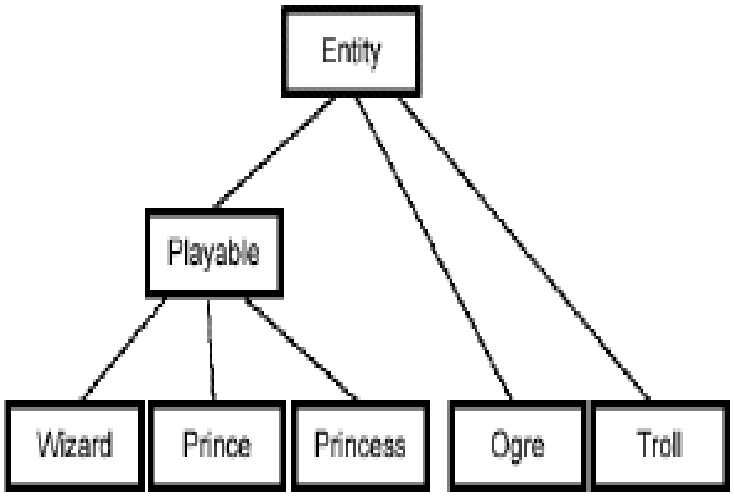
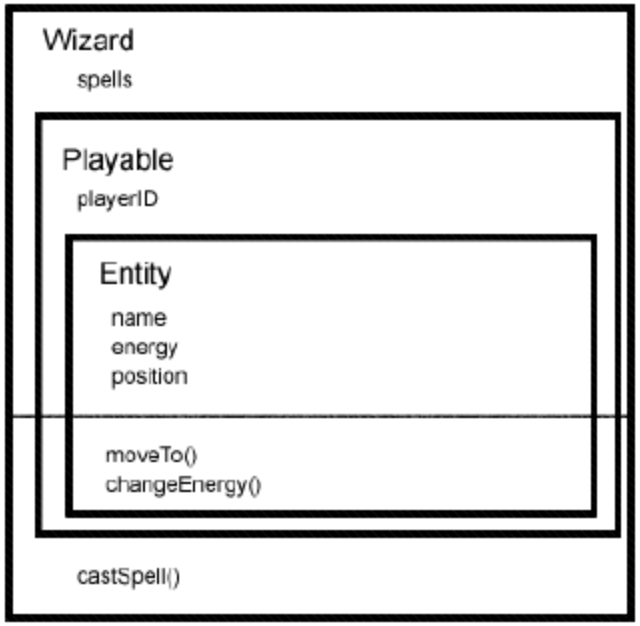
INHERITANCE && POLYMORPHISM

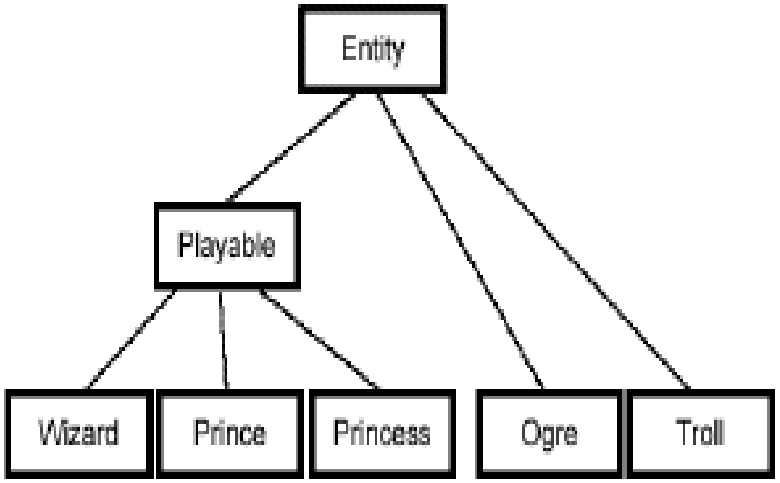
Inheritance

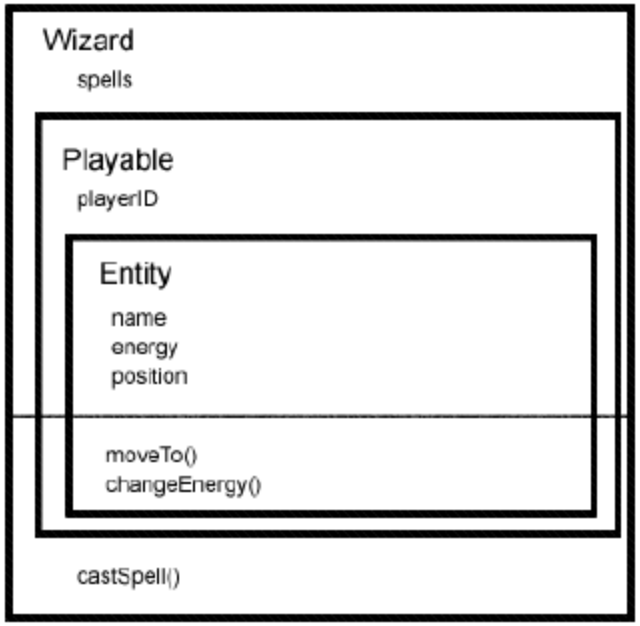
- A subclass inherits all the variables and methods from its superclasses, including its immediate parent as well as all the ancestors.
- It is important to note that a subclass is not a "subset" of a superclass.
- In contrast, subclass is a "superset" of a superclass.
- It is because a subclass inherits all the variables and methods of the superclass
- in addition, it extends the superclass by providing more variables and methods.





- Any Wizard object contains all the elements inside its box, including those of the base classes
- so, for example, the complete set of properties in a Wizard object is:
  - name
  - energy
  - position
  - playerID
  - spells





- a Wizard reference to a Wizard object has access to any public elements from any class in the inheritance chain from Object to Wizard
- code inside the Wizard class has access to all elements of the base classes (except those defined as private in the base class -those are present, but not directly accessible)
- Note: although it appears that a base class object is physically located inside the derived class instance, it is not actually implemented that way→this is logical representation

Private access in inheritance

- instances of the subclass will have copies of a private fields of the parent class.
- they will however not be visible to the subclass, so the only way to access them is via methods of the parent class.
- private fields are designed to be accessible on the class where its declared.
- The only way for them to be accessible outside the class (that includes the derived class(es)) is through public methods.
- If you want to make the members in the base class inherited by the derived class, use "protected" instead of private.
- Protected Access modifier allows the member to be inherited and have it assigned its own access modifier.

Example from Eclipse

- Point, Point3D classes

Method Overriding & Variable Hiding

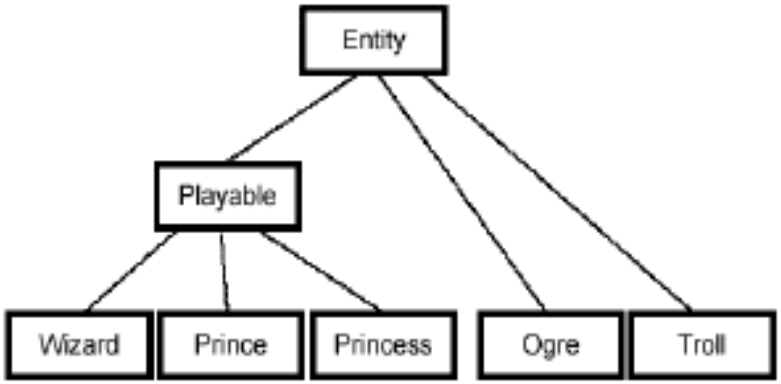
- A subclass inherits all the member variables and methods from its superclasses (the immediate parent and all its ancestors).
- It can use the inherited methods and variables as they are.
- It may also override an inherited method by providing its own version, or hide an inherited variable by defining a variable of the same name.
- Overriding methods, hiding variables

Hiding Fields

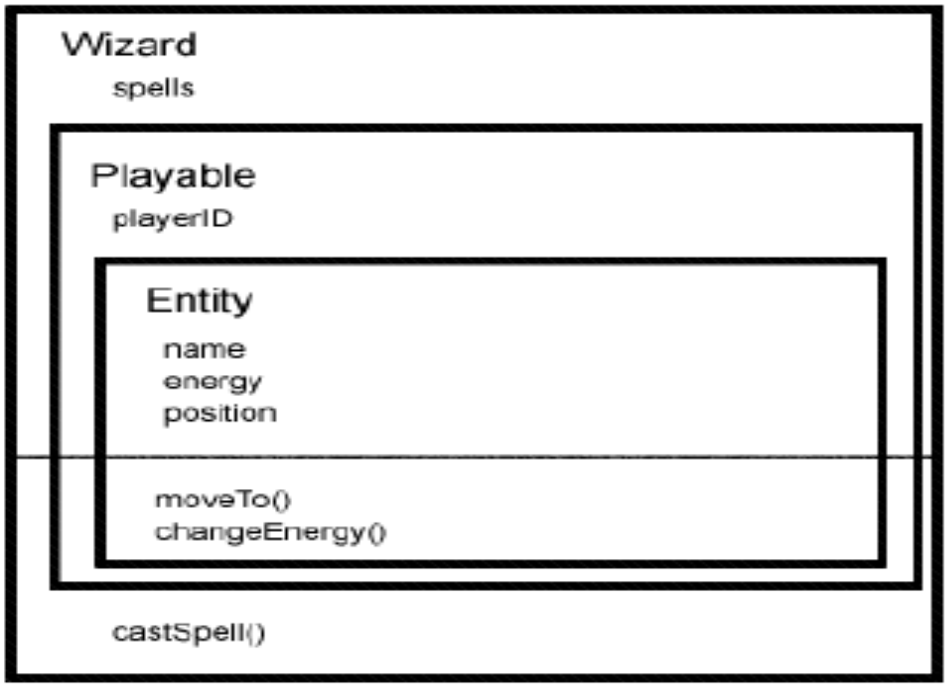
- Within a class, a field that has the same name as a field in the superclass hides the superclass's field, even if their types are different.
- Within the subclass, the field in the superclass cannot be referenced by its simple name.
- Instead, the field must be accessed through super.
- Generally speaking, we don't recommend hiding fields as it makes code difficult to read.

Inheritance and Polymorphism

- If a derived class extends a base class, it is not only considered an instance of the derived class, but an instance of the base class
- the compiler knows that all the features of the base class were inherited, so they are still there to work in the derived class
- This demonstrates what is known as an *IsA relationship - a derived class object Is A base class instance*
- it is an example of *polymorphism - that one reference can store several different types of objects*
- *Reference to base class can store reference to all sub classes*



- for example, in the game example above, for any character that is used in the game, an Entity reference variable could be used, so that at runtime, any subclass can be instantiated to store in that variable
  - Entity shrek = new Ogre();
  - Entity merlin = new Wizard();
- for the player's character, a Playable variable could be used
  - Playable charles = new Prince();



- When this is done, however, the only elements immediately available through the reference are those known to exist; that is, those elements defined in the *reference type object*
- the compiler decides what to allow you to do with the variable based upon the type *declared for the variable*
- Entity merlin = new Wizard();
- **merlin.moveTo() would be legal,** since that element is guaranteed to be there
- **merlin.castSpell() would not be legal,** since the definition of Entity does not include it, even though the actual object referenced

Dynamic Method Invocation

- When a method is called through a reference, the JVM looks to the actual class of the instance to find the method.
- If it doesn't find it there, it backs up to the ancestor class (the class this class extended) and looks there (and if it doesn't find it there, it backs up again, potentially all the way to Object).
- Sooner or later, it will find the method, since if it wasn't defined somewhere in the chain of inheritance, the compiler would not have allowed the class to compile.
- In this manner, most derived version of the method will run, even if you had a base class reference.
- So an Entity reference could hold a Wizard, and when the move method is called, the Wizard version of move will run.

Substitutability

- A subclass possesses all the attributes and operations of its superclass that are visible to it
  - because a subclass inherited all the visible attributes and operations from its superclass
- This means that a subclass object can do whatever its superclass can do.
- As a result, we can *substitute* a subclass instance when a superclass instance is expected, and everything shall work fine. This is called ***substitutability***.
- fields are not polymorphic therefore they are printed according to reference type

Example 1
What will be the output?

```
class Super {
    String s = "Super";
}
class Sub extends Super {
    String s = "Sub";
}

public class FieldOverriding {

    public static void main(String[] args) {
        Sub c1 = new Sub();
        System.out.println(c1.s);
        Super c2 = new Sub();
        System.out.println(c2.s);
    }
}
```

Example 2

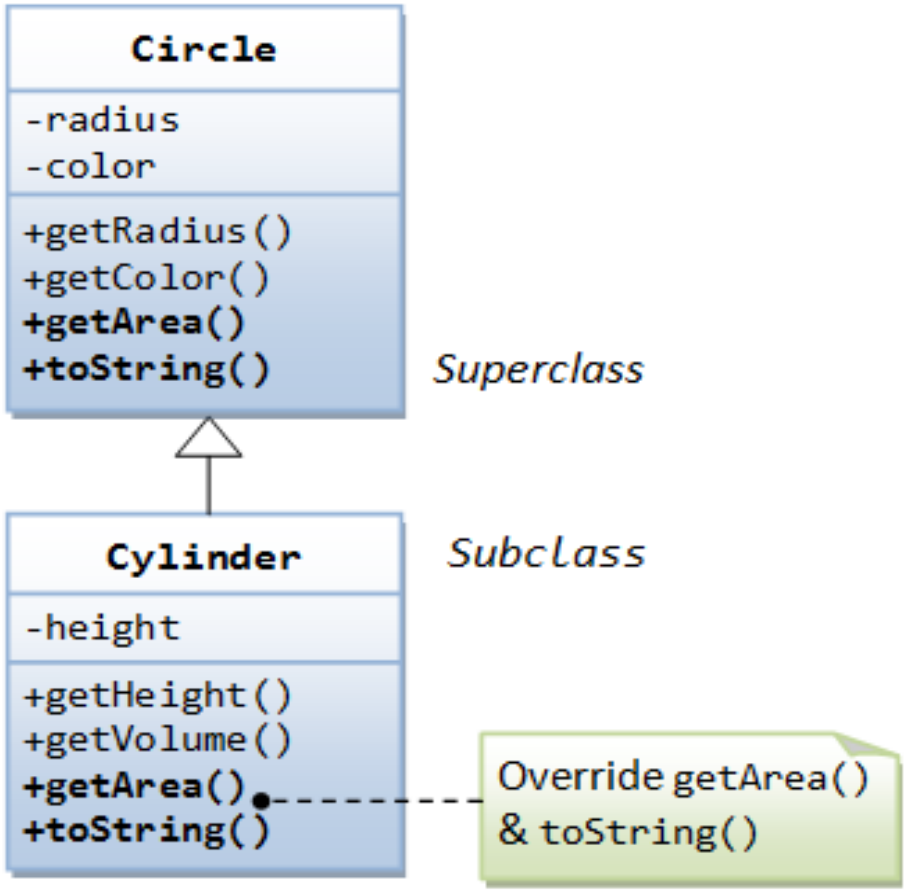
What will be the output?

```
public class A{
        protected int myvalue = 1;
}
public class B extends A{
        private int myvalue = 2;
}
public class C extends A{
        private int myvalue = 3;
}
......
ArrayList< A > myList= new ArrayList();
myList.add(new B());
myList.add(new C());
for(int i =0;i<myList.size();i++)
        System.out.println("value is:" + myList.get(i).myvalue);
```

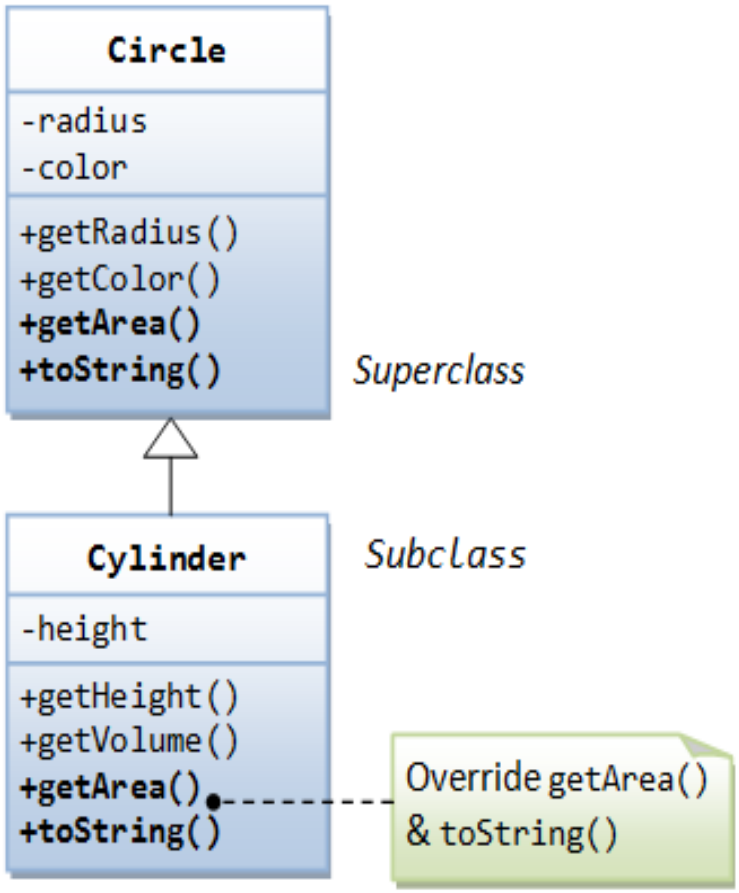
Example 3
What will be the output?

```
public class A{
        protected int myvalue = 1;
}
public class B extends A{
        public B() {
                myvalue = 2;
        }
}
public class C extends A{
        public C() {
                myvalue = 3;
        }
}
.....
        ArrayList<A> myList= new ArrayList();
        myList.add(new B());
        myList.add(new C());
        for(int i =0;i<myList.size();i++)
                System.out.println("value is:" + myList.get(i).myvalue);
```

- Circle c1 = new Cylinder(5.0);
- You can invoke all the methods defined in the Circle class for the reference c1, (which is actually holding a Cylinder object),
- **e.g. c1.getRadius()and c1.getColor().**
- This is because a subclass instance possesses all the properties of its superclass.
- However, you cannot invoke methods defined in the Cylinder class for the reference c1, **e.g. c1.getHeight() and c1.getVolume().**



- This is because c1 is a reference to the Circle class, which does not know about methods defined in the subclass Cylinder.
- c1 is a reference to the Circle class, but holds an object of its subclass Cylinder.
- **The reference c1, however, *retains its internal identity*.**
- In our example, the subclass Cylinder overrides methods getArea() and toString().
- c1.getArea() or c1.toString() invokes the *overridden* version defined in the subclass Cylinder, instead of the version defined in Circle.
- This is because c1 is in fact holding a Cylinder object internally.



Upcasting a Subclass Instance to a Superclass Reference

- Substituting a subclass instance for its superclass is called "*upcasting*".
- Upcasting is *always safe* because a subclass instance possesses all the properties of its superclass and can do whatever its superclass can do.
- The compiler checks for valid upcasting and issues error "incompatible types" otherwise.
- For example,
- Circle c1 = new Cylinder(); // OK, Cylinder is a Circle.

  Circle c2 = new String(); // Compilation error: incompatible types

Downcasting a Substituted Reference to Its Original Class

- You can revert a substituted instance back to a subclass reference. This is called "*downcasting*".
- Downcasting requires *explicit type casting operator*
- Downcasting is not always safe, and throws a runtime ClassCastException if the instance to be downcasted does not belong to the correct subclass.
- For example,

```
Circle c1 = new Cylinder(5.0); // upcast is safe
Cylinder aCylinder = (Cylinder) c1; // downcast needs the
    casting operator
```

Downcasting a Substituted Reference to Its Original Class

- Compiler may not be able to detect error in explicit cast, which will be detected only at runtime.
- For example, for any other Point class

```
Circle c1 = new Circle(5);
Point p1 = new Point();
c1 = p1; // compilation error: incompatible types (Point is not a subclass of Circle)
c1 = (Circle)p1; // runtime error: java.lang.ClassCastException: Point cannot be casted to Circle
```

EXAMPLE 4

```
class Parent {
        int x = 5;
        public void method(){
                System.out.println("Parent"+ x); }
        }
 }
  public class Child extends Parent{
        int x = 4;
        public void method(){
                System.out.println("Child"+ x);
        }
        public static void main(String[] args){
                Parent p = new Child();
                System.out.println(((Child) p).x);
                System.out.println(p.x);
        }

   }
```

The "instanceof" Operator

- Java provides a binary operator called **instanceof** which returns true if an object is an instance of a particular class.
- The syntax is as follows:

  anObject **instanceof** *aClass*

```
Circle c1 = new Circle();
System.out.println(c1 instanceof Circle); // true
if (c1 instanceof Circle) {
        ......
}
```

The "instanceof" Operator

- An instance of subclass is also an instance of its superclass.
- For example, for a class Cylinder that extends Circle;

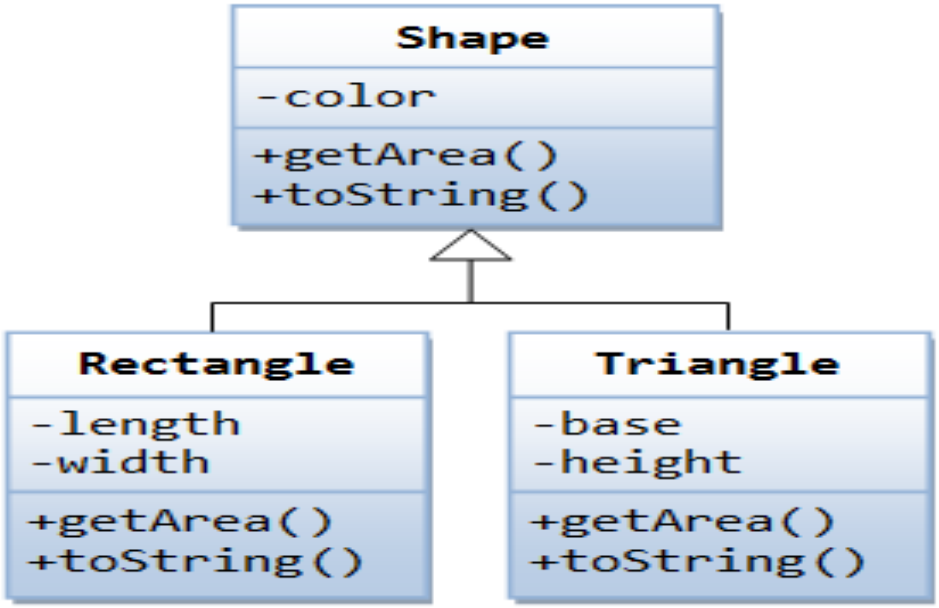
```
Circle c1 = new Circle(5);

Cylinder cy1 = new Cylinder(5, 2);
System.out.println(c1 instanceof Circle);
System.out.println(c1 instanceof Cylinder);
System.out.println(cy1 instanceof Cylinder);
System.out.println(cy1 instanceof Circle);

Circle c2 = new Cylinder(5, 2);

System.out.println(c2 instanceof Circle);
System.out.println(c2 instanceof Cylinder);
```

Summary of Polymorphism

- A subclass instance processes all the attributes and operations of its superclass.
- When a superclass instance is expected, it can be substituted by a subclass instance.
- In other words, a reference to a class may hold an instance of that class or an instance of one of its subclasses - it is called substitutability.
- If a subclass instance is assign to a superclass reference, you can invoke the methods defined in the superclass only.
- You cannot invoke methods defined in the subclass.
- However, the substituted instance retains its own identity in terms of overridden methods and hiding variables.
- If the subclass overrides methods in the superclass, the subclass's version will be executed, instead of the superclass's version.

Example on Polymorphism

- Polymorphism is very powerful in OOP to *separate the interface and implementation* so as to allow the programmer to *program at the interface* in the design of a *complex system*.
- Consider the following example.
- Suppose that our program uses many kinds of shapes, such as triangle, rectangle and so on.
- We should design a superclass called Shape, which defines the public interface (or behaviors) of all the shapes.
- For example, we would like all the shapes to have a method called **getArea()**, which returns the area of that particular shape.



- we have a problem on writing the getArea() method in the Shape class, because the area cannot be computed unless the actual shape is known.
- We shall print an error message for the time being.
- But we will solve this problem using abstract classes and interfaces
- We can then derive subclasses, such as Triangle and Rectangle, from the superclass Shape.
- The subclasses override the getArea() method inherited from the superclass, and provide the proper implementations for getArea().
- See from Eclipse!

```
Shape s1 = new Rectangle("red", 4, 5);
System.out.println(s1);
System.out.println("Area is " + s1.getArea());

Shape s2 = new Triangle("blue", 4, 5);
System.out.println(s2);
System.out.println("Area is " + s2.getArea());
```

- The beauty of this code is that *all the references are from the superclass* (i.e., *programming at the interface level*).
- You could extend your program easily by adding in more subclasses, such as Circle, Square, etc, with ease.

- However, the above definition of Shape class poses a problem, if someone instantiate a Shape object and invoke the getArea() from the Shape object

```
public class TestShape {
    public static void main(String[] args) {
      // Constructing a Shape instance poses problem!
      Shape s3 = new Shape("green");
      System.out.println(s3);
      System.out.println("Area is " + s3.getArea());
    }
}
```

  - This is because the Shape class is meant to provide a common interface to all its subclasses, which are supposed to provide the actual implementation.
  - We do not want anyone to instantiate a Shape instance. This problem can be resolved by using the so-called abstract class.

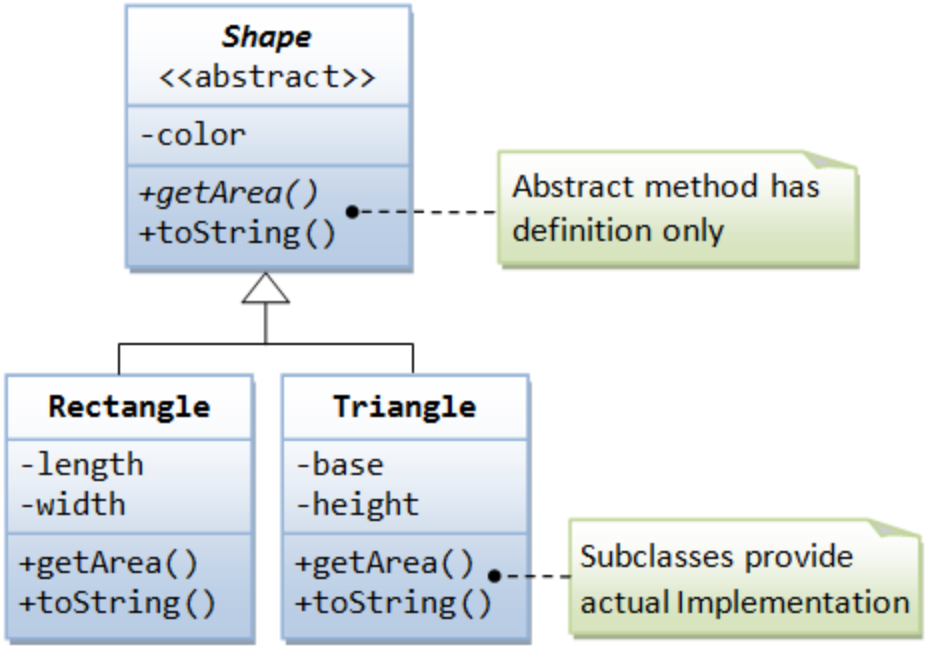
- An abstract method is a method with only signature (i.e., the method name, the list of arguments and the return type) without implementation (i.e., the method's body).
- You use the keyword abstract to declare an abstract method.
- For example, in the Shape class, we can declare abstract methods getArea(), draw(), as follows:
- **abstract** public class **Shape** {

```
        ......
        public abstract double getArea();
        public abstract void draw();
}
```

- Implementation of these methods is not possible in the Shape class, as the actual shape is not yet known.
- (How to compute the area if the shape is not known?)
- Implementation of these abstract methods will be provided later once the actual shape is known.
- These abstract methods cannot be invoked because they have no implementation.

Abstract Class

- A class containing one or more abstract methods is called an abstract class.
- An abstract class must be declared with a class-modifier abstract.
- Let rewrite our Shape class as an abstract class, containing an abstract method getArea() as in Eclipse!



In summary

- an abstract class provides *a template for further development*.
- The purpose of an abstract class is to provide a common interface (or protocol, or contract, or understanding, or naming convention) to all its subclasses.
- For example, in the abstract class Shape, you can define abstract methods such as getArea() and draw().
- No implementation is possible because the actual shape is not known.
- However, by specifying the signature of the abstract methods, all the subclasses are *forced* to use these methods' signature.
- The subclasses could provide the proper implementations.
- Coupled with polymorphism, you can upcast subclass instances to Shape, and program at the Shape level, i,e., program at the interface.
- The separation of interface and implementation enables better software design, and ease in expansion.

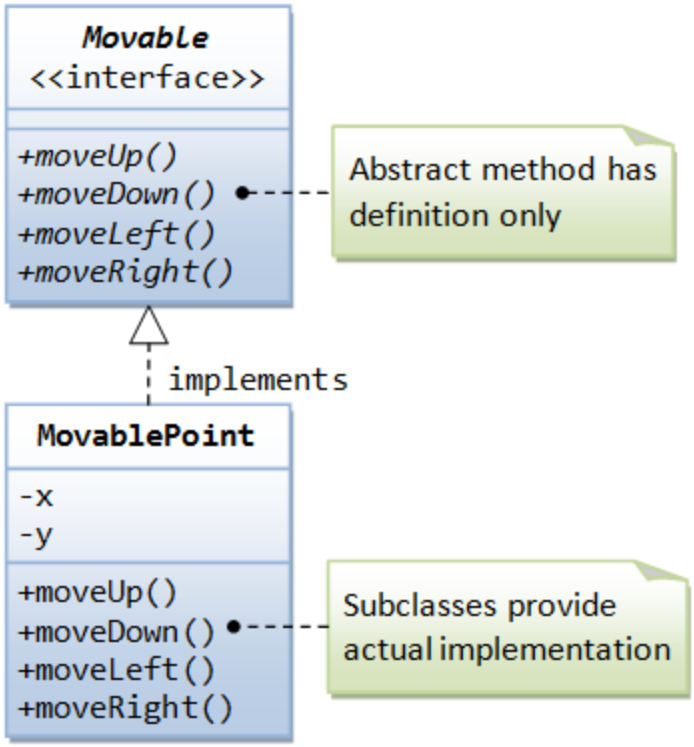
Rules

- An abstract method cannot be declared final, as final method cannot be overridden.
- An abstract method, on the other hand, must be overridden in a descendent before it can be used.
- An abstract method cannot be private (which generates a compilation error).
- This is because private method are not visible to the subclass and thus cannot be overridden.

Interfaces

- A Java interface is a *100% abstract superclass* which define a set of methods its subclasses must support.
- An interface contains only public *abstract methods* (methods with signature and no implementation) and possibly *constants* (public static final variables).
- You have to use the keyword "interface" to define an interface (instead of keyword "class" for normal classes).
- The keyword public and abstract are not needed for its abstract methods as they are mandatory.
- An interface is a *contract* for what the classes can do.
- It, however, does not specify how the classes should do it.

- Suppose that our application involves many objects that can move.
- We could define an interface called movable, containing the signatures of the various movement methods.



- Similar to an abstract class, an interface cannot be instantiated; because it is incomplete (the abstract methods' body is missing).
- To use an interface, again, you must derive subclasses and provide implementation to all the abstract methods declared in the interface.
- The subclasses are now complete and can be instantiated.
- To derive subclasses from an interface, a new keyboard "implements" is to be used instead of "extends" for deriving subclasses from an ordinary class or an abstract class.
- It is important to note that the subclass implementing an interface need to override ALL the abstract methods defined in the interface; otherwise, the subclass cannot be compiled.
- Example

Implementing Multiple interfaces

- Java supports only *single inheritance*.
- a subclass can be derived from one and only one superclass.
- Java does not support *multiple inheritance* to avoid inheriting conflicting properties from multiple superclasses.
- A subclass, however, can implement more than one interfaces.
- This is permitted in Java as an interface merely defines the abstract methods without the actual implementations and less likely leads to inheriting conflicting properties from multiple interfaces.
- In other words, Java indirectly supports multiple inheritances via implementing multiple interfaces.
- For example,
- public class **Circle extends Shape implements Movable, Displayable** {
- // One superclass but implement multiple interfaces .....
- }

One of the many uses of interfaces: to support multiple inheritance

- class Animal {
  void walk() { } ....
  void chew() { } //concentrate on this
  }

Now, Imagine a case where:

- class Reptile extends Animal {
  //reptile specific code here
  } //not a problem

  but,

- class Bird extends Animal {
  //other Bird specific code
  } **//now Birds cannot chew so this would a problem in the sense Bird classes can also call chew() method which is unwanted**

Better design would be:

- class Animal {
  void walk() { }
  //other methods
  }
- Animal does not have the chew() method and instead is put in an interface as :
- interface Chewable {
  void chew();
  }

  and have Reptile class implement this and not Birds (since Birds cannot chew) :
- class Reptile extends Animal implements Chewable { }
- class Bird extends Animal { }

- **public class Shape {**

  abstract public double getArea();

  }

Compilation error!! Why?

Example-5

```
interface A{
        int f();
}
interface B{
        int f();
}
class Test implements A, B{
        public static void main(String... args)
        {
        }
        public int f() {
        .......
        }
}
```

Example-6

```
interface A{
        int f();
}
interface B{
        void f();
}
class Test implements A, B{
        public static void main(String... args)
        {
        }
        public int f() {
        // from which interface A or B return 0;
        }
}
```

Example-7

```
class Super {
        public int number = 1;
        public char superText='a';
        public String getColor() {
                return "red";
        }
}

class Sub extends Super {
        public int number = 2;
        public char subText='b';
        public String getColor() {
                return "blue";
        }
}

public class Sample2 {
        public static void main(String[] args) {
                Super supersub = new Sub();
                System.out.println( supersub.getColor() + supersub.number +
                                        supersub.superText );
        }
}
```