+ The Diamond Problem

- So why can’t classes just extend as many classes as desired?
- The “diamond problem” is an ambiguity that arises in situations like the following:
  - Two classes B and C inherit from A
  - Class D inherits from both B and C
  - There is a method in A that B and C have overridden, but D does not override it
  - Which version of the method does D inherit: that of B, or that of C?
- The diamond problem goes away when there are no methods implementations to inherit!

+ Interfaces

- An interface is a “class-like” structure in Java that can be used to declare common functionality between classes
Question: Which class(es) would make the best pets?

- If you answered Dogs and RoboChickens, you’re right!
- Pets in our program should have a special method called “petMe()” that can be called to virtually interact with the pet.
- The problem here is that we don’t want to include that functionality on some of the less desirable pets in this hierarchy, such as Tigers and DestructoBoots.
- How should we go about formalizing that relationship in Java?
- One idea would be to have both Dog and RoboChicken have a common superclass, but that’s not possible in Java.
- Classes in Java can only inherit from a single superclass!

The answer is to build an “interface”:
- An interface is a “class-like” construct in Java that only contains abstract methods and constant data fields.
- Interfaces allow us to define common operating behavior that can be “implemented” by any class, regardless of its inheritance lineage.
- This means that we can “cut across” inheritance hierarchy to implement functionality as we see fit.
- They are great for modeling behaviors (like the ability to be petted!)
Creating an Interface

public interface Pet
{
    // only abstract methods & constants
    public abstract void petMe();
}

Interfaces

To associate a class with an interface we need to tell Java that the class will “implement” that interface.
The “implements” keyword goes after the class definition, like this:

public class Dog extends Animal implements Pet

Concrete classes that implement an interface MUST implement all its is methods in order to compile.

Interfaces & Inheritance

Remember! a critical difference between interfaces and abstract classes is that a class can implement many interfaces, whereas it can only extend one class!
If we want to get really fancy, we can implement an interface in an abstract class and provide some implementation (or none at all), leaving the rest for subclasses to implement.

Programming Example

- Write an interface called “Dangerous”
- The dangerous interface should have a single method called “warn()” that prints out a warning message that is unique to the type of danger that the object represents
- Tigers, Wolves and DestructoBots should implement the dangerous interface.

Examples:

   - Just like a superclass, interfaces can be used as a data type, allowing you to store previously unrelated objects in the same collection.
   - For example, RoboChickens and Dogs both implement the Pet interface. Therefore the following is valid:

```java
Pet[] thePets = new Pet[2];
thePets[0] = new RoboChicken("fred");
thePets[1] = new Dog("george");
```
Abstract Classes Vs Interfaces

- Abstract classes...
  - can have method implementations.
  - can have instance variables.
  - can have constructors.
  - classes can only 'extend' one abstract class
- Interfaces...
  - can only define method signatures.
  - can only define public static fields only.
  - cannot have constructors.
  - classes can 'implement' multiple interfaces.

How to Choose

- Remember the GeometricObject example?
  - A Circle 'is-a' GeometricObject and shares methods with it.
  - Dogs and RoboChickens 'can-behave-as' a pet-table thing
    - They share only the ability to be petted.
    - 'is-a' vs 'can-behave-as' - think about your classes and decide which of the relationships make sense.
- Abstract Classes are good for things which fit naturally into a hierarchy
- Interfaces are good for mixing together to create classes that have a common behaviors

Important Difference

- One obvious difference between the two is that abstract classes are permitted to contain implementations for some methods while interfaces are not.
- A more important difference is that to implement the type defined by an abstract class, a class must be a subclass of the abstract class.
- Any class that defines all of the required methods is permitted to implement an interface, regardless of where the class resides in the class hierarchy.
- Because Java permits only single inheritance with classes, abstract classes are severely constrained in their usefulness.

Advantages of Interfaces: Retrofitting

- Suppose I have this class representing a duck.
  - All ducks quack so no problem to have it as part of an abstract class.
  - But what if I want to have ducks fly?

Advantages of Interfaces: Retrofitting

- Not all ducks fly so adding a method to the abstract class doesn’t work.
  - Perfect use case for an interface, simple to retrofit.
  - Note that the Flyable could be used in things other than ducks!

Advantages of Interfaces: Mix-in Inheritance

- Interfaces are ideal for defining 'mix-ins'.
  - a ‘mix-in’ is a type that a class can implement in addition to its "primary type" to declare that it provides some optional behavior. (like we just saw with Duck and Flyable)
  - Example: Comparable is a 'mix-in’ interface that allows a class to declare that its instances are ordered with respect to other instances.
  - Example: Cloneable is a 'mix-in’ interface that allows a class to declare that it can be copied.
  - Abstract classes can’t be used to define mix-ins for the same reason that they can’t be retrofitted onto existing classes.
Advantages of Interfaces: Nonhierarchical Types

• Interfaces allow the construction of nonhierarchical type frameworks.
• As we’ve seen, type hierarchies are great for organizing some things, but other things don’t fall neatly into a rigid hierarchy.
• Would ‘Comparable’ make sense to be part of a hierarchy of classes?

Advantages of Interfaces: Nonhierarchical Types

• Suppose we have an interface representing a ‘singer’ and another representing a ‘songwriter’.
• Some singers are also songwriters.

Advantages of Abstract Classes: Class Evolution

• It is easier to evolve an abstract class than an interface. *
• If you want to add a new method to an abstract class, you can always add a concrete method containing a reasonable default implementation.
• Adding a method to an interface will break all existing classes that implement the interface.
• Classes that previously implemented the interface will be missing the new method and won’t compile.

Java 8

• That last slide is not exactly true anymore, but it’s good to understand that to understand the changes in Java 8 w.r.t. interfaces.
• Remember I said ‘interfaces only contain method declarations, no implementations!’. That’s not actually entirely true, at least anymore.
• Java 8, released in 2015, introduced many new features including ‘default’ and static methods for interfaces.

* not the whole story, but we’ll get there

The Comparable Interface
The Comparable Interface

- The Comparable interface is an interface that is part of the Java standard library.
- This interface imposes a total ordering on the objects of each class that implement it.
- This ordering is referred to as the class’s natural ordering.
- Classes can implement Comparable in order to allow for existing sorting algorithms implementations to work with your object.

The Comparable Interface

- The Comparable interface only has one method that you need to implement, named “compareTo()” - this method accepts a reference to another object and returns an integer.
  - It will return a positive int if the object in question is ‘greater than’ the one it is being compared to.
  - It will return a 0 if the objects are of ‘equal value’.
  - It will return a negative int if the object in question is ‘less than’ the one it is being compared to.