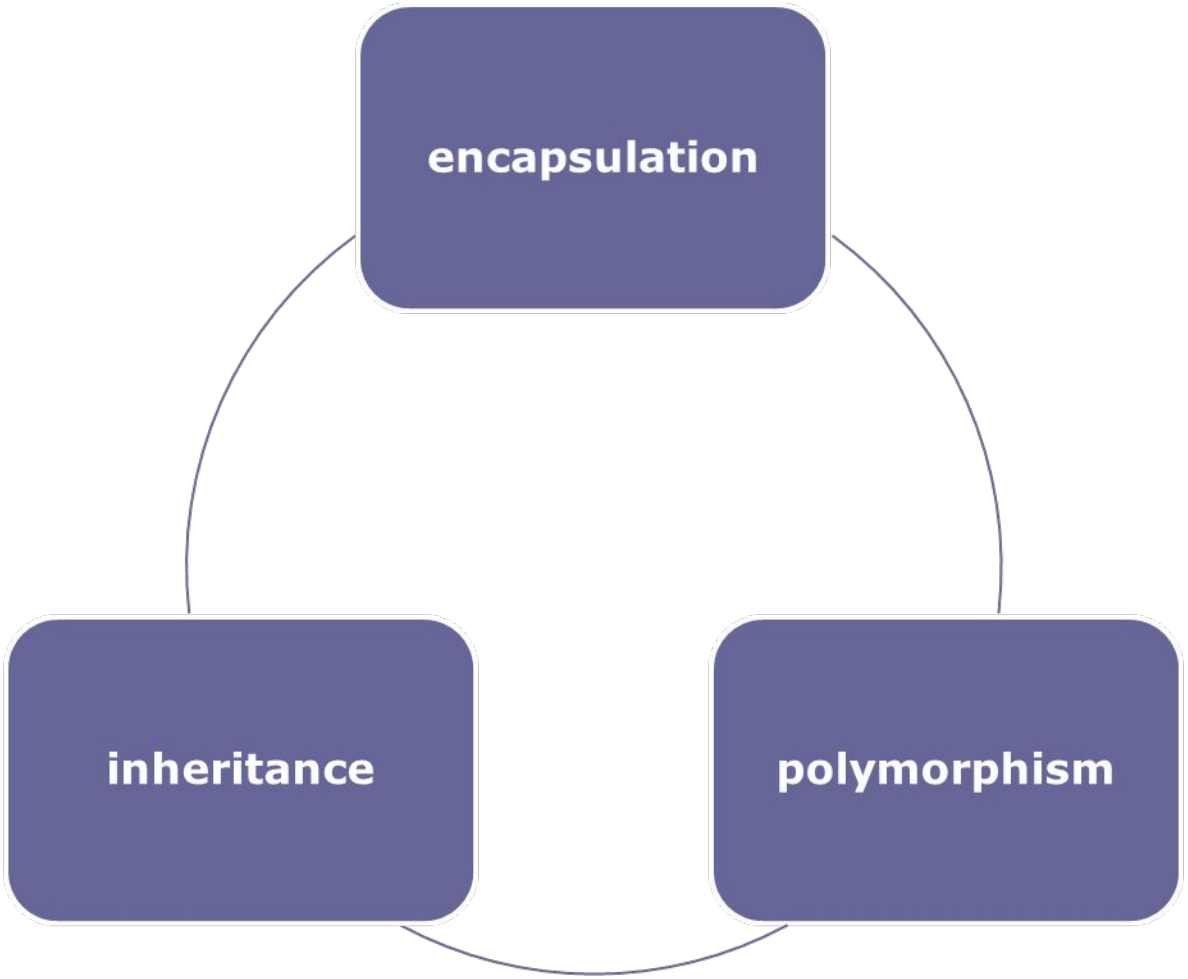
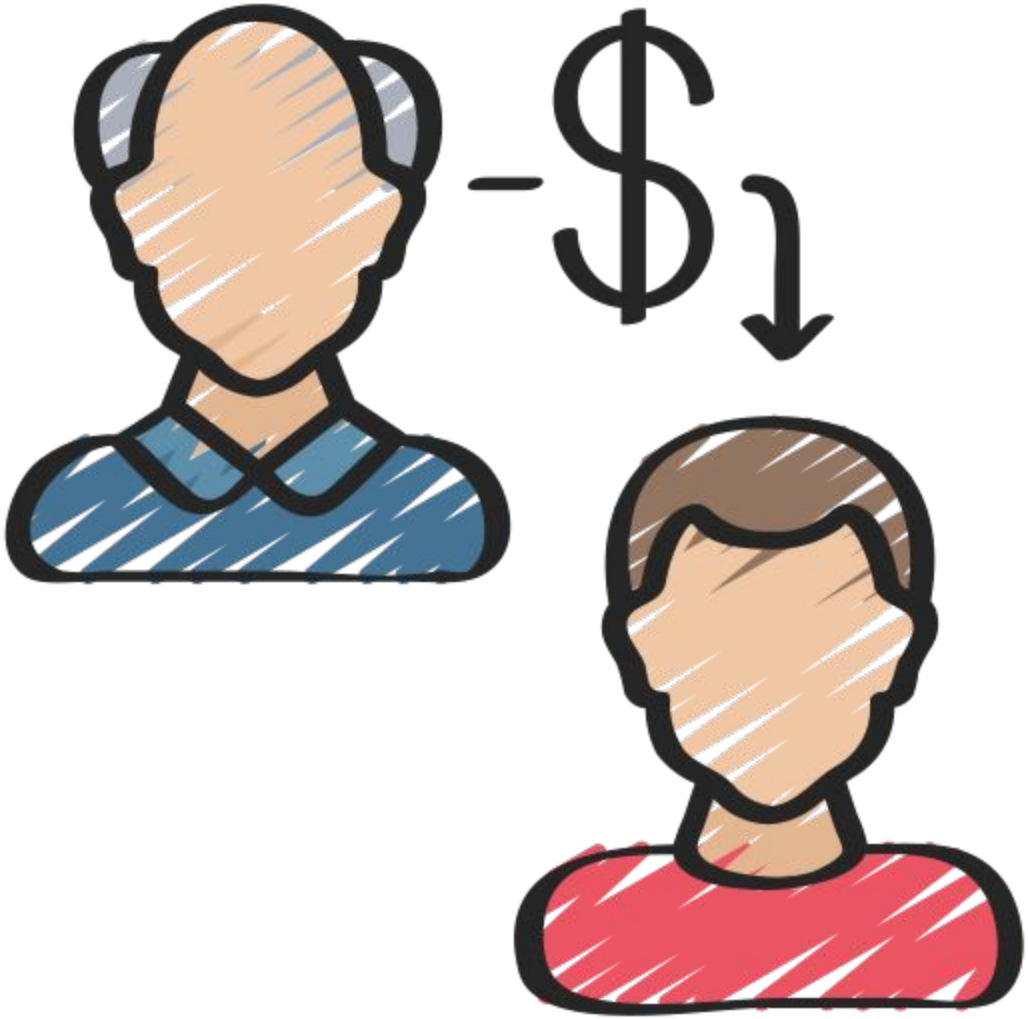
OOP Principles
Inheritance & Polymorphism

Moumita Asad
Lecturer
IIT, DU

OOP Principles

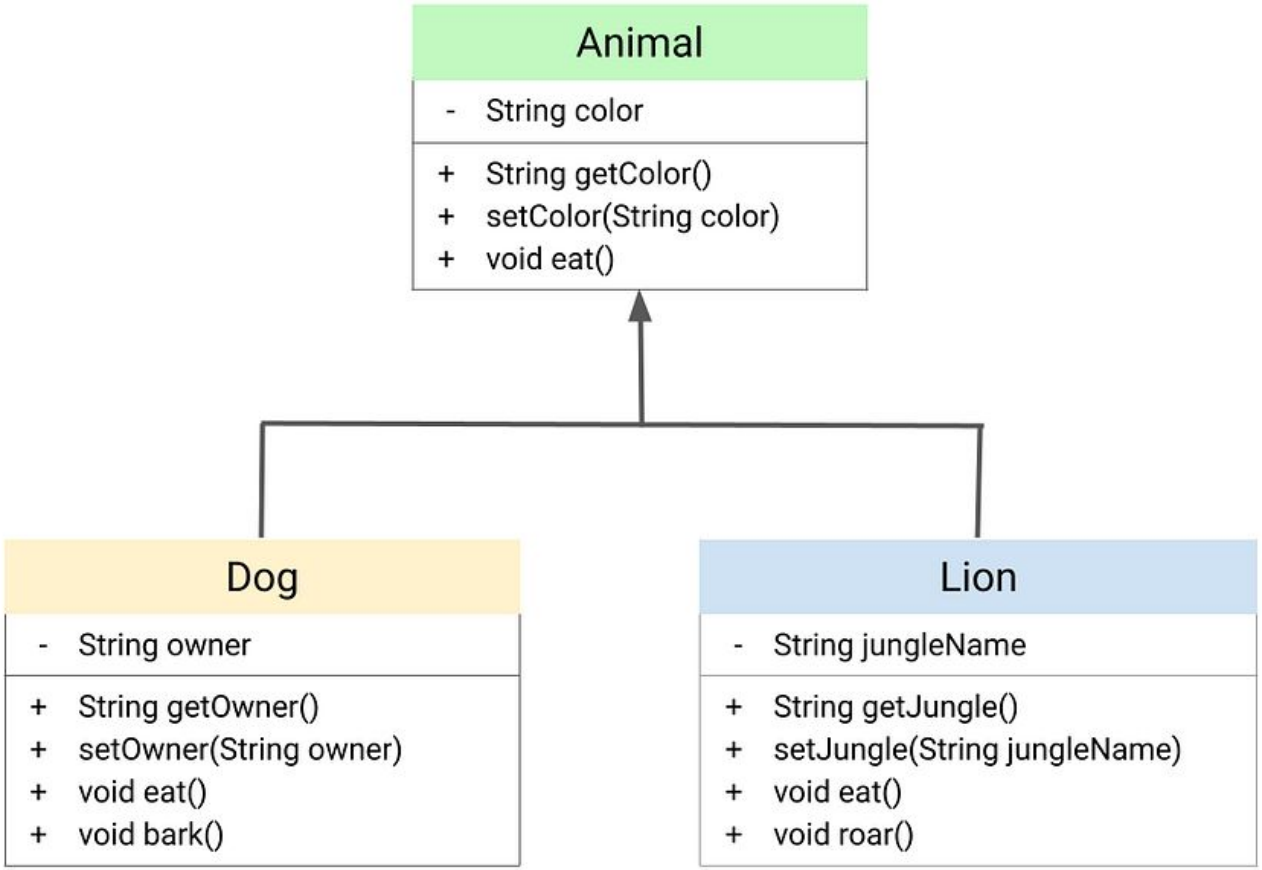


Inherit



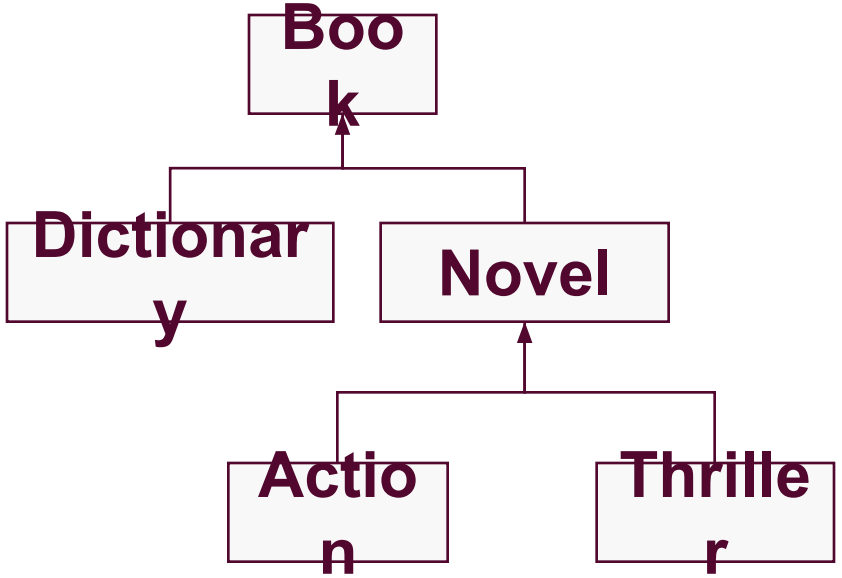
Inheritance

- In OOP, one class can inherit the attributes and methods of another class



Inheritance

- It allows a software developer to derive a new class from an existing one
- The existing class is called the *parent class* or *superclass*
- The derived class is called the *child class* or *subclass*
- The subclass is a more specific version of the Original



Inheritance

- The child class inherits the methods and data defined for the parent class
- To tailor a derived class, the programmer can add new variables or methods, or can modify the inherited ones
- *Software reuse* is at the heart of inheritance

Deriving Subclasses

- In Java, we use the reserved word `extends` to establish an inheritance relationship

```
class Dictionary extends Book {

      // class contents
}
```

```
public class Book {

    protected int pages = 1500;

    public String message() {
        System.out.println("Number of pages: " + pages);
    }
}

public class Dictionary extends Book {

    private int definitions = 52500;

    public void defMessage() {
        System.out.println("Number of definitions" +
                           definitions);
        System.out.println("Definitions per page: " +
                           (definitions/pages));
    }
}

Dictionary webster = new Dictionary();

webster.message();

webster.defMessage();
```

Number of pages: 1500

Number of definitions: 52500

Definitions per page: 35

Some Inheritance Details

- An instance of a child class does not rely on an instance of a parent class
  - Hence we could create a Dictionary object without having to create a Book object first

- Inheritance is a one-way street
  - The Book class cannot use variables or methods declared explicitly in the Dictionary class

The protected Modifier

- Visibility modifiers determine which class members are inherited and which are not
- Variables and methods declared with `public` visibility are inherited; those with `private` visibility are not
- But `public` variables violate the principle of encapsulation
- There is a third visibility modifier that helps in inheritance situations: `protected`

The protected Modifier

- The `protected` modifier allows a member of a base class to be inherited into a child
- Protected visibility provides
  - more encapsulation than public visibility does
  - the best possible encapsulation that permits inheritance

Using super

- Whenever a subclass needs to refer to its immediate superclass, it can do so by use of the keyword **super**
- **super** has two general forms:
  1. The first calls the superclass' constructor
  2. The second is used to access a member of the superclass that has been hidden by a member of a subclass

```
public class Student {
    protected String name;
    protected int roll;
    private String registrationNumber;
    public Student(String name, int roll, String
    registrationNumber) {
        this.name = name;
        this.roll = roll;
        this.registrationNumber = registrationNumber;
    }
    void submitAssignment() {
    System.out.println("roll " + this.roll + " submitted
    assignment");
    }
    void showIDCard() {
        System.out.println("name: " + name + "\nregistrationNumber:
        " + registrationNumber);
    }
}
```

```
public class GradStudent extends Student {
    int meritPosition;
    public GradStudent(String name, int roll, String
    registrationNumber, int meritPosition) {
        super(name, roll, registrationNumber);
        this.meritPosition = meritPosition;
    }
    void submitThesis() {
        System.out.println("roll " + this.roll + " submitted
        thesis");
    }
}
```

super( ) must always be the first statement executed inside a subclass constructor

Constructors of Subclasses

- Can invoke a constructor of the direct superclass.
  - super(…) must be the first statement.
  - If the super constructor call is missing, by default the no-arg super() is invoked implicitly.
- Can also invoke another constructor of the same class.
  - this(…) must be the first statement.

Example of “this” Calls

```
public class Point {
  private int x, y;

  public Point(int x, int y) {
    this.x = x;
    this.y = y;
  }

  public Point() { // default constructor

     this(0,0);

  }
}
```

Example of “super” Calls

```
public class ColoredPoint extends Point {
  private Color color;

  public ColoredPoint(int x, int y, Color color) {

     super(x,y);

    this.color = color;
  }

  public ColoredPoint(int x, int y) {
    this(x, y, Color.BLACK); // point with default value
  }

  public ColoredPoint() {
    color = Color.BLACK;
  }
}
```

Default Constructor

- If no constructor is defined, the following form of no-arg default constructor is automatically generated by the compiler.

```
public ClassName() {
  super();
}
```

The Execution Order of Constructors

```
class A {
   A() {System.out.println("Inside A's constructor.");}
}
class B extends A {
   B() {System.out.println("Inside B's constructor.");}
}
class C extends B {
   C() {System.out.println("Inside C's constructor.");}
}
class CallingCons {
   public static void main(String args[]) {
   C c = new C();
   }
}
```

Accessing a Member of the Superclass Hidden by the Subclass

```
class A {
    int i;
}
class B extends A {
    int i;
    B(int a, int b) {
    super.i = a;
    i = b;
    }
    void show() {
       System.out.println("i in superclass: " + super.i);
       System.out.println("i in subclass: " + i);
    }
}
```

Suppose,

```
B subOb = new B(1, 2);
subOb.show();
```

Guess the output

Practice

- Creating a Doctor and a Surgeon class by following the inheritance principle

Substitution Property

- A Superclass Variable Can Reference a Subclass Object

```
class Student { … }
class Undergraduate extends Student { … }
class Graduate extends Student { … }

Student s1, s2;
s1 = new Undergradute();
s2 = new Graudate();
```

Overriding Methods

- 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.
  - **Signature:** method's name along with number, type, and order of its parameters
- The new method must have the same signature as the parent's method, but can have a different body
- The type of the object executing the method determines which version of the method is invoked

Overriding Methods (Cont.)

```
public class T {
  public void m() { … }
}

public class S extends T {
  public void m() { … }
}

T t = new T();
S s = new S();
t.m(); // invoke m of class T
s.m(); // invoke m of class S
```

Overriding Methods (Cont.)

- Dynamic dispatch (binding): The method to be invoked is determined at runtime by the runtime type of the object, not by the declared type (static type).

```
class Student {
  public int maxCredits() { return 15; }
  …
}
class GraduateStudent extends Student {
  public int maxCredits() { return 12; }
  …
}

Student s = new GraduateStudent();
// …
s.MaxCredits(); // which maxCredits method is called?
```

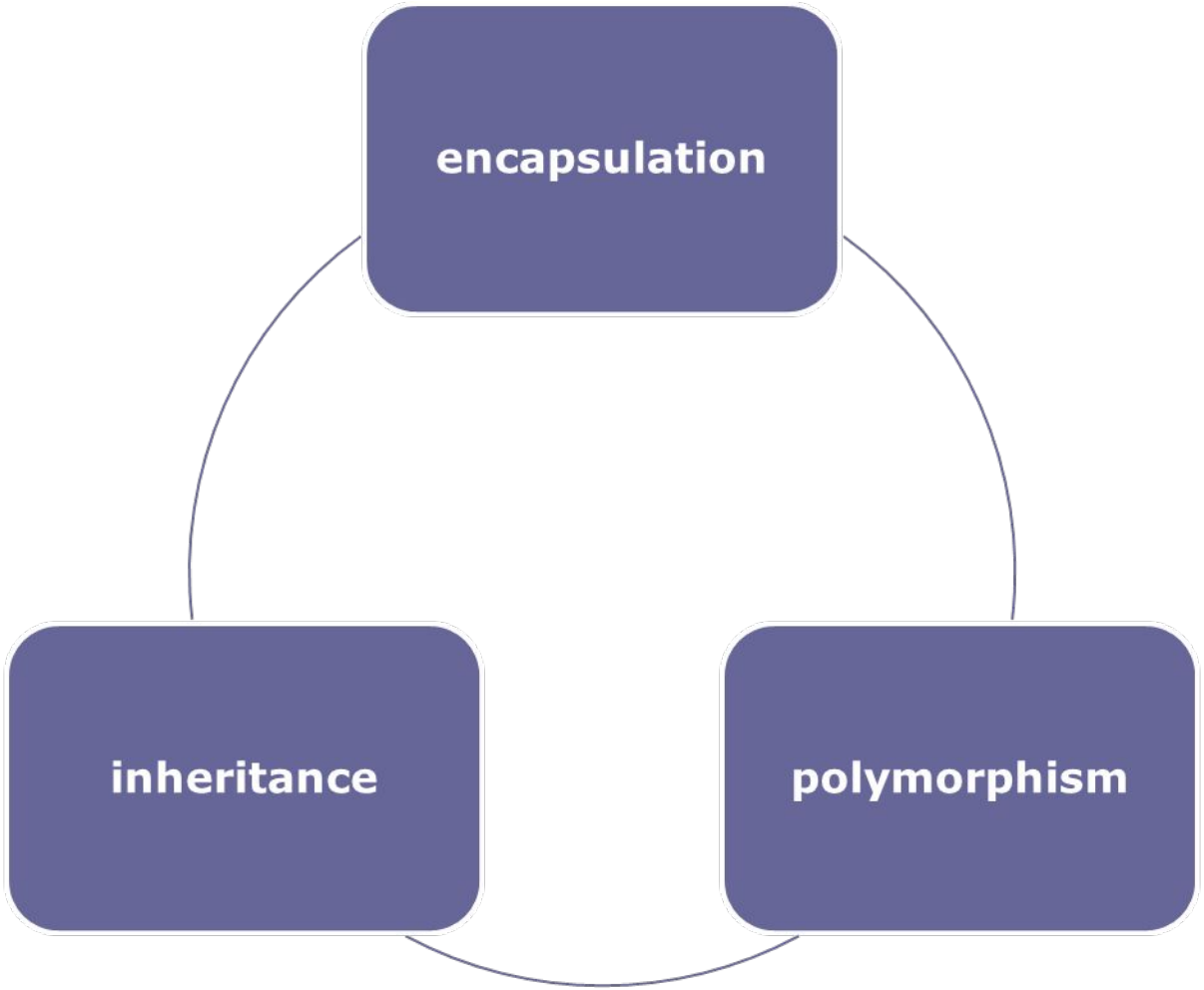
```
class A {
    void callme() {
    System.out.println("Inside A's callme method");
    }
}
class B extends A {
    void callme() {
    System.out.println("Inside B's callme method");
    }
}
class C extends A {
    void callme() {
    System.out.println("Inside C's callme method");
    }
}
```

Guess the output

```
A a = new A();
A r;
r = a;
r.callme();
```

```
public class Book {

    protected int pages;

    Book(int numPages) {
        pages = numPages;
    }

    public void message() {
        System.out.println("Number of pages: " + pages);
    }
}
public class Dictionary extends Book{

    protected int definitions;

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

    public void message() {
        System.out.println("Number of definitions" +
                           definitions);
        System.out.println("Definitions per page: " +
                           (definitions/pages));
        super.message();
    }
}
```

OOP Principles



Polymorphism

- from Greek
- meaning “many forms”

Coal

Graphite

Diamond

Static vs Dynamic Polymorphism

Static Polymorphism

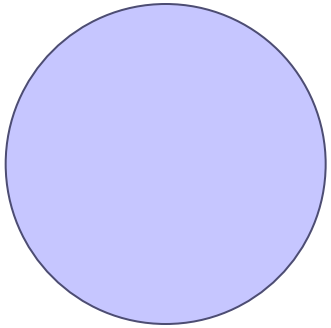
```
void sum (int a , int b);
void sum (float a, double b);
int sum (int a, int b);
//compiler gives error.
```

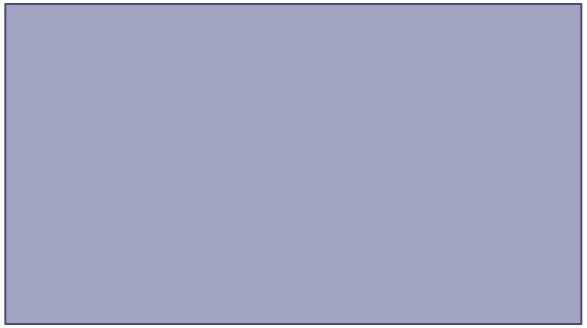
Dynamic Polymorphism

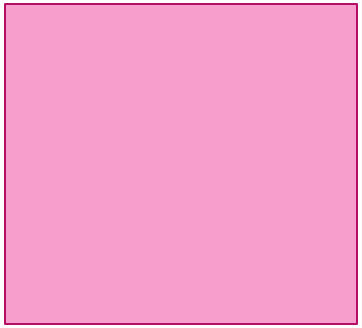
```
//reference of parent pointing to child object
 Doctor obj = new Surgeon();
// method of child called
obj.treatPatient();
```

Practice

- Create three classes named Circle, Rectangle and Square where each of these inherits the Shape class
- The Shape class must have a method called calculateArea()





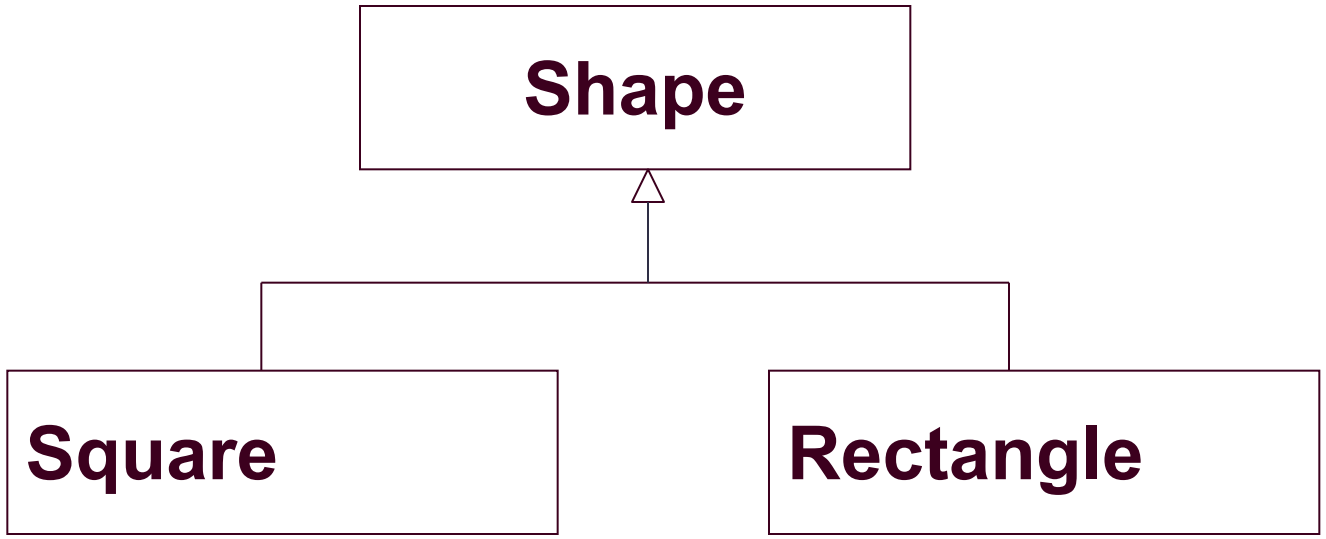


Abstract Classes

- There are situations in which you will want to define a superclass that declares the structure of a given abstraction without providing a complete implementation of every method.
- You may have methods that must be overridden by the subclass in order for the subclass to have any meaning, which are called abstract method.
- Any class that contains one or more abstract methods, is called abstract class.
- An Abstract class cannot be instantiated – objects cannot be created.

Abstract Class - Example

- Shape is a abstract class



Abstract Class Syntax

```
abstract class ClassName
{
        ...
        …
        abstract Type MethodName1();
        …
        …
        Type Method2()
        {
            // method body
        }
}
```

- When a class contains one or more abstract methods, it should be declared as abstract class.
- The abstract methods of an abstract class must be defined in its subclass.
- We cannot declare abstract constructors or abstract static methods.

The Shape Abstract Class

```
public abstract class Shape {
        public abstract double area();
        public void move() { // non-abstract method
          // implementation
      }
}
```

- Is the following statement valid?
  - Shape s = new Shape();
- No. It is illegal because the Shape class is an abstract class, which cannot be instantiated to create its objects.

Rectangle Class

```
public Rectangle extends Shape {
    protected double w, h;
    public Rectangle() { w = 0.0; h=0.0; }
    public double area() { return w * h; }
}
```

Using final to Prevent Overriding

- Methods declared as final cannot be overridden

```
class A {
   final void meth() {
   System.out.println("This is a final method.");
   }
}
class B extends A {
   void meth() { // ERROR! Can't override.
   System.out.println("Illegal!");
   }
}
```

Using final to Prevent Inheritance

- To prevent a class from being inherited, precede the class declaration with final.
- Declaring a class as final implicitly declares all of its methods as final, too.
- It is illegal to declare a class as both abstract and final since an abstract class is incomplete by itself and relies upon its subclasses to provide complete implementations.

```
final class A {
    //...
}
// The following class is illegal.
class B extends A { // ERROR! Can't subclass A
    //...
}
```