# CMSC131

Inheritance

# Object

When we talked about **`Object`**, I mentioned that all Java classes are "built" on top of that.

This came up when talking about the Java standard equals operator:

```
boolean equals(Object obj);_
```

When this came up, I said we'd return to talk about it in more detail towards the end of the semester…

# Inheritance

- In OO langs a new class can **extend** an existing class.
- Every class that doesn't explicitly extend some other class will still extend **`Object`**.
- The existing class is called the *base* or *super* class.
- The new (typically called the *derived* or *sub*) class will inherit the static and instance variables and methods from the *base*.
- However, it might not have direct access to some of the things it inherits!
- Base references can be used to point to derived objects.

# Syntax

```
Public Class Base {
}


Public Class Derived Extends Base {
}
```

# `protected`

In addition to **`public`**, **`private`**, and the **default** levels, there is an additional one that was briefly mentioned called **`protected`**.

With this level, a **`protected`** variable or method can be accessed by the class or by any class derived from it or by other classes in the same package as the class.

# Access / Visibility Summary

| Modifier | Within Class | Within Package | Subclass (more Wed/Fri) | Outsiders |
|---|---|---|---|---|
| `public` | YES | YES | YES | YES |
| `protected` (more Wed/Fri) | YES | YES | YES | NO |
| `default` (no modifier) | YES | YES | NO | NO |
| `private` | YES | NO | NO | NO |

# Will it compile?

1. Yes
2. No

```
X myX = new X();
myX.inc();
```

# Example: InheritanceHitsAndMisses

What can and can't we access from where?

# More on Casting

You can always cast a derived class back down to its base class.

If you try to cast a base object into a derived class it will throw a runtime error.

- This is important if you have (for example) an array of base references but you think the object at the end of the reference is actually a derived one and you want to access something in that part of the object.
- We can (and probably should) test before casting so we can handle things ourselves.

Class X extends Class A
Class Y extends Class A

# Will this…

1. Not compile.
2. Compile but crash.
3. Compile and run fine.
4. No idea.

```
X var = new A();
```

Class X extends Class A
Class Y extends Class A

# Will this…

1. Not compile.
2. Compile but crash.
3. Compile and run fine.
4. No idea.

```
A var = new X();
```

Class X extends Class A
Class Y extends Class A

# Will this…

1. Not compile.
2. Compile but crash.
3. Compile and run fine.
4. No idea.

```
Y var = new X();
```

Class X extends Class A
Class Y extends Class A

# Will this…

1. Not compile.
2. Compile but crash.
3. Compile and run fine.
4. No idea.

```
X var1 = new X();
A var2 = (A)var1;
```

Class X extends Class A
Class Y extends Class A

# Will this…

1. Not compile.
2. Compile but crash.
3. Compile and run fine.
4. No idea.

```
A var1 = new A();
X var2 = (X)var1;
```

# instanceof

Before casting, you might want to test to make sure you are correct about the type.

```
if (objectRef instanceof DerivedName) {
   ((DerivedName)objectRef).funInDer();
}
```

# Overriding Methods

- It is possible for a derived class to **override** a method that has been defined and implemented in the base class.
- The overriding method needs to have the same name and parameter list and return type as the one in the base class.

An interesting issue arises if you have base reference pointing at a derived object if you then use that reference to invoke an overridden method.

  - Which version does it invoke?
  - It depends on the language!  In Java it will invoke the derived version.  In C++ it will invoke the base version.

# Some methods **`Object`** gives you…

```
boolean equals(Object);
Class getClass();
int hashCode();
String toString();
```

Also a variety of methods useful for multi-threaded programming…

# equals

The default **equals** inherited from **Object** checks to see if the two references are to the same exact object in memory.

The following is likely how it should be overridden…

```
public boolean equals (Object other) {
  if (other == null) {
    return false;
  }
  else if (this.getClass()!=other.getClass()) {
      return false;
  }
  else {
    //cast other and do real equals tests here
  }
}
```

# "Is A" versus "Has A"

Many of the classes we have written have "contained" other types.

We could say that a **`Restaurant`** "has a" **`String`** and "has a" **`SortedListOfImmutables`** as "has a" another **`SortedListOfImmutables`**.

This is referred to as class *composition*.

With inheritance if **D** is derived from **B**, we typically say that a **D** "is a" **B**.

# How to make a **`Stack`**?

In **Java**, rather than extend a list structure you might want to just use one (composition).

This way your Stack can have push() and pop() but not have the methods "visible" that it would inherit.

```
public class Stack<Type> {
   private ArrayList<Type> mydata;
   :
   :
   :
}
```

Some other languages have other options…

# Shadow Variables

One thing to be aware of is the names of any variables declared in the base class.

- If you declare a variable with the same name in the derived class then that is the one that is seen by default (though the one it inherited from the base is still inside the object too).
- Within a method of the derived class (but not in places in the rest of the project) you can access the base's version by using **`super.`** to refer to the variable from the base (also known as the super) class.

# Constructors

When a derived object's constructor is called, before the body of it is executed, the base class' constructor will be called.

If a derived class has a copy constructor, the base class' regular constructor will be called before the body of the derived class' copy constructor is executed unless the derived class' copy constructor explicitly calls the base copy constructor.

# Multiple Inheritance

Some languages that support polymorphism support multiple inheritance.

- Java is NOT one of those languages. Why?

Consider base classes: **Helicopter**, **Jet**

Consider a new class: **HeliJet**

- Could be good use of polymorphism.
- Could create challenges like naming conflicts if both bases classes have a matching field since they would be at the same "level".

# Multiple Interfaces

In the previous example, if **Helicopter** and **Jet** both defined a **`String`** name and **HeliJet** was built by extending both of them, you would have two different name objects.

What if **Helicopter** and **Jet** were interfaces and **HeliJet** implemented both? Is there any potential for conflict or confusion?