Section 8

Interfaces & Ploymorphism

Learning Outcomes

- After this section you should be able to:
  - Distinguish between public private and protected access modifiers.
  - Distinguish between Abstract and Concrete classes
  - Create abstract classes and abstract methods
  - Write programs that are easily extensible and modifiable by applying polymorphism in program design.

  - ***Reading*** and ***studying*** recommended text is essential to improve your understanding of the above

Access to Class Members

- A class may be declared with the modifier public, in which case that class is visible to all classes everywhere.
- If a class has no modifier (the default, also known as *package-private*), it is visible only within its own package (packages are named groups of related classes — more later).
- At the member level, you can also use the public modifier or no modifier (*package-private*) just as with top-level classes, and with the same meaning.
- For members, there are two additional access modifiers: private and protected.
  - The private modifier specifies that the member can only be accessed in its own class.
  - The protected modifier specifies that the member can only be accessed within its own package (as with *package-private*) and, in addition, by a subclass of its class in another package.

Access Levels

Modifier	Class	Package	Subclass	World
public	Y	Y	Y	Y
protected	Y	Y	Y	N
No modifier	Y	Y	N	N
private	Y	N	N	N

Tips:
- Use the most restrictive access level that makes sense for a particular
- member.
- Use private unless you have a good reason not to.
- Avoid public fields except for constants.

Protected Modifier

```
public class SalesEmployee {

    // Data Members
    private String firstName;
    private String lastName;
    private String ppsNumber;
    protected double sales = 0.0;
    protected double commission = 0.0;
    …
}


public class SalesPerson extends SalesEmployee {

    …

    public void calculateCommission(){
        commission = sales * 0.15;
    }

}
```

Interfaces

- There are a number of situations in software engineering when it is important for disparate groups of programmers to agree to a "contract" that spells out how their software interacts. Each group should be able to write their code without any knowledge of how the other group's code is written. Generally speaking, *interfaces* are such contracts.
- In the Java programming language, an *interface* is a reference type, similar to a class, that can contain *only* constants, method signatures, default methods, static methods, and nested types. Method bodies exist only for default methods and static methods. Interfaces cannot be instantiated—they can only be *implemented* by classes or *extended* by other interfaces.

Class Instantiation

- A Beagle reference to a Beagle Object
  - `Beagle beagle = new Beagle();`
- A Siamese reference to a Siamese object
  - `Siamese saimese = new Siamese();`
- A Dog reference to a Dog object
  - `Dog dog = new Dog();`
- A Cat reference to a Cat object
  - `Cat cat = new Cat();`
- An Animal reference to an Animal object
  - `Animal animal = new Animal();`
- Does an Animal object or a Cat object or a dog object make sense?

Abstract Classes

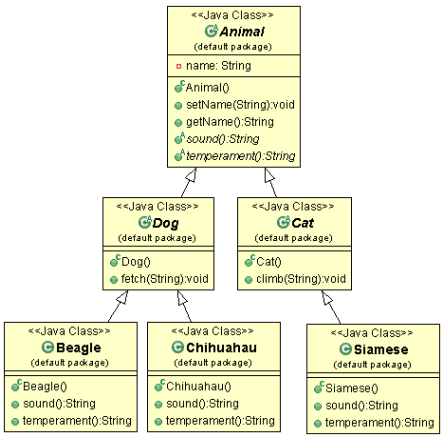
- To prevent a class from being instantiated you can make the class abstract.
- This tells the compiler to prevent the use of **new** on the type.
- Abstract classes can still be used as a reference type.
  - Can use it as a polymorphic argument, return type or create a polymorphic array.
- When designing a class inheritance structure, you have to decide whether classes are going to be abstract or concrete
  - Concrete classes are classes that are specific enough to be instantiated (it's OK to make objects of that type).

You Can't Instantiate Abstract classes

- An abstract class means you cannot make an new instance of the class.
- For polymorphic purposes you can use an abstract class as a declared reference type.
- To declare a class abstract, use the abstract keyword
  - `abstract class Animal {`
- An abstract class has virtually no value unless it is extended (There is an exception as an abstract class can have static members – more in the next section).
  - `abstract class Dog extends Animal{`
- *With an abstract class the work at runtime is carried out by instances of a subclass of your abstract class.*
- Note: you can use the public keyword when using abstract

Abstract vs. Concrete

- A class that is not abstract is called a concrete class.



- In our Animal inheritance tree, if we make Animal, Dog and Cat abstract classes – Beagle, Chihuahau and Siamese are concrete classes. (Note: A to indicate abstract)

Abstract Methods

- An abstract class means the class **must** be extended
- An abstract method means the method **must** be overridden.
- If you decide some behaviours in an abstract class make more sense if they are implemented in a more specific subclass, make the method abstract.
- Am abstract method has no body and ends with a semicolon.
  - `public abstract String sound();`
  - `public abstract String temperament();`
- If you declare a method abstract you **must** also mark the class abstract.
  - `public abstract class Animal`
- You cannot have an abstract method in a non-abstract class, but you can mix both abstract and concrete methods in an abstract class.

Implementing Abstract Methods

- Implementing an abstract method is like overridding a method.
- Abstract methods exist solely for polymorphism, the first concrete class in the inheritance tree **must** implement all abstract methods.
- If both Animal and Dog are abstract, class Dog does not have to implement the abstract methods from Animal.
- The first concrete subclass, i.e. Beagle, **must** implement all of the abstract methods in Animal and if there are any abstract methods in Dog, they **must** also be implemented.
- An abstract class can have abstract and non-abstract methods, so Dog could implement an abstract method from Animal, so that Beagle didn't have to.
- If Dog doesn't provide an implementation of Animals abstract methods, Beagle has to implement all of Animals abstract methods.

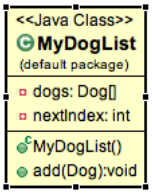
Abstract Summary

- An *abstract class* is a class that is declared abstract—it may or may not include abstract methods. An *abstract class* is a class that is declared abstract—it may or may not include abstract methods.
- Abstract classes cannot be instantiated, but they can be subclassed.
- An *abstract method* is a method that is declared without an implementation (without braces, and followed by a semicolon).
- When an abstract class is subclassed, the subclass usually provides implementations for all of the abstract methods in its parent class.
- However, if it does not, then the subclass **must** also be declared abstract.

Back to Polymorphism

- Build a Dog specific list, which will hold Dog objects.

```
public class MyDogList {

    private Dog[] dogs;
    private int nextIndex;

    public MyDogList(){
        dogs = new Dog[5];
        nextIndex = 0;
    }

    public void add(Dog d) {

        if (nextIndex < dogs.length){
            dogs[nextIndex] = d;

            System.out.println("Dog added at " + nextIndex);

            nextIndex++;
        }

    } // end method

} // end class
```



Back to Polymorphism (cont.)

- Test drive MyDogList.

```
public class ListTest {

    public static void main (String[] args) {

        MyDogList list = new MyDogList();

        Dog d = new Dog();
        list.add(d);
    }
}
```

- Now we discover we need to add Cats!!!
- Options?
  - Make a separate list for Cats!
  - Make a single DogAndCatList class, with two separate arrays and two add() methods.
  - Make a more generic AnimalList class that takes any kind of Animal subclass

Prevent Instantiation – use abstract

```
abstract class Animal { ... }

public class Dog extends Animal{ ... }

public class Cat extends Animal{ ... }


Dog c = new Dog();

Cat d = new Cat();

Animal a = new Animal();
```

Animal List

```
public class MyAnimalList {

    private Animal[] animals = new Animal[5];

    private int nextIndex = 0;

    public void add(Animal a) {

    if (nextIndex < animals.length){
        animals[nextIndex] = a;

        if(animals[nextIndex] instanceof Dog) {
           System.out.println("Dog added at " + nextIndex);
        }
        else if (animals[nextIndex] instanceof Cat){
            System.out.println("Cat added at " + nextIndex);
        }
        else
             System.out.println("Another type of Animal added at " + nextIndex);
             nextIndex++;
        }
    }
}
```

Non-Animal Types?

- Change the type of Array to something more generic than Animal
- Remember ArrayList?
- Every class in Java extends class Object
  - Object is the superclass of everything.
  - Every class you write implicitly extends Object

```
public class Dog extends Object{ }
```

  - However Dog already extends Animal – The compiler will make Animal extend Object (Note: Dog does extend Object indirectly)

```
public class Dog extends Animal{ }
```

- Any class that doesn't explicitly extend another class explicitly extends Object.

class ArrayList<E>

- Aim to create a class with methods that take and return polymorphic types.
- Without a common superclass you couldn't create classes and methods that can take custom types.
- Many ArrayList methods use the ultimate polymorphic type Object.
  - As every class in Java is a subclass of Object, ArrayLists methods can take any type of object
  - E type parameter in ArrayList, o Object class in Java.land

boolean contains(Object o)	Returns true if this list contains the specified element.
boolean isEmpty()	Returns true if this list contains no elements
int indexOf(Object o)	Returns the index of the first occurrence of the specified element in this list, or -1 if this list does not contain the element.
E get(int index)	Returns the element at the specified position in this list.
boolean add(E e)	Appends the specified element to the end of this list.
boolean remove(Object o)	Removes the first occurrence of the specified element from this list, if it is present.

class ArrayList <E>

```
java.lang.Object
    java.util.AbstractCollection<E>
        java.util.AbstractList<E>
            java.util.ArrayList<E>
```

```
public class ArrayList<E>
extends AbstractList<E>
implements List<E>, RandomAccess, Cloneable, Serializable
```

- Resizable-array implementation of the List interface. Implements all optional list operations, and permits all elements, including null. In addition to implementing the List interface, this class provides methods to manipulate the size of the array that is used internally to store the list.

class Object

- (java.lang.Object) class Object is the root of the class hierarchy. Every class has Object as a superclass. All objects, including arrays, implement the methods of this class.
- Every class you write inherits all of the methods of class Object.

String toString()	Returns a string representation of the object.
int hashCode()	Returns a hash code value for the object.
boolean equals(Object obj)	indicates whether some other object is "equal to" this one.
Class<?> getClass()	Returns the runtime class of this Object.

- *In general, the toString method returns a string that "textually represents" this object. The result should be a concise but informative representation that is easy for a person to read. It is recommended that all subclasses override this method.*

ArrayList<Object>

- Using type Object as a reference.
- Put an object of type Dog into an ArrayList<Dog>, it goes in as a Dog and comes out as a Dog.

```
ArrayList<Dog> myArrayList = new ArrayList<Dog>();
Dog dog = new Dog();
myArrayList.add(dog);
Dog d = myArrayList.get(0);
```

- Declare the ArrayList to take any type of Object - ArrayList<Object>

```
ArrayList<Object> myArrayList = new ArrayList<Object>();
Dog dog = new Dog();
myArrayList.add(dog);
Dog d = myArrayList.get(0); // compile error get() returns type Object
```

  - Problem when you try to get the Dog object and assign it to a Dog reference.
  - Everything returned from ArrayList<Object> as a reference of type Object regardless of what the actual object is, or what the reference type was when the object was added to the list.

Returning types

- Compiler won't let you assign the returned reference to anything but Object.

```
public void go(){
    Dog dog = new Dog();
    Dog sameDog - getObject();
}

public Object getObject(Object o){
    return o;
}
```

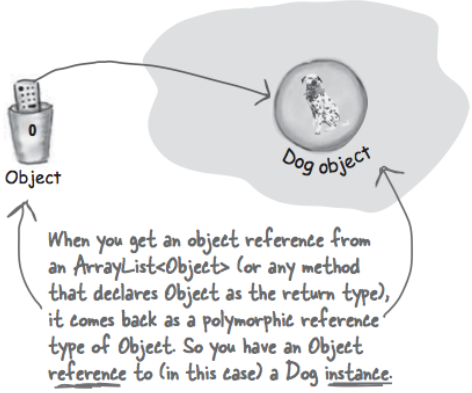
- The following works but may not be of much use! (see later)

```
public void go(){
    Dog dog = new Dog();
    Object sameDog - getObject();
}

public Object getObject(Object o){
    return o;
}
```

- You can assign anything to a reference of type Object (all classes pass the IS-A test with Object).

Polymorphic Reference

- When an object is referenced by a variable declared as type Object, it can't be assigned to a variable declared with the actual object's type.
- Can happen when return type or argument is declared as type Object.



Objects can't fetch()

- The compiler decides whether you can call a method based on the reference type, not the actual object type.

```
Object o = myArrayList.get(0);
int i = o.hashCode(); // Object has a hash code
o.fetch("stick"); // won't compile,
```

- Object doesn't have a fetch() method, even though the object stored in the ArrayList is of type Dog.
- This is because the variable 'o' is declared as type Object not Dog and the compiler sees it as the generic class Object (the compiler checks the class of the reference type not the object type).
- If the reference is declared as type Object, you can call methods only if those methods are in class Object

Casting

- When you put an object in an ArrayList<Object>, you can only treat it as an Object, regardless of the type it was when you put it in.
- When you get a reference from an ArrayList<Object>, the reference is always of type Object

```
Object o = myArrayList.get(0);
int i = o.hashCode(); // Object has a hash code
Dog d = (Dog) o; // Cast the Object back to the Dog it really is
o.fetch("stick"); // o can now see the Dog methods
```

- You will get a ClassCastException at runtime if you try to cast another type like Cat to Dog.

```
if (o instanceof Dog){
    Dog d = (Dog) o;
}
```

Multiple Inheritance

- Say you need two superclasses at the top of the inheritance hierarchy.
- Multiple inheritance is not allowed in Java, you can explicitly extend from one class only.
- In Java, an interface solves the multiple inheritance problem and allows for polymorphism.
- A Java interface is like a 100% abstract class, it defines only abstract methods.
- If you make all of the methods abstract, the subclass must implement the methods and avoids confusion for the JVM as to which of the inherited versions to call.
- Interfaces are declared using the interface keyword.

```
public interface Pet {...}
```

Implementing an Interface

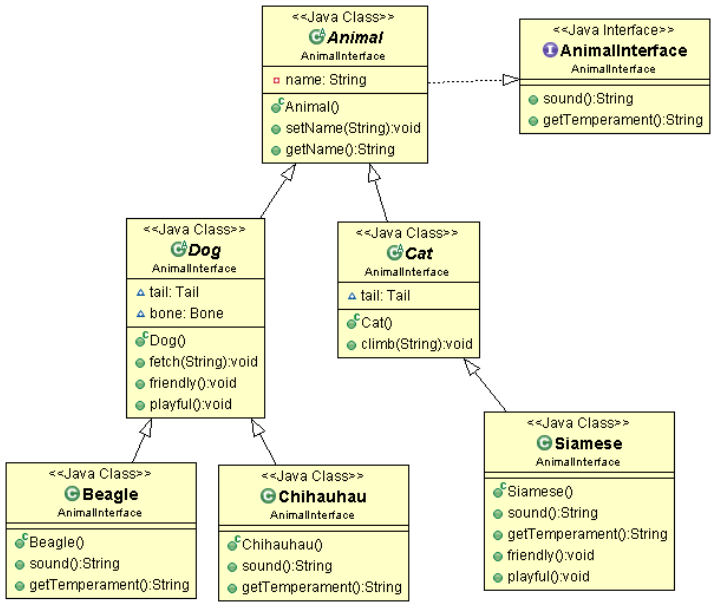
- To implement an interface use the keyword implements

```
public class Dog extends Animal implements Pet {...}

abstract class Animal implements AnimalInterface {...}
```

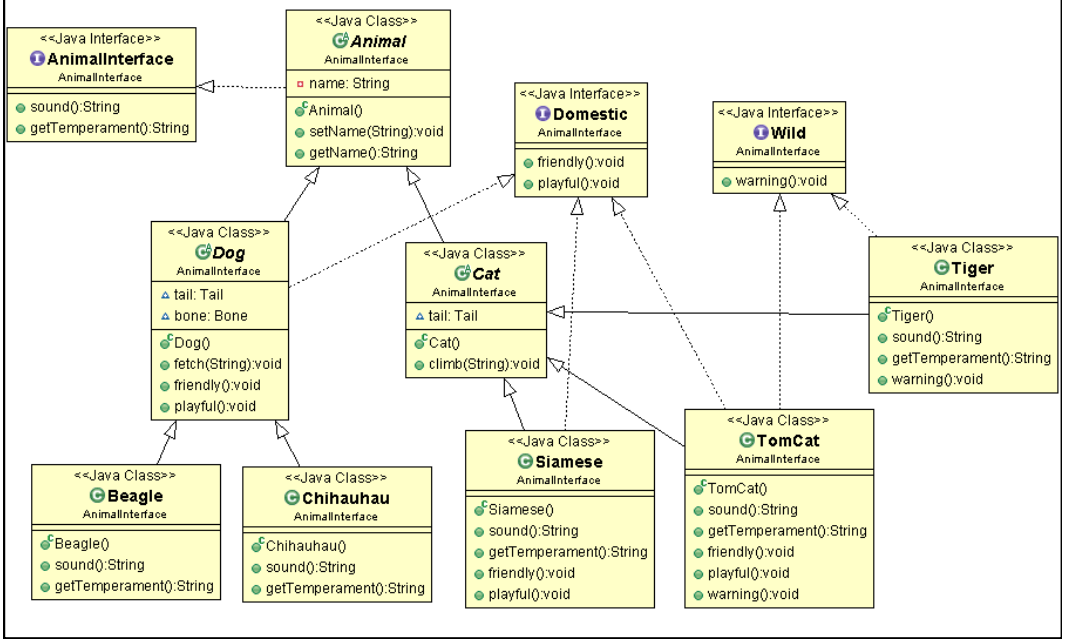
- Your class can implement multiple interfaces
- Interface methods are explicitly public and abstract, it's not considered good style to use them.

```
interface AnimalInterface {

    // public and abstract by default
    String sound();
    String getTemperament();

}
```

Animal Interface



Using Multiple Interfaces



Implementing Multiple Interfaces

```
interface Domestic {

    //abstract by default
    void friendly();
    void playful();
}
```

```
interface Wild {

    //abstract by default
    void warning();

}
```

```
abstract class Dog extends Animal implements Domestic{ ... }

public class Tiger extends Cat implements Wild{ ... }

public class TomCat extends Cat implements Domestic, Wild { ... }
```

Interface or Abstract?

- Interface
  - Defines behaviour

    Contract
  - Cannot be instantiated
  - A class can implement multiple interface
- Abstract Class
  - Defines behaviour
  - Can have implementation code
  - Cannot be instantiated
  - A class can inherit from a single abstract class

Which to choose?

- Duplicating code is not a good thing. So, you want to avoid it whenever possible.
- abstract class can be very useful for reducing duplicate code, but you should be sure that it's warranted before using one.
- If (almost) all classes implementing the behaviour would have the same code, then you can use an abstract class to implement it.
- Often, even with abstract classes, we create an interface for it. Interfaces define the contract of the behavior by expressing the signature of the method and its return type.

Interfaces & Ploymorphism

- When you use an interface as a polymorphic type (i.e. to create an array of interface types), the objects can come from anywhere in the inheritance tree.
- The only requirement is that the objects are from a class that implements the interface.
- Thus allowing classes in different inheritance trees to implement a common interface.
  - Want an object to be able to save its state to a file – implement the Serializable interface
  - Need objects to run their methods in a separate thread of execution – implement Runnable.
  - Classes from anywhere in the inheritance tree may need to implement the Serializable or Runnable interfaces.

When to do What!

- How do you know when to make a class a subclass, an abstract class, or an interface.
  - Make a class that doesn't extend anything (other than Object) when your new class doesn't pass the IS-A test for any other type.
  - Make a subclass (extend a class) only when you need to make a more specific version of a class and need to override or add new behaviours.
  - Use an abstract class when you want to define a template for a group or subclasses and you have at least some implemented code that all subclasses could use. Make the class abstract when you want to guarantee that nobody can make objects of that type.
  - Use an interface when you want to define a role that other classes can play, regardless of where these classes are in the inheritance tree.

Invoking the superclass version

- To invoke the superclass version of a method, use the super keyword.

```
super.methodName();
```

- calls the superclass version of the method, rather than the overridden version contained in the subclass.

Summary

- When you don't want a class to be instantiated (can't make a new object of that class type) create the class using the *abstract* keyword.
- An abstract class can have both abstract & non-abstract methods.
- If a class has even one abstract method, the class must be marked abstract.
- An abstract method has no body and the declaration ends with a semicolon (no curly brackets).
- All abstract methods must be implemented in the first concrete subclass in the inheritance tree.
- Every class in Java is either a direct or indirect subclass of class Object (java.lang.Object).

Summary (cont.)

- Methods can be declared with Object arguments and/or return types.
- You can call methods on an object only if the methods are in the class (or interface) used as the reference variable type, regardless of the actual object type.
- Therefore, a reference variable of type object can be used only to call methods defined in class Object, regardless of the type of the object to which the reference refers.
- A reference variable of type Object can't be assigned to any other reference type without a cast.
- A cast can be used can be used to assign a reference variable of one type to a reference variable of a subtype, but at runtime the cast will fail if the object on the heap is NOT a type compatible with the cast.

Summary (cont.)

- All objects come out of an ArrayList<Object> as type Object (meaning they can be referenced only be an Object reference variable, unless you use a cast).
- Multiple inheritance is not allowed in Java. You can only extend one class (can only have one immediate superclass).
- AN interface is like a 100% pure abstract class. It defines only abstract methods.
- Create an interface using the interface keyword instead of class.
- Implement an interface using the keyword implements
  - Dog implements Pet
- A class can implement multiple interfaces.

Summary (cont.)

- A class that implements an interface must implement all the methods of the interface, since all of the methods are implicitly public and abstract.
- To invoke the superclass version of a method, use the super keyword.
  - `super.methodName();`