Chapter 1001: Inheritance &Interfaces

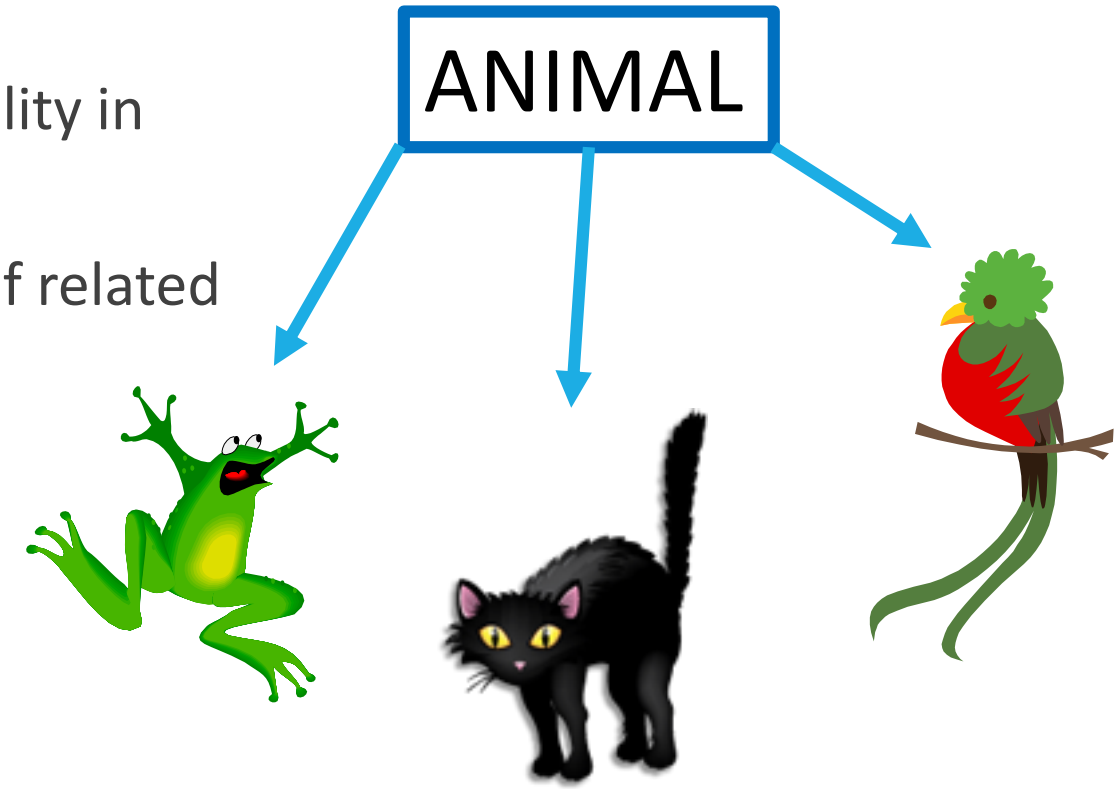
WEEK 1: SHARING CODE

BETWEEN CLASSES

(MAKE SURE IT'S ALL APPROPRIATELY SHARED!)

Inheritance Basics

- Extending a class and overriding methods
- Allows greater flexibility in reusing code
- Creates hierarchies of related object types



The Cat, Frog, and Bird classes are all *subclasses* that EXTEND the Animal *superclass*

"Is-a" Relationships

- Subclasses have an "is-a" relationship with the class they are derived from (the superclass)
- In the example, a *Cat* "is a" type of *Animal*
- Because of this relationship, a superclass contains attributes and methods that may be useful in the subclass
- ***EXTENDING*** the superclass allows the derived ("child" classes or "subclasses") to inherit all the state and behavior of the parent
  - **REMEMBER**: We want to MAXIMIZE code REUSE and AVOID code REDUNDANCY

What if I want my subclass to do something different?

- You CAN do this! -- **OVERRIDE** the inherited method
  - Implement a NEW version of the inherited method in the subclass
  - The new method MUST HAVE the SAME ***name*** and ***signature*** as the method inherited from the superclass

Animal	Cat	Frog	Bird
getOffspring()	getOffspring()	getOffspring()	getOffspring()
Returns “babies”	Returns “kittens”	Returns “tadpoles”	Returns “chicks”

Subclass Constructors

- **should *ALWAYS* be overridden!**
  - **ONLY** the ***LOCAL CONSTRUCTOR*** will know how to initialize ***LOCAL VARIABLES*** – these variables **DO NOT EXIST** in the parent class
    - To avoid redundancy, call the superclass constructor as a first step in the LOCAL constructor – do this using the keyword **"super"**

```
public class Cat extends Animal{
    private String makesSound;

    public Cat(int age) {
        super(age);
        makesSound = "purr";
    }
}
```

The Cat constructor doesn't need to include an "age" attribute – this, along with other attributes and methods that may exist, are inherited from the parent class and initialized in the local class by calling the superclass constructor. Only NEW attributes need to be listed locally.

Accessing Inherited Fields

- Although a subclass will inherit all the fields of its parent, it will **NOT** be able to access any inherited fields that are declared *private* in the parent class
- While declaring attributes in the superclass to be public might seem like an easy way around this issue, it would violate the **encapsulation** of the superclass and should be avoided
- To access these private fields, public mutator methods (**setters and getters**) need to be written into the superclass

The Object Class

- In Java, **ALL** classes are derived, either directly or indirectly, from the Object Class
- Thus, there are certain methods that are inherited by ALL objects – two important examples are listed below:
  - **toString()**
  - **equals(obj)**
- As you have probably discovered, these methods don't always behave the way you want them to
- Typically, you want to write new local versions (overriding methods) for the inherited classes you create to override the undesirable default behavior of the method

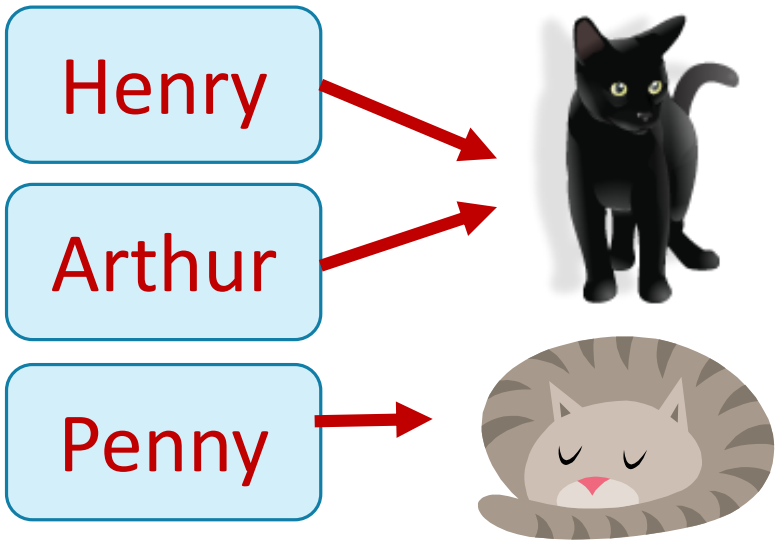
The equals(obj) method

- The equality operator (==) does NOT work when applied to objects – it only returns "true" when the two variables it is considering are holding the same object

3 variables hold Cat objects:

2 variables refer to a cat that has adopted two families – each of which have given him a different name.

BOTH cats are the same age – this is how you want to determine "equality"



The equals method would only return **true** when comparing Henry with Arthur

Equality, continued…

- By overriding the equals method, you can dictate how the comparison that determines equality is made
- One thing to keep in mind – your equals method should always take a parameter of type object
  - This creates a potential issue – even though all derived classes inherited, ultimately, from the Object Class, direct comparisons of the derived object with an object of type Object WON'T work
  - The Java compiler won't know ahead of time what type of "Object" will be passed in to the message, opening up the possibility that the object passed in won't contain the fields needed to perform the comparison

The “instanceof” keyword

- To avoid comparisons between 2 incomparable objects, the equals method should first check that the object it has been passed is of the correct type
- The **“instanceof”** keyword accomplishes this
- In addition, for the program to compile, the object passed into the equals method should be ***type cast*** to the appropriate type

```
public boolean equals(Object o){
    if (o instanceof Cat){
        Cat other = (Cat) o;
        return age == other.age;
    }
    else{
        return false;
    }
}
```

Polymorphism

- **POLYMORPHISM** is the ability of the same code to be used with objects of different types, producing different results.
- Polymorphism works because a superclass variable can be used to refer to an object of one of its subclasses.
- Ex: **Fruit gala = new Apple();**
- This is a legal assignment statement because Apple is a subclass that extends Fruit (and an Apple object is also a Fruit object)
- If methods are then called on gala, the version of the method local to the Apple class will be used

One caveat:

- When calling methods on subclass objects that are stored in superclass variables, the methods being called MUST exist in the superclass **AS WELL AS** in the subclass
- In other words, you CANNOT call a local subclass (i.e. unique to the Apple subclass) method on **gala** – you will get a compiler error because the compiler can't guarantee that the variable will ALWAYS refer to an object that has the method being called. The compiler only determines what kind of an object the variable is holding when it's actually running.

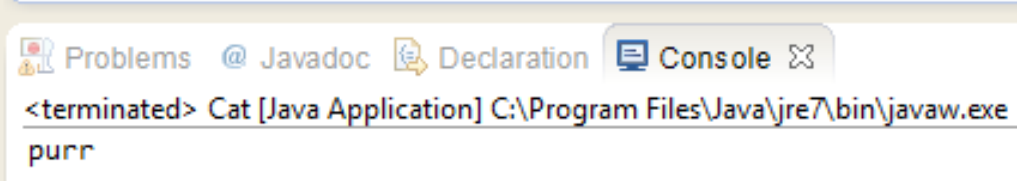
Yes, there's a way around this…

- To use a subclass method on an object you have instantiated using a superclass variable, you must first **cast** the variable into an appropriate type, thus **promising** the compiler that the object will be of the correct type for the method you intend to call.
- However, you may only cast a reference to a type one level above or below it

```
    public String makesSound(){
        return "purr";
    }

    public static void main(String[] args){

        Animal tom = new Cat(3);
        System.out.println(((Cat) tom).makesSound());
    }
}
```



As you can see – this code compiles and produces the desired result from the subclass method makesSound()

Interfaces

- Inheritance is useful as a way to share code, but in Java, you are limited to single inheritance – i.e. a class can only extend **ONE** superclass
- What can you do if you want to set up **MULTIPLE** "is-a" relationships?
- A feature called an **INTERFACE** can represent a common supertype between multiple classes without code sharing

Interfaces, continued…

- At first glance, an interface can look like a very odd construct. It consists **SOLELY** of a set of method declarations.
- When classes promise to ***implement*** an interface, those classes can be treated similarly in any code you write based on the interface.
- A class implements an interface by providing **ALL** of the methods listed in the interface, with **EXACTLY** the same ***names*** and ***signatures*** as are in the interface method declaration.

Example:
An Interface for Shapes

```
public interface Shape {

    public double getArea();

    public doube getPerimeter();

}
```

Any class that implements Shape **PROMISES** to provide a ***getArea()*** method and a ***getPerimeter()*** method

The methods in an interface (e.g. getArea() and getPerimeter(), in this case) are sometimes called **abtract methods**, because their implementations are not specified

How are interfaces used?

- Since interfaces contain ONLY method names and signatures, they CANNOT be instantiated – any code trying to create a new Shape() would NOT COMPILE
- It IS, however, possible to declare variables of type Shape, provided they refer to objects that implement the Shape interface
- For example, a Rectangle class could implement the Shape interface by declaring "implements Shape" in its class declaration:

```
public class Rectangle implements Shape  {
        private double width;
        private double height;
}
```

Implementing Interfaces

- The Rectangle class would then need to provide implementations of the methods getArea() and getPerimeter()
- Additional Shape subclasses could easily be added by implementing the interface and providing the specifiecd methods.
- Polymorphism can then be achieved without the code sharing that occurs in inheritance

Abstract Classes

- Abstract classes occupy turf halfway between fully concrete class implementations and fully abstract interfaces
- An **ABSTRACT CLASS** is a class that cannot itself be instantiated, but that exists instead as a superclass to hold common code and to declare abstract behavior.
- Not surprisingly, an abstract class is created by adding the word "abstract" to the class header. In addition to making the class non-instantiable, it allows the class to declare abstract methods without bodies
- **UNLIKE** interfaces, abstract classes MAY ALSO ***declare fields*** and ***implement methods*** WITH bodies that can be used by their subclasses

Abstract classes may implement interfaces

- When an abstract class implements an interface, its subclasses are committed to implementing the methods of the interface.
- Because the subclasses inherit from the abstract class, they themselves **DO NOT** have to declare that they implement the interface.
- Code in the abstract superclass may call any of the abstract methods contained in the superclass, even if they are not implemented in the superclass, because it can rely on the subclasses to implement these abstract methods.

In conclusion…

Taken together, inheritance, interfaces, and abstract classes provide programs (and programmers) with a lot of flexibility in how code is shared and organized.