

And now for something  
completely different

MORE JAVA

# The keyword *this*

- This is the keyword referring to the very class its in

- ```
Public class Num{  
    private int x;  
    public Num(int x){x =x}  
}
```

- But which x is which??

- ```
public Num(int x){this.x =x}
```

- Both the reader and the computer know for sure which x is which

# *this* continued

- This can also be used as a constructor

- ```
public class Rectangle {  
    private int x, y, width, height;  
    public Rectangle() { this(0, 0, 0, 0); }  
    public Rectangle(int width, int height) {  
        this(0, 0, width, height); }  
    public Rectangle(int x, int y,  
                     int width, int height) {  
        this.x = x;  
        this.y = y;  
        this.width = width;  
        this.height = height; }  
    ... }
```

# Inheritance

- OO inheritance is supposed to be like real life inheritance. We get attributes, and behaviours from our parents and so do Java Classes
- We do this by using the keyword "extends".
- Only classes inherit, not packages, fields or methods

# Inheritance example

- ```
Public class Bicycle{  
    private int cadence;  
    private int gear;  
    private int speed;  
    ...  
    public getSpeed(){return speed;}  
    ...  
}
```
- ```
Public class MountainBike extends Bicycle{  
}
```
- MountainBike will have all the same fields and methods of Bicycle. You can also add more

# More on Inheritance

- Both logically and in our code, a Bicycle can be a MountainBike, but a MountainBike is not just any Bicycle

- ...  
`Bicycle myBike = new MountainBike();`  
...

- Both logically and in our code, a Bicycle can be a MountainBike, but a MountainBike
- This is dynamic typing since we don't truly know the type until runtime.

# The keyword *super*

- Super works just like this, only instead of referring to the current class, it refers to the parent class
- Super can refer to fields, methods or the constructor so long as the interface chosen allows it.

# Back to Interfaces

- There are really 4 main levels of protection, last time I mentioned 2; public and private.
- There is also protected – means that it is visible to the package and to all subclasses.
- Also you get a different interface when you type nothing – means that it is visible to the package

# Interface Review

| Modifier      | Class | Packag<br>e | Subclas<br>s | World |
|---------------|-------|-------------|--------------|-------|
| Public        | Y     | Y           | Y            | Y     |
| protect<br>ed | Y     | Y           | Y            | N     |
| nothing       | Y     | Y           | N            | N     |
| private       | Y     | N           | N            | N     |

# Method overriding vs overloading

- We override methods when we declare methods with the same argument list and name as one in the class (usually inherited)
- We overload a method when we declare it to have a different argument list and the same name
- We cannot have a method with the same argument list and a different return type

# The object class

- Implicitly, every class in Java extends the Object class which contains the following useful methods (and more)
- `public boolean equals(Object obj)`  
Indicates whether some other object is "equal to" this one.
- `public String toString()`  
Returns a string representation of the object. You should override this method to make it useful.

# The keyword *final*

- The final keyword will do the following
  - It will make fields constant
  - It will make it so methods cannot be overridden in subclasses. (Good idea for things called within constructors)
  - It will make it so classes cannot be subclassed (The string class is final)

# The keyword *abstract*

- Abstract classes are classes that cannot be instantiated, only subclassed.
- They can have fields, methods just like any class, but no actual objects.
- We can also have abstract methods, these have a signature but no interface