# COMP2511

## WEEK 2

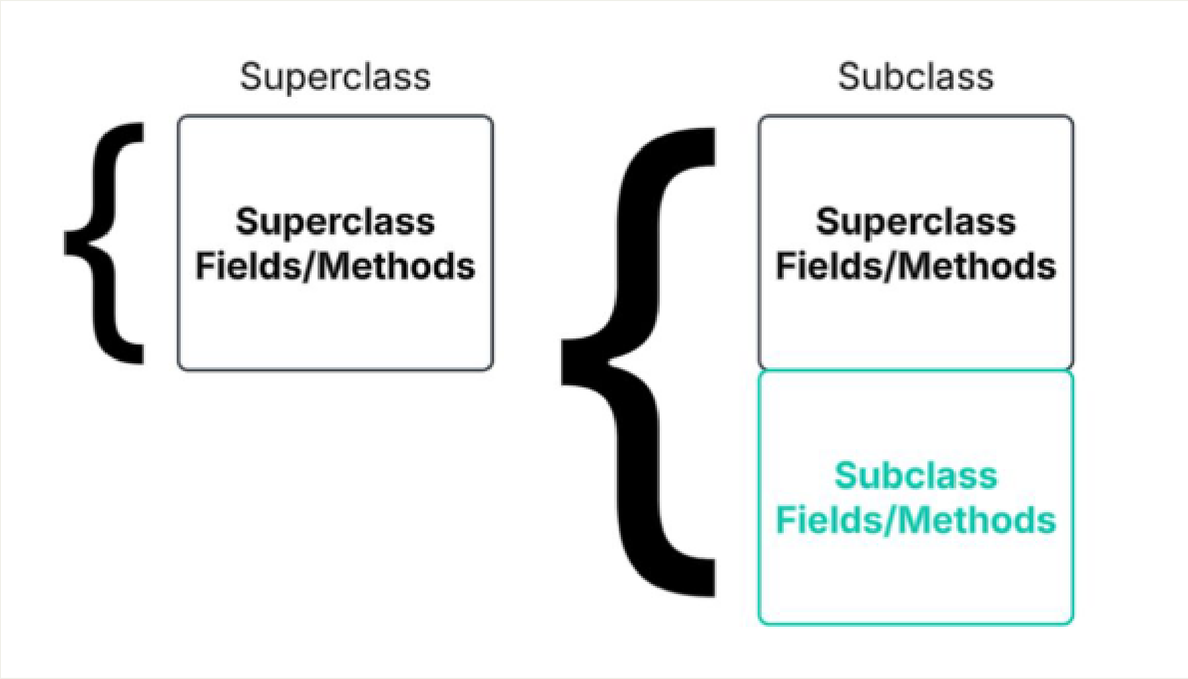
How did the first programmer program the first programming language for programmers to program programs?

# A G E N D A

- Announcement - Assignment I released,  due Week 5 Wednesday, 3pm.
- Code Review (super vs this)
- JavaDoc & Commenting
- Basic Inheritance & Polymorphism
- toString and equals
- Access Modifiers & Packages

# INHERITANCE

In Java, a class can inherit attributes and methods from another class. The class that inherits the properties is known as the sub-class or the child class. The class from which the properties are inherited is known as the superclass or the parent class.



# INHERITANCE

- Known as a "is-a" relationship
  - If B is a subclass of A, then an instance of B is also an instance of A.
  - E.g. a Cat is an Animal
- All classes in Java are subclasses of the Object class.
- In Java, the **extends** keyword makes a class inherit from another.

# CODE REVIEW

```java
public abstract class Shape {
    private String colour;

    private static int count = 0;

    public Shape(String colour) {
        System.out.println("Inside Shape constructor");
        this.colour = colour;
        count++;
    }

    public void setColour(String colour) {
        this.colour = colour;
    }

    public String getColour() {
        return colour;
    }

    public static int getCount() {
        return count;
    }

    public abstract int getArea();

}
```

```java
package shapes;

public class Rectangle extends Shape {
    public int height;
    public int width;

    public Rectangle(String color) {
        super(color);
        this.width = 1;
        this.height = 1;
        System.out.println("Inside Rectangle constructor with one argument");
    }

    public Rectangle(String color, int width, int height) {
        this(color);
        this.width = width;
        this.height = height;
        System.out.println("Inside Rectangle constructor with three arguments");
    }

    public int getArea() {
        return height * width;
    }

    public static void main(String[] args) {
        Rectangle r1 = new Rectangle("red", 10, 20); // What will this print?
        Rectangle r2 = new Square("blue", 20);
        System.out.println(r2.getArea());
        System.out.println(Shape.getCount());
    }
}
```

# SUPER VS THIS

1. What is the difference between super and this?
2. What about super(...) and this(...)?
3. What will the main code in Rectangle print and why?

# SUPER VS THIS

1. What is the difference between super and this?
- super refers to the immediate parent class whereas this refers to the current class

2. What about super(...) and this(...)?
- **super()** acts as a parent class constructor and should be the first line in a child class constructor
- **this()** acts as a current class constructor (can be used for method overloading)

3. What will the main code in Rectangle print and why?
- It will go into the Rectangle constructor with 3 arguments, which will then call the Rectangle constructor with 1 argument, which finally calls the Shape constructor

# ABSTRACT CLASSES VS INTERFAC ES

1. What is the purpose of using an abstract class in this code?
2. What are some downsides of the use of the abstract class here?
3. What is the difference between an abstract class and an interface? Why would you use one or the other?

# ABSTRACT CLASSES VS INTERFAC ES

1. What is the purpose of using an abstract class in this code?

- It allows the user to enforce the inclusion of a method getArea without having to define a default implementation, forcing subclasses to provide an implementation.

2. What are some downsides of the use of the abstract class here?

- Java forces us to only be able to inherit from one class (including abstract classes)

3. What is the difference between an abstract class and an interface? Why would you use one or the other?

- An interface cannot provide any implementation or fields - only methods which need to exist in classes which implement the interface. On the other hand, abstract classes can provide some implementations of methods and have fields, with the option to not provide implementation of some (or all) methods.

# STATIC KEYWORD

1. Can you override a static method?
2. What is the output of running r2.getArea() in main?
3. What is the output of running Shape.getCount() in main?

# STATIC KEYWORD

1. Can you override a static method?

- No - the method is attached to the class, not the instance.

2. What is the output of running r2.getArea() in main?

- It will print 400, making use of the Shape constructor as it always will use the runtime type of the object, not the pointer type's method

3. What is the output of running Shape.getCount() in main?

- It will print 2 as each constructor call updates the static count, which is shared across instances. Hence, the first one updates 0 to 1, and the second updates 1 to 2. This is in contrast to non-static variables which have different values across instances - in that case, the count would be 1 for the two separate instances.

# DOCUMENTATION

- Why is documentation important? When should you use it?
- What does the term "self-documenting" code mean?
- When can comments be bad (code smell)?

# D O C U M E N T A T I O N

- Why is documentation important? When should you use it?
- What does the term "self-documenting" code mean?
  - Code that documents itself. It is readable inherently. Usually accomplished through variable name and function names
- When can comments be bad (code smell)?
  - Comments become stale & does not get updated with new changes
  - Possibly hinting that your design/code is too complex

# DOCUMENTATION

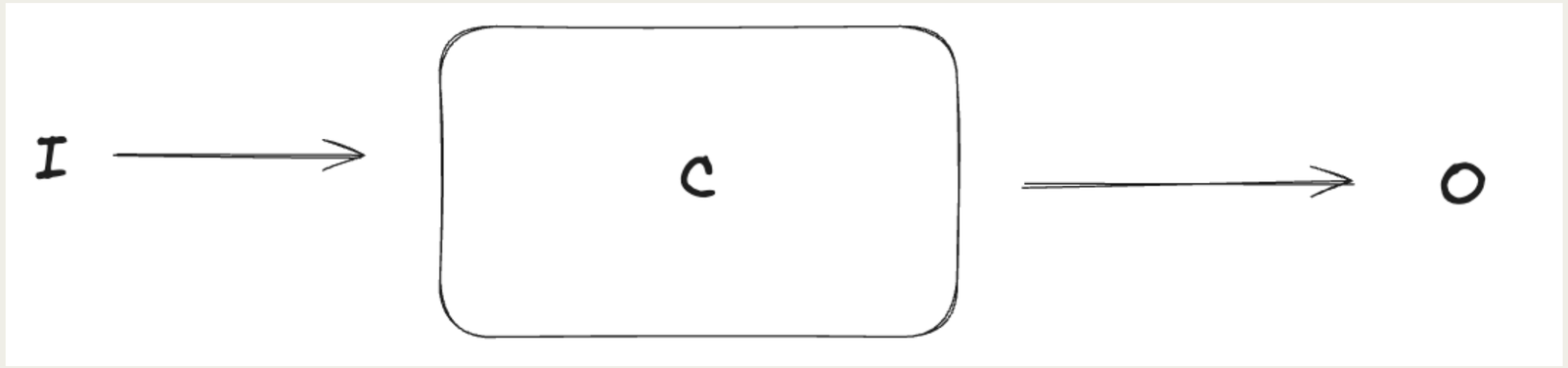
```
// Single line comment
```

```
/**
 * This is multi-line
 * documentation
 */
```

```
/**
 * Constructor used to create a file
 * @param fileName the name of the file
 * @param content contents of the file
 */
```

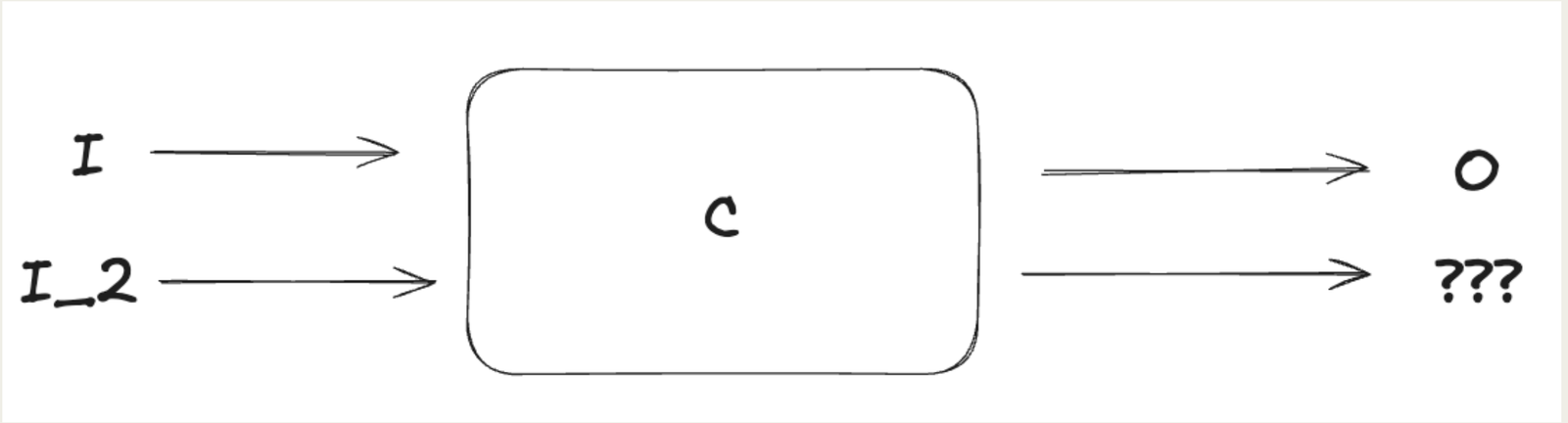
# POLYMORPHISM

- Derived from Greek, meaning “having many forms”
  - Poly - many, morphism - form
- Consider the diagram below, we have a method which takes an input, I and outputs O



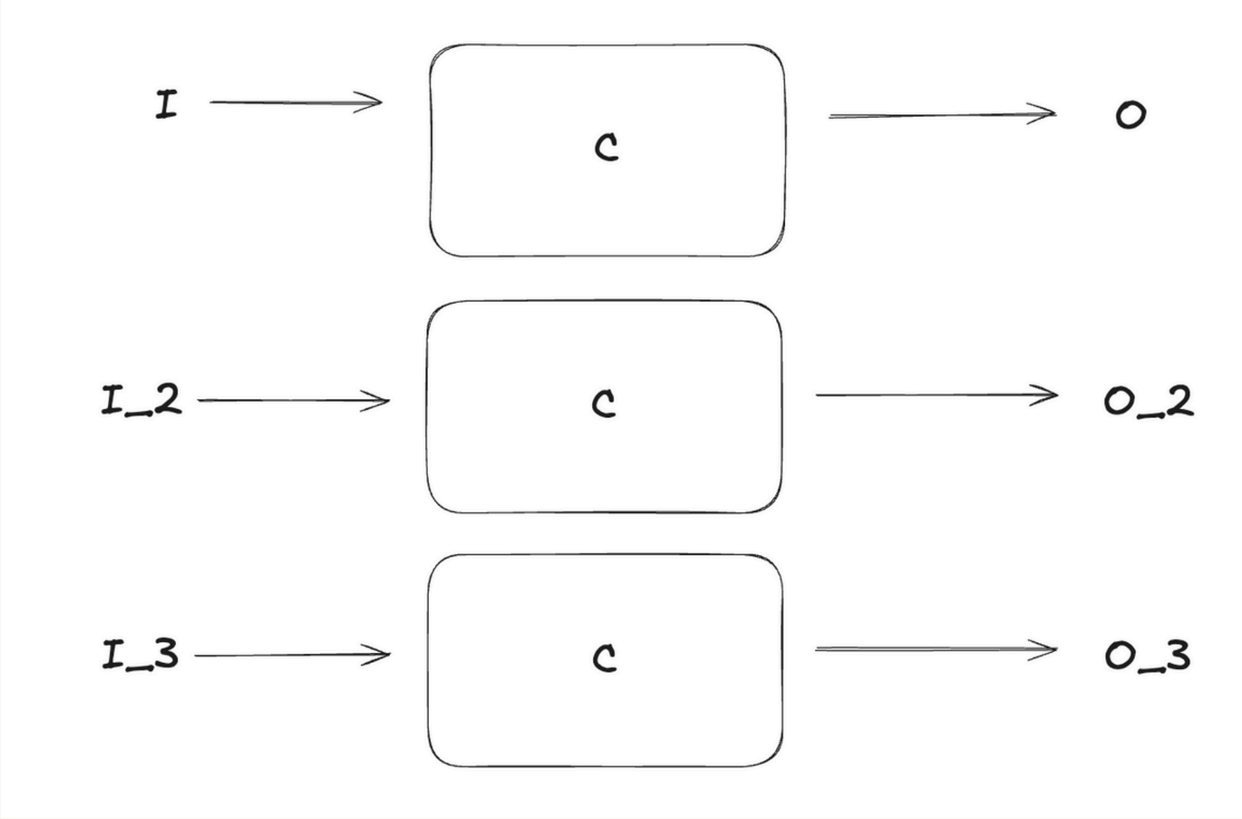
# POLYMORPHISM

- But what if we wanted more inputs?



# POLYMORPHISM

- We can just create more methods to do different things!
- And this is the basic idea behind polymorphism!



# POLYMORPHISM

```java
public class Animal {
    public void eat() {
        System.out.println("nom nom");
    }

    public static void main(String[] args) {
        Animal myAnimal = new Animal();
        myAnimal.eat();

        Dog myDog = new Dog();
        myDog.eat();

        Cat myCat = new Cat();
        myCat.eat();
    }
}
```

```java
public class Dog extends Animal {
    public void bark() {
        System.out.println("woof");
    }
}
```

```java
public class Cat extends Animal {
    public void eat() {
        System.out.println("meow");
    }
}
```

## CODE DEMO

1. Create a **Employee** class with a **name** and **salary**
2. Create setters & getters with JavaDoc
3. Create a **Manager** class that inherits **Employee** with a **hireDate**
4. Override **toString()** method
5. Write **equals()** method

# HOW MANY CONSTRUCTORS DOES A CLASS NEED?

Technically none. If a class is defined without a constructor, Java adds a default constructor.

However, if a class needs attributes to be assigned (e.g., has a salary), then a constructor must be assigned.

If your class has attributes with no default values, then the **constructor must set these attributes**. This is because variables with no values are dangerous (null), and is also the constructor responsibility.

Each class's constructor is also only responsible for setting **its own attributes**. Do not set the superclass's attributes within the subclasses without using a super(...) constructor call. (In fact super is implicitly called for all subclasses)

# toString()

The documentation for the toString() method states that it should return a string that "textually represents" the object. In this case, it should contain the name, salary and hire date (in the case of Manager), but also the runtime class of the object.

# EQUALS - MOTIVIATION

- What does the ‘==’ operator do when comparing objects?
- Where have you seen this sort of behaviour before in other languages? How is the underlying data checked for equality in that scenario?
- How can we compare two objects for equality?

# EQUALS

- Since we are overriding an existing method (in the super most class called Object), we must follow the conditions described.
- The conditions can be found in the Java Docs

# EQUALS

Typical Structure of the equals method will include:

- Check that the passed in object is not null.
  - if (object == null) return false
- Check if the passed in object is the same instance as the calling object.
  - if (this == object) return true
- Check the concrete type of the calling object matches the concrete type of the passed in object.
  - if (!this.getClass().equals(object.getClass())) return false
- Typecast and then check if all fields are equal

# EQUALS - WITH INHERITANCE

What is the relationship between a super type and a sub type in terms of equality? Can a concrete instance of an Employee be equal to an instance of a Manager

- No because of the .getclass() call which will check and compare that the classes are indeed different.

# EQUALS - WITH INHERITANCE

How can the method in Manager utilise code in Employee to avoid repetition?

- Obviously, we can use the super method!!!

# ACCESS MODIFIERS & PACKAGES

| | default | private | protected | public |
|---|---|---|---|---|
| Same Class | Yes | Yes | Yes | Yes |
| Same package subclass | Yes | No | Yes | Yes |
| Same package non-subclass | Yes | No | Yes | Yes |
| Different package subclass | No | No | Yes | Yes |
| Different package non-subclass | No | No | No | Yes |

# LABBING :eyes:

BRRRRR