Section: Lifecycle of an Object

Constructors

```
MyObject m = new MyObject();
```

- You will have noticed that the RHS looks rather like a function call, and that's exactly what it is.
- It's a method that gets called when the object is constructed, and it goes by the name of a **constructor** (it's not rocket science).
- We use constructors to initialise the state of the class in a convenient way.
  - A constructor has the same name as the class
  - A constructor has no return type

Constructor Examples

Java

```
public class Person {
   private String mName;

   // Constructor
   public Person(String name) {
      mName=name;
   }

   public static void main(
         String[] args) {
     Person p =
         new Person("Bob");
   }

}
```

C++

```
class Person {
   private:
      std::string mName;

   public:
      Person(std::string &name){
         mName=name;
      }
};


int main (int argc,
              char ** argv) {
   Person p ("Bob");
}
```

Default Constructor

```
public class Person {
   private String mName;

   public static void main(String[] args) {
     Person p = new Person();
   }

}
```

- If you specify no constructor at all, Java fills in an empty one for you
- Here it creates Person() for us
- The default constructor takes no arguments (since it wouldn't know what to do with them!)

Multiple Constructors

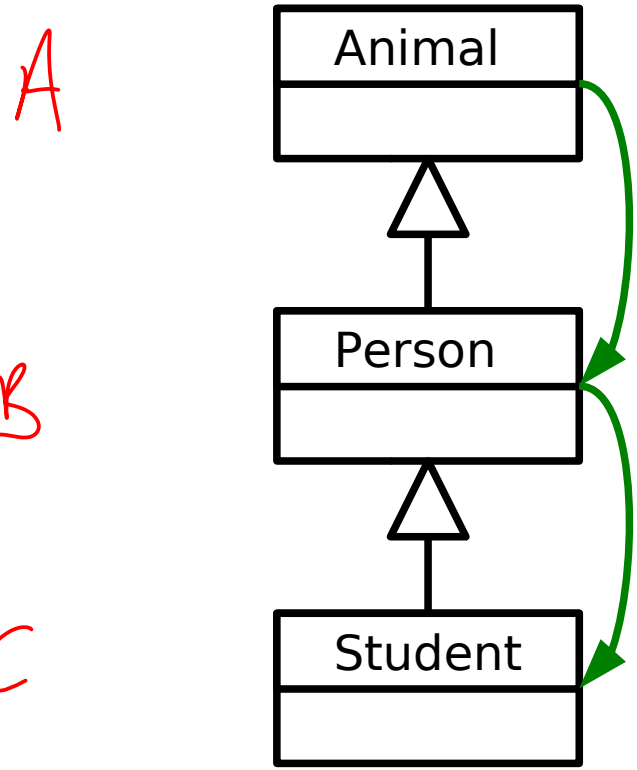
```
public class Student {
    private String mName;
    private int mScore;

    public Student(String s) {
       mName=s;
       mScore=0;
    }
    public Student(String s, int sc) {
        mName=s;
        mScore=sc;
    }

    public static void main(String[] args) {
      Student s1 = new Student("Bob");
      Student s2 = new Student("Bob",55);
    }
  }
```

- You can specify as many constructors as you like.
- Each constructor must have a different signature (argument list)

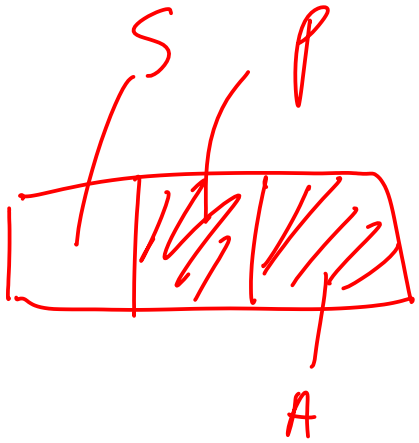
Constructor Chaining

- When you construct an object of a type with parent classes, we call the constructors of all of the parents in sequence

```
Student s = new Student();
```

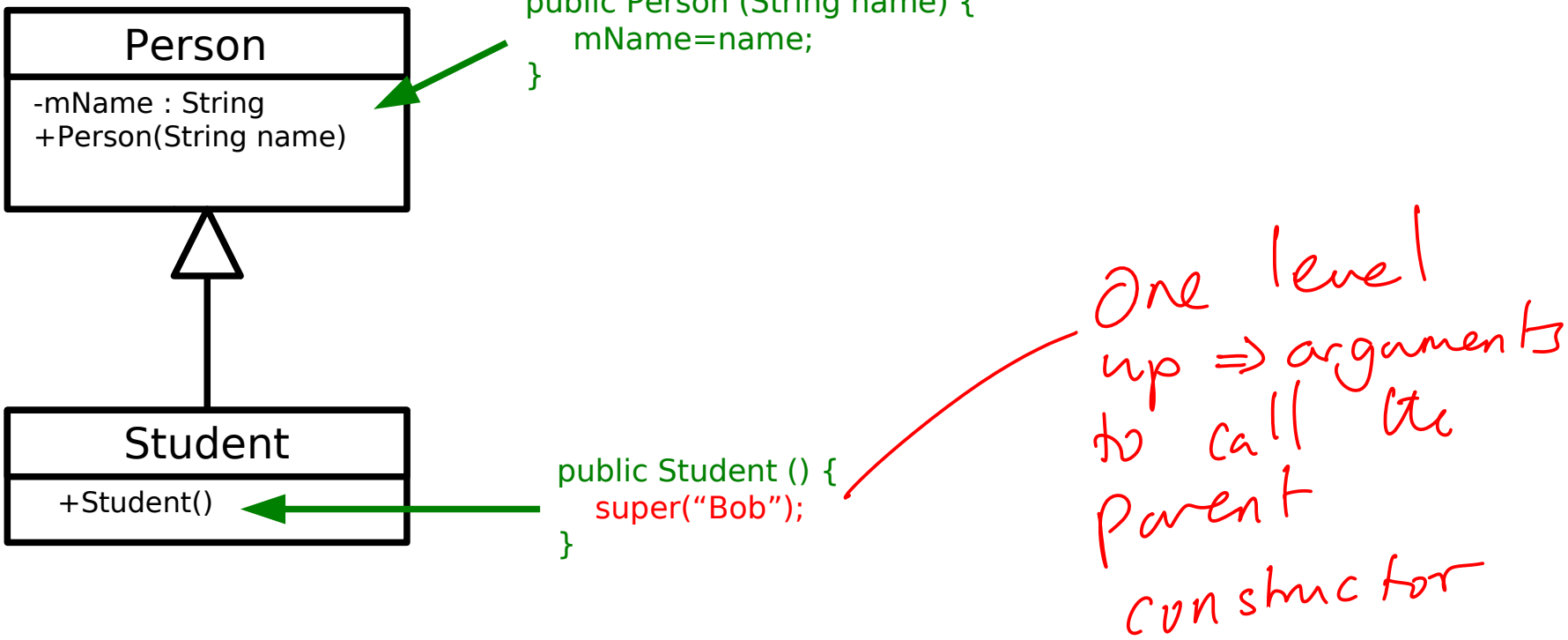


1. Call Animal()
2. Call Person()
3. Call Student()



Chaining without Default Constructors

- What if your classes have explicit constructors that take arguments? You need to explicitly chain:



Destructors

- Most OO languages have a notion of a destructor too
  - Gets run when the object is destroyed
  - Allows us to release any resources (open files, etc) or memory that we might have created especially for the object

C++

```
class FileReader {
  public:

    // Constructor
    FileReader() {
      f = fopen("myfile","r");
    }

    // Destructor
    ~FileReader() {
      fclose(f);
    }

  private :
    FILE *f;
}
```

```
int main(int argc, char ** argv) {

  // Construct a FileReader Object
  FileReader *f = new FileReader();

  // Use object here
  ...

  // Destruct the object
  delete f;

}
```

Cleaning Up

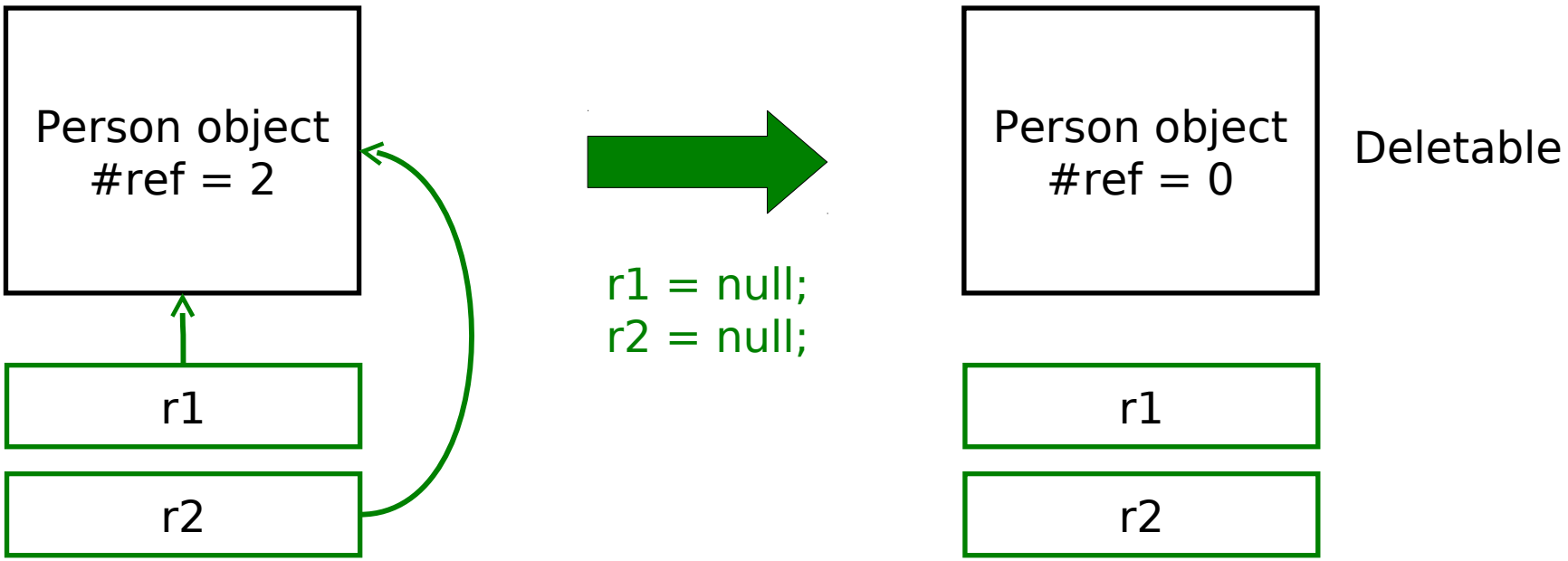
- A typical program creates lots of objects, not all of which need to stick around all the time

- **Approach 1:**
  - Allow the programmer to specify when objects should be deleted from memory
  - Lots of control, but what if they forget to delete an object?
    - A "memory leak"

- **Approach 2:**
  - Delete the objects automatically (**Garbage collection**)
  - But how do you know when an object will never be used again and can be deleted??

Cleaning Up (Java) I

- Java *reference counts*. i.e. it keeps track of how many references point to a given object. If there are none, the programmer can't access that object ever again so it can be deleted



Cleaning Up (Java) II

- Actual deletion occurs through a **garbage collector**
  - A separate process that periodically scans the objects in memory for any with a reference count of zero, which it then deletes.
  - Running the garbage collector is obviously not free. If your program creates a lot of short-term objects, you will soon notice the collector running
    - Gives noticeable pauses to your application while it runs.
    - But minimises memory leaks (it does not prevent them...)

Cleaning Up (Java) III

- One problem with GC is we have no idea *when* an object will actually be deleted. The GC may even decide to defer the deletion until a future run.
- This causes issues for destructors – it might be ages before a resource is closed and available again!
- Therefore Java doesn't have destructors
- It does have **finalizers** that gets run when the GC deletes an object
  - BUT there's no guarantee an object will ever get garbage collected in Java...
  - Garbage Collection != Destruction

Section: Class Variables

Class-Level Data and Functionality I

```
public class ShopItem {
  private float price;
  private float VATRate = 0.2;

  public float GetSalesPrice() {
    return price*(1.0+VATRate);
  }

  public void SetVATRate(float rate) {
    VATRate=rate;
  }

}
```

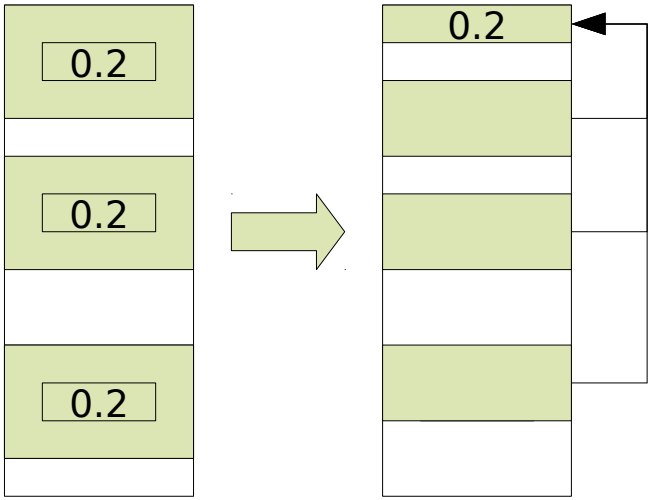
- This is one solution to incorporating VAT into a shop application
- Bad: Every instance will contain a float with the same number
- Bad: If the VAT rate changes, how can we be sure every single object with such a float is properly changed?!

- It can be useful to have *class variables* a.k.a. *static variables*. These are variables that exist per class and not per object
- Create them in Java using the **static** keyword:

```
public class ShopItem {
  private float price;
  private static float VATRate;
  ....
}
```

Variable created only once and has the lifetime of the *program*, not the *object*

Class-Level Data and Functionality II



- We now have one place to update
- More efficient memory usage

Can also make methods **static** too

- A static method must be instance independent i.e. it can't rely on member variables in any way

Sometimes a static method is obviously needed. E.g

```
public class Whatever {
   public static void main(String[] args) {
      ...
   }
}
```

Must be able to run this function without creating an object of type Whatever (which we would have to do in the main()..!)

Why use other static methods?

- A static function is like a function in ML – it can depend only on its arguments + other static data
  - Easier to debug (not dependent on any state)
  - Self documenting
  - Allows us to group related methods in a Class, but does not require us to create an object to run them
  - The compiler can produce more efficient code since no specific object is involved

```
public class Math {
   public float sqrt(float x) {…}
   public double sin(float x) {…}
   public double cos(float x) {…}
}
```

vs

```
public class Math {
   public static float sqrt(float x) {…}
   public static float sin(float x) {…}
   public static float cos(float x) {…}
}
```

```
…
Math mathobject = new Math();
mathobject.sqrt(9.0);
…
```

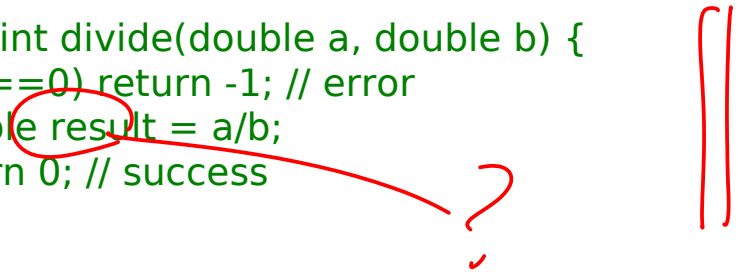
```
…
Math.sqrt(9.0);
…
```

Section: Exceptions

Error Handling

- You do a lot on this in your practicals, so we'll just touch on it here
- The traditional way to handle errors is to return a value that indicates success/failure/error

```
public int divide(double a, double b) {
   if (b==0) return -1; // error
   double result = a/b;
   return 0; // success
}

...

if ( divide(x,y)<0) System.out.println("Failure!!");
```



- Problems:
  - Could ignore the return value
  - Have to keep checking what the return values are meant to signify, etc.
  - The actual result often can't be returned in the same way

Exceptions I

- An exception is an object that can be *thrown* or *raised* by a method when an error occurs and *caught* by the calling code

Base class: Exception

```
public double divide(double a, double b)
                    throws DivideByZeroException {
   if (b==0) throw DivideByZeroException();
   else return a/b
}

...

try {
   double z = divide(x,y);
}
catch(DivideByZeroException d) {
   // Handle error here
}
```

Exceptions II

- Advantages:
  - Class name can be descriptive (no need to look up error codes)
  - Doesn't interrupt the natural flow of the code by requiring constant tests
  - The exception object itself can contain state that gives lots of detail on the error that caused the exception
  - Can't be ignored, only handled