The final keyword and immutability
Using the `final` keyword on classes

- Often we want to be able to create subclasses based off of superclass definitions.

- However, there are times when we want to explicitly prevent a class from being extended.

- This can be enforced by declaring the class `final`

- Example:

  ```java
  public final class MyClass
  ```

- In the example above `MyClass` is at the end of the inheritance chain – no classes can extend it.
Why use the `final` keyword on classes?

- `String` in the Java Standard Library is in fact a final class for security and performance reasons.
  - It is often used to protect the internal state of immutable classes.
- The idea is to stop users of your class from doing things that could confuse themselves and others.
- Imagine a physics library where you have some defined constants or calculations abstracted into classes.
  - Unless marked `final`, a programmer could redefine basic calculations or constants that should *never* change.
Using the **final** keyword on methods

- The **final** keyword can also be used on methods within a class.
- Example:
  ```java
  public final void myMethod() {
  ... }
  ```
- Methods marked as final can not be overriden by subclasses.
Why use the `final` keyword on methods

- We might do this if our superclass has some algorithm that must not be modified by subclasses.

- For example, what would happen in the case here?

- We would be wise to mark reset as a `final` method.
The final keyword can also be applied to variables, like so:

```java
public final int numShoes = 100;
```

When used in this way, the final keyword will mark the variable in question as being read-only. It can no longer be changed by the current class or any sub-classes.

Note: If the variable is a reference, this means that the variable cannot be re-bound to reference another object. But the object that it references may still be mutable!

A static, final class variable is known as a constant. By convention we capitalize their names and use underscores.

```java
public final static int NUM_SHOES = 100;
```
Why use the `final` keyword on variables

- A `final` variable can only be initialized once, either via an initializer or an assignment statement.

- It does not need to be initialized at the point of declaration: this is called a "blank final" variable.

- A blank `final` instance variable of a class must be initialized in the constructor of the class in which it is declared.
Immutability

- You can use the `final` keyword to create your own immutable classes.

- An object is immutable if its state doesn’t change once the object has been created. Consequently, a class is immutable if its instances are immutable.

- Maximum reliance on immutable objects is widely accepted as a sound strategy for creating simple, reliable code.

- Immutable classes are easier to reason about!
Immutability

- How do you ensure that a class is immutable?
  - Don’t provide any public methods that modify the object’s state (no mutators)
  - Prevent the class from being extended. Make class `final`.
  - Make all fields `final` and `private`. Making fields private ensures that any mutable reference type such as `ArrayList` cannot have its contents changed by another class.
  - If the instance fields include references to mutable objects, don't allow those objects to be changed by other classes.
    - Don't provide methods that modify the mutable objects.
    - Don't share references to the mutable objects.
Programming Example

- See `FinalExample.java` for an example of `final`'s usage in all three ways.

- See `ImmutablePerson.java` to see an example of an immutable class.
Abstract Classes
Abstract Classes

- In the inheritance hierarchy, classes become more specific with each new sub-class.
  - Down the hierarchy classes represent more concrete concepts.
- And if you go up the hierarchy, classes become less specific.
  - Another way to say is that that become more conceptually abstract.
- An animal is an abstraction. A three-toed sloth is a very specific, ‘concrete’ example of an animal.
Abstract Classes

- An abstract class represents a class in the inheritance hierarchy that is so generalized that perhaps should not be able to create an instance.

- For example:

```java
// very general, abstract
public class Animal

// more specific, concrete
public class ThreeToedSloth
```
Abstract Classes

- Abstract classes in Java behave as follows:
  - Abstract classes are marked with the “abstract” keyword in their class definition, like this:

    ```java
    public abstract class Animal
    ```

  - Abstract classes cannot be instantiated using the “new” keyword
  - Abstract classes can contain a mix of concrete (implemented) methods and abstract (unimplemented) methods.
  - Unimplemented methods do not have a body associated with them – just a method header:

    ```java
    public abstract void sleep();
    public abstract void eat();
    ```
**Programming Example**

- Create an abstract class for a GeometricObject

- All geometric objects have a position \((x & y)\)

- All geometric objects have getters and setters for their position

- All geometric objects should be able to calculate their area and perimeter (but you can’t do that yet – this class is too abstract)

- See `abstractclasses/GeometricObject.java`
Extending Abstract Classes

- In order to create an instance of an Abstract class you must first extend it from a Concrete (non-abstract) class.
- Extending an abstract class forces as a concrete class forces the class subclass to implement all *abstract methods*.
- The compiler will complain if it do not.
- This can be done by overriding the abstract methods in the concrete class.
- Note that you can extend an abstract class with another abstract class though!
Programming Example

- Implement Circle, it should extend the Abstract class GeometricObject.
- Implement the Quadrilateral which should be an abstract class that extends GeometricObject and implements the features common to all Quadrilaterals. (do not implement getArea, though)
- Implement Rectangle it should extend the Abstract class Quadrilateral, and implement getArea.
- See the abstractclasses package
Why Abstract Classes?

- An abstract class helps a programmer to define a “base case” for a specific set of objects
- It forces the programmer to implement certain shared behavior among this group
- Note that abstract classes are also enable Polymorphism. For example, the following is valid and will work correctly due to dynamic dispatch:

```java
GeometricObject a = new Circle(1,1,5);
GeometricObject b = new Rectangle(5,6,10,2);

System.out.println( a.getArea() )
System.out.println( b.getArea() )
```
- Write a method called `getTotalArea` that can accept two of any `GeometricObject`
- Compute the total area and return it to the caller
- See `abstractclasses/Polymorphism.java`
Some Notes on Abstract Classes

- Abstract *methods* can only be contained in abstract *classes*.
- You can’t instantiate an abstract class, but you can define a constructor for one. If you do this then its subclasses may call this constructor via the `super()` keyword as usual.
- You can use an an abstract class as you would any other data type.
Interfaces
Interfaces

- An interface is a “class-like” structure in Java that can be used to declare common functionality between classes.
Interfaces

Animal (abstract)

name : String
speak() : void
eat() : void
Interfaces

- Animal (abstract)
  - name : String
  - speak() : void
  - eat() : void

- Tiger
- Wolf
Interfaces

```
interface Animal {
    String name;
    void speak();
    void eat();
}
```
Interfaces

Robot (abstract)

name : String
on : boolean;
speak() : void
turnOn() : void
turnOff() : void
Interfaces

Robot (abstract)
- name : String
- on : boolean;
- speak() : void
- turnOn() : void
- turnOff() : void

DestructoBot

RoboChicken
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 DestructoBots.

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!
Interfaces

- 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!)
Interfaces

- **Robot** (abstract)
  - name : String
  - on : boolean
  - speak() : void
  - turnOn() : void
  - turnOff() : void

- **Animal** (abstract)
  - name : String
  - speak() : void
  - eat() : void

- **DestructoBot**
- **RoboChicken**
- **Tiger**
- **Wolf**
- **Dog**

- **Pet Interface**
  - petMe() : void
Creating an Interface

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

// and since that's true, public and abstract // are redundant redundant
Creating an Interface

```java
public interface Pet {
    // only abstract methods & constants
    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:

```java
public class Dog extends Animal implements Pet
```
- Concrete classes that implement an interface MUST implement all its is methods in order to compile.
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.

See interfaces/petsdangerous package
Interfaces

- 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");
```
• 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 abstract class called Weapon that contains a name (String) and a method attack(). Weapon should implement the Dangerous interface.

- Then create a subclass of Weapon called Sword. Sword must implement all abstract methods in its superclass as well as all methods in the Danger interface.

- See interfaces/weapons package
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.
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 has 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!
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
The most obvious difference between the two mechanisms 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: Retrofitting

- Existing classes can be easily retrofitted to implement a new interface.
  - Simply add the required methods and an implements clause to the class declaration.
  - In order for two classes extend the same abstract class, you must place the abstract class high up in the type hierarchy where it subclasses an ancestor of both classes.
  - This causes great collateral damage to the type hierarchy, forcing all descendants of the common ancestor to extend the new abstract class whether or not it is appropriate for them to do so.
Advantages: 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: 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: Mixins

- Interfaces are ideal for defining ‘mixins’.
  - A ‘mixin’ 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)
  - Another example, Comparable is a ‘mixin’ interface that allows a class to declare that its instances are ordered with respect to other instances. (We’ll look at this in detail in a moment.)
- Abstract classes can’t be used to define mixins for the same reason that they can’t be retrofitted onto existing classes.
Advantages: Nonhierarchical Types

- Interfaces allow the construction of nonhierarchical type frameworks.
- Type hierarchies are great for organizing some things, but other things don’t fall neatly into a rigid hierarchy.
Advantages: Nonhierarchical Types

- Suppose we have an interface representing a ‘singer’ and another representing a ‘songwriter’.

- Some singers are also songwriters.

```java
public interface Singer {
    AudioClip sing(Song s);
}

public interface Songwriter {
    Song compose(boolean hit);
}
```
Advantages: Nonhierarchical Types

- Because we used interfaces, it is permissible for a single class to implement both Singer and Songwriter.

- In fact, we can define a third interface that extends both Singer and Songwriter and adds new methods that are appropriate to the combination.

```java
public interface Singer {
    AudioClip sing(Song s);
}

public interface Songwriter {
    Song compose(boolean hit);
}

public interface SingerSongwriter extends Singer, Songwriter {
    AudioClip strum();
    void actSensitive();
}
```
Advantages: Nonhierarchical Types

- You don’t always need this level of flexibility, but when you do, interfaces are a lifesaver.
- The alternative is a bloated class hierarchy containing a separate class for every supported combination of attributes.
- If there are n attributes in the type system, there are ~2^n possible combinations that you may have to support.
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.)

* not the whole story, but we’ll get there
Java 8

- That last slide is not exactly true anymore.. but its 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!’”. Thats not actually entirely true, at least anymore.
- Java 8, released in 2015, introduced many new features including ‘default’ and static methods for interfaces.
**Default Methods**

- A default method is an instance method defined in an interface whose method header begins with the default keyword.
- It also provides an implementation.
- Every class that implements the interface inherits the interface's default methods and can override them.

```java
public interface Addressable {
    String getStreet();
    String getCity();

default String getFullAddress() {
    return getStreet() + "", " + getCity();
}
```
Default Methods

- So what happens in this case?
- Compile error!
- Still no ‘Diamond Problem’
- No state
- No method ambiguity
Default Method Use Case

- Evolving existing interfaces…
  - As we talked about earlier, one problem with interfaces has historically been evolving them and breaking existing implementations.
  - Default methods enable you to add new functionality to the interfaces of your libraries and ensure binary compatibility with code written for older versions of those interfaces.
  - With default methods, a major drawback of interfaces is now eliminated, as they behave like abstract classes in this case.
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.
- It has one method, compareTo.
- Classes can implement compareTo in order to allow for existing sorting algorithm implementations to work with your object.
- Polymorphism FTW!
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 1 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 -1 if the object in question is ‘less than’ the one it is being compared to
Programming Example

- Write a class called “Pokemon” that stores a name (String) and a hitPoint value (integer)
- Implement the Comparable interface
- See comparable/Pokemon.java
The Comparable Interface

- Consider this syntax:

  ```java
  public class Pokemon implements Comparable<Pokemon>
  ```

- Note the generic modifier – this tells Java that Pokemon can only be compared to other Pokemon. You could open this up and make it so that Pokemon could be compared to ANY object, like this:

  ```java
  public class Pokemon implements Comparable
  ```

- However, this is probably not what you want to do since it would be impossible to compare the size of a Pokemon to an unrelated object, like a Dog, String or ArrayList.
Using the Comparable interface

- If an object is comparable then you can invoke the following method call to compare its size against a qualifying object:

```java
Pokemon a = new Pokemon("Pikachu", 100);
Pokemon b = new Pokemon("Charmander", 50);
println("Pikachu>Charmander?" + a.compareTo(b));
```
Using the Comparable Interface

- This is useful for comparing two objects, but what about hundreds or thousands of them?
- The comparable interface is used throughout Java to support sorting operations.
- For example, you can sort an array if it is storing data of type “int” because Java can easily compare the relative size of one element to another:

```java
int[] myList = new int[10];
// fill array with random numbers
java.util.Arrays.sort(myList);
// now it’s sorted!
```
But how can you sort Pokemon?

The answer is in the Comparable interface!

If you implement this interface you can use Java’s built-in sort functionality to perform the sort without doing anything special!

```java
Pokemon[] myPokey = new Pokemon[5];

// randomly create 5 pokemon

// sort the Pokemon!
java.util.Arrays.sort(myPokey);
```
Using the Comparable Interface

- Note that you can also sort ArrayLists this way if the object they are storing implements the Comparable interface.
- However, you wouldn’t use java.util.Arrays.sort() to sort an ArrayList because it isn’t an Array.
- Instead, use this syntax:

  ```java
  Collections.sort(myArrayList);
  ```
  
- Note that the above line will only work if myArrayList is storing an object that implements the Comparable interface.
For an example of sorting arrays and ArrayLists of Pokemon leveraging Comparable.

See comparable/TestComparable.java