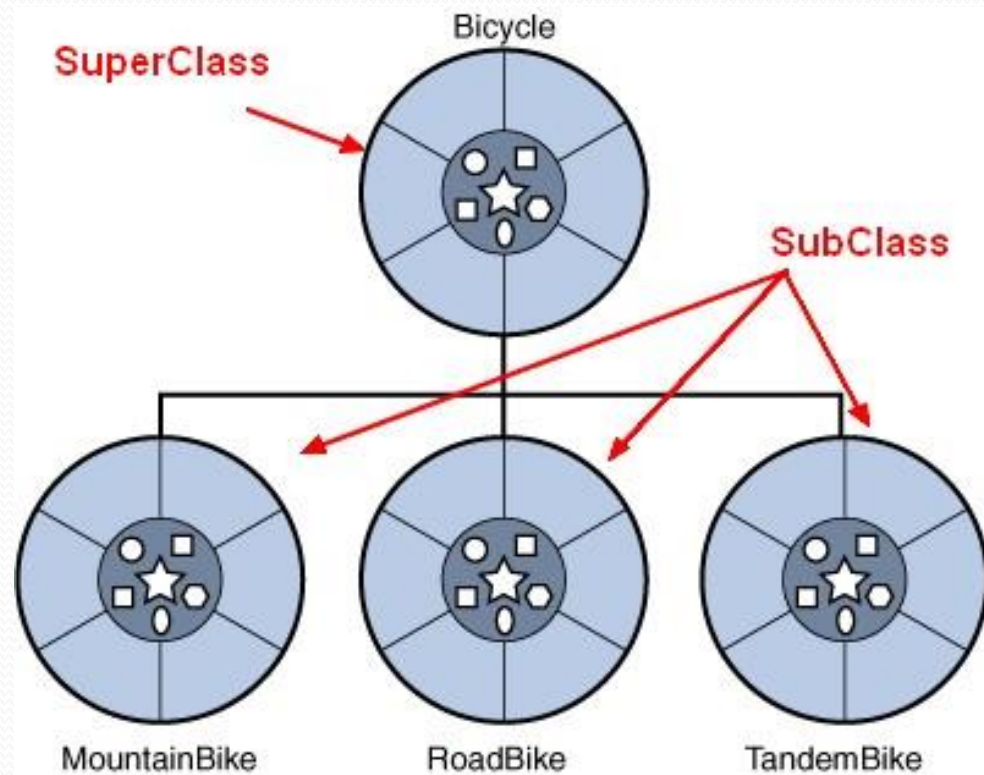


Object Oriented Programming

Module 4. Inheritance and
Polymorphism

Inheritance

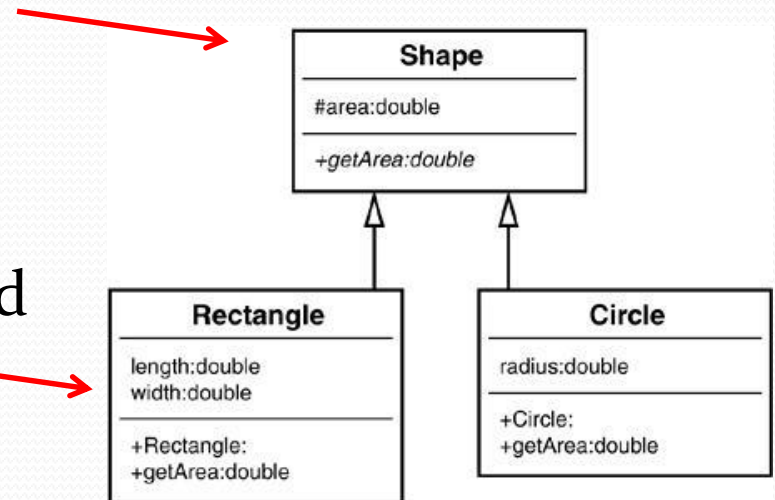


Inheritance

- Is a fundamental technique for **organizing** and **creating** classes.
- Is the process of **deriving** a new class from an existing one.
- Is a powerful software development technique and a defining characteristic of object-oriented programming.
- Is one way to support the idea of **software reuse**.

Inheritance

- The original class that is used to derive a new one is called the parent class, superclass, or base class.
- The derived class is called a child class, or subclass.
- Java uses the reserved word **extends** to indicate that a new class is being derived from an existing class.



Inheritance

- Creates an **IS-A** relationship between the parent and child classes.
- Via inheritance, the new class automatically **contains** the variables and methods in the original class.
- New variables and methods can be added to the new class. Or the inherited ones can be modified.
- The child class should be a more specific version of the parent.

Inheritance

Example:

- A horse **is a** mammal.
- Not all mammals are horses, but all horses are mammals.

For any class X that is derived from class Y, you should be able to say that “**X is a Y**”.



Inheritance

Example:

```
public class Calculation {  
    int z;  
    public void addition(int x, int y) {  
        z = x + y;  
        System.out.println("The sum of the given numbers:"+z);  
    }  
    public void subtraction(int x, int y) {  
        z = x - y;  
        System.out.println("The difference between the given  
        numbers:"+z);  
    }  
}
```

Inheritance

```
public class MyCalculation extends Calculation {  
    public void multiplication(int x, int y) {  
        z = x * y;  
        System.out.println("The product of the given numbers:"+z);  
    }  
}
```

```
public static void main(String args[]) {  
    int a = 20,  
    b = 10;  
    MyCalculation demo = new MyCalculation();  
    demo.addition(a, b);  
    demo.subtraction(a, b);  
    demo.multiplication(a, b);  
}
```

```
}
```



What is the result?

Inheritance: The Super Reference

- The reserved word **super** can be used in a class to refer to its parent class.
- Like the `this` reference, what the word `super` refers to depends on the class in which it is used.
- One use of the `super` reference is to invoke a parent's constructor.

Inheritance: The Super Reference

- Accessing the constructor of the superclass:

```
//superclass
```

```
public Book(int numPages) {  
    pages = numPages;  
}
```

```
//subclass
```

```
public Dictionary(int numPages, int numDefinitions) {  
    super(numPages);  
    definitions = numDefinitions;  
}
```

Inheritance: The Super Reference

- Accessing super class methods:

```
public class Superclass {  
    public void printMethod() {  
        System.out.println("Printed in Superclass.");  
    }  
}
```

```
public class Subclass extends Superclass {  
    // overrides printMethod in Superclass  
    public void printMethod() {  
        super.printMethod();  
        System.out.println("Printed in Subclass");  
    }  
}
```

Polymorphism



In Shopping malls behave like Customer

In Bus behave like Passenger

In School behave like Student

At Home behave like Son

Polymorphism

Polymorphism means **many forms** and concerns the ability of an object to dynamically take on a different form depending on the runtime context.

- Polymorphism involves the behaviour of objects.
- Polymorphism is another fundamental principle of object-oriented software.

Upcast Vs. Downcast

What is type casting?

- Assigning a value of one type to a variable of another type.
- **Upcasting** is casting a subtype to a supertype.
- **Downcasting** is casting to a subtype.



Type casting and upcasting/downcasting -> **NOT** the same!

Polymorphism

Example:

- The type of a reference variable matches the class of the object to which it refers exactly. For example:

ChessPiece piece;

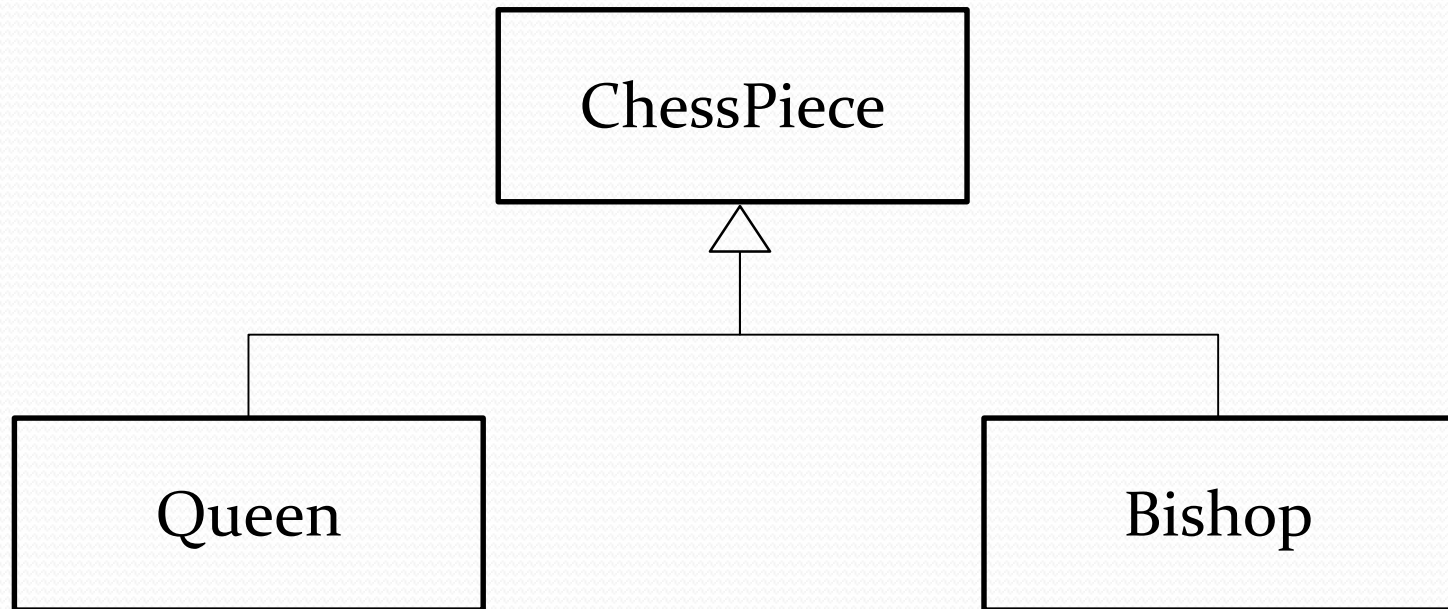
- This variable may be used to point to an object of the class ChessPiece. But it is not mandatory.
- It can refer to a compatible type.

ChessPiece piece = new Bishop(); //upcasting

ChessPiece piece = new Queen(); //upcasting

A polymorphic reference is a reference variable that can refer to **different** types of objects at **different** points in time.

Polymorphism



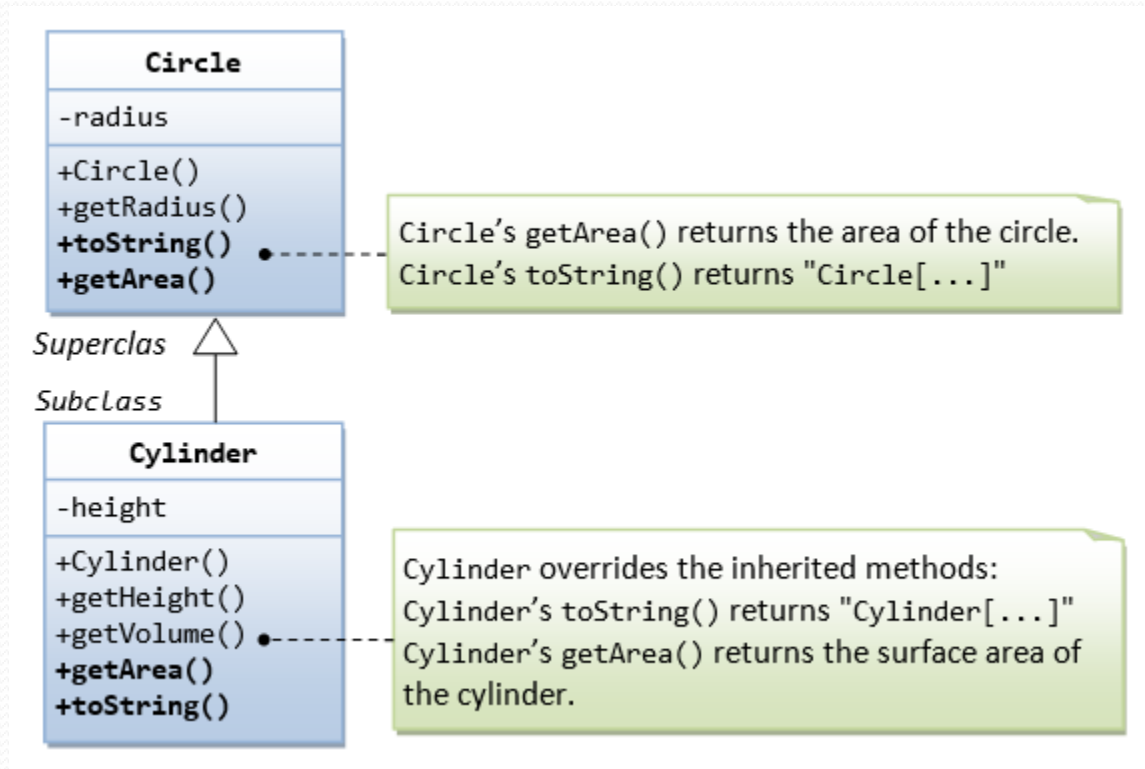
We can create a polymorphic reference in Java in two ways: using **inheritance** and using **interfaces**.

Polymorphism Via Inheritance

- A reference variable can refer to **any object** created from any class related to it by **inheritance**.
- The type of the object, not the type of the reference, is used to determine which version of a method to invoke.
- For example:
 1. If the class Mammal is the parent of the class Horse, then a Mammal reference can be used to refer to any object of class Horse.
 2. The reverse operation, assigning the Mammal object to a Horse reference, can also be done but it requires an explicit cast. **(less useful and more likely to cause problems).**

Polymorphism Via Inheritance

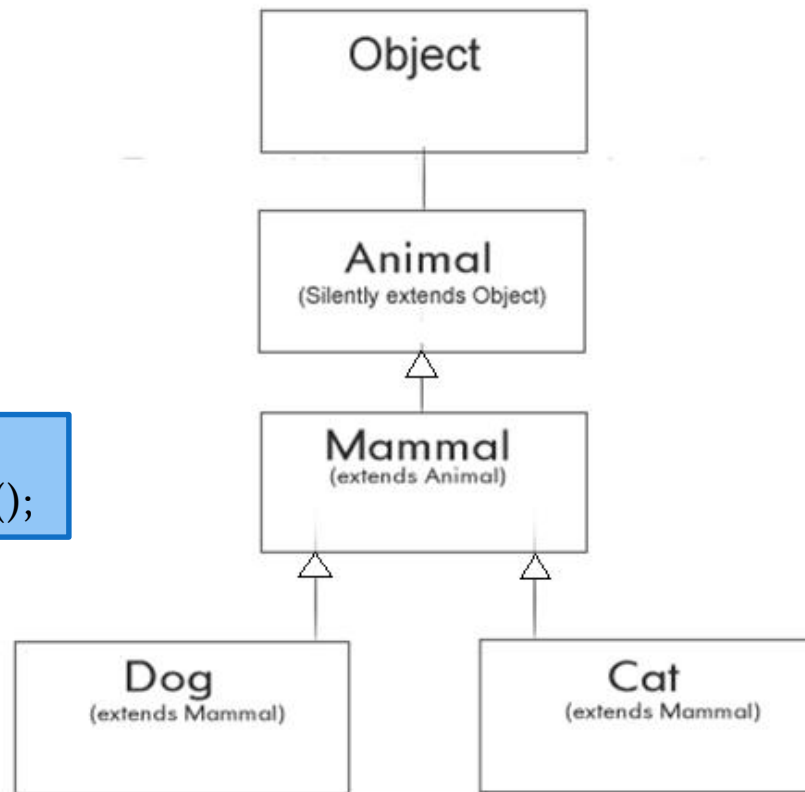
- Circle c1 = new Cylinder(1.1, 2.2); // upcast is safe
- Cylinder cy1 = (Cylinder) c1; //downcast needs the casting operator



Polymorphism Via Inheritance

Case 1:
Mammal pet;
Dog myDog= new Dog();
pet = myDog;

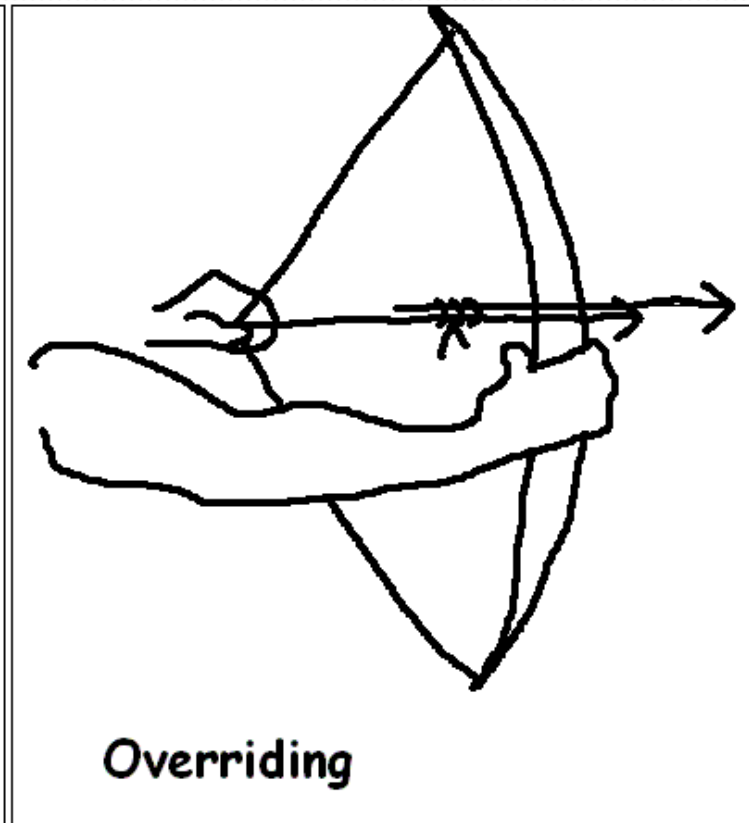
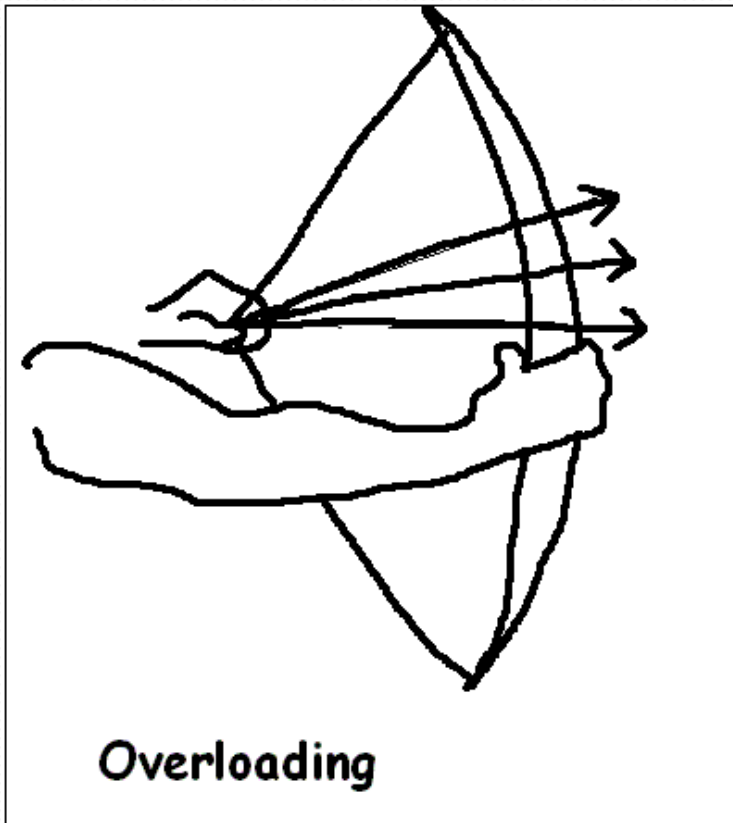
Case 2:
Animal creature = new Dog();



upcasting

downcasting

Overloading and Overriding



Overloading

- An overloaded method occurs when two or more methods with the **same name** in the **same class** have **different parameters**.
- For example, the withdraw method in the following example class is overloaded.
- In this example the withdraw method is overloaded since the parameter type of the single parameter is different in each method.

Account
-id : int -balance : float
+withdraw(in amount : float) : void +withdraw(in amount : int) : void +withdraw(in amount : String) : void

Overloading

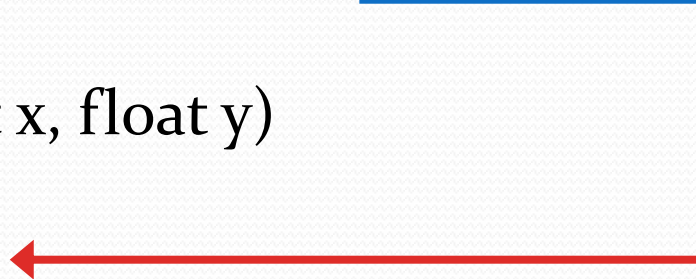
- The compiler determines which method is being invoked by analyzing the parameters:

```
float tryMe(int x)
{
    return x + .375;
}
```

```
float tryMe(int x, float y)
{
    return x*y;
}
```

Invocation

result = tryMe(25, 4.32)



Overloading

- The **println** method is an example of a method that is overloaded several times:
 - `println(String s)`
 - `println(int i)`
 - `println(double d)`
 - `println(char c)`
 - `println(boolean b)`

Invoking the different methods with the same name

```
System.out.println("Number of students: ");  
System.out.println(count);
```

Overriding

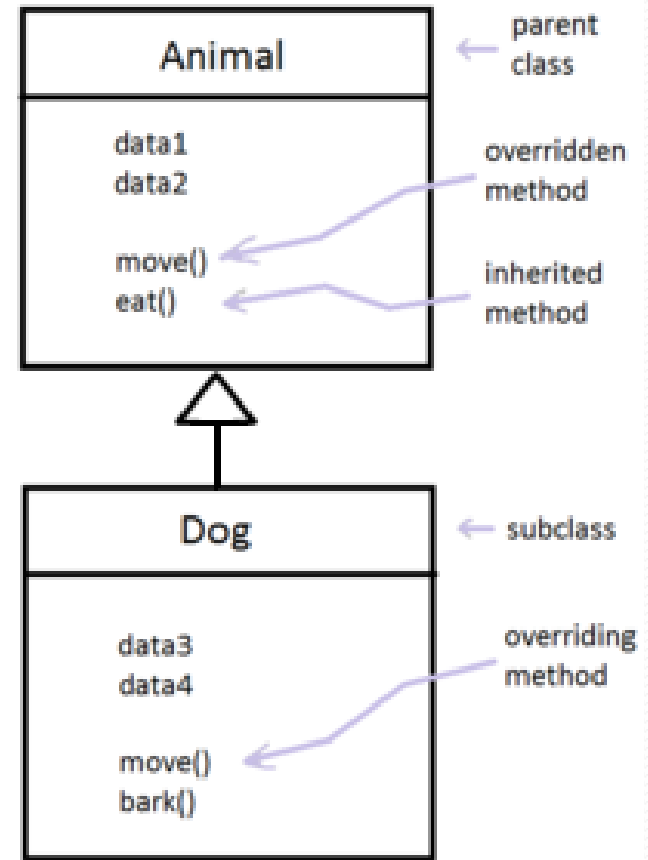
When a child class defines a method with the **same name** and **signature** as a method in the parent class, we say that the child's version overrides the parent's version in favor of its own.

The need for overriding occurs often in inheritance situations.

Why???

Overriding

- Method overriding is a key element in object-oriented design.
- It allows two objects that are related by **inheritance** to use the same naming conventions for methods that accomplish the same general task in **different ways**.
- Overriding becomes even more important when it comes to **polymorphism**.



Overriding

- A method can be defined with the **final** modifier.
- A child class CANNOT override a final method.
- This technique is used to ensure that a derived class uses a particular definition of a method.

```
class UNIX {  
    protected final void whoAmI () {  
        System.out.println("I am UNIX");  
    }  
}  
  
class Linux extends UNIX {  
    public void whoAmI () {  
        System.out.println("I am Linux");  
    }  
}
```

Cannot override the final method from UNIX

1 quick fix available:

➔ [Remove 'final' modifier of 'UNIX.whoAmI'\(..\)](#)

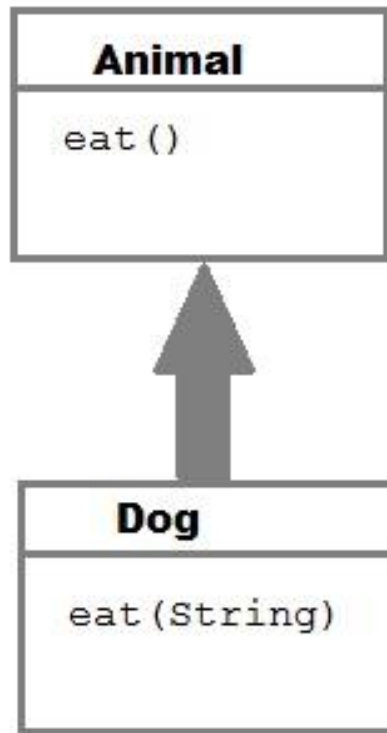
Press 'F2' for focus

Overriding

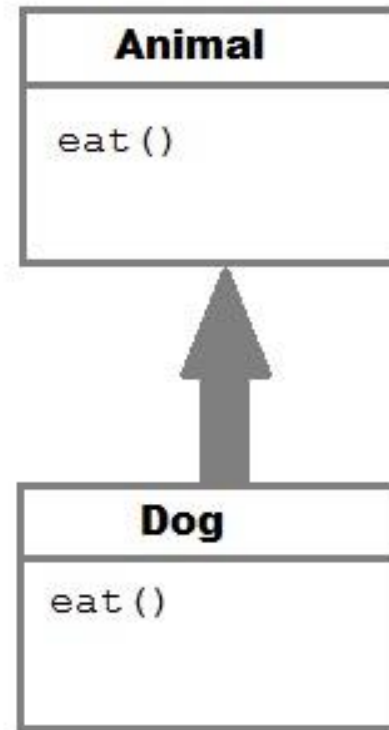
```
class Animal {  
    public void move() {  
        System.out.println("Animals can move"); }  
}  
class Dog extends Animal {  
    @Override  
    public void move() { System.out.println("Dogs can walk and run"); }  
}  
public class TestDog {  
    public static void main(String args[]) {  
        Animal a = new Animal(); // Animal reference and object  
        Animal b = new Dog(); // Animal reference but Dog object  
        a.move(); // runs the method in Animal class  
        b.move(); // runs the method in Dog class  
    }  
}
```

Overloading vs. Overriding

overloading



overriding



Overloading vs. Overriding

Overriding

```
class Dog{
    public void bark(){
        System.out.println("woof ");
    }
}
class Hound extends Dog{
    public void sniff(){
        System.out.println("sniff ");
    }

    public void bark(){
        System.out.println("bowl");
    }
}
```

Same Method Name,
Same parameter

Overloading

```
class Dog{
    public void bark(){
        System.out.println("woof ");
    }

    //overloading method
    public void bark(int num){
        for(int i=0; i<num; i++)
            System.out.println("woof ");
    }
}
```

Same Method Name,
Different Parameter

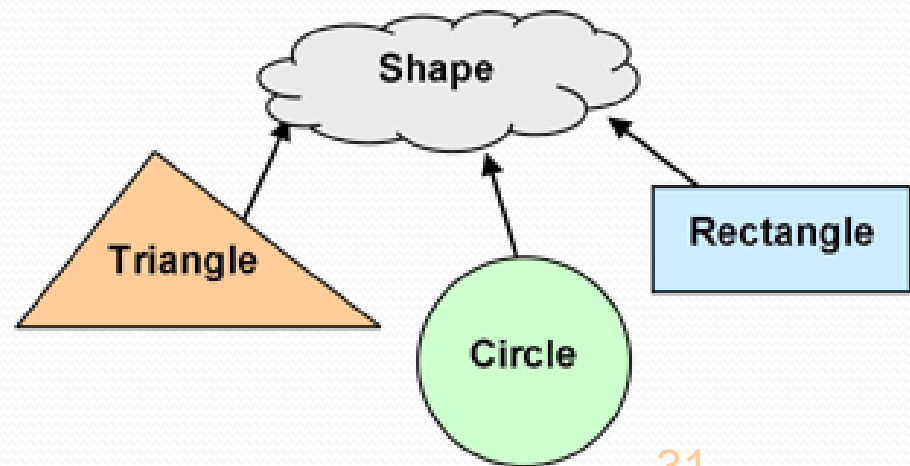


Abstract
Class

Interface

Abstraction

- Separation of interface and implementation is an **abstraction mechanism** in object-oriented programming languages.
- In Java abstraction is achieved by the use of:
 - Abstract classes
 - Interfaces



Abstraction: Abstract Classes

An abstract class:

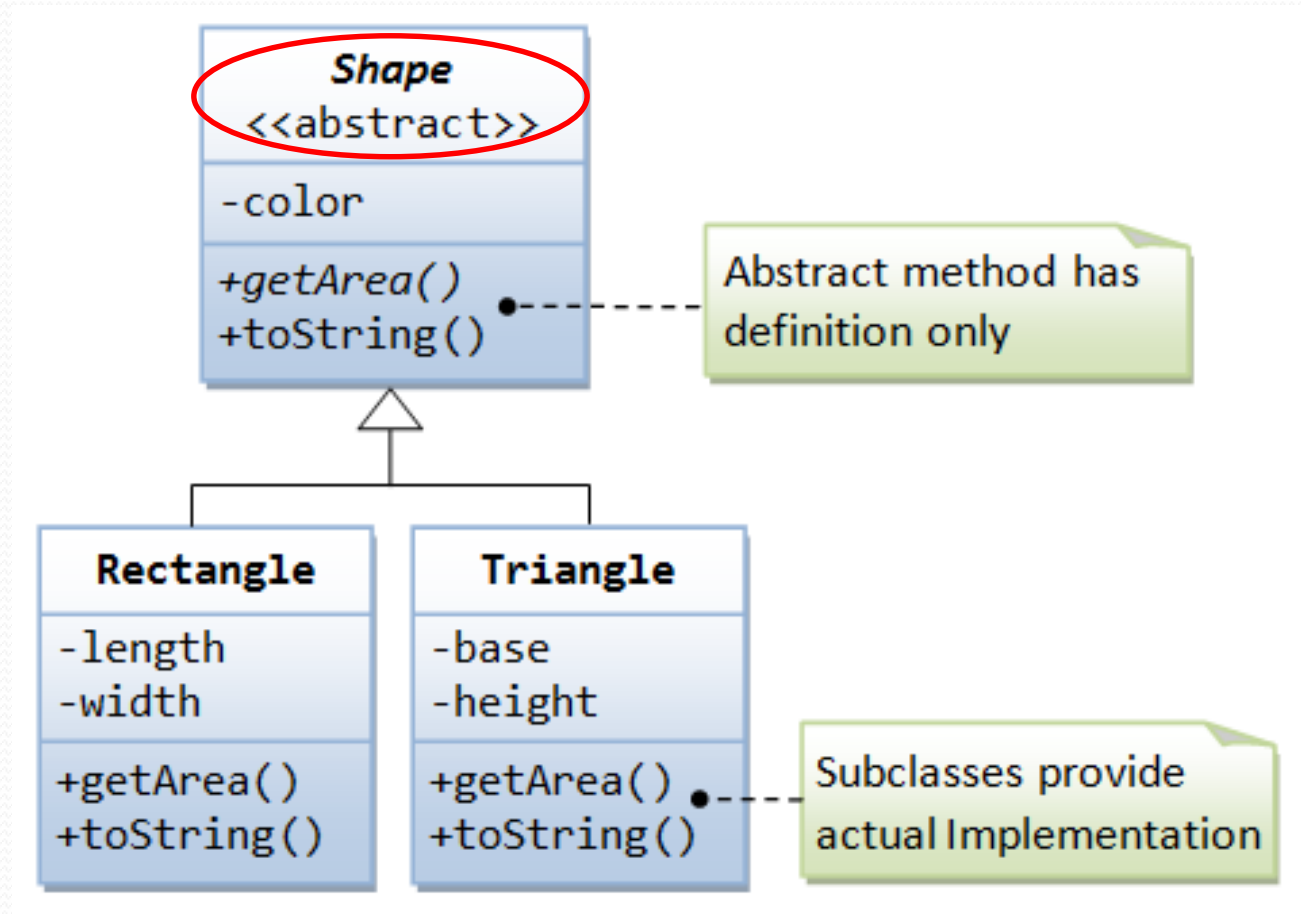
- Represents a **generic concept** in a class hierarchy.
- **Cannot** be instantiated.
- Usually contains one or more **abstract methods**, which have no definition.
- Represent a concept on which other classes can build their definitions.
- Is modelled with its name in **italics**, as opposed to concrete classes (classes from which objects are instantiated), whose names are in normal text.

Abstraction: Abstract Classes

- A class is declared as abstract by including the **abstract modifier** in the class header.
- Any class that contains one or more **abstract methods** must be declared as **abstract**.
- In abstract classes, the abstract modifier must be applied to each abstract method.
- A class derived from an abstract parent must **override** all of its parent's abstract methods.

An abstract class may contain a partial description that is **inherited** by all of its descendants in the class hierarchy. Its children, which are more specific, fill in the gaps.

Abstraction: Abstract Classes



Abstract Class

Abstraction: Abstract Classes

```
public abstract class Employee {  
    private String name;  
    private String address;  
    private int number;  
    public abstract double computePay();  
    //...  
}
```

Abstract Method – doesn't have implementation.

```
public class Executive extends Employee {  
    private double salary;  
    public double computePay() {  
        return salary/52;  
    }  
    //...  
}
```

Abstraction: Interfaces

- An interface is similar to an abstract class that **does not** have any concrete methods at all.
- An interface is simply a collection of **constants** and **method signatures** (abstract methods).
- An interface is used to establish a set of methods that a class will implement.
- A class can implement multiple interfaces. For example:

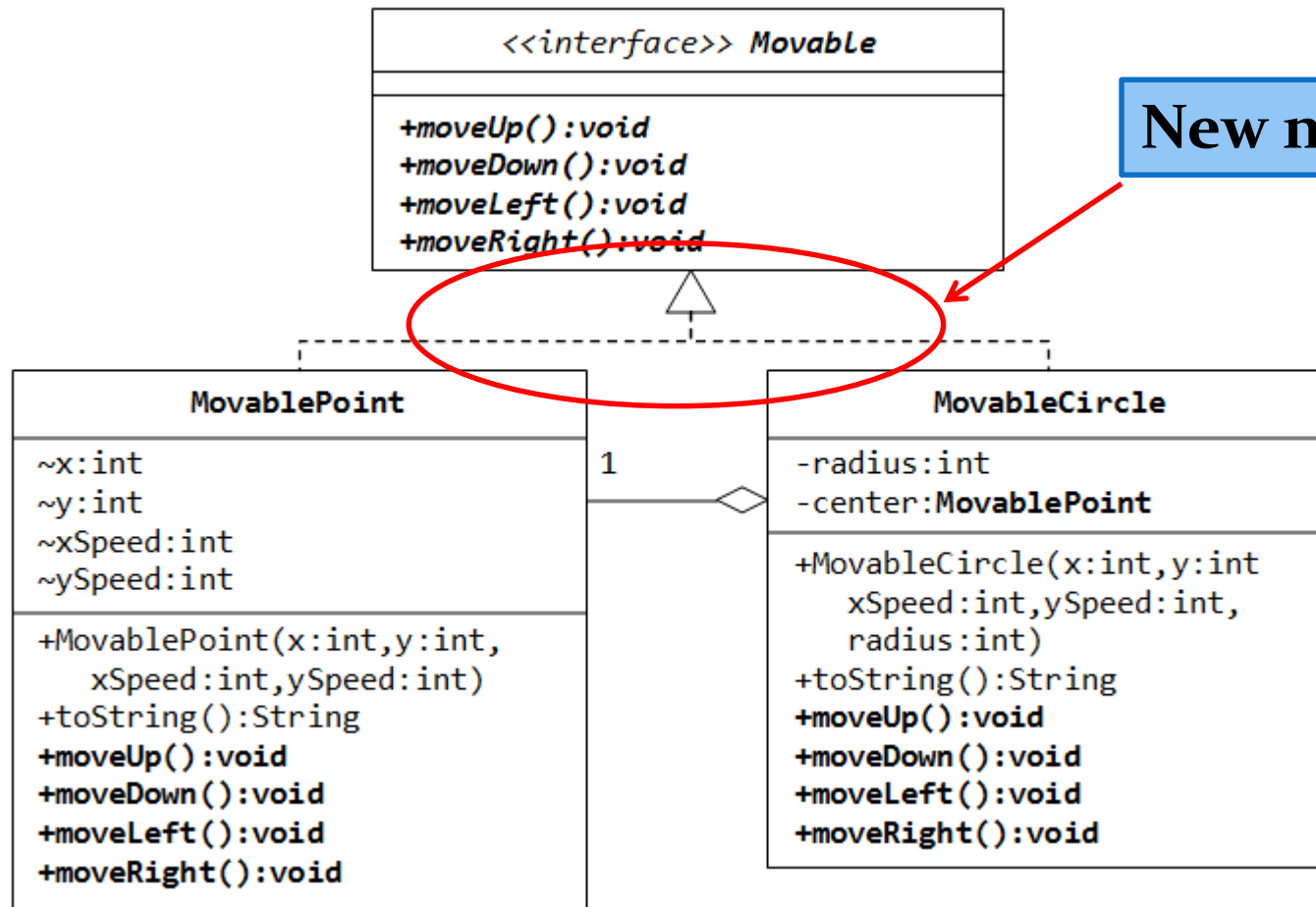
```
class ManyThings implements Interface1, Interface2, Interface3{  
    // contains all methods of all interfaces  
}
```

Abstraction: Interfaces

- An interface **cannot** be instantiated.
- Methods in an interface have **public visibility** by default.
- A class formally implements an interface by:
 - Stating so in the class header.
 - Providing implementations for each abstract method in the interface.


If a class asserts that it implements an interface, it **must define** all methods in the interface.

Abstraction: Interfaces



Abstraction: Interfaces

Interface



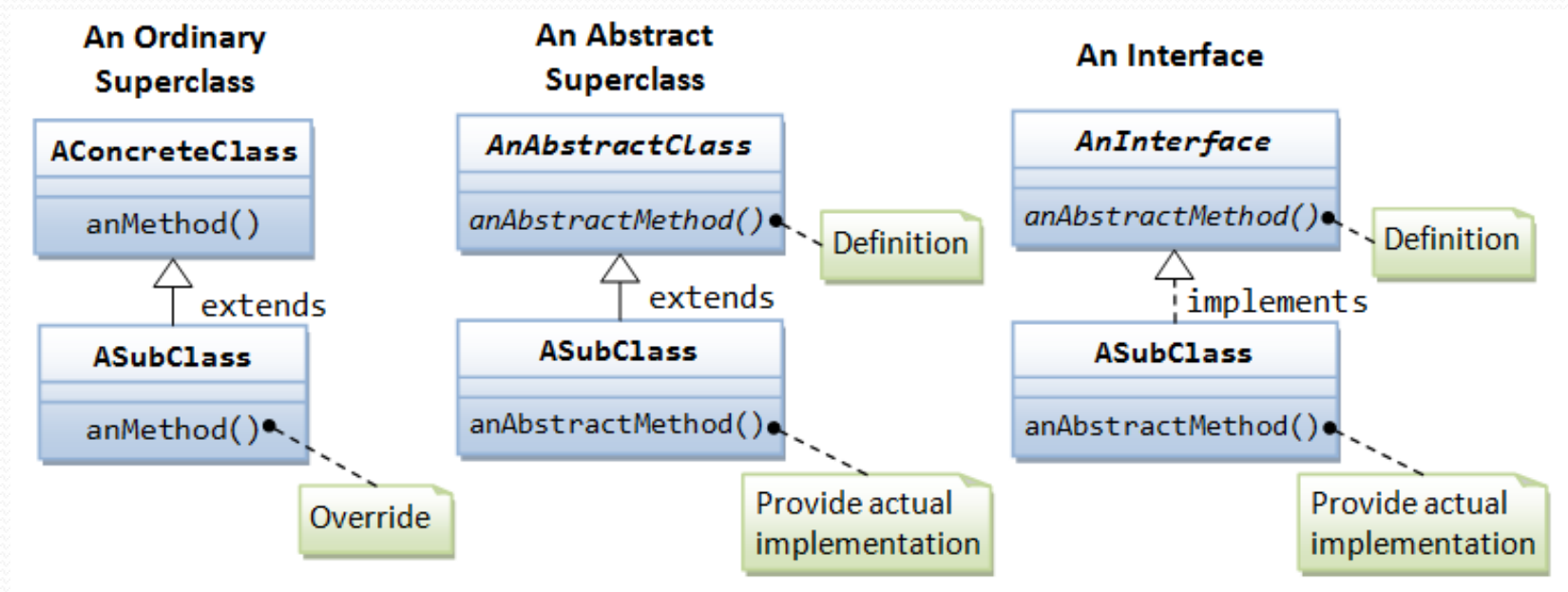
```
interface Movable {  
    public void moveUp();  
    public void moveDown();  
    ....  
}
```

```
public class MovablePoint implements Movable{  
    public void moveUp(){  
        System.out.println("MoveUp");  
    }  
    public void moveDown(){  
        System.out.println("MoveDown");  
    }  
    public String toString(){  
        System.out.println("A message");  
    }  
}
```

....

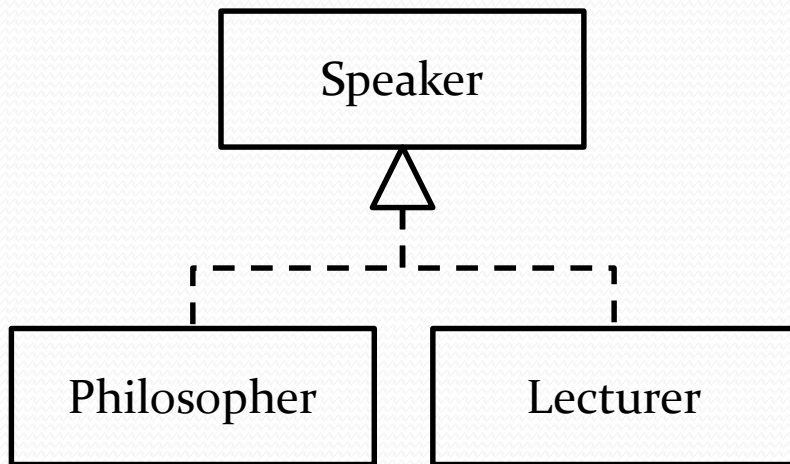
Abstraction: Interfaces

- Understanding relationships between classes and interfaces:



Polymorphism Via Interfaces

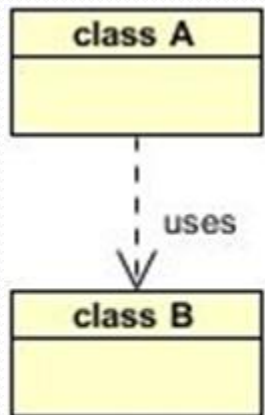
- An interface name can be used to declare an **object reference** variable.
- An interface reference can refer to any object of any class that implements that interface.



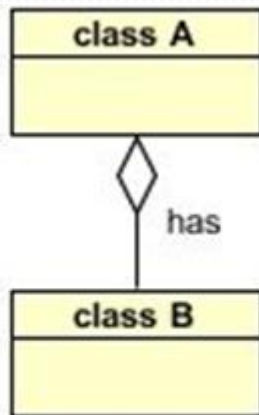
```
Speaker current;  
current = new Philosopher();  
current.someMethod();
```

```
current = new Lecturer();  
current.someMethod();
```

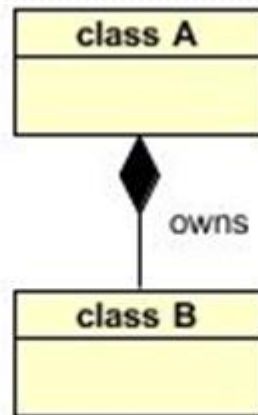
Summary



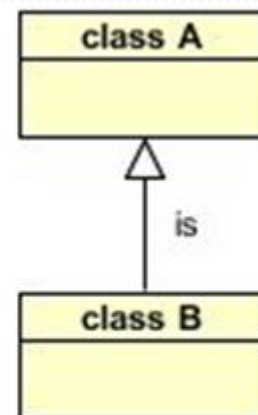
dependency/
usage



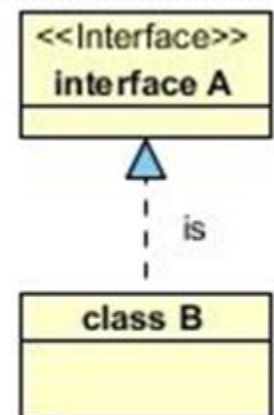
aggregation



composition



inheritance

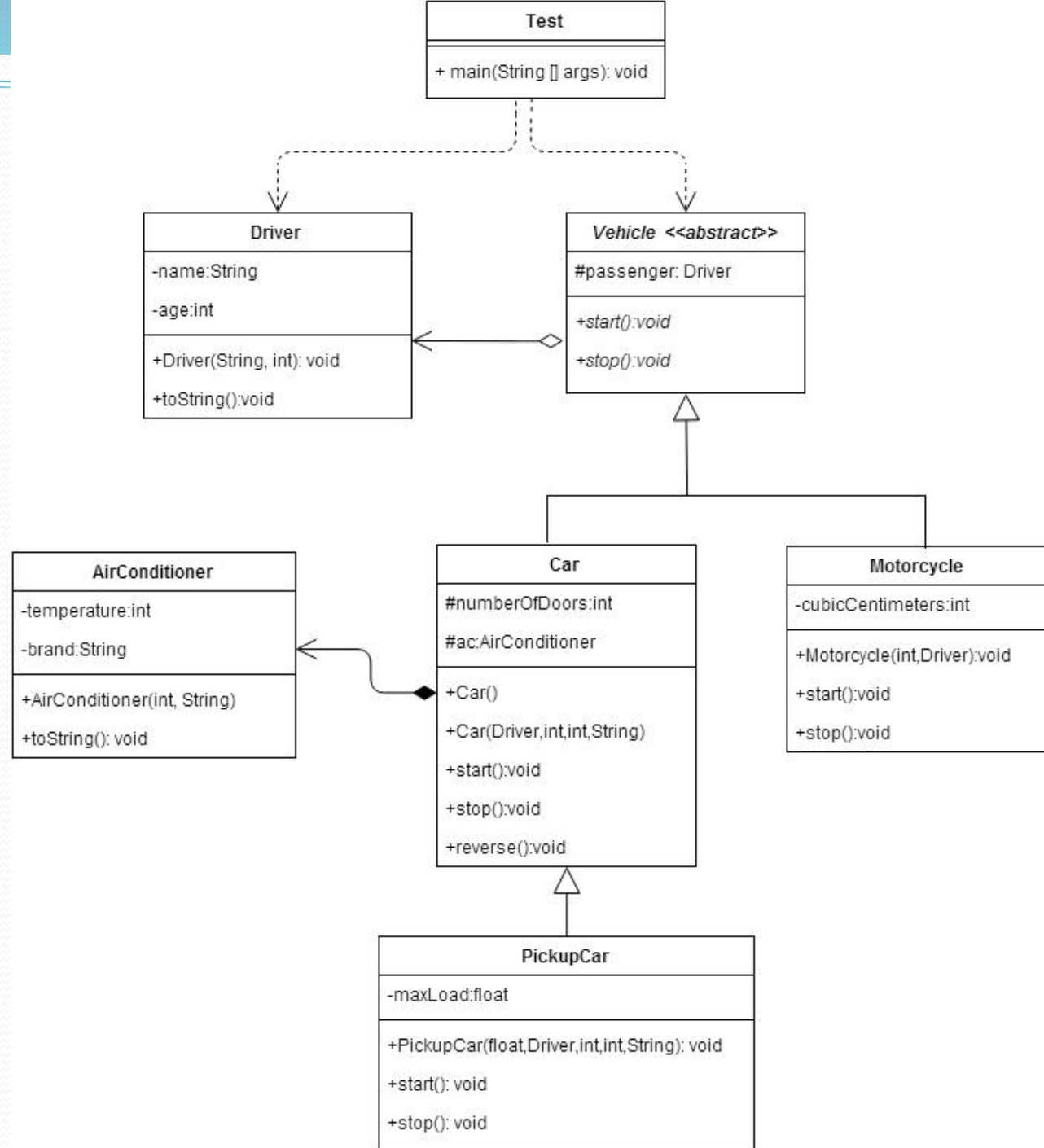


implementation

Has a

Is a

Lets Do Some Code



Lets Do Some Code

Detailed methods' signature:

- +Driver(String name, int age)
- +AirConditioner(int temp, String brand)
- +Car(Driver passenger, int numDoors, int temp, String brand)
- +PickupCar(float maxLoad, Driver passenger, int numDoors, int temp, String brand)
- +Motorcycle(int cc, Driver driver)

Discussion

