

# Interface & Polymorphism





# Outlines

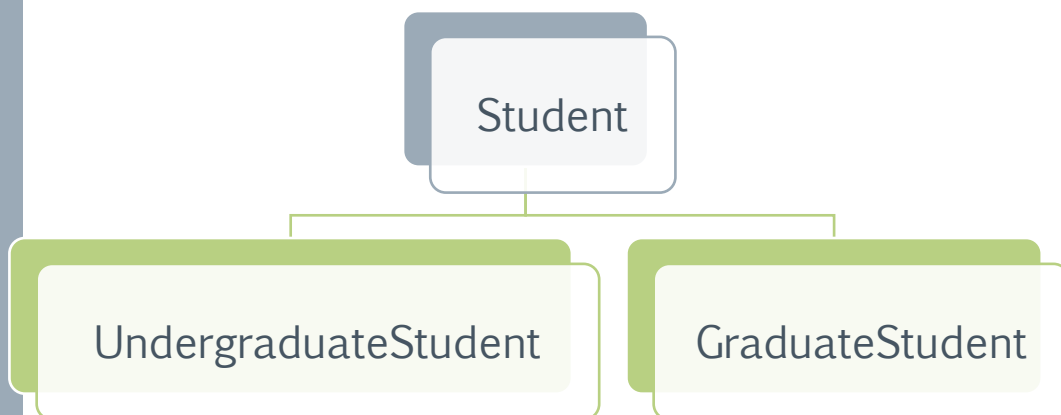
- › Review the concepts of inheritance and abstract class
- › Interface
- › Multiple inheritance
- › Polymorphism
- › Interface vs. Abstract
- › Case study: DataSet, Coin, BankAccount
  
- › Inner class
- › Some of Java's Most used Interfaces
  - Iterator, Cloneable, Serializable, Comparable
- › Tag/Marker Interface, Functional Interface



# Review the concepts of inheritance and abstract class: Inheritance

- › Inheritance allows data of one type to be treated as data of **a more general type**.
- › Components:
  - Base/parent class, Superclass
  - Derived/child class, subclass

```
public class UndergraduateStudent extends Student
public class GraduateStudent extends Student
```



- › Keyword `extends`
  - Achieve inheritance in Java
  - Can extends from only **one** superclass
- › Subclass contains data and methods defined in original superclass
  - Overriding data and methods
- › More concepts:
  - “protected” access modifier
  - Keyword “super”
  - “static” and “final” methods cannot be overridden.
  - Upcasting/downcasting

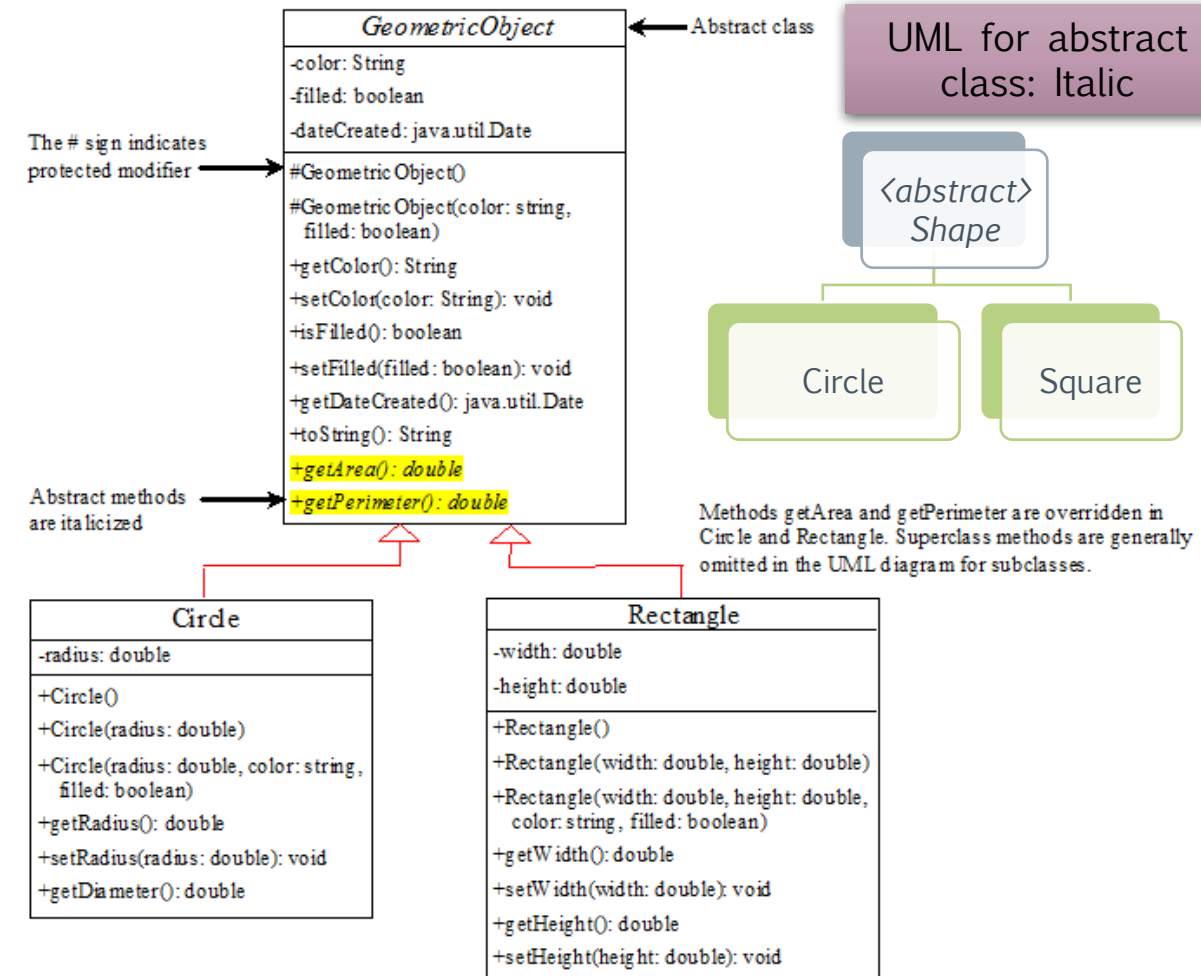


# Review the concepts of inheritance and abstract class: Abstract

- › Abstract is a concept to create “a template.” It can be applied to class and method.
- › Abstract class **cannot** be instantiated. The purpose of an abstract class is to function as a base for subclasses.
- › Abstract method has **no** implementation. It just has a method signature.

## Code

```
public abstract class MyAbstractClass {
    public abstract void abstractMethod();
}
```





# Interface

- › An interface is a classlike construct that contains **only** constants and abstract methods.
- › In many ways, an interface is similar to an abstract class, but the intent of an interface is to specify **behavior** for objects.
- › For example, you can specify that the objects are comparable, edible, cloneable using appropriate interfaces.
- › Usage
  - Declaration begins with **interface** keyword
  - Classes **implement** an interface (and its methods)
  - We **can't instantiate** an interface in java.



# Interface (cont.)

- › Interfaces **can't** have **constructors** because we can't instantiate them and interfaces can't have a method with body.
- › By default any **attribute (variable)** of interface is “**public static final**” (= **CONSTANT**), so we don't need to provide access modifiers to the attributes but if we do, compiler doesn't complain about it either.
- › By default interface **methods** are implicitly **abstract** and **public**, it makes total sense because the method don't have body and so that subclasses can provide the method implementation.
- › An interface **can't** extend any class, but it **can** extend another interface.

## Shape.java

```
public interface Shape {  
    // implicitly public, static and final  
    String LABEL = "Shape";  
    // interface methods are implicitly abstract and public  
    void draw();  
    double getArea();  
}
```

## Shape.java

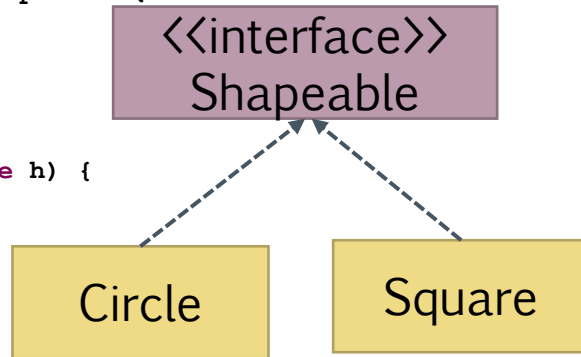
COMPUTER

```
public interface Shapeable {  
    // implicitly public, static and final  
    public String LABEL = "Shape";  
    // interface methods are implicitly abstract and public  
    void draw();  
    double getArea();  
}
```

Can implement > 1 interface  
(multiple-inheritance)

## Rectangle.java

```
public class Rectangle implements Shapeable {  
    private double width;  
    private double height;  
    public Rectangle(double w, double h) {  
        this.width = w;  
        this.height = h;  
    }  
    public void draw() {  
        System.out.println("Drawing Rectangle");  
    }  
    public double getArea() {  
        return this.height * this.width;  
    }  
}
```



## ShapeTest.java

```
public static void main(String[] args) {  
    Shapeable shape = new Circle(10);  
    shape.draw();  
    System.out.println("Area=" + shape.getArea());  
    shape = new Rectangle(10, 10);  
    shape.draw();  
    System.out.println("Area=" + shape.getArea());  
}
```

## Circle.java

```
public class Circle implements Shapeable {  
    private double radius;  
    public Circle(double r) {  
        this.radius = r;  
    }  
    public void draw() {  
        System.out.println("Drawing Rectangle");  
    }  
    public double getArea() {  
        return Math.PI * this.radius * this.radius;  
    }  
}
```

### Result

```
Drawing Rectangle  
Area=314.1592653589793  
Drawing Rectangle  
Area=100.0
```



# Interface (cont.)

- › A class implementing an interface must provide implementation for **all** of its method **unless it's an abstract class**.

## Shapeable.java

```
public interface Shapeable {  
    // implicitly public, static and final  
    public String LABEL = "Shape";  
    // interface methods are implicitly abstract and public  
    void draw();  
    double getArea();  
}
```

## ShapeAbs.java

```
abstract class ShapeAbs implements Shapeable{  
    public double getArea() {  
        return 0;  
    }  
}
```





# Benefits & Disadvantages

## BENEFITS

- › Provide a template for all the implementation classes, so it is good to code from interfaces since implementation classes can't remove the methods we are using.
- › Good for starting point to define type and create top level hierarchy in our code.
- › Support multiple-inheritance

## DISADVANTAGES

- › Need to choose interface methods very carefully at the time of designing our project because we can't add or remove any methods from the interface at later point of time
- › If the implementation classes has its own methods, we can't use them directly in our code.
  - To overcome this, we can use **typecasting (downcasting)** and use the method like this:

### Code

```
Shape s = new Circle(10);  
  
Circle c = (Circle) s;  
  
c.getRadius();
```

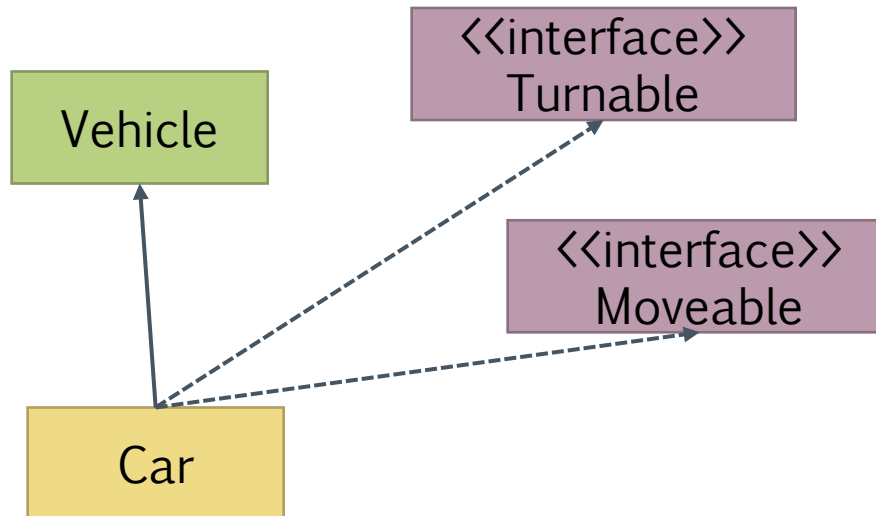


# Multiple inheritance

- › Java does **not allow** having a class extending from more than one class (multiple inheritance is not allowed)
- › A class can implement any number of interfaces, but extend at most one class
- › Interface refers to a protocol of behavior that **can be implemented by any class anywhere in the class hierarchy.**
- › **Conflict Interfaces:**
  - In rare occasions, a class may implement two interfaces with conflict information (e.g., two same constants with different values or two methods with same signature but different return type). **This type of errors will be detected by the compiler.**



# Multiple inheritance (cont.)



## Turnable.java

```
interface Turnable{

    void turnLeft();

    void turnRight();

}
```

## Moveable.java

```
interface Moveable{

    void forward();

    void backward();

}
```

## Car.java

```
public class Car extends Vehicle
    implements Turnable, Moveable {

    public void turnLeft() { ... }

    public void turnRight() { ... }

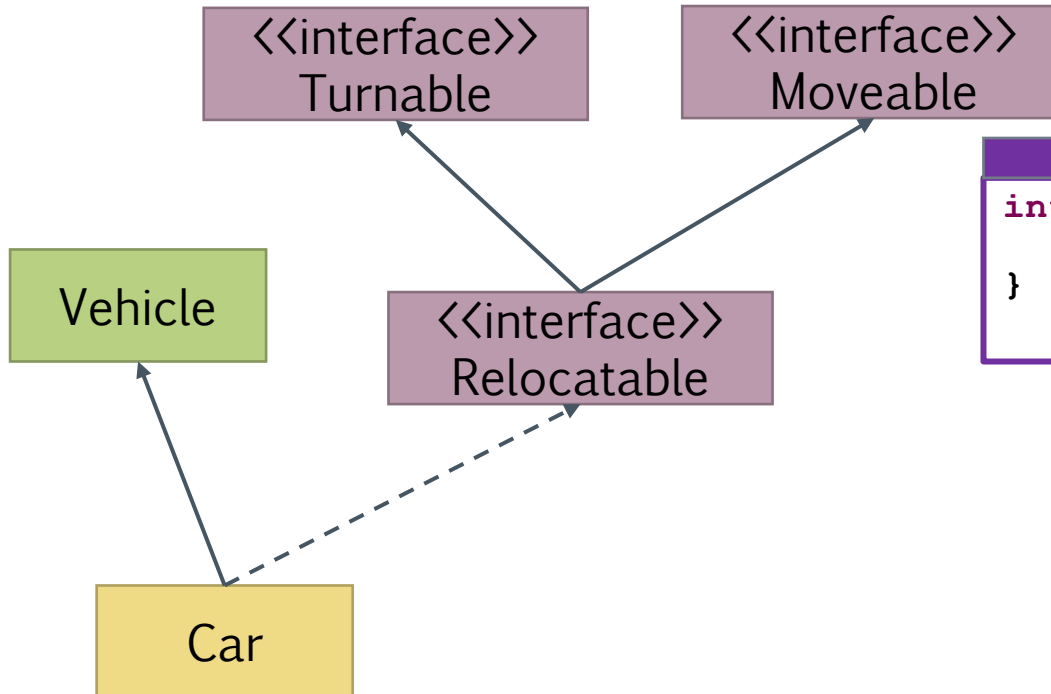
    public void forward() { ... }

    public void backward() { ... }

}
```



# Multiple inheritance (cont.)



## Interface & another interface

- Can extend
- **Cannot** implement

### Turnable.java

```
interface Turnable {
    void turnLeft();
    void turnRight();
}
```

### Moveable.java

```
interface Moveable {
    void forward();
    void backward();
}
```

### Moveable.java

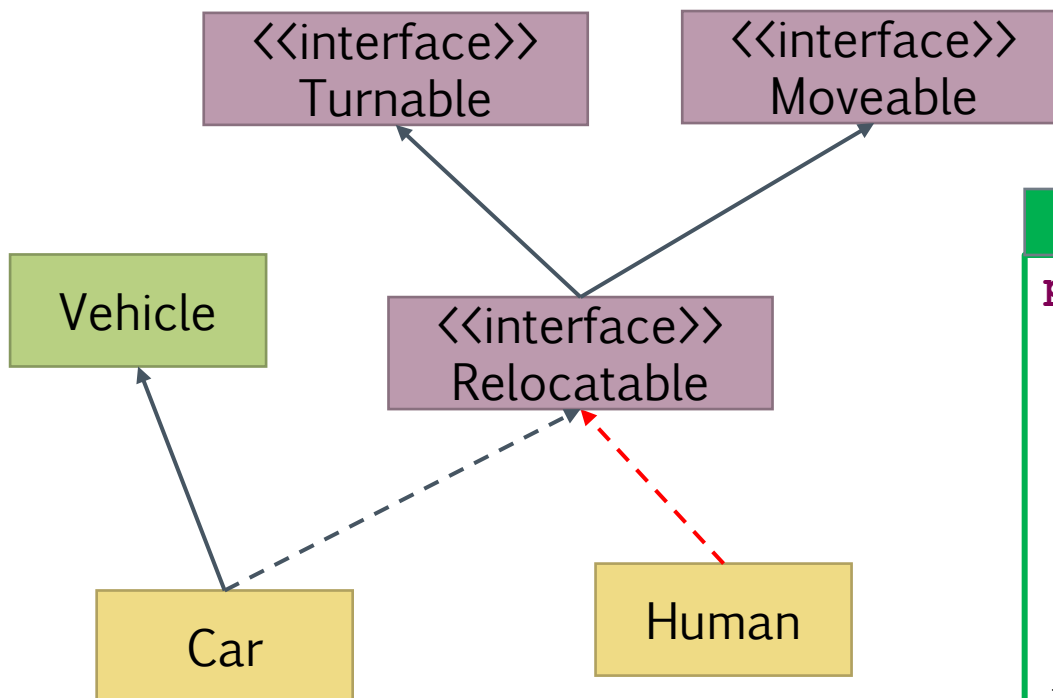
```
interface Relocatable extends Turnable, Moveable {
}
```

### Car.java

```
public class Car extends Vehicle
    implements Relocatable {
    public void turnLeft() { ... }
    public void turnRight() { ... }
    public void forward() { ... }
    public void backward() { ... }
}
```



# Multiple inheritance (cont.)



Although Car and Human implements the same interfaces, their implementations are totally different!

## Human.java

```
public class Human implements Relocatable {
    public void turnLeft() { ... }
    public void turnRight() { ... }
    public void forward() { ... }
    public void backward() { ... }
}
```



# Polymorphism

- › Polymorphism: Using same method name to indicate **different implementations**
- › When multiple classes implement **the same interface**, each class implements the methods of the interface in **different ways**.
  - How is the correct method executed when the interface method is invoked?
- › **Late binding**: JVM makes a special effort to locate the correct method that belongs to the actual object “during the run time (not compile time).”



# Polymorphism (cont.)

## Bad Code

```
public void uTurn(Car c) {
    c.turnRight();
    c.turnRight();
}

public void uTurn(Human h) {
    h.turnRight();
    h.turnRight();
}
```

## Good Code1: Polymorphism

```
public void uTurn(Turnable m) {
    m.turnRight();
    m.turnRight();
}
```

## Better Code2:

```
public void uTurn(Object obj) {
    if(obj instanceof Turnable) {
        Turnable t = (Turnable) obj;
        t.turnRight();
        t.turnRight();
    }
}
```

## Code2:

```
public void uTurn(Object obj) {
    if(obj instanceof C) {
        Car c = (Car) obj;
        c.turnRight();
        c.turnRight();
    }
    else if(obj instanceof Human) {
        Human h = (Human) obj;
        h.turnRight();
        h.turnRight();
    }
}
```

What if there are Car, Human, Dog, Cat, Rat, etc. How many "uTurn" methods do we need to implement?





# Abstract Class vs. Interface

	Abstract Class	Interface
Variables	No restrictions	All variables must be “public static final” ( <b>CONSTANT</b> )
Constructors	It cannot be instantiated.  Have constructor Constructors are invoked by subclasses through constructor chaining.	It cannot be instantiated.  No constructor
Methods	No restrictions	All methods must be “public abstract”
Is it class?	Yes, it is class.	No, it is <b>not</b> class.
The Object class	All classes share a single root, the Object class.	There is <b>no</b> single root for interfaces.





# Abstract Class vs. Interface (cont.): Whether to use an interface or a class?

## CLASS

- › In general, a strong is-a relationship that clearly describes a parent-child relationship should be modeled using classes.
  - For example, a staff member is a person. So their relationship should be modeled using class inheritance.

### Design guidelines

Use classes for specialization and generalization  
Use interfaces to add properties to classes.

## INTERFACE

- › A weak is-a relationship or a behavior can be modeled using interfaces.
  - For example, all strings are comparable, so the String class implements the Comparable interface.
- › In the case of multiple inheritance, you have to design one as a superclass, and others as interface.





# Case Study: Original DataSet (Bad Design)

// Modified for BankAccount objects

```
public class DataSet {  
  
    private double sum;  
  
    private BankAccount maximum;  
  
    private int count;  
  
    public void add(BankAccount x) {  
  
        sum = sum + x.getBalance();  
  
        if (count == 0 || maximum.getBalance() < x.getBalance())  
            maximum = x;  
  
        count++;  
    }  
  
    public BankAccount getMaximum() {  
  
        return maximum;  
    }  
}
```



DataSet.maximum can be added by "BankAccount" and "Coin"; however, they have different method names.  
BankAccount.getBalance() vs. Coin.getValue()

// Modified for Coin objects

```
public class DataSet {  
  
    private double sum;  
  
    private Coin maximum;  
  
    private int count;  
  
    public void add(Coin x) {  
  
        sum = sum + x.getBalance();  
  
        if (count == 0 || maximum.getValue() < x.getValue())  
            maximum = x;  
  
        count++;  
    }  
  
    public Coin getMaximum() {  
  
        return maximum;  
    }  
}
```



# Case Study (cont.): DataSet with Interface

```
// Modified for BankAccount objects
```

```
public class DataSet {  
    private double sum;  
    private Measurable maximum;  
    private int count;  
    public void add(Measurable x) {  
        sum = sum + x.getMeasure();  
        if (count == 0 || maximum.getMeasure() < x.getMeasure())  
            maximum = x;  
        count++;  
    }  
    public Measurable getMaximum() {  
        return maximum;  
    }  
}
```



“BankAccount” and “Coin” implement “Measurable”, so they both have the same method “getMeasure()”.

**Now we can compare two different object types!**

- Which getMeasure() to call?
- “Polymorphism” called late binding: resolved at runtime
- Different from “overloading” called early binding: resolved at compilation

```
public class BankAccount implements Measurable{
```

```
    private double balance;
```

```
    public double getMeasure() {
```

```
        return balance;
```

```
    }
```

```
}
```

```
public class Coin implements Measurable{
```

```
    private double value;
```

```
    public double getMeasure() {
```

```
        return value;
```

```
    }
```

```
}
```

Measurable.java

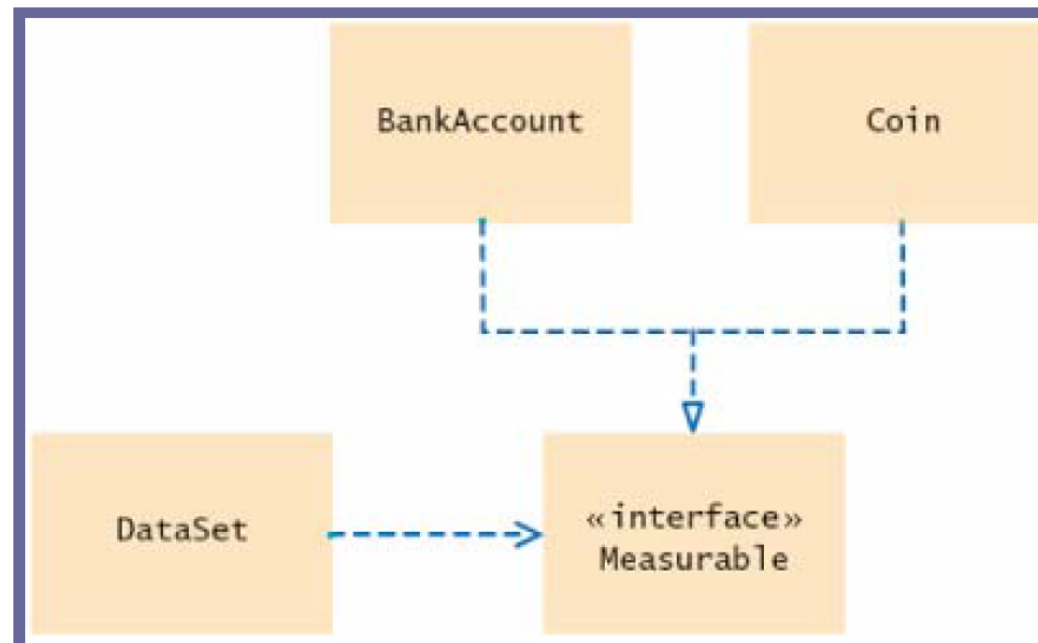
```
public interface Measurable {
```

```
    double getMeasure();
```



# Case Study (cont.): UML Diagram

- › Interfaces are tagged with a "stereotype" indicator **«interface»**
- › A dotted arrow with a triangular tip (---▶) denotes the “is-a” relationship between a class and an interface (implements)
- › A dotted line with an open v-shaped arrow tip (---->) denotes the “(just) uses” relationship or dependency (no implements)





# Case Study (cont.): DataSetTester.java

```
public class DataSetTester {  
  
    public static void main(String[] args) {  
  
        DataSet bankData = new DataSet();  
  
        bankData.add(new BankAccount(0));  
        bankData.add(new BankAccount(1000));  
        bankData.add(new BankAccount(2000));  
  
        Measurable max = bankData.getMaximum();  
        System.out.println(max.getMeasure());  
    }  
}
```

## Result

2000.00

0.25

1000.00

```
        DataSet coinData = new DataSet();  
        coinData.add(new Coin(0.25, "quarter"));  
        coinData.add(new Coin(0.1, "dime"));  
        coinData.add(new Coin(0.05, "nickel"));  
  
        max = coinData.getMaximum();  
        System.out.println(max.getMeasure());  
  
        DataSet mixedData = new DataSet();  
        mixedData.add(new Coin(0.25, "quarter"));  
        mixedData.add(new BankAccount(1000));  
  
        max = mixedData.getMaximum();  
        System.out.println(max.getMeasure());  
    }  
}
```



# Inner (nested) class

- › An inner class is any class that is defined **inside another class**.
- › Declare an inner class inside “a method,” so it is only available to that method.
- › You can also define an inner class inside an enclosing class, but outside of its methods. Then the inner class is available to all methods of the enclosing class.

## Measurable.java

```
public interface Measurable {  
    double getMeasure();  
}
```

## DataSetTester1.java (inside method)

```
// assume interface "Measurable" and class "DataSet"  
  
public class DataSetTester1 {  
    public static void main(String[] args) {  
        class MeasurableRectangle implements Measurable{...}  
  
        Measurable m = new MeasurableRectangle();  
  
        DataSet data = new DataSet();  
  
        data.add(m);  
    }  
}
```

## DataSetTester2.java (outside method)

```
// assume interface "Measurable" and class "DataSet"  
  
public class DataSetTester2 {  
    class MeasurableRectangle implements Measurable{...}  
  
    public static void main(String[] args) {  
        Measurable m = new MeasurableRectangle();  
  
        DataSet data = new DataSet();  
  
        data.add(m);  
    }  
}
```

- › When you compile the source files for a program that uses inner classes, you will find that the inner classes are stored in files with curious names

DataSetTester1\$1MeasurableRectangle.class



# Accessing outer class' variables from inner class

- › Methods of the inner class can **directly access** members of the outer class.
- › The inner class concept is employed when we want to **hide** it from other classes, e.g., hide "MeasurableRectangle".

## DataSetTester3.java (inside method)

```
// assume interface "Measurable" and class "DataSet"

public class DataSetTester3 {
    static int outData;

    public static void main(String[] args) {

        class MeasurableRectangle implements Measurable{
            outData = 10;
        }

        Measurable m = new MeasurableRectangle();

        DataSet data = new DataSet();

        data.add(m);

        System.out.println(outData);
    }
}
```



# Anonymous Inner Class

- › It is an inner class **without defined name**.
- › It is often used in “event listener” in GUI.

Measurable.java

```
public interface Measurable {  
    double getMeasure();  
}
```

Anonymous1

```
Measurable x = new Measurable(){  
    public double getMeasure(){  
        ...  
    }  
};
```

Create Measurable's instance

Anonymous2

```
dataSet.add(new Measurable(){  
    public double getMeasure(){  
        ...  
    }  
});
```

Use Measurable's instance as input





# Commonly Used Java Interfaces

- › Iterator
  - To run through a collection of objects without knowing how the objects are stored, e.g., in array, list, bag, or set.
- › Comparable
  - To make a total order on objects, e.g., 3, 56, 67, 879, 3422, 34234
- › Cloneable
  - To make a copy of an existing object via the clone() method on the class Object.
- › Serializable
  - Pack a web of objects such that it can be send over a network or stored to disk. An naturally later be restored as a web of objects.



# The Iterator Interface

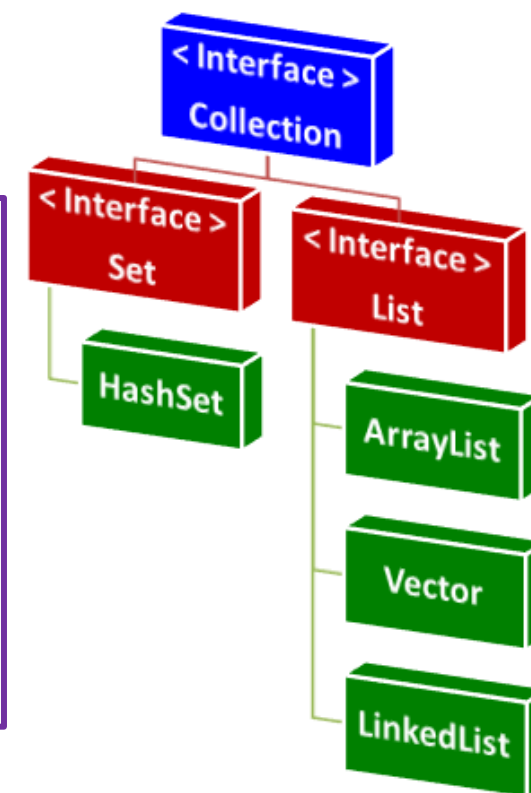
- › The Iterator interface in the package java.util is a basic iterator that works on collections.

## Iterator.java

```
Package java.util;  
  
public interface Iterator {  
    bool hasNext();  
    Object next();  
    // optional, throws exception  
    void remove();  
}
```

## Main

```
myShapes = getSomeCollectionOfShapes();  
Iterator iter = myShapes.iterator();  
while(iter.hasNext()) {  
    Shape s = (Shape) iter.next(); //downcast  
    s.draw();  
}
```





# The Iterator Interface (cont.)

## ArrayListLoopingExample.java

```
public class ArrayListLoopingExample {
    public static void main(String[] args) {
        List<String> list = new ArrayList<String>();
        list.add("Text 1");
        list.add("Text 2");
        list.add("Text 3");

        System.out.println("#1 normal for loop");
        for (int i = 0; i < list.size(); i++) {
            System.out.println(list.get(i));
        }

        System.out.println("#2 for each loop");
        for (String temp : list) {
            System.out.println(temp);
        }
    }
}
```

```
System.out.println("#3 while loop");
int j = 0;
while (list.size() > j) {
    System.out.println(list.get(j));
    j++;
}

System.out.println("#4 iterator");
Iterator<String> iterator = list.iterator();
while (iterator.hasNext()) {
    System.out.println(iterator.next());
}
}
```

#1 normal for loop

Text 1

Text 2

Text 3

#2 for each loop

Text 1

Text 2

Text 3

#3 while loop

Text 1

Text 2

Text 3

#4 iterator

Text 1

Text 2

Text 3



# The Comparable Interface

- › In the package `java.lang`.
- › Returns a negative integer, zero, or a positive integer as this object is less than, equal to, or greater than the specified object.

```
package java.lang;  
public interface Comparable {  
    int compareTo(Object o);  
}
```



# The Comparable Interface (cont.)

```
// IPAddress example revisited
public class IPAddress implements Comparable{
    private int[] n; // here IP stored, e.g., 125.255.231.123

    /** The Comparable interface */
    public int compareTo(Object o){
        IPAddress other = (IPAddress) o; // downcast
        int result = 0;
        for(int i = 0; i < n.length; i++){
            if (this.getNum(i) < other.getNum(i)){
                result = -1;
                break;
            }
            if (this.getNum(i) > other.getNum(i)){
                result = 1;
                break;
            }
        }
        return result;
    }
}
```



# The Cloneable Interface

- › A class X that implements the Cloneable interface tells clients that X objects can be cloned.
- › The interface is empty, i.e., has no methods.
- › Returns **an identical copy** of an object.
  - A *shallow copy*, by default.
  - A *deep copy* is often preferable.
- › **clone** method should throw an exception:
  - › CloneNotSupportedException

[http://www.jusfortechies.com/java/core-java/deepcopy\\_and\\_shallowcopy.php](http://www.jusfortechies.com/java/core-java/deepcopy_and_shallowcopy.php)



To have `clone()`, one must implement `Cloneable`, otherwise `CloneNotSupportedException` will be thrown.

# The Cloneable Interface (cont.): Example 1

```
public class Person {  
  
    private String name;  
  
    public Person(String name) {  
        this.name = name;  
    }  
  
    public String getName() {  
        return name;  
    }  
}
```

```
public static void main(String[] args) {  
    Person p = new Person("Sam");  
    try {  
        Person pClone = (Person) p.clone();  
        System.out.println(pClone.getName());  
    } catch (CloneNotSupportedException e) {  
        e.printStackTrace();  
    }  
}
```

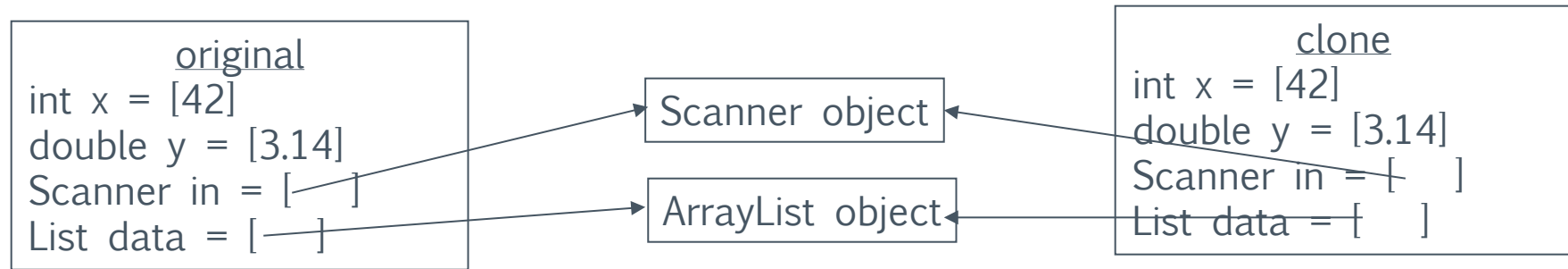
## Result

```
java.lang.CloneNotSupportedException: Person  
at java.lang.Object.clone(Native Method)  
at Person.main(Person.java:19)
```

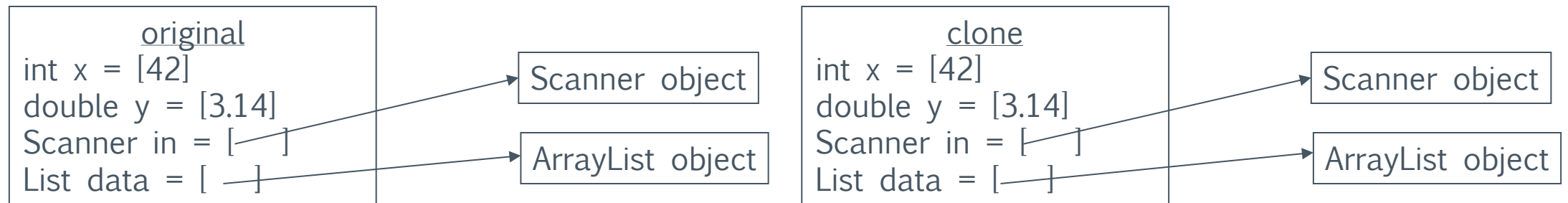


# The Cloneable Interface (cont.): Shallow vs. deep copy

- › **shallow copy:** Duplicates an object without duplicating any other objects to which it refers.



- › **deep copy:** Duplicates an object's entire *reference graph*: copies itself and deep copies any other objects to which it refers.



- Object's clone method makes a shallow copy by default.





# The Cloneable Interface (cont.): Shallow vs. deep copy

```
public class Person implements Cloneable {  
    private String name;  
    public Person(String name) {  
        this.name = name;  
    }  
    public String getName() {  
        return name;  
    }  
    public Person clone() {  
        try {  
            return (Person) super.clone();  
        } catch (CloneNotSupportedException e) {  
            e.printStackTrace();  
            throw new RuntimeException();  
        }  
    }  
}
```

Shallow copy

```
public class Person implements Cloneable {  
    private String name;  
    public Person(String name) {  
        this.name = name;  
    }  
    public String getName() {  
        return name;  
    }  
    public Person clone() {  
        return new Person(this.name);  
    }  
}
```

Deep copy



# The Cloneable Interface (cont.): Example 3

```
public class Person implements Cloneable {  
    private String name;  
    private Date dob;  
    public Person(String name) { this.name = name; }  
    public String getName() { return name; }  
    public Person clone() {  
        Person p;  
        try {  
            p = (Person) super.clone();  
            p.dob = this.dob;  
        } catch (CloneNotSupportedException e) {  
            e.printStackTrace();  
            throw new RuntimeException();  
        }  
    }  
}
```



# The `Serializable` Interface

- › A class X that implements the **`Serializable`** interface tells clients that X objects can be stored on file or other persistent media (an object can be represented as a sequence of bytes).
- › The interface is empty, i.e., **has no methods**.

```
public class Car implements Serializable {  
    // rest of class unaltered  
    snip  
}
```

```
// write to and read from disk  
import java.io.*;  
public class SerializeDemo{
```

```
    Car myToyota, anotherToyota;  
    myToyota = new Car("Toyota", "Carina", 42312);  
    ObjectOutputStream out = getOutput();  
    out.writeObject(myToyota);
```

```
    ObjectInputStream in = getInput();  
    anotherToyota = (Car)in.readObject();
```

```
}
```

Should be  
put inside  
main()



# Tag or Marker Interface

- › An interface with no methods
- › Use as common type
- › Serializable interface



# Functional Interface

- › An interface that has only ONE method.
- › Example

```
interface Execute {  
    public void doIt();  
}
```

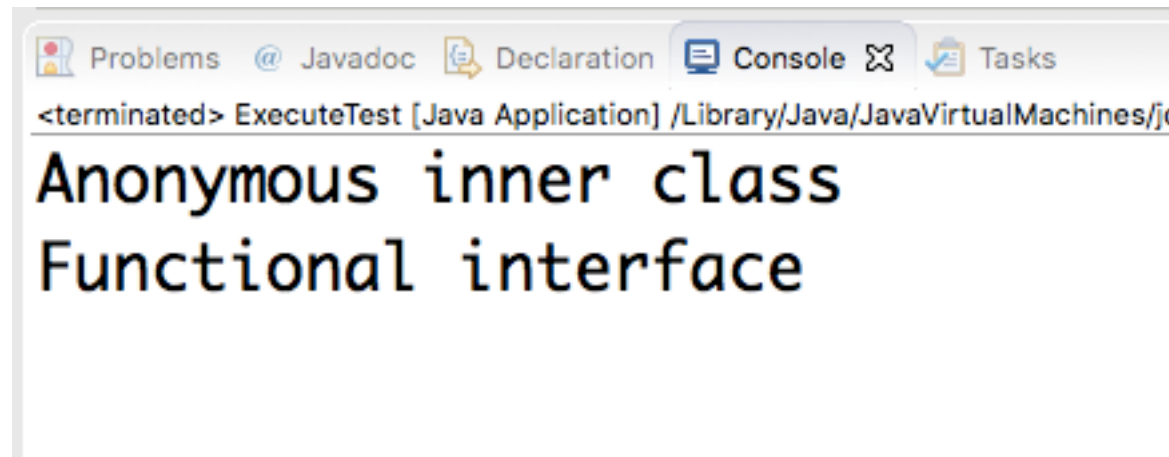


# Using functional Interface (1)

```
public class ExecuteTest {  
    public static void main(String[] args) {  
        Execute e = new Execute() {  
            public void doIt() {  
                System.out.println("Anonymous inner class");  
            }  
        };  
    }  
};
```

```
Execute f = () -> System.out.println("Functional interface");
```

```
    e.doIt();  
    f.doIt();  
}  
}
```

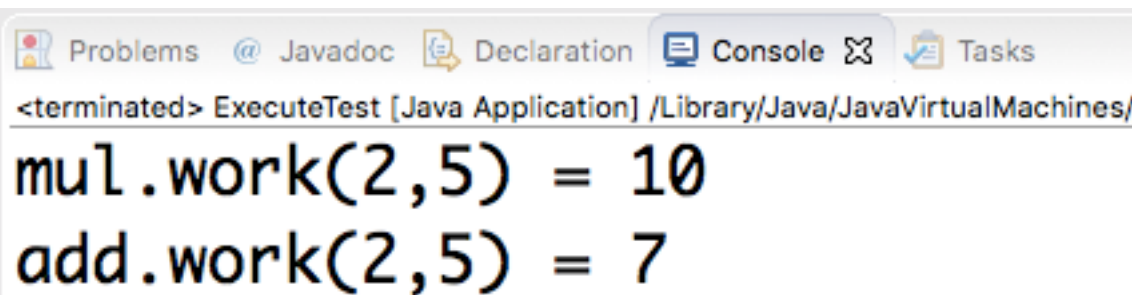




# Using functional Interface (2) method with arguments

```
interface Perform {  
    int work(int a, int b);  
}
```

```
public class ExecuteTest {  
    public static void main(String[] args) {  
        Perform mul = (x, y) -> x * y;  
        Perform add = (x, y) -> x + y;  
        System.out.println("mul.work(2, 5) = " + mul.work(2, 5));  
        System.out.println("add.work(2, 5) = " + add.work(2, 5));  
    }  
}
```



```
<terminated> ExecuteTest [Java Application] /Library/Java/JavaVirtualMachines/  
mul.work(2,5) = 10  
add.work(2,5) = 7
```



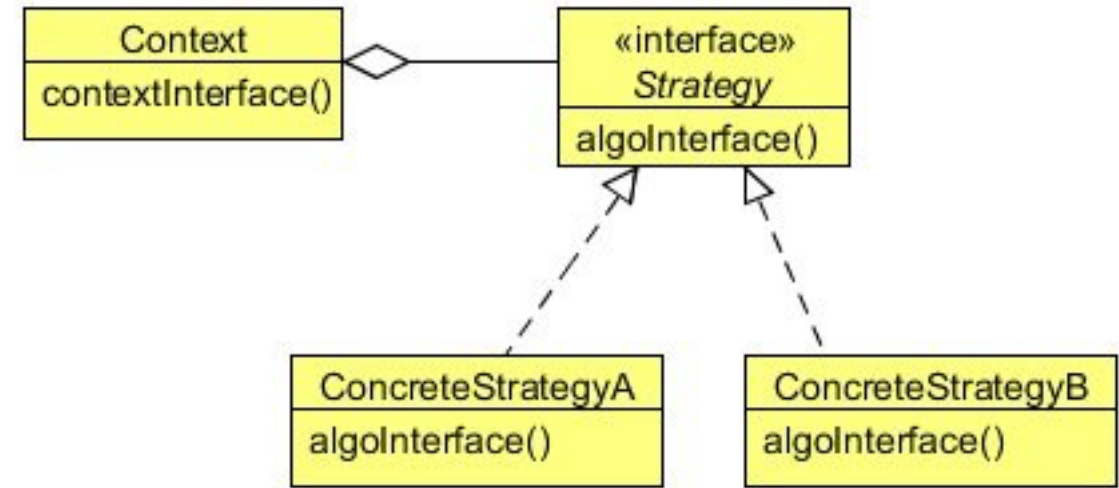
# Strategy Pattern

- › The Strategy Design Pattern defines a family of algorithms, encapsulating each one, and making them interchangeable. Strategy lets the algorithm vary independently from the clients that use it.





# Strategy Pattern



## › Strategy

- Declares an interface common to all supported algorithms. Context uses this interface to call the algorithm defined by a **ConcreteStrategy**.

## › ConcreteStrategy

- Implements the algorithm using the **Strategy** interface.

## › Context

- Is configured with a **ConcreteStrategy** object.
- Maintains a reference to a **Strategy** object.
- May define an interface that lets Strategy access its data.



# Strategy Pattern Example

```
// strategy
```

```
public interface TextFormatter {  
    public void format(String text);  
}
```

```
// concrete strategy
```

```
public class CapTextFormatter implements TextFormatter {
```

```
@Override
```

```
public void format(String text) {  
    System.out.println("[CapTextFormatter]: " +  
        text.toUpperCase());  
}
```

```
}
```

```
// concrete strategy
```

```
public class LowerTextFormatter implements TextFormatter {
```

```
@Override
```

```
public void format(String text) {
```

```
    System.out.println("[LowerTextFormatter]: " +  
        text.toLowerCase());  
}
```

```
}
```



```
// context
```

```
public class TextEditor {  
    private final TextFormatter textFormatter;  
  
    public TextEditor(TextFormatter textFormatter) {  
        this.textFormatter = textFormatter;  
    }  
  
    public void publishText(String text) {  
        textFormatter.format(text);  
    }  
}
```

```
3 public class TestStrategyPattern {  
4  
5     public static void main(String[] args) {  
6         TextFormatter formatter = new CapTextFormatter();  
7         TextEditor editor = new TextEditor(formatter);  
8         editor.publishText("Testing text in caps formatter");  
9  
10        formatter = new LowerTextFormatter();  
11        editor = new TextEditor(formatter);  
12        editor.publishText("Testing text in lower formatter");  
13  
14    }  
15 }
```

Problems @ Javadoc Declaration Console Tasks

<terminated> TestStrategyPattern [Java Application] /Library/Java/JavaVirtualMachines/jdk1.8.0\_65.jdk/Contents/Home/bin/java (Oct 9, 2018, 11:20:09 PM)

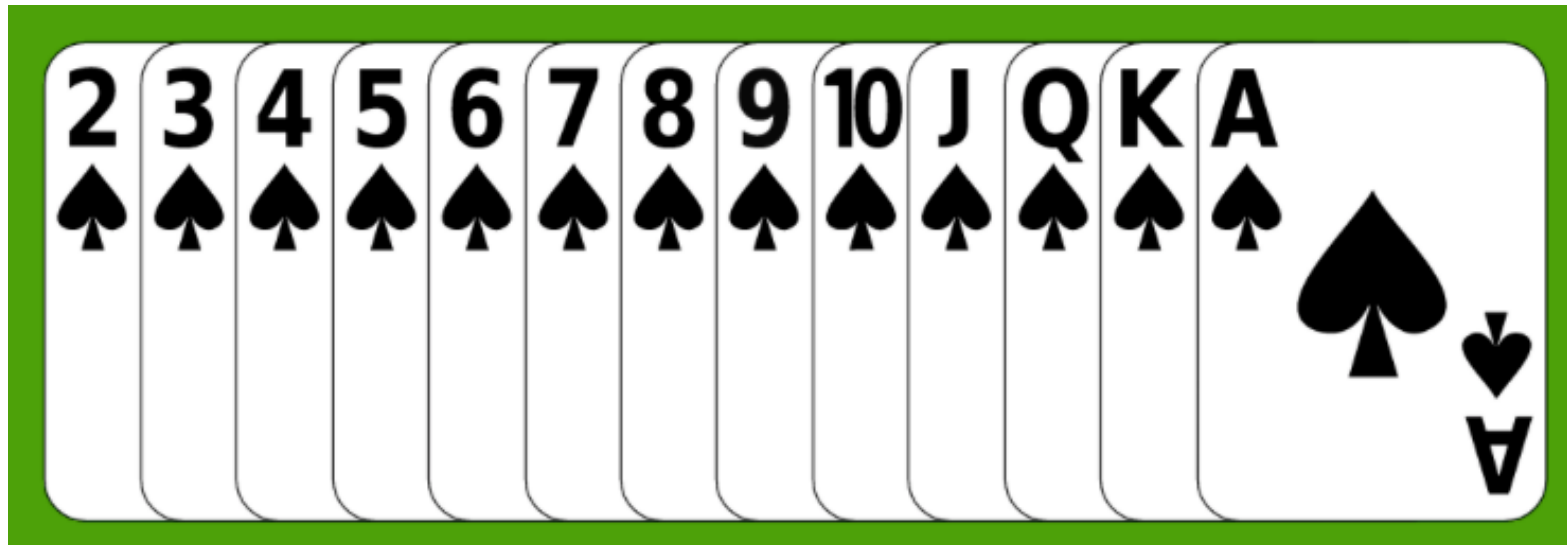
[CapTextFormatter]: TESTING TEXT IN CAPS FORMATTER

[LowerTextFormatter]: testing text in lower formatter



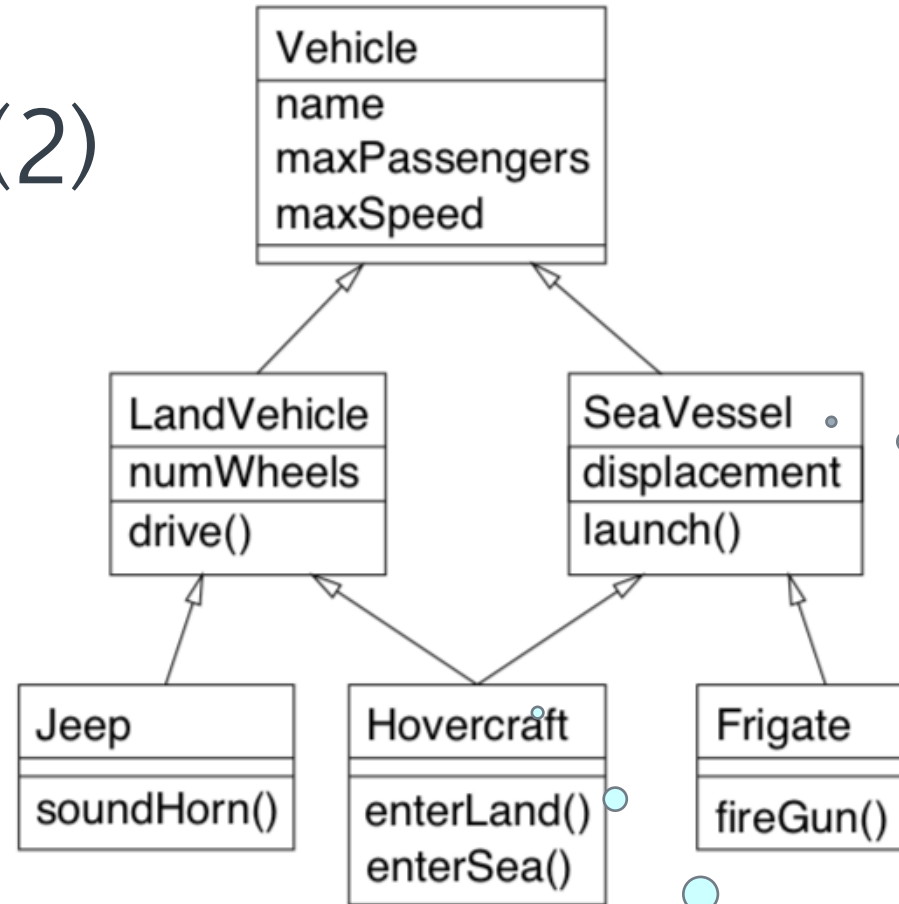
## Exercise (1)

- › Write a class that implements the Iterator interface found in the java.util package. The ordered data for this exercise is the 13 cards in a suit from a deck of cards. The first call to next returns 2, the subsequent call returns the next highest card, 3, and so on, up to Ace. Write a small main method to test your class.





# Exercise (2)



Can be travel  
in water

Redesign this, so they can  
be implemented in Java.

Can be travel  
in both land  
and water



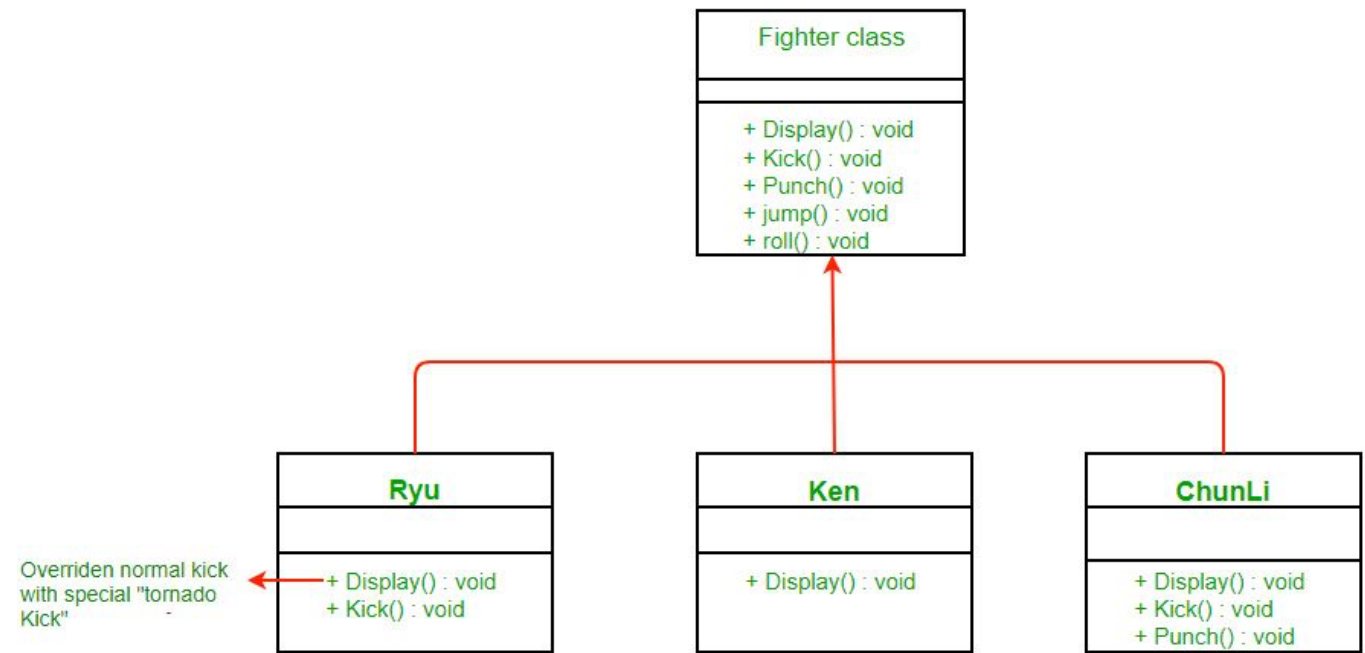


## Exercise (3)

Suppose we are building a game “Street Fighter”. For simplicity assume that a character may have four moves that is **kick**, **punch**, **roll** and **jump**. Every character has kick and punch moves, but roll and jump are optional. How would you model your classes?

Suppose initially you use inheritance and abstract out the common features in a **Fighter** class and let other characters subclass **Fighter** class.

**Fighter** class will we have default implementation of normal actions. Any character with specialized move can override that action in its subclass. Class diagram would be as follows:



What are the problems with this design?

How can we solve them?