Inheritance

Introduction

- Create a general class that defines traits common to a set of related items.
- This class can then be inherited by other, more specific classes, each adding those things that are unique to it.
- A class that is inherited is called a *superclass.*
- *The class that* does the inheriting is called a *subclass.*
- A subclass is a specialized version of a superclass.
  - It inherits all of the instance variables and methods defined by the superclass and adds its own, unique elements.

```
class subclass-name extends superclass-name {
// body of class
}
```

- Expressed using keyword “extends”.
- Main advantage is code reusability.

Con’t

- The subclass and the superclass has an "is-a" relationship.
- E.g. class Car extends Vehicle { }
  - Car is a vehicle // is-a relationship

Accessibility

- Within a subclass you can access its superclass's public and protected methods and fields , but not the superclass's private methods.
- If the subclass and the superclass are in the same package, you can also access the superclass's default methods and fields.

- Practical Example
  - DemoBoxWeight.java
- A Superclass Variable Can Reference a Subclass Object
  - Test.java
  - RefDemo.java
- Super Keyword
  - The keyword super represents an instance of the direct superclass of the current object.

Super has two general forms

- First is you can explicitly call the parent's constructor from a subclass's constructor by using the super keyword.
  - 'super' must be the first statement in the constructor.
  - super(*parameter-list);*
  - DemoSuper.java
- The second is used to access a member of the superclass that has been hidden by a member of a subclass.

- super.*member*
- *member can be either a method or an instance variable.*

```
// Using super to overcome name hiding.
class A {
    int i;
}
// Create a subclass by extending class A.
class B extends A {
     int i;                              // this i hides the i in A
     B(int a, int b) {
            super.i = a;             // i in A
             i = b;                      // i in B
      }
void show() {
     System.out.println("i in superclass: " + super.i);
     System.out.println("i in subclass: " + i);
     }
}
class UseSuper {
       public static void main(String args[]) {
       B subOb = new B(1, 2);
       subOb.show();
    }
}
```

Creating a Multilevel Hierarchy

- DemoShipment.java

When Constructors Are Called

- When a class hierarchy is created, in what order are the constructors for the classes that make up the hierarchy called?
  - Constructors are called from superclass to subclass.

Method Overriding

- When a method in a subclass has the same name and type signature as a method in its superclass, then the method in the subclass is said to *override the method in the superclass.*
- Override.java

Method Overloading

- If B is subclass of A and
- In class A
  - void show()
- In class B
  - void show(String msg)   // overload show()

Dynamic Method Dispatch

- Dynamic method dispatch is the mechanism by which a call to an overridden method is resolved at run time(i.e. dynamic binding), rather than compile time.
- Java implements run-time polymorphism using it.
- It is the type of the object being referred to (not the type of the reference variable) that determines which version of an overridden method will be executed.
    - FindArea.java

Abstract (class and method)

Class

- An abstract class is one that cannot be instantiated.
- If a class is abstract and cannot be instantiated, the class does not have much use unless it has subclasses.
- Use the **abstract** keyword to declare a class abstract.
  - Ex: public abstract class Figure

Method

- If you want a class to contain a particular method but you want the actual implementation of that method to be determined by child classes, you can declare the method in the parent class as abstract.
- The abstract keyword is used to declare a method as abstract.

Con’t

- An abstract methods consist of a method signature, but no method body.
  - abstract *type name(parameter-list);*
- If a class contains an abstract method, the class must be abstract as well.
- A child class that inherits an abstract method must override it. If they do not, they must be abstract, and any of their children must override it.
  - AbstractArea.java

final with Inheritance

- Using final to Prevent Overriding (final method)

```
class A {
    final void meth() {
    System.out.println("This is a final method.");
    }
}
class B extends A {
      void meth() { // ERROR! Can't override.
      System.out.println("Illegal!");
      }
}

//gives compile time error
```

- Improves performance
  - compiler is free to inline calls to them because it “knows” they will not be overridden by a subclass.
- final methods cannot be overridden, a call to one can be resolved at compile time. This is called *early binding.*

final to Prevent Inheritance (final class)

- Prevent a class from being inherited.
- Declaring a class as final implicitly declares all of its methods as final, too.
- It is illegal to declare a class as both abstract and final since an abstract class is incomplete by itself and relies upon its subclasses to provide complete implementations.

```
final class A {
        // ...
}

// The following class is illegal.
class B extends A { // ERROR! Can't
        subclass A
        // ...
}
```

instanceof operator

- The **instanceof operator** is used to test whether the object is an instance of the specified type (class or subclass or interface).
- The instanceof operator is also known as type comparison operator because it compares the instance with type.
- It returns either true or false.
- If we apply the instanceof operator with any variable that have null value, it returns false.

```
class Animal{}
class Dog extends Animal{//Dog inherits Animal
        public static void main(String args[]){
                Dog d = new Dog();
                Dog d1 = null;
                System.out.println(d instanceof Animal);//true
                System.out.println(d1 instanceof Dog);//false
        }
}
```

Casting

- Anytime an object of that base class type is type cast into a derived class type, it is called a downcast.
  - Dog d=new Animal(); //Compilation error
- Upcasting is allowed in Java, however downcasting gives a compile error.
- The compile error can be removed by adding a cast but would anyway break at the runtime.
  - Dog d=(Dog)new Animal();

//Compiles successfully but ClassCastException is thrown at runtime

Example

```
class Animal{}
class Dog extends Animal{
        static void method(Animal a){
                if(a instanceof Dog){
                        Dog d=(Dog)a;
                        System.out.println("ok downcasting performed");
                }
        }
        public static void main(String args[]){
                Animal a = new Dog();
                Dog.method(a);
        }
}
```

Output : ok downcasting performed

- remove instanceof and write Animal a = new Animal(); and check
- ClassCastException is thrown at runtime

Interface

- An interface is a collection of final, static fields and abstract methods.
- A class implements an interface, thereby inheriting the abstract methods of the interface.
- An interface is not a class.
- Writing an interface is similar to writing a class, but they are two different concepts.
- A class describes the attributes and behaviors of an object.
  - An interface contains behaviors that a class implements.
- Unless the class that implements the interface is abstract, all the methods of the interface need to be defined in the class.
- One class can implement any number of interfaces.
- Provides run time polymorphism i.e. “one interface, multiple methods”.

Con’t

An interface is similar to a class in the following ways:

- An interface can contain any number of methods.
- An interface is written in a file with a **.java** extension, with the name of the interface matching the name of the file.
- The bytecode of an interface appears in a **.class** file.
- Interfaces appear in packages, and their corresponding bytecode file must be in a directory structure that matches the package name.

However, an interface is different from a class in several ways:

- You cannot instantiate an interface.
- An interface does not contain any constructors.
- All of the methods in an interface are abstract.
- An interface cannot contain instance fields. The only fields that can appear in an interface must be declared both static and final.
- An interface is not extended by a class; it is implemented by a class.
- An interface can extend multiple interfaces.

Declaring Interfaces:

- **interface** keyword is used to declare an interface.

```
/* File name : NameOfInterface.java */

public interface NameOfInterface
{   //Any number of final, static fields
   //Any number of abstract method declarations
}
/* File name : TestIface.java */
    interface Callback {
    void callback(int param);
    }
```

- **Interfaces have the following properties:**
- An interface is implicitly abstract. You do not need to use the **abstract** keyword when declaring an interface.
- Each method in an interface is also implicitly abstract, so the abstract keyword is not needed.
- Methods in an interface are implicitly public.

Implementing Interfaces:

- When a class implements an interface, you can think of the class as signing a contract, agreeing to perform the specific behaviors of the interface.
- If a class does not perform all the behaviors of the interface, the class must declare itself as abstract.
- A class uses the **implements** keyword to implement an interface.
- The implements keyword appears in the class declaration following the extends portion of the declaration.

```
class Client implements Callback
```

General form:

access class classname [extends superclass]
[implements *interface [,interface...]] {*
    // class-body
}

*access is either **public or not used.***

one or more classes can implement that interface.

Syntax

```
interface i1{
        variable declaration;
        methods declaration;
}
interface i2 extends i1{
        variable declaration;
        methods declaration;
}
class c1 implements i1{
        variable declaration;
        methods declaration;
}
class c2 extends c1 implements i1,i2{
        Body of class
 }
```

Interface and Polymorphism

- TestIface.java

Partial Implementations

- If a class includes an interface but does not fully implement the methods defined by that interface, then that class must be declared as **abstract**

```
abstract class Incomplete implements Callback {
    int a, b;
    void show() {
        System.out.println(a + " " + b);
    }
    // ...
}
```

- The class Incomplete does not implement callback( ) and must be declared as abstract.
- Any class that inherits Incomplete must implement callback( ) or be declared abstract itself.

Interface

- When overriding methods defined in interfaces there are several rules to be followed:
  - The signature of the interface method and the same return type or subtype should be maintained when overriding the methods.
  - An implementation class itself can be abstract and if so all interface methods need not be implemented.
- When implementation interfaces there are several rules:
  - A class can implement more than one interface at a time.
  - A class can extend only one class, but implement many interface.
  - An interface can extend another interface, similarly to the way that a class can extend another class.

Extending Interfaces

- An interface can extend another interface, similarly to the way that a class can extend another class.
- The extends keyword is used to extend an interface, and the child interface inherits the methods of the parent interface.
- When a class implements an interface that inherits another interface, it must provide implementations for all methods defined within the interface inheritance chain.
  - IFExtend.java

Extending Multiple Interfaces:

- A Java class can only extend one parent class.
- Multiple inheritance is not allowed.
- Interfaces are not classes, however, and an interface can extend more than one parent interface.
- The extends keyword is used once, and the parent interfaces are declared in a comma-separated list.
- For example, if the Hockey interface extended both Sports and Event, it would be declared as:

  public interface Hockey extends Sports, Event
- An interface with no methods in it is referred to as a **tagging** interface.

Variables in Interfaces

- Group of contants in interface.
  - Date1.java

The differences between abstract class an interface

1. Abstract class has the constructor, but interface doesn’t.
2. Abstract classes can have implementations for some of its members (Methods), but the interface can’t have implementation for any of its members.
3. Abstract classes should have subclasses else that will be useless..
4. Interfaces must have implementations by other classes else that will be useless
5. Only an interface can extend another interface, but any class can extend an abstract class..
6. All variable in interfaces are final by default
7. Interfaces provide a form of multiple inheritance. A class can extend only one other class.
8. Interfaces are limited to public methods and constants with no implementation. Abstract classes can have a partial implementation, protected parts, static methods, etc.

The differences between abstract class an interface

9. A Class may implement several interfaces. But in case of abstract class, a class may extend only one abstract class.
10. Interfaces are slow as it requires extra indirection to find corresponding method in the actual class. Abstract classes are fast.
11. Accessibility modifier(Public/Private/protected) is allowed for abstract class. Interface doesn’t allow accessibility modifier.
12. Abstract scope is upto derived class.
13. Interface scope is upto any level of its inheritance chain.

Object Class

- The object is universal super class. This class sits at the top of the class hierarchy tree in the Java development environment.
- Every class in the Java system is a descendent (direct or indirect) of the Object class.
- The Object class defines the basic state and behavior that all objects must have, such as the ability to compare oneself to another object, to convert to a string, to return the object's class etc.
- You can use a variable of type Object to refer to objects of any type:

```
Object obj = new Employee("abc", 15000);
```

- Of course, a variable of type Object is only useful as a generic holder for arbitrary values. If you have some knowledge about the original type and then apply a cast:

```
Employee e = (Employee) obj;
```

Wrapper Classes

- All primitive types have class counterparts. For example, a class Integer corresponds to the primitive type int.
- These kinds of classes are usually called wrappers.
- The wrapper classes have obvious names: Integer, Long, Float, Double, Short, Byte, Character, Void, and Boolean.
- The first six inherit from the common superclass Number.
- The wrapper classes are immutable you cannot change a wrapped value after the wrapper has been constructed.
- They are also final, so you cannot subclass them.