MORE ON INHERITANCE

• Last time we introduced the concept of inheritance, and creating sub-classes that added specialized functionality.

• When we need to add new functionality to a class we some options:
  1. Create a new class that replicates the old functionality as well as the new features.
  2. Add the new functionality (variables & methods) to the class itself.
  3. Create a new subclass that extends the original class functionality.

• Typically the option 3 has the most advantages.
MORE ON INHERITANCE

• Issues with creating a new class that replicates the old functionality as well as the new features:
  
  • Requires you to re-implement existing methods.
  
  • Introduces duplication of code that needs to be maintained in parallel.

MORE ON INHERITANCE

• Potential issues with updating the existing class:
  
  • Changing the code in the existing class requires recompiling the class, and any packages or applications that use that class.
  
  • Adding the new functionality to the existing class means that every instance of the class has the new feature, even if it doesn’t need it.
  
  • It may require creating and maintaining multiple versions of the same methods, which can lead to complexity.
MORE ON INHERITANCE

Benefits of creating a new sub class:

- Inheritance allows for the creation of a new class of objects that behave like some other objects without duplicating code from the original class.
- The new class allows for using different instances of different classes in different applications.
- Eliminates the chance of introducing new bugs into the already functional class.
- Does not require modifying existing programs that can still use instances of the original class, that don’t need instances of the new class.
- By taking advantage of polymorphism, we can interchange our objects in certain situations to provide additional flexibility.

DESIGNING A SUBCLASS

- Questions to ask yourself:
  - What new instance variable(s) do I need?
  - What new methods do I need to implement?
  - What existing methods do I need to override?

- Remember:
  - You can call the super class’ methods from the sub class using the super keyword.
  - You need to implement a constructor for each one inherited from the super class.
  - If you need to access private instance variables in the super class you must either:
    1) change the variable to protected or
    2) write a protected accessor method to get the data.

(Try not to need to do this).
GENERALIZATION & SPECIALIZATION

• Recap:
  – Inheritance allows for creating a hierarchy of classes where all of the general behaviors are pulled out into the super class.

  – Each subclass then adds additional specializations:
    – New attributes
    – New methods
    – Overriding existing methods

MAKING THINGS FINAL

• In some cases we want to prevent a subclass from overriding a method.
  • To do this we declare the method as final.
  • For example, we may not want debit or credit methods to be changed outside of our BankAccount class for security reasons.

• In other cases we may declare an entire class as final.
  • This will prevent any subclasses to be made from the class.
AN EXAMPLE

• Now on to a new example
• Suppose we wanted to create a class Animal are animals with have the following attributes:
  • name
  • Age

• We want our animals to be able to respond to the following messages:
  • What is your name?
  • How old are you?
  • Speak.

AN EXAMPLE

• So we implement the methods:
  • public String getName() – returns its name.
  • public int getAge() – returns its age.

• How do we implement public String speak() in the Animal class?

  It depends on the what type of animal it is.
ABSTRACT CLASSES & METHODS

• There are situations that you may want to define class and its interface, *without* providing a *complete implementation* of each and every method in the class.

• We use the *abstract* keyword to do this.

• This ensures that any subclasses are *required* to be implemented in the subclass, but *not* in the super class.

ABSTRACT CLASSES & METHODS

• Because abstract classes are *not* fully implemented, and have no constructors you cannot *instantiate* them.

• The purpose is to create *subclasses* that *fully* implement the *general* behaviors.

• This allows for the use of polymorphism.
AN EXAMPLE:

public abstract class Animal{
    private String name;
    private int age;
    public abstract String speak();
    public String getName(){
        return name;
    }
    public int getAge(){
        return age;
    }
    protected void setName(String aName){
        name = aName;
    }
    protected void setAge(int aAge){
        age = aAge;
    }
}

AN EXAMPLE OF A SUBCLASS:

public class Cat extends Animal{
    public Cat(String itsName, int itsAge) {
        setName(itsName);
        setAge(itsAge);
    }
    public String speak(){
        return "Meow";
    }
}
BOOK EXAMPLE

• The example in the book uses a *Shape* class.
  – In this example all Shapes are expected to be able to *draw* themselves.
    (pp 145-147)

• He creates an abstract class *Shape*, with an *abstract* draw method.

CONTRACT

• The *public interface* is defined within the API.

• The public interface includes all method signatures that *must* be *implemented* in the class.

• When using an *abstract* class, the methods in the interface are identified and defines what *another* developer is responsible for when *implementing* a subclass.
ABSTRACT CLASSES IN UML

- Abstract classes and methods can be identified in UML by italics.

In summation...

- Abstract classes by design take advantage of inheritance and polymorphism.
- We will implement the general behaviors that are in common with all subclasses in the abstract class.
- We identify the methods that subclasses must implement, but leave that specialized implementation to be completed in the subclasses.