Object Basics: how OO works

** instance variables
** reference variables
** using objects
** overloading
** scope
** local variables *versus* instance variables
** creating objects
** methods
** access control
** overriding
** constructors

Chapter 2 (sections 2.1–2.3; 2.5–2.8; 2.10–2.11) – "Big Java" book
Chapters 2+3 – "Head First Java" book
Chapters 4+6 – "Introduction to Java Programming" book
Chapter 3 – "Java in a Nutshell" book

Objects and Classes in Java

- In software terms, the description of a group of objects is known as a class, whereas a particular member of a class is known as an object (or instance). For example,
  - `myCar` is an instance of the overall class `Car`
  - you are a specific instance of the class `Student`
- A class only exists at compile time; an object only exists at runtime.

- ClassName is the name of the class, which must be the same as the file name.
- Typically a class provides the template for an object: in the "class body", we declare the attributes and operations of the class.
  - attributes are called instance variables;
  - operations are called methods.

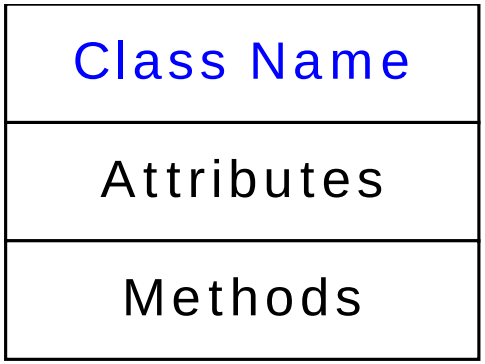
Attributes

- Attributes represent the state of an object. For example,
  - **`number`**, **`owner`**, and **`balance`** all contribute to the internal state of a particular ‘bank account’ object.

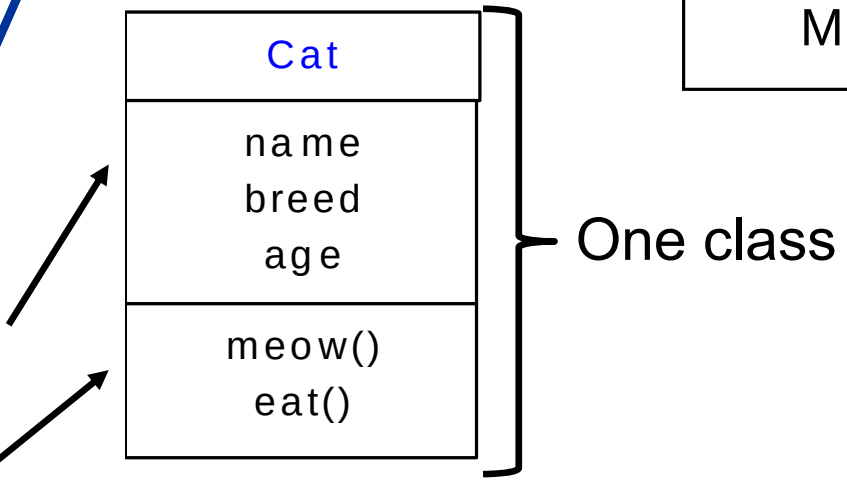
- There are 2 main types of attributes:
  - Instance variables (discussed now)
  - Class variables (discussed later)

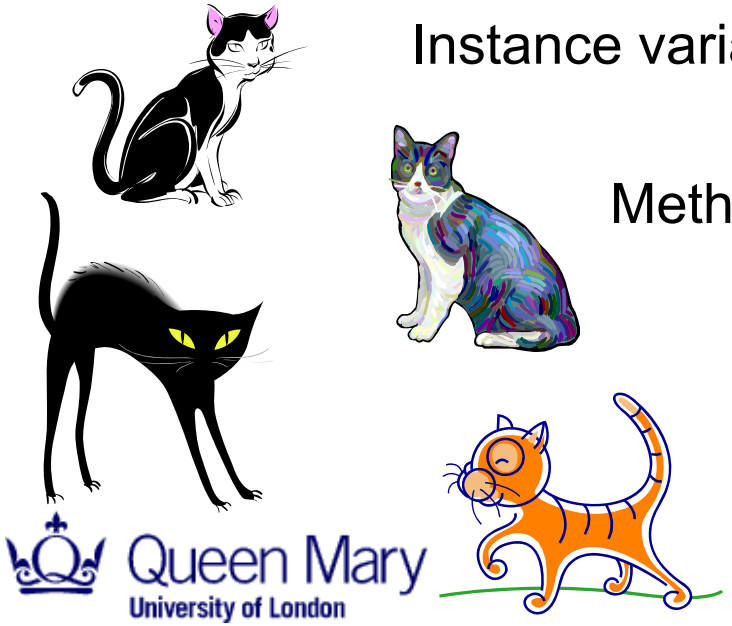
UML and Classes *versus* Objects

- Class diagram notation is part of a widely used software design notation called UML.
- It consists of 3 rectangles with horizontal lines:



- A class is like a template (or blueprint) for an object.





There can exist many objects or instances of a class.

UML = Unified Modelling Language

Creating a Cat (1/2)

Write your class: Cat.java

Where is the **main()**?

Instance variables first →

Our first method! →

```
public class Cat {
  String name;
  String colour;
  int age;

  public void meow() {
    System.out.println("Meow! Meow");
  }

  public void eat() {
    System.out.println(“Yummy! Yummy");
  }
} // end class cat
```

Creating a Cat (2/2)

Write a test class:

CatTestClass.java

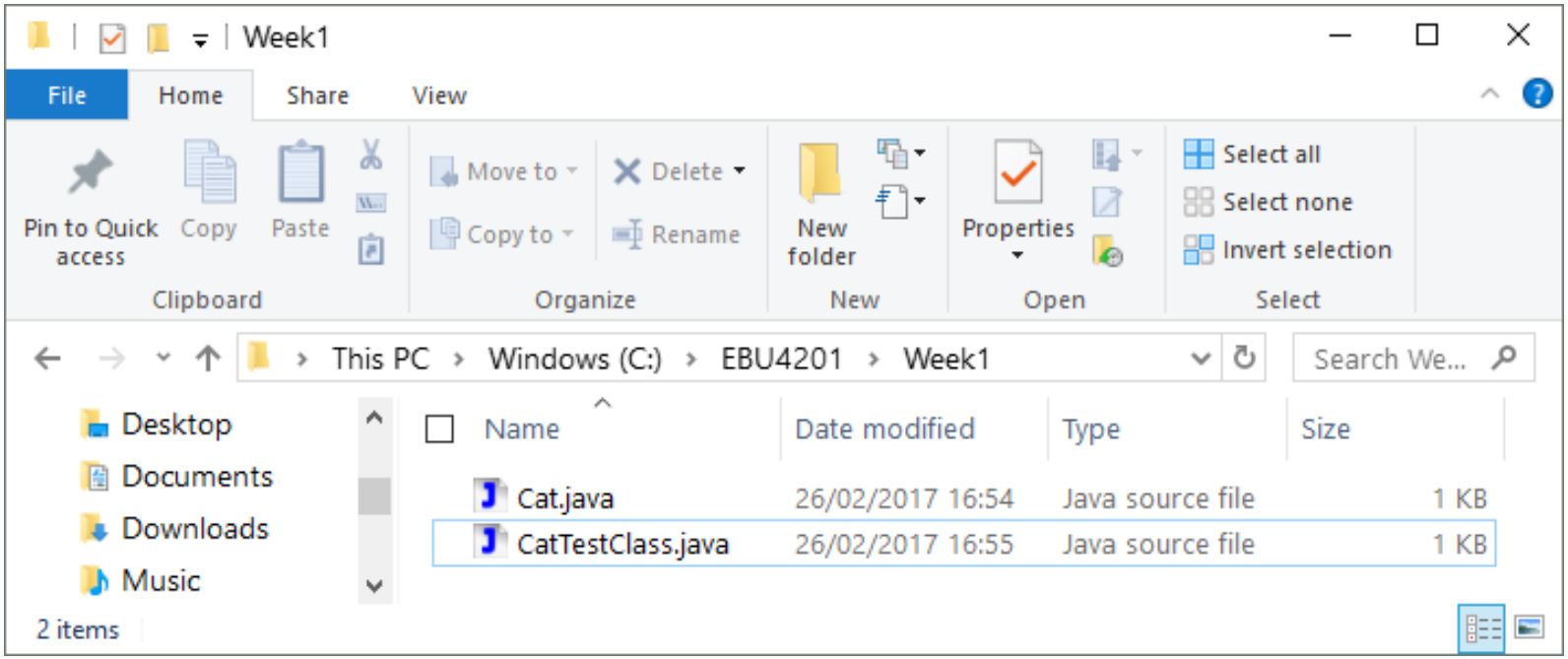
```
public class CatTestClass {
   public static void main(String[] args) {
      // cat test code ...
      Cat myCat = new Cat(); // make a new cat
      myCat.name = "Fluffy"; // set the cat's name
      myCat.meow();          // make it meow!
   }
} //end class CatTestClass
```

There are only 2 real uses for the **`main()`** method:

1. To “test” your real class.
2. To launch (or start) your Java application.

Compiling and using our Cat!

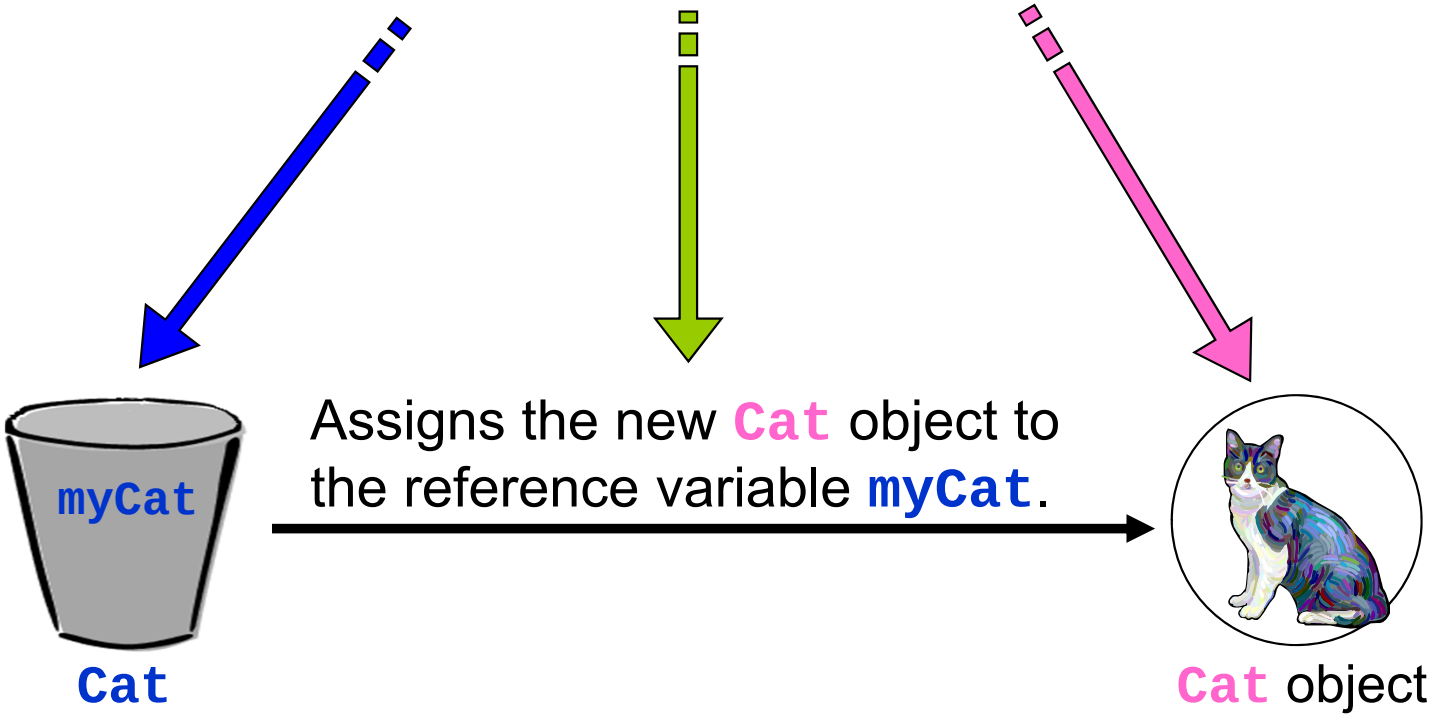
Create the files, and store them in the same directory.



Compile both files and run the one that has the `main()` method!

Cat's creation

```
Cat myCat = new Cat();
```



Tells JVM to allocate space for reference variable of type `Cat` (forever) called `myCat`.

Tells JVM to allocate space for a new `Cat` object on the heap.

Exercises

5 minutes

1. Also print the cat's name, such that the program displays:

```
Fluffy Meow! Meow
```

2. Now create another `Cat` object, such that the program displays:

```
Fluffy Meow! Meow
Catty Yummy! Yummy
```

Using instance variables and methods

- General rules:
  - to access a (public) instance variable **v** of an object **o**, we reference it using the dot notation*:*

```
o.v
```

  - to invoke a (public) method **m** of an object **o**, we also reference it using the dot notation:

```
o.m()
```

Constructors (1/2)

- A constructor is a special method, with same name as the class name, used for initialisation.
  - A constructor always has the same name as the class.
  - It does not have a return type, not even **`void`**!
  - An empty no-argument constructor is provided for you by Java.

```
public Cat() { }
```

We have been using this special constructor without even knowing it!

- Constructors may have parameters if you write your own.

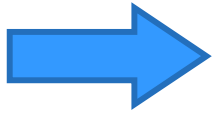
```
public class Cat {
  String name;
  String colour;
  int age;
  public Cat(String name) {
     this.name = name;
  }
  public void meow() {
    System.out.println("Meow! Meow");
  }
  public void eat() {
    System.out.println("Yummy! Yummy");
  }
}
```

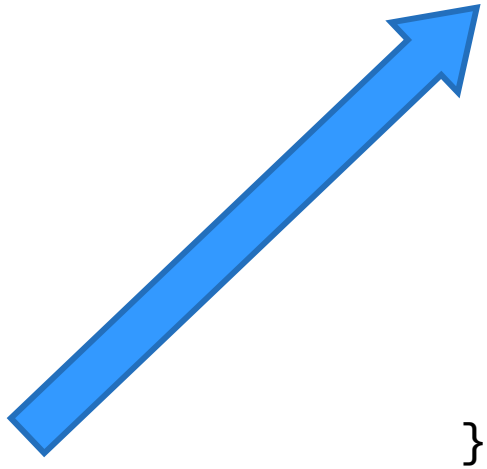
`Cat myCat = new Cat();` will now NOT work!
Must use **`Cat myCat = new Cat("Fluffy");`**

Constructors (2/2)

- A class can have many constructors.

```
Cat myCat = new Cat();
```





```
public class Cat {
  String name;
  String colour;
  int age;
  public Cat() {

  }
  public Cat(String name) {
     this.name = name;
  }
  public void meow() {
    System.out.println("Meow! Meow");
  }
  public void eat() {
    System.out.println(“Yummy! Yummy");
  }
}
```

```
Cat anotherCat = new Cat(“Fluffy”);
```

Data Encapsulation

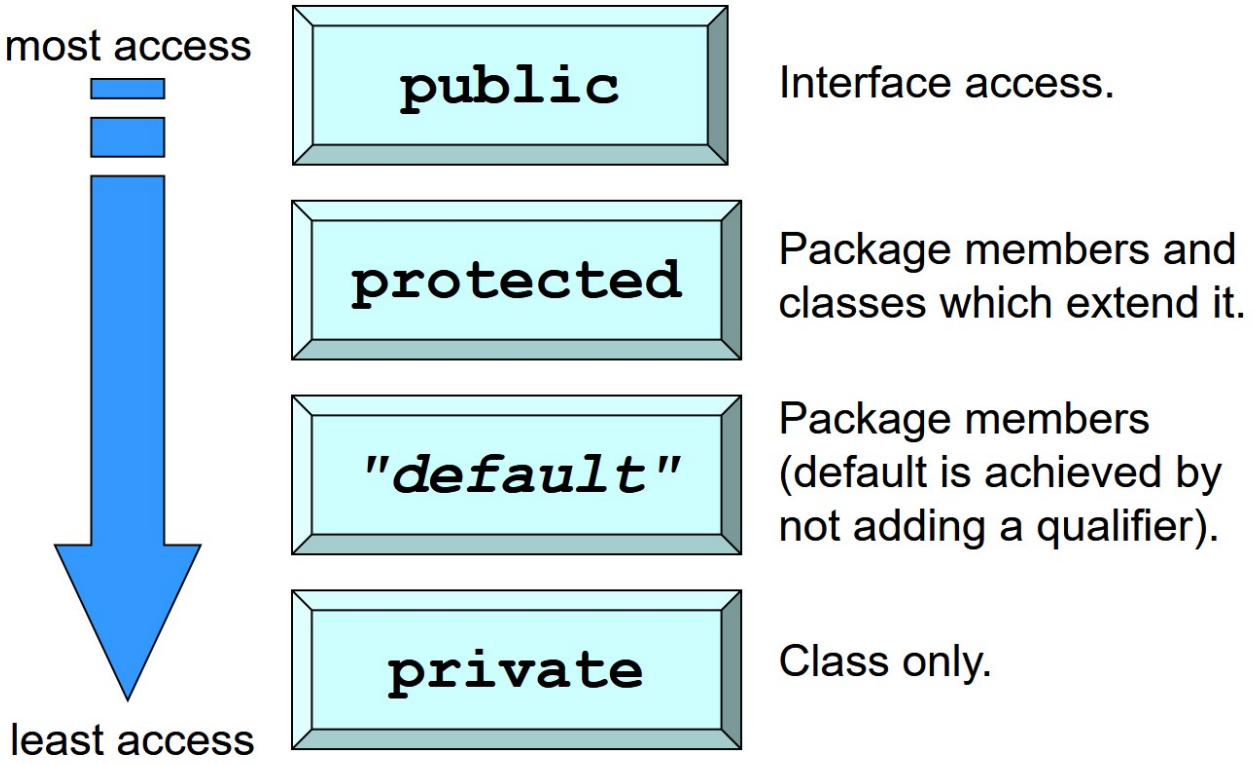
- Data Encapsulation (or information hiding) refers to when the internal state and operation are hidden from others.
  - This is a good thing! Since objects are only accessible through well defined interfaces, ideally nothing unexpected should happen!
  - The more information class A knows about class B, the greater the possibility that changing class A will adversely affect Class B. In an ideal world, making internal changes to class A should have no, or very little, effect on other classes.

- An object should be self-governing (or work by itself).
- We should NOT allow direct access to an object's variables.
- Any changes to the object's state (i.e. its variables) should be made ONLY by that object's methods
- Give **`private`** access, unless there is a good reason not to!

Access control allows for encapsulation!

Queen Mary University of London

Access Modifiers



- We do not want to make all our instance variables and methods public – this defeats the purpose of information hiding.
  - We can state whether we want methods and instance variables to be public or private by qualifying them with the **`public`** or **`private`** keywords.
  - Each object has a public interface through which we can manipulate it.
  - The only way that we can manipulate an object is via its interface.
  - The object's state and internal operation are kept behind the scenes.

Accessing the Cat

```
public class Cat {
    private String name;
    private String colour;
    private int age;
    // other code
}
```

- Now only the **Cat** itself can access its attributes, e.g.

```
Cat c = new Cat();
c.age = 5; // ERROR
```

- If we want an attribute to be accessible outside the class, we now provide an interface to it by accessor and mutator methods.

Accessor and Mutator methods for Cat

```
/**
 * This method gets the colour of the cat.
 * @return String  Colour of the cat.
 */
public String getColour() {
  return colour;
}
```

Accessor methods should only be used in cases where you want to provide access to an attribute – if you don't need to provide it, don't!

```
/**
 * This method sets the name of the cat.
 * @param name  Name the cat should have.
 */
public void setName(String name) {
  this.name = name;
}
```

Sometimes, an object needs to be able to refer to itself – the keyword **this** is used to do that. In the **setName** method, **name** refers to the variable passed in and **this.name** refers to the cat's instance variable **name**.

More on Accessor and Mutator methods

- Using accessor (or getter) and mutator (or setter) methods is always preferable to declaring things as **`public`**.
  - By using those methods, the object can control who sees and does what with it and thus has a better chance of remaining consistent.

- You should get into the habit of always qualifying your instance variables and methods.

- All getters and setters should have names that conform to the following:

```
variableType getVariableName()
void setVariableName(VariableType)
```

Cat class

```
public class Cat {
  private String name;
  private String colour;
  private int age;

  public Cat() { }

  public Cat(String name) {
     this.name = name;
  }

  /**
   * This method gets the colour of the cat.
   * @return String colour of the cat.
   */
  public String getColour() {
    return colour;
  }

  /**
   * This method sets the name of the cat.
   * @param name name of the cat should have.
   */
  public void setName(String name) {
      this.name = name;
  }

    public void meow() {
      System.out.println("Meow! Meow");
    }

    public void eat() {
      System.out.println(“Yummy! Yummy");
    }
}
```

… and things for you to try out!

Instance and Local Variables

- Instance variables are declared inside a class but not inside a method.
  - are initialised to the default value;
  - are valid (or have scope) throughout the entire class.

- Local variables are declared within a method.
  - are NOT initialised to the default value, and so must be initialised.
  - have scope only within that method!

```
public class Example {
   private String aString;
   private int anInt;          instance variables

   public void aMethod() {
      int loop = 5;            local variable
   }
}
```

Example: instance and local variables

```
public class SomeThing {
  private int a, b = 12;

  public int doIt() {
    int total;
    a = total * 2;
    if (b < 20)
       total = a + b;
    return total;
  }
}
```

✓ **a** is an instance variable and thus initialised to its default value (**0**)

✓ **b** is an instance variable but initialised to the value given (**12**)

✘ **total** is a local variable, so it does not get initialised to anything – the compiler will complain at line:

```
a = total * 2;
```

- Compiler will complain that the local variable **total** may not have been initialised.

fixing the problem

```
public class SomeThing {
  private int a, b = 12;

  public int doIt() {
    int total = 0;
    a = total * 2;
    if (b < 20) total = a + b;
    return total;
  }
}
```

Queen Mary University of London

Method parameters and local variables …

```
public void doIt(int b)
```

- Method parameters are virtually the same as local variables!
  - They are declared inside the method.
  - They are valid (or in scope) only inside the method.
  - They are always initialised (by the caller of the method).

A Flower class (1/2)

```
public class Flower {
  private String petalColour;
  private double height;

  public Flower() {  }

  public Flower(String petalColour, double height) {
    this.petalColour = petalColour;
    this.height = height;
  }

  /**
    * This method sets the petalColour of a Flower.
    * @param petalColour The colour of the petals.
    */
  public void setPetalColour(String petalColour) {
    this.petalColour = petalColour;
  }
```

A Flower class (2/2)

```
  /**
    * This method gets the petalColour of a Flower.
    * @return The colour of the petals.
    */
  public String getPetalColour() {
    return this.petalColour;
  }

  public void setHeight(double height) {
    if ((height > 0) && (height < 4.7)) {
       this.height = height;
    }
     else System.out.println("Invalid height. Height must
                              be between 0 and 4.7");
  }

  public double getHeight() { return this.height; }
}
```

Test programs for Flower

```
public class FlowerTest {
   public static void main (String[] args) {
     Flower f1 = new Flower();
     f1.setPetalColour("Red");
     f1.setHeight(2.5);

     Flower f2 = new Flower();
     f2.setPetalColour("Blue");
     f2.setHeight(5.5);   // will print off an error
     f2.setHeight(4.5);   // this is better!
   }
}
```

```
public class FlowerTest2 {
  public static void main (String[] args) {
    Flower f1 = new Flower("Red", 2.5);
    Flower f2 = new Flower("Blue", 4.5);
  }
}
```

Does everything work?

- What happens if we write another test class?

```
public class FlowerTest3 {
   public static void main (String[] args) {
     Flower f1 = new Flower("Red", 1332.5);
     Flower f2 = new Flower("Blue", -25.1);
   }
}
```

Remember, we wanted our flowers to be between **0** and **4.7**!

```
public void setHeight(double height) {
  if ((height > 0) && (height < 4.7)) {
    this.height = height;
  } else
      System.out.println("Invalid height. Height must be
                          between 0 and 4.7");
}
```

Does the constructor work?
If so, does it obey that **height** rule?

Modifying our constructor …

We could remove the default constructor and rewrite our own constructor: the checks that we did in **`setHeight(double)`**!

```
public Flower(String petalColour, double height) {
  this.petalColour = petalColour;
  if ((height > 0) && (height < 4.7)) {
    this.height = height;
  } else
      System.out.println("Invalid height. Height must
                          be between 0 and 4.7");
}
```

This works! A **`Flower`** will now always have the correct size!
(Whether set by the constructor or by the setter method.)

Changing the specification (again!)

- What if the specification changes?
  - Assume that the code has all been written and you are told that actually `Flowers` should be between `0.5` and `5.6` in height.

- You now have to change the code in 2 places!
- This is the easiest way to introduce errors; what if you forget about one of the checks?
  (Not likely with 2 checks, but what about with 500?)
- If you forget once, your program could set flowers to be the wrong height!

Reuse! Reuse! Reuse!

Reuse your code! If we always use the setter methods we have created, we only need to worry about changing the code in one place!

```
public Flower(String petalColour, double height) {
  this.setPetalColour(petalColour);
  this.setHeight(height);
}
```

Can still use the keyword **`this`** for easier reading!
(Can be used to refer to an object's instance variables and methods!)

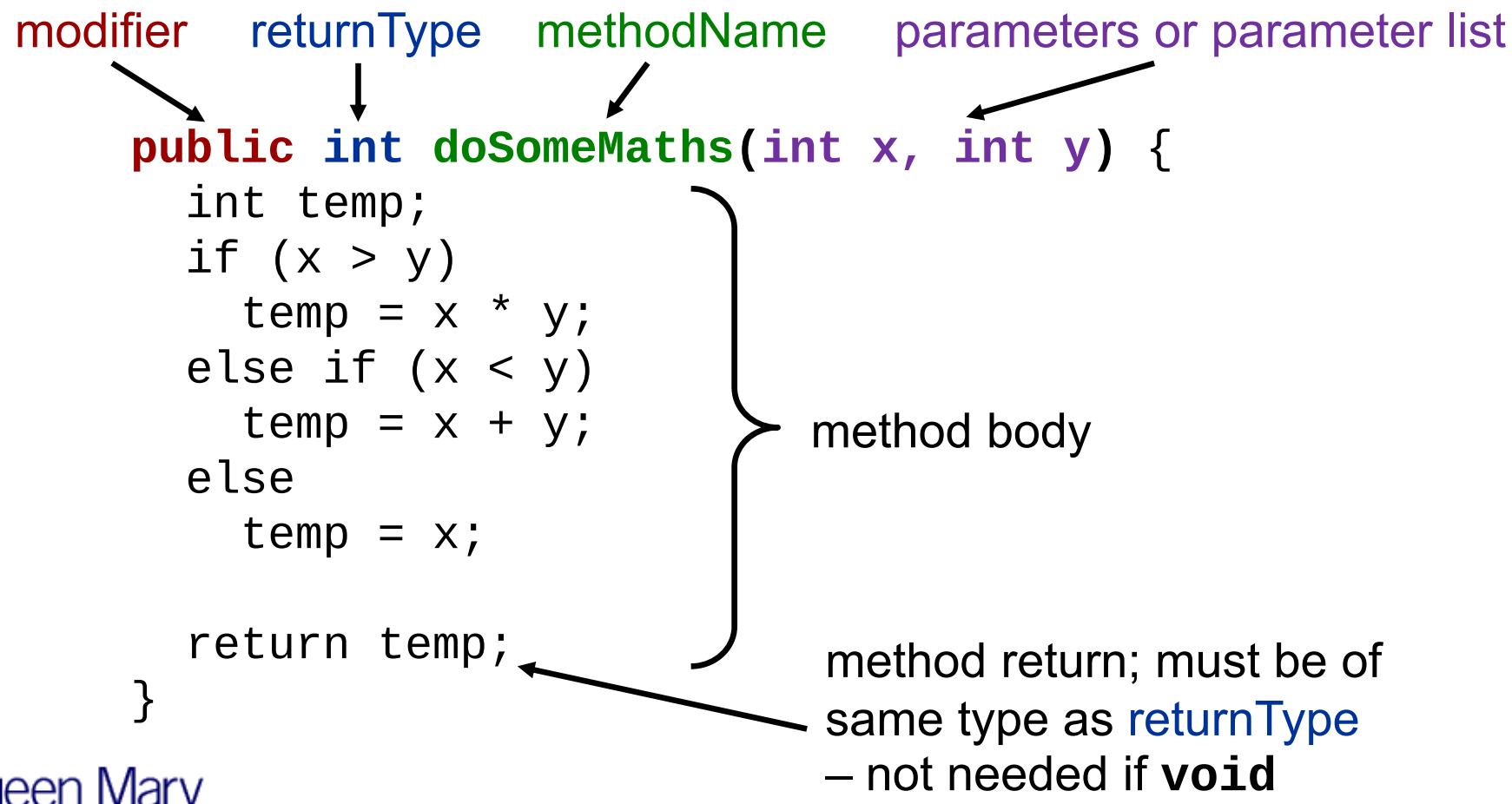
This solution works for validating the input and is easier to maintain!

… and things for you to try out!

More about methods …

General Template

```
modifiers returnType methodName(parameters) {
   statements;
}
```



Calling a method …

- To use a method, we call or invoke it.

```
public class TestMaths {
  public int doSomeMaths(int x, int y) {
    int temp;
    if (x > y)       temp = x * y;
    else if (x < y) temp = x + y;
    else             temp = x;
    return temp;
  }
  public static void main(String[] args) {
    int i = 5;
    int j = 7;
    TestMaths m = new TestMaths();
    int result = m.doSomeMaths(i,j);
    System.out.println(result + " " + m.doSomeMaths(i,j));
  }
}
```

can store it in a variable

or use it directly

Program flow …

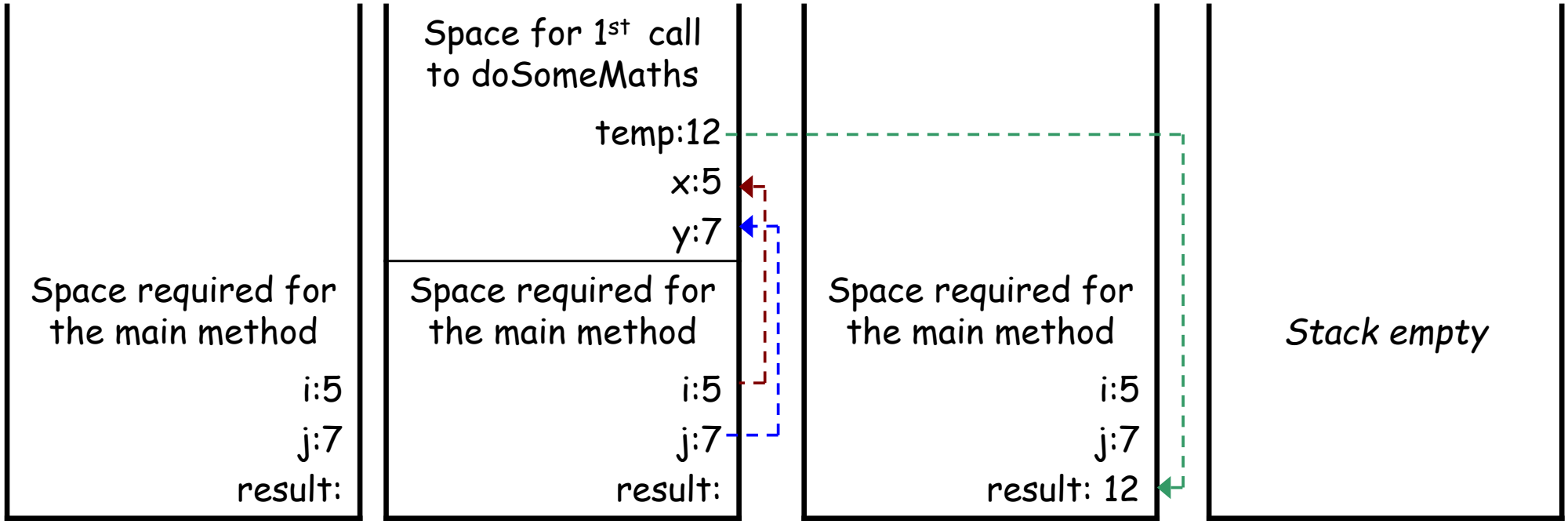
This slide has lots of animation; you must be in class to fully understand.

```
public class TestMaths {
    public int doSomeMaths(int x, int y){
       int temp;
       if (x > y)
           temp = x * y;
       else if (x < y)
           temp = x + y;
       else
           temp = x;

        return temp;
    }
    public static void main(String args[]){
       int i = 5;
       int j = 7;
       TestMaths m = new TestMaths();
       int result = m.doSomeMaths(i, j);
       System.out.println(result + " " +m.doSomeMaths(j,i));

    }
}
```

Call Stack for `TestMaths` – first run only …



More about this later …

More about Methods …

- A method uses parameters.
- The caller passes arguments.

```
class PassByWhat {
  void go(int z) {          <-- z is a parameter
    // do something
    z = z + 7;
  }

  public static void main(String[] args) {
    PassByWhat p = new PassByWhat();
    int x = 7;
    p.go(x);                <-- x is an argument
    // what is x now?
  }
}
```

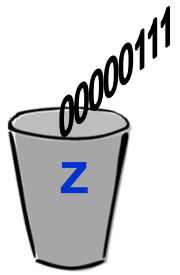
Pass-by-value

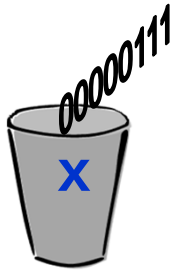
This slide has lots of animation; you must be in class to fully understand.

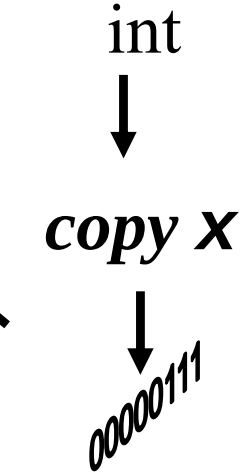
- Pass-by-value → Pass-by-copy

```
class PassByWhat{
  void go(int z){
    // do something
    z = z + 7;
  }

  public void static main(String[] args){
    PassByWhat p = new PassByWhat();
    int x = 7;
    p.go(x);
    // what is x now?
  }
}
```







x doesn't change even if **z** does!
So what is **x** now?