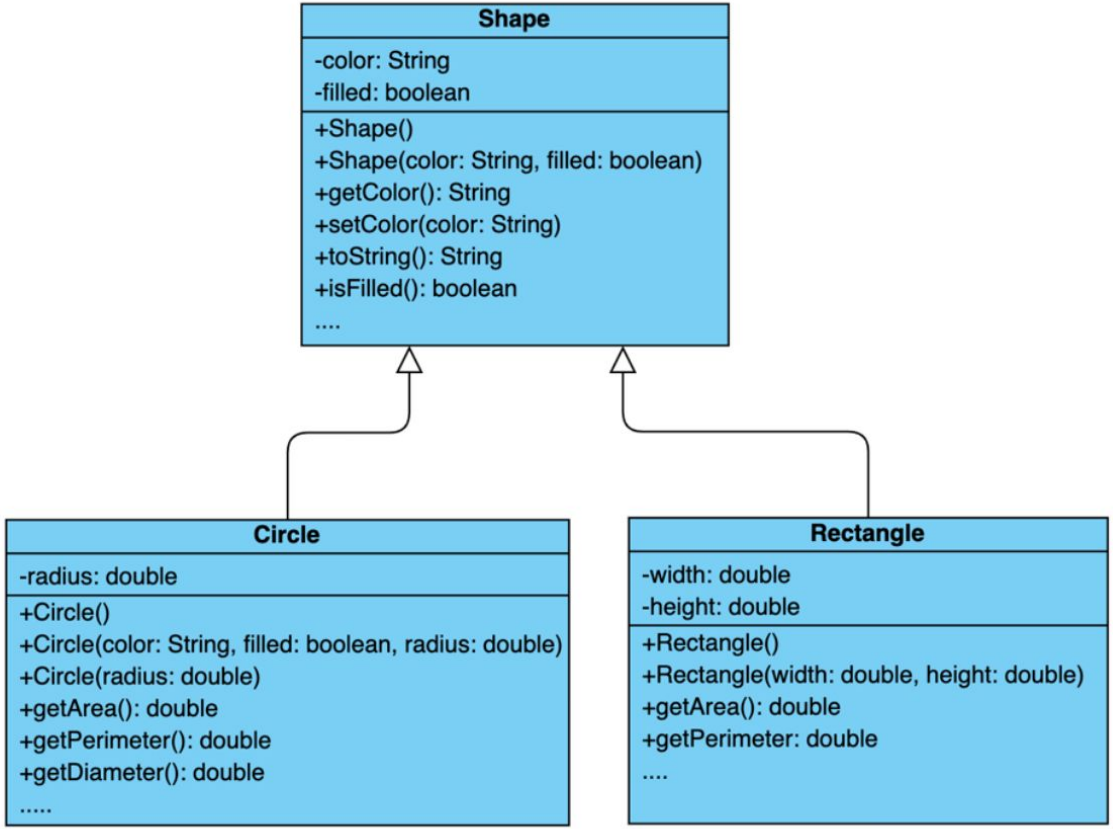
CS213 Recitation 3

- **Superclass (parent class) and Subclass (child class) in Java**
- **Overriding and overloading**
- **Polymorphism**
- **The Protected Data and Methods**
- **Abstract Class and Java Interface**

Superclass (parent class) and Subclass (child class) in Java



In Java, the keyword **extends** is used to tell the compiler that the subclass extends the superclass. For example, the **Circle** class above extends the **Shape** class with the following syntax and inherits the data fields and all the public methods defined in the Shape class.

```
public class Circle extends Shape
```

subclass superclass

The statement **super()** at the first line in the subclass constructor invokes the default constructor of its superclass, and the statement **super(arguments)** invokes the superclass constructor that matches the **arguments**. The statement **super()** or **super(arguments)** must be the first statement of the subclass's constructor. Invoking a superclass constructor's name in a subclass causes a syntax error. Using "super" is the only way to explicitly invoke a superclass constructor. For example,

```
public Circle(double radius, String color, boolean filled) {
    super(color, filled); //invoke the superclass constructor
    this.radius = radius;

}
```

```
// Define the superclass Animal
class Animal {
    String name;

    Animal(String name) {
        this.name = name;
    }

    void speak() {
        System.out.println("Generic animal sound");
    }
}

// Define the subclass Dog which inherits from Animal
class Dog extends Animal {

    Dog(String name) {
        super(name);  // Calls the constructor of the superclass Animal
    }

    @Override
    void speak() {
        System.out.println(name + " says Woof!");
    }
}
```

```
// Define the subclass Cat which inherits from Animal
class Cat extends Animal {

    Cat(String name) {
        super(name);  // Calls the constructor of the
superclass Animal
    }

    @Override
    void speak() {
        System.out.println(name + " says Meow!");
    }
}

// Main class to test the functionality
public class Main {
    public static void main(String[] args) {
        Dog dog = new Dog("Buddy");
        Cat cat = new Cat("Kitty");

        dog.speak();
        cat.speak();
    }
}
```

```
Buddy says Woof!
Kitty says Meow!
```

A constructor may invoke an overloaded constructor or its superclass constructor. If neither is invoked explicitly, the compiler automatically puts **super()** as the first statement in the constructor. For example:

```
public ClassName() {
  // some statements
}
```

Equivalent

```
public ClassName() {
  super();
  // some statements
}
```

```
public ClassName(parameters) {
  // some statements
}
```

Equivalent

```
public ClassName(parameters) {
  super();
  // some statements
}
```

- **Overriding vs. Overloading**

Overloading means to define multiple methods with the same name but different signatures. Overriding means to provide a new implementation for a method in the subclass. For example, in TestOverriding class, the method **p(double i)** in class **A** overrides the same method defined in class **B**. In the TestOverloading class however, the class **A** has two overloaded methods: **p(double i)** and **p(int i)**. The method **p(double i)** is inherited from **B**.

```
public class TestOverloading {
    public static void main(String args[]) {
        A a = new A();
        a.p(10);
        a.p(10.0);
    }
}

class B {
    public void p(double i) {
        System.out.println(i * 2);
    }
}

class A extends B {
    public void p(int i) {
        System.out.println(i); //overload the method in B
    }
}
```

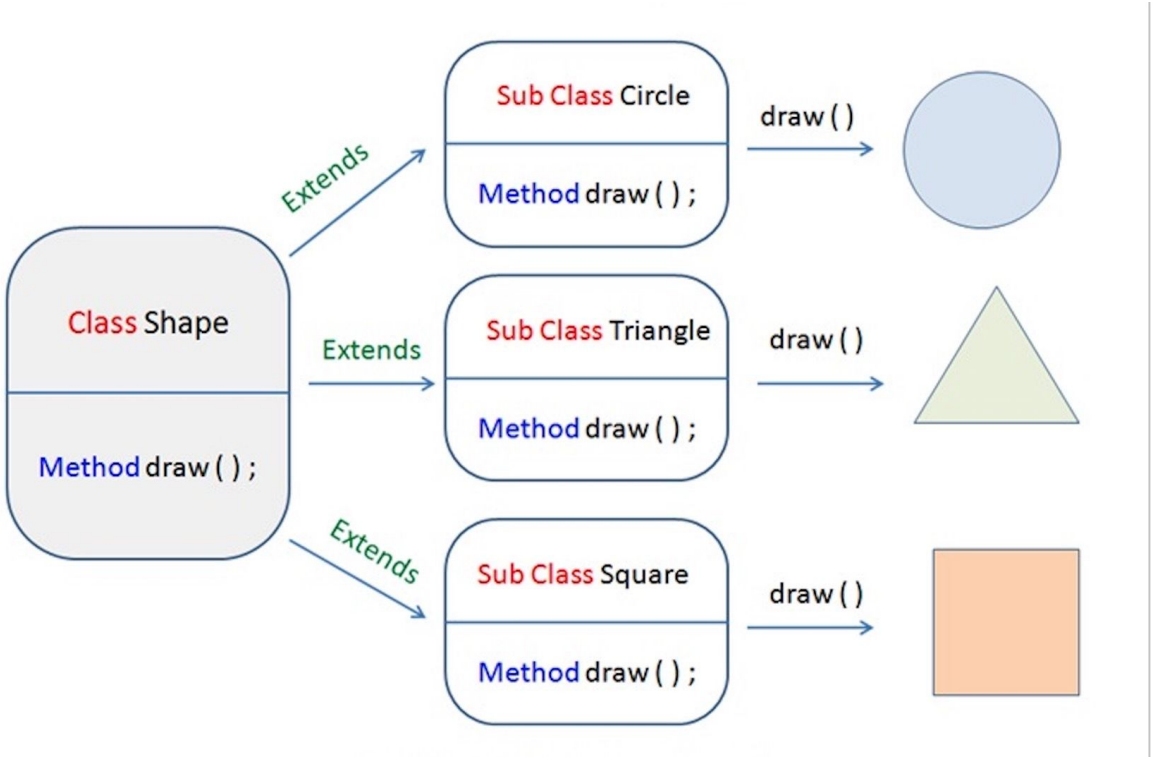
```
public class TestOverriding {
    public static void main(String args[]) {
        A a = new A();
        a.p(10);
        a.p(10.0);
    }
}

class B {
    public void p(double i) {
        System.out.println(i * 2);
    }
}

class A extends B {
    public void p(double i) {
        System.out.println(i); //override the method in B
    }
}
```

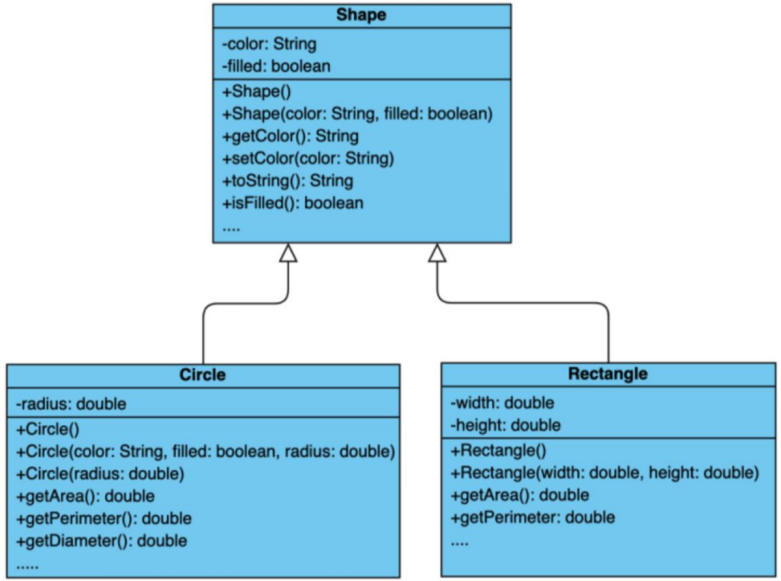
- **Polymorphism**

An object of a subclass can be used wherever its superclass object is used. This is commonly known as polymorphism.



```java
public class PolymorphismDemo {
    public static void main(String[] args) {
        displayObject(new Circle(1, "red", false));
        displayObject(new Rectangle(1, 1, "black", true));
    }

    public static void displayObject(Shape object) {
        System.out.println(object.toString() + " color is "
                           + object.getColor());
    }
}
```



Since a method can be implemented in several classes along the inheritance chain, which method will be invoked?

The Java Runtime System checks **the data type of the actual object** and uses the method from that type (rather than the method from the type of the reference variable).

So, in the following example, Object o will call the method toString() defined by class Shape.

```
Object o = new Shape("red", false);
System.out.println(o.toString());
```

A variable must be declared a type. The type that declares a variable is called the variable's declared type.

```
Object o = new Shape("red", false);
System.out.println(o.toString());
```

The actual type of the variable is the actual class for the object referenced by the variable.

```
Object o = new Shape("red", false);
System.out.println(o.toString());
```

This is known as **dynamic binding**.

If Class Shape don't have toString() method, which method will be used?

Dynamic binding works as follows: Suppose that an object o is an instance of classes C_1. There is an inheritance chain C_1, C_2, . . . , C_{n-1}, and C_n, where C_1 is a subclass of C_2, C_2 is a subclass of C_3, . . . , and C_{n-1} is a subclass of C_n, as shown below. That is, C_n is the most general class, and C_1 is the most specific class. In Java's inheritance chain, C_n is the Object class. If object o is an instance of C_1 and invokes a method p, the JVM searches for the implementation of the method p in the order of C_1, C_2, . . . , C_{n-1}, and C_n, until the p method is found. Once an implementation is found, the search stops and **the first-found implementation is invoked**.

How to take advantages of these features of JAVA? Remember this:

- **LSP (Liskov Substitution Principle)**

1. A child class (subclass) can be used wherever a parent class (superclass) is expected; child class must be completely substitutable for their parent class
2. For every overriding method in a child class: require no more, promise no less; make sure a child class just extend without replacing the functionality of the superclass
3. Use the notion of "is-a" and LSP to determine if your inheritance is good - don't overdo inheritance!

- **Casting Objects and the instanceof Operator**

One object reference can be typecast into another object reference. This is called casting object.

It is always possible to cast an instance of a subclass to a variable of a superclass (known as **upcasting**) because an instance of a subclass is always an instance of its superclass. When casting an instance of a superclass to a variable of its subclass (known as **downcasting**), explicit casting must be used to confirm your intention to the compiler with the (SubclassName) cast notation. For the casting to be successful, you must make sure the object to be cast is an instance of the subclass.

```
Student s = o;    Student s = (Student) o;
```

- **The Protected Data and Methods**

If you want to allow subclasses to access data fields or methods defined in the superclass, but not to allow non-subclasses in different packages to access these data fields and methods. You can use the **protected** keyword.

When a data member, method or constructor is declared protected, it can be accessed:

- **within the same class,**
- **within the package by other classes,**
- **outside the package but through inheritance by subclass.**

```
// Define a class 'Animal' in a file named Animal.java
public class Animal {
    // A protected data member 'name' of type String
    protected String name;

    // A protected method 'sound' that prints a generic sound
    protected void sound() {
        System.out.println("Some generic animal sound");
    }
}
```

```
// Define a subclass 'Dog' of 'Animal' in a file named Dog.java
public class Dog extends Animal {

    // Constructor for Dog class
    public Dog(String name) {
        // Accessing the protected data member 'name' from superclass
        this.name = name;
    }

    // Overriding the protected method 'sound' from superclass
    @Override
    protected void sound() {
        // Accessing the protected method 'sound' from superclass
        super.sound();
        // Printing a specific sound for Dog class
        System.out.println("Bark");
    }

    public static void main(String[] args) {
        // Creating an object of Dog class
        Dog dog = new Dog("Buddy");
        // Accessing the protected method 'sound' through Dog object
        dog.sound();
    }
}
```

- **Abstract Class and Java Interface**

A class containing abstract methods is called an abstract class. In other words, an abstract class is a class with one or more abstract methods, which are the methods with no body, not even an empty one

```
public abstract class Figure {
   private int fixedX, fixedY;
   public Figure( int fx, int fy ) {
   ......
   }
   public abstract void draw();   // No way to draw it!
}

public class Rectangle extends Figure {
//must implement draw(), or abstract keyword is needed
   private int width, height;
   public Rectangle ( ...... ) {
   }
   public void draw() {
      ......  // the concrete method to draw the rectangle
   }
}
```

Java doesn't allow multiple inheritance (while C++ does) since multiple inheritance is problematic. However, Java has another type of inheritance: **Interface Inheritance**. Adding interface inheritance gives a lot of the power of multiple inheritance, but none of the problems. A Java “Interface” is a special "class" with no data and only abstract methods. A class can "extend" only one class but can implement many interfaces. For example,

```
public class AAA extends BBB implements CCC, DDD, EEE {
   .....
}
```

As an example, in the following class diagram, you can use the **Edible interface** to specify whether an object is edible. This is accomplished by letting the class for the object implements this interface using the **implements** keyword in Java code. Note that, in a class diagram, you use the stereotype **<<interface>>** above the class name to represent an interface class. The name of the interface is italicized. An arrow with a dashed line for the arrow shaft indicates implementation of an interface. In UML, such a relationship is called a "realization".