CSC 142

Abstract classes and interfaces

protected keyword

- protected members are visible to
  - any class within the same package
  - any subclass even if it is not in the same package

```
// file B.java
package com.javaorbust;
public class B {protected int i;}
// file D.java
import com.javaorbust.B;
public class D extends B{
  public void update(){ i=6; /* OK */}}
```


Visibility summary

Modifier	Visibility
private	Class only
none (default)	Classes in the package
protected	Classes in package and subclasses inside or outside the package
public	All classes

Abstract classes

- Some classes are so abstract that instances of them shouldn't even exist
  - What does it mean to have an instance of **`Animal`**?
- An *abstract class* is one that should not or can not be instantiated .
- ≠ A *concrete class* can have instances
- It may not make sense to attempt to fully implement all methods in an abstract class
  - What should **`Animal.speak()`** do?

abstract keyword

- declare a method with the abstract modifier to indicate that it just a prototype. It is not implemented.

```
public abstract void speak();
```

- A class that contains an abstract method must be declared abstract

```
public abstract class Animal{
    public abstract void speak();
     // more code
  }
```


Using abstract classes

- An abstract class can't be instantiated.
- An abstract class can contain other non abstract methods and ordinary variables
- To use it, subclass it. Implement the abstract methods in the subclass
- If a subclass doesn't implement all of the superclass abstract methods, the subclass is also abstract and must be declared as such.
- Abstract classes provides a framework to be filled in by the implementor
  - Hierarchy: Shape(abstract) →Triangle, Rectangle, Circle

Abstract class example

```
public abstract class Accommodation{
  protected boolean vacancy;
  protected int NumberOfRooms;
   public abstract void reserveRoom();
   public abstract void checkIn();
   // etc...
}
public class Motel extends Accommodation{
   //must implement all of the abstract
   //methods of Accommodation
   //(if we want the class to be instantiated)
   //code would follow
}
```


Interfaces

- An interface is a purely abstract class
- An interface specifies a set of methods that a class must implement (unless the class is abstract)
- Everything inside an interface is implicitly public

```
public interface Driveable{
  // methods are always public (even if
  // public is omitted)
  // using abstract is optional
  boolean startEngine();
  void stopEngine();
  boolean turn(Direction dir);
 }
```


Using interfaces (1)

- An interface defines some set of behavior for an object. Think of an interface as a badge that can be worn by a class to say "I can do that".

```
public class Automobile implements Driveable {
    // implements the methods of Driveable
    public boolean startEngine()
    { if (notTooCold) engineRunning = true;
      // more code
    }
    // other methods
   }
```


Using interfaces (2)

- Interface types act like class types.
  - Variables can be of an interface type
  - formal parameters can be of an interface type
  - A method return type can be an interface type
  - Any object that implements the interface can fill that spot.
- A class can implement as many interfaces as desired

```
public class C extends B implements I1, I2, I3
  { /* class code */}
```

- This is how Java deals with multiple inheritance (≠ C++)

Interface variables

- An interface can contain constants (static final variables)

```
   public interface Scaleable
   {
    //static final is implicit and can be
    //omitted
     static final int BIG=0, MEDIUM=1, SMALL=2;
     void setScale(int size);
     }
```


subinterfaces

- An interface can extend another interface, e.g.

```
public interface DynamicallyScaleable
  extends Scaleable{
    void changeScale(int size);
  }
```

- A class that implements a subinterface must implement all of the methods in the interfaces of the hierarchy.
- An interface can extend any number of interfaces

```
public interface I extends I1, I2 {
     /* interface code */ }
```