Multiple Constructors

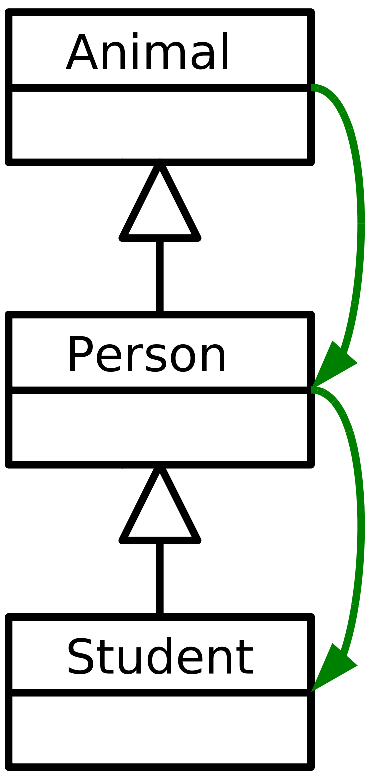
```
public class Student {
    private String mName;
    private int mScore;

    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)

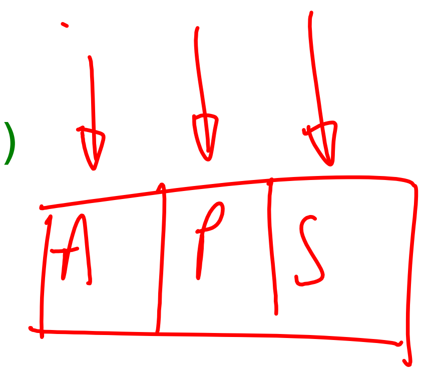
Constructor Chaining

- 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()

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

```
class FileReader {
  public:

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

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

  private :
    FILE *file;
}
```

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

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

  // Use object here
  ...

  // Destruct the object
  delete f;

}
```

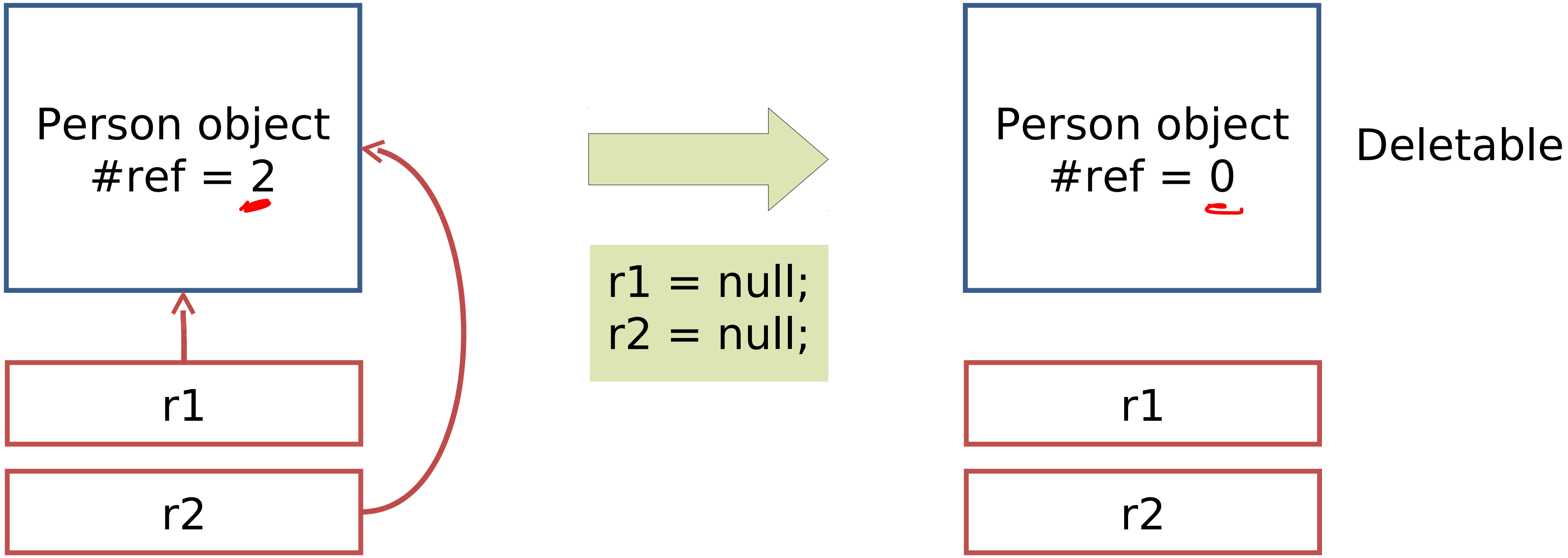
C++

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?
- **Approach 2:**
  - Delete the objects automatically (**Garbage collection**)
  - But how do you know when an object is finished with if the programmer doesn't explicitly tell you it is?

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

- Good:
  - System cleans up after us

- Bad:
  - It has to keep searching for objects with no references. This requires effort on the part of the CPU so it degrades performance.
  - We can't easily predict when an object will be deleted

Cleaning Up (Java) III

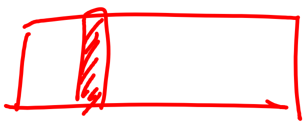
- So we can't tell when a destructor would run – so Java doesn't have them!!
- It does have the notion of a **finalizer** that gets run when an object is garbage collected
  - BUT there's no guarantee an object will <u>ever</u> get garbage collected in Java...
  - Garbage Collection != Destruction

Class-Level Data

Class-Level Data and Functionality I

```
public class ShopItem {
   private float price;
   private float VATRate = 0.175;

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

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

}
```

- Imagine we have a class ShopItem. Every ShopItem has an individual core price to which we need to add VAT
- Two issues here:
  1. If the VAT rate changes, we need to find every ShopItem object and run SetVATRate(...) on it. We could end up with different items having different VAT rates when they shouldn't...
  2. It is inefficient. Every time we create a new ShopItem object, we allocate another 32 bits of memory just to store exactly the same number!



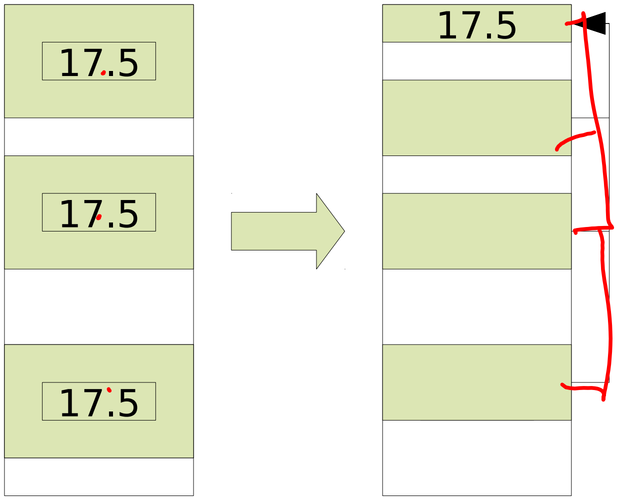
- What we have is a piece of information that is class-level not object level
  - Each individual object has the same value at all times
- We throw in the **static** keyword:

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

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

[Workbook 3]

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 this 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 Functions?

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

```
public static int do() {
  if (x) return 1
  else return 0
}
```

```
private static int x
```

```
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) {…}
}
```

create object →

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

```
…
Math.sqrt(9.0);
…
```

Exceptions

[Workbook 4]

Error Handling

- You do a lot on this in your practicals, so we'll just touch on it here
- The traditional way of handling 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 'codes' are for success, etc.
  - The result can't be returned in the usual way

Exceptions I

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

```
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
}
```

Exception

catch(Exception e) {

}

Exceptions II

- Advantages:
  - Class name is descriptive (no need to look up 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

Copying Java Objects

Immutable objects

Vector2D v = new Vector2D();

Vector2D v2 = v;