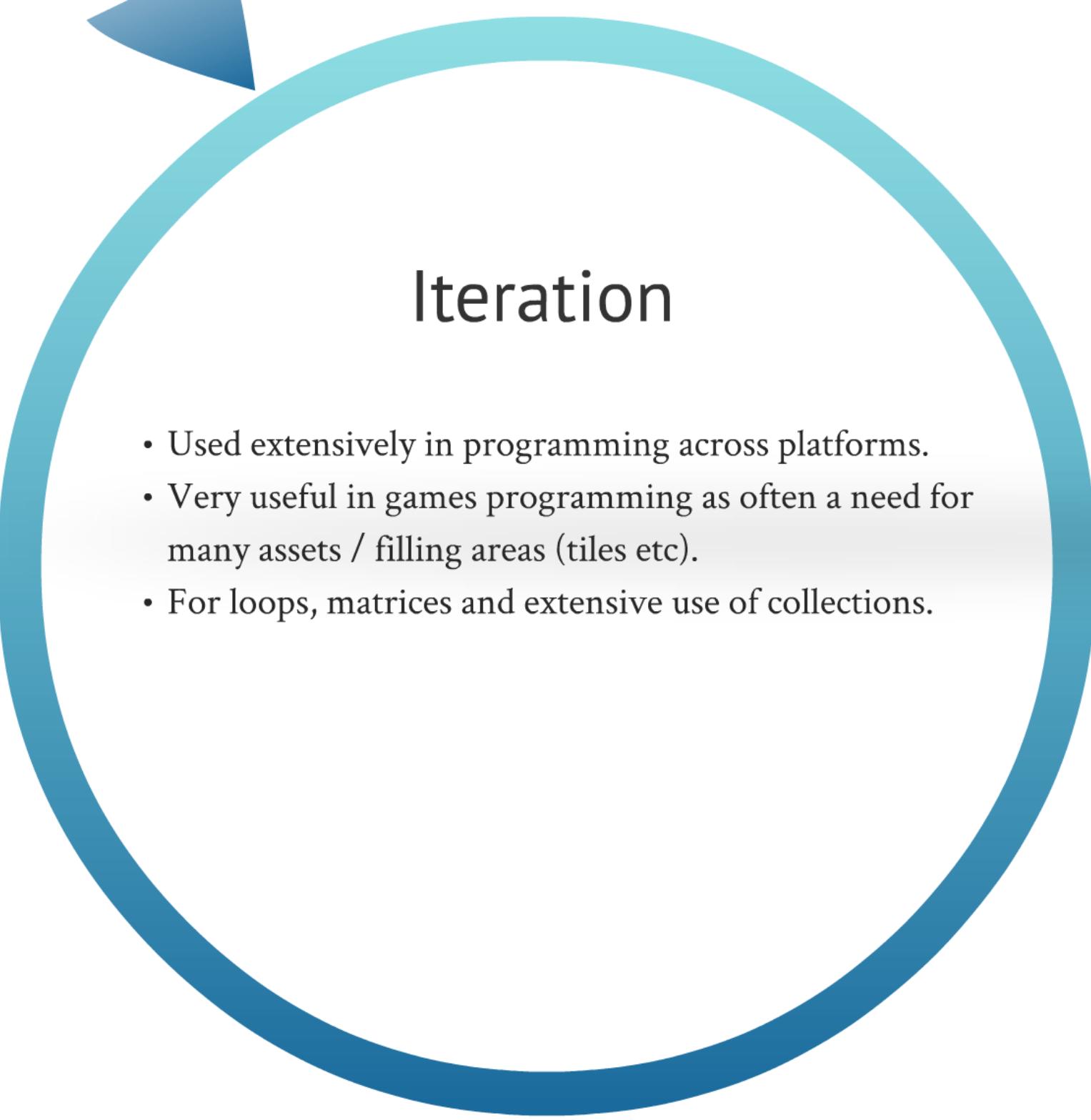
Dr Jethro Shell
jethros_at_dmu.ac.uk



- 
- This is known as **throwing** an exception.
 - Android allows us to use the same exception

Lecture 2 - Introduction to Java Part 2

- Iteration
- Objects and classes
- Abstract classes
- Interfaces
- Exceptions
- Random



Iteration

- Used extensively in programming across platforms.
- Very useful in games programming as often a need for many assets / filling areas (tiles etc).
- For loops, matrices and extensive use of collections.

Iteration

"For" Loop

- The for statement provides a compact way to iterate over a range of values.
- It can be used as a for loop, incrementing the value of the variable in each iteration or counting down from a particular value.
- It can also be used as a do-while loop.
- The general form of the for statement can be:

```
for (initialization; condition; increment){  
    // code block  
}
```

While and Do-While

- While test this condition, continue to execute this block of statements.
- The generic form of the for statement can be:

```
while (condition){  
    // code block  
}
```

↓

```
while (count < 10){  
    Log("Count is " + count);  
    count++;  
}
```

- Do while has a do-while loop.
- The condition is evaluated at least once in opposition to while.
- This takes the form of:

```
int count = 1;  
do {  
    Log("Count is " + count);  
    count++;  
} while (count < 10);
```

In this example, it will run once and start to increment count.

"For" Loop

- The **for** statement provides a compact way to iterate over a range of values.
 - Referred to it as the "for loop" because of the way in which it repeatedly loops until a particular condition is satisfied.
 - The general form of the for statement can be expressed as follows:
- ```
for (initialization; termination;
 increment) {
 statement(s)
}
```
- ```
for(int i=0; i<10; i++){  
    Log.d("Count is: "+i);  
}
```



In this example, the variable **item** holds the current value from the numbers array.

```
int[] numbers = {1,2,3,4,5,6,7,8,9,10};  
for (int item : numbers) {  
    Log.d("Count is: "+item);  
}
```

- The **for** statement provides a compact way to iterate over a range of values.
- Referred to it as the "for loop" because of the way in which it repeatedly loops until a particular condition is satisfied.
- The general form of the for statement can be expressed as follows:

```
for (initialization; termination;  
      increment) {  
    statement(s)  
}
```

```
for(int i=0; i<10; i++){  
    Log.d("Count is: ",""+i);  
}
```

- There is also an "enhanced for" loop.
- This iterates through collections and arrays.
- Similar to iterators in C++.
- This is formed as:

```
int[] numbers = {1,2,3,4,5,6,7,8,9,10};  
for (int item : numbers) {  
    Log.d("Count is: ", ""+item);  
}
```

In this example, the variable **item** holds the current value from the numbers array.



```
static class Foo {
    int mSplat;
}

Foo[] mArray = ...

public void zero() {
    int sum = 0;
    for (int i = 0; i < mArray.length; ++i) {
        sum += mArray[i].mSplat;
    }
}

public void one() {
    int sum = 0;
    Foo[] localArray = mArray;
    int len = localArray.length;

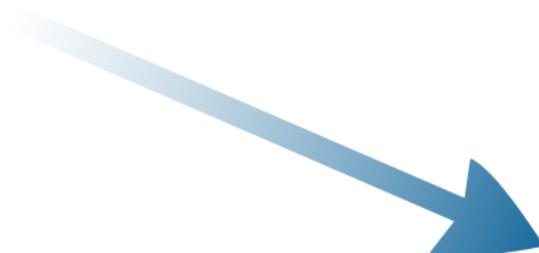
    for (int i = 0; i < len; ++i) {
        sum += localArray[i].mSplat;
    }
}

public void two() {
    int sum = 0;
    for (Foo a : mArray) {
        sum += a.mSplat;
    }
}
```

While and Do-While

- While **true**, this condition continues to execute this block of statements.
- The general form of the for statement can be expressed as follows:

```
while (expression) {  
    statement(s)  
}  
  
while (count < 11) {  
    Log.d("Count is: ",""+count;  
    count++;  
}
```



- Java also has a do-while loop.
- This is executed at least once as opposed to while.
- This takes the form of:

```
int count = 1;  
do {  
    Log.d("Count is: ",""+count);  
    count++;  
} while (count < 11);
```

In this example, it will run once and start to increment count.

- While **true**, this condition continues to execute this block of statements.
- The general form of the for statement can be expressed as follows:

```
while (expression) {
    statement(s)
}
```



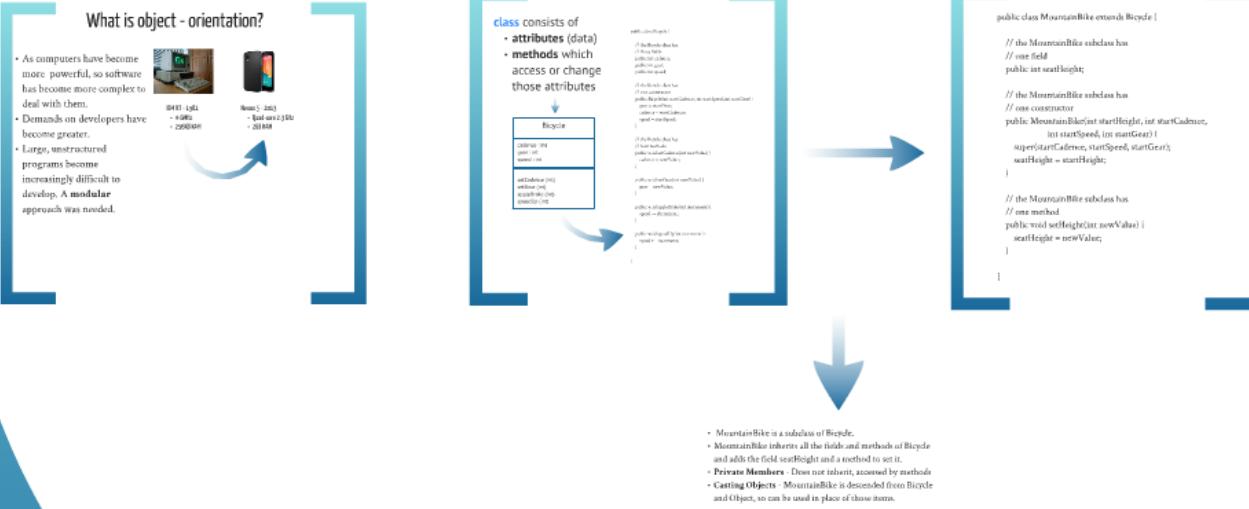
```
while (count < 11) {
    Log.d("Count is: ", ""+count;
    count++;
}
```

- Java also has a do-while loop.
- This is executed at least once as opposed to while.
- This takes the form of:

```
int count = 1;  
    do {  
        Log.d("Count is: ", ""+count);  
        count++;  
    } while (count < 11);
```

In this example, it will run once and start to increment count.

Objects and Classes



What is object - orientation?

- As computers have become more powerful, so software has become more complex to deal with them.
- Demands on developers have become greater.
- Large, unstructured programs become increasingly difficult to develop. A **modular** approach was needed.



IBM XT - 1981

- 4-6MHz
- 256KB RAM



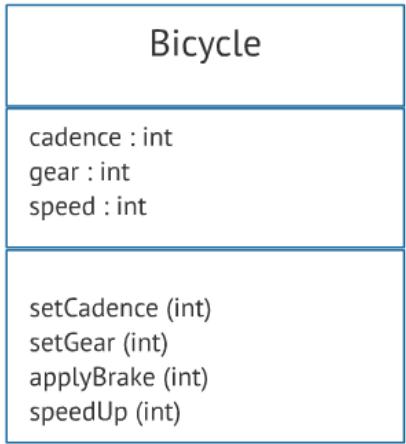
Nexus 5 - 2013

- Quad-core 2.3 GHz
- 2GB RAM



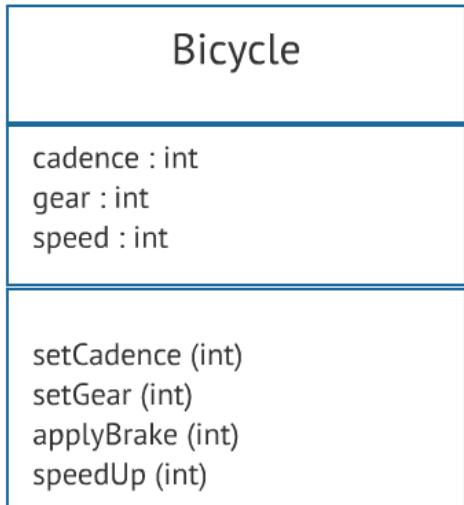
class consists of

- **attributes** (data)
- **methods** which access or change those attributes



```
public class Bicycle {  
  
    // the Bicycle class has  
    // three fields  
    public int cadence;  
    public int gear;  
    public int speed;  
  
    // the Bicycle class has  
    // one constructor  
    public Bicycle(int startCadence, int startSpeed, int startGear) {  
        gear = startGear;  
        cadence = startCadence;  
        speed = startSpeed;  
    }  
  
    // the Bicycle class has  
    // four methods  
    public void setCadence(int newValue) {  
        cadence = newValue;  
    }  
  
    public void setGear(int newValue) {  
        gear = newValue;  
    }  
  
    public void applyBrake(int decrement) {  
        speed -= decrement;  
    }  
  
    public void speedUp(int increment) {  
        speed += increment;  
    }  
}
```

- class** consists of
- **attributes (data)**
 - **methods which access or change those attributes**



```
public class Bicycle {  
  
    // the Bicycle class has  
    // three fields  
    public int cadence;  
    public int gear;  
    public int speed;  
  
    // the Bicycle class has  
    // one constructor  
    public Bicycle(int startCadence, int startSpeed, int startGear) {  
        gear = startGear;  
        cadence = startCadence;  
        speed = startSpeed;  
    }  
  
    // the Bicycle class has  
    // four methods  
    public void setCadence(int newValue) {  
        cadence = newValue;  
    }  
  
    public void setGear(int newValue) {  
        gear = newValue;  
    }  
  
    public void applyBrake(int decrement) {  
        speed -= decrement;  
    }  
  
    public void speedUp(int increment) {  
        speed += increment;  
    }  
}
```



```
public class MountainBike extends Bicycle {  
  
    // the MountainBike subclass has  
    // one field  
    public int seatHeight;  
  
    // the MountainBike subclass has  
    // one constructor  
    public MountainBike(int startHeight, int startCadence,  
                        int startSpeed, int startGear) {  
        super(startCadence, startSpeed, startGear);  
        seatHeight = startHeight;  
    }  
  
    // the MountainBike subclass has  
    // one method  
    public void setHeight(int newValue) {  
        seatHeight = newValue;  
    }  
}
```



- MountainBike is a subclass of Bicycle.
- MountainBike inherits all the fields and methods of Bicycle and adds the field seatHeight and a method to set it.
- **Private Members** - Does not inherit, accessed by methods
- **Casting Objects** - MountainBike is descended from Bicycle and Object, so can be used in place of those items.

Modifiers: Public, Private and Static

Java provides a number of access modifiers to set access levels for classes, variables, methods and constructors.

Default Access Modifier - No keyword:

A variable or method declared without any access control modifier is available to any other class in the same package

```
String nokeyword = "Helloiverse";
```

Private Access Modifier - private:

Methods, Variables and Constructors that are declared private can only be accessed within the declared class itself.



Public Access Modifier - public:

A class, method, constructor, interface etc declared public can be accessed from any other class. Therefore fields, methods, blocks declared inside a public class can be accessed from any class belonging to the Java Universe.

Static Keyword:

The static keyword is used to create variables that will not addressed by one instance created in the class. They are shared by all instances and are visible to the whole number of instances of the class.

Static variables are also known as class variables. Local variables require no declared static.

Static Methods:

The static keyword is used to create methods that will not addressed by one instance created in the class.

Static methods can be accessed in the class they are defined in. Static methods do not have access to the fields of any other class. Static methods can be accessed from any other class.



Default Access Modifier - No keyword:

A variable or method declared without any access control modifier is available to any other class in the same package

```
String noKeyword = "NoKeyword";|
```

Private Access Modifier - private:

Methods, Variables and Constructors that are declared private can only be accessed within the declared class itself.

```
public class EnemySpaceCraft extends SpaceCraft{  
    private int score;  
  
    @Override  
    void draw() {  
        /*Draw the ship*/  
    }  
  
    public int getScore(){  
        return score;  
    }  
}
```

```
public class EnemySpaceCraft extends SpaceCraft{  
  
    private int score;  
  
    @Override  
    void draw() {  
        /**Draw the ship**/  
    }  
  
    public int getScore(){  
        return score;  
    }  
}
```

Public Access Modifier - public:

A class, method, constructor, interface etc declared public can be accessed from any other class. Therefore fields, methods, blocks declared inside a public class can be accessed from any class belonging to the Java Universe.

Static Variables: The static key word is used to create variables that will exist independently of any instances created for the class. Only one copy of the static variable exists regardless of the number of instances of the class.

Static variables are also known as class variables. Local variables cannot be declared static.

Static Methods: The static key word is used to create methods that will exist independently of any instances created for the class.

Static methods do not use any instance variables of any object of the class they are defined in. Static methods take all the data from parameters and compute something from those parameters, with no reference to variables.

```
public class EnemySpaceCraft extends SpaceCraft{  
    private static int score = 0;  
  
    protected static int getScore() {  
        return score;  
    }  
  
    public static void addScore() {  
        score++;  
    }  
  
    @Override  
    void draw() {  
        /*Draw the ship*/  
    }  
}
```

```
public class EnemySpaceCraft extends SpaceCraft{  
    private static int score = 0;  
  
    protected static int getScore() {  
        return score;  
    }  
  
    public static void addScore() {  
        score++;  
    }  
  
    @Override  
    void draw() {  
        /**Draw the ship**/  
    }  
}
```



Abstract Classes

- There are situations where inheritance from a class is needed, but the class will never be called.
- We can prevent people from creating instances of the class.
- This is an **abstract class**.
- An abstract class forces all subclasses to implement the method.



Retro Space Commanders



- Location
- Thrust
- Size
- Velocity
- Position
- Weapons

Breakout



- Need a class for the blocks.
- Different types of blocks.
- Different attributes.
- Different blocks, similar functions.

How is it constructed?

- Uses the abstract class definition.
- Primitives can be defined within the class.
- Unlike an interface, not all methods need to be used.
- Defining methods as abstract can force those to be used.

```
abstract class Spaceship {
    private int width;
    private int height;
    private int velocity;
    private int health;

    public Spaceship(int widthIn, int heightIn, int velocityIn, int healthIn){
        width = widthIn;
        height = heightIn;
        velocity = velocityIn;
        health = healthIn;
    }

    public int getWidth(){
        return width;
    }

    public int getHeight(){
        return height;
    }

    public int getVelocity(){
        return velocity;
    }

    public int getHealth(){
        return health;
    }

    abstract public int getSize(); //this forces all subclasses of the class
                                //to have this method
}
```

```
public class ArmySpaceship extends Spaceship{
    public void f()
    {
        System.out.println("I am a spaceship");
    }
}
```



```
public class EnemySpaceCraft extends SpaceCraft{  
  
    @Override  
    void draw() {  
        // TODO Auto-generated method stub  
  
    }  
  
}
```

Interfaces

- It is possible to have a class in which all methods are abstract.
- This is called an **interface**.
- Many classes used are interfaces.
- Interfaces force the subclass to implement all of the methods within the class.
- The methods are abstract by definition.



Interface Class

- In developing a game we may want a framework.
- When a class from the framework is implemented, all methods are used by the subclass.



Implement Alien class,
forced to have:

- run()
- fight()
- getStrength()
- setWeapons()

```
interface Weapons {  
    public int weaponType (String t); //No body is defined  
}
```

- set

```
interface Weapons {  
    public int weaponType (String t); //No body is defined  
}
```

```
public class EnemySpacecraft extends Spacecraft implements Weapons{
    private boolean cloakDevice;

    public EnemySpacecraft (int widthIn, int heightIn,
                           int velocityIn, int healthIn, boolean cloak){
        super(widthIn,heightIn,velocityIn,healthIn);
        cloakDevice = cloak;
    }

    public boolean getcloakStatus(){
        return cloakDevice;
    }

    public int getSize(){
        return getWidth() * getHeight();
    }

    public int weaponType(String t){
        int powerLevel;
        if (t == "HEAVY"){
            powerLevel = 10;
        }
        else if (t == "MEDIUM"){
            powerLevel = 5;
        }
        else {
            powerLevel = 0;
        }

        return powerLevel;
    }
}
```

Exceptions

- These are events that occur during the life of the program that could cause unreliability.
- Accessing an array with an invalid index etc.
- Java system already knows most events that can lead to an exception.
- **Exception class** can be called to determine which exception has occurred.
- This is known as **throwing** an exception.
- Android allows us to use the same exceptions as java.

```
@Override
public void onDraw(Canvas canvas) {
    try{
        if (blocks.size() > 0){
            for (int i = 0; i < blocks.size(); i++){
                blocks.get(i).drawRect(paint, canvas);
            }
        }
    }catch(Exception e){
        Log.e("Error", "Out of scope");
    }
}
```

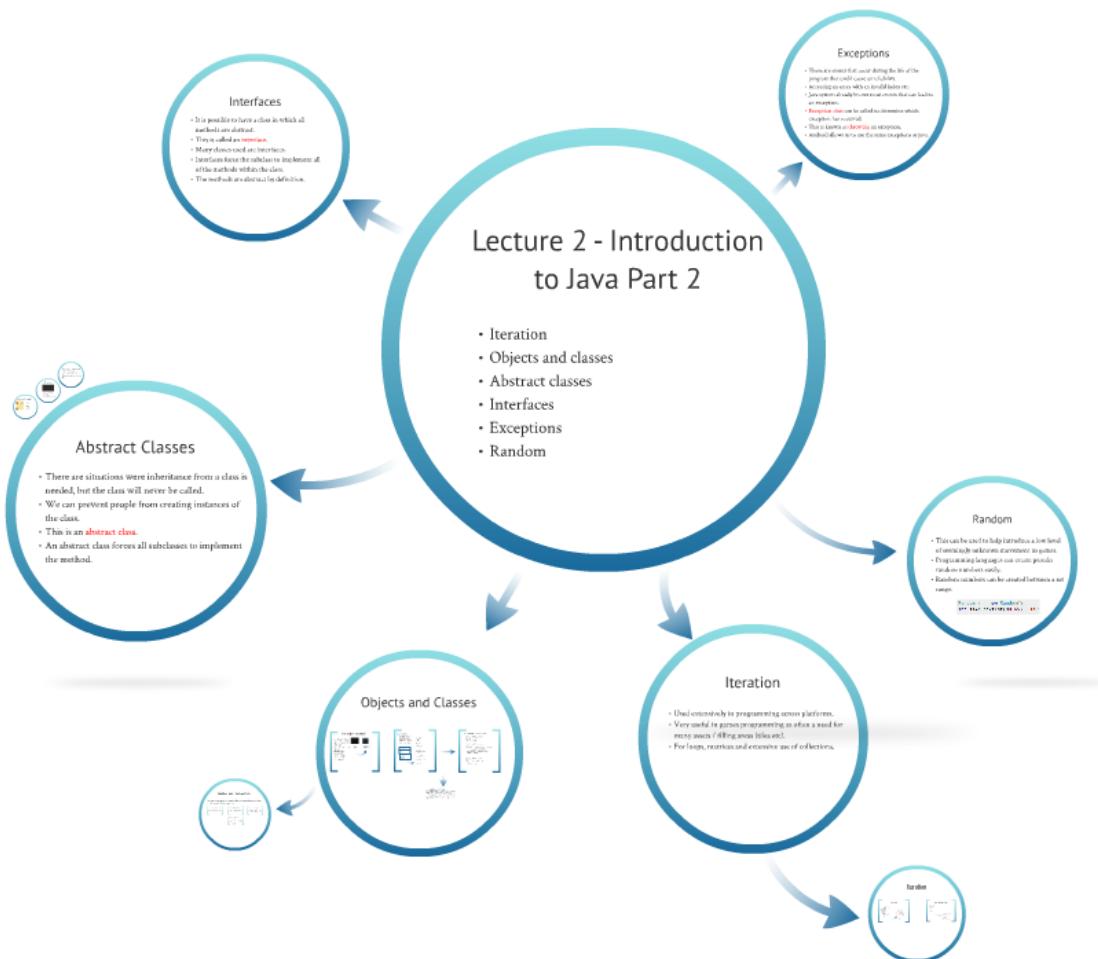
Random

- This can be used to help introduce a low level of seemingly unknown movement to games.
- Programming languages can create pseudo random numbers easily.
- Random numbers can be created between a set range.

```
Random r = new Random();
int i1=r.nextInt(80-65) + 65;
```

Mobile Games Development (IMAT2608/3608) - Lecture 2

Dr Jethro Shell
jethros_at_dmu.ac.uk



- As computers have become more powerful, so software has become more complex to deal with them.
- Demands on developers have become greater.
- Large, unstructured programs become increasingly difficult to develop. A **modular** approach was needed.



IBM XT - 1981

- 4-6MHz
- 256KB RAM