Types and Variables

- We write code like:

```
int x = 512;
int y = 200;
int z = x+y;
```

- The high-level language has a series of *primitive* (built-in) types that we use to signify what's in the memory
  - The compiler then knows what to do with them
  - E.g. An "int" is a primitive type in C, C++, Java and many languages. It's usually a 32-bit signed integer
- A variable is a name used in the code to refer to a specific instance of a type
  - x,y,z are variables above
  - They are all of type int

E.g. Primitive Types in Java

- “Primitive” types are the built in ones.
  - They are building blocks for more complicated types that we will be looking at soon.
- boolean – 1 bit (true, false)
- char – 16 bits
- byte – 8 bits as a signed integer (-128 to 127)
- short – 16 bits as a signed integer
- int – 32 bits as a signed integer
- long – 64 bits as a signed integer
- float – 32 bits as a floating point number
- double – 64 bits as a floating point number

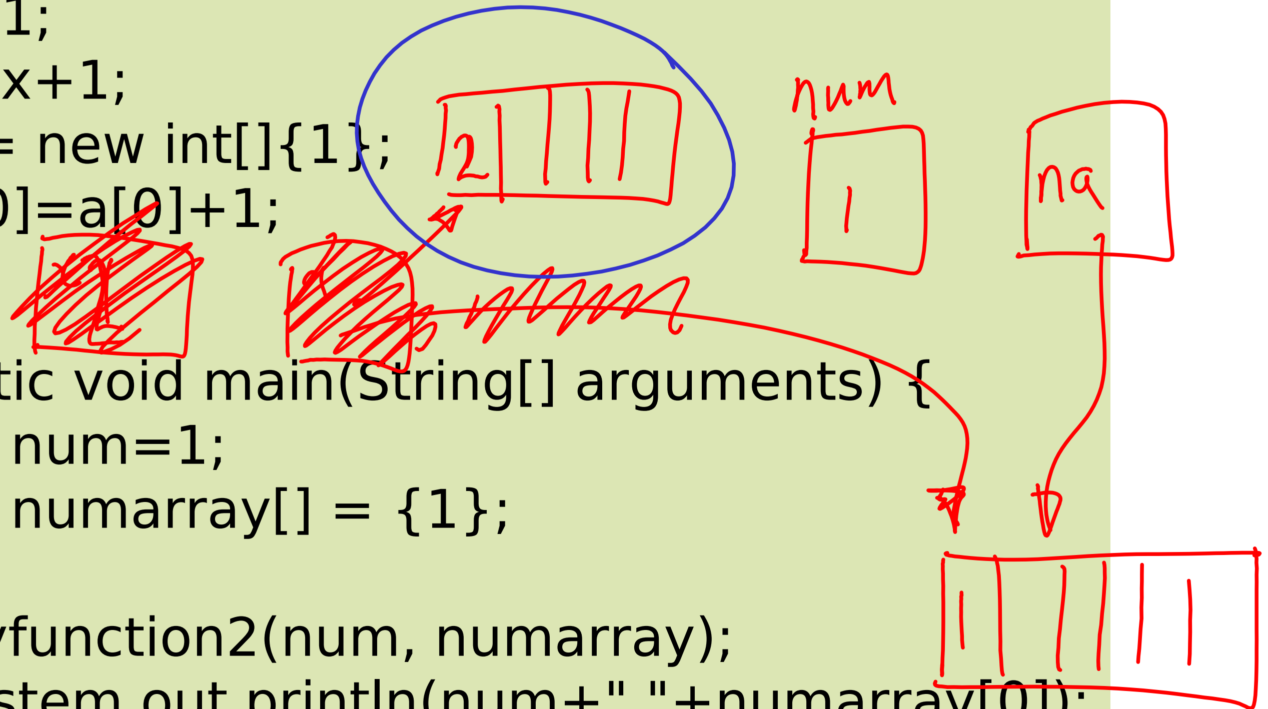
See Workbook 1

Check...

```
public static void myfunction(int x, int[] a) {
        x=x+1;
        a[0]=a[0]+1;
}


public static void main(String[] arguments) {
        int num=1;
        int numarray[] = {1};

        myfunction(num, numarray);

        System.out.println(num+" "+numarray[0]);
}
```

A. “1 1”
B. “1 2”
C. “2 1”
D. “2 2”

Check...



A. “1 1”
B. “1 2”
C. “2 1”
D. “2 2”

Passing Procedure Arguments In Java

```
class Reference {

   public static void update(int i, int[] array) {
      i++;
      array[0]++;
   }

   public static void main(String[] args) {
      int test_i = 1;
      int[] test_array = {1};
      update(test_i, test_array);
      System.out.println(test_i);
      System.out.println(test_array[0]);
   }

}
```

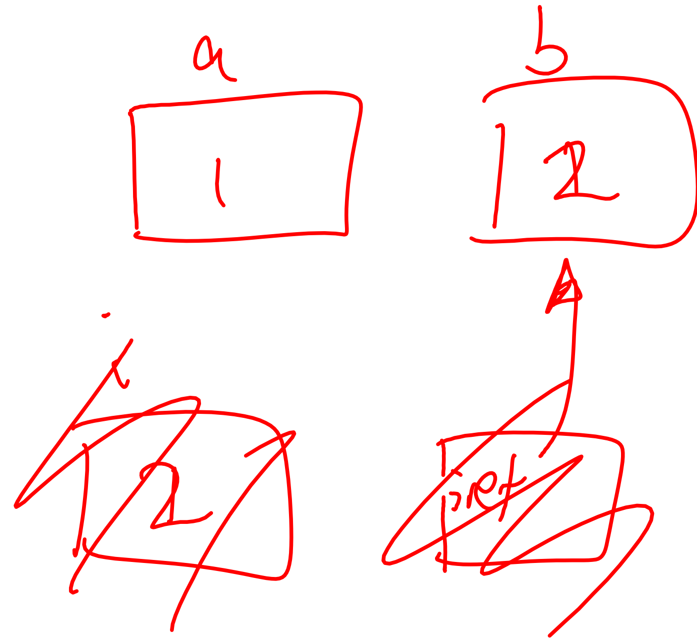
“1”

“2”

See Workbook 3

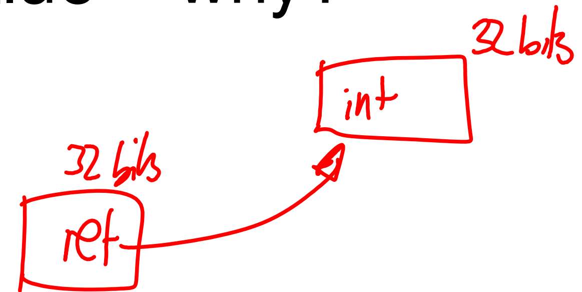
Passing Procedure Arguments In C

```
void update(int i, int &iref){
  i++;
  iref++;
}

int main(int argc, char** argv) {
  int a=1;
  int b=1;
  update(a,b);
  printf("%d %d\n",a,b);
}
```



Reference Types in Java

- Back to Java
  - Primitives always passed by value – why?
    - Set size in memory
    - Of similar size to a reference...



  - Anything that isn't a primitive type is passed by reference
    - We call them reference types
      - Arrays
      - Classes
      - Interfaces

See Workbook 3

Object Oriented Programming

Custom Types

- You saw that there was an advantage to declaring your own types in ML
  - First you declared a type and then you wrote functions that could act on it

- In OOP we go a step further
  - We think of types as having both state *and* procedures
  - The idea is that each type groups together *related* state and procedures, providing an implementation of a single *concept*
  - We call our types **classes**

See Workbook 3

Classes, Instances and Objects I

- Primitive types are pre-defined e.g. int defines 32-bit integer in Java
- We create **instances** of a primitive type by declaring a variable of that type
  - E.g.

    ```
    int x=7;
    int y=6;
    ```

    declares two instances of type int

Classes, Instances and Objects II

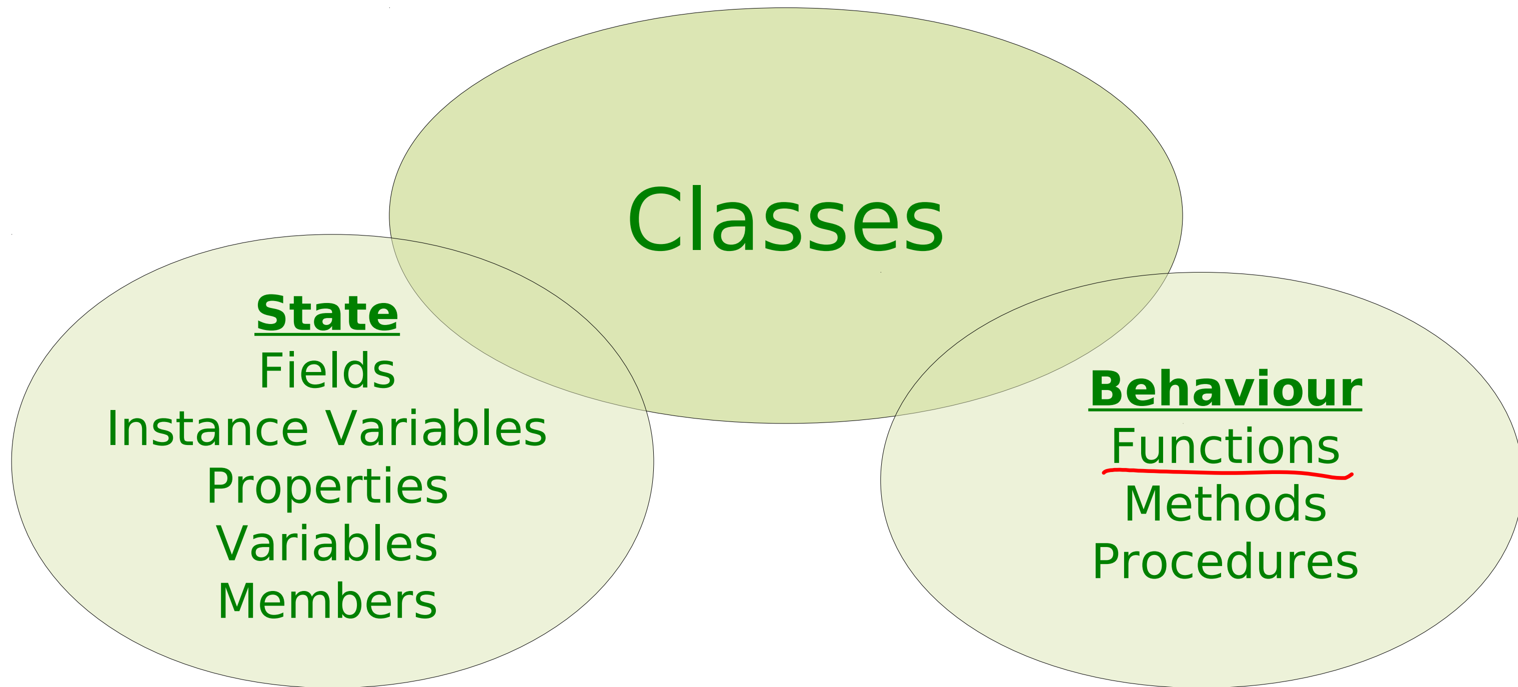
- Classes map to the the type in that they are basically a template for that concept
- We create instances of classes in a similar way. e.g.

```
MyMegaCoolClass m = new  MyMegaCoolClass();
MyMegaCoolClass n = new  MyMegaCoolClass();
```

  makes two instances of class MyMegaCoolClass.

- An instance of a class is called an **object**

Loose Terminology (again!)

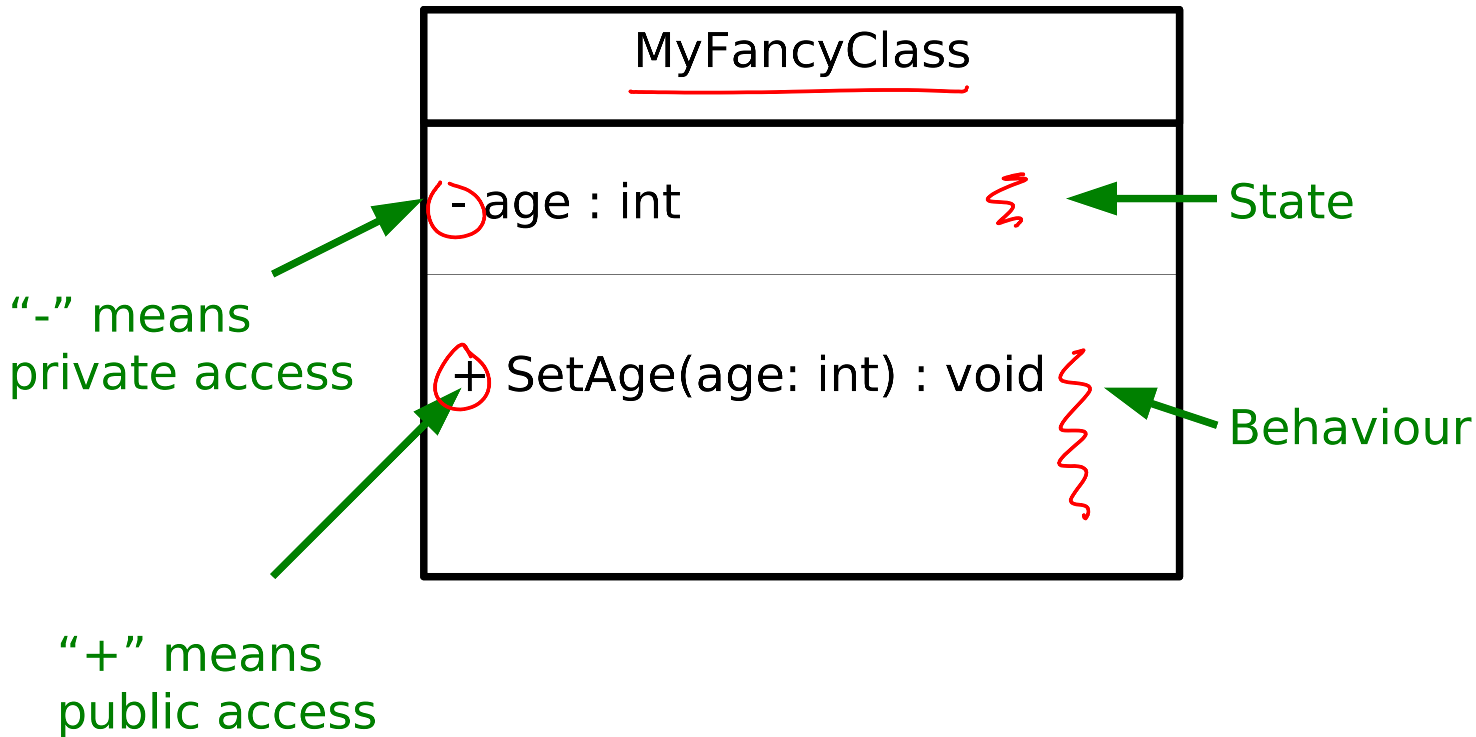


Identifying Classes

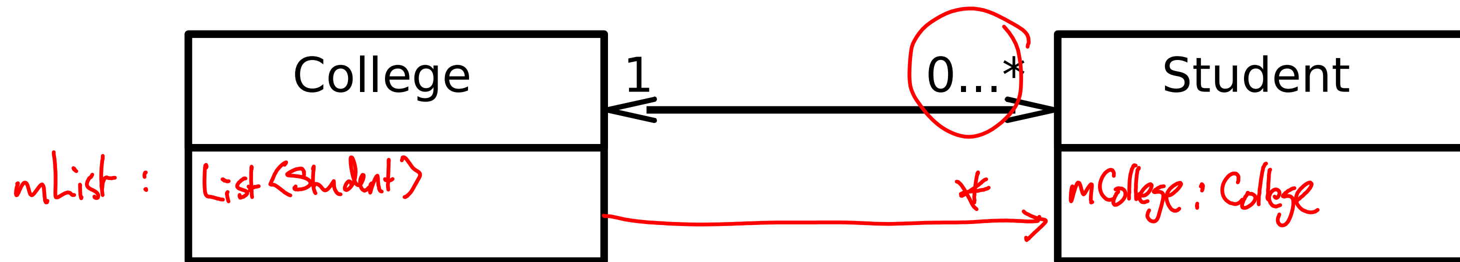
- We want our class to be a grouping of conceptually-related state and behaviour
- One popular way to group is using English grammar
  - Noun → Object
  - Verb → Method

  “Write a simulation of the Earth's orbit around the Sun”

Representing a Class Graphically (UML)



The 'has-a' Association



- Arrow going left to right says “a College has zero or more students”
- Arrow going right to left says “a Student has exactly 1 College”
- What it means in real terms is that the College class will contain a variable that somehow links to a set of Student objects, and a Student will have a variable that references a College object.
- Note that we are only linking *classes*: we don't start drawing arrows to primitive types.

Anatomy of an OOP Program (Java)

Class name

```
public class MyFancyClass {

        public int someNumber;
        public String someText;

        public void someMethod() {

        }

        public static void main(String[] args) {
                MyFancyClass c = new
                        MyFancyClass();
        }

}
```

Class state (properties that an object has such as colour or size)

Class behaviour (actions an object can do)

'Magic' start point for the program (named main by convention)

Create an object of type MyFancyClass in memory and get a reference to it

State

Functionality

Anatomy of an OOP Program (C++)

Class name

```
class MyFancyClass {

public:
        int someNumber;
        public String someText;

        void someMethod() {

        }

};

void main(int argc, char **argv) {
        MyFancyClass c;

}
```

Class state

Class behaviour

'Magic' start point for the program

Create an object of type MyFancyClass

OOP Concepts

OOP Concepts

- OOP provides the programmer with a number of important concepts:
  - Modularity
  - Code Re-Use
  - Encapsulation
  - Inheritance
  - Polymorphism
- Let's look at these more closely...

Modularity and Code Re-Use

- You've long been taught to break down complex problems into more tractable sub-problems.
- Each class represents a sub-unit of code that (if written well) can be developed, tested and updated independently from the rest of the code.
- Indeed, two classes that achieve the same thing (but perhaps do it in different ways) can be swapped in the code
- Properly developed classes can be used in other programs without modification.

Encapsulation I

```
class Student {
   int age;
};

void main() {
   Student s = new Student();
   s.age = 21;

   Student s2 = new Student();
   s2.age=-1;

   Student s3 = new Student();
   s3.age=10055;
}
```

- Here we create 3 Student objects when our program runs
- Problem is obvious: nothing stops us (*or anyone using our Student class*) from putting in garbage as the age
- Let's add an *access modifier* that means nothing outside the class can change the age

Encapsulation II

```
class Student {
  private int age;

  boolean SetAge(int a) {
    if (a>=0 && a<130) {
        age=a;
        return true;
    }
    return false;
  }

  int GetAge() {return age;}
}

void main() {
  Student s = new Student();
  s.SetAge(21);

}
```

- Now nothing outside the class can access the *age* variable directly
- Have to add a new method to the class that allows *age* to be set (but only if it is a sensible value). i.e. **SetAge()**
- Also needed a **GetAge()** method so external objects can find out the age.

Encapsulation III

- We hid the state implementation to the outside world (no one can tell we store the age as an int without seeing the code), but provided mutator methods to... errr, mutate the state
- This is *data encapsulation*
  - We define interfaces to our objects without committing long term to a particular implementation
- Advantages
  - We can change the internal implementation whenever we like so long as we don't change the interface other than to add to it (E.g. we could decide to store the age as a float and add GetAgeFloat())
  - Encourages us to write clean interfaces for things to interact with our objects

Data Encapsulation / Info Hiding

1. Change internal representation
   - better efficiency (storage)
   - better accuracy (float / double)
   - better readability

2. Enforce constraints
   → set()

3. Compute results on fly
   → Get()

4. Promotes decoupling