



Polymorphism & OO Programming



Three Pillars of O-O Programming

Encapsulation:

an object contains both data and the methods that operate on the data. It may **expose** some of these members to the outside and hide others.

This design separates the **public interface** from the **implementation**, and enforces **data integrity**.

Inheritance:

one class can inherit attributes and methods from another class.

Polymorphism:

a behavior can be invoked on different kinds of objects, **without knowing** the type of object that will perform the behavior



Encapsulation

In OOP, encapsulation means that an object *contains both the data and the methods that operate on the data.*

- ❑ A Purse encapsulates the Coins and behavior of a purse.
- ❑ We can use a Purse without knowing *how* a Purse performs its job.
- ❑ We don't need to know *how* the Purse implements its methods or *how* it stores Coins.



Polymorphism

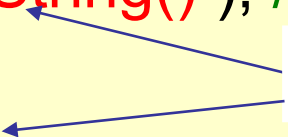
Polymorph = poly + morph
= many + forms

We can invoke a behavior **without knowing** what kind of object will perform the behavior.

Many kinds of objects can respond to the same message.

```
Object x;  
x = new Double(Math.PI);  
System.out.println( x.toString() ); // toString() of Double  
x = new Date( );  
System.out.println( x.toString() ); // toString() of Date
```

Polymorphism





Polymorphism Example

x.toString() always calls toString of the **correct** class.
We don't need to know the type of object that x refers to.

```
Object x = new Date( );  
System.out.println( x ); // calls x.toString()  
        // invokes toString of Date class  
  
x = new Double(Math.PI);  
System.out.println( x ); // calls x.toString()  
        // invokes toString of Double class
```



any kind of Object



Polymorphism Example

`Arrays.sort()` - a method in `java.util.Arrays` class

```
// sort array of String
String[] words = new String[] { "dog", "bird", "ant" };
Arrays.sort( words ); ← uses string1.compareTo( string2 )
// sort array of Student
Student [] students = new Student[] { ... };
Arrays.sort( students ); uses student.compareTo(student)
```

`Arrays.sort` can sort *any kind of objects* if they have a `compareTo` method. ("X implements Comparable")

`Arrays.sort` *doesn't know* the *type of object* that performs `compareTo`.



Enabling Polymorphism

The *key* to polymorphism asking an object to do something (call its method) **without knowing the *kind*** of object.

```
Object a = ...;  
a.toString( );  
a.compareTo( b );
```

a.toString() always works for any kind of object. Why?

a.compareTo(b) doesn't always work. Why?

- How can we tell Java that a "has a compareTo method" even if we don't know what class a belongs to?



Two Ways to Enable Polymorphism

Java provides two ways to "guarantee" that a class object has some behavior.

1. Inheritance

If a *superclass* provides a behavior, then all its *subclasses* *inherit* that behavior.

Subclasses can use the method code of the parent class, or *override* it with their own implementation.

2. Interface

An interface is a *specification* of a behavior.

A class that *implements* an *interface* must provide that behavior.



Methods from Object

Every Java class is a **subclass** of the **Object** class.

The **Object** class defines basic behavior (see below).

Most classes will override these methods to provide useful implementations.

Every class **inherits**
these methods
automatically.

So, we can *always* use
`obj.toString()` or
`obj.equals(obj2)`
for any kind of object.

Object

`equals(Object): boolean`

`getClass(): Class`

`hashCode(): int`

`toString(): String`

`notify()`

`wait()`

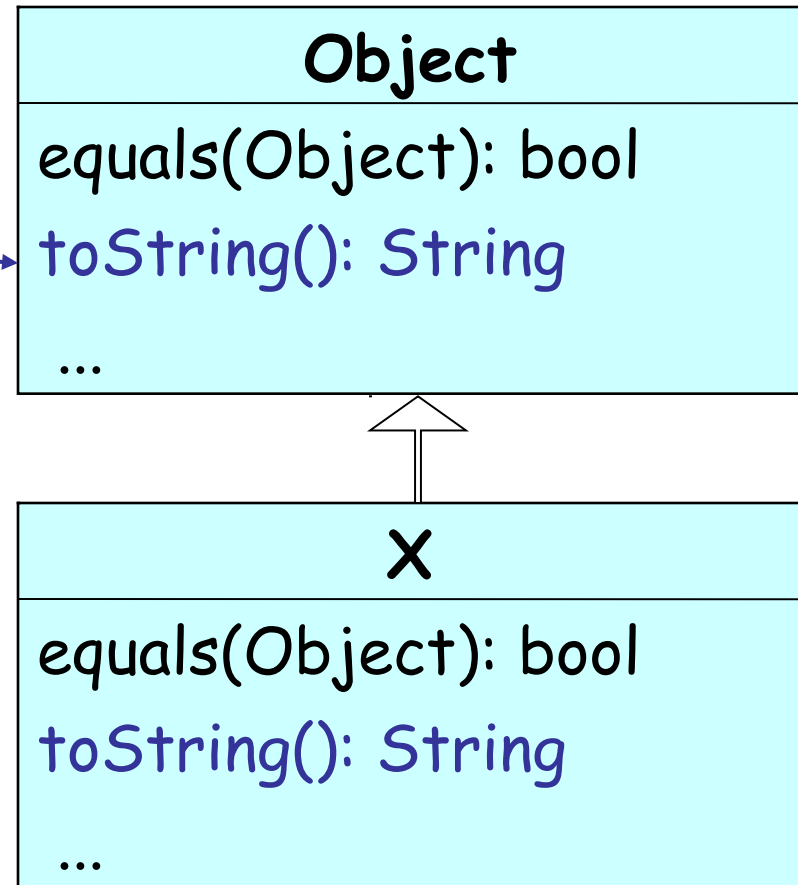
...

Inheritance and Polymorphism

Since every class inherits the non-private behavior of its parent class, we can treat objects from subclasses just like the parent.

Every object is guaranteed to have a `toString()` method.

`System.out.print(obj)` can print any object by calling `obj.toString()`, without knowing the object's type.





Interface

A Java *interface* specifies behavior which will be provided by classes that *implement* the interface.

- Interface is a *specification* for a required behavior, *without an implementation*.

Example: USB interface specifies (a) connector size, (b) electrical properties, (c) communications protocol, ...

Anyone can *implement* the USB interface on their device.

java.util.Comparable interface

```
public interface Comparable {  
    /**  
     * compare two objects of type T.  
     * @param obj is another object to  
     *         compare to this object.  
     * @return ...  
     */  
    public int compareTo(Object obj);  
}
```

abstract method = method **signature**, but no
implementation



Comparable example

```
public class Coin  
    implements Comparable<Coin> {
```

Declare that this class has the behavior specified by the Comparable interface.

```
    int compareTo(Coin other) {  
        if (other == null) return -1;  
        return this.value - other.value;  
    }  
}
```

Implement the required method.



Polymorphism using Interface

- A class can *guarantee* to have a compareTo method by **implementing** the Comparable interface.
- Arrays.sort() can sort **any kind** of objects that implement Comparable

Example:

Date **implements** Comparable (interface)

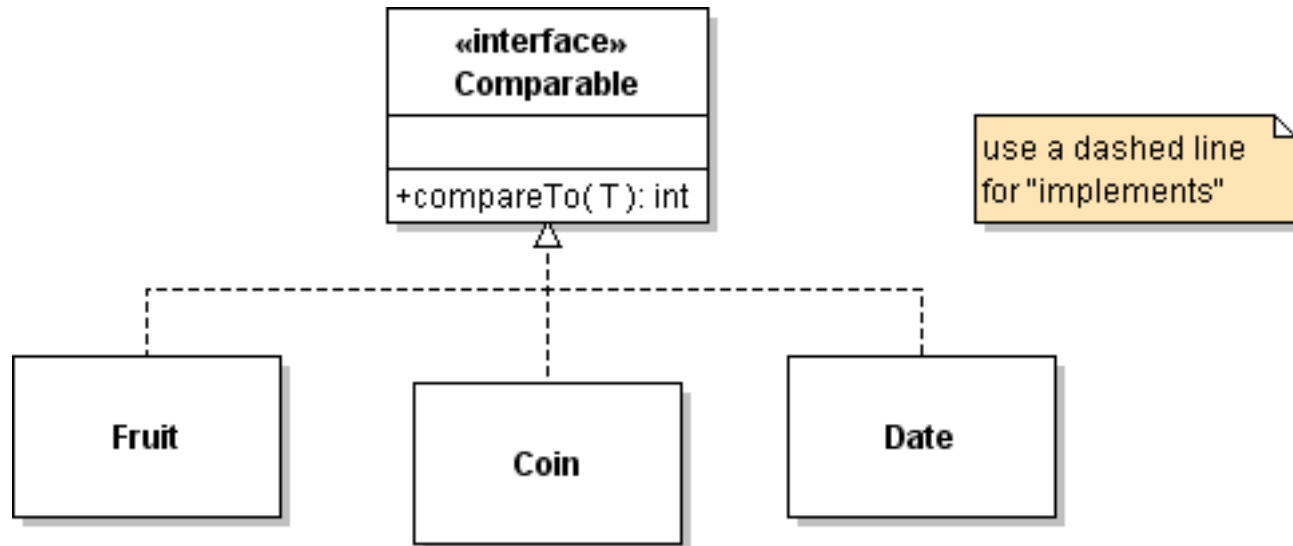
String **implements** Comparable

Double **implements** Comparable

Coin **implements** Comparable

=> we can call compareTo of any of these objects.

Using interface for Polymorphism



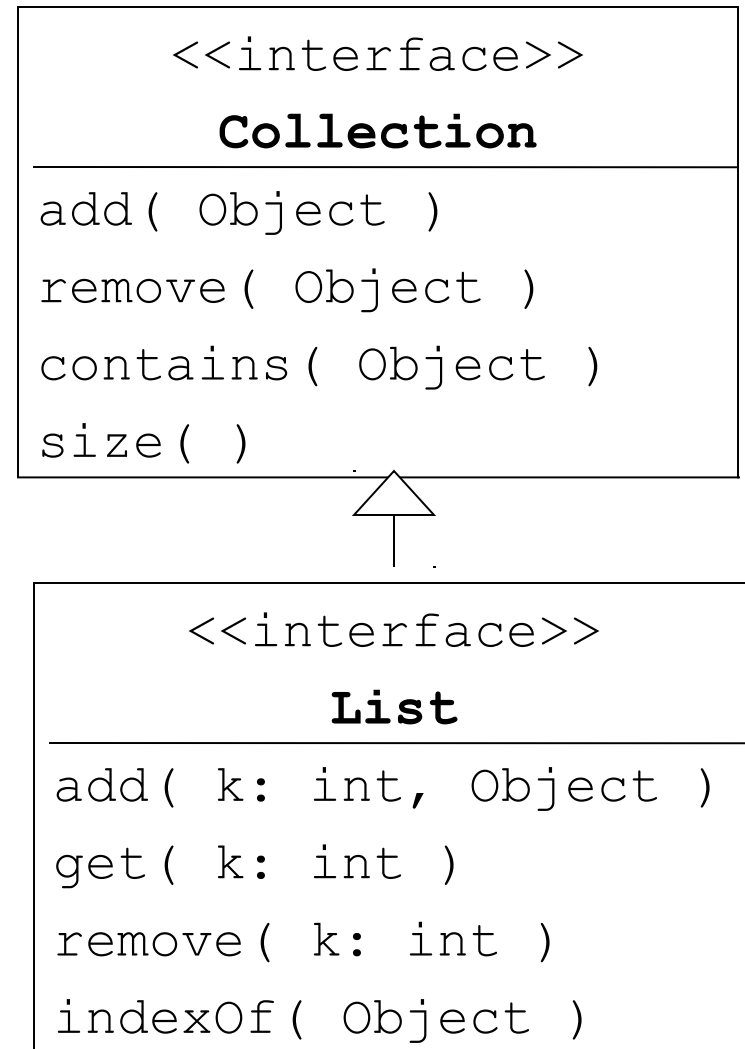
```
Comparable x = new Coin();
Comparable y = new Coin();
x.compareTo( y )    // calls compareTo of Coin
x = new Fruit( );
y = new Fruit();
x.compareTo( y )    // calls compareTo of Fruit
```



Polymorphism and Collections

Collection specifies methods
for all types of collections
(List, Set, ...)

List interface specifies
additional methods that Lists
must have.





What Polymorphism is called...

"Don't ask what kind"

Not Polymorphism:

```
if (shape instanceof Circle) {  
    ((Circle)shape).getCenter();  
}
```

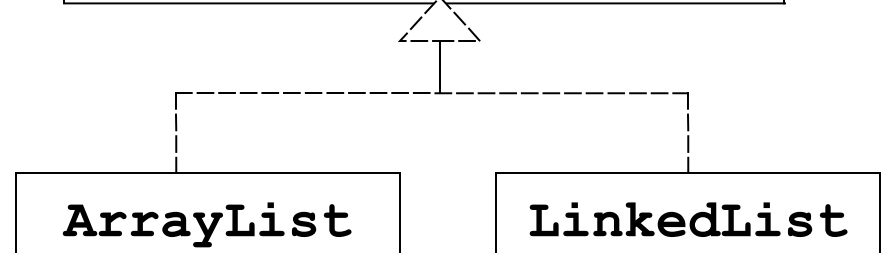


Polymorphism and Collections

The List interface defines the public methods for all kinds of lists.

```
<<interface>>  
  
    List  
    -----  
    add( k: int, Object )  
    get( k: int )  
    remove( k: int )  
    indexOf( Object )
```

ArrayList and LinkedList are specific implementations of the list interface.



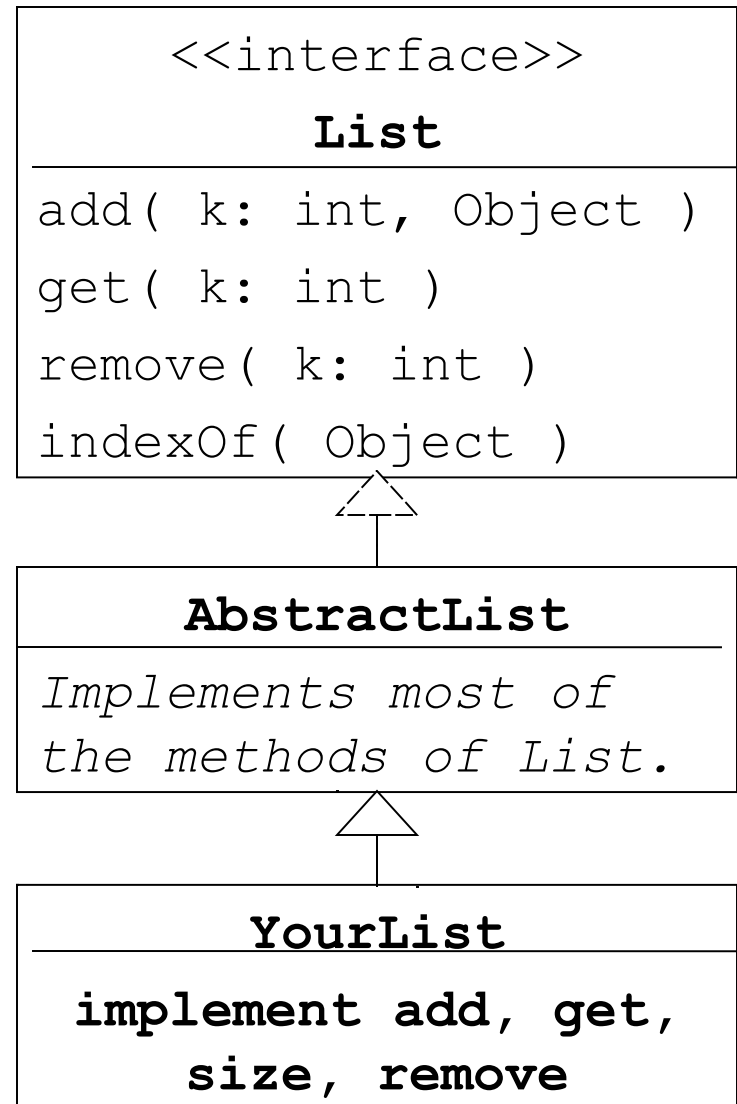


Interface and Inheritance

The List interface has 25 methods. Its a lot of work to implement all of them.

AbstractList implements most of the methods for you.

Your List class implements only a few methods.





Enabling Polymorphism (1)

- To enable polymorphism, we have to *guarantee* that the object always has the method we invoke.

`x.toString()` // the class must have toString method

- How can we ensure this??

1. **Use inheritance:** a subclass has all the methods of its superclass.

Example: Object class defines toString().

Since every class is a subclass of Object, Java knows that every object has a toString()



Methods from Object

Every object has the methods defined in Object.
Usually the classes will override these methods to provide useful implementations.

Object

`equals(Object): boolean`

`getClass(): Class`

`hashCode(): int`

`toString(): String`

`notify()`

`wait()`

...



Enabling Polymorphism (2)

2. **Use an Interface:** an interface specifies a required behavior without implementing it.

Every class that implements the interface is promising that it provides the interface's behavior.

Example:

Date implements Comparable (interface)

String also implements Comparable (interface).

Therefore, we can sort an array of Date or String.

java.util.Comparable (interface)


```
public interface Comparable<T> {  
    /**  
     * compare two objects of type T.  
     * @param obj is another object to compare  
     *         to this object.  
     * @return ...  
     */  
    public int compareTo(T obj);  
}
```

abstract method = method signature, but no implementation



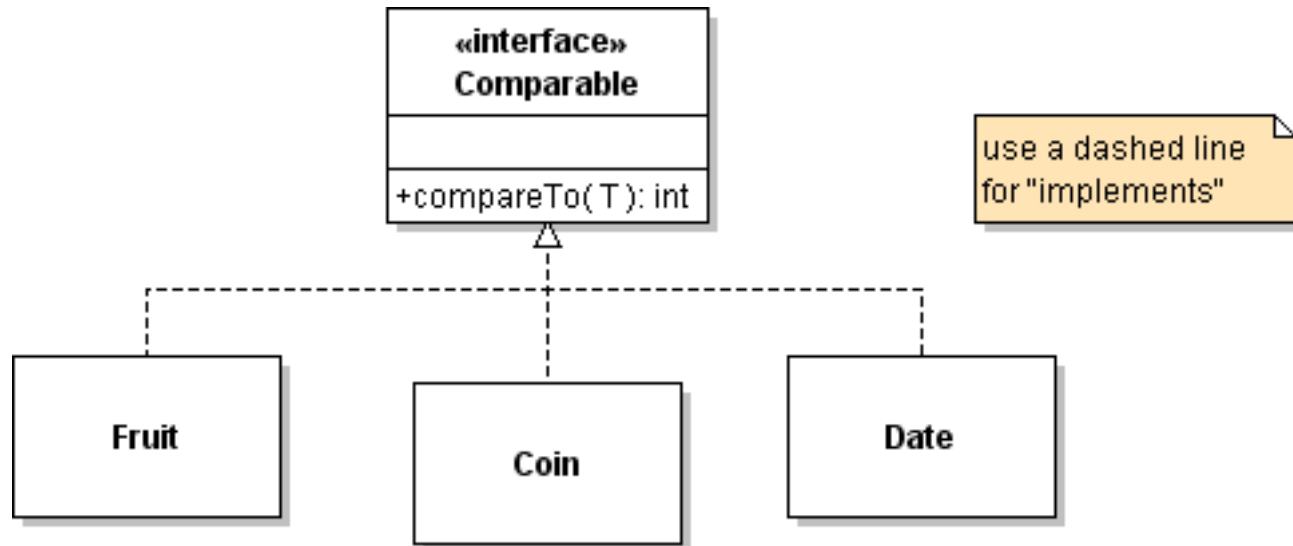
Comparable example

```
public class Coin  
    implements Comparable<Coin> {  
}
```



Declare that this class has a **compareTo** method as required by the interface.

Using interface for Polymorphism



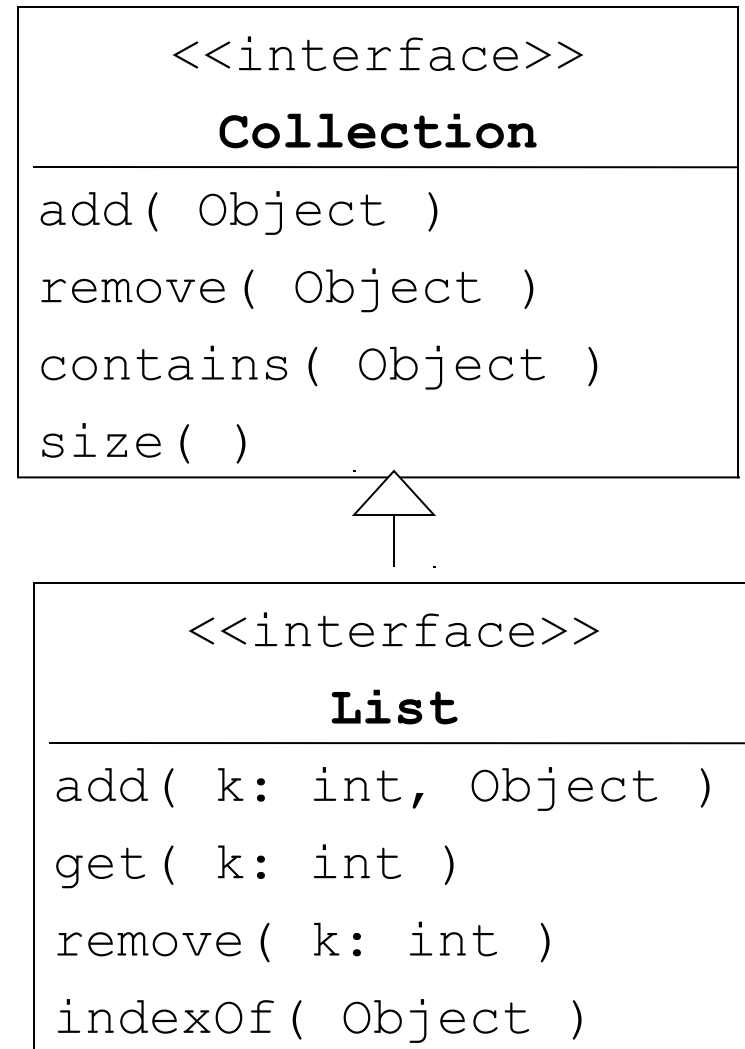
```
Comparable x = new Coin();
Comparable y = new Coin();
x.compareTo( y )    // calls compareTo of Coin
x = new Fruit( );
y = new Fruit();
x.compareTo( y )    // calls compareTo of Fruit
```



Polymorphism and Collections

Collection specifies methods
for all types of collections
(List, Set, ...)

List interface specifies
additional methods that Lists
must have.



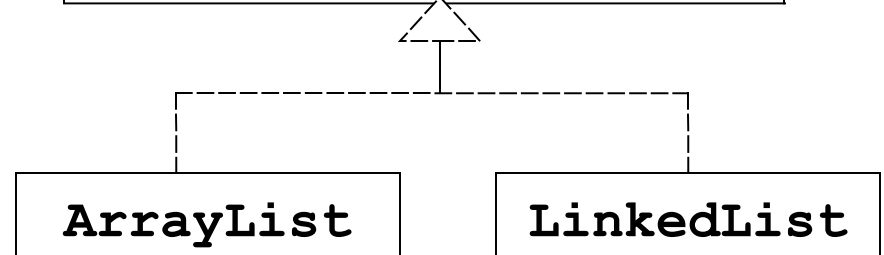


Polymorphism and Collections

The List interface defines the public methods for all kinds of lists.

```
<<interface>>  
  
    List  
    -----  
    add( k: int, Object )  
    get( k: int )  
    remove( k: int )  
    indexOf( Object )
```

ArrayList and LinkedList are specific implementations of the list interface.



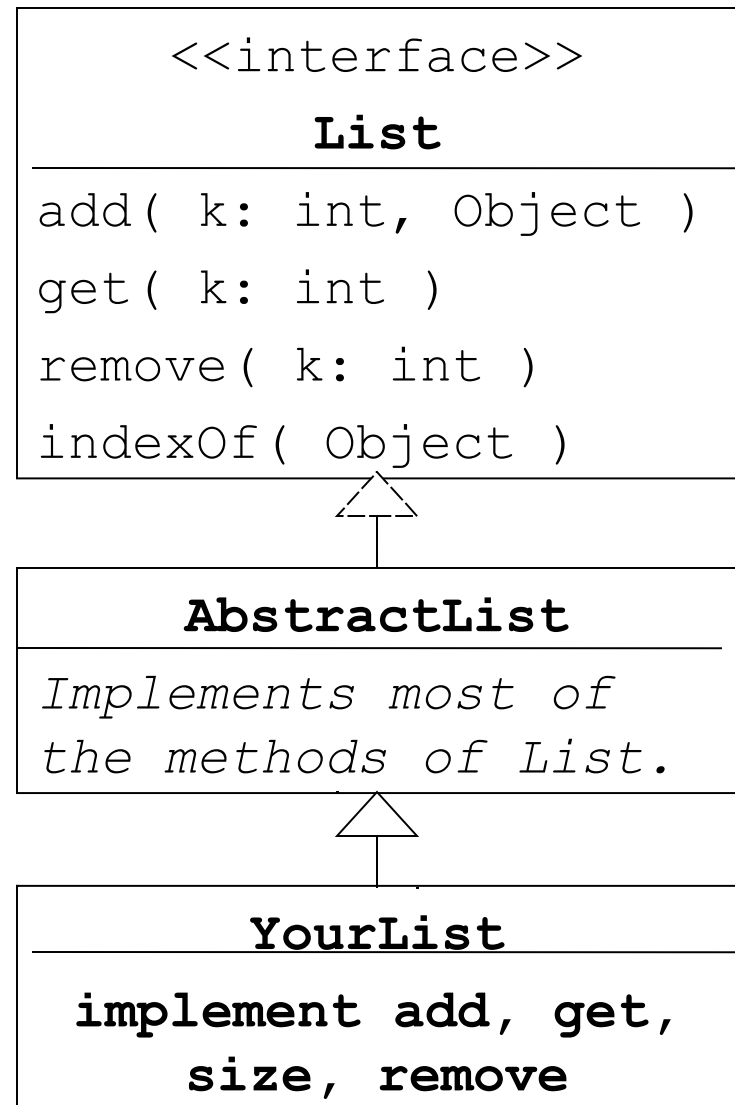


Interface and Inheritance

The List interface has 25 methods. Its a lot of work to implement all of them.

AbstractList implements most of the methods for you.

Your List class implements only a few methods.





Inheritance

One class can "inherit" all the behavior and attributes of another class.

- The subclass can "override" (redefine) any methods its wants to change.

Example: we override toString(), equals() in our classes.

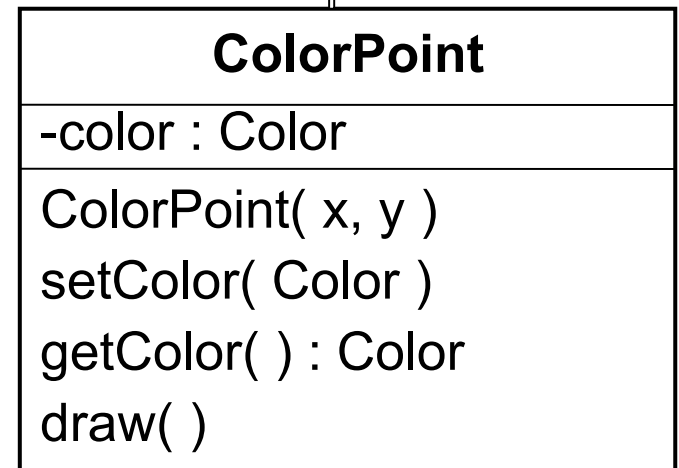
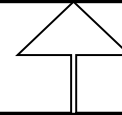
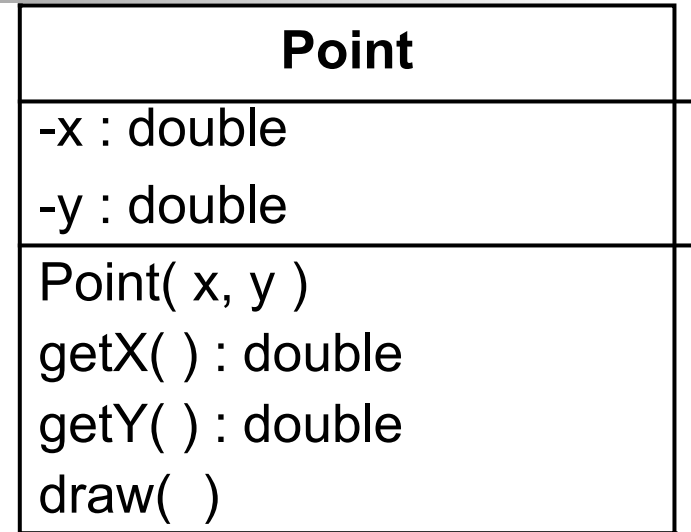
```
public class CountingGame extends Game {  
    // inherits attributes from the Game class  
    // inherits methods from the Game class  
}
```

A Point Class

A Point **encapsulates** its state (x,y) and the behavior of a point.

ColorPoint **inherits** all the behavior of a Point.

ColorPoint can choose to **override** some of the behavior, e.g. `draw()`.





Inheritance Example

Design a JButton that toggles between ON and OFF

- ❑ OnOffButton is a **subclass** of JButton.
- ❑ OnOffButton can reuse all behavior of JButton, so we don't have to write them again
- ❑ We can add custom behavior to implement the on/off status and change appearance. constructor

```
class OnOffButton extends JButton {  
    private boolean on; // true if button on  
    public OnOffButton ( boolean on ) {  
        super( on ? "on" : "off" );  
        this.on = on ;  
    }  
    public boolean isOn( ) { return on; }  
}
```

Objects and Programs

An OO program consists of **objects** that interact with each other.

