Inheritance and Polymorphism Part Two

Class Core Components

- Contain:
  - private instance variables
  - constructors
  - public getters and setters (with appropriate validity checking)
  - toString method (overridden)
  - equals method (overridden)
  - public and private class-specific methods

Inheritance

- Child class *is a* (specific kind) of the parent class.
- Child class inherits variables and methods.
- Child class can override methods to change or add on to the functionality.
- Use super to invoke the parent constructor or parent method.

Polymorphism

- The *declared type* defines what methods *are allowed to be* invoked at compile time.
- The *actual type* defines what *version* of a method is invoked at run time.
- Use instanceof and downcasting to temporarily treat an object of one declared type as another.

ABSTRACT CLASSES

Abstract Classes

- An *abstract class* is a placeholder in a class hierarchy that represents a general concept.
- Abstract classes allow you to establish common elements in a hierarchy.
- An abstract class represents an object that is too generic to instantiate.
  - An abstract class *cannot* be instantiated.

Abstract Methods

- An abstract class can (and often does) contain abstract methods.
  - An abstract class is not *required* to have abstract methods.
- An *abstract method* is a method header without a method body (no implementation).
- The `abstract` modifier is included in the method header
  - An abstract method cannot be `final` or `static`

Abstract Methods

- Abstract methods represent behavior that all objects have, but with different functionality.

Extending Abstract Classes

- The child of an abstract class *must* override the abstract methods of the parent, or else it too will be considered abstract.

- An abstract method forces all child objects to have that functionality.
  - Each child class can define it differently.

Abstract Classes- Implemented Methods

- An abstract class can (and often does) contains non-abstract (implemented) methods with full definitions.
- These implemented versions are inherited by the child classes.
  - They can be overridden!

Abstract Classes- Implemented Methods

- Implements represent behavior that is the same for all objects.

Abstract Class Constructors

- Abstract classes can have constructors, even though you cannot instantiate an object.
- You can use the constructors from child classes with super.
- If there are variables in the abstract class, you *should* provide a constructor that initializes those variables.

Abstract Classes and Polymorphism

- An abstract class can be used as a declared type.
  - But not as an actual type, since you cannot instantiate it!

Bottom Line

- Class should be abstract if…
  - It doesn’t make sense to instantiate objects of that type.
- Implemented method goes in the parent class if…
  - All child classes will use the same functionality.
- Abstract method goes in the parent class if…
  - All child classes should have that behavior but they will have different ways of implementing the behavior.

Practice

- Make the Employee and AudioItem classes abstract.
- Decide methods should be abstract in the Employee and AudioItem classes.

INTERFACES

Interfaces

- A Java *interface* is a collection of abstract methods and constants
  - An abstract method can be declared with the modifier `abstract`

- As of the newest revision to Java (Java 8), interfaces can now also contain *default methods*, which are implemented.

Interfaces

- An interface is used to establish a set of methods that a class will implement
  - It's like a contract
- An interface is declared with the reserved word `interface`
- A class indicates that it is implementing an interface with the reserved word `implements` in the class header

Interfaces

interface is a reserved word

None of the methods in an interface are given a definition (body)

```
public interface Doable {
   public void doThis();
   public int doThat();
   public void doThis2 (float value, char ch);
   public boolean doTheOther (int num);
}
```

A semicolon immediately follows each method header

Interfaces

```
public class CanDo implements Doable{
    public void doThis ()
    {
        // whatever
    }

    public void doThat ()
    {
        // whatever
    }

    // etc.
}
```

implements is a reserved word

Each method listed in Doable is given a definition

Interface Constants

- Interfaces can also provide public, final, static constants.

Properties of Interfaces

- An interface cannot be instantiated
- Methods of an interface have public visibility
- If a parent class implements an interface, then by definition, all child classes do as well.
  - That functionality is inherited!

Properties of Classes that Implement an Interface

- Provide implementations for *every* method in the interface
  - Can choose whether to override default methods.
- Can have additional methods as well
- Have access to the constants in that interface
- Can implement multiple interfaces but must implement all methods in each interface

```
class DoesALot implements interface1, interface2 {
   // all methods of both interfaces
}
```

Using Interfaces

- Interfaces describe common *functionality* across classes rather than common *features* (which is more suited for inheritance)
  - Inheritance “is a”
  - Interface “does …” “can …” “is …able”
- Interfaces are Java’s way of ensuring that a class contains an implementation for a specific method.
  - That an object has a specific functionality.

Interfaces and Polymorphism

- An interface can be used as a declared type.
  - But not as an actual type, since you cannot instantiate it!
- The variable can be instantiated with any class that implements the interface
  - The method that is invoked is based on the actual type.

Interfaces and Polymorphism

```
public interface Speaker {
   public abstract void speak();
}
public class Dog implements Speaker {
      public void speak() {
               System.out.println("Woof");
      }
}
public class Philosohper implements Speaker {
      public void speak() {
               System.out.println("Let's discuss life…");
      }
}

Speaker[] speakers = new Speaker[2];
speakers[0] = new Philosopher();
speakers[1] = new Dog();
for(Speaker sp : speakers) {
      sp.speak();
}
```

Practice

- Write interfaces that represents whether an object can be downloaded and whether an object can be streamed.
- Use these interfaces in the AudioItem classes.
  - Think about which classes should implement which interfaces.
- Update the driver program to go through the list of AudioItem classes and create a list of all items that can be streamed.

Abstract Classes vs. Interfaces

	Abstract Classes	Interfaces
Can be used as a declared type?	Yes	Yes
Can be instantiated (used as an actual type)?	No	No
Can contain constructors?	Yes	No
Can contain abstract methods?	Yes	Yes
Can contain non-abstract, implemented methods?	Yes	As of Java 8, yes (default methods)
Can contain public, static, final constants?	Yes	Yes
Can contain variables that are not public, static, final?	Yes	No
Other classes can…	Inherit from only one class	Implement multiple interfaces

COMPARABLE

The `Comparable` Interface

- Specifies that two objects can be *compared* or *ordered* to each other.
- The `compareTo` method defines how that ordering is done.
- Many Java classes implement compareTo.
  - `String` , which is why we can call the `compareTo` method on two `Strings`
- Any class we write can implement `Comparable`
  - We decide how our objects are ordered.

The `Comparable` Interface

- The `Comparable` interface has one abstract method used to compare two objects

```
public int compareTo(Object obj)
```

- You can (and should!) use generics to improve the method:

```
public MyClass implements
                Comparable<MyClass>
     public int compareTo(MyClass obj)
```

The `Comparable` Interface

- The value returned from `compareTo` is :
  - negative if `obj1` is less than `obj2`
  - 0 if they are equal
  - positive if `obj1` is greater than `obj2`

```
if (obj1.compareTo(obj2) < 0)
   // obj1 less than obj2
else if(obj1.compareTo(obj2) > 0
   // obj1 greater than obj2
else
   // they are equal
```

The `Comparable` Interface

- It's up to you how to determine what makes one object greater to, less than, or equal to another
  - Example: For an `Employee` class, you could order employees by name (alphabetically), by employee ID number, or by start date
- The implementation of the `compareTo` method can be as straightforward or as complex as needed

The `Comparable` Interface

- Implementing the `Comparable` interface allows us to use nice methods from the Java standard class library, such as sorting methods.
  - Collections.sort(myArrayList)
  - Arrays.sort(myArray)
- These methods only works if the class implements `Comparable`

Comparable and Sorting

- Note that implementing compareTo doesn’t actually sort anything!
- It only defines *how* to compare two objects to each other.
- This is needed in order to sort. But to actually do the sort, we need another method.

Practice

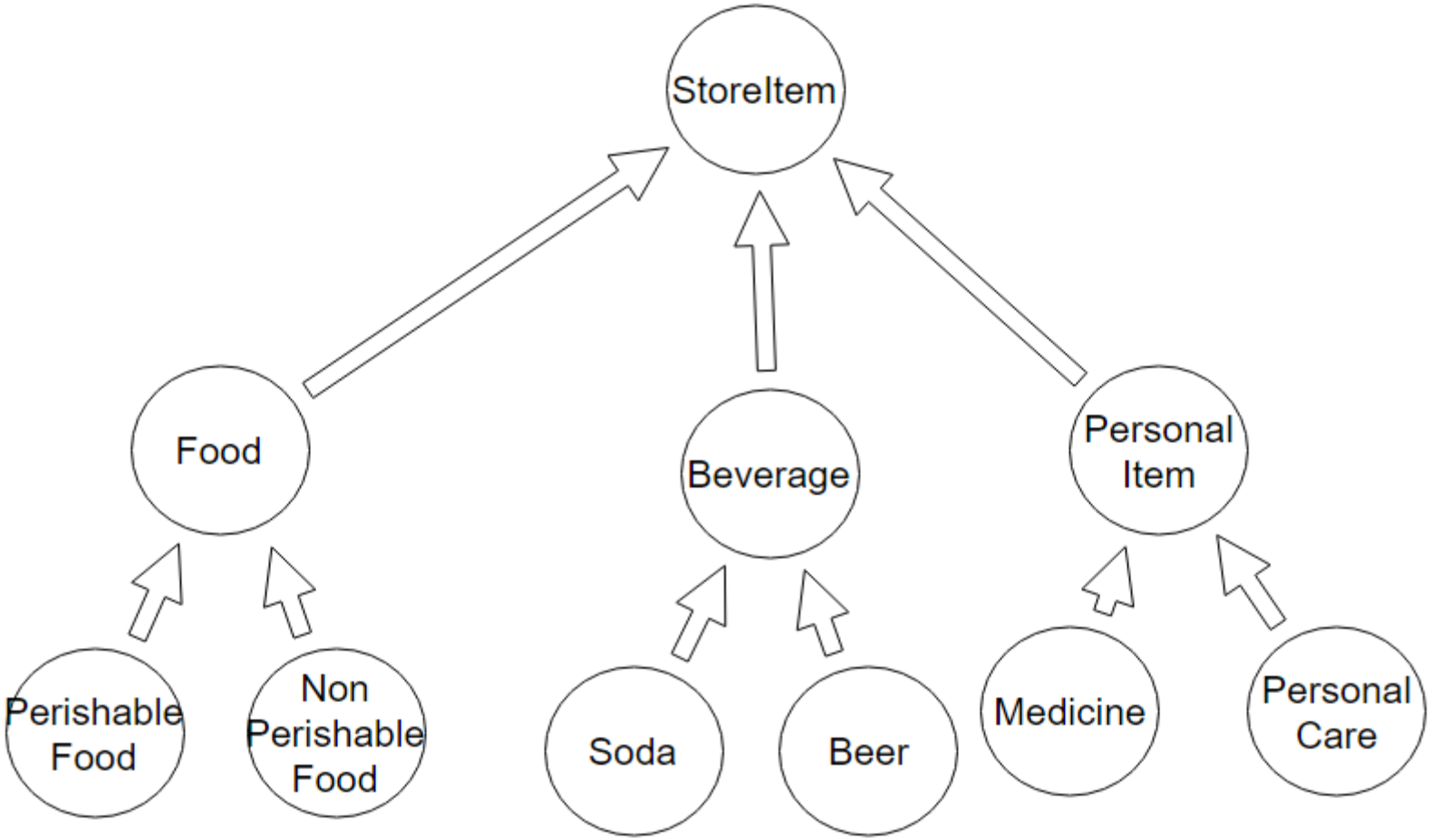
- Modify the Employee and AudioItemclasses to implement Comparable.
- Sort a list of AudioItems.

Practice

- Write a class to represent a playing card and a deck of cards.
  - Define the natural ordering of Card objects.

Practice

- Write a collection of classes to describe items sold at a small gas-station convenience store.
  - All items are described by a brand, name, price, and whether or not their sale is restricted.
  - Beverages are described by a container type: glass, plastic, cardboard, and aluminum.
  - Food is described by weight.
- Write an interface to represent items that are taxable and items that expire.
- Create an inventory list for a store.
  - Print out all restricted items.
  - Print out all items that are expiring soon.


StoreItem
Food
Beverage
Personal Item
Perishable Food
Non Perishable Food
Soda
Beer
Medicine
Personal Care