Lesson 14:
Abstract Classes and Interfaces
March 6, 2012
Not all Classes do Objects Make
Preventing a Class from Being Instantiated

```java
public abstract class Player {

    private String sport, team, position, first, last;

    public Player() { }

    public Player(String sp, String t, String pos, String f_name, String l_name) {
        sport = sp;
        team = t;
        position = pos;
        first = f_name;
        last = l_name;
    }

    public String toString() {
        return first + " " + last + " " + sport;
    }
}
```
Facts about Abstract Classes

- An abstract class can not be instantiated. If you try to create a `Player` object:

  ```java
  Player p = new Player();
  ```

  you will get a compiler error that reads something like this: `Player is abstract; cannot be instantiated`

- Even though you can’t instantiate a `Player`, you can still declare a `Player` reference:

  ```java
  public void displayPlayers() {
      for(Player p: players) {
          System.out.println(p);
      }
  }
  ```
Abstract Methods

Employee
- name: String
- hireYear: int
- monthlyPay()
- annualPay()

SalariedEmployee
- annualSalary: int
- monthlyPay()

Manager
- annualPay()

Staff
- annualPay()

FullTime
- annualPay()

PartTime
- annualPay()
Abstract Methods (cont.)

public abstract class Employee {
  private String name;
  private int hireYear;

  public Employee(String n, int hy) {
    name = n;
    hireYear = hy;
  }

  public abstract double monthlyPay();
  public abstract double annualPay();

  public int getYear() {
    return hireYear;
  }

  public String getName() {
    return name;
  }
}

If Employee is abstract, why does it have a constructor?
When you declare a method abstract, you don’t provide the body of the method.

When you declare a method abstract, you don’t provide the body of the method.
Abstract or Concrete?

- Since class Employee is only intended as a starting point for creating new classes and is declared Abstract, its monthlyPay method will not be implemented.
- What about SalariedEmployee and HourlyEmployee? You can provide a meaningful implementation for the monthlyPay method for both classes;
- Can they be instantiated? Why or Why not?
Terminology Update

- Abstract Class - class which cannot be instantiated or a method which does not have a body
- Concrete Class - class which can be instantiated or a method which does have a body
- Concrete is NOT a keyword as is abstract; it is simply a term used to differentiate between abstract and non-abstract classes and methods
public abstract class SalariedEmployee extends Employee {
    private int annualSalary;
    public SalariedEmployee(String name, int year, int annualSal) {
        super(name, year);
        annualSalary = annualSal;
    }
    public double monthlyPay() {
        return annualSalary/12.0;
    }
    public double getAnnualSalary() {
        return annualSalary;
    }
}
Bottom Line w.r.t. Abstract Classes

- Abstract classes contain:
  - data members/attributes
  - concrete methods/behaviors
  - abstract methods/behaviors

- **Can not create** an object from an abstract class

- **May declare reference** variables of the abstract type

- **If method** declared **abstract**, **containing class** must also be **abstract**

- Subclasses of an abstract superclass must override all of the abstract methods in the superclass (either through inheritance or directly in the subclass), otherwise subclass must be declared abstract.
Abstract Classes vs. Interfaces

Separating “what” from the “how”

- Abstract Classes began separation process
  - In Employee example, abstract methods used to establish requirements for what behaviors must be implemented (annualPay and monthlyPay)
  - how behavior implemented left to the classes that extended from Employee

- Abstract Classes don’t complete the process
  - Abstract Classes let you mix undefined methods (abstract methods) with fully defined methods (concrete methods)
  - Example: SalariedEmployee - it is abstract yet method monthlyPay is fully defined.

- Separation process completed with introduction of Interfaces
  - Specify set of required behaviors (methods) classes must provide
  - How they are provided (method implementation) not allowed
Interfaces

- Conceptually, can be thought of as “Abstract Class” without concrete methods.
- When a class implements an interface, class agrees to implement all methods of the interface.
- Interface doesn’t care how method is implemented, just that it is done.
- Interfaces are not actually classes; they are a set of requirements where the requirements are a set of required behaviors (methods).
  - Provide a way of describing what classes should do without specifying how. You can think of it as a purely abstract class, no concrete methods are allowed.
Interfaces- Example

- Suppose we have a program for managing store merchandise. The store sells:
  - **Items**, each of which has the attributes:
    - description
    - price
  - Items are either:
    - **Food**: with an attribute **calories**
    - **Clothing**: with an attribute **size**
    - **Book**: with an attribute **author**.
  - Of these items, clothing and books are taxable, but food is not
Interfaces- Example (cont.)

- Since many other things are taxable, such as services or entertainment, we want to have the concept "taxable" as a separate concept, not part of the concept Items.

- The concept **Taxable** could then consist of:
  - a **taxRate** of 5 percent
  - a **calculateTax( )** method

- Because Taxable represents a **required behavior** that an item must include and not a different kind of item, we represent it as an **interface** in Java
In the diagram, solid lines represent inheritance and dashed line represents class implementing the interface.

Similar to inheritance, “is a” relationship exists between a class and its implemented interface;

Example, **Clothing** “is a” **Taxable** object and **Book** “is a” **Taxable** object.
Taxable Interface Definition

```java
interface Taxable {
    double RATE = .05;
    double calculateTax();
}
```

Notes:
- Interfaces are always public so you don’t have to explicitly label them as public
- Methods are always public as well
- The **ONLY** data elements you can include in an interface are constants. By default the data is `final` as well as `public` and `static`. Therefore, declaration:
  ```java
double RATE
  ```
  above is in essence,
  ```java
public final static double RATE
  ```
Implementing an Interface

Two steps to interface implementation:

- Class must declare its intention to implement the interface using `implements` keyword
- Class must implement all of the methods (requirements) described in the interface
Implementing *Taxable* Interface

Here’s how the **Book** class would implement the **Taxable** interface:

```java
public class Book extends Item implements Taxable {

    public String author;

    public Book(String author, double price, String desc) {
        super (price, desc);
        this.author = author;
    }

    public double calculateTax() {
        return Taxable.RATE*this.getPrice();
    }
}
```
public static void main(String[] args) {
    Clothing c = new Clothing("Med", 15.00, "t-shirt");
    Food f = new Food(350, .70, "Snickers");
    Book b = new Book("Horstmann", 49.99, "Core Java 2");

    ArrayList<Taxable> stuff = new ArrayList<Taxable>() ;
    stuff.add(c);
    stuff.add(b);

    double taxTotal = TaxCalculator.totalTaxes(stuff); 

    System.out.println("Total taxes: " +
        String.format("%.2f", taxTotal));}

Another Taxable (or taxing?) Example
Another *Taxable* (or taxing?) Example (cont.)

- The `totalTaxes` method in the `TaxCalculator` class takes an `ArrayList<Taxable>` as a parameter and returns the sum of the calculated taxes.

```java
public class TaxCalculator {

  public static double totalTaxes(ArrayList<Taxable> taxableStuff) {
    double taxTotal = 0.0;
    for (Taxable taxItem: taxableStuff) {
      taxTotal = taxItem.calculateTax() + taxTotal;
    }
    return taxTotal;
  }

}
```
Another Use for Interfaces: Multiple Inheritance

From: http://cs.smu.ca/~porter/csc/342/notes/oop_in_cpp2.html
Another Use for Interfaces: Multiple Inheritance (cont.)

From: http://cs.smu.ca/~porter/csc/342/notes/oop_in_cpp2.html
Multiple Inheritance—“The Deadly Diamond of Death”

From: http://cs.smu.ca/~porter/csc/342/notes/oop_in_cpp2.html
Multiple Inheritance Resolved- Interfaces
In-Class Exercise

Consider the following declarations:
Clothing clothes = new Clothing("Med", 15.00, "t-shirt");
Food candy = new Food(350, .70, "Snickers");
Book coreJava = new Book("Horstmann", 49.99, "Core Java2");
Taxable taxableItem;
Item myItem;
taxableItem = clothes; legal / error
myItem = clothes; legal / error
Clothing c = taxableItem; legal / error
Taxable t = coreJava; legal / error
myItem = coreJava; legal / error
System.out.println("coreJava tax is " + myItem.calculateTax()); legal / error
taxableItem = myItem; legal / error
Taxable t1 = new Taxable(); legal / error
Summary

- Preventing a Class from Being Instantiated- Abstract Classes
- Abstract Methods
- Abstract vs. Concrete Classes and Methods
- Interfaces
- Multiple Inheritance