Object-Oriented Programming

Inheritance and Polymorphism

HUS
VNU UNIVERSITY OF SCIENCE

Introducing inheritance through creating subclasses

- Improve code reusability
- Allowing overriding to replace the implementation of an inherited method

Outline

Inheritance

- Four fundamental concepts of OOP:
  - Encapsulation
  - Inheritance
  - Polymorphism
  - Abstraction
- Inheritance allows new classes to inherit properties of existing classes
- Main concepts in inheritance
  - Subclassing
  - Overriding

Outline

1. Overriding Methods (revisit) (1/2)

- Recall in previous lectures that a user-defined class automatically inherits some methods – such as toString() and equals() – from the Object class

- The Object class is known as the parent class (or superclass); it specifies some basic behaviours common to all kinds of objects, and hence these behaviours are inherited by all its subclasses (derived classes)

- However, these inherited methods usually don’t work in the subclass as they are not customised

1. Overriding Methods (revisit) (2/2)

- Hence, to make them work, we customised these inherited methods – this is called overriding

MyBall/MyBall.java

```java
// Overriding toString() method
public String toString() {
  return "[" + getColour() + ", " + getRadius() + "]";
}

// Overriding equals() method
public boolean equals(Object obj) {
  if (obj instanceof MyBall) {
    MyBall ball = (MyBall)obj;
    return this.getColour().equals(ball.getColour()) &&
           this.getRadius() == ball.getRadius();
  }

  return false;
}
```

Outline

2. Creating a Subclass (1/6)

- Object-oriented languages allow inheritance
  - Declare a new class based on an existing class
  - So that the new class may inherit all of the attributes and methods from the other class
- Terminology
  - If class *B* is derived from class *A*, then class *B* is called a child (or subclass or derived class) of class *A*
  - Class *A* is called a parent (or superclass) of class *B*

2. Creating a Subclass (2/6)

- Recall the BankAccount class in previous lecture

BankAccount.java

```java
class BankAccount {
  private int accountNumber;
  private double balance;

  public BankAccount() { ... }
  public BankAccount(int number, double aBalance) { ... }

  public int getAccountNumber() { ... }
  public double getBalance() {... }

  public boolean withdraw(double amount) { ... }
  public void deposit(double amount) { ... }

  public void print() { ... }
}
```

2. Creating a Subclass (3/6)

- Let's define a SavingAccount class
- Basic information:
  - Account number, balance
  - Interest rate
- Basic functionality:
  - Withdraw, deposit
  - Pay interest

New requirements

- Compare with the basic bank account:
  - Differences are highlighted above
  - SavingAccount shares more than 50% of the code with BankAccount
- So, should we just cut and paste the code from BankAccount to create SavingAccount?

2. Creating a Subclass (4/6)

- Duplicating code is **undesirable** as it is hard to maintain
  - Need to correct all copies if errors are found
  - Need to update all copies if modifications are required
- Since the classes are logically unrelated if the codes are separated:
  - Code that works on one class cannot work on the other
- Incompatible data types
- Hence, we should create SavingAccount as a subclass of BankAccount

2. Creating a Subclass (5/6)

BankAccount.java

```java
class BankAccount {

  protected int accountNumber;
  protected double balance;

  // Constructors and methods not shown
}

```

The "protected" keyword allows subclass to access the attributes directly

SavingAccount.java

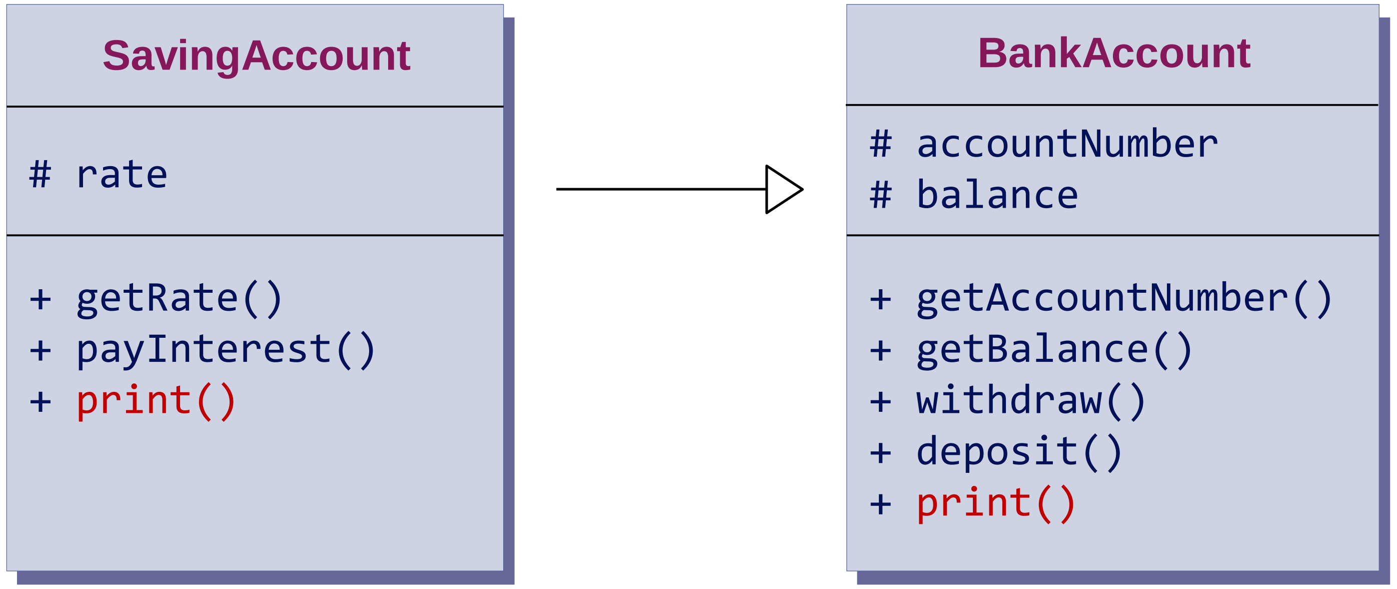
```java
class SavingAccount extends BankAccount {

  // interest rate
  protected double rate;

  public void payInterest() {
    balance += balance * rate;
  }
}

```

The "extends" keyword indicates inheritance

This allows subclass of SavingAccount to access rate. If this is not intended, you may change it to "private".

2. Creating a Subclass (6/6)

- The subclass-superclass relationship is known as an “is-a” relationship, i.e. SavingAccount is-a BankAccount
- In the UML diagram, a solid line with a closed unfilled arrowhead is drawn from SavingAccount to BankAccount
- The symbol # is used to denoted protected member



2.1 Observations

- Inheritance greatly reduces the amount of redundant coding
- In SavingAccount class,
  - No definition of accountNumber and balance
  - No definition of withdraw() and deposit()
- Improve maintainability:
  - Eg: If a method is modified in BankAccount class, no changes are needed in SavingAccount class
- The code in BankAccount remains untouched
  - Other programs that depend on BankAccount are unaffected ← very important!

2.2 Constructors in Subclass

- Unlike normal methods, constructors are NOT inherited
  - You need to define constructor(s) for the subclass

SavingAccount.java

```java
class SavingAccount extends BankAccount {
  protected double rate;  // interest rate

  public SavingAccount(int number, double aBalance, double rate) {
    accountNumber = number;
    balance = aBalance;
    this.rate = rate;
  }

  //......payInterest() method not shown

}
```

2.3 The “super” Keyword

- The “super” keyword allows us to use the methods (including constructors) in the superclass directly
- If you make use of superclass’ constructor, it must be the first statement in the method body

SavingAccount.java

```java
class SavingAccount extends BankAccount {
  protected double rate;  // interest rate

  public SavingAccount(int number, double aBalance, double rate) {
    super(number, aBalance);
    this.rate = rate;
  }

    //......payInterest() method not shown

}
```

Using the constructor in BankAccount class

2.4 Using SavingAccount

TestSavingAccount.java

```java
public class TestSavingAccount {

  public static void main(String[] args) {

    SavingAccount savingAccount = new SavingAccount(2, 1000.0, 0.03);

    savingAccount.print();
    savingAccount.withdraw(50.0);

    savingAccount.payInterest();
    savingAccount.print();
  }
}
```

Inherited method from BankAccount

Method in SavingAccount

How about print()?
Should it be the one in BankAccount class, or should SavingAccount class override it?

2.5 Method Overriding (1/2)

- Sometimes we need to modify the inherited method:
  - To change/extend the functionality
  - As you already know, this is called method overriding

- In the SavingAccount class:
  - The print() method inherited from BankAccount should be modified to include the interest rate in output

- To override an inherited method:
  - Simply recode the method in the subclass using the same method header
  - Method header (prototype) refers to the name and parameters type of the method (also known as method signature)

2.5 Method Overriding (2/2)

SavingAccount.java

```java
class SavingAccount extends BankAccount {
  protected double rate;  // interest rate

  public double getRate() {
    return rate;
  }

  public void payInterest() { ... }

  public void print() {
    System.out.println("Account Number: " + getAccountNumber());
    System.out.printf("Balance: $%.2f\n", getBalance());
    System.out.printf("Interest: %.2f%%\n", getRate());
  }
}
```

The first two lines of code in print() are exactly the same as print() of BankAccount.

Can we reuse BankAccount's print() instead of recoding?

2.6 Using "super" Again

- The super keyword can be used to invoke superclass' method
  - Useful when the inherited method is overridden

SavingAccount.java

```java
class SavingAccount extends BankAccount {
  protected double rate;  // interest rate

  public double getRate() {
    return rate;
  }

  public void payInterest() { ... }

  public void print() {
    super.print();
    System.out.printf("Interest: %.2f%%\n", getRate());
  }
}
```

To use the print() method from BankAccount

Outline

3. Subclass Substitutability (1/2)

- An added advantage for inheritance is that:
  - Whenever a superclass object is expected, a subclass object is acceptable as substitution!
    - **Caution:** the reverse is NOT true (Eg: A cat is an animal; but an animal may not be a cat.)
  - Hence, all existing functions that works with the superclass objects will work on subclass objects with no modification!

- Analogy:
  - We can drive a car
  - Honda is a car (Honda is a subclass of car)
  - We can drive a Honda

3. Subclass Substitutability (2/2)

TestAccountSubclass.java

```java
public class TestAccountSubclass {
  public static void transfer(BankAccount fromAccount,
                              BankAccount toAccount,
                              double amount) {
    fromAccount.withdraw(amount);
    toAccount.deposit(amount);
  }

  public static void main(String[] args) {
    BankAccount bankAccount = new BankAccount(1, 234.56);
    SavingAccount savingAccount = new SavingAccount(2, 1000.0, 0.03);

    transfer(bankAccount, savingAccount, 123.45);

    bankAccount.print();
    savingAccount.print();
  }
}
```

transfer() method can work on the SavingAccount object savingAccount!

4. The “Object” Class

- In Java, all classes are descendants of a predefined class called Object
  - Object class specifies some basic behaviors common to all objects
  - Any methods that works with Object reference will work on object of any class
  - Methods defined in the Object class are inherited in all classes
  - Two inherited Object methods are
    - toString() method
    - equals() method
  - However, these inherited methods usually don’t work because they are not customised

5. “is-a” versus “has-a” (1/2)

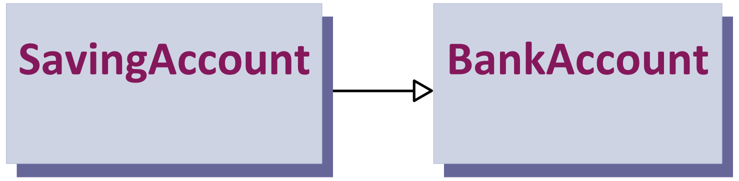
- Words of caution:
  - Do not overuse inheritance
  - Do not overuse protected
    - Make sure it is something inherent for future subclass
- To determine whether it is correct to inherit:
  - Use the “is-a” rules of thumb
    - If “B is-a A” sounds right, then **B is a subclass of A**
  - Frequently confused with the “has-a” rule
    - If “B has-a A” sounds right, then **B should have an A attribute** (hence B depends on A)

5. “is-a” versus “has-a” (2/2)

```
class BankAccount {
   ...
}

class SavingAccount extends BankAccount {
   ...
}
```

Inheritance: SavingAccount IS-A BankAccount

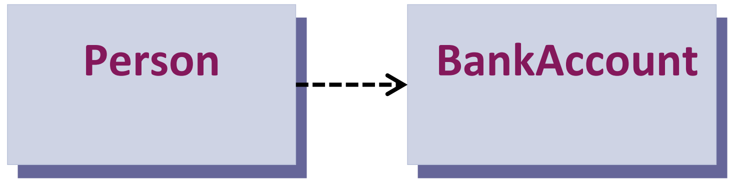
UML diagrams



Solid arrow

```
class BankAccount {
   ...
}

class Person {
  private BankAccount myAccount;
}
```

Attribute: Person HAS-A BankAccount



Dotted arrow

6. Preventing Inheritance ("final")

- Sometimes, we want to prevent inheritance by another class (eg: to prevent a subclass from corrupting the behaviour of its superclass)
- Use the final keyword
  - Eg: final class SavingAccount will prevent a subclass to be created from SavingAccount
- Sometimes, we want a class to be inheritable, but want to prevent some of its methods to be overridden by its subclass
  - Use the final keyword on the particular method:
-

```
public final void payInterest() { … }
```

will prevent the subclass of SavingAccount from overriding `payInterest()`

7. Constraint of Inheritance in Java

- Single inheritance: Subclass can only have a single superclass
- Multiple inheritance: Subclass may have more than one superclass
- In Java, only single inheritance is allowed
- (Side note: Java’s alternative to multiple inheritance can be achieved through the use of interfaces – to be covered later. A Java class may implement multiple interfaces.)

8. Quick Quiz #1

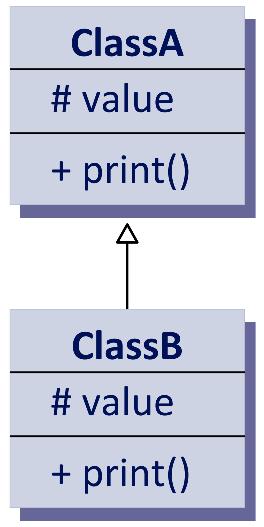
8. Quick Quiz #1 (1/2)

ClassA.java

```java
class ClassA {
  protected int value;

  public ClassA() {  }
  public ClassA(int val) { value = val; }
  public void print() {
    System.out.println("Class A: value = " + value);
  }
}
```

ClassB.java

```java
class ClassB extends ClassA {
  protected int value;

  public ClassB() {  }
  public ClassB(int val) {
    super.value = val - 1;
    value = val;
  }
  public void print() {
    super.print();
    System.out.println("Class B: value = " + value);
  }
}
```



8. Quick Quiz #1

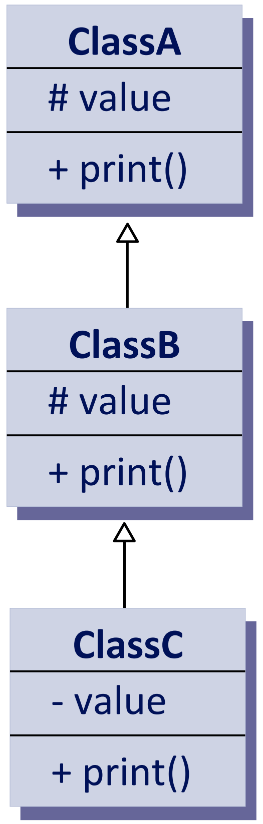
8. Quick Quiz #1 (2/2)

ClassC.java

```java
final class ClassC extends ClassB {
  private int value;
  public ClassC() {  }
  public ClassC(int val) {
    super.value = val - 1;
    value = val;
  }
  public void print() {
    super.print();
    System.out.println("Class C: value = " + value);
  }
}
```

TestSubclasses.java

```java
public class TestSubclasses {
  public static void main(String[] args) {
    ClassA objA = new ClassA(123);
    ClassB objB = new ClassB(456);
    ClassC objC = new ClassC(789);

    objA.print(); System.out.println("---------");
    objB.print(); System.out.println("---------");
    objC.print();
  }
}
```



8. Quick Quiz #2 (1/2)

- Assume all methods print out message of the form

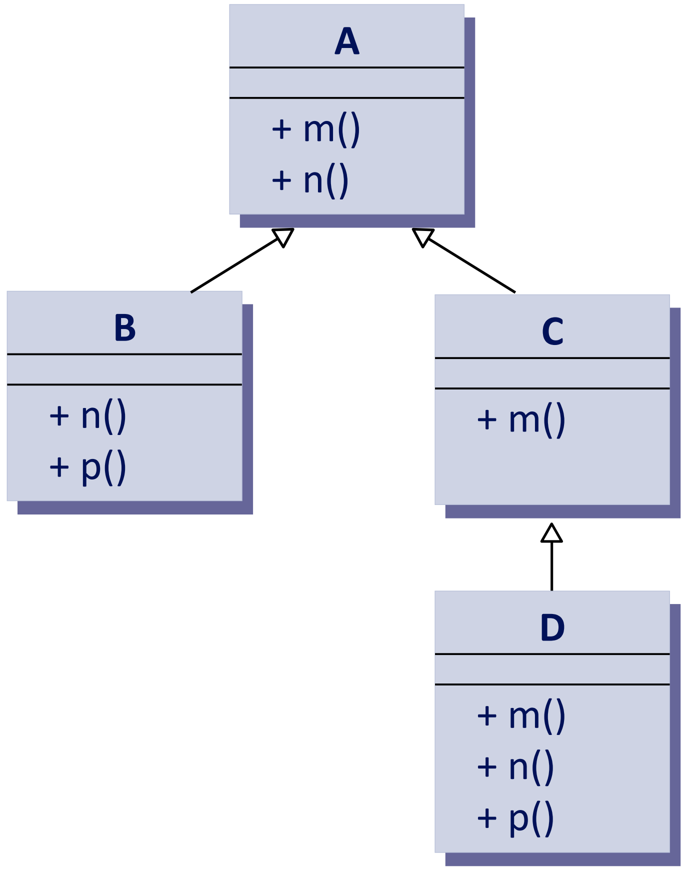
  <class name>. <method name>

- Eg: method m() in class A prints out "A.m".
- If a class overrides an inherited method, the method's name will appear in the class icon. Otherwise, the inherited method remains unchanged in the subclass.
- For each code fragment below, indicate whether:
  - The code will cause compilation error, and briefly explain; or
  - The code can compile and run. Supply the execution result.

Code fragment (example)	Compilation error? Why?	Execution result

A a = new A();
a.m();
		A.m

A a = new A();
a.k();
	Method k() not defined in class A	



8. Quick Quiz #2

8. Quick Quiz #2 (2/2)

Code fragment	Compilation error?	Execution result

A a = new C();
a.m();		

B b = new A();
b.n();		

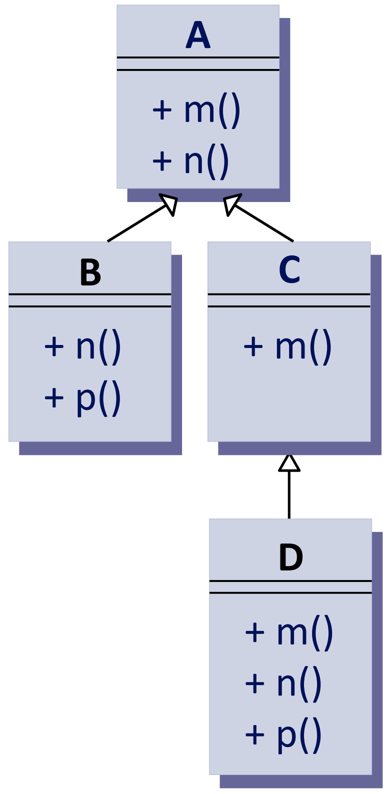
A a = new B();
a.m();		

A a;
C c = new D();
a = c;
a.n();		

B b = new D();
b.p();		

C c = new C();
c.n();		

A a = new D();
a.p();		



Summary

- Inheritance:
  - Creating subclasses
  - Overriding methods
  - Using “super” keyword
  - The “Object” class

Polymorphism

Polymorphism

- When a class inherits from another class it inherits both the **states** and **methods** of that class
- In the case of the Car class inheriting from the Vehicle class, the Car class inherits the methods of the Vehicle class, such as engineStart(), gearChange(), lightsOn(), etc.
- The Car class will also inherit the **states** of the Vehicle class, such as isEngineOn, isLightsOn, numberWheels, etc.

Polymorphism

- **Polymorphism** means "multiple forms". It comes from Greek word "poly" (means many) and "morphos" (means form).
- In OOP these multiple forms refer to multiple forms of the same method, where the exact same method name can be used in **different classes**, or the same method name can be used in the same class with slightly **different paramaters**.
- There are two forms of polymorphism, **overriding** and **overloading**.

Overloading

- Overloading is the second form of polymorphism. The **same method name** can be used, but the number of parameters or the types of parameters can differ, allowing the correct method to be chosen by the compiler.
- For example
  - Two different methods that have the same name and the same number of parameters. However, when we pass two String objects instead of two int variables then we expect different functionality.

Overloading: the types of parameters differ

```
add(int x, int y);
add(String x, String y);
```

  - The number of arguments can also determine which method should be run. The first method may simply display the current channel number, but the second method will set the channel number to the number passed.

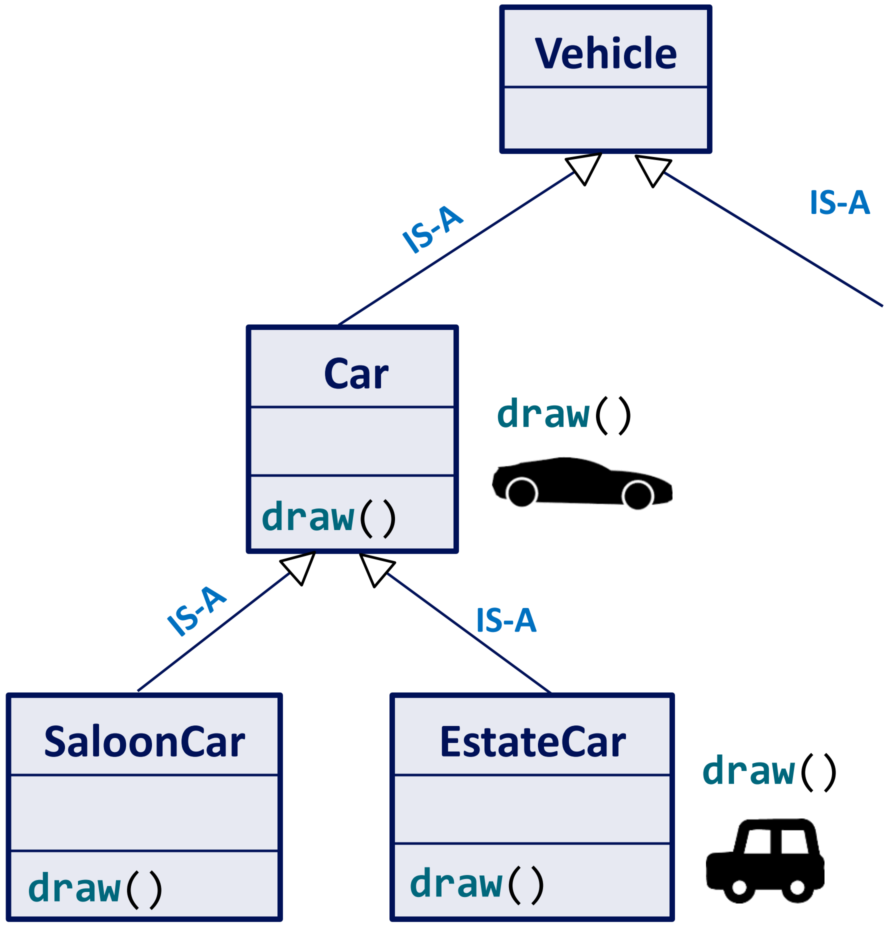
Overloading: number of parameters differ

```
channel();
channel(int x);
```

- A derived class inherits its methods from the base class. It may be necessary to redefine an inherited method to provide specific behaviour for a derived class - and so alter the implementation.
- **Overriding** is the term used to describe the situation where the same method name is called on two different objects and each object responds differently.
- Overriding allows different kinds of objects that share a common behaviour to be used in code that only requires that common behaviour.

Overriding

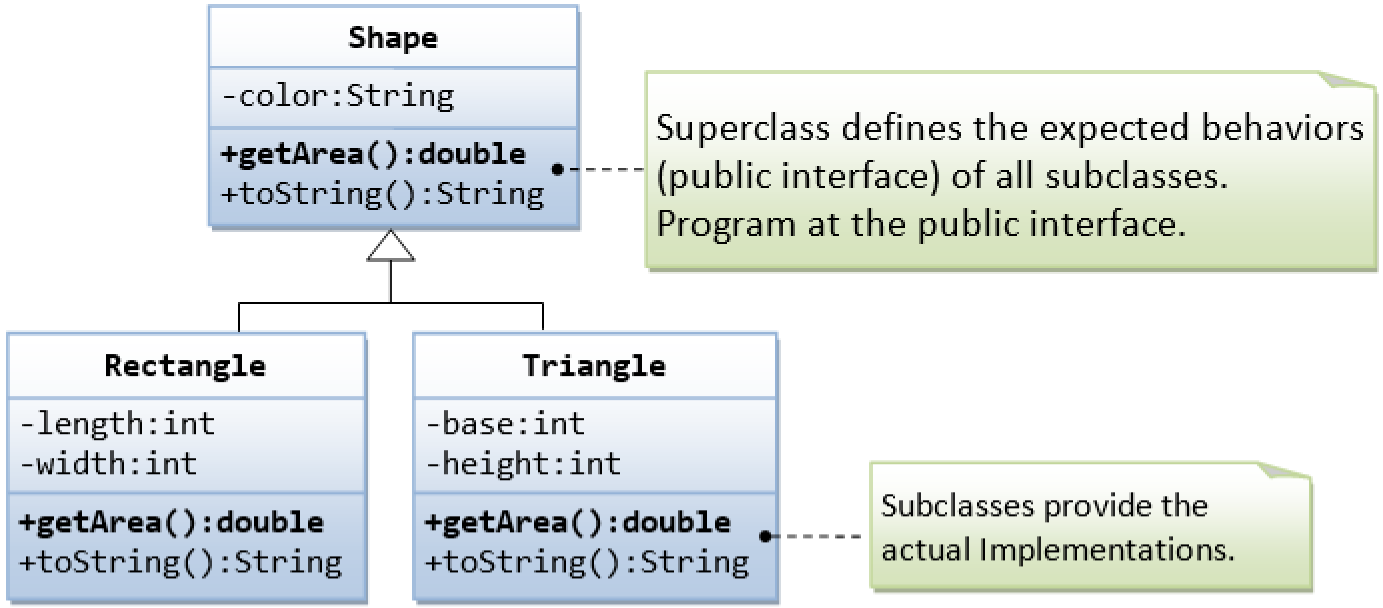
- **Car** inherits from **Vehicle** and from this class **Car** there are further derived classes **SaloonCar** and **EstateCar**.
- If a draw() method is added to the **Car** class, that is required to draw a picture of a generic vehicle. This method will not adequately draw an estate car, or other child classes.
- Overriding allows us to write a specialised draw() method for the EstateCar class - There is no need to write a new draw() method for the **SaloonCar** class as the **Car** class provides a suitable enough draw() method.
- All we have to do is write a new draw() method in the **EstateCar** class with the exact same method name.
- So, Overriding allows:
  - A more straightforward API where we can call methods the same name, even thought these methods have slightly different functionality.
  - A better level of abstraction, in that the implementation mechanics remain hidden.



The overridden draw() method.

Polymorphism example

- Suppose that our program uses many kinds of shapes, such as triangle, rectangle and so on.
- We should design a superclass called Shape, which defines the public interfaces (or behaviors) of all the shapes.
- For example, we would like all the shapes to have a method called getArea(), which returns the area of that particular shape.



Polymorphism example

Shape.java

```java
public class Shape {
  // Private member variable
  private String color;

  /** Constructs a Shape instance with the given color */
  public Shape(String color) {
    this.color = color;
  }

  /** Returns a self-descriptive string */
  @Override
  public String toString() {
    return "Shape[color = " + color + "]";
  }

  /** All shapes must provide a method called getArea() */
  public double getArea() {
    // We have a problem here!
    // We need to return some value to compile the program.
    System.err.println("Shape unknown! Cannot compute area!");
    return 0;
  }
}
```

Polymorphism example

Rectangle.java

```java
public class Rectangle extends Shape {
  // Private member variables
  private int length;
  private int width;

  // Constructs a Rectangle instance with the given color,
  // length and width
  public Rectangle(String color, int length, int width) {
    super(color);
    this.length = length;
    this.width = width;
  }

  // Returns a self-descriptive string
  @Override
  public String toString() {
    return "Rectangle[length = " + length + ", width = "
            + width + "," + super.toString() + "]";
  }

  // Override the inherited getArea() to provide the proper
  // implementation for rectangle
  @Override
  public double getArea() {
    return length * width;
  }
}
```

Triangle.java

```java
public class Triangle extends Shape {
  // Private member variables
  private int base;
  private int  height;

  // Constructs a Triangle instance with the given color,
  // base and height
  public Triangle(String color, int base, int height) {
    super(color);
    this.base = base;
    this.height = height;
  }

  // Returns a self-descriptive string
  @Override
  public String toString() {
    return "Triangle[base = " + base + ", height = "
            + height + ",  " + super.toString() + "]";
  }

  // Override the inherited getArea() to provide the proper
  // implementation for triangle
  @Override
  public double getArea() {
    return 0.5 * base * height;
  }
}
```

Polymorphism example

TestShape.java

```java
/**
 * A test driver for Shape and its subclasses
 */
public class TestShape {
  public static void main(String[] args) {
    Shape shape1 = new Shape(“yellow");
    System.out.println(shape1); // Run Shape's toString()
    // Shape[color = red]

    Shape shape2 = new Rectangle("red", 4, 5); // Upcast
    System.out.println(shape2); // Run Rectangle's toString()
    // Rectangle[length = 4, width = 5, Shape[color = red]]
    System.out.println("Area is " + shape2.getArea()); // Run Rectangle's getArea()
    // Area is 20.0

    Shape shape3 = new Triangle("blue", 4, 5); // Upcast
    System.out.println(shape3); // Run Triangle's toString()
    // Triangle[base = 4, height = 5, Shape[color = blue]]
    System.out.println("Area is " + shape3.getArea()); // Run Triangle's getArea()
    // Area is 10.0
  }
}
```

Thank you!

HUS
VNU UNIVERSITY OF SCIENCE