Today’s topic:

Using Inheritance
Inheritance is an "is-a" relationship.
- A dog "is-a" mammal.

Our base class (or super class) is a generalization.
- It contains attributes and behaviors that are common to all of the sub classes.

Our sub classes are specializations.
- They contain additional functionality that isn’t part of all of the objects from that base class.
- Sometimes they perform the same behaviors, but in a different manner.
Using inheritance in your design.
- Sometimes you will know in advance that you will use inheritance.
- Other times you may anticipate future inheritance.
- Even if you don’t use inheritance in your first iteration, it is a good idea to design with it in mind.

The book gives an example of a system for a veterinarian.
- You may not treat cats now, but sometime in the future you may want to.

Inheritance Relationship
- Used to represent that one class is subclass of another.
- The Dog and Cat classes are subclasses of Mammal.
OO Concept: Inheritance

Let’s look at a real world example:

- This is a knife.
  - It has a blade
  - It has a handle
  - It can cut

Let’s look at a real world example:

- This is a switchblade knife
  - It has a blade.
  - It has a handle.
  - It can cut.
  - It folds up.


Let’s look at a real world example:

This is a Swiss Army knife
- It has a blade.
- It has a handle.
- It can cut.
- It folds up.
- It has a can opener.
- It has a screwdriver.
- It has a bottle opener.
- It has a wire stripper.
- It has a leather punch.

Let’s look at a real world example:

The switch blade and Swiss Army are types of knives.

They share the basic features of a knife, but also have additional ones.
Some vocabulary:

- The base class is called the **super class** (i.e. knife).
- The **sub class** is the specialization (i.e. folding knife).
- The sub class **inherits** instance variables and methods from the super class.

**A UML representation:**

\[
\text{Knife} \quad \rightarrow \quad \text{FoldingKnife}
\]

Some vocabulary:

- You will find the term **polymorphism** used in different ways in OOP. We’ll talk about one way today.

- The word comes from **poly = many** and **morph = forms**.

- In **general** polymorphism, an objects of different subtypes can be used **in place** of each other in your code.
OO Concept: Inheritance - Example

- Suppose we have a class:
  - `Knife`
    ```java
    int bladeLength;
    public void cut() {...}
    ```

- The class `FoldingKnife` is a sub class of `Knife`. It inherits everything from `Knife`, and adds more:
  - `FoldingKnife`
    ```java
    int bladeLength;
    boolean open;
    
    public void cut() {...}
    public void openBlade() {...}
    ```

OO Concept: Inheritance - Example

- Anytime I need a knife, I don’t necessarily care what kind of knife, since all knives have a blade length and can cut.
  - If I create a method in my `Butcher` class that takes a knife as a parameter:
    ```java
    public void cutSteak( Knife myKnife)
    {
        myKnife.cut();
    }
    ```
  - Either of these would work when I call the method:
    ```java
    myButcher.cutSteak(new Knife());
    OR
    myButcher.cutSteak(new FoldingKnife());
    ```
The inverse is **not** true:

- If I create a method in my Butcher class that takes a knife as a parameter:
  ```java
  public void cutSteak(FoldingKnife myKnife)
  {
    myKnife.cut();
  }
  ```

- Now I must call the method:
  ```java
  myButcher.cutSteak(new FoldingKnife());
  ```

because **this won’t work**:
  ```java
  myButcher.cutSteak(new FoldingKnife());
  ```

---

We have already seen and used sub classes in our examples.

- In Java, **all** classes are a subclass of the base class `Object`.

- That’s why all objects have a `toString()` method. They inherit it from the class `Object`.

- You will also see that many methods take an `Object` as their `input parameter`. This means you can literally pass in any instance of any `Object`. 
In Java, the extends keyword is used to identify subclasses:

```java
public class FoldingKnife extends Knife {...}
```

Within my FoldingKnife class I do not need to replicate any instance variables or methods because they already are inherited.

I can add additional ones if needed.

**Rules for overriding a method**

- The argument list has to be the same as that of the overridden method.
- The return type has to be the same or a subtype.
- The access level cannot be more restrictive than the overridden method’s access level.
- A method declared final cannot be overridden.
- A method declared static cannot be overridden but can be re-declared.
- Constructors cannot be overridden.