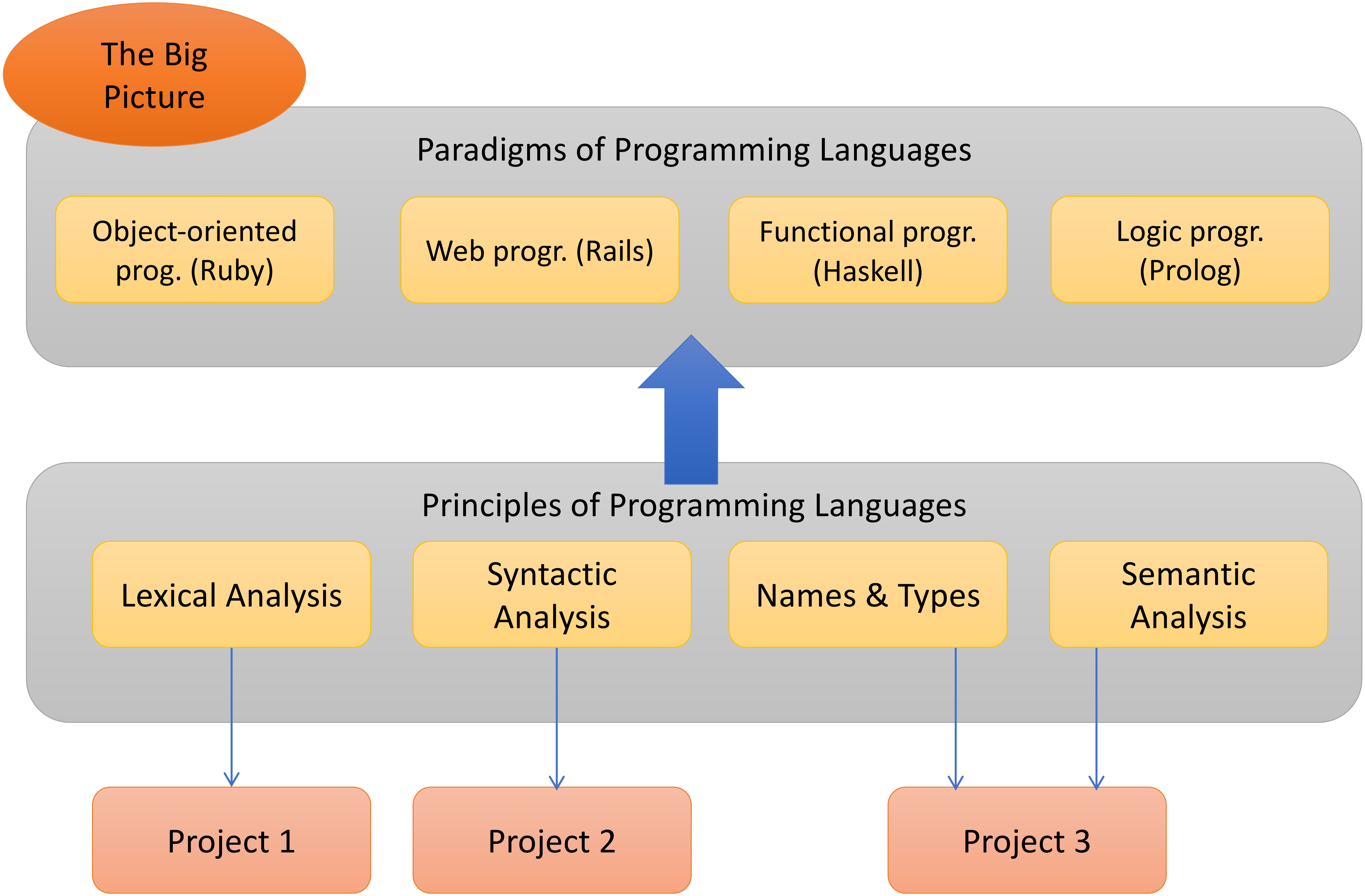
CSCI 2320: Principles of Programming Languages

Object-Oriented Programming (OOP)

Reading: Ch 13

Mohammad T. Irfan



The Big Picture

Paradigms of Programming Languages

Object-oriented prog. (Ruby)

Web progr. (Rails)

Functional progr. (Haskell)

Logic progr. (Prolog)

Principles of Programming Languages

Lexical Analysis

Syntactic Analysis

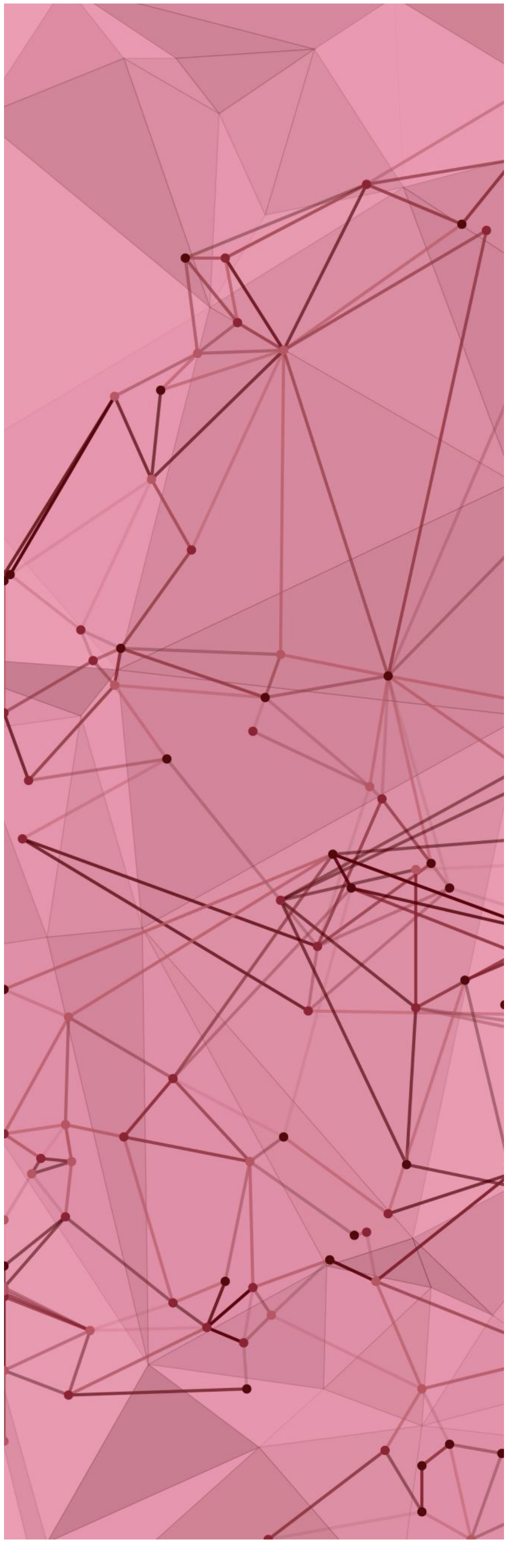
Names & Types

Semantic Analysis

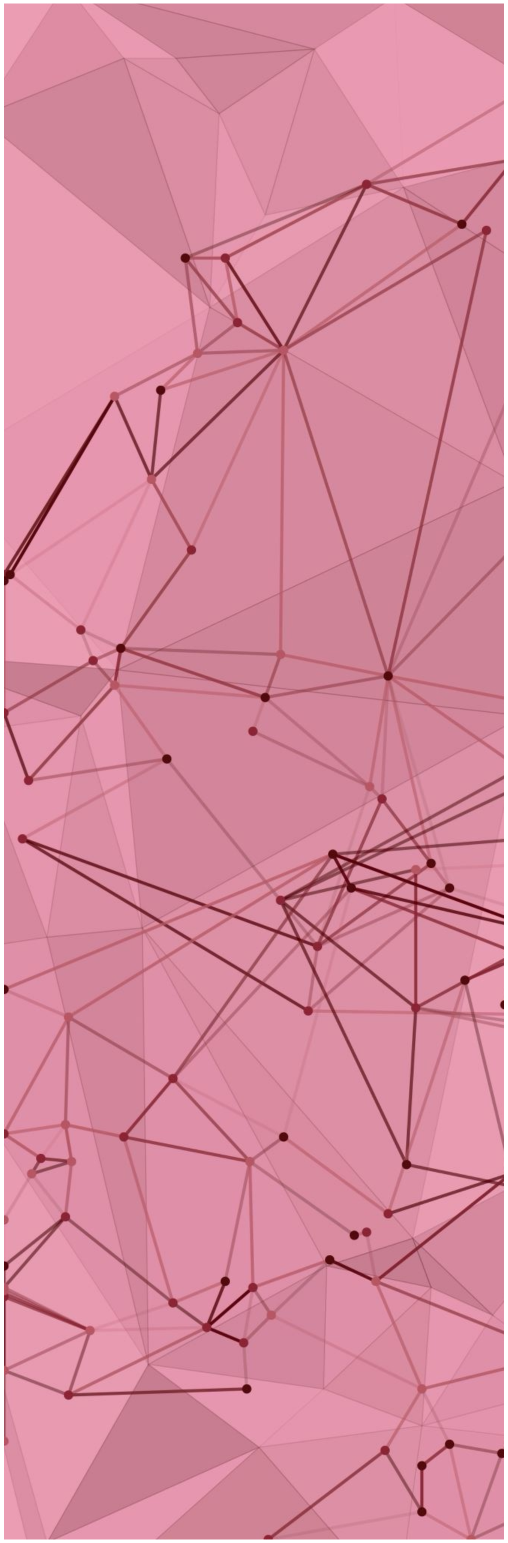
Project 1

Project 2

Project 3



Imperative vs. object-oriented paradigms



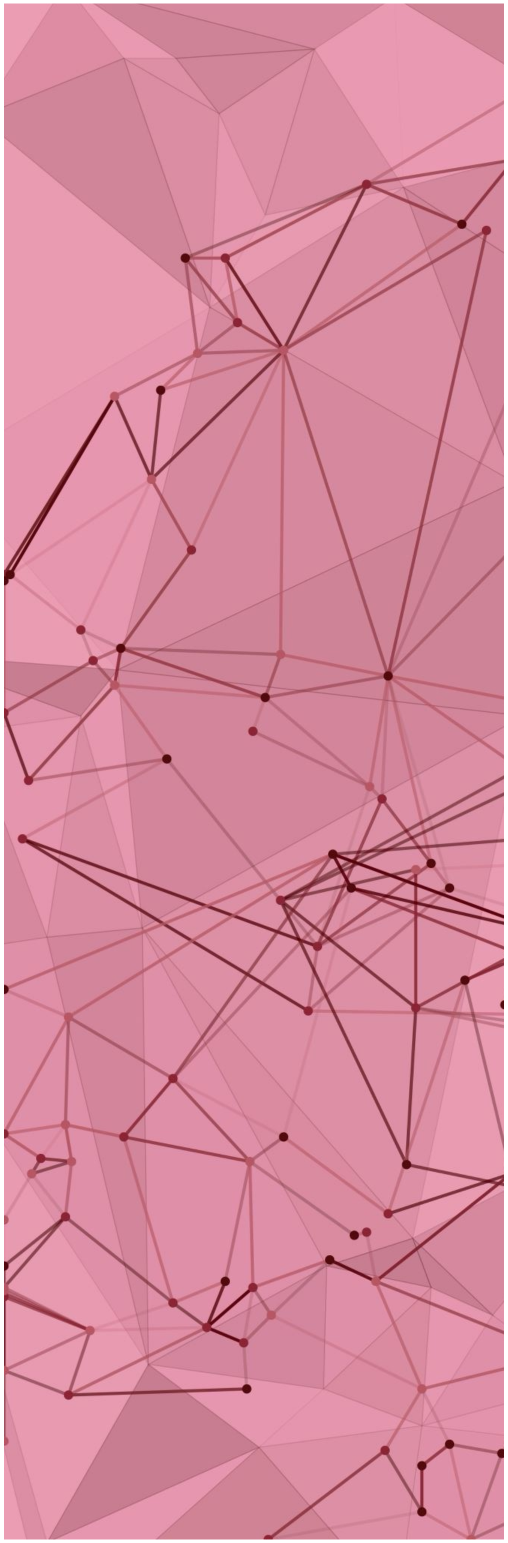
OOP Principles

Examples: Java

OOP principles

1. Inheritance
2. Polymorphism
3. Encapsulation
4. Abstraction

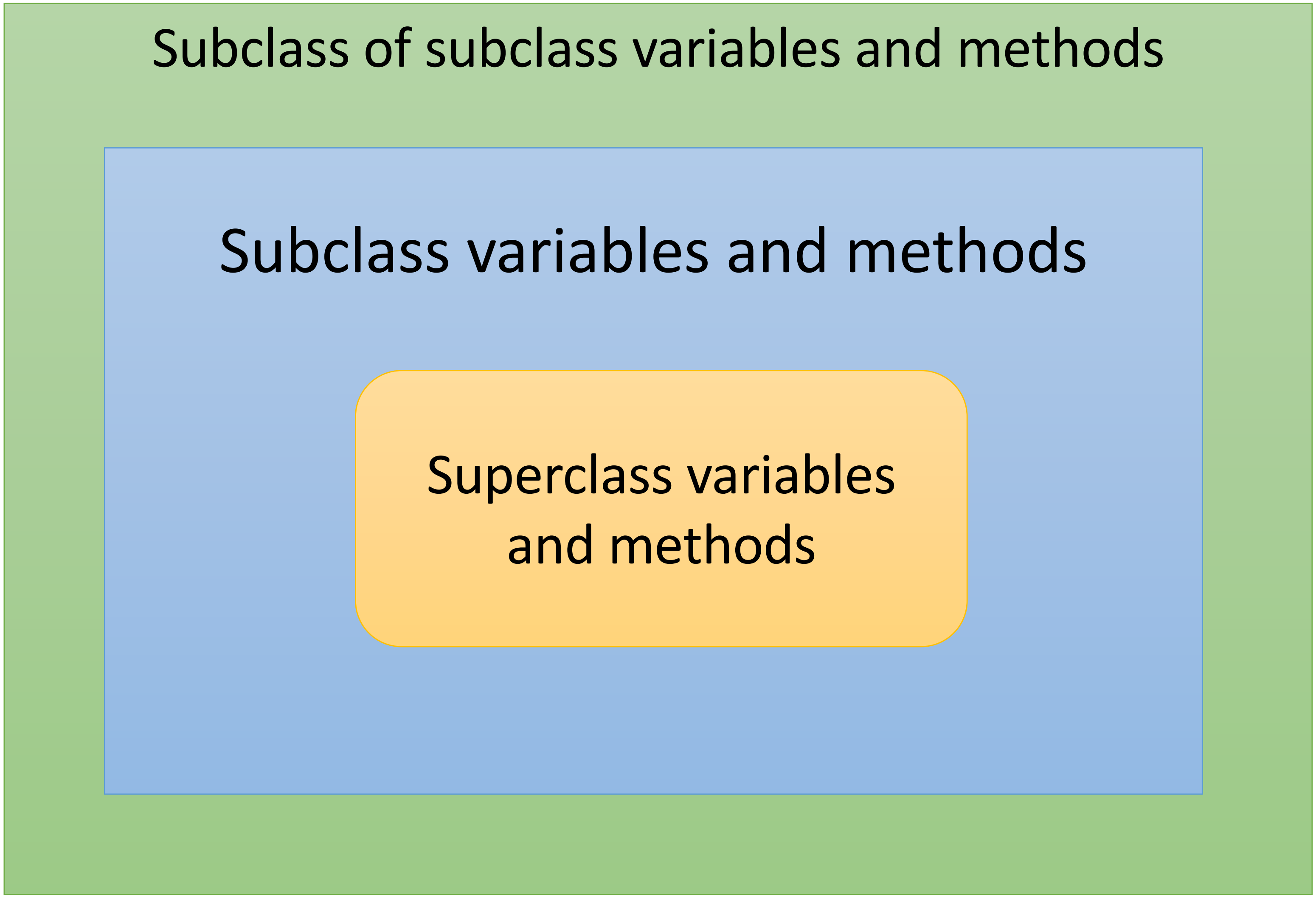
They typically interact with one another.



1. Inheritance

Inheritance in picture

Hierarchical organization



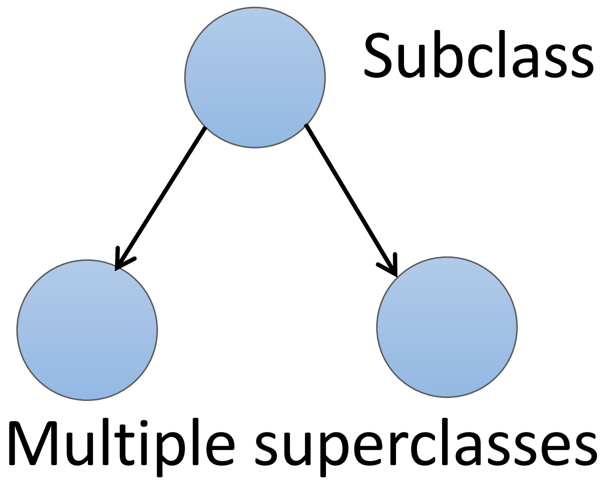
Subclass of subclass variables and methods

Subclass variables and methods

Superclass variables and methods

Inheritance in Java

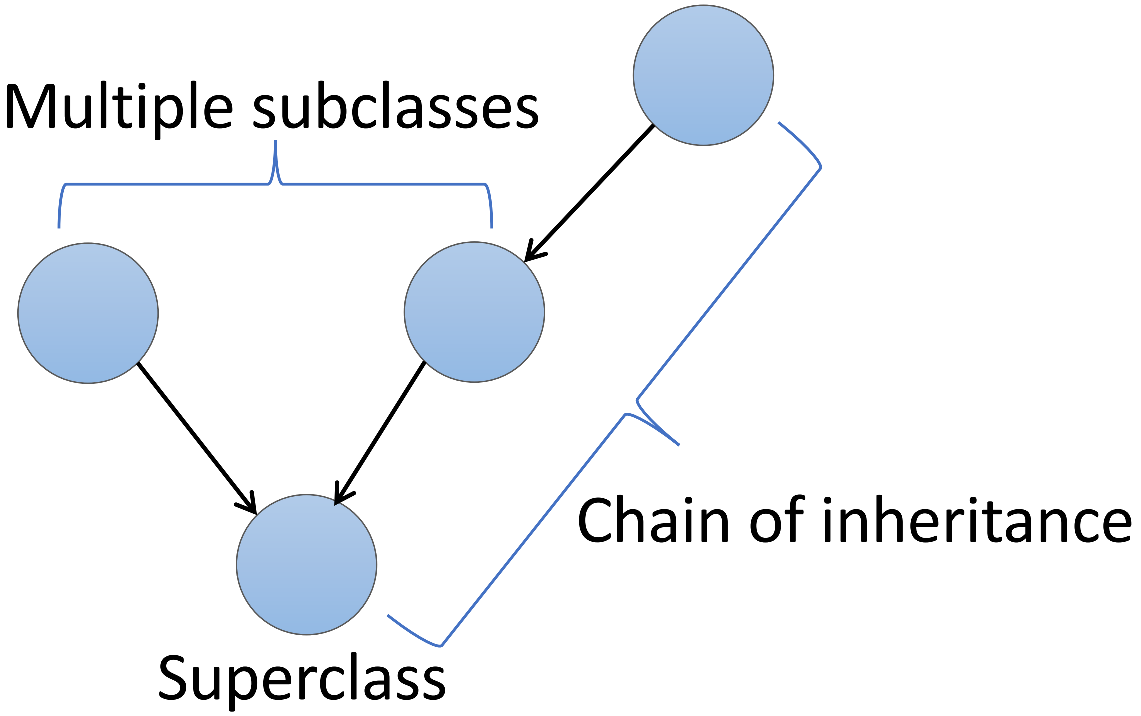
✗



Subclass

Multiple superclasses

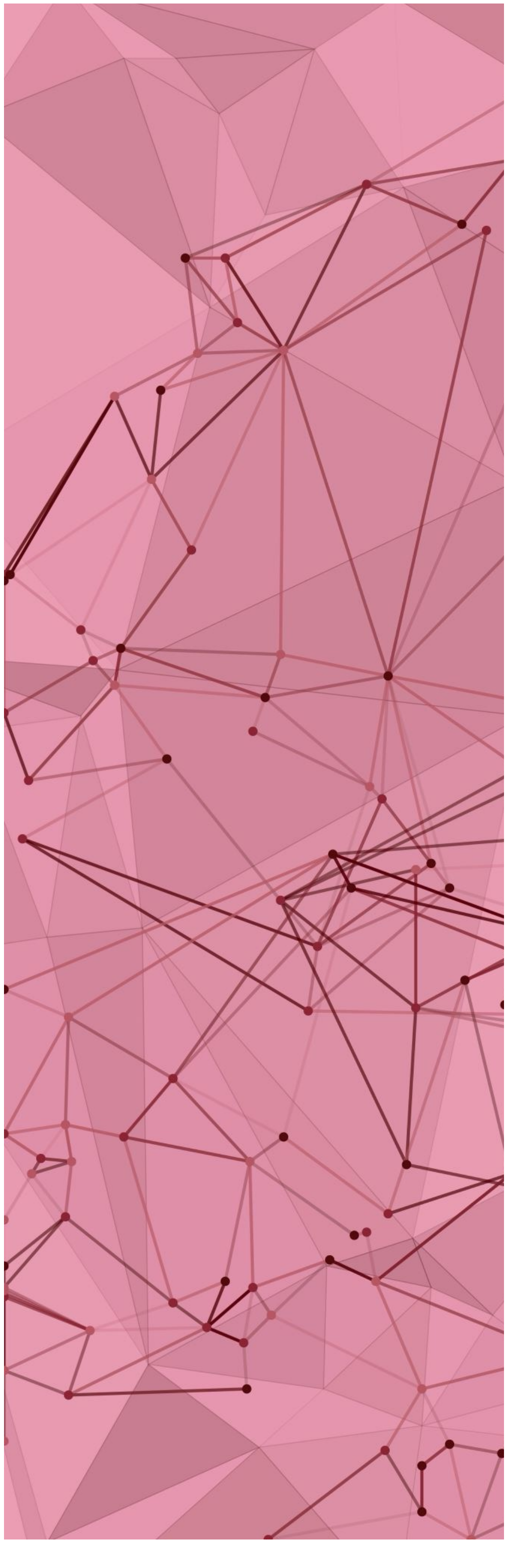
✓



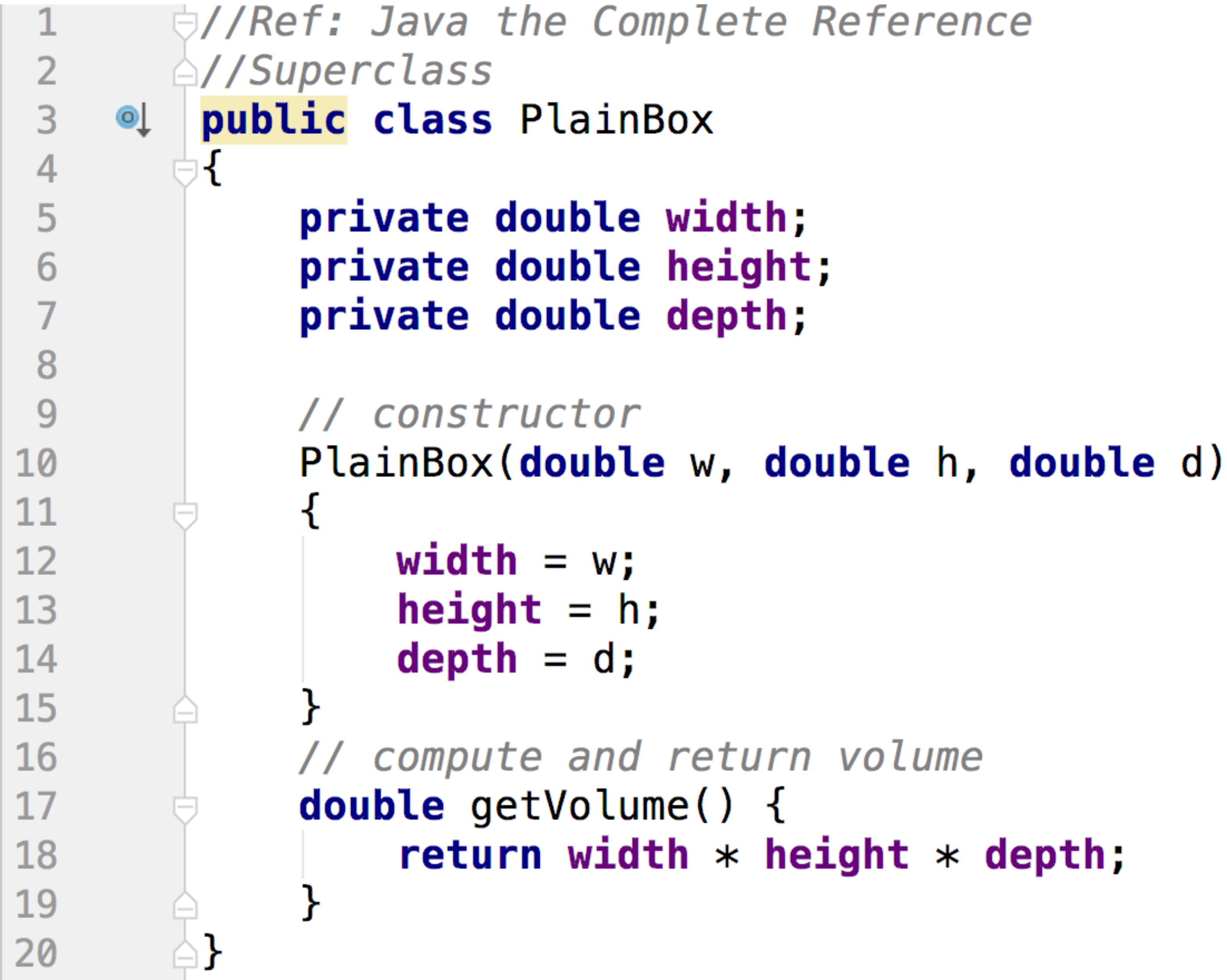
Multiple subclasses

Chain of inheritance

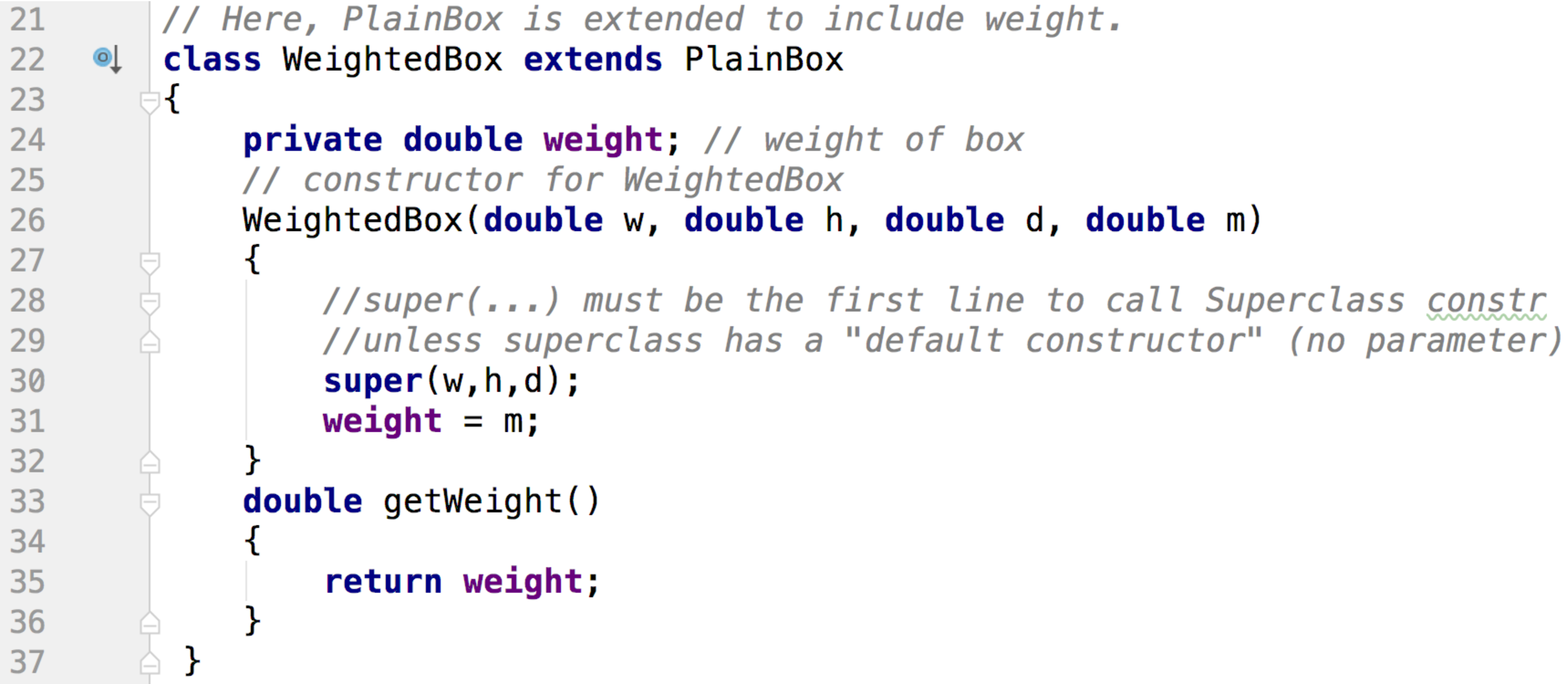
Superclass



Inheritance Demo



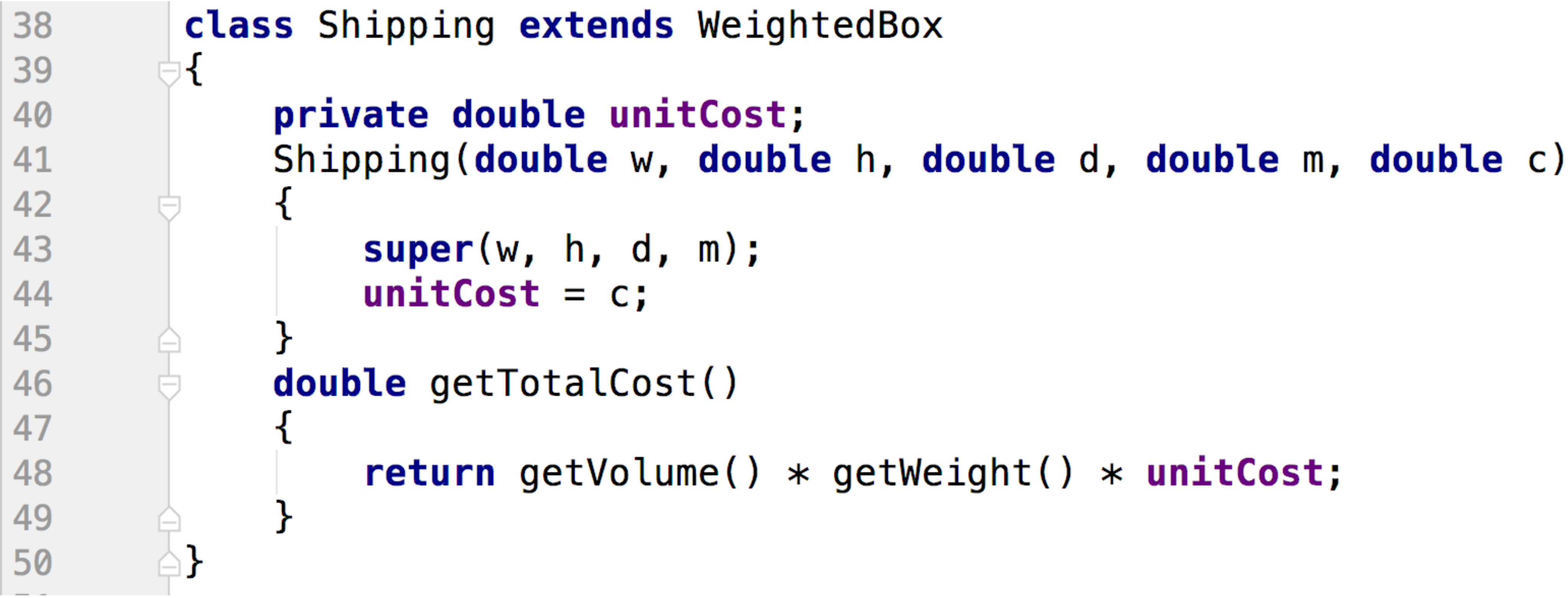
```java
//Ref: Java the Complete Reference
//Superclass
public class PlainBox
{
    private double width;
    private double height;
    private double depth;

    // constructor
    PlainBox(double w, double h, double d)
    {
        width = w;
        height = h;
        depth = d;
    }
    // compute and return volume
    double getVolume() {
        return width * height * depth;
    }
}
```



```java
// Here, PlainBox is extended to include weight.
class WeightedBox extends PlainBox
{
    private double weight; // weight of box
    // constructor for WeightedBox
    WeightedBox(double w, double h, double d, double m)
    {
        //super(...) must be the first line to call Superclass constr
        //unless superclass has a "default constructor" (no parameter)
        super(w,h,d);
        weight = m;
    }
    double getWeight()
    {
        return weight;
    }
 }
```

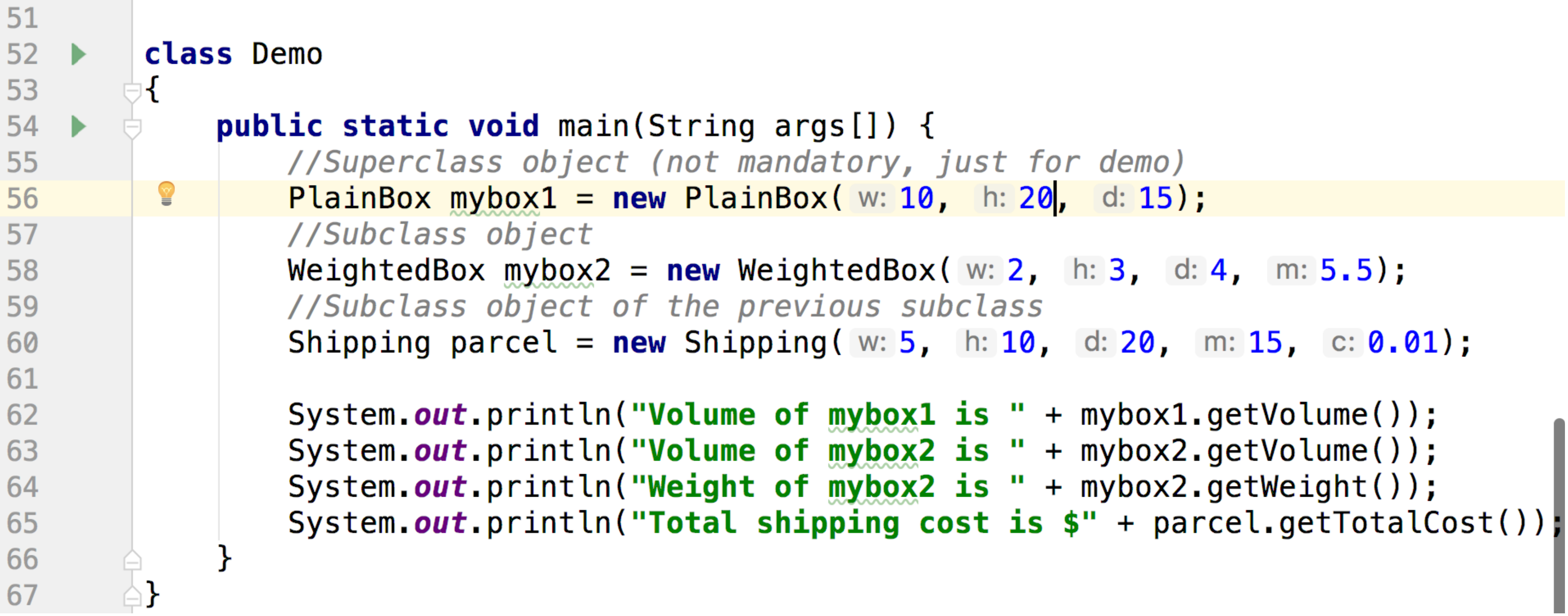
Chain of inheritance (Multilevel inheritance)

New class for shipping a box, inherits WeightedBox

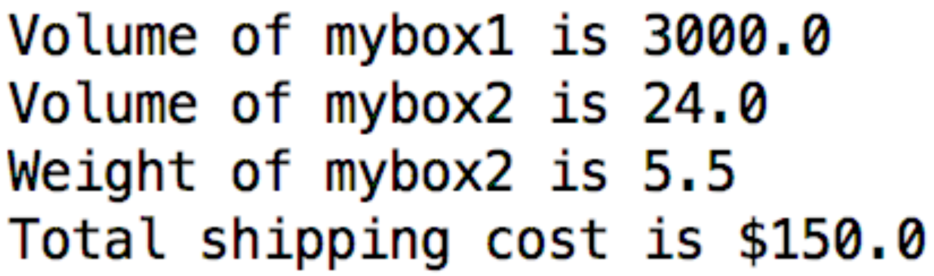


```
class Shipping extends WeightedBox
{
    private double unitCost;
    Shipping(double w, double h, double d, double m, double c)
    {
        super(w, h, d, m);
        unitCost = c;
    }
    double getTotalCost()
    {
        return getVolume() * getWeight() * unitCost;
    }
}
```

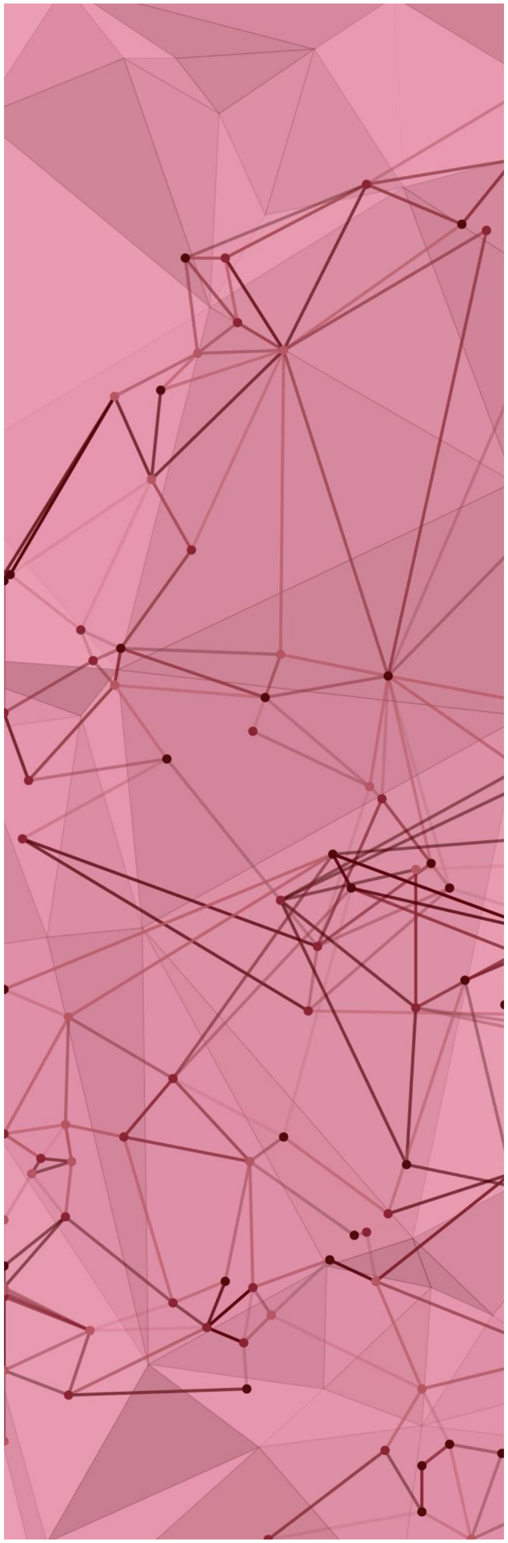
Demo



```java
class Demo
{
    public static void main(String args[]) {
        //Superclass object (not mandatory, just for demo)
        PlainBox mybox1 = new PlainBox( w: 10,  h: 20,  d: 15);
        //Subclass object
        WeightedBox mybox2 = new WeightedBox( w: 2,  h: 3,  d: 4,  m: 5.5);
        //Subclass object of the previous subclass
        Shipping parcel = new Shipping( w: 5,  h: 10,  d: 20,  m: 15,  c: 0.01);

        System.out.println("Volume of mybox1 is " + mybox1.getVolume());
        System.out.println("Volume of mybox2 is " + mybox2.getVolume());
        System.out.println("Weight of mybox2 is " + mybox2.getWeight());
        System.out.println("Total shipping cost is $" + parcel.getTotalCost());
    }
}
```



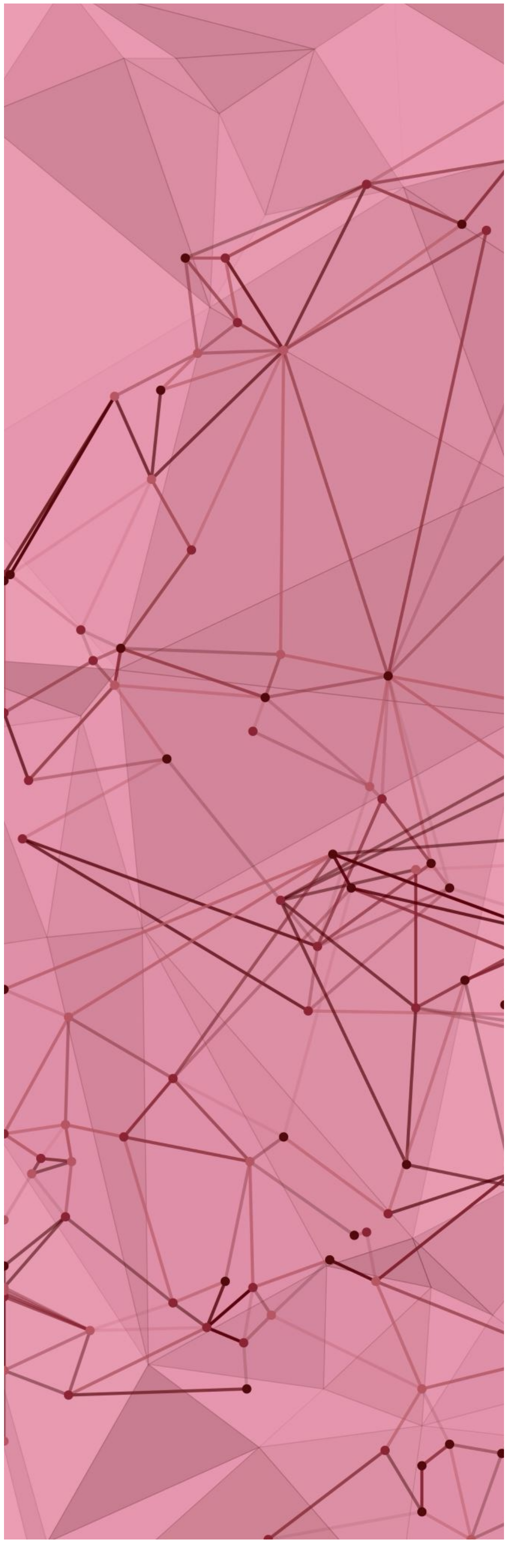
```
Volume of mybox1 is 3000.0
Volume of mybox2 is 24.0
Weight of mybox2 is 5.5
Total shipping cost is $150.0
```



2. Polymorphism

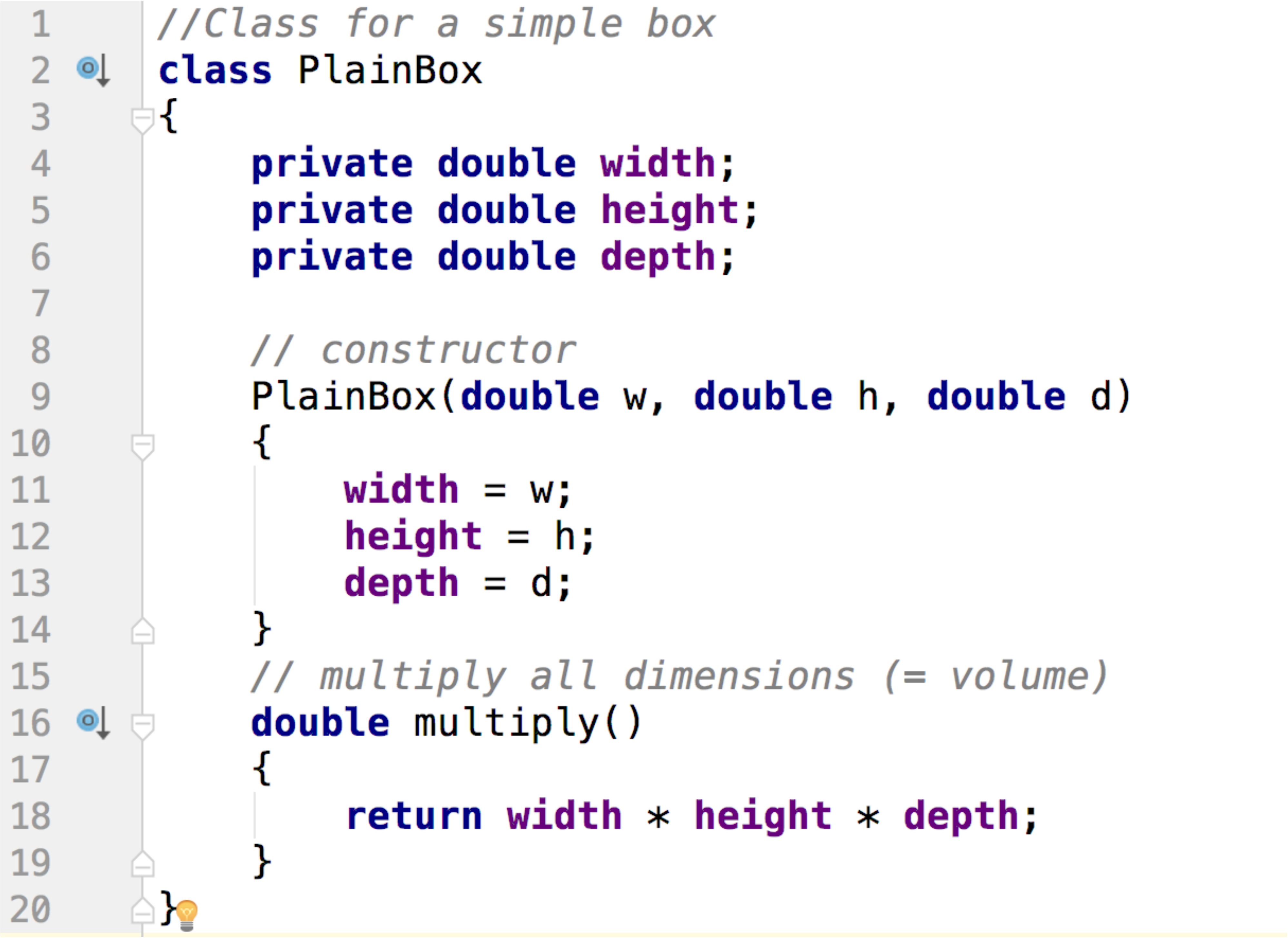
Functional vs.

dynamic polymorphism

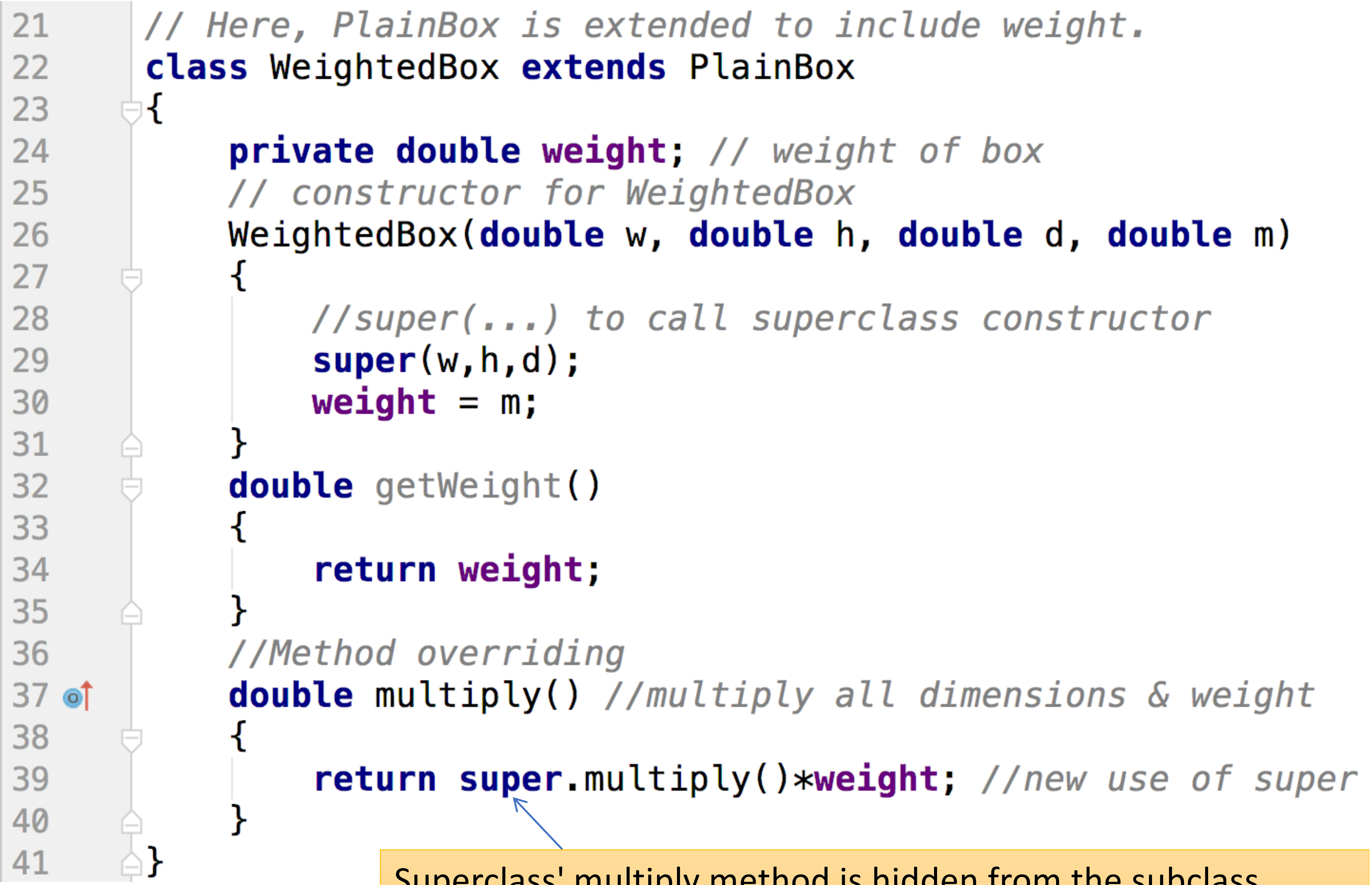


OOP Polymorphism

Why is OOP polymorphism called dynamic?

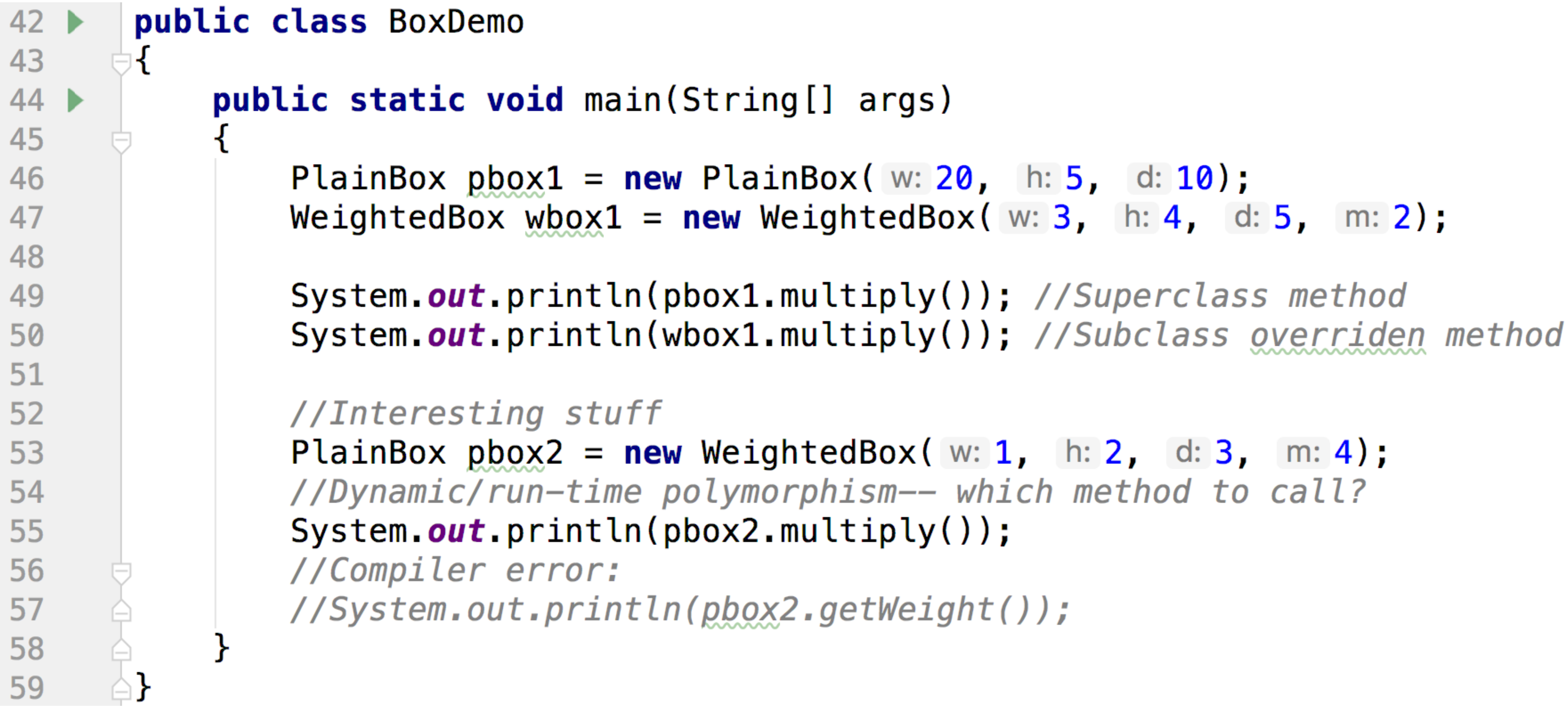


```java
//Class for a simple box
class PlainBox
{
    private double width;
    private double height;
    private double depth;

    // constructor
    PlainBox(double w, double h, double d)
    {
        width = w;
        height = h;
        depth = d;
    }
    // multiply all dimensions (= volume)
    double multiply()
    {
        return width * height * depth;
    }
}
```



```java
// Here, PlainBox is extended to include weight.
class WeightedBox extends PlainBox
{
    private double weight; // weight of box
    // constructor for WeightedBox
    WeightedBox(double w, double h, double d, double m)
    {
        //super(...) to call superclass constructor
        super(w,h,d);
        weight = m;
    }
    double getWeight()
    {
        return weight;
    }
    //Method overriding
    double multiply() //multiply all dimensions & weight
    {
        return super.multiply()*weight; //new use of super
    }
}
```

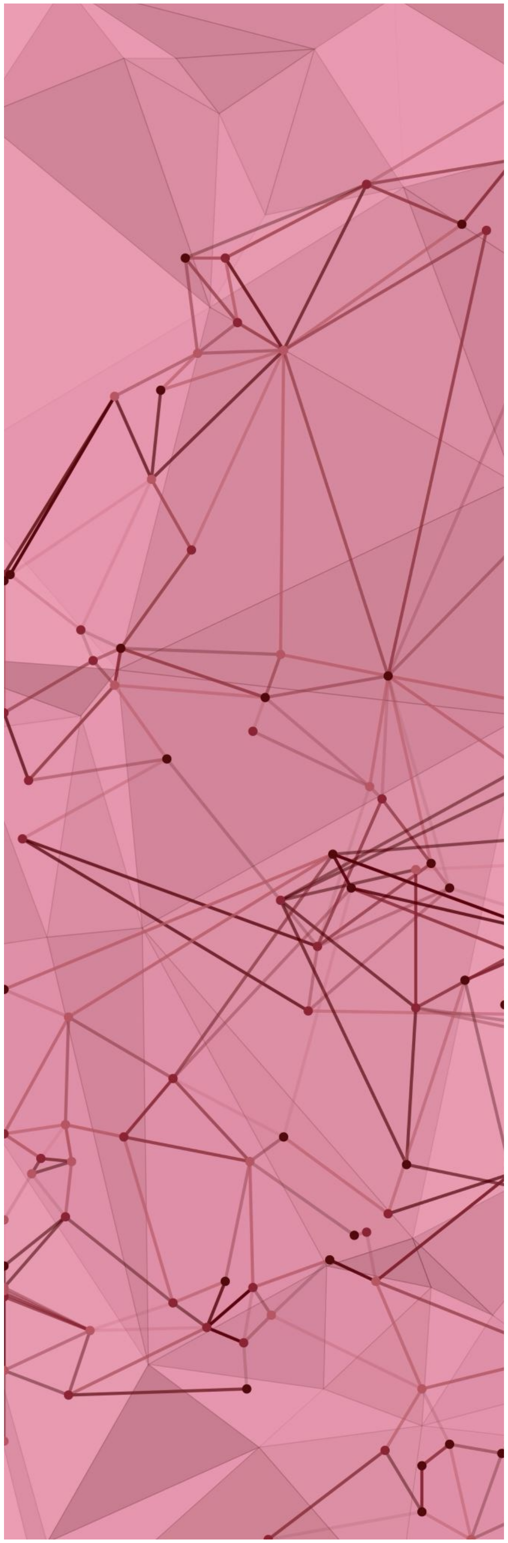
Superclass' multiply method is hidden from the subclass unless the subclass explicitly calls it using **super**



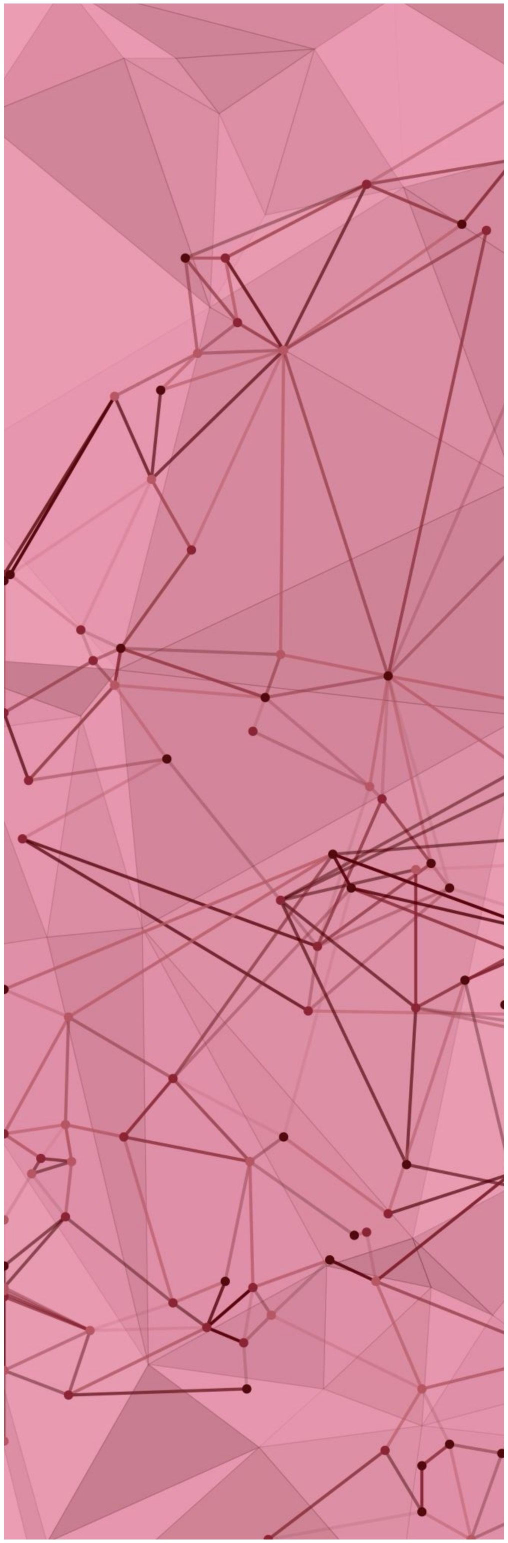
```java
public class BoxDemo
{
    public static void main(String[] args)
    {
        PlainBox pbox1 = new PlainBox( w: 20,  h: 5,  d: 10);
        WeightedBox wbox1 = new WeightedBox( w: 3,  h: 4,  d: 5,  m: 2);

        System.out.println(pbox1.multiply()); //Superclass method
        System.out.println(wbox1.multiply()); //Subclass overriden method

        //Interesting stuff
        PlainBox pbox2 = new WeightedBox( w: 1,  h: 2,  d: 3,  m: 4);
        //Dynamic/run-time polymorphism-- which method to call?
        System.out.println(pbox2.multiply());
        //Compiler error:
        //System.out.println(pbox2.getWeight());
    }
}
```

Output

```
1000.0
120.0
24.0
```



3. Encapsulation

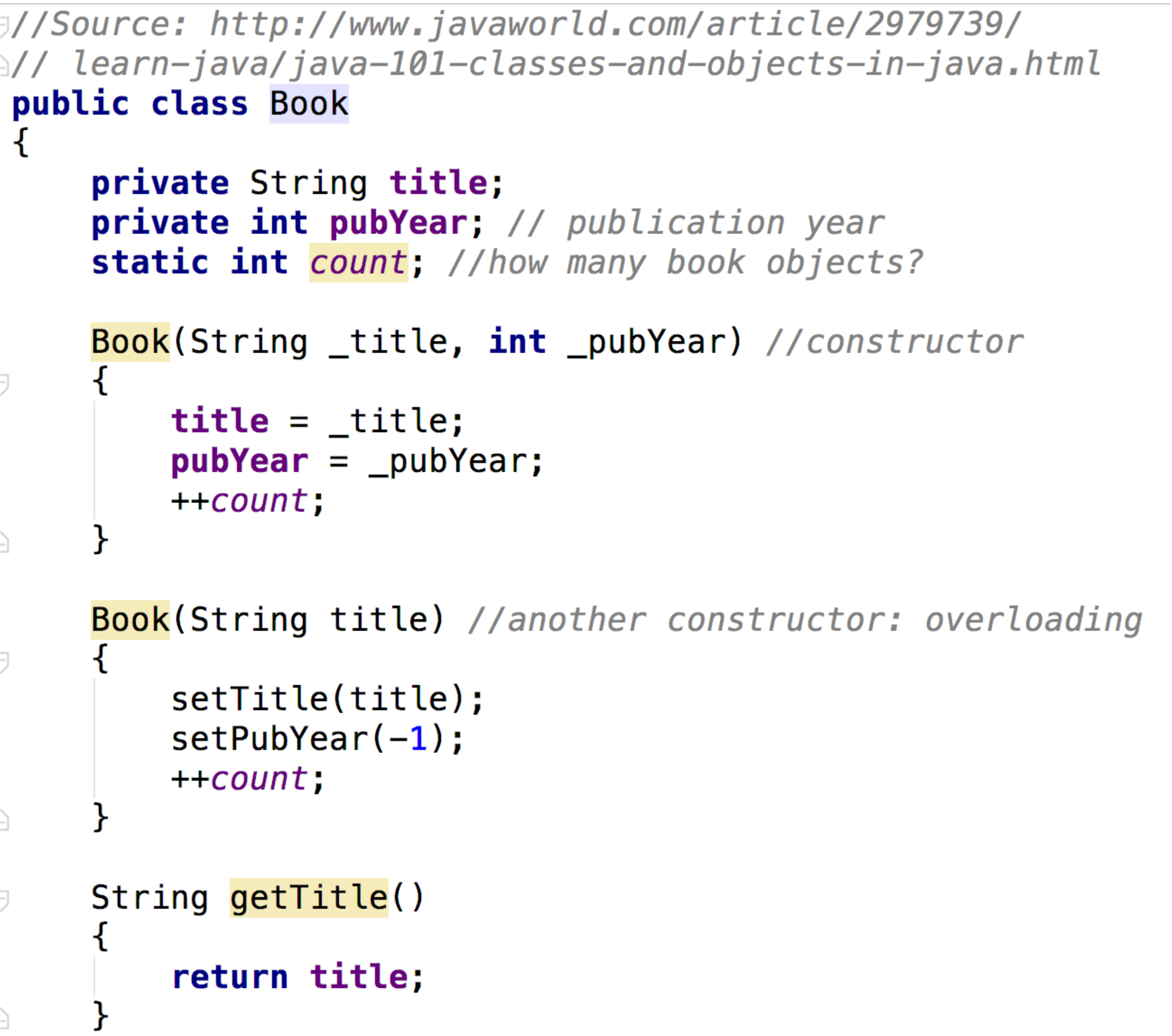


Encapsulation Demo

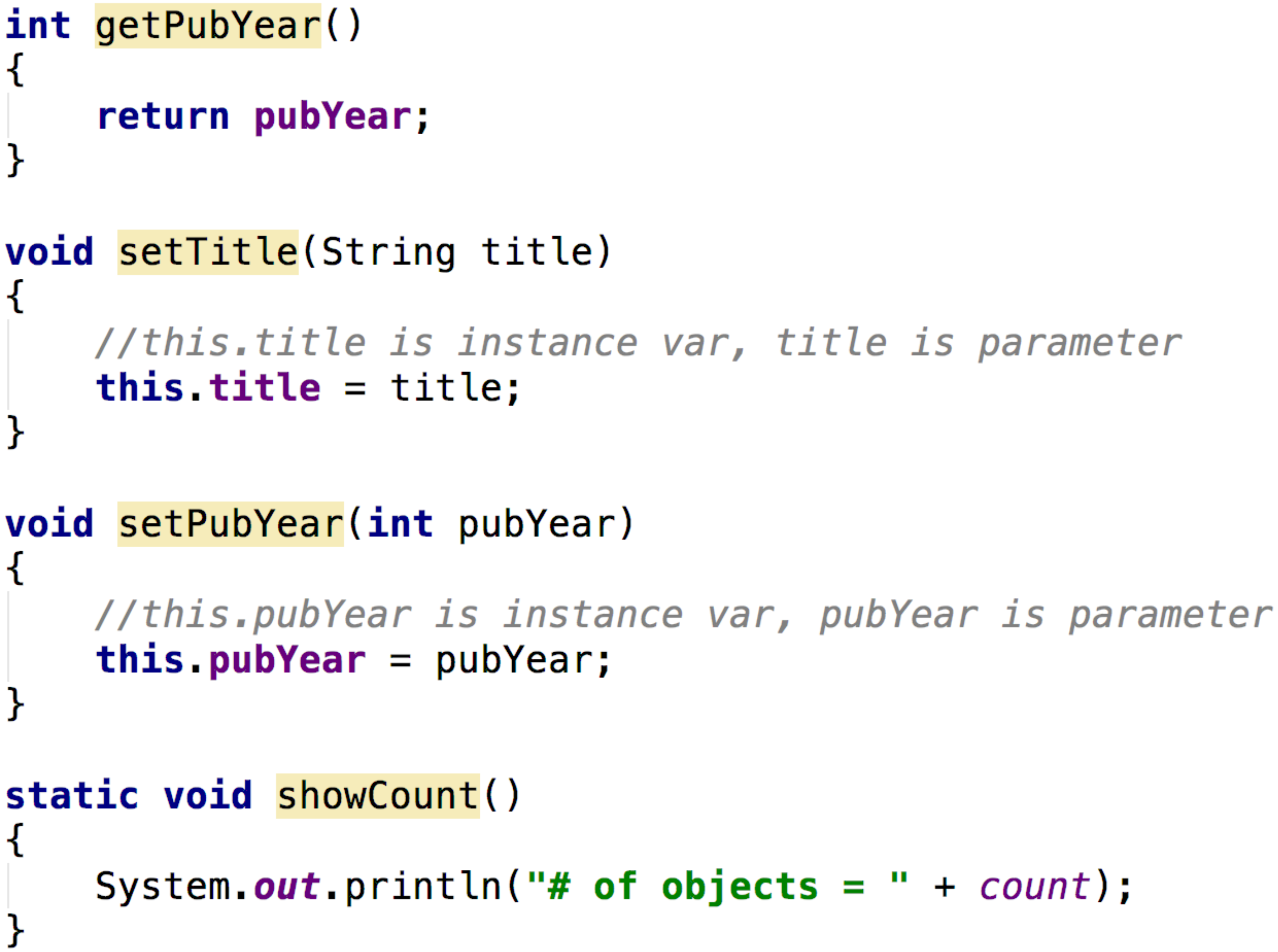
Encapsulation example: Book class

- Want a class for representing certain information about a book
  - Note: each object is <u>one</u> single book
  - Multiple objects → many books

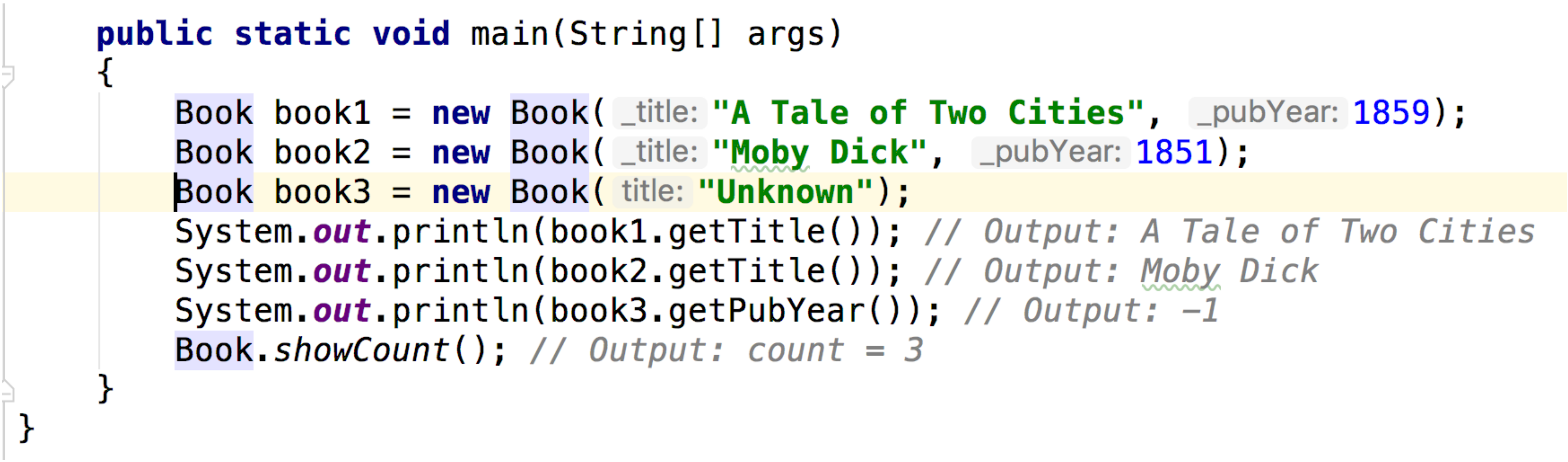
1. What are the attributes or properties of a book?
2. What are the actions or behaviors that you can apply on <u>book data</u>?
   - Helps with data abstraction



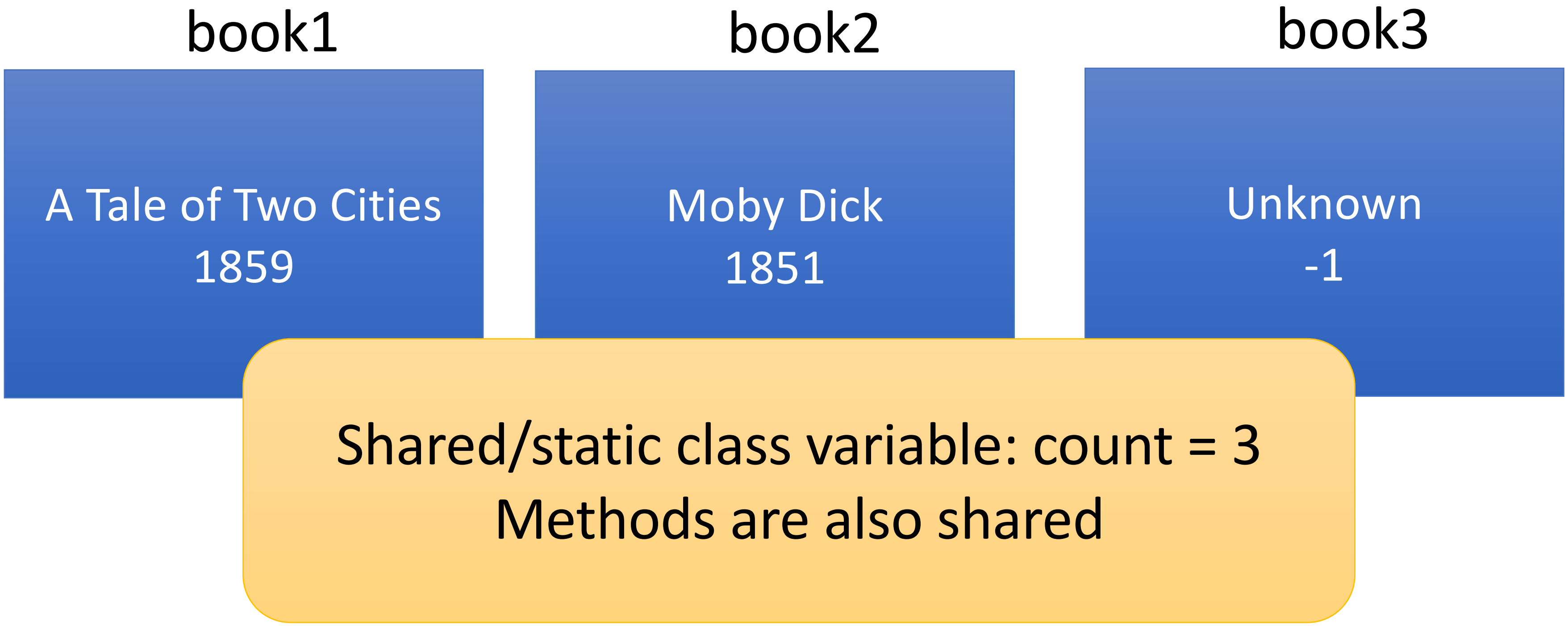
```java
//Source: http://www.javaworld.com/article/2979739/
// learn-java/java-101-classes-and-objects-in-java.html
public class Book
{
    private String title;
    private int pubYear; // publication year
    static int count; //how many book objects?

    Book(String _title, int _pubYear) //constructor
    {
        title = _title;
        pubYear = _pubYear;
        ++count;
    }

    Book(String title) //another constructor: overloading
    {
        setTitle(title);
        setPubYear(-1);
        ++count;
    }

    String getTitle()
    {
        return title;
    }
```



```
int getPubYear()
{
    return pubYear;
}

void setTitle(String title)
{
    //this.title is instance var, title is parameter
    this.title = title;
}

void setPubYear(int pubYear)
{
    //this.pubYear is instance var, pubYear is parameter
    this.pubYear = pubYear;
}

static void showCount()
{
    System.out.println("# of objects = " + count);
}
```



```
    public static void main(String[] args)
    {
        Book book1 = new Book( _title: "A Tale of Two Cities", _pubYear: 1859);
        Book book2 = new Book( _title: "Moby Dick", _pubYear: 1851);
        Book book3 = new Book( title: "Unknown");
        System.out.println(book1.getTitle()); // Output: A Tale of Two Cities
        System.out.println(book2.getTitle()); // Output: Moby Dick
        System.out.println(book3.getPubYear()); // Output: -1
        Book.showCount(); // Output: count = 3
    }
}
```



book1

A Tale of Two Cities
1859

book2

Moby Dick
1851

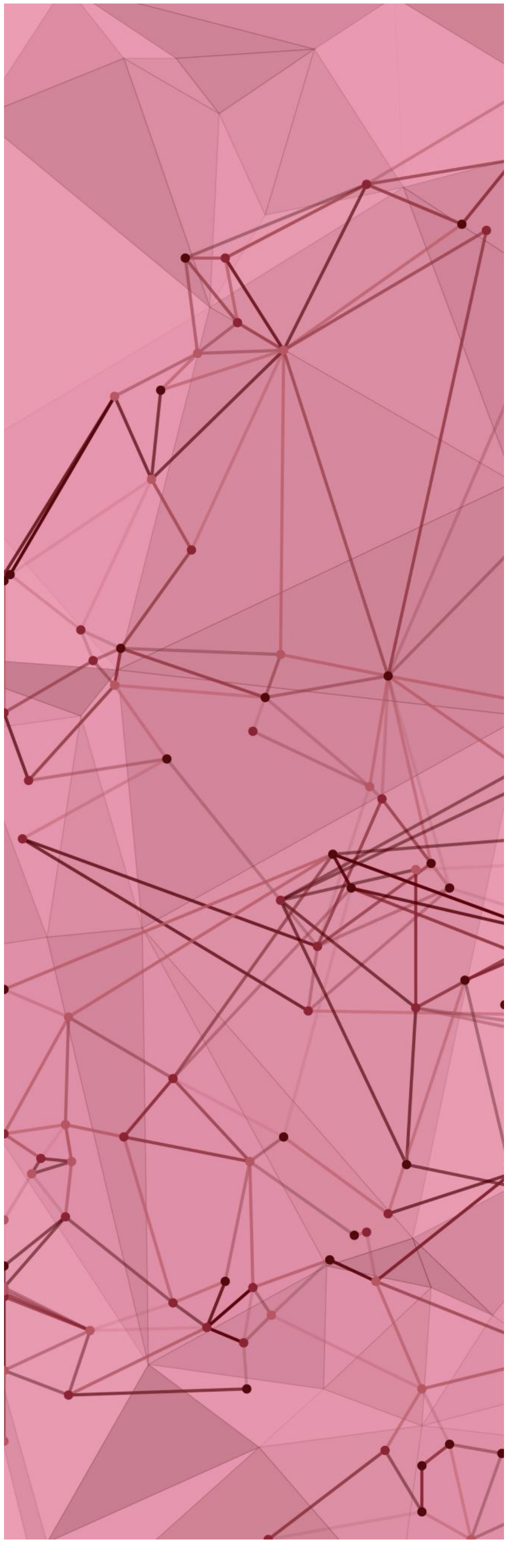
book3

Unknown
-1

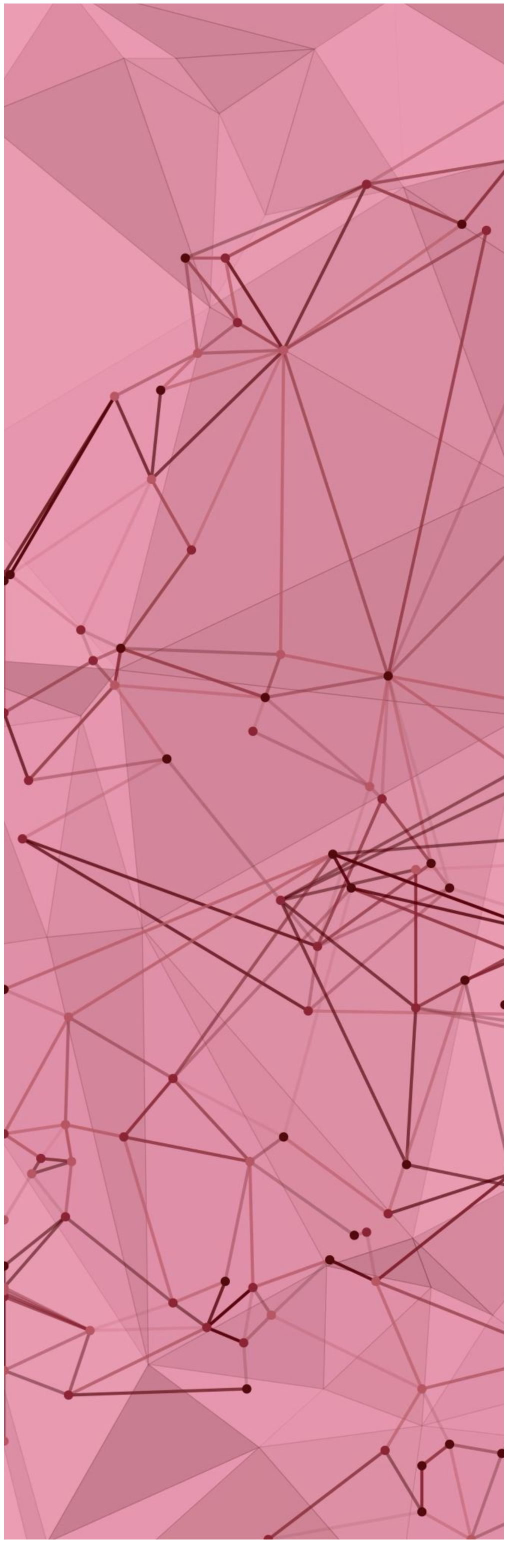
Shared/static class variable: count = 3
Methods are also shared

Encapsulation question

Build on the Book class to include author names. How would you represent multiple authors?



4. Abstraction

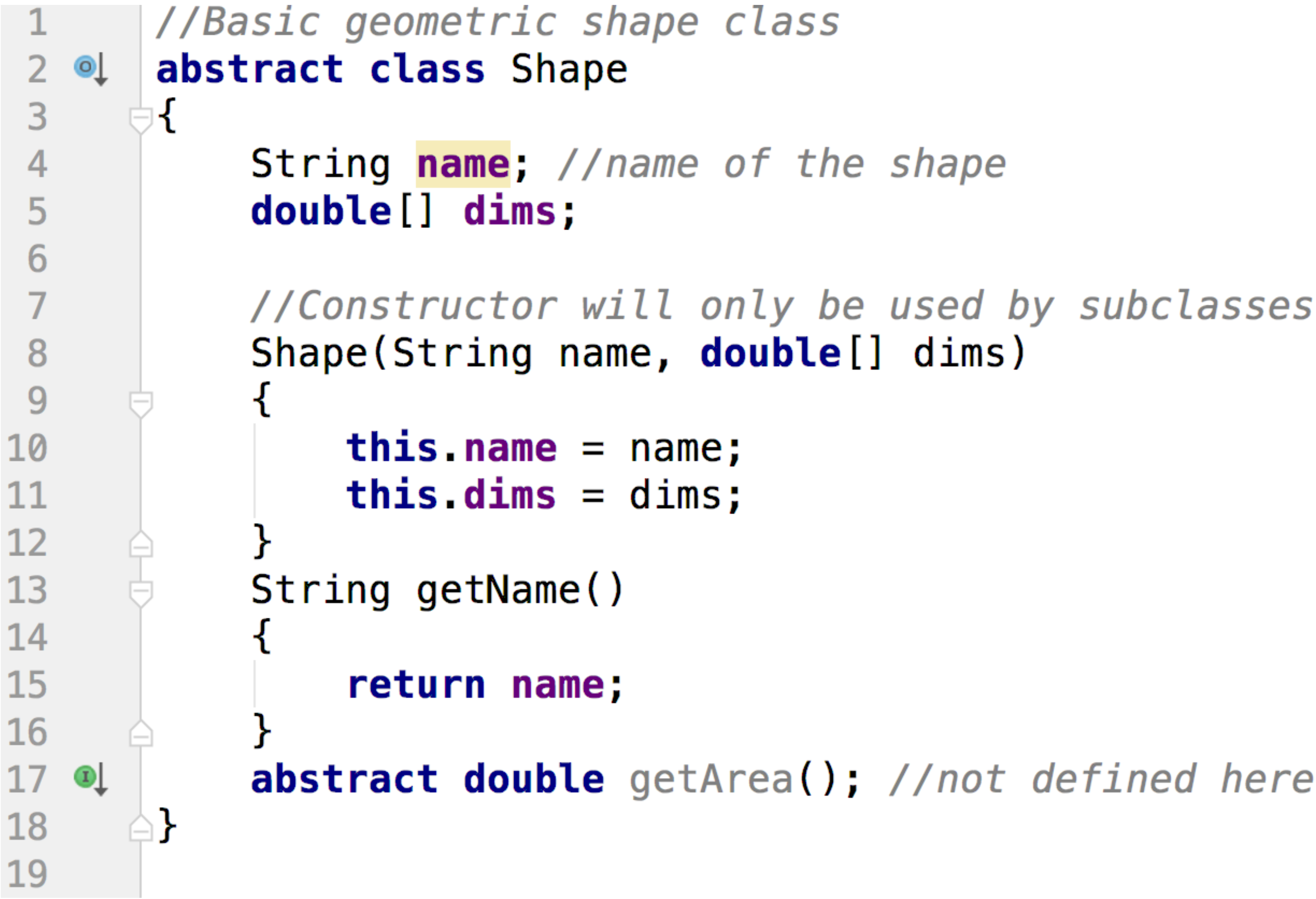


Abstraction Demo

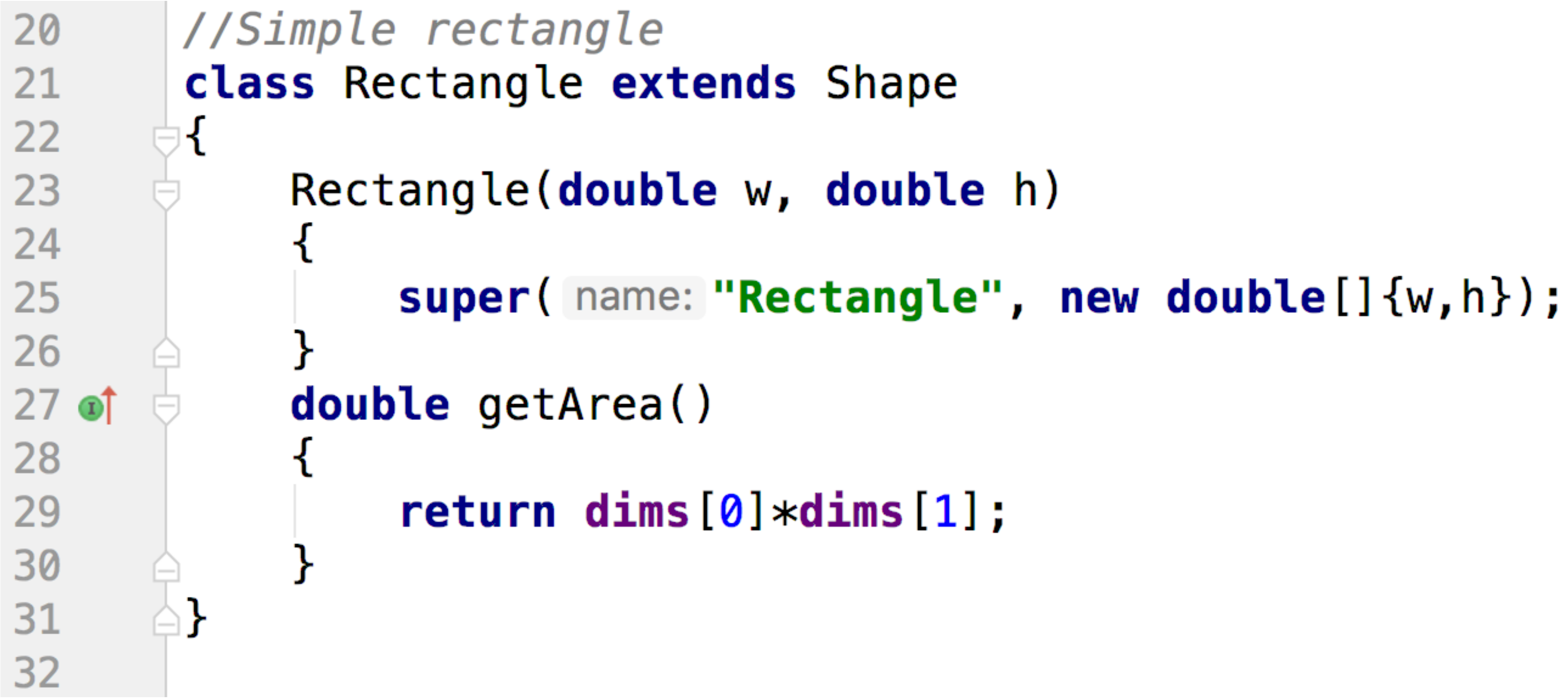
Abstract class in Java

- Give high-level ideas while hiding implementation details
- Use: manage complexity
- Next few slides
  - Abstract class Shape outlines a geometric shape
    - Which shape?
    - getArea(): Area depends on shape!
  - Subclasses of Shape: defines the getArea() method
    - Rectangle
    - Triangle

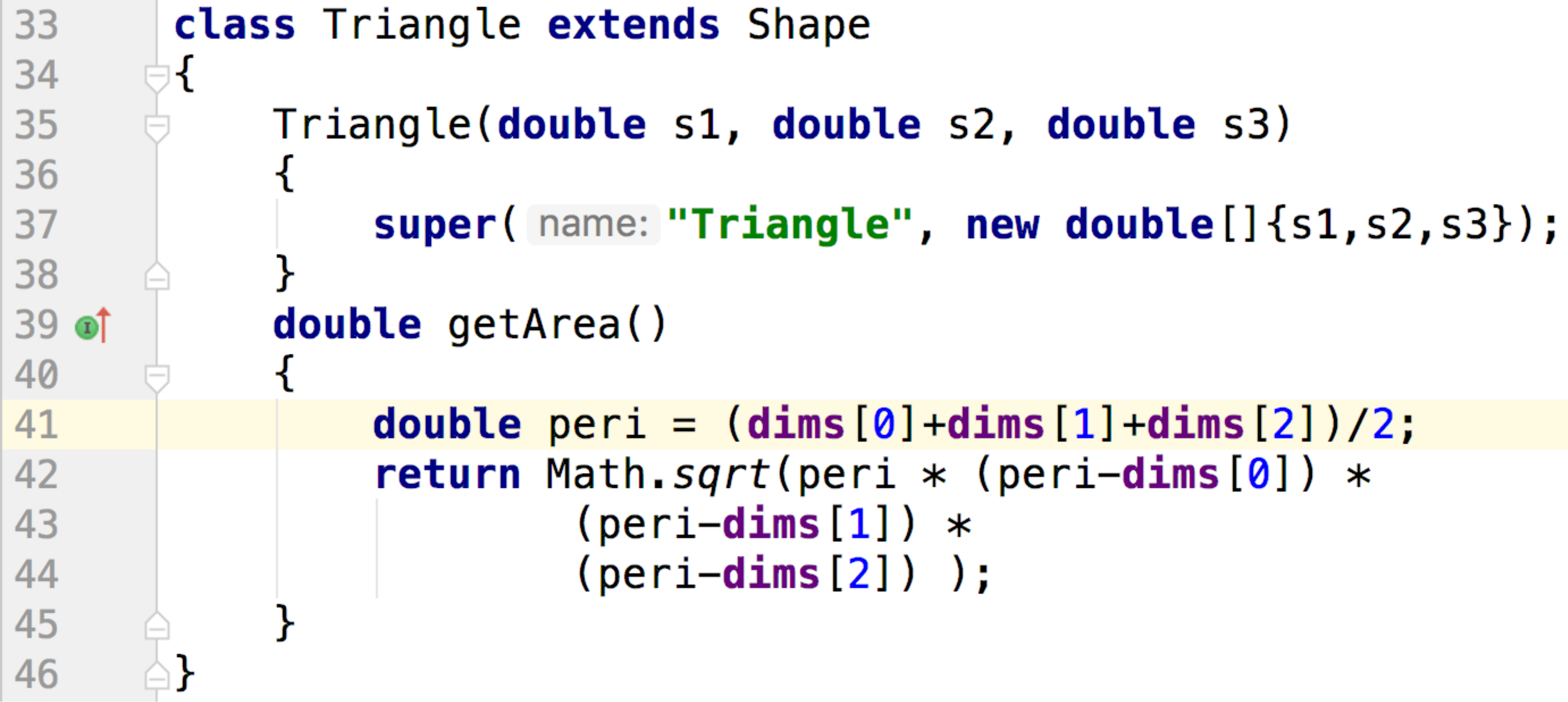
Cannot create any object of abstract class!



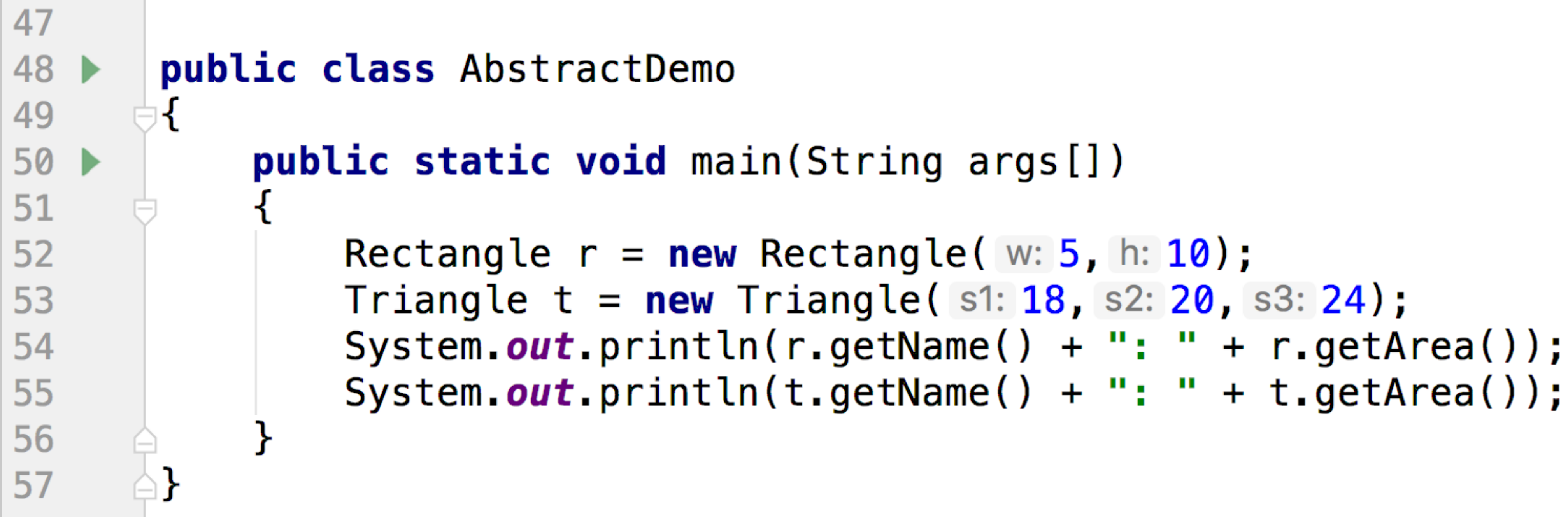
```java
//Basic geometric shape class
abstract class Shape
{
    String name; //name of the shape
    double[] dims;

    //Constructor will only be used by subclasses
    Shape(String name, double[] dims)
    {
        this.name = name;
        this.dims = dims;
    }
    String getName()
    {
        return name;
    }
    abstract double getArea(); //not defined here
}
```



```java
//Simple rectangle
class Rectangle extends Shape
{
    Rectangle(double w, double h)
    {
        super( name: "Rectangle", new double[]{w,h});
    }
    double getArea()
    {
        return dims[0]*dims[1];
    }
}
```

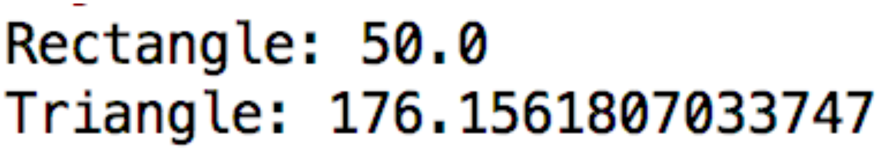


```java
class Triangle extends Shape
{
    Triangle(double s1, double s2, double s3)
    {
        super( name: "Triangle", new double[]{s1,s2,s3});
    }
    double getArea()
    {
        double peri = (dims[0]+dims[1]+dims[2])/2;
        return Math.sqrt(peri * (peri-dims[0]) *
                (peri-dims[1]) *
                (peri-dims[2]) );
    }
}
```



```
public class AbstractDemo
{
    public static void main(String args[])
    {
        Rectangle r = new Rectangle( w: 5, h: 10);
        Triangle t = new Triangle( s1: 18, s2: 20, s3: 24);
        System.out.println(r.getName() + ": " + r.getArea());
        System.out.println(t.getName() + ": " + t.getArea());
    }
}
```

Output



```
Rectangle: 50.0
Triangle: 176.1561807033747
```

Encapsulation vs abstraction vs information hiding

- Debates on orthogonality of concepts
- Roughly–
  data abstraction (capsule) vs
  process abstraction (generalization)
- http://www.tonymarston.co.uk/php-mysql/abstraction.txt
  - By Edward V. Berard

Snapshot of debate

“Encapsulation or equivalently information hiding refers to the practice of including within an object everything it needs and furthermore doing this in such a way that no other object need ever be aware of this internal structure.”
-- [Ian Graham, 1991]

“If encapsulation was "the same thing as information hiding," then one might make the argument that "everything that was encapsulated was also hidden." This is obviously not true. ... It is indeed true that encapsulation mechanisms such as classes allow some information to be hidden. However, these same encapsulation mechanisms also allow some information to be visible. Some even allow varying degrees of visibility, e.g., C++'s public, protected, and private members.”
-- Edward V. Berard

Encapsulation: good definition

"Encapsulation is used as a generic term for techniques which realize data abstraction. Encapsulation therefore implies the provision of mechanisms to support both modularity and information hiding. There is therefore a one to one correspondence in this case between the technique of encapsulation and the principle of data abstraction."
-- [Blair et al, 1991]

Abstraction: good definition

"Abstraction is generally defined as 'the process of formulating generalised concepts by extracting common qualities from specific examples.'"

-- [Blair et al, 1991]