Abstract Classes and Interfaces

Inheritance carried to an extreme.
Abstract Class

- A template for related classes.
  - The related classes will have some stuff in common.

- public abstract class Instrument {...} says “Inheritance is required!”

- An abstract class cannot be instantiated, only extended.
  - public class Piano extends Instrument{...} /*valid*/
  - Instrument m = new Instrument(); /*error*/
Why Bother? (Part 1)

- **Use these classes to help organize your code.**
  - Put all the common features in the abstract base class.
    - Example: All Pets have certain features in common.
      - Name
      - Tricks
      - Begs for food
    
    - So put all this into an abstract base class. Each specific pet can inherit from this class.

- **Use when the abstract base class is not something that should be instantiated as an object.**
  - e.g., might claim that there is no such *object* as an “instrument” but there is such a thing as a “piano”, “drum”, etc.
Pet Example

```java
public abstract class Pet {
    private String name = null;

    public Pet() {
        name = " ";
    }

    public void setName(String name) {
        this.name = name;
    }

    public String getName() {
        return name;
    }

    public abstract void doTrick();
    public abstract void begForFood();
}
```

A constructor, just like you are used to, but it can’t be used except when called by the child class.

A method, just like you are used to.

Not implemented! Must override in the child class.

If have abstract methods then class MUST also be abstract (but not vice-versa).
Using the Abstract Pet

```java
public class PetRock extends Pet {
    
    public void doTrick() {
        System.out.println("Sitting quietly.");
    }

    public void begForFood() {
        for(int i = 0; i<100; i++) {
            doTrick();
        }
    }
}
```

As **required**, overrides these abstract methods from the base class.
Why Bother Again?

- We want a Pet class that outlines all the necessary features of a pet.

- But we don’t know what the trick will be.
  - So we can’t fill in this method!
  - Make it abstract and let the child class fill it in.
    - The child class will know what trick it can do.
What if Don’t Implement?

- Suppose child class doesn’t implement an abstract method. Then the child class is also abstract.

```java
public abstract class Dog extends Pet {
    public void begForFood() {
        for(int i = 0; i<100; i++) {
            doTrick();
        }
    }
}

public class Husky extends Dog {
    public void doTrick() {
        System.out.println("bow-wow");
    }
}
```

- Doesn’t implement the “doTrick” method. So the class is still abstract.

  Now, finally implements the “doTrick”.
Polymorphism and Abstractness

- Cannot instantiate an abstract class.

- But we CAN declare it.
  - e.g., Pet rover = new Husky();

- Why’s this work?
  - Casting!

- Why would we do this?
  - Good for writing methods that take a Pet as a parameter.
    - Remember our Vet class with the “getName(Mammal a)” method?
    - Remember our NoiseFest with the “setTruck(Truck t)”
General Rules

1. Use abstract classes to organize code into templates.
2. Abstract classes can be inherited.
3. Abstract classes cannot be instantiated.
   • But it may have a constructor (which would be called either by default or explicitly from the child class).
4. Abstract classes can be declared.
5. Abstract classes do not have to have abstract methods.
6. If a method is abstract, then the class must be abstract.
Your Turn

- Create an abstract class with an abstract method.
  - e.g., Animal, Instrument, Pet, Person, etc.

- Extend the class.
Too Much of A Good Thing?

- Inheritance is useful.
- So why not **multiple inheritance**?
  - Can get features of both base classes.

```java
public class Dog extends Animal, Pet {
    ...
}
```

- Can do this in C++.
- Can’t do this in Java.
Why Not Multiple Inheritance?

- Can be good if used well, but easy to use poorly.
  - Requires complex rules.
    1. What if both parents have the same method?
       - Which one should it use?
    2. What if “A extends B, C” and both B and C extend D.
       - Then A extends D twice? Gets methods and public instance variables twice?
       - What if B and C override one of D’s methods in different ways!
         - Which one does A use?
       - Called “dreaded diamond.”

- All can be resolved, but usually too much trouble.
Java’s Answer to Multiple Inheritance

- Create interfaces.
  - Similar to abstract classes.
  - Class can “implement” as many interfaces as desired.
  - All methods are abstract
    - So any identical conflicting methods are just implemented once.
    - E.g.,
      1. If class uses two interfaces that have the same abstract method,
      2. And the class also inherits an implemented method with the same name, return type, and parameters,
      3. Then the implemented method applies to both of the interfaces as well.
    - NO CONFLICT!
Interfaces

- An abstract class carried to an extreme.
  - Nothing but empty method declarations.
  - May contain constants as well.

```java
public interface CuteTricks {
    public void rollOver();
    public void shakeHands();
    public String speak();
}
```

```java
public interface Flyable {
    public void takeOff();
    public void land();
}
```
Pictures of Interface

CuteTricks

Pet

Vehicle

Car

Flyable

FlyingCar
Using Interfaces

Use "implements" keyword.
   Similar to "extends".

```java
public class Pet implements CuteTricks {
    public void rollOver() {
        ...
    }

    public void speak() {
        ...
    }

    public void shakeHands() {
        ...
    }
}
```

Similar to abstract methods, these must be implemented (because they are in the interface).
public class FlyingCar extends Car implements Flyable {
    public void takeOff() {
        ...
    }
    public void land() {
        ...
    }
}
public class JamesBondCar extends Car implements Flyable, Floatable, SpaceWorthy {
    ...
}
By contract you must make this method do the following:

- Return a negative integer, zero, or a positive integer (when this object is less than, equal to, or greater than the specified object).
- Throw a ClassCastException if the object \( o \) isn’t the correct type.
  - What’s the correct type? The type of the class that implements the interface.
Comparable

Used by Integer, Double, BigInteger, Float, and other classes.

- public final class Integer extends Number
  implements Comparable

- public final class Double extends Number
  implements Comparable

Each of these implements the compareTo method.
public class Wine implements Comparable
{
    private int age;
    private String name;

    public Wine(int age, String name)
    {
        this.age = age;
        this.name = name;
    }

    public int getAge()
    {
        return age;
    }

    public String toString()
    {
        return name;
    }
}

Continued on next page.

Overrides this method that is in the Object class!
public int compareTo(Object o) {
    if(!(o instanceof Wine)) {
        throw new ClassCastException();
    }
    Wine vino = (Wine) o;
    if(age < vino.getAge()) {
        return -1;
    } else if(age > vino.getAge()) {
        return 1;
    } else {
        return 0;
    }
}
If a class implements an interface, then it can be cast to that type.

```java
public class Wine implements Comparable {
    ...

    public class Snooty {
        public void printBetterOne(Comparable c1, Comparable c2) {
            if (c1.compareTo(c2) >= 0 ) {
                System.out.println(c1.toString());
            } else {
                System.out.println(c2.toString());
            }
        }
    }
}
```

```java
public static void main(String[] args) {
    Snooty s = new Snooty();
    Wine w1 = new Wine(100, "Pinot Noir");
    Wine w2 = new Wine(20, "Merlot");
    s.printBetterOne(w1, w2);

    Beer samAdams = new Beer("Sam");
    Beer guinness = new Beer("Guinn");
    s.printBetterOne(samAdams, guinness);
}
```