COMPSCI 121: ABSTRACT CLASSES

SPRING 2020

BYTE FROM COMPUTING HISTORY

See the fascinating Timeline of Computer History.

INHERITANCE REVIEW

sub-class

When class **B** `extends` **A** {   super class

Class **B** is **subclass/derived** class, **specialized** class of **A**

1. **Super class** instance variables if **private**: can't use directly. If **protected** **can use directly**:
2. **Super class** **constructor** can do some of the work of B's constructor.
3. **Super class** methods, if **public** **or** **protected**, available directly in B.
4. **Sub-class** methods, when overriding A's methods, can use **super.method**() to do some of the work of A's methods.

INHERITANCE GUIDELINES

In **Sub-classes** you can:

1. declare **new fields (instance variables)** in the subclass that are **not** in the superclass.
2. write a **new instance method** in the subclass that has the same signature as the one in the superclass, thus **overriding** it.
3. declare new methods in the subclass that are **not** in the superclass.

REVIEW: ACCESS BY MEMBERS OF DERIVED CLASSES

Members of sub-class have access to public members of the super class NOT to private members.

SPECIFIER	DESCRIPTION
private	Accessible by self ("this").
protected	Accessible by self, derived classes, and other classes in the same package.
public	Accessible by self, derived classes, and everyone else.
~~no spec~~	~~Accessible by self and other classes in the same~~

Best Practice: NEVER leave this unspecified!!

GOALS FOR TODAY'S CLASS

Continue with Object-oriented programming (OOP) - powerful programming paradigm.

You know:

- A *class* provides data/behaviors for objects.
- *Inheritance* creates a subclass that has its own specific behaviors relative to a superclass.

TODAY! **Abstract Classes**

PROBLEM 1

The federal bank decides that all banks must provide their rate of interest to the public. Imagine building an app that collects bank info to help you determine the best interest rate. The code design:

Superclass: `Bank` Generic Bank

Subclasses: `CitiBank, BankOfAmerica, etc.` Actual Banks

- all must have method: `getRateOfInterest()`

Q. How do you *ensure* that all subclasses have implemented this method in the program?

SOLUTION 1

We could use *inheritance* - the superclass has a `getRateOfInterest` method that the subclass can override to set its own rate of interest.

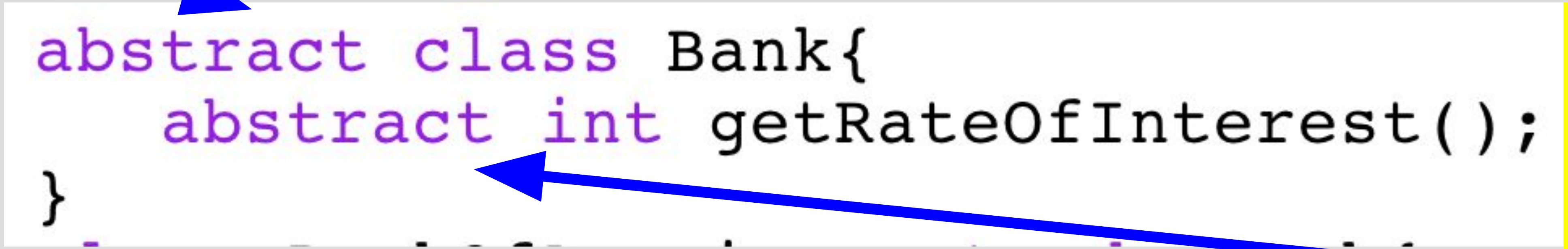
Problem: The subclass is ***not bound*** to override the superclass method. That means we could have subclasses that do not behave correctly.

Q. How do we *enforce* the implementation of the method in all subclass?

SOLUTION 2

We use an abstract class to specify how the subclass must be implemented. A sub-class that extends it MUST implement the abstract method!

Abstract class



```
abstract class Bank{
    abstract int getRateOfInterest();
}
```

Abstract method that subclass must implement. NOTE: method has *no* body!

A program cannot use the `new` operator to create an instance of an abstract class!

EXAMPLE 1: CLASSES THAT CAN BE INSTANTIATED

```
class BankOfAmerica extends Bank{
      int getRateOfInterest(){return 7;}}

class CitiBank extends Bank{
      int getRateOfInterest(){return 8;}}

class TestBank{
   public static void main(String args[]){
      Bank boa = new BankOfAmerica();
      Bank cb = new CitiBank();
      System.out.println("BoA Rate of Interest is:
       "+boa.getRateOfInterest()+" %");
     System.out.println("CB Rate of Interest is:
       "+cb.getRateOfInterest()+" %"); }}
```

Concrete versions. Sub-classes *extend* super class and implement specified method.

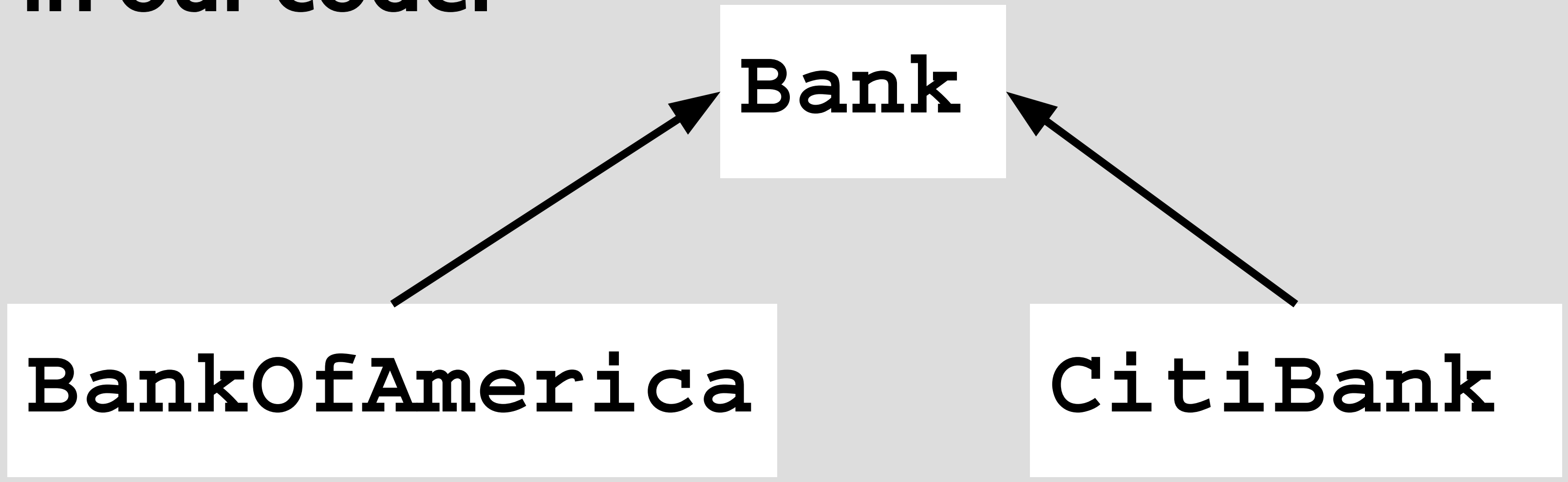
Method calls to get interest.

ABSTRACT CLASSES & METHODS - RULES!

1. Any class with an abstract method must be abstract.
2. An abstract class *can* also contain non-abstract methods and variables that are shared by subclasses.
3. An abstract method is denoted by the abstract keyword in front of the method signature.
4. If a subclass does not implement an abstract method, then the sub-class must also be defined as abstract.

5. An abstract class can not be instantiated.

In the Bank example this is good because we do not want an instance of Bank in our code. We only want instances of its subclasses in our code.



USE CASES FOR ABSTRACT CLASSES

1. We want to encapsulate some common functionality and data in one place (code reuse) that multiple, related subclasses will share.
2. We need to partially define an API that our subclasses can easily extend and refine.
3. We want to ensure that a superclass is never instantiated (because it is intended only as a template for subclasses).

USE CASES FOR ABSTRACT CLASSES

Another way to look at abstract classes is that you use them to provide a partial implementation of a solution that subclasses can finish in their own manner.

As a Template:

The abstract class defines the methods that subclasses must implement to finish the algorithm.

Clicker Question #1

```
abstract class AClass{
 public abstract void firstMethod();
 public void secondMethod(){
   System.out.println("Second 
Method");
 }}
```

Which one of the following is valid?

A. `AClass myABC = new AClass();`

B. `AClass` enforces that a subclass implements `firstMethod`.

C. `AClass.secondMethod();`

D. `AClass` ensures that subclasses are also abstract.

Clicker Question #1 ANSWER

```
abstract class AClass{
 public abstract void firstMethod();
 public void secondMethod(){
   System.out.println("Second Method");
 }
}
```

Which one of the following is valid?

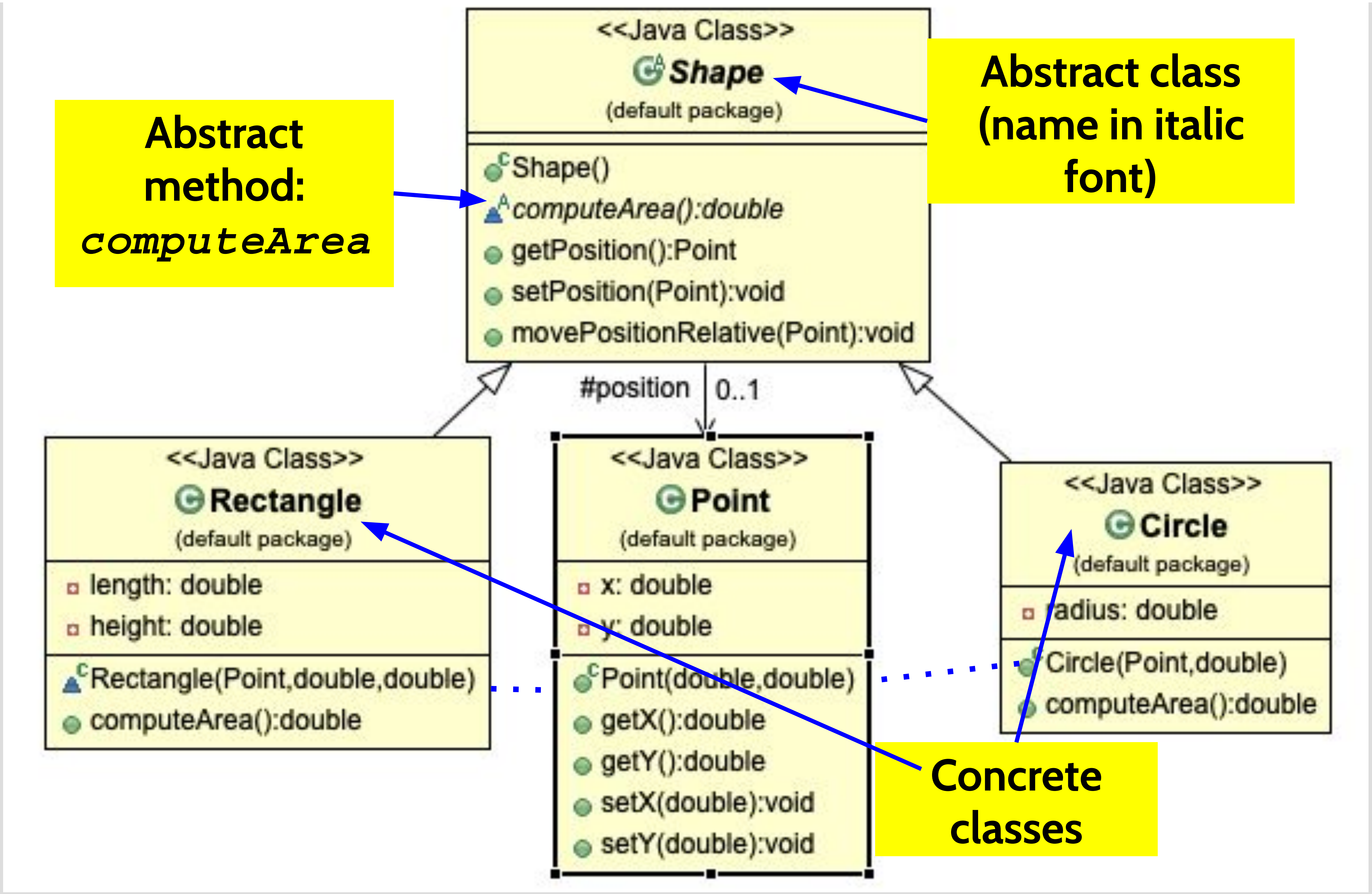
A. `AClass myABC = new AClass();` cannot instantiate

B. `AClass` enforces that a subclass implements `firstMethod`.

C. `AClass.secondMethod();` cannot call method

D. `AClass` ensures that subclasses are also abstract. sub-class need not be abstract

EXAMPLE 2: UML FOR SHAPE PROJECT FROM zyBOOKS(see the code 11.2)

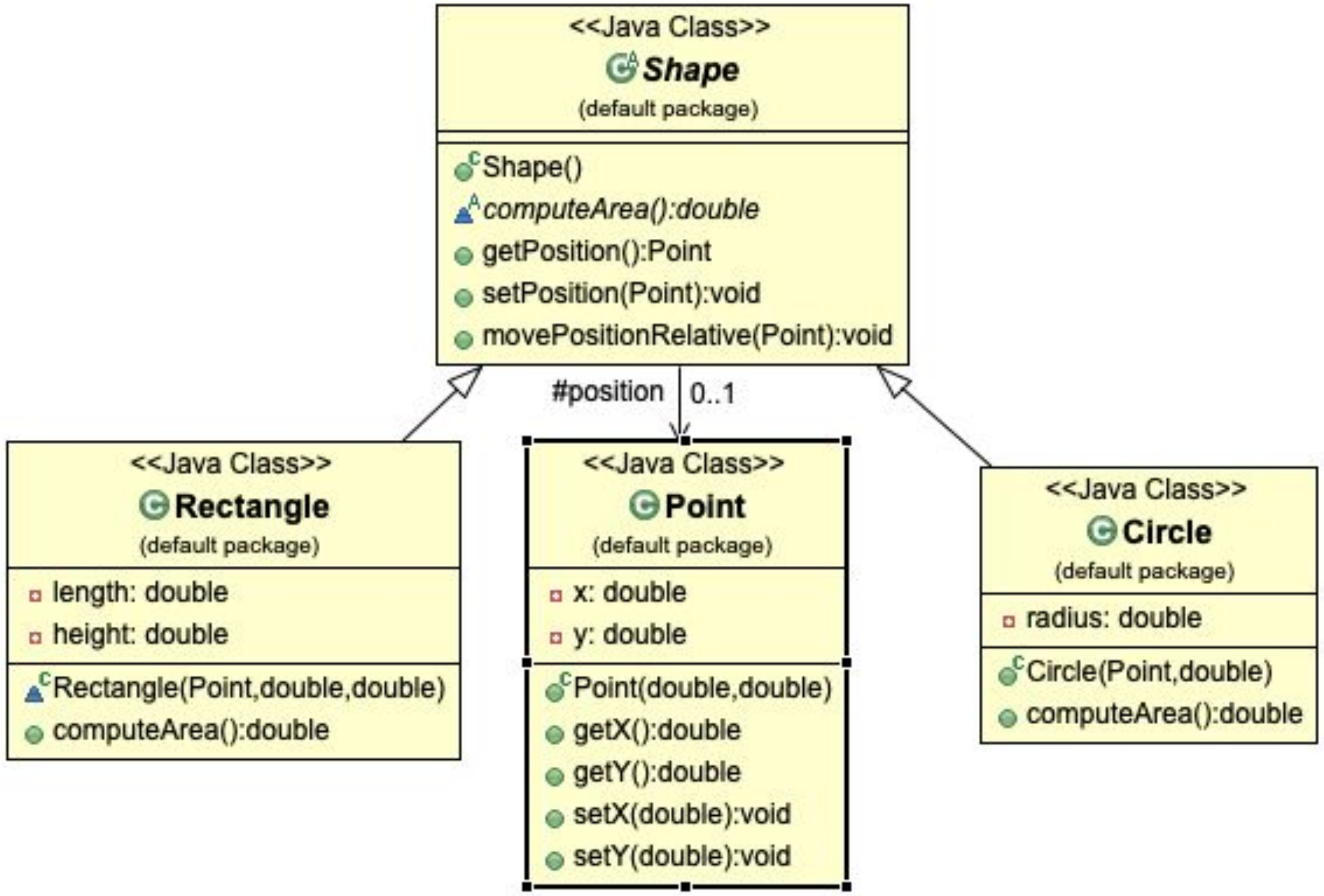


Clicker Question #2

```
Shape shape1 = new Circle(new Point(0.0, 0.0), 1.0);
Shape shape2 = new Shape();
Shape rectangle = new Rectangle(new Point(0.0, 1.0), 1.0, 1.0);
Shape shape3 = new Rectangle(new Point(0.0, 0.0), 2.0, 2.0);
```

Which of the classes cannot be instantiated using the `new` operator?

A. Circle
B. Shape
C. Rectangle
D. Point



Clicker Question #2 ANSWER

```
Shape shape1 = new Circle(new Point(0.0, 0.0), 1.0);
Shape shape2 = new Shape();
Shape rectangle = new Rectangle(new Point(0.0, 1.0), 1.0, 1.0);
Shape shape3 = new Rectangle(new Point(0.0, 0.0), 2.0, 2.0);
```

A. Circle
B. Shape Abstract class cannot be instantiated.
C. Rectangle
D. Point

Which of the classes cannot be instantiated using the `new` operator?

POLYMORPHISM SUMMARY

Recall: Polymorphism is not just about inheritance.

- **Compile-time Polymorphism** - compiler decides which of several identically-named methods to call based on the method's arguments.

  Example: method *overloading*.

- **Runtime polymorphism** - the determination of which method to call is made while the program is running.

  Example: subclasses *overriding* superclass method.

POLYMORPHISM EXAMPLE USING ARRAYLIST

Consider: In the ShapeProject code, you have a list of Shape objects. Shape is an abstract superclass. All of the objects in the list are (concrete) subclasses of Shape. You call the computeArea() method on all objects in the list.

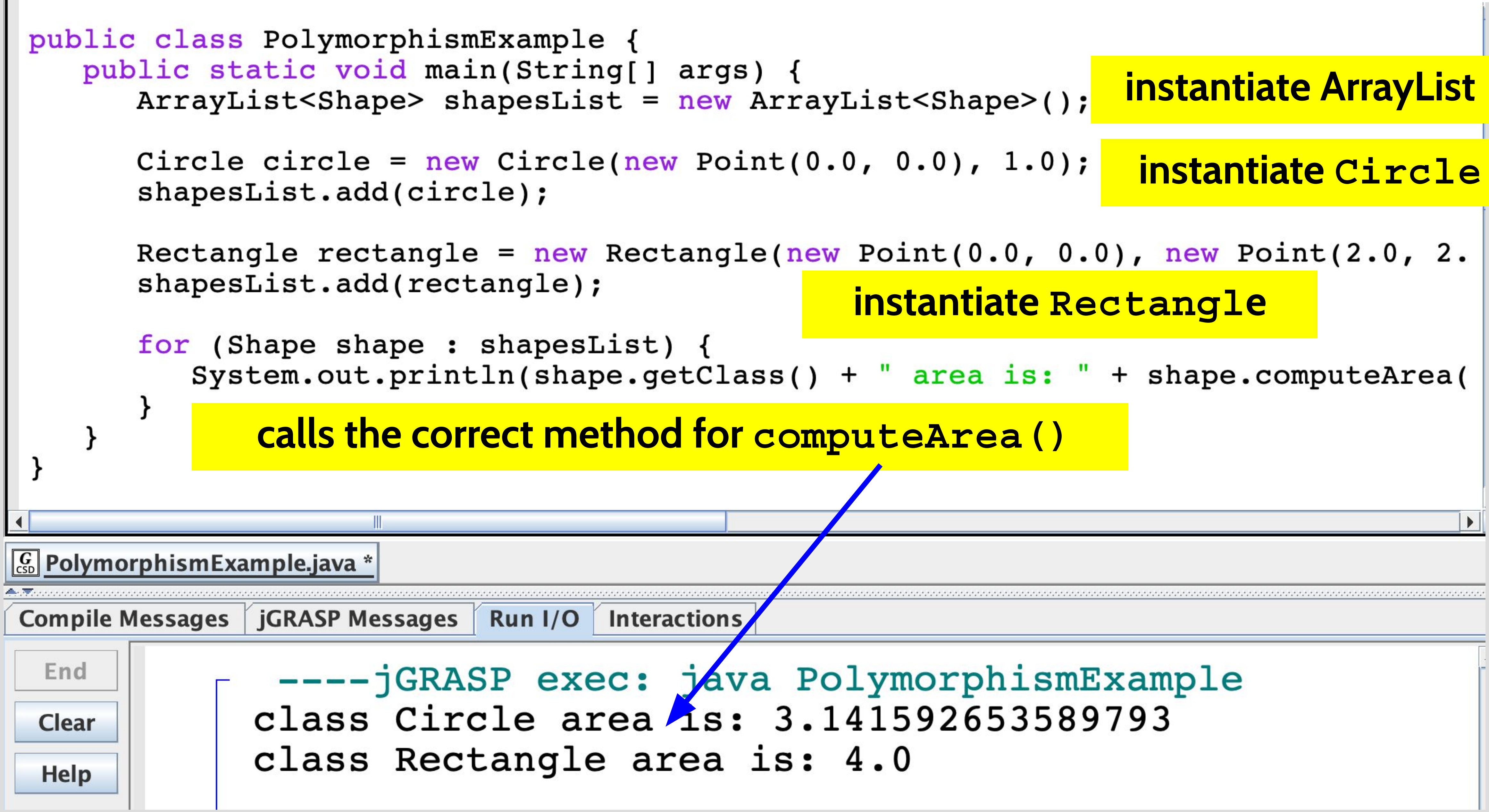
Problem:

When the program executes, how does the JVM automatically call the method of the concrete subclass?

Solution:

Runtime polymorphism calls the concrete class that implements the abstract method.

DEMO: POLYMORPHISM EXAMPLE USING ARRAYLIST



```
public class PolymorphismExample {
    public static void main(String[] args) {
        ArrayList<Shape> shapesList = new ArrayList<Shape>();

        Circle circle = new Circle(new Point(0.0, 0.0), 1.0);
        shapesList.add(circle);

        Rectangle rectangle = new Rectangle(new Point(0.0, 0.0), new Point(2.0, 2.
        shapesList.add(rectangle);

        for (Shape shape : shapesList) {
            System.out.println(shape.getClass() + " area is: " + shape.computeArea(
        }
    }
}
```

instantiate ArrayList

instantiate `Circle`

instantiate `Rectangle`

calls the correct method for `computeArea()`

```
 ----jGRASP exec: java PolymorphismExample
class Circle area is: 3.141592653589793
class Rectangle area is: 4.0
```

SUMMARY – ABSTRACT CLASSES

https://docs.oracle.com/javase/tutorial/java/IandI/abstract.html

- An abstract class is declared abstract— may or may not include abstract methods.
- Abstract classes cannot be instantiated, but can be subclassed.
- An abstract method is declared without an implementation (without braces, and followed by a semicolon).
- If a class includes abstract methods, then the class itself must be declared abstract.
- When an abstract class is subclassed, the subclass provides implementations for the abstract methods in its parent class. If not, then the subclass must also be declared abstract.