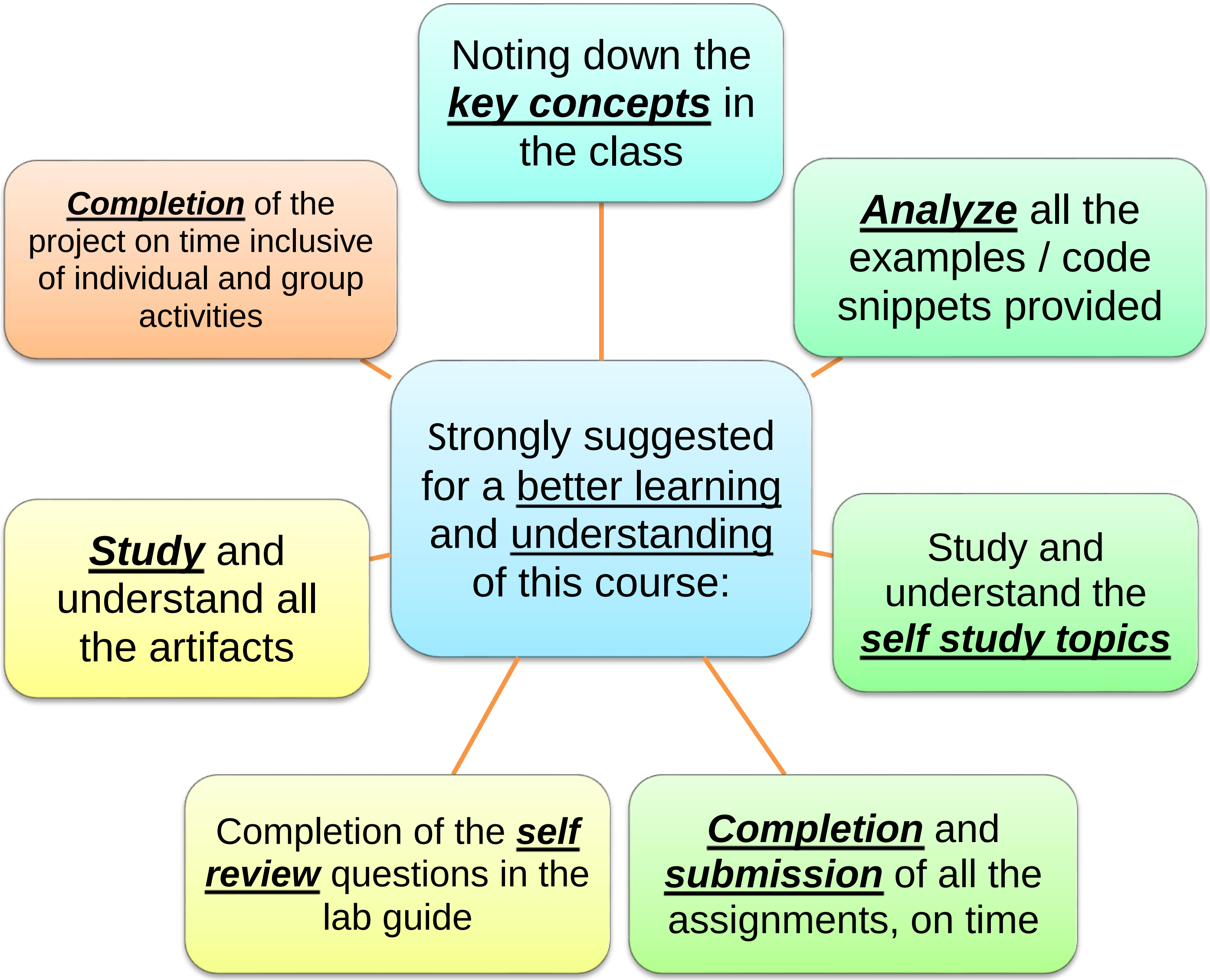
ADVANCED OOP WITH JAVA

Instructor:

Table of contents

- ◊ **Polymorphism**
  - ✓ Types of Polymorphism
  - ✓ Method Overloadding
  - ✓ Method Overriding
- ◊ **Abstraction:**
  - ◊ Abstract class
  - ◊ Interfaces
- ◊ **Static and Dynamic binding**
- ◊ **Inner classes**

Learning Approach



Strongly suggested for a better learning and understanding of this course:

Noting down the ***key concepts*** in the class

Analyze all the examples / code snippets provided

Study and understand the ***self study topics***

Completion and ***submission*** of all the assignments, on time

Completion of the ***self review*** questions in the lab guide

Study and understand all the artifacts

Completion of the project on time inclusive of individual and group activities

Section 1

POLYMORPHISM

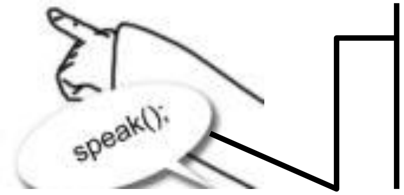
4 major principles of OOP

- **Inheritance**
- **Encapsulation**
- **Polymorphism**
- **Abstraction**

Polymorphism Overview (1/2)

- In object-oriented programming, **polymorphism** (from the Greek (Hy Lạp) meaning "**having multiple forms**").
  - ✓ Polymorphism is derived from 2 greek words: poly and morphs.
  - ✓ The word "*poly*" means *many* and "*morphs*" means *forms*.

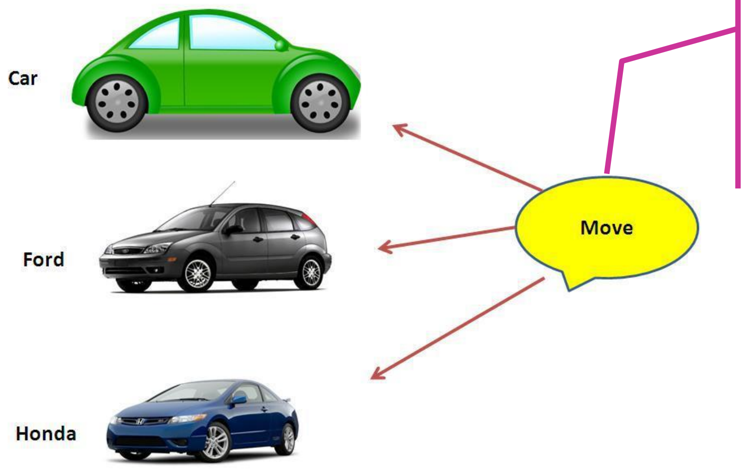
- There are two types of polymorphism in java:
  - ✓ compile time polymorphism: method overloading.
  - ✓ runtime polymorphism: method overriding.



one name, many forms.

Polymorphism Overview (2/2)

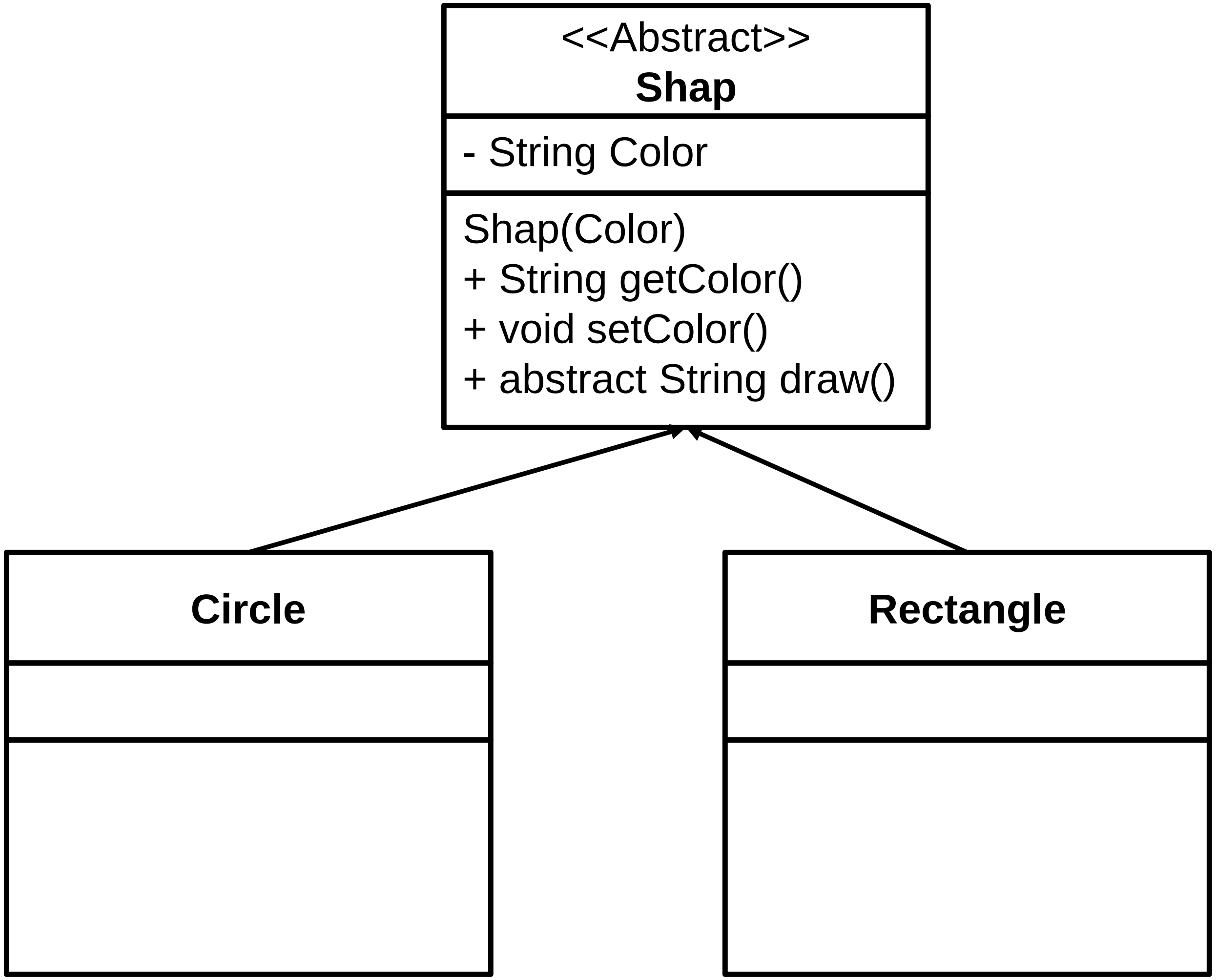
Polymorphism is a generic term for having many forms.



- Car uses normal engine to move
- Ford uses V engine to move
- Honda uses i-vtec technology to move

- You can use the same name for several different things
  - ✓ the compiler automatically figures out which version you wanted.
- There are several forms of polymorphism supported in Java, shadowing, overriding, and overloading.

Polymorphism example



<<Abstract>>
Shap

- String Color

Shap(Color)
+ String getColor()
+ void setColor()
+ abstract String draw()

Circle

Rectangle

Polymorphism example (1/4)

```java
public abstract class Shape {
    private String color;

    public Shape(String color) {
        this.setColor(color);
    }

    public String getColor() {
        return Color;
    }

    public void setColor(String color) {
        Color = color;
    }
    // abstract method
    abstract public String draw();
}
```

Polymorphism example (2/4)

```java
public class Circle extends Shape {
    /**
     * @param color
     */
    public Circle(String color) {
        super(color);
    }

    @Override
    public String draw() {
            return "I'm a " + this.getColor() + " circle.";
    }
}
```

Polymorphism example (3/4)

```java
public class Rectangle extends Shape {
    /**
     * @param color
     */
    public Rectangle(String color) {
        super(color);
    }

    @Override
    public String draw() {
        return "I'm a " + this.getColor() + " rectangle.";
    }
}
```

Polymorphism example (4/4)

```java
public class PolymorphismExample {
    private List<Shape> shapes = new ArrayList<Shape>();
    public PolymorphismExample() {
        Shape myFirstCircle = new Circle("Red");
        Circle mySecondCircle = new Circle("Blue");
        Rectangle myFirstRectangle = new Rectangle("Green");
        shapes.add(myFirstCircle);
        shapes.add(mySecondCircle);
        shapes.add(myFirstRectangle);
    }
    public List<Shape> getShapes() {
     return shapes;
    }
    public static void main(String[] args) {
        PolymorphismExample example = new PolymorphismExample();
        for (Shape shape : example.getShapes()) {
              System.out.println(shape.draw());
        }
    }
}
```

Output:

```
I'm a Red circle.
I'm a Blue circle.
I'm a Green rectangle.
```

Types of Polymorphism

- There are two types of polymorphism in java:
  - ✓ **Static Polymorphism** also known as **compile time** polymorphism
  - ✓ **Dynamic Polymorphism** also known as **runtime** polymorphism
- **Compile time Polymorphism** (or Static polymorphism)
  - ✓ Polymorphism that is resolved during compiler time is known as static polymorphism. Method overloading is an example of compile time polymorphism.
  - ✓ **Method Overloading**: This allows us to have **more than one method** having the **same name**, if the parameters of methods are different in number, sequence and data types of parameters.

Method Overloading

- Overloaded methods let you reuse the same method name in a class, but with different arguments (and optionally, a different return type).
- In a subclass,
  - ✓ you can overload the methods inherited from the superclass.
  - ✓ such overloaded methods neither hide nor override the superclass methods
  - ✓ they are new methods, unique to the subclass.

Method Overloading

- Some main rules for overloading a method:
  - ✓ Overloaded methods **must change the argument list**.
  - ✓ Overloaded methods **can change the return type**.
  - ✓ Overloaded methods **can change the access modifier**.
    - *With override: Cannot reduce the visibility*
  - ✓ A method can be overloaded in the same class or in a subclass.

Types of Polymorphism

```
public class SimpleCalculator {

  int add(int number1, int number2) {
    return number1 + number2;
  }

  int add(int number1, int number2, int number3) {
    return number1 + number2 + number3;
  }

  double add(double number1, double number2) {
    return number1 + number2;
  }
}
```

```
public class TestSimpleCalculator {

  public static void main(String[] args) {
    SimpleCalculator simpleCalculator = new SimpleCalculator();

    System.out.println(simpleCalculator.add(10, 20));
    System.out.println(simpleCalculator.add(10, 20, 30));
    System.out.println(simpleCalculator.add(12.5, 20.5));
  }
}
```

Practice time

- A program calculates and displays bonus amounts to pay various types of employees. There are 3 separate departments, numbered 1, 2, and 3.
  - ✓ ***Department 1*** employees are paid a bonus based on their sales: If their sales amount is over $5000 they get 5% of those sales, otherwise they get nothing.
  - ✓ ***Department 2*** employees are paid a bonus based on the number of units they sell: They get $100 per unit sold, and an extra $50 per unit if they sell more than 25 units; if they sell no units, they get nothing.
  - ✓ ***Department 3*** employees assemble parts in the plant and are paid a bonus of 10 cents per part if they reach a certain level: Part-time employees must assemble more than 250 parts to get the 10-cent-per-part bonus, and full-time employees must assemble more than 700.
- Write a set of 3 overloaded methods called getBonus() that works with the program below, according to the specifications described above.

Method Overriding

- Declaring a method in **subclass** which is already present in **parent class** is known as method overriding.
- The main advantage of method overriding is:
  - ✓ the class can give its own specific implementation to a inherited method without even modifying the parent class(base class).
- **Rules of method overriding in Java:**
  - ✓ Argument list: must be same as that of the method in parent class,
  - ✓ Access Modifier: cannot be more restrictive than the overridden method of parent class.

Overriding Examples

```java
class Mammal {
    String makeNoise() {
        return "generic noise";
    }
}

class Zebra extends Mammal {
    String makeNoise() {
        return "bray";
    }
}

public class ZooKeeper {
    public static void main(String[] args) {
        Mammal m = new Zebra();
        System.out.println(m.makeNoise());
    }
}
```

Shadowing

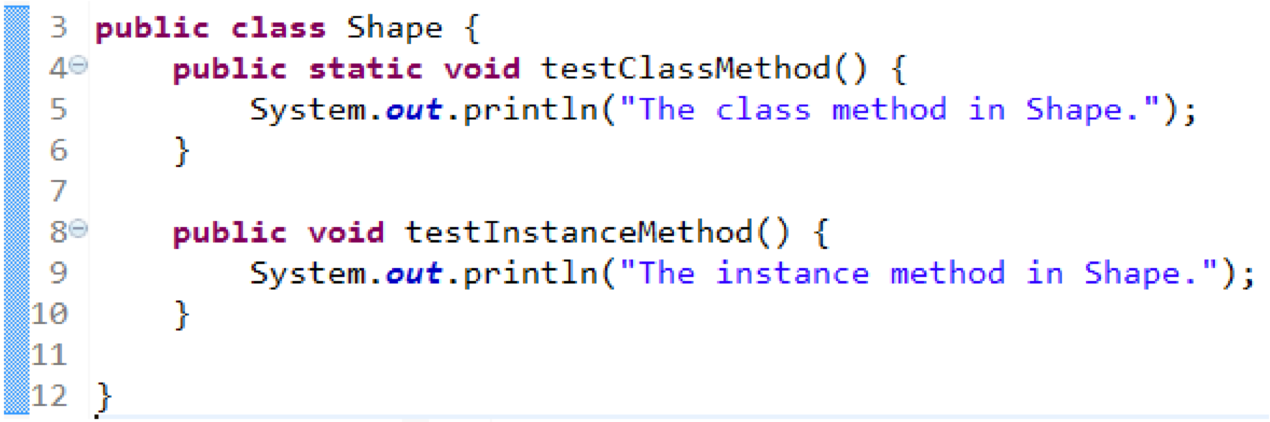
- This is called shadowing—name in class Dog shadows name in class Animal

```
public class Animal {
    String name = "Animal";
     public void speak() {
         System.out.println("generic speak!");
      }
    public static void main(String args[]) {
           Animal animal1 = new Animal();
     Animal animal2 = new Dog();
           Dog dog = new Dog();
           System.out.println(animal1.name + " " + animal2.name +" " + dog.name);
    }
}
class Dog extends Animal {
    String name = "Dog";
     public void speak() {
         System.out.println("dog speak!");
      }
}
```

```
Animal Animal Dog
```

Hiding method

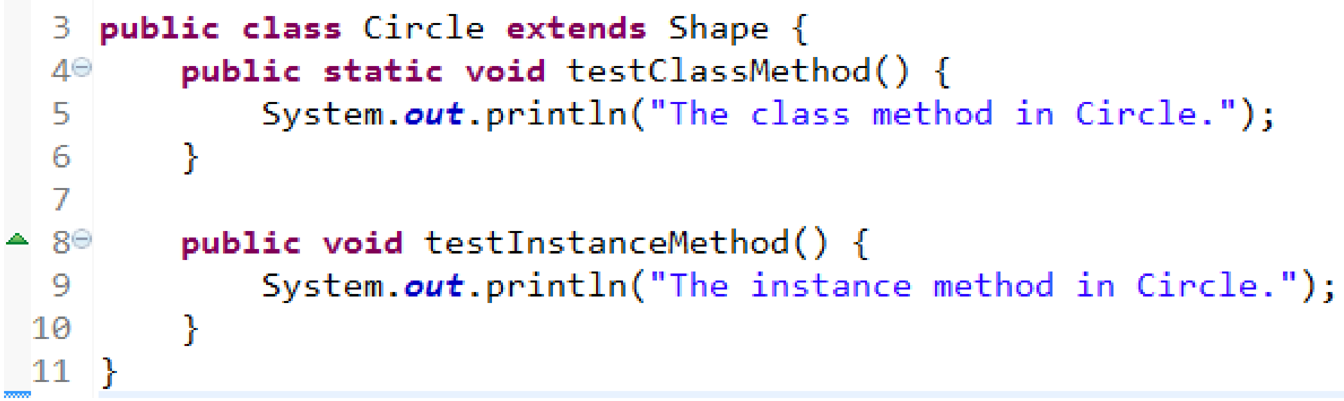
- If a subclass defines a **class method** with the same signature as a **class method** in the superclass, the method in the subclass hides the one in the superclass.



```java
public class Shape {
    public static void testClassMethod() {
        System.out.println("The class method in Shape.");
    }

    public void testInstanceMethod() {
        System.out.println("The instance method in Shape.");
    }

}
```



```java
public class Circle extends Shape {
    public static void testClassMethod() {
        System.out.println("The class method in Circle.");
    }

    public void testInstanceMethod() {
        System.out.println("The instance method in Circle.");
    }
}
```

Overriding and Hiding Example

```java
public class TestOverridingAndHiding {
    public static void main(String[] args) {
        Circle myCircle = new Circle();
        Shape myShape = myCircle;

        Shape.testClassMethod();
        myShape.testInstanceMethod();
    }
}
```

Output:

```
The class method in Shape.
The instance method in Circle.
```

Section 2

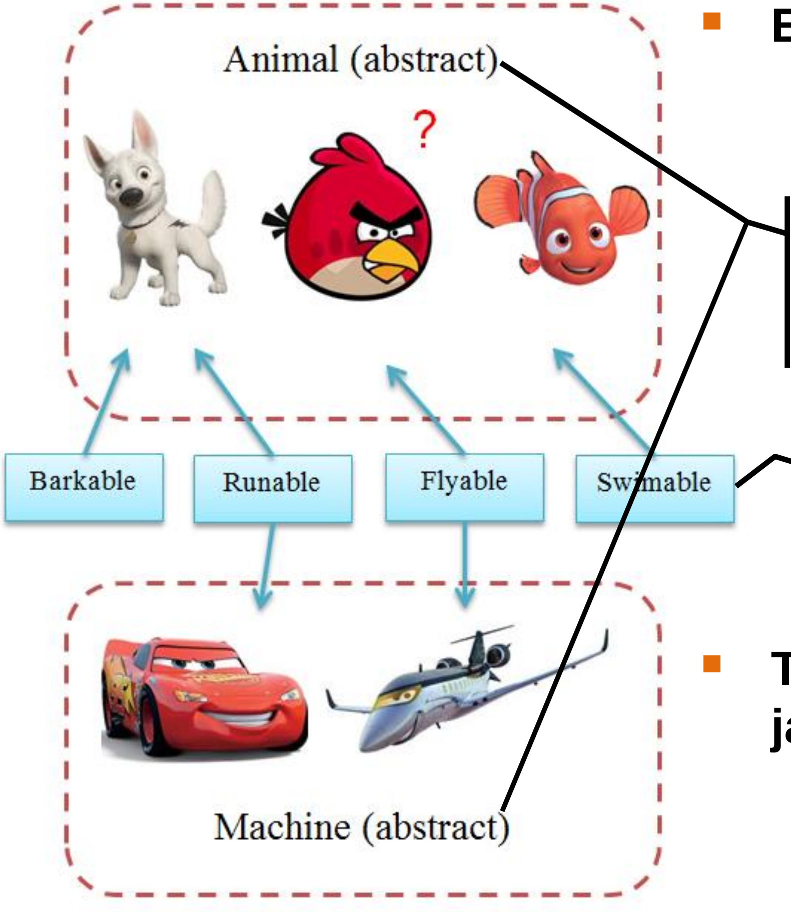
ABSTRACTION

4 major principles of OOP

- **Inheritance**
- **Encapsulation**
- **Polymorphism**
- **Abstraction**

Abstraction Overview

- **Abstraction** is a process of hiding the implementation details and showing only functionality to the user.



- **Example:**
  - ✓ Sending sms, you just type the text and send the message. You don't know the internal processing about the message delivery

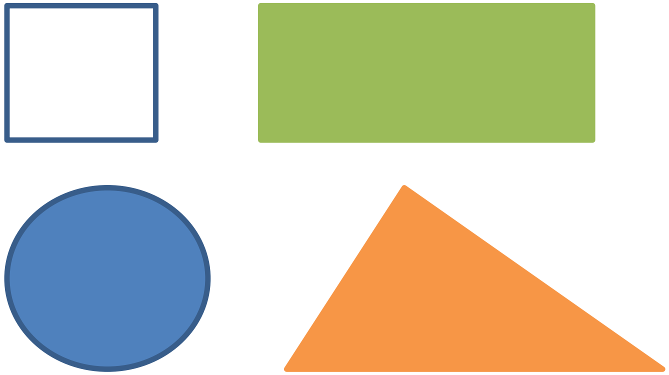
It shows only important things to the user and hides the internal details

Focus on what the object does instead of how it does it.

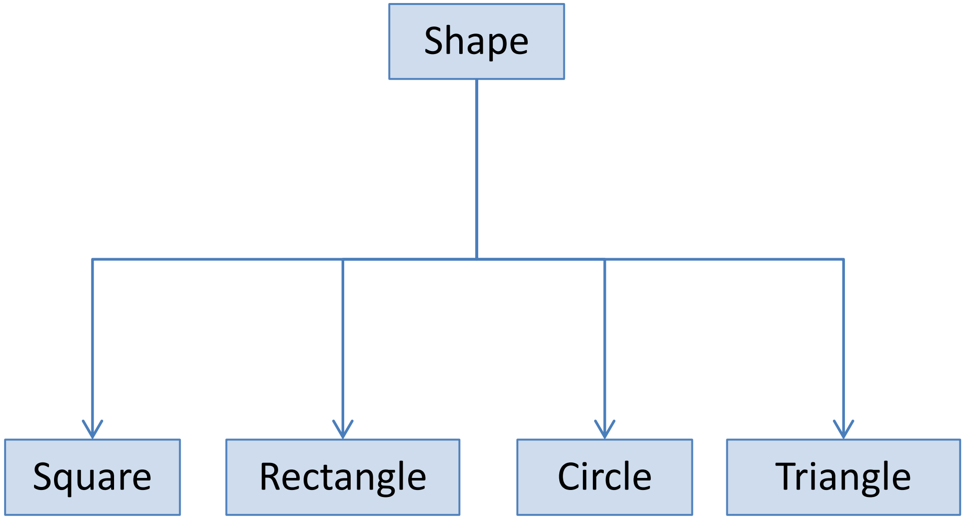
- **There are two ways to achieve abstraction in java:**
  - ✓ Abstract class (0 to 100%)
  - ✓ Interface (100%)

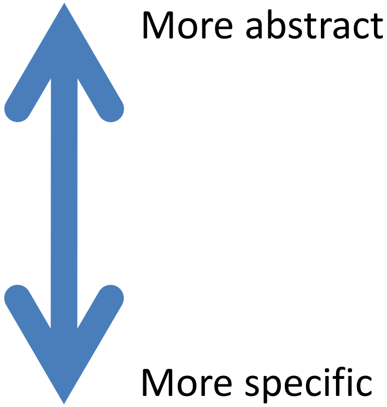
Purposes of Abstraction

- Abstraction is used to **manage complexity.**



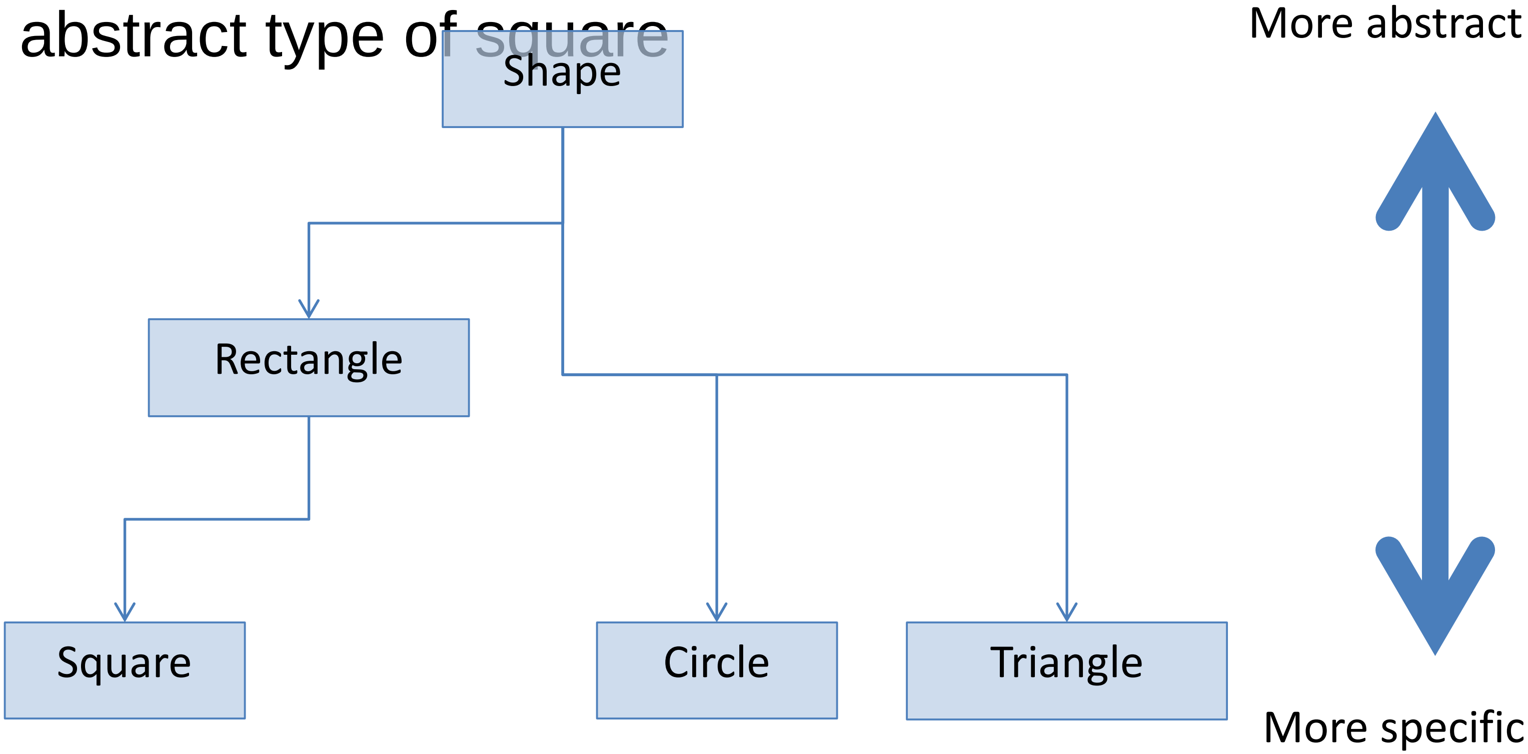
- All **Square**, **Rectangle**, **Circle** and **Triangle**:
  - ✓ Have *color*
  - ✓ Can *display*
- A shape may has all above characteristics:
  - ✓ So we call "**Shape**" is an abstract type of Square, Rectangle, Circle and Triangle.





Go abstraction

- Then, we analysis deeper:
  - ✓ A **rectangle** has four sides with lengths **w** and **h**
  - ✓ A **square** has all of the characteristics of a rectangle; in addition, **w** = **h**
- So, **square** is a type of rectangle, or, rectangle can be an abstract type of square



Types of Abstraction

- Abstraction can apply to **control** or to **data**
  - ✓ **Control abstraction** is the abstraction of actions:
    - Focus on “What”.
    - Define **just the behavior**
      - With very limited or no implementation logic.
    - Using *abstract* keyword at method and class level.
  - ✓ **Data abstraction** is the abstraction of data structures:
    - Collection API’s Collection, List, Set and Map interfaces are example of data abstractions.

Question?

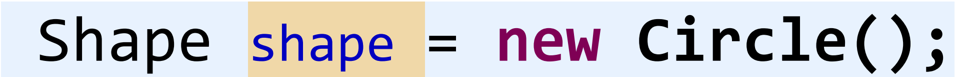
- Please answer this question:
  - ✓ A **circle** has *center* and *radius*.
  - ✓ A **dot** has all of the characteristics of a circle; in addition, *radius = 0*.
  - ✓ So, which is superclass and which is subclass?
- **Answer:**

- **Note:**
  - ✓ A dot has center
  - ✓ A circle has all of the characteristic of a dot; in addition, has radius

Abstract Class

Abstract class

- ✓ Is declared `abstract`
- ✓ Are superclasses (called abstract superclasses)
- ✓ An abstract class can not be **instantiated** (you are not allowed to create **object** of Abstract class), but they can be subclassed.



```
Shape shape = new Circle();
```

- ✓ May or may not include abstract methods
- ✓ When an abstract class is subclassed, the subclass usually provides implementations for all of the abstract methods in its parent class.
  - However, if it does not, the subclass must also be declared abstract.

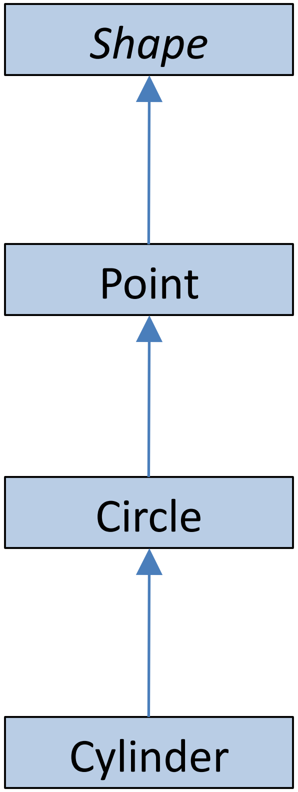
Abstract Method

- An abstract method is a method that is declared without an implementation.
  - ✓ **Example**: `abstract void moveTo(int x, int y);`
- If a class includes abstract methods, the class itself must be declared abstract.
- All of the methods in an interface are implicitly abstract:
  - ✓ so the abstract modifier is not used with interface methods.

Example

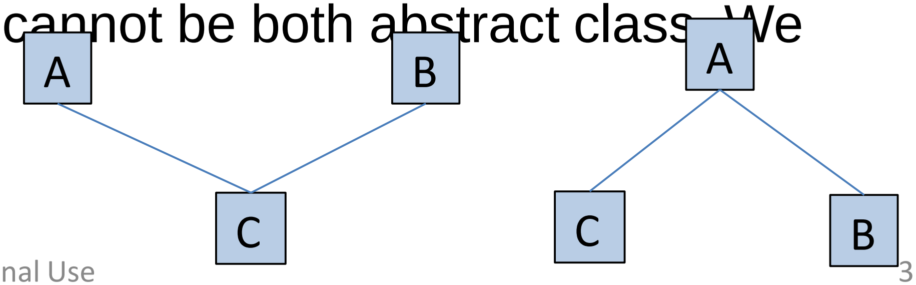
Application example

- ✓ Abstract class `Shape`
  - Declares `draw` as abstract method
- ✓ `Point, Circle, Cylinder` extends `Shape`
  - Each object can draw itself by implement `draw`



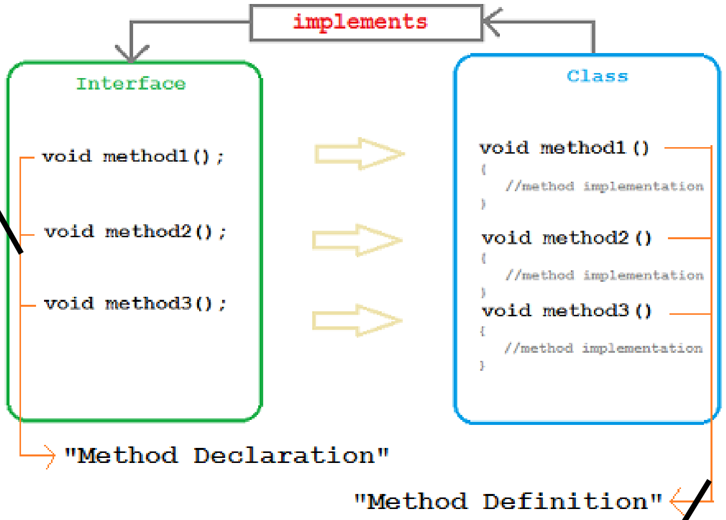
Interfaces

- Think of interface as a “pure” abstract class.
- It allows the creator to establish the form for a class: *method names*, *parameter lists* and *return types*, but no method bodies.
- **When to use interface**
  - ✓ Let B & C be classes. Assume we make A the parent class of B and C so A can hold the methods and fields that are common between B and C.
    - We can make A an abstract classes. The methods in A then indicate which methods must be implemented in B and C.
    - Sometimes all the methods of B must be implemented differently than the same method in C. Make A an **interface.**
  - ✓ Assume that C is a subclass of A and B. Since Java doesn’t support multi inheritance, so A and B cannot be both abstract class. We must use interface here.



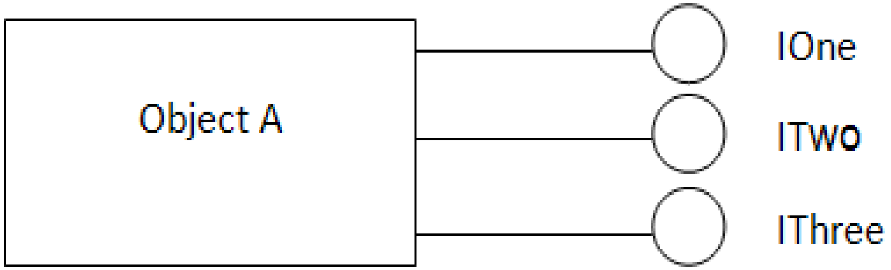
Interfaces

- ✓ An interface is a definition of method prototypes and possibly some constants (static final fields).
- ✓ An interface does not include the implementation of any methods.



- ✓ A class can **implement** an interface, this means that it provides implementations for all the methods in the interface.

- Java classes can implement any number of interfaces (multiple interface inheritance).

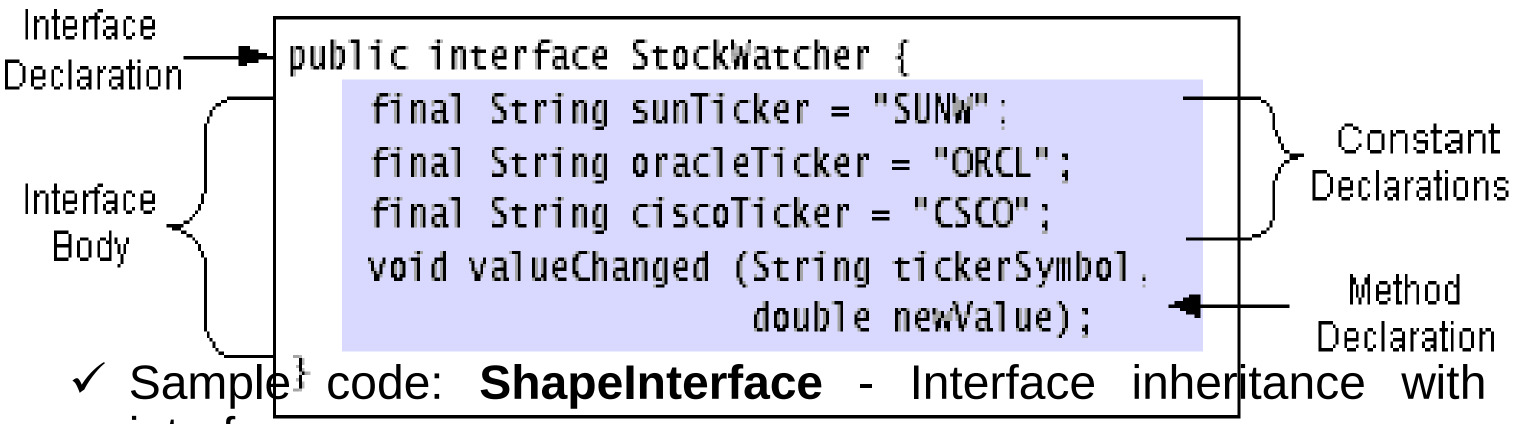


Interface Implementation

- **Syntax:**

  [public] **interface** <InterfaceName>[extends SuperInterface]
  {
  // InterfaceBody
  }

✓ **Example**:



```
public interface StockWatcher {
    final String sunTicker = "SUNW";
    final String oracleTicker = "ORCL";
    final String ciscoTicker = "CSCO";
    void valueChanged (String tickerSymbol,
                       double newValue);
}
```

✓ Sample code: **ShapeInterface** - Interface inheritance with Shape interface.

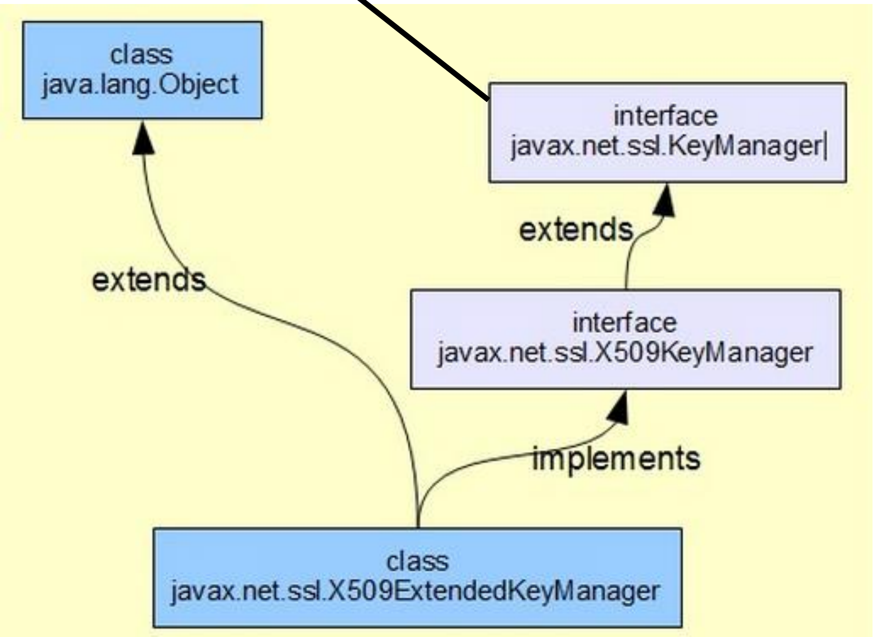
REFERENCE

Interfaces can be extended

- **Example:**

```
public interface Forward {
   void drive();
}
public interface Stop {
   void park();
}
public interface Speed {
   void turbo();
}
public class GearBox {
   public void move() {
}
```

one interface can extend another.



Interfaces can be extended (cont.)

- **Example:**

```
class Automatic extends GearBox implements Forward, Stop,
  Speed
{
    public void drive() {
      System.out.println("drive()");
    }
    public void park() {
      System.out.println("park()");
    }
    public void turbo() {
      System.out.println("turbo()");
    }
    public void move() {
      System.out.println("move()");
    }
}
```

Interfaces can be extended **(cont.)**

- **Example:**

```
    public class Car {
        public static void cruise(Forward x) {
          x.drive();
        }
        public static void park(Stop x) {
          x.park();
        }
        public static void race(Speed x) {
          x.turbo();
        }
        public static void move(GearBox x) {
          x.move();
        }
        public static void main(String[] args) {
            Automatic auto = new Automatic();
            cruise(auto); // Interface Forward
            park(auto); // Interface Stop
            race(auto); // Interface Speed
            move(auto); // class GearBox
    }
}
```

Practice time

- In class diagrams, as shown in following Figure. Let’s implement it using Java:

Abstract class and Interface

- **Abstract class** and **interface** both are used to achieve abstraction where we can declare the abstract methods. Abstract class and interface both can't be instantiated.
- But there are many differences between abstract class and interface that are given below.

No.	Abstract class	Interface
1	Abstract class can **have abstract and non-abstract** methods.	Interface can have **only abstract** methods.
2	Abstract class **doesn't support multiple inheritance**.	Interface **supports multiple inheritance**.
3	Abstract class **can have final, non-final, static and non-static variables**.	Interface has **only static and final variables**.
4	Abstract class **can have static methods, main method and constructor**.	Interface **can't have static methods, main method or constructor**.
5	Abstract class **can provide the implementation of interface**.	Interface **can't provide the implementation of abstract class**.
6	The **abstract keyword** is used to declare abstract class.	The **interface keyword** is used to declare interface.

7	**Example:**
	public abstract class Shape{
	public abstract void draw();
	}
		Example:
		public interface Drawable{
		void draw();
		}

Section 3

STATIC AND DYNAMIC BINDING

Static and Dynamic binding

- There are two types of binding:
  - ✓ **Static Binding** that happens at compile time and
  - ✓ **Dynamic Binding** that happens at runtime
- Static binding
  - ✓ The binding which can be resolved at compile time by compiler is known as static or early binding.
  - ✓ The binding of static, private and final methods is **compile-time**. The reason is that the these method cannot be overridden and the type of the class is determined at the compile time.

Static binding

```
public class Boy extends Human {
  public static void walk() {
    System.out.println("Boy walks");
  }

  public static void main(String args[]) {

    /* Reference is of Human type and object is
     * Boy type
     */
    Human obj = new Boy();
    /* Reference is of HUman type and object is
     * of Human type.
     */
    Human obj2 = new Human();
    obj.walk();
    obj2.walk();
  }
}

class Human {
  public static void walk() {
    System.out.println("Human walks");
  }
}
```

Output:

```
Human walks
Human walks
```

Dynamic binding

- When compiler is not able to resolve the call/binding at compile time, such binding is known as Dynamic or late Binding.
- Method Overriding is a perfect example of dynamic binding as in overriding both parent and child classes have same method and in this case the **type of the object** determines which method is to be executed.
- The type of object is determined at the run time so this is known as dynamic binding.

Dynamic binding

```
public class Boy extends Human {
  public void walk() {
    System.out.println("Boy walks");
  }

  public static void main(String args[]) {

    /* Reference is of Human type and object is
     * Boy type
     */
    Human obj = new Boy();
    /* Reference is of HUman type and object is
     * of Human type.
     */
    Human obj2 = new Human();
    obj.walk();
    obj2.walk();
  }
}

class Human {
  public void walk() {
    System.out.println("Human walks");
  }
}
```

Output:

```
Boy walks
Human walks
```

Summary

- Polymorphism, which means "many forms,"
  - ✓ is the *ability to treat* an *object of any subclass* of a base class as if it *were an object of the base class*.
- Abstract class is a class that may contain abstract methods and implemented methods.
  - ✓ An *abstract method* is one *without a body* that is declared with the reserved word abstract
- An interface is a collection of constants and method declarations.
  - ✓ When a class implements an interface, it must declare and provide a method body for each method in the interface

Thank you