Object Oriented Programming (OOP)

- What is OOP?
- OOP is the idea of modelling the world by "**Objects**" .
- Humans, Cars, Bicycles, and others are examples of objects.
- What do they have in common?
  - **Attributes** and **Behaviours**.

Object Oriented Programming (OOP)

- They all have a different set of **attributes** and **behaviours** to distinguish them as different classes of objects.
- For example: A Human has a Heart which pumps blood. A Car will have an Engine which provides many functional features. A Bicycle will have Gears that can be changed.
- The point is to define them based on attributes and behaviour. Lets take the Bicycle as an example.

Object Oriented Programming (OOP)

- Attributes of a Bicycle may be:
  - Number of Front Gears
  - Number of Back Gears
  - Current Front Gear
  - Current Back Gear
  - Etc....

Object Oriented Programming (OOP)

- Behaviours may include:
  - Change front gear Up
  - Change front gear Down
  - Change back gear Up
  - Change back gear Down
  - Etc...

Object Oriented Programming (OOP)

- Now lets put the **attributes** into a Java Class format: (Ignore the word "**public**" for now)

```
public class Bicycle {

    int numFrontGears;
    int numBackGears;
    int currentFrontGear;
    int currentBackGear;

}
```

Object Oriented Programming (OOP)

- This is now with **methods** to adjust the class attributes according to the **behaviour** 2 pages ago.
- Note: No bound checking is done. These methods are declared between the curly braces of “public class Bicycle”.

```
void incrementFrontGear(){
    ++currentFrontGear;
}

void decrementFrontGear(){
    --currentFrontGear;
}

void incrementBackGear(){
    ++currentBackGear;
}

void decrementBackGear(){
    --currentBackGear;
}
```

Object Oriented Programming (OOP)

- Note that this class is simply a “blueprint” (Also known as a **Class)** for a bicycle with Gears and does not actually exist yet.
- For people who have little experience with programming it may be tempting **not** to create these blueprints and simply put everything in one huge method. This is wrong and should be avoided!

Object Oriented Programming (OOP)

- You may be wondering about the word "**public**" used earlier.
- This word, along with its relatives (called **Access Modifiers**) are at the core of OOP. Without it many of the key concepts in OOP would not exist.

Access Modifiers

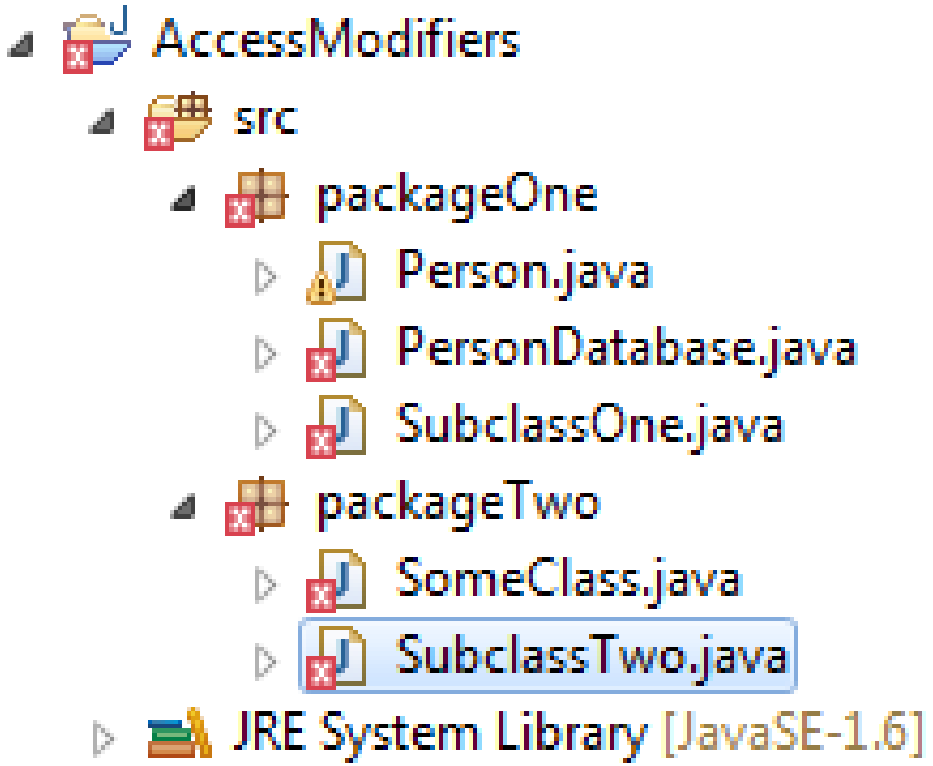
- The word **public** is an **access modifier**. In addition the words **private** and **protected** are also access modifiers.
- But what is an **access modifier**? They are words you add to the declaration of an **attribute** or **method** to decide who can get access to them.
- By “who” we mean a different area in the source code which does not belong to that class blueprint.

“**public**, **protected**, **private**, **no-modifier**” keywords

- **Public** – **any** code can access it.
- **Protected** – only code within the **same package** or **subclasses** have access.
- **Private** – **only** that class it is contained in can access it.
- **No-modifier** – accessible only by the current class and classes in the same package.

“**public**, **protected**, **private**, **no-modifier**” keywords

- An example will clear things up:




```
package packageOne;

public class Person {

    public int height;
    protected int age;
    int someAttribute;
    private String name;

}
```

“public, protected, private, no-modifier” keywords

```java
package packageOne;

public class SubclassOne extends Person {

    public SubclassOne(){
        this.height = 0;
        this.age = 0;
        this.someAttribute = 0;
        this.name = 0;
    }
}
```

```java
package packageTwo;

import packageOne.Person;

public class SubclassTwo extends Person {

    public SubclassTwo(){
        this.height = 0;
        this.age = 0;
        this.someAttribute = 0;
        this.name = 0;
    }
}
```

```java
package packageTwo;
import packageOne.Person;

public class SomeClass {
    Person person;

    public SomeClass(){
        this.person = new Person();
        this.person.height = 0;
        this.person.age = 0;
        this.person.someAttribute = 0;
        this.person.name = 0;
    }

}
```

```java
package packageOne;

public class PersonDatabase {

    private Person person;

    public PersonDatabase(){
        this.person = new Person();
        this.person.height = 0;
        this.person.age = 0;
        this.person.someAttribute = 0;
        this.person.name = 0;
    }

}
```

“**this**” keyword

- The word **this** allows you to refer to an objects **attributes** or **methods** directly.
- For example, you may have method parameters with the same names and want to make sure the object attributes are set. (Note: x = x; does nothing.)

```
public class Point2D {
    private float x;
    private float y;

    public Point2D(float x, float y){
        this.x = x;
        this.y = y;
    }

    public void setX(float x){
        this.x = x;
    }

    public void setY(float y){
        this.y = y;
    }

}
```

“**this**” keyword

- In addition this allows you to call constructor methods within a constructor method.
- However it does not allow you to call constructor methods outside of constructor methods.
- The code on the next page shows an example of this. There exists an error in “someMethod” and no error in the Constructor.

“**this**” keyword

```
public class Point2D {

    float x;
    float y;

    public Point2D(){
        this(5.0f, 10.0f);
    }

    public Point2D(float x, float y){
        this.x = x;
        this.y = y;
    }

    public void someMethod(){
        this(5.0f,10.0f);
    }

}
```

Constructor Methods

- **Constructor Methods** are methods that **initialize** an objects **Class Attributes**.
- Imagine a Point class with two class attributes of **x** and **y**. By default they are both zero. A Constructor will initialize the attributes based on how you implement it.
- Constructors can be **overloaded** allowing for different behaviour.

Constructor Methods (Example)

- Point2D is the name of this class.
- Point2D is also the name of the constructors.
- Note how there are two **constructors** with the name Point2D (**Overloading**).

```
public class Point2D {

    float x;
    float y;

    public Point2D(){
        this.x = 5.0f;
        this.y = 10.0f;
    }

    public Point2D(float x, float y){
        this.x = x;
        this.y = y;
    }

}
```

“**new**” keyword and **Constructors**

- The **new** keyword can only be used with Constructor methods.
- The following slides will explain the **new** operator.

“**new**” keyword

- The **new** keyword is used to dynamically allocate memory for an object. This process is known as **instantiation**.
- It tells Java to create some space for your object on the **heap**. However unlike other languages you need not worry about cleaning up memory.
- Conveniently Java's Garbage Collector does the clean up for you.

“**new**” keyword (Continued)

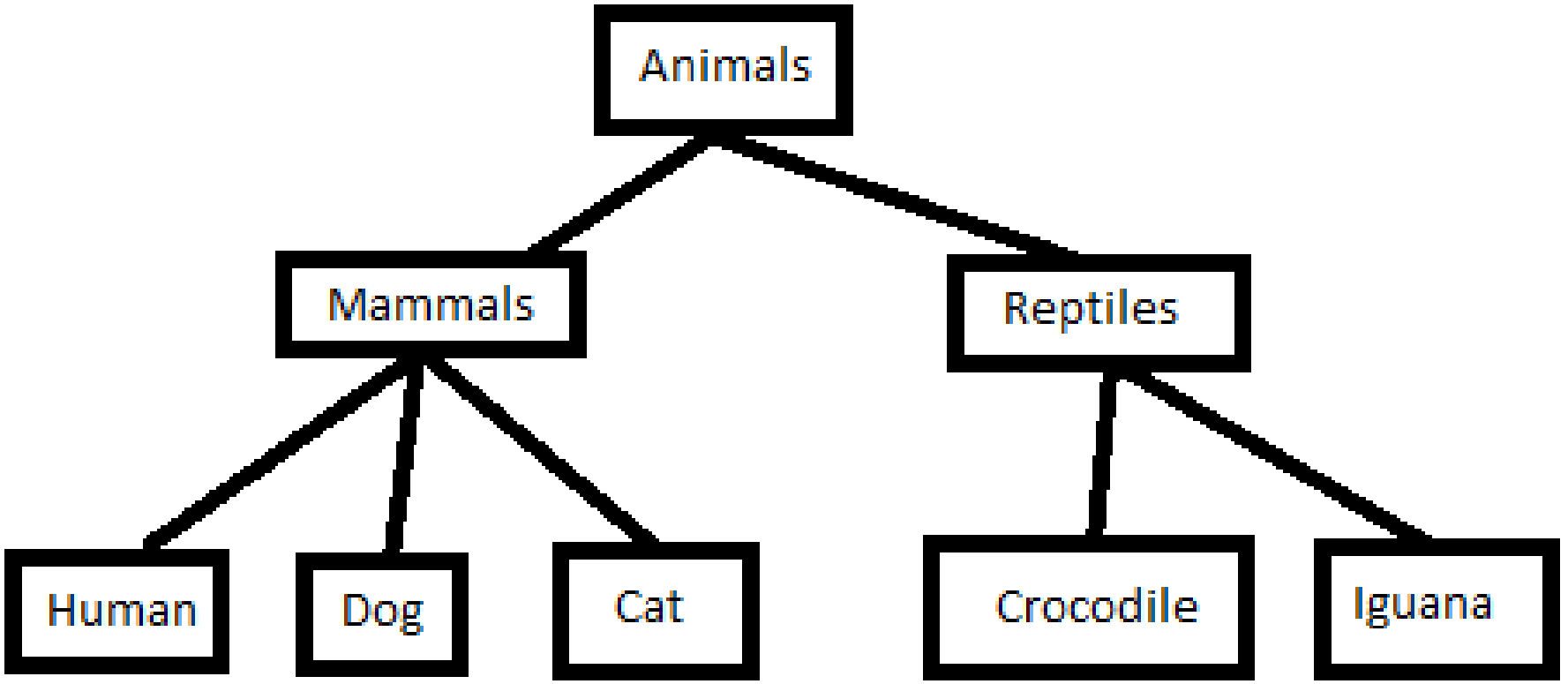
```
public static void main(String [] args){

    Point2D point = new Point2D(10,10);

}
```

- Programmer: “Hey Java, can you give me some space for my point object.”
- Java: “Yo Programmer, here is space for your Point2D object. I also set both x and y to 10 as you requested. Dun worry, I'll clean up the mess.”

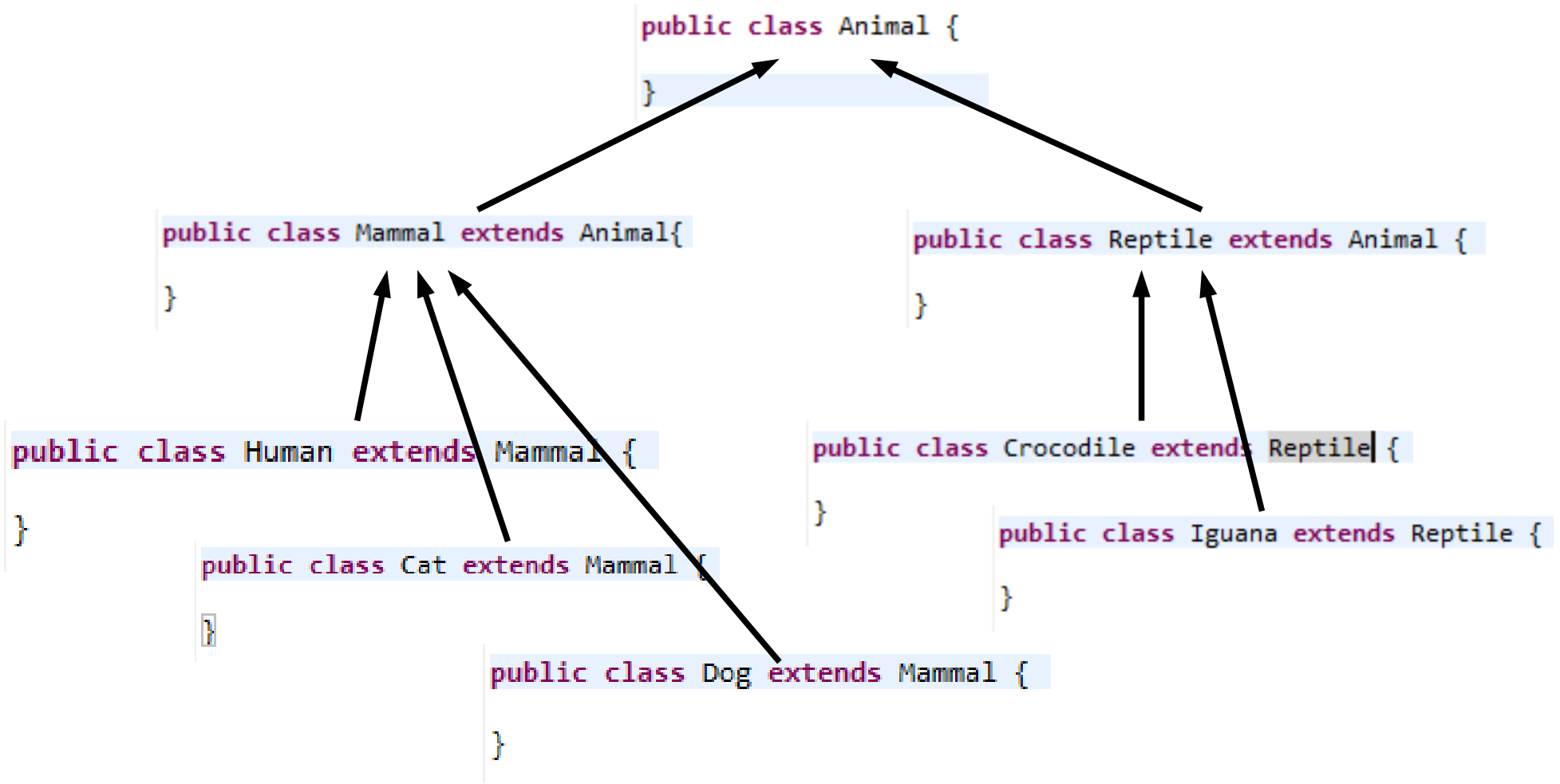
Inheritance

- This is the idea of hierarchical categorization.
- For example, we can think of Humans, Dogs, and Cats as Mammals. In addition we can think of Crocodiles and Iguana's as Reptiles. Furthermore Reptiles and Mammals can be thought of as Animals.
- This is shown graphically next:

Inheritance



Inheritance



Inheritance

- **extends** and **super** are two keywords used in inheritance relationships:
  - **extends** – creates the link b/w parent and subclass.
  - **super** – in a subclass you can call parent methods.

Inheritance

- Example: Lets make Mammal and Dog more functional.

  The purpose of this example is to show how **super** is used.

```
public class Mammal {

    public Mammal(){
        System.out.println("Mammal Constructor");
    }

    public void makeNoise(){
        System.out.println("Mammal Noise!");
    }

}
```

```
public class Dog extends Mammal{

    public Dog(){
        super();
        System.out.println("Dog Constructor");
    }

    public void makeNoise(){
        super.makeNoise();
        System.out.println("Dog Noise!");
    }

}
```

Inheritance

- Here we run the program and show the output of these two lines of instruction.

```
public class Application {

    public static void main(String [] args){
        Mammal dog = new Dog();
        dog.makeNoise();
    }
}
```

```
Mammal Constructor
Dog Constructor
Mammal Noise!
Dog Noise!
```

Polymorphism

- Polymorphism is based on Inheritance.
- The idea is to give an object dynamic behaviour.
- Lets continue our previous example and focus on the Reptile class along with its subclasses.

Polymorphism

```java
public class Reptile extends Animal {

    public void talk(){
        System.out.println("I am a Reptile!");
    }

}
```

```java
public class Crocodile extends Reptile {

    public void talk(){
        System.out.println("I am a Crocodile!");
    }

}
```

```java
public class Iguana extends Reptile {

    public void talk(){
        System.out.println("I am an Iguana!");
    }

}
```

```java
public class Application {
    public static void main(String [] args){
        Reptile reptile = new Iguana();
        reptile.talk();
        reptile = new Reptile();
        reptile.talk();
        reptile = new Crocodile();
        reptile.talk();
    }
}
```

- Here is the Output!

```
I am an Iguana!
I am a Reptile!
I am a Crocodile!
```

Encapsulation

- “Information Hiding”
- The idea is to hide member variables of classes in order to maintain the integrity of the object.
- For example, imagine a Vector object that allows you to change the variable keeping track of the number of elements. If so then you can wind up with a corrupt Vector object.
- Thus it is important to hide certain variables.

Encapsulation

- The use of **access modifiers** allows **encapsulation** to be possible.
- **private** and **protected** allows you to hide data.
- Occasionally you want to change the values of these “hidden” variables. This is done through public methods in the class.
- Depending on the design you may or may not want to create methods to modify these member variables.

Encapsulation

- Example: Getting and Setting an Angle.

```
public class Angle {

    private float angle; //degrees b/w 0 and 360.

    public void setAngle(float angle){
        this.angle = angle;
        while(this.angle < 0.0f || this.angle > 360.0f){
            if(this.angle < 0.0f)
                this.angle += 360.0f;
            else
                this.angle -= 360.0f;
        }
    }

    public float getAngle(){
        return this.angle;
    }

}
```

Encapsulation

```
public class RunAngle {
    public static void main(String [] args){
        Angle angle = new Angle();
        angle.setAngle(800.0f);
        System.out.println("800.0f to: " + angle.getAngle());
        angle.setAngle(-240.0f);
        System.out.println("-240.0f to: " + angle.getAngle());
        angle.setAngle(50.0f);
        System.out.println("50.0f to: " + angle.getAngle());
    }
}
```

```
<terminated> RunAngle [
800.0f to: 80.0
-240.0f to: 120.0
50.0f to: 50.0
```

Abstract Classes

- Remember our hierarchical categorization of Animals.
- We can take the top class and assume that it can be **abstract**. This is because it is the most abstract description of any given instance of an animal.
- Humans, Dogs, and Cats can be expressed as having a similar set of attributes and behaviours. They all can “communicate”.

Abstract Classes

- However they each differ slightly on how they communicate. A dog barks, a cat meows, and a human speaks.
- This important thing is their communication mechanisms can be described abstractly without knowing how a concrete instance behaves.
- This is the essence of abstract classes.

Abstract Classes

- Described abstractly can be thought of as a method declaration without actually implementing it.
- Ex: public abstract void communicate();
  - An abstract method as above does not contain a method body and is ended with a semicolon.
- A subclass like the Dog would then **have** to implement this communicate method.

Abstract Classes

- Abstract classes can have normal methods with implemented bodies.
- Additionally they can have member variables.
- However since they have the potential to have abstract methods they are not allowed to be instantiated:

Abstract Classes

- Ex: Let “A” be an abstract class with abstract void method m:
  - A a = new A();
  - a.m();
- a.m() would have nothing to execute because there is no definition. This is why you cannot instantiate it.
- However, you can call abstract methods within an abstract class's non-abstract method. This is important for certain **Design Patterns**.

Abstract Classes

- Lets get back to the Dog, Human, Cat and Animal example.
- Animal is an abstract class now with an abstract method called communicate. Let's assume Human extends Animal directly. Then Human must implement the communicate method.
- Then we can instantiate a Human using an Animal as the reference variable:
  - Animal animal = new Human();
  - animal.communicate();

Abstract Classes

```
public abstract class Animal {

    public abstract void communicate();

}
```

```
public class Human extends Animal {

    public void communicate() {
        System.out.println("I can speak!");
    }

}
```

Interfaces

- Interfaces are similar to abstract classes in that they can have methods that do not have bodies.
- However when it comes to member variables they differ greatly. If you declare a variable in an interface it is automatically assigned **public, static**, and **final** modifiers. This does not happen in abstract classes.

Interfaces

- An abstract class is **extended** whereas an interface is **implemented**.
- Methods with bodies in interfaces are not allowed. In addition you cannot call a method private.
- The purpose of an interface is to define a set of behaviours that must be followed exactly. This allows a class to implement methods without the user of the interface knowing how it was implemented.

Interfaces

- Therefore the user of the interface knows that no matter how it is implemented the methods will return a certain type of data given its input.
- This allows for dynamic behaviour in the system. Classes that implement interfaces may appear to have nothing in common.
- A video game is a good example.

Interfaces

- Imagine a simple game that makes a character move up, down, left, and right on the screen.
- We can call the interface Movement with four methods up(), down(), left(), and right().
- There would be a Character Class that implements Movement. Then a Character instance would be passed to the KeyboardController or PS3Controller classes.

Interfaces

- The KeyboardController and PS3Controller classes do not care how the Character class implements the Movement interface.
- All that matters is that the KeyboardController and PS3Controller classes see the Character class as a Movement interface instead. This eliminates the need to see the referenced character as a Character, but rather as a Movement interface referenced variable.

Interfaces

- KeyboardController can assume that the reference variable can invoke up(), down(), left(), and right().
- So whenever a left arrow is pressed then the left() method is invoked via the reference variable and similarly for the other arrow keys.
- Similarly this can be done on the D-Pad for the PS3Controller class.

Interfaces

- Taking this a step further we can assume that the Character class is abstract and we have many subclasses such as an Ogre, Wizard, and Human.
- Given this we can implement the movements differently for each Ogre, Wizard, and Human.
- Alternatively we could make the KeyboardController and PS3Controller classes expect abstract classes of the form Character.

Interfaces

- However doing this limits Movement type objects to Characters. If we keep the behaviour in interface form we can allow for objects that usually are not considered movable to be moveable since in Graphics terms they all have locations on the Screen.
- Thus we could have non-Character objects controlled simply by allowing them to implement the Movement interface and be passed into the controller classes as reference varaibles.

Interfaces

```java
public interface Movement {

    public void up();
    public void down();
    public void left();
    public void right();
    public void print();

}
```

```java
public class Character implements Movement{

    private float x;
    private float y;

    public Character(float x, float y){
        this.x = x;
        this.y = y;
    }
    public void up() {
        ++this.y;
    }
    public void down() {
        --this.y;
    }
    public void left() {
        --this.x;
    }
    public void right() {
        ++this.x;
    }
    public void print() {
        System.out.println("(" + x + " , " + y + ")");
    }

}
```

Interfaces

```
public class KeyboardController extends JFrame implements KeyListener {

    private Movement charcterMovement;

    public KeyboardController(Movement characterMovement){
        super("Keyboard Controller");
        this.charcterMovement = characterMovement;
        setVisible(true);
        this.setDefaultCloseOperation(DISPOSE_ON_CLOSE);
        this.setSize(300,300);
        this.addKeyListener(this);

    }
```

Interfaces

This method exists within the KeyboardController class.

```
public void keyReleased(KeyEvent keyE) {

    int code = keyE.getKeyCode();

    switch(code){
    case KeyEvent.VK_UP:
        this.charcterMovement.up();
        break;
    case KeyEvent.VK_DOWN:
        this.charcterMovement.down();
        break;
    case KeyEvent.VK_LEFT:
        this.charcterMovement.left();
        break;
    case KeyEvent.VK_RIGHT:
        this.charcterMovement.right();
        break;
    default:
        break;
    }

    this.charcterMovement.print();

}
```

Interfaces

```
public class Application {
    public static void main(String [] args){
        Character character = new Character(0.0f,0.0f);
        KeyboardController keyboard = new KeyboardController(character);
    }
}
```

The output is to the right:

Notice how each consecutive (x,y) point is different by 1 in either direction. This corresponds to the single arrow presses.

```
(0.0 , 1.0)
(-1.0 , 1.0)
(0.0 , 1.0)
(0.0 , 0.0)
(-1.0 , 0.0)
(0.0 , 0.0)
(1.0 , 0.0)
(1.0 , 1.0)
(1.0 , 2.0)
(0.0 , 2.0)
(-1.0 , 2.0)
(-1.0 , 1.0)
(-1.0 , 0.0)
(-1.0 , 1.0)
(0.0 , 1.0)
(1.0 , 1.0)
(1.0 , 2.0)
(1.0 , 3.0)
(0.0 , 3.0)
(-1.0 , 3.0)
```

“**static**” keyword

- A single instance across the lifetime of a process.
- Often confused with a non-changing variable. But this is wrong, it still can be modified.
- The idea behind this modifier is to tell Java that you want this variable to be the only instance within a class. It exists whether or not you instantiate the class it is defined in.

“.” and Eclipse

If you have a reference to an object and press the “.” (dot) operator then you can get **info** on the available methods of that object. This eliminates the need to search elsewhere.