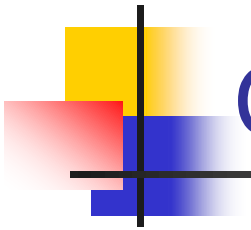
CÔNG NGHỆ JAVA

Trường: Đại học Giao thông vận Tải

Khoa: Công nghệ thông tin



CÔNG NGHỆ JAVA

Chương 3:

Lập trình hướng đối tượng trong Java

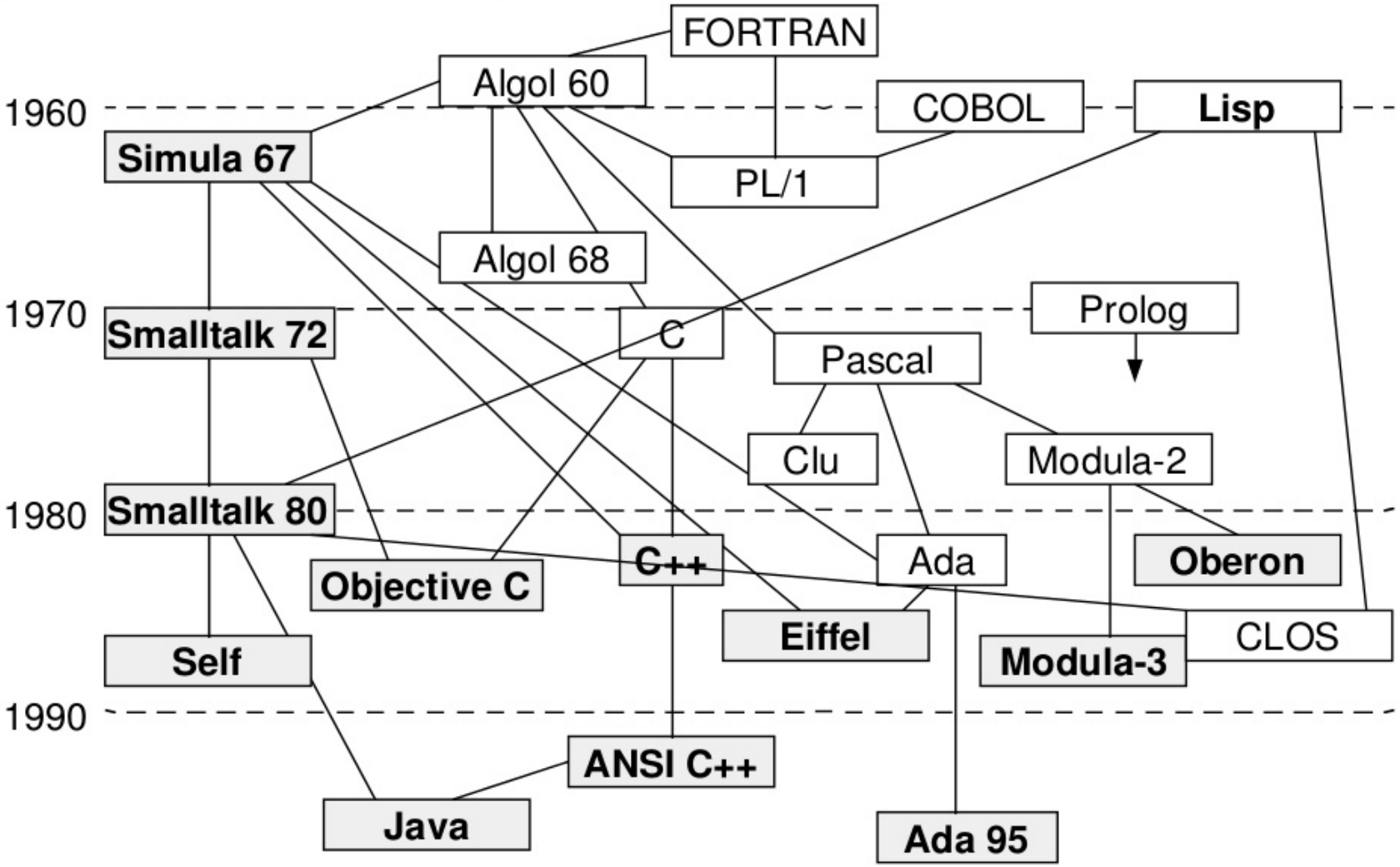
Nội dung

- Nội dung
  - Các khái niệm về OOP
    - Khái niệm về OOP
    - Các đặc trưng của OOP: lớp, đối tượng, thuộc tính, phương thức, kế thừa, đóng gói, đa hình và trừu tượng.
  - OOP trong Java
    - Lớp và đối tượng
    - Tính đóng gói
    - Tính kế thừa
    - …
  - Mở rộng
    - Inner/Outer class.
    - Debug

Chương 3:

1. Khái niệm về OOP

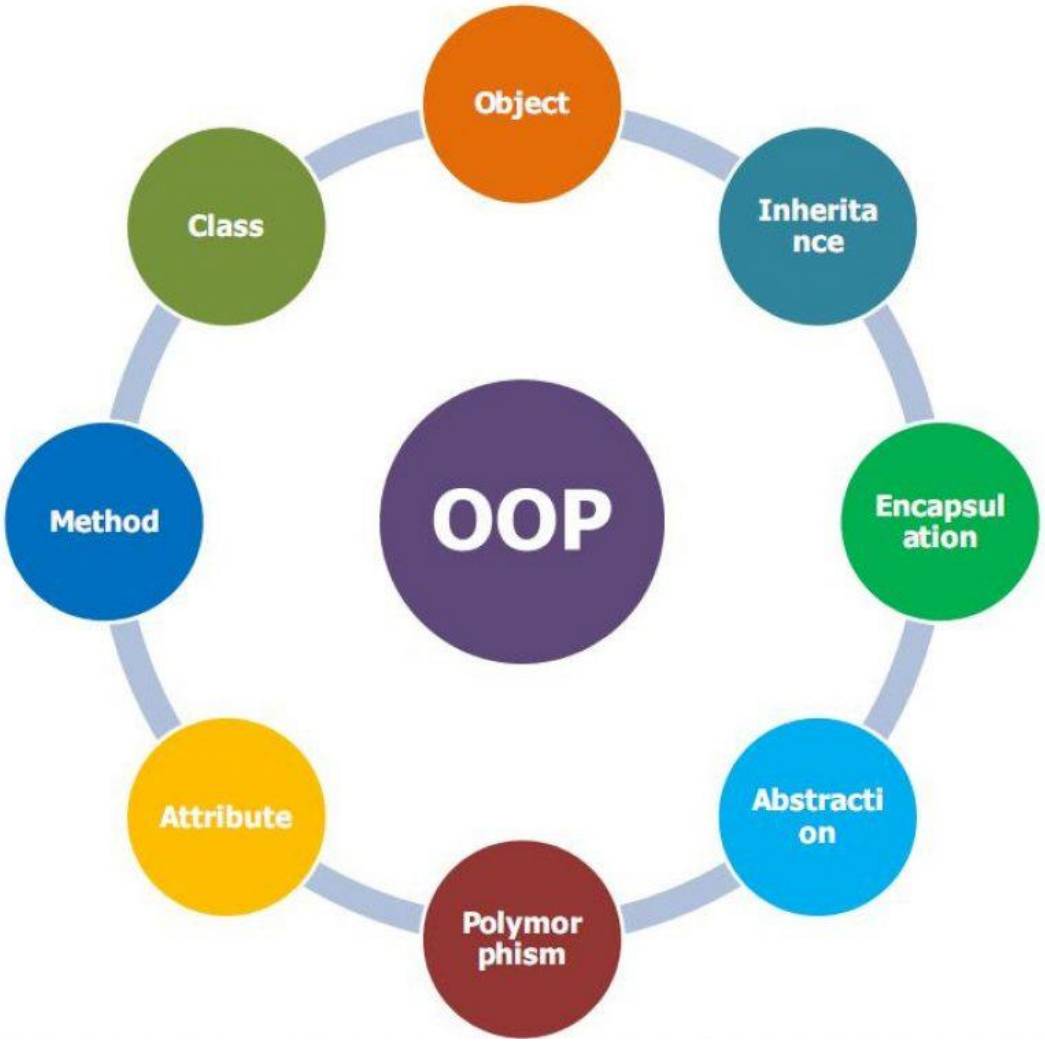
Khái niệm về OOP



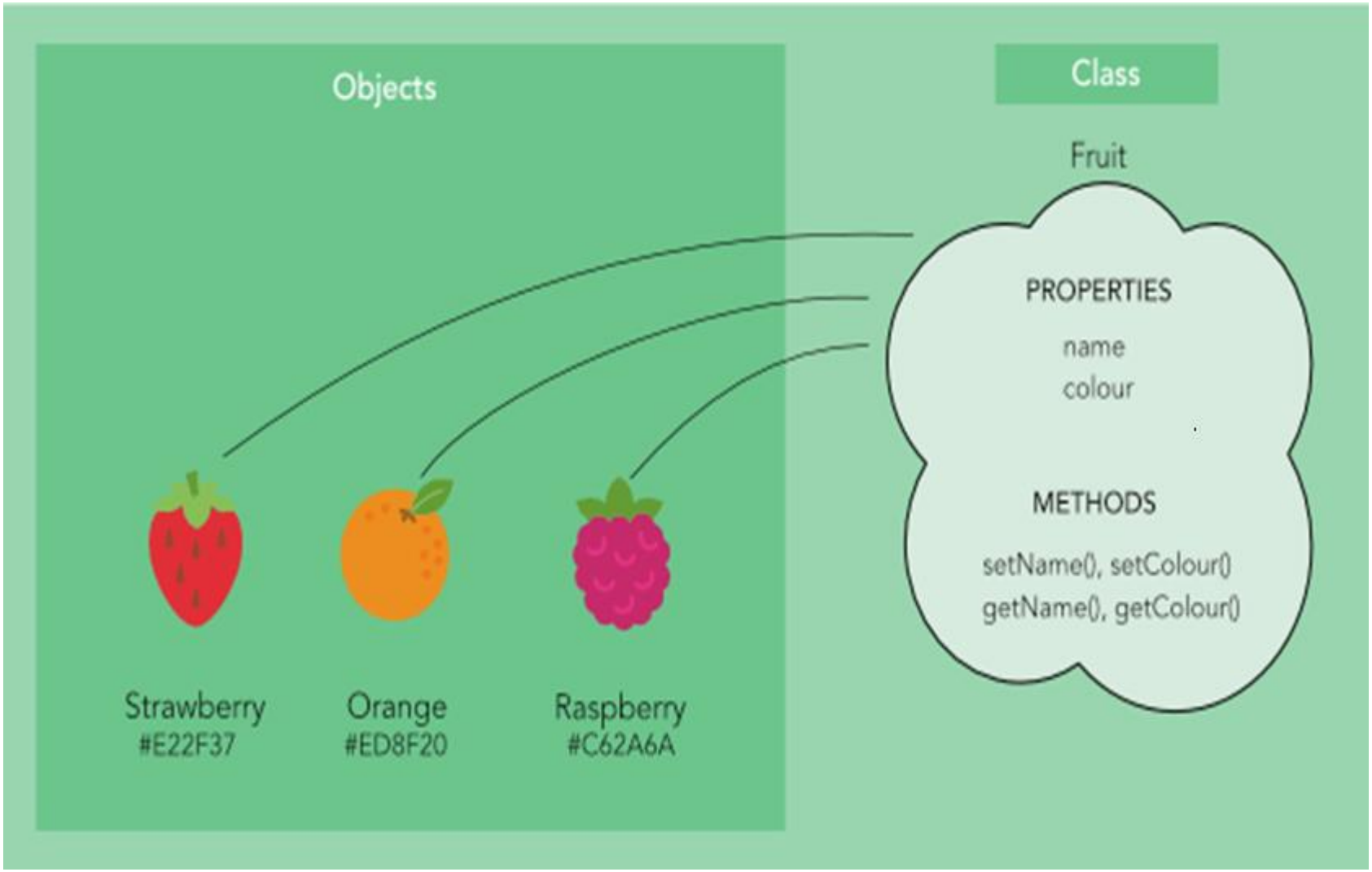
Khái niệm về OOP

- Khái niệm về đối tượng – object, thực thể – instance ra đời từ những năm 1960 tại MIT trong dự án phát triển phần mềm sử dụng ngôn ngữ Simular.
- Khái niệm OOP được đưa ra bởi Xerox PARC trong dự án phát triển sử dụng ngôn ngữ Smalltalk – coi các đối tượng là nền tảng của mọi thao tác, tính toán.
- Simular với tư tưởng của mình chính là nền tảng cho việc xây dựng và phát triển các ngôn ngữ cung như tư tưởng về OOP. Các ngôn ngữ khác trên tư tưởng đó cũng được hình thành như C++, Java, …
- 1980s, OOP trở thành tư tưởng để phát triển mà đại diện tiêu biểu chính là C++: chiếm 73% thị phần 1990.
- Ngày nay: hầu hết các ngôn ngữ đều hỗ trợ OOP.

Khái niệm về OOP



OOP – Class vs Object



OOP – Class vs Object

Breed: Bulldog
Size: large
Colour: light gray
Age: 5 years

Dog_Object

Breed: Beagle
Size: large
Colour: orange
Age: 6 years

Dog_Object

Breed: German Shepherd
Size: large
Colour: white & orange
Age: 6 years

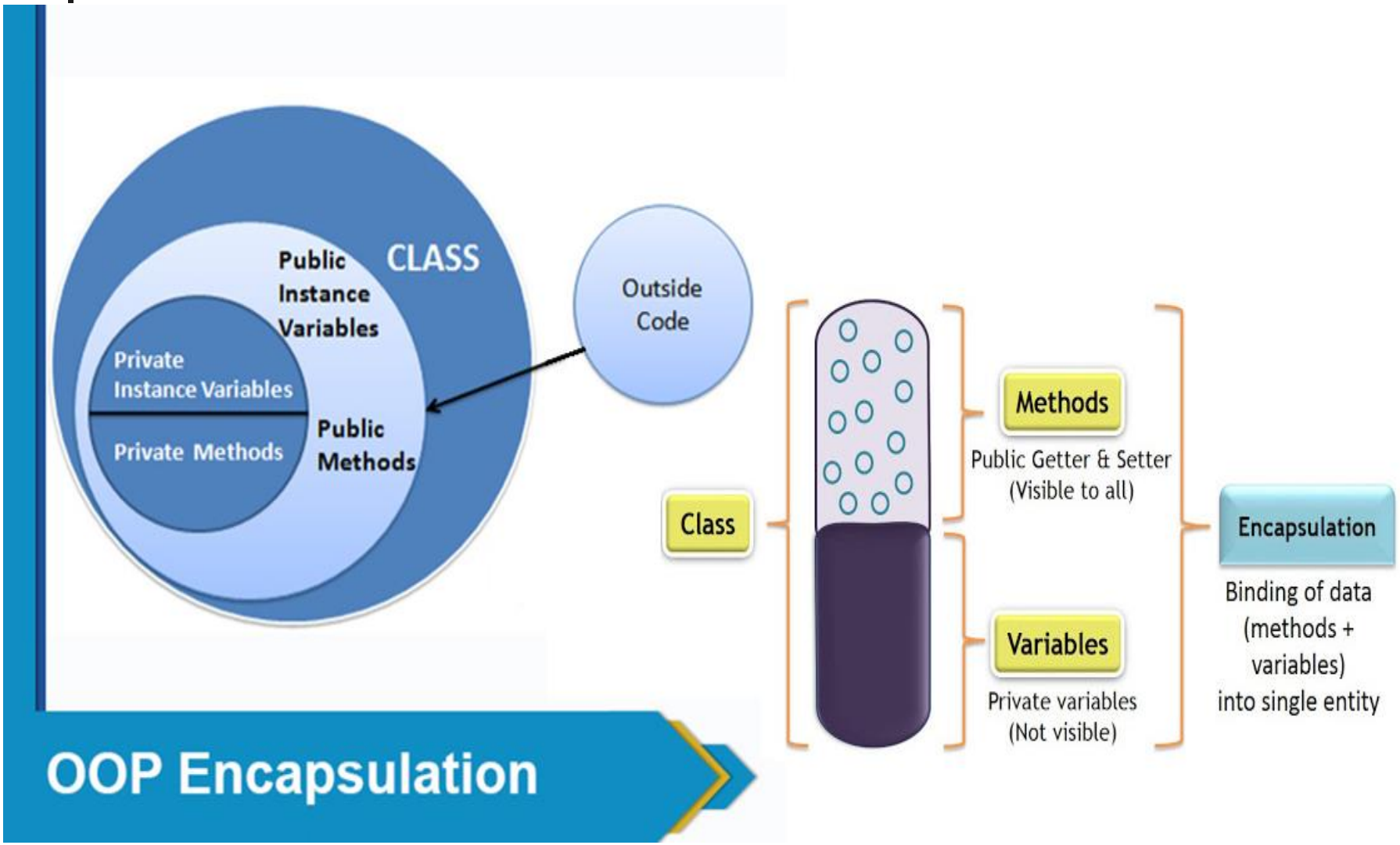
Dog_Object

Dog

Fields
Breed
Size
Colour
Age

Methods
Eat()
Run()
Sleep()
Name()

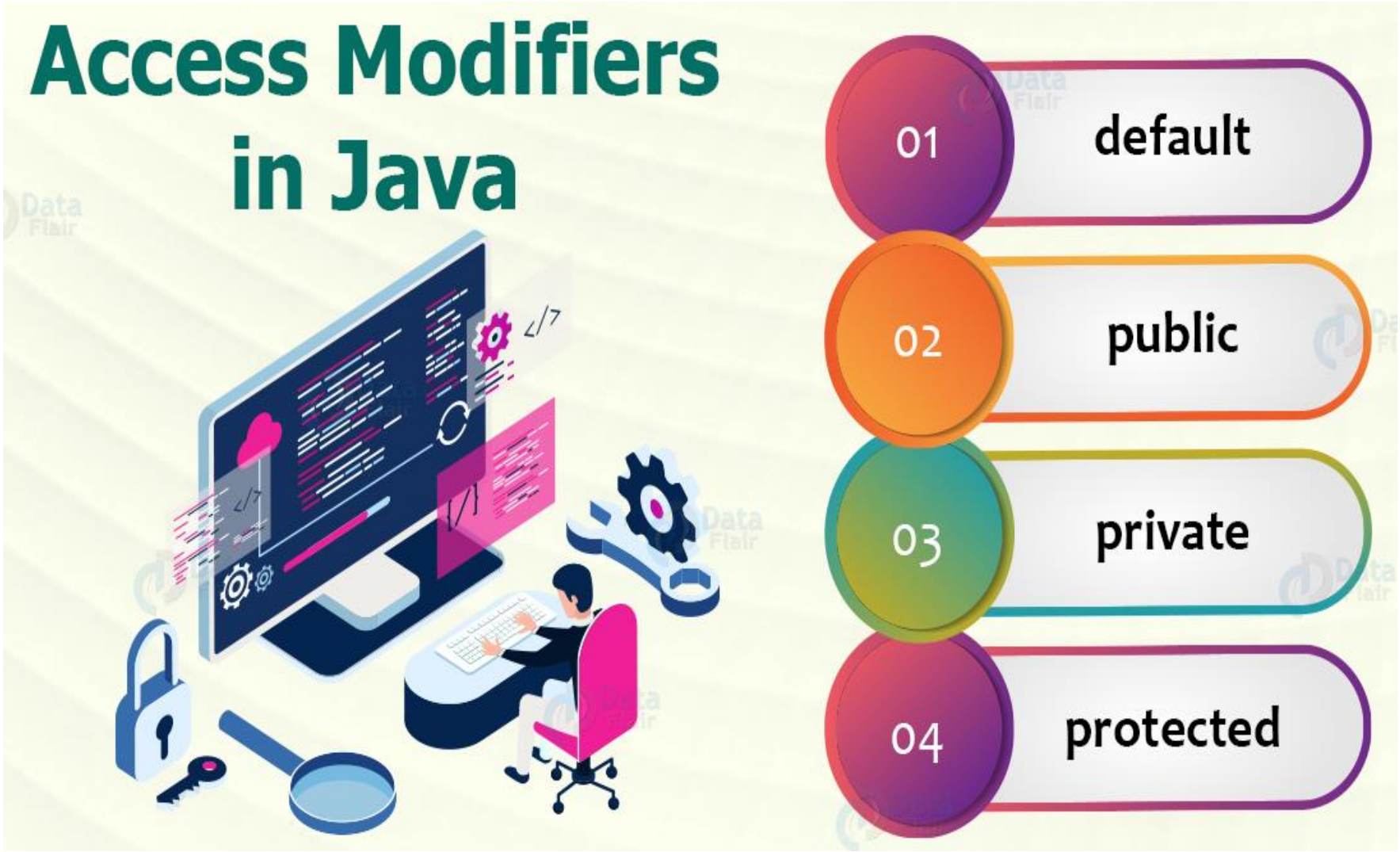
OOP – Encapsulation



OOP – Encapsulation

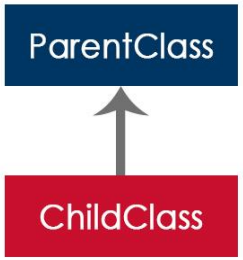
- Data hiding: che giấu việc triển khai bên trong của lớp. Các thuộc tính cũng như một số phương thức được bảo vệ, các lập trình viên chỉ có thể truy cập thông qua các phương thức cho phép.
- Increased flexibility: các thuộc tính của lớp có thể giới hạn ở mức chỉ đọc hoặc chỉ ghi (read, write only) thông qua các getters và setters.
- Reusability: tính đóng gói cho phép chúng ta có thể tái sử dụng và dễ dàng thay đổi khi có yêu cầu mà không cần thay đổi đến các đoạn mã của người khác.
- Easy testing: việc đóng góp dễ dàng cho phép thực hiện unit test – tách dữ liệu và phương thức độc lập.

OOP – Encapsulation

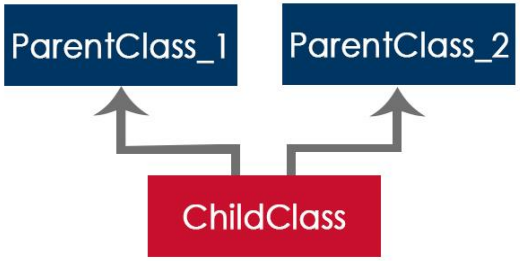


OOP – Inheritance

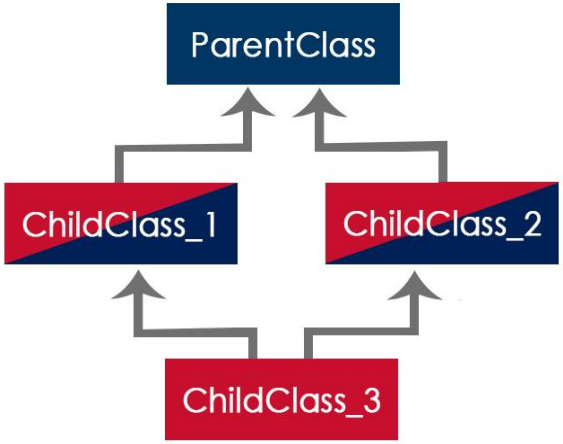
Simple Inheritance



Multiple Inheritance

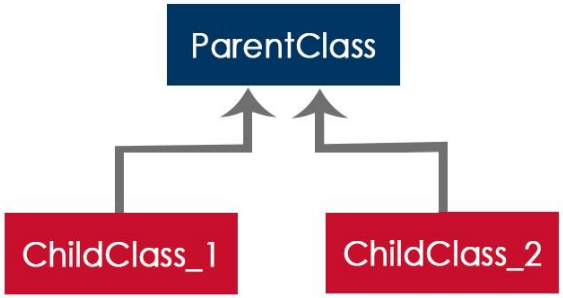


Hybrid Inheritance

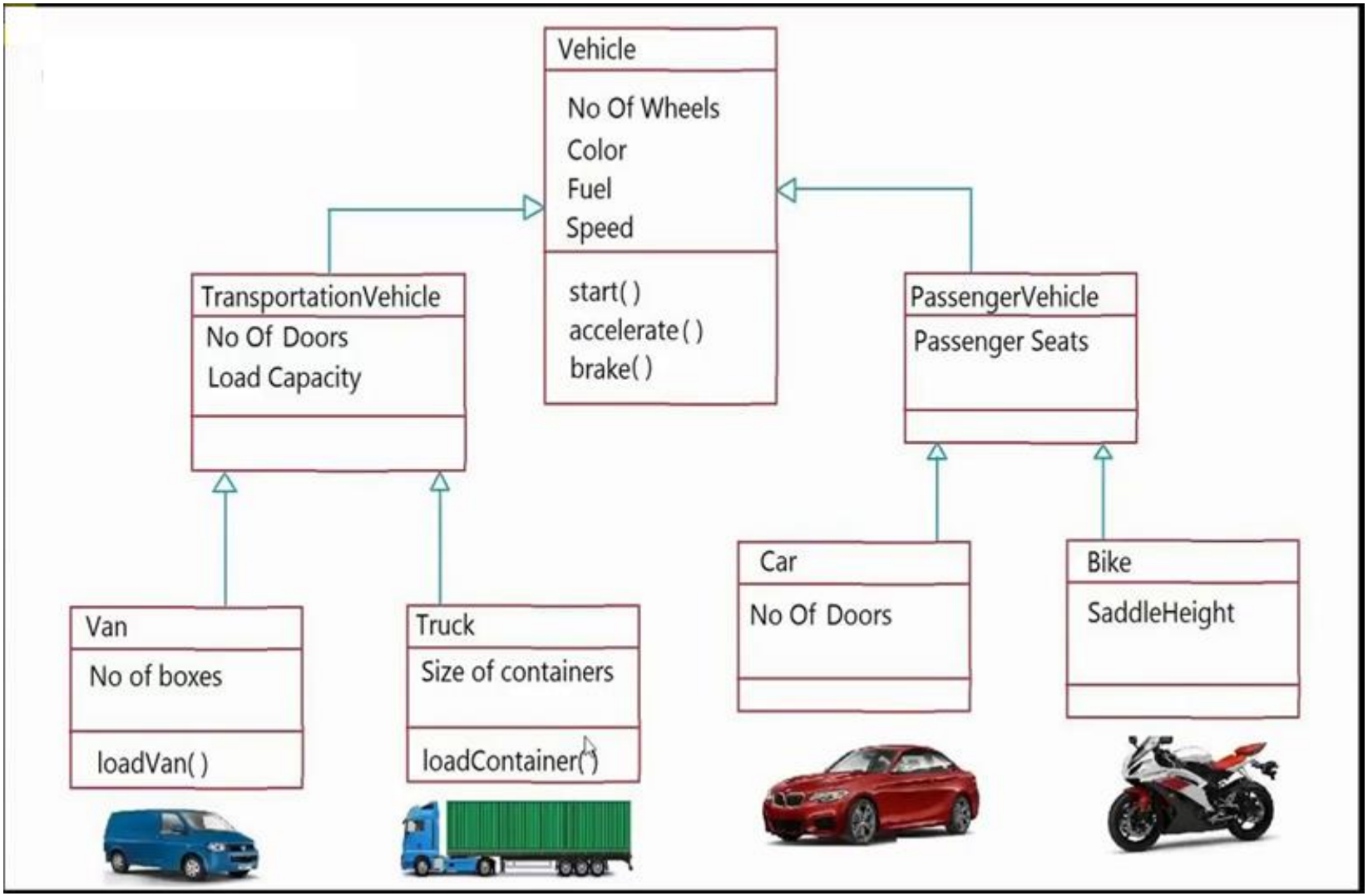


Multi Level Inheritance

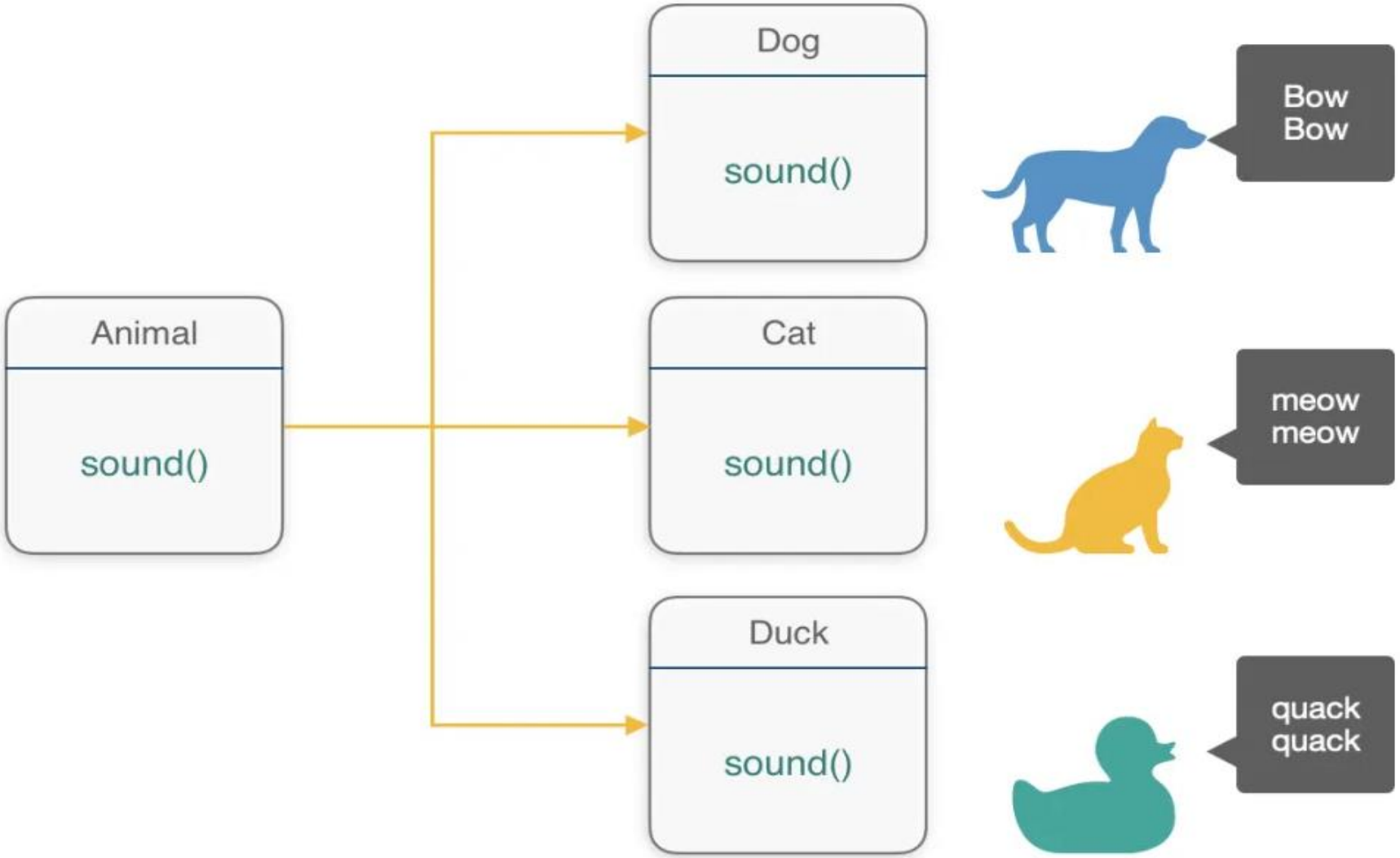
Hierarchical Inheritance



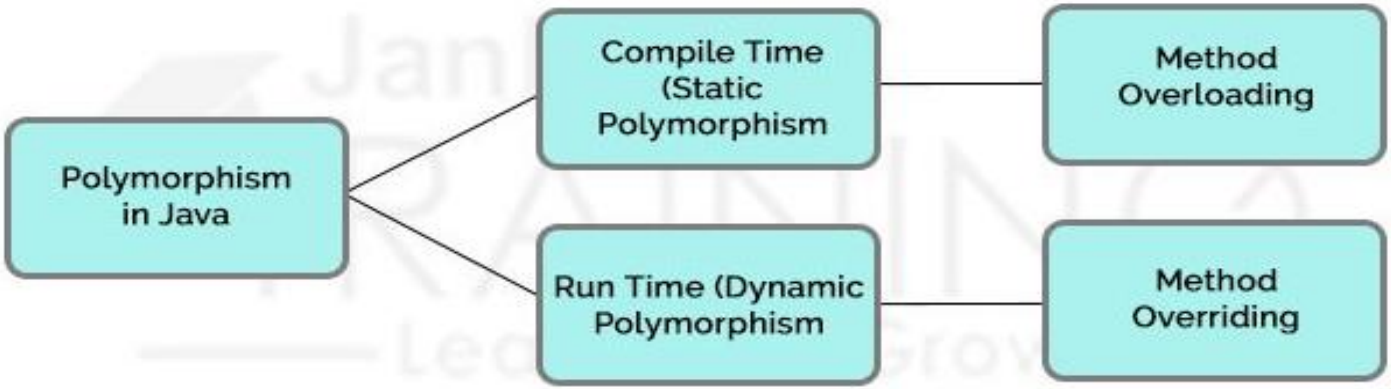
OOP – Inheritance



OOP – Polymorphism



OOP – Polymorphism



Overriding

```
class Dog{
    public void bark(){
        System.out.println("woof ");
    }
}
class Hound extends Dog{
    public void sniff(){
        System.out.println("sniff ");
    }

    public void bark(){
        System.out.println("bowl");
    }
}
```

Same Method Name, Same parameter

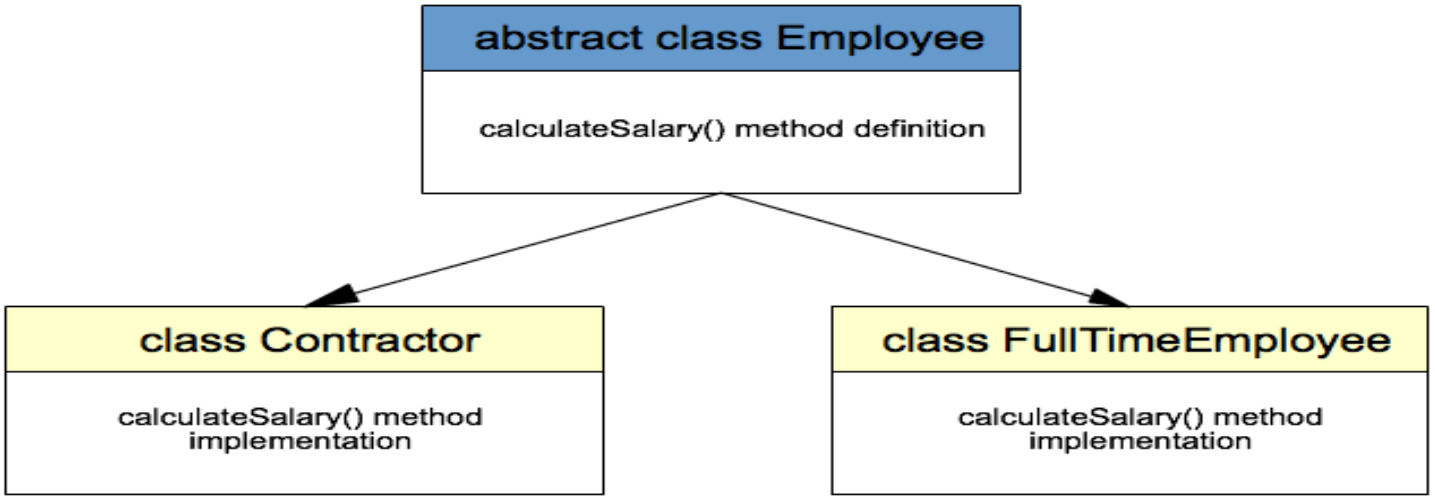
Overloading

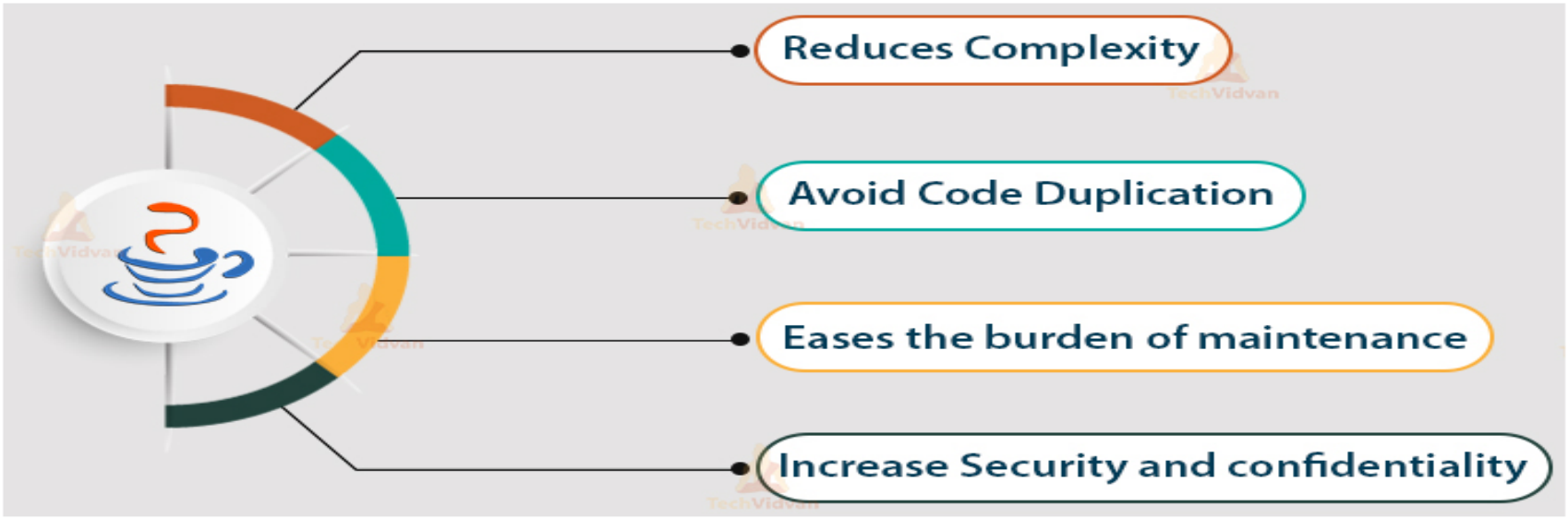
```
class Dog{
    public void bark(){
        System.out.println("woof ");
    }

    //overloading method
    public void bark(int num){
        for(int i=0; i<num; i++)
            System.out.println("woof ");
    }
}
```

Same Method Name, Different Parameter

OOP – Abstraction



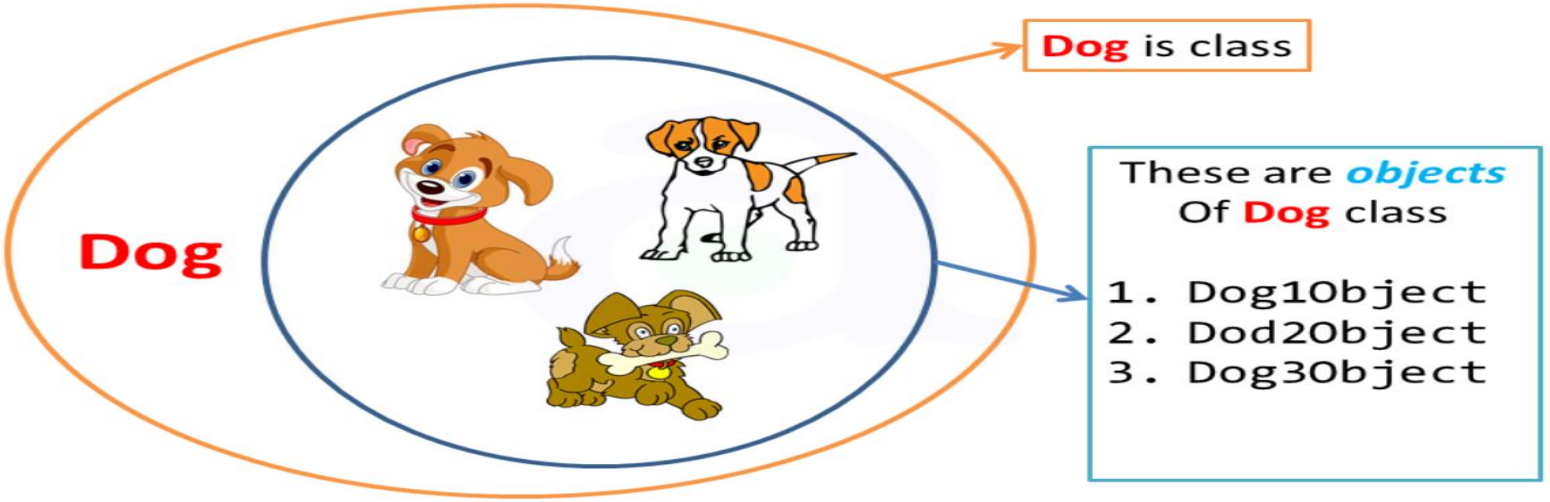


Chương 3:

2. OOP trong Java

Lớp và đối tượng

- Đối tượng – object: là một thực thể có thuộc tính và hành vi nhằm xác định cụ thể.
- Đối tượng: thuộc tính + phương thức.
- Lớp – class: khuôn mẫu chung của một họ các đối tượng cho phép định nghĩa các thuộc tính và các phương thức của họ đối tượng đó cũng như cho phép tạo ra các đối tượng thực thể – instance.

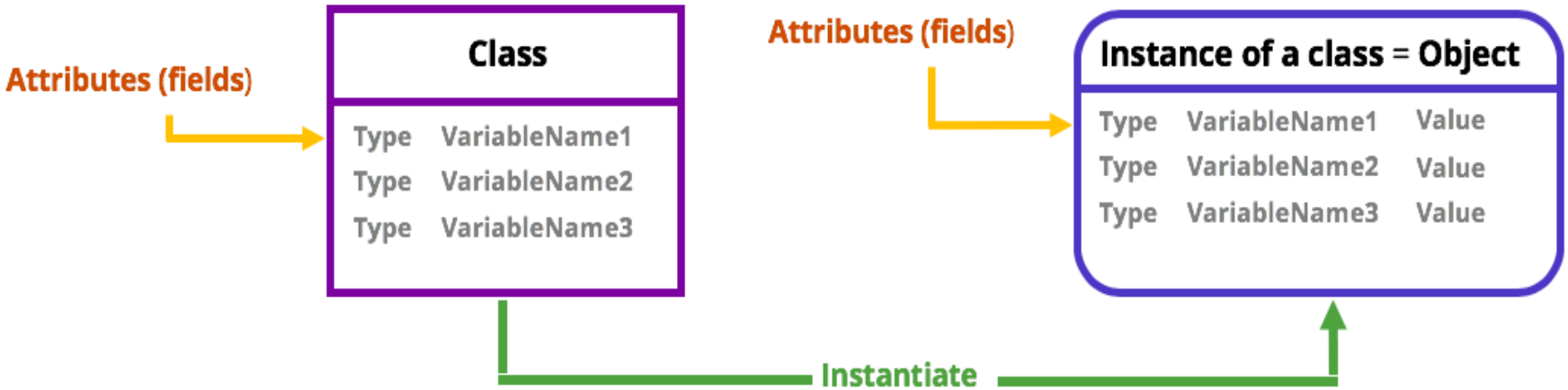


Lớp và đối tượng

No	Đối tượng	Lớp
1.	Đối tượng là thể hiện của một lớp.	Lớp là một khuôn mẫu hay thiết kế để tạo ra các đối tượng trong cùng một nhóm.
2.	Đối tượng là một thực thể trong thực tiễn.	Lớp là một nhóm các đối tượng tương tự nhau.
3.	Đối tượng là một thực thể vật lý	Lớp là một thực thể logic
4.	Đối tượng được tạo ra chủ yếu từ từ khóa new. *Student s=new Student();*	Lớp được khai báo bằng việc sử dụng từ khóa class. *class Student{}*
5.	Đối tượng có thể được tạo nhiều lần.	Lớp được khai báo một lần duy nhất.
6.	Đối tượng được cấp bộ nhớ khi nó được tạo ra.	Lớp không được cấp bộ nhớ khi nó được tạo ra.

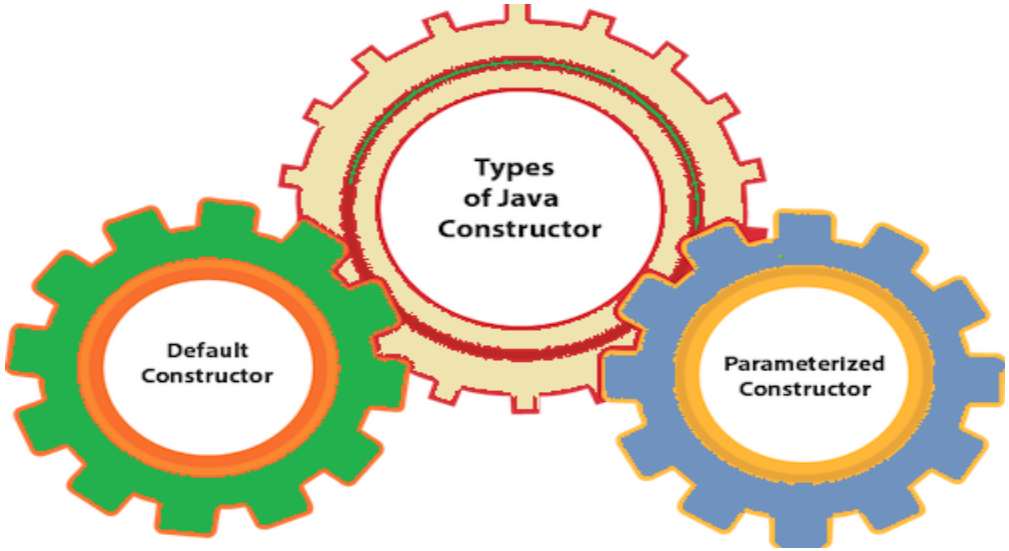
Lớp và đối tượng – thuộc tính

- Thuộc tính cho phép định danh các đặc tính của một lớp và sẽ nhận giá trị ứng với mỗi đối tượng thực thể được tạo ra từ lớp.
- Thuộc tính: kiểu dữ liệu + tên thuộc tính + giá trị mặc định.
- Thuộc tính của đối tượng thường là ***private*** và được xác định thông thường qua hàm tạo, hàm setters.



Lớp và đối tượng – hàm tạo

- Là phương thức đặc biệt cho phép khởi tạo một đối tượng từ lớp tương ứng.
- Hàm tạo có tên trùng với tên lớp và không có kiểu dữ liệu trả về.
- Hàm tạo: mặc định, không đối, có đối.
- Từ hàm tạo, để tạo đối tượng ta có thể dùng toán tử ***new***.



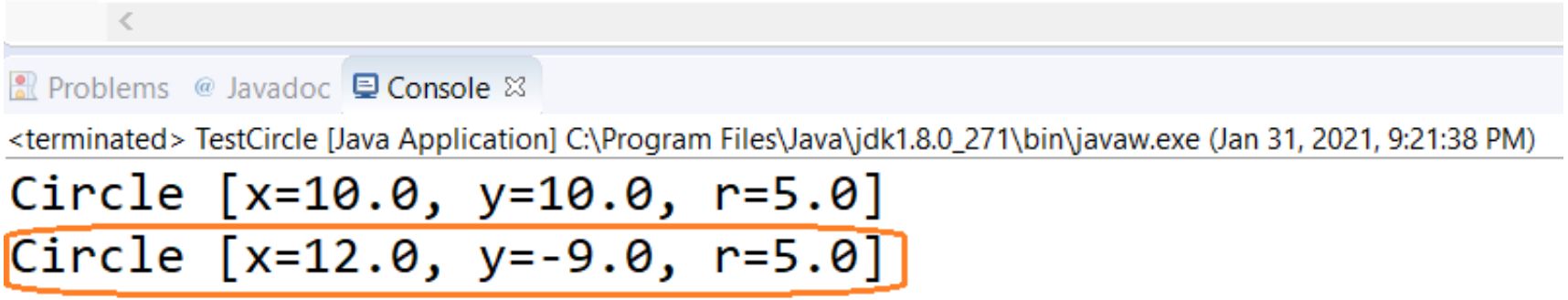
Lớp và đối tượng – ví dụ

```
public class Circle {
    /*
     * Attributes
     */
    private double x;
    private double y;
    private double r;

    /*
     * Methods - constructors
     */
    public Circle(double x, double y, double r) {
        super();
        this.x = x;
        this.y = y;
        this.r = r;
    }
```

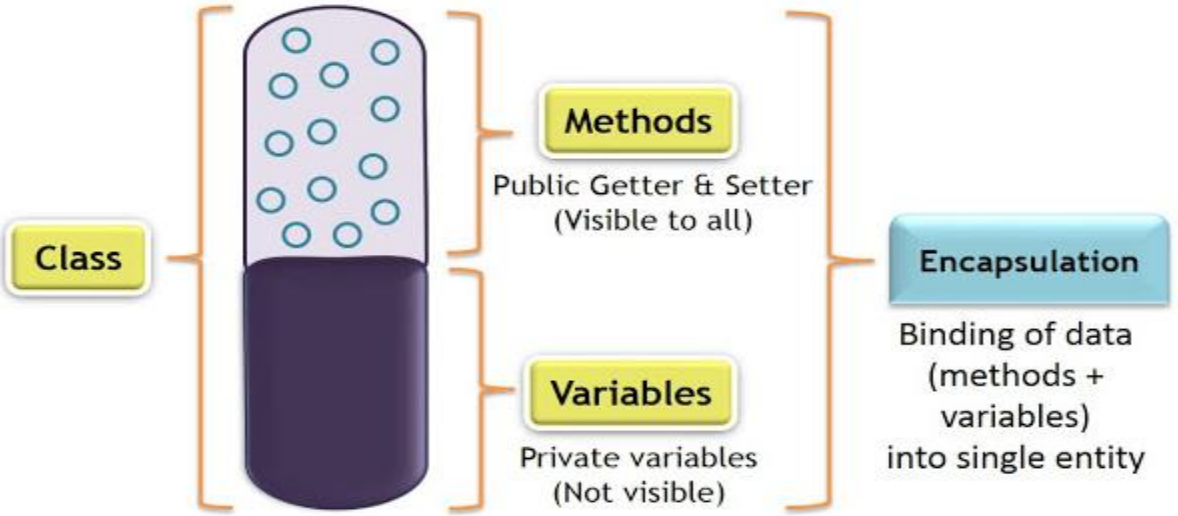
Lớp và đối tượng – ví dụ

```
public class TestCircle {
    public static void main(String[] args) {
        Circle c1 = new Circle(10.0, 10.0, 5.0);
        System.out.println(c1);
        Circle c2 = new Circle(12.0, -9.0, 5.0);
        System.out.println(c2);
    }
}
```



Lớp và đối tượng – getters & setters

- Trong tiếp cận hướng đối tượng, các thuộc tính được bảo vệ. Để truy xuất các thuộc tính, chúng ta sử dụng các phương thức getters và setters.
- Thông thường, các getters và setters được định nghĩa cho mỗi thuộc tính và thường là ***public***.
- ***Chú ý:*** phạm vi truy xuất có thể quyết định tính read/write only của đối tượng.



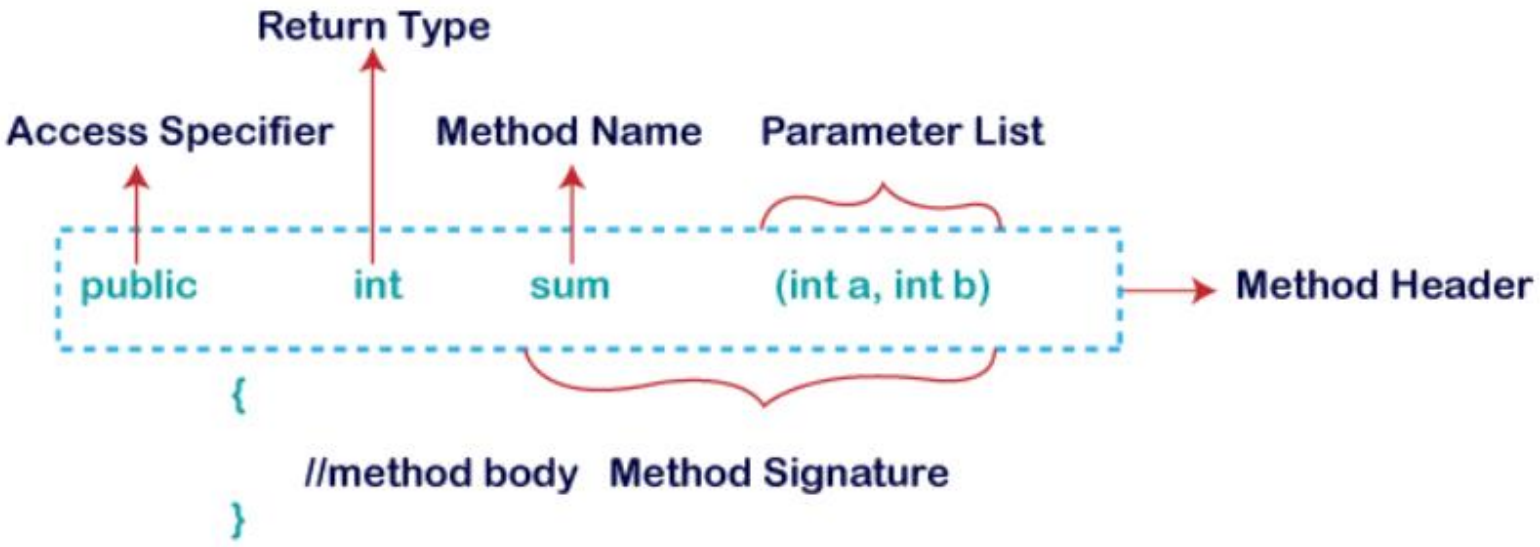
Lớp và đối tượng – ví dụ

```
/*
 * Methods - getters and setters
 */
public double getX() {
    return x;
}

public void setX(double x) {
    this.x = x;
}
/*
 * Methods - toString
 */
@Override
public String toString() {
    return "Circle [x=" + x + ", y=" + y + ", r=" + r + "]";
}
```

Lớp và đối tượng – methods

- Phương thức cho phép định nghĩa hành vi của đối tượng và được thực thi khi có lời gọi từ đối tượng.
- Mỗi đối tượng có thể định nghĩa các phương thức để từ đó có thể đáp ứng yêu cầu bài toán.
- ***Chú ý:*** phạm vi truy xuất thông thường của phương thức là ***public*** hoặc ***protected***.



Lớp và đối tượng – ví dụ

```
/*
 * Methods - others
 */
public double area(){
    return Math.PI * this.r * this.r;
}

public void move(double dx, double dy){
    this.x += dx;
    this.y += dy;
}
```

```
public class TestCircle {
    public static void main(String[] args) {
        Circle c1 = new Circle(10.0, 10.0, 5.0);
        System.out.println(c1);
        double S = c1.area();
        System.out.println("S = " + S);
        c1.move(15.0, 5.0);
        System.out.println(c1);
    }
}
```

```
Circle [x=10.0, y=10.0, r=5.0]
S = 78.53981633974483
Circle [x=25.0, y=15.0, r=5.0]
```

Bài tập – 1

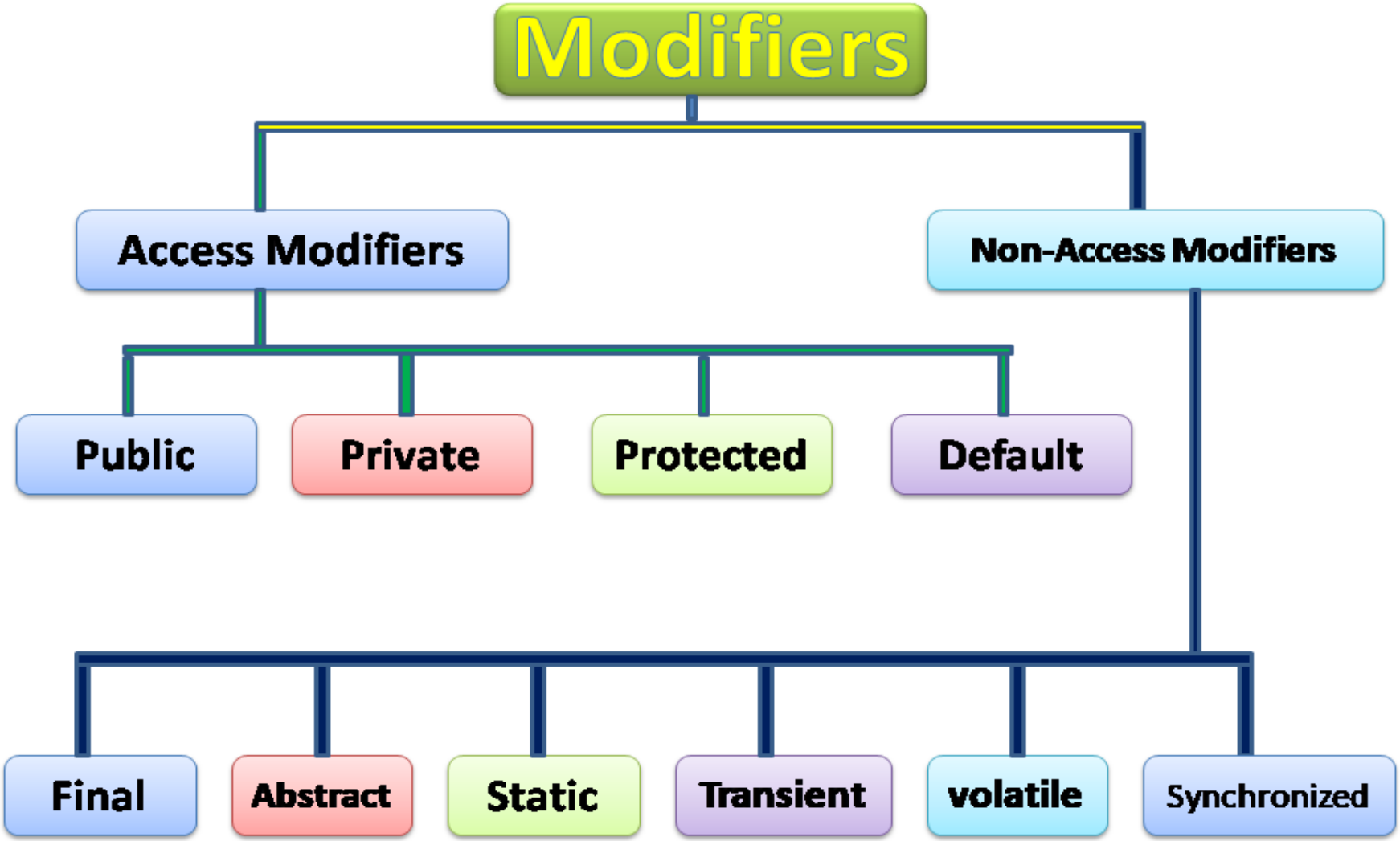
- **Class – object**

1. Mỗi phân số P được định nghĩa bằng 2 thành phần là tử số T và mẫu số M và được biểu diễn P = T/M trong đó tử số và mẫu số là các giá trị nguyên. Hãy xây dựng lớp phân số.
   - Định nghĩa các thuộc tính của lớp phân số.
   - Xây dựng các hàm tạo: mặc định, có đối.
   - Xây dựng các hàm getters, setters và hàm toString.
   - Xây dựng hàm riêng:
     - Tính tổng, hiệu, tích và thương của 2 phân số.
     - So sánh 2 phân số.
     - Rút gọn phân số.
2. Tạo một tập hợp các phân số. Hãy thực hiện chương trình:
   - Tìm phân số lớn nhất trong tập phân số.
   - Tính tổng các phân số.
   - Xác định số phân số không phải số nguyên.

Đặc tính Modifiers

- Modifiers là một đặc tính trong tính đóng gói của OOP, nó có thể quyết định phạm vi truy xuất đến thuộc tính, phương thức của một đối tượng hay phạm vi can thiệp, mở rộng đối với một lớp, một giao diên hay một phương thức.
- Modifier bao gồm:
  - Access modifier: đặc tính truy xuất – xác định phạm vi truy xuất đến các thuộc tính, phương thức.
    - public, default, protected, private
  - Non-access modifier: đảm bảo các đặc tính khác của lớp, thuộc tính, phương thức, etc…
    - static: định nghĩa các thuộc tính và phương thức lớp – biến/hàm toàn cục.
    - final: định nghĩa hằng số, không cho phép override hay kế thừa.
    - abstract: định nghĩa lớp ảo, phương thức ảo.
    - etc …

Đặc tính Modifiers

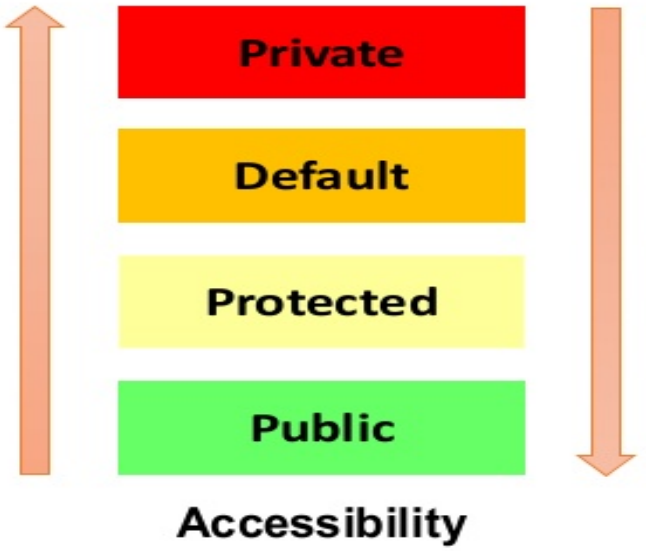


Đặc tính Modifiers

element / modifier	Data field	Method	Constructor	Class		Interface	
				top level (outer)	nested (inner)	top level (outer)	nested (inner)
abstract	no	yes	no	yes	yes	yes	yes
final	yes	yes	no	yes	yes	no	no
native	no	yes	no	no	no	no	no
private	yes	yes	yes	no	yes	no	yes
protected	yes	yes	yes	no	yes	no	yes
public	yes	yes	yes	yes	yes	yes	yes
static	yes	yes	no	no	yes	no	yes
synchronized	no	yes	no	no	no	no	no
transient	yes	no	no	no	no	no	no
volatile	yes	no	no	no	no	no	no
strictfp	no	yes	no	yes	yes	yes	yes

Đặc tính truy xuất – access modifiers

	ACCESS LEVELS			
MODIFIER	Class	Package	Subclass	Everywhere
public	Y	Y	Y	Y
protected	Y	Y	Y	N
default	Y	Y	N	N
private	Y	N	N	N



Đặc tính truy xuất – access modifiers

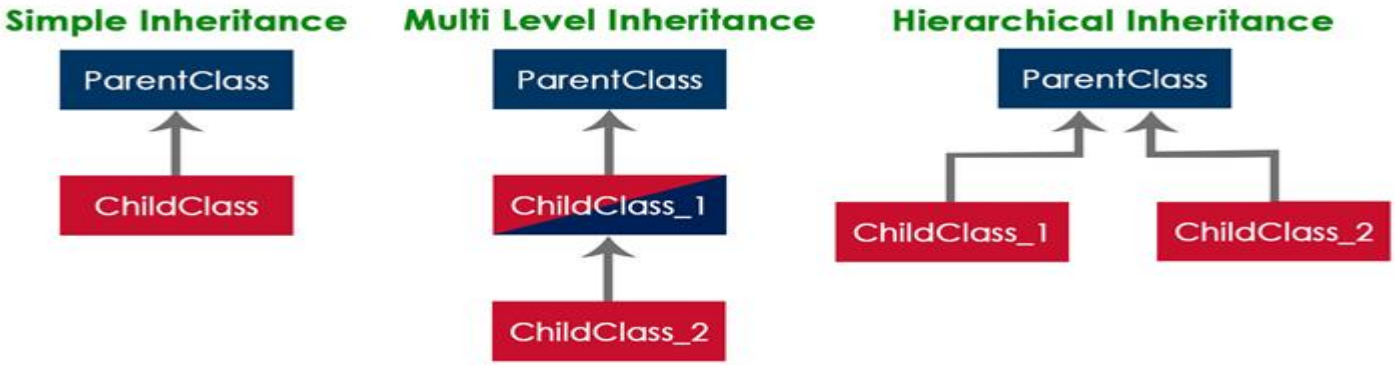
```
public class Circle {
    /*
     * Attributes
     */
    private double x;
    private double y;
    private double r;
    /*
     * Methods - constructors
     */
    public Circle(double x, double y, double r) {
        super();
        this.x = x;
        this.y = y;
        this.r = r;
    }
    /*
     * Methods - getters and setters
     */
    public double getX() {
        return x;
    }
    public void setX(double x) {
        this.x = x;
    }
```

Kế thừa trong Java

- Kế thừa là một trong các đặc tính quan trọng nhất trong OOP mà ở đó, lớp con khi kế thừa lớp cha sẽ được:
  - ***Thừa hưởng toàn bộ các thuộc tính và các phương thức*** của lớp cha.
  - ***Định nghĩa thêm các thuộc tính và các phương thức*** của riêng nó.
  - ***Cho phép định nghĩa lại các phương thức*** phù hợp với thế hệ của lớp con.
- Tư tưởng của kế thừa đó là xây dựng mối quan hệ cha-con trong thực tiễn ➔
  - Tránh phải thực hiện code lại – ***reusability***.
  - Cho phép tính mở của chương trình – các thay đổi chỉ tác động đến lớp con, không ảnh hưởng đến lớp cha cũng như các phần cốt lõi của chương trình – ***extensibility***.
  - Dễ dàng tiếp cận cho việc thiết kế ban đầu và mở rộng thiết kế sau này – ***designability***.

Kế thừa trong Java

- Trong Java, chỉ hỗ trợ đơn kế thừa mà không hỗ trợ đa kế thừa. Đa kế thừa trong Java được thiết kế dựa trên khái niệm ***interface***.



- Cú pháp của kế thừa trong Java

```
public sub_class extends super_class{
  …
}
```

  - ***Chú ý***: *việc truy xuất vào thuộc tính và phương thức của lớp cha phụ thuộc vào access modifier mà lớp cha định nghĩa.*

Kế thừa trong Java – ví dụ

```
package week_04;

public class Point2D {
    private double x;
    private double y;
    public Point2D(double x, double y) {
        this.x = x;
        this.y = y;
    }
    public double getX() {
        return x;
    }
    public void setX(double x) {
        this.x = x;
    }
    public double getY() {
        return y;
    }
    public void setY(double y) {
        this.y = y;
    }
    @Override
    public String toString() {
        return "Point2D [x=" + x + ", y=" + y + "]";
    }
    public double distance(){
        return Math.sqrt(this.x * this.x + this.y * this.y);
    }
}
```

attributes

constructors

getters and setters

methods

Kế thừa trong Java – ví dụ

```
package week_04;

public class Point3D extends Point2D {

    private double z;

    public Point3D(double x, double y, double z) {
        super(x, y);
        this.z = z;
    }

    public double getZ() {
        return z;
    }
    public void setZ(double z) {
        this.z = z;
    }

    @Override
    public String toString() {
        return "Point3D [x="+ this.getX() +
                ", y=" + this.getY() + ", z=" + this.z + "]";
    }

    @Override
    public double distance() {
        return Math.sqrt(super.distance() * super.distance() + this.z * this.z);
    }
}
```

thêm thuộc tính mới

định nghĩa lại hàm tạo cho phù hợp với lớp con

thêm hàm getters và setters cho thuộc tính mới

định nghĩa lại các hàm cho phù hợp với lớp con

Kế thừa trong Java – ví dụ

```
public static void main(String[] args) {
    List<Point2D> l_points = new ArrayList<Point2D>();
    Point2D p_2D_1 = new Point2D(1.0, 5.0);
    Point2D p_2D_2 = new Point2D(3.0, 7.0);
    // Đưa các điểm vào trong danh sách
    l_points.add(p_2D_1);
    l_points.add(p_2D_2);
    // Hiển thị các điểm trong danh sách
    for (Point2D point : l_points) {
        System.out.println(point.toString());
    }
    // Tính tổng khoảng cách.
    double sum = 0.0;
    for (Point2D point : l_points) {
        sum += point.distance();
    }
    System.out.println(sum);

}
```

```
Point2D [x=1.0, y=5.0]
Point2D [x=3.0, y=7.0]
12.714792619456693
```

Kế thừa trong Java – ví dụ

```
public static void main(String[] args) {
    List<Point2D> l_points = new ArrayList<Point2D>();
    Point2D p_2D_1 = new Point2D(1.0, 5.0);
    Point2D p_2D_2 = new Point2D(3.0, 7.0);
    // Đưa các điểm vào trong danh sách
    l_points.add(p_2D_1);
    l_points.add(p_2D_2);
    // Mở rộng chương trình, tạo thêm các điểm 3D
    Point3D p_3D_1 = new Point3D(1.0, 5.0, 3.5);
    Point3D p_3D_2 = new Point3D(3.0, 7.0, 2.0);
    // Đưa các điểm mới vào trong danh sách
    l_points.add(p_3D_1);
    l_points.add(p_3D_2);

    // Hiển thị các điểm trong danh sách
    for (Point2D point : l_points) {
        System.out.println(point.toString());
    }
    // Tính tổng khoảng cách.
    double sum = 0.0;
    for (Point2D point : l_points) {
        sum += point.distance();
    }
    System.out.println(sum);
}
```

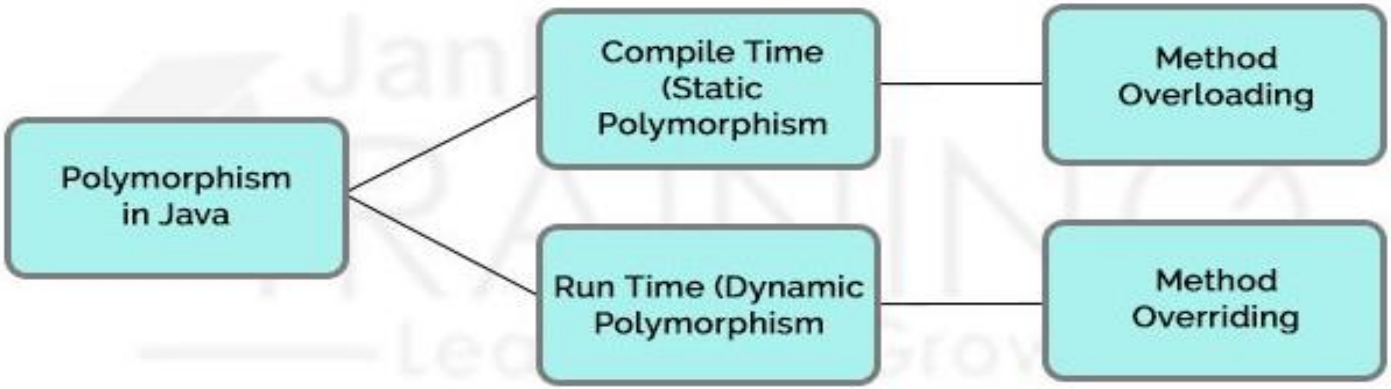
Không cần phải thay đổi code, việc kế thừa giúp cho phần tính toán phù hợp với các đối tượng cha - con.

```
Point2D [x=1.0, y=5.0]
Point2D [x=3.0, y=7.0]
Point3D [x=1.0, y=5.0, z=3.5]
Point3D [x=3.0, y=7.0, z=2.0]
26.773458931894993
```

Kế thừa trong Java

- ***this***: dùng để tham chiếu đến instance của lớp hiện tại để thông qua đó có thể truy cập thuộc tính và phương thức của đối tượng.
- ***super***: dùng để tham chiếu đến instance của lớp cha gần nhất qua đó có thể truy cập thuộc tính và phương thức của đối tượng cha.
- ***final***:
  - class: không cho phép tạo lớp con dẫn xuất từ lớp này.
  - variable: định nghĩa hằng số.
  - method: không cho phép lớp con override phương thức của lớp cha.
- ***static***: dùng để định nghĩa thuộc tính và phương thức của lớp – khái niệm biến và phương thức toàn cục.

Tính đa hình trong Java



Overriding

```
class Dog{
    public void bark(){
        System.out.println("woof ");
    }
}
class Hound extends Dog{
    public void sniff(){
        System.out.println("sniff ");
    }

    public void bark(){
        System.out.println("bowl");
    }
}
```

Same Method Name, Same parameter

Overloading

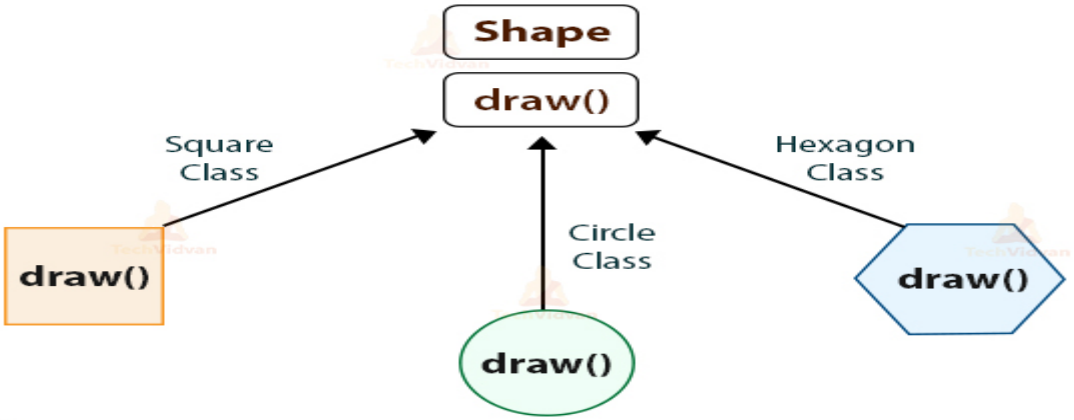
```
class Dog{
    public void bark(){
        System.out.println("woof ");
    }

    //overloading method
    public void bark(int num){
        for(int i=0; i<num; i++)
            System.out.println("woof ");
    }
}
```

Same Method Name, Different Parameter

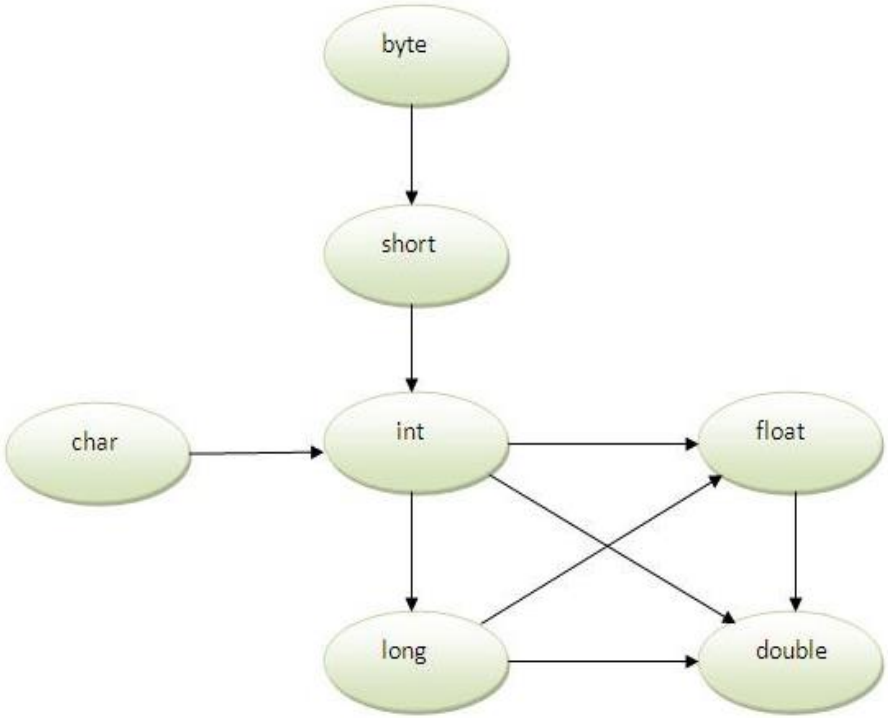
Tính đa hình trong Java

- ***Override***: là khi lớp con thừa kế phương thức của lớp cha nhưng phương thức này không còn phù hợp với lớp con nên lớp con cần định nghĩa lại ➔ tồn tại 2 phương thức trùng tên và tham số: *@Override.*
  - Phương thức override phải có cùng kiểu dữ liệu trả về.
  - Hàm constructors không thể override [i.e. khác gói].
  - *Phạm vi của modifier một phương thức trong lớp con phải nằm trong phạm vi modifier của lớp cha.*
  - Không hỗ trợ cho thuộc tính bị override ➔ luôn lấy giá trị của lớp cha.



Tính đa hình trong Java

- ***Overload***: là khi một lớp có nhiều phương thức trùng tên nhưng khác kiểu tham số hoặc số lượng tham số.
  - Tránh phải thay đổi tên phương thức khi các phương thức có cùng mục đích.
  - ***Chú ý***: thay đổi kiểu dữ liệu trả về không phải là overload ➔ compile error.
  - Java tự động ép kiểu:



Tính đa hình trong Java – ví dụ

```
public class Point2D {
    private double x;
    private double y;
    public double distance() {
        return Math.sqrt(this.x * this.x + this.y * this.y);
    }
}
public class Point3D extends Point2D {
    private double z;
    @Override
    public double distance() {
        return Math.sqrt(super.distance() * super.distance() + this.z * this.z);
    }
    //@Overload
    public double distance(double x, double y, double z) {
        double dx = (this.getX()- x) * (this.getX()- x);
        double dy = (this.getY()- y) * (this.getY()- y);
        double dz = (this.z- z) * (this.z - z);
        return Math.sqrt(dx * dx + dy * dy + dz * dz);
    }
}
```

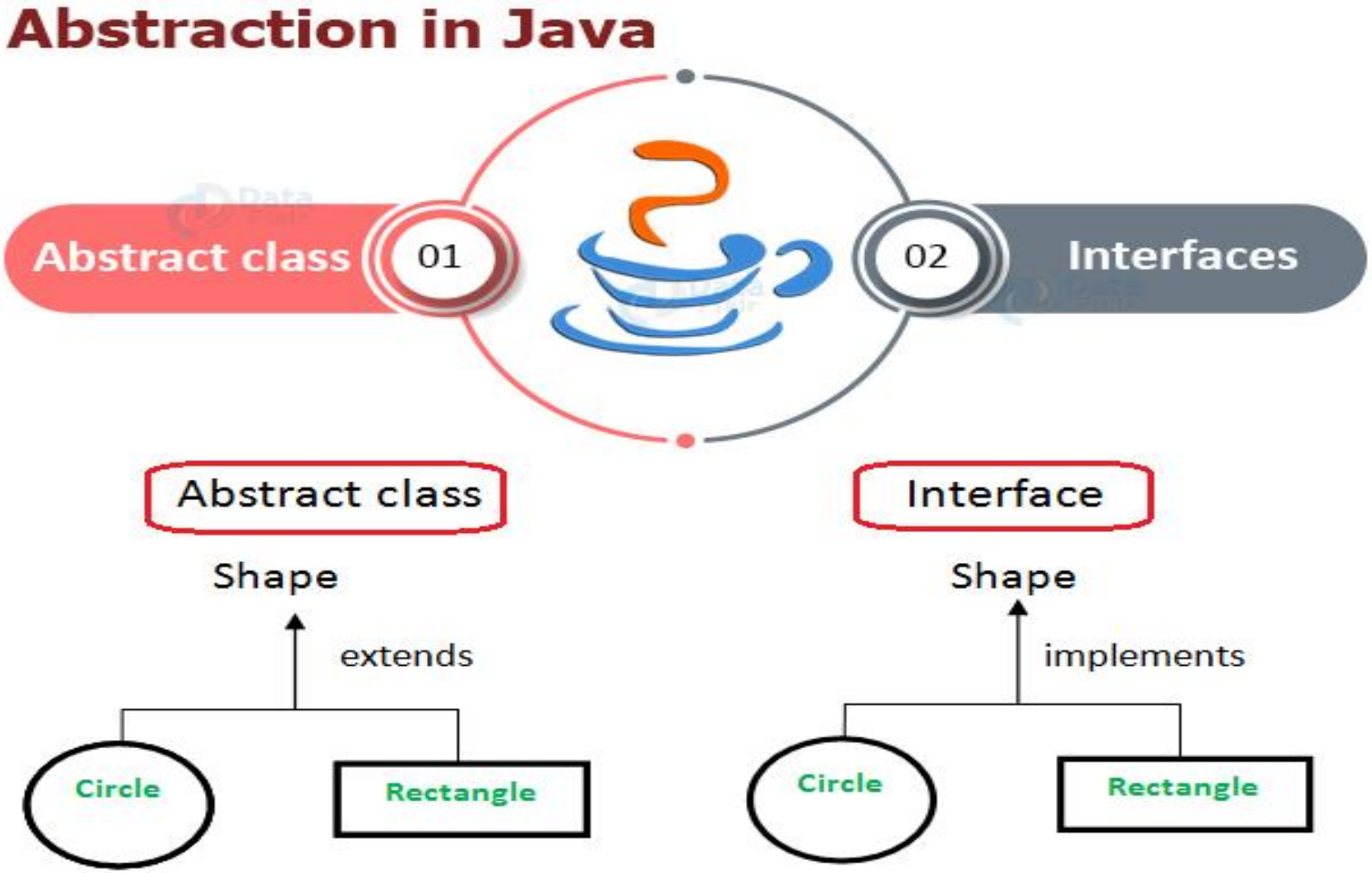
Bài tập – 2

- **Class – inheritance/modifiers**

1. Hoàn thành lớp điểm 2D với các phương thức:
   - Tính khoảng cách giữa 2 điểm.
   - Xác định điểm đối xứng qua gốc tọa độ.
   - Tịnh tiến điểm đi một vị trí với độ lệch Δx, Δy.
2. Hoàn thành lớp điểm 3D kế thừa từ lớp điểm 2D:
   - Tính khoảng cách giữa 2 điểm 3D.
   - Định nghĩa lại hàm xác định điểm đối xứng qua gốc tọa độ.
   - Tịnh tiến điểm đi một vị trí với độ lệch Δx, Δy và Δz.
3. Xây dựng chương trình chính:
   - Tạo một danh sách điểm bao gồm 2D và 3D.
   - Tính tổng khoảng cách các điểm 2D và tổng khoảng cách các điểm 3D.
   - Đưa các điểm đối xứng vào danh sách và hiển thị danh sách các điểm.

- Chú ý:
  - Sử dụng các khái niệm kế thừa, cũng như access modifier đối với các thuộc tính và phương thức.

Tính trừu tượng trong Java

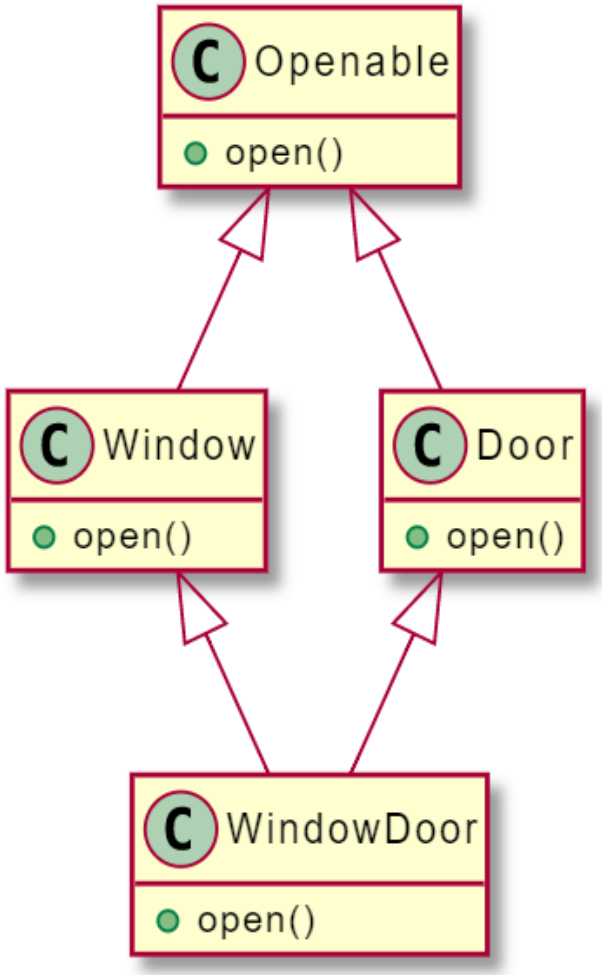


Tính trừu tượng trong Java

- **Mục đích:**
  - Cho phép ẩn đi các cài đặt chi tiết, chỉ hiển thị cách thức sử dụng đối với người dùng ➔ giúp cho các lập trình viên cài đặt các nghiệp vụ phức tạp dựa trên các hành vi mà đối tượng được cung cấp.
  - Ví dụ:
    - Khi pha cà phê, chúng ta chỉ quan tâm đến để có thể tạo ra cốc café thì cần lựa chọn loại cafe, độ đậm nhạt, liều lượng còn việc để vận hành như thế nào chúng ta không cần quan tâm.
    - Ứng dụng: các frameworks, các libraries cung cấp cách tiếp cận trừu tượng cho phép chúng ta dựa trên đó để phát triển các ứng dụng riêng.
  - Trong Java, tiếp cận trừu tượng dựa trên kế thừa với:
    - Abstract class
    - Interface

Tính trừu tượng trong Java

- **Đa kế thừa – vấn đề gặp phải**
  - ***Diamond problem***: phương thức open() mà lớp con WindowDoor được kế thừa từ nhiều lớp cha ➔ lựa chọn phương thức của lớp cha nào để kế thừa?
  - Trong C++:
    - Yêu cầu lớp cha khai báo các phương thức open() là virtual (abstract) ➔ lớp con phải định nghĩa lại.
    - Làm thế nào tạo instance cho lớp cha? ➔ **pure virtual** vs **virtual**.
  - Trong Java:
    - Nhóm các phương thức pure virtual ➔ **interface**.
    - Các phương thức virtual được giữ nguyên ➔ **abstract**.



Tính trừu tượng trong Java

- **Abstract**
  - Là một class trong Java trong đó có chứa ít nhất một phương thức trừu tượng: phương thức này được định nghĩa prototype, phần cài đặt sẽ được thực hiện trong lớp dẫn xuất.

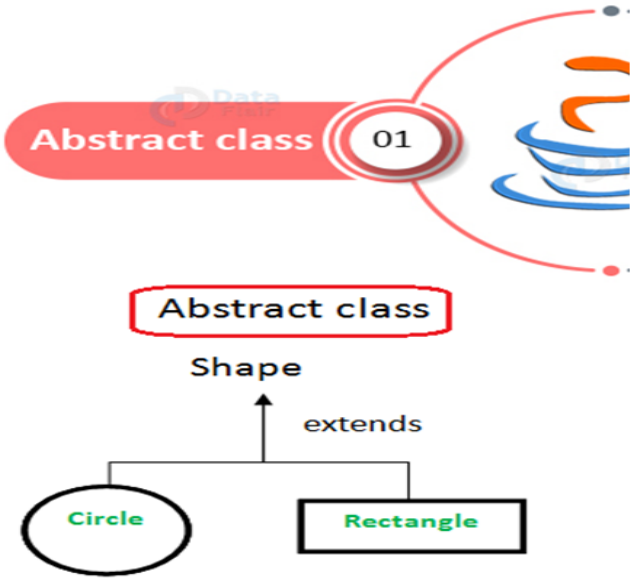
```
package week_06;

public abstract class Shape {
    abstract void draw();

    public static void main(String[] args) {
        Rectangle r = new Rectangle();
        Circle c = new Circle();
        r.draw();
        c.draw();
    }

    // Example - abstract [use the inner class]
    public static class Circle extends Shape {
        void draw() {
            System.out.println("I'm circle!");
        }
    }

    public static class Rectangle extends Shape {
        void draw() {
            System.out.println("I'm rectangle!");
        }
    }
}
```



Tính trừu tượng trong Java

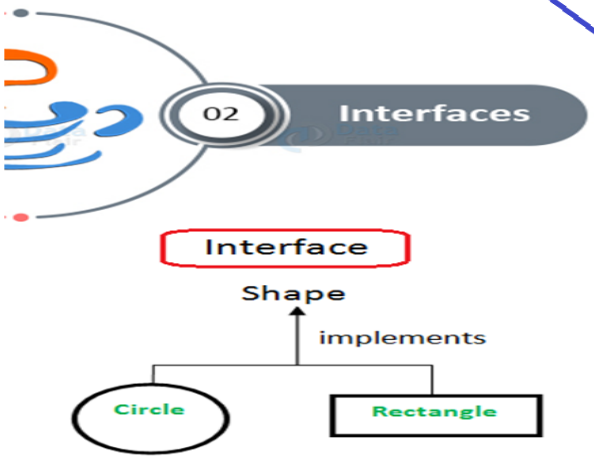
- **Abstract**
  - Lớp abstract là lớp trừu tượng nên không cho phép khởi tạo đối tượng từ lớp abstract.
  - Lớp dẫn xuất của lớp abstract có thể:
    - Là một lớp abstract ➔ chưa cần cài đặt các phương thức abstract.
    - Là một lớp thường ➔ bắt buộc phải cài đặt các phương thức abstract của lớp abstract.
  - Trong Java, chỉ có thể kế thừa một lớp ➔ kế thừa một lớp thường hoặc một lớp abstract.
  - Phạm vi truy cập của phương thức abstract không thể là private vì nó cần được định nghĩa lại trong lớp dẫn xuất.
  - Từ Java 7, lớp abstract có thể có hoặc không có phương thức abstract nhưng lớp abstract không thể tạo ra thực thể dù không có phương thức abstract.

Tính trừu tượng trong Java

- **Interface**
  - Là một cách thể hiện của đa kế thừa trong Java trong đó một interface định nghĩa prototype cho các phương thức – pure abstract: các phương thức này cài đặt sẽ được thực hiện trong lớp dẫn xuất [lớp cài đặt].

```
package week_06;

public interface IShape {
    public void draw();
    public double area();
    public double perimeter();
}
```




```
public class Circle implements IShape {
    protected double x;
    protected double y;
    protected double r;

    public Circle(double x, double y, double r) {
        this.x = x;
        this.y = y;
        this.r = r;
    }

    public void draw() {
        System.out.println("I'm circle!");
    }

    public double area() {
        return this.r * this.r * Math.PI;
    }

    public double perimeter() {
        return this.r * 2.0 * Math.PI;
    }

    public static void main(String[] args) {
        Circle c = new Circle(2.0, 3.0, 5.0);
        c.draw();
        System.out.println("Area: " + c.area());
        System.out.println("Perimeter:" + c.perimeter());
    }
}
```

Tính trừu tượng trong Java

- **Interface**
  - Các phương thức được khai báo trong Interface đều là các phương thức pure abstract ➔ không cần dùng từ khóa abstract.
  - Lớp cài đặt của interface có thể:
    - Là một lớp abstract.
    - Là một lớp thường ➔ bắt buộc phải cài đặt các phương thức abstract của interface.
  - Trong Java, chỉ có thể kế thừa một lớp nhưng hỗ trợ đa cài đặt ➔ kế thừa một lớp nhưng cài đặt nhiều interface.
  - Phạm vi truy cập của phương thức trong interface không thể là private hoặc protected vì nó cần được cài đặt trong lớp cài đặt.
  - Interface không phải là một lớp, chỉ là khai báo các phương thức cho họ các lớp ➔ không thể tạo ra thực thể từ interface.
  - Trong interface không có khái niệm thuộc tính, chỉ có thể định nghĩa các biến/hàm toàn cục thông qua static/final.
  - Một interface có thể kế thừa nhiều interface khác.

Tính trừu tượng trong Java

Abstract	Interface
Abstract class có phương thức **abstract** (không có thân hàm) hoặc phương thức **non-abstract** (có thân hàm).	Interface chỉ có phương thức **abstract**. Từ Java 8, nó có thêm **các phương thức default và static**.
Abstract class **không hỗ trợ đa kế thừa**.	Interface **có hỗ trợ đa kế thừa**
Abstract class có các biến **final, non-final, static and non-static**.	Interface chỉ có các biến **static và final**.
Abstract class **có thể cung cấp nội dung cài đặt cho phương thức của interface**.	Interface **không thể cung cấp nội dung cài đặt cho phương thức của abstract class**.
Từ khóa **abstract** được sử dụng để khai báo abstract class.	Từ khóa **interface** được sử dụng để khai báo interface.

Bài tập – 3

- **Interface – abstract class**

Để xây dựng ứng dụng Paint cho phép người dùng thực hiện các thao tác với các hình bao gồm: Point, Circle, Line, Triangle, Rectangle, … người ta xây dựng nên không gian các hình với cách tiếp cận OOP.

- Xây dựng Interface Shape với các phương thức
  - Nhóm tính toán: diện tích (**area**), chu vi (**perimeter**), khoảng cách (**distance**),..
  - Nhóm biến đổi: tịnh tiến (**move**), xoay (**rotate**), thay đổi (**zoom**), …
- Dựa trên Interface Shape, xây dựng các lớp tương ứng với các đối tượng.

Yêu cầu:

1. Cài đặt *<<interface>>* Shape dựa trên các mô tả yêu cầu
   - Định nghĩa các phương thức của Shape (tính toán + biến đổi).
2. Cài đặt các lớp tương ứng với các phương thức đã được mô tả trong *<<interface>>* Shape.
   - Point, Circle, Line, Triangle, Rectangle …
     - attributes/constructors/getters and setters/toString
     - area/perimeter/distance/move/rotate/zoom

Bài tập – 3

- **Interface – abstract class**

Yêu cầu:

3. Xây dựng chương trình chính:
   - Tạo một tập các hình ngẫu nhiên và đưa vào trong danh sách để quản lý.
   - Thực hiện hiển thị danh sách các hình đã tạo ra.
   - Tính tổng chu vi/diện tích các hình đã tạo ra và tìm hình có diện tích lớn nhất/ nhỏ nhất.
   - Biến đổi các hình bằng cách cho phép phóng to các hình với tỉ lệ ratio.
   - …
- Chú ý:
   - Sử dụng các khái niệm kế thừa, cũng như access modifier đối với các thuộc tính và phương thức.
   - Có thể xây dựng và sử dụng các abstract class nếu cần thiết.
   - Hiểu rõ cách thức khai báo và sử dụng hàm.

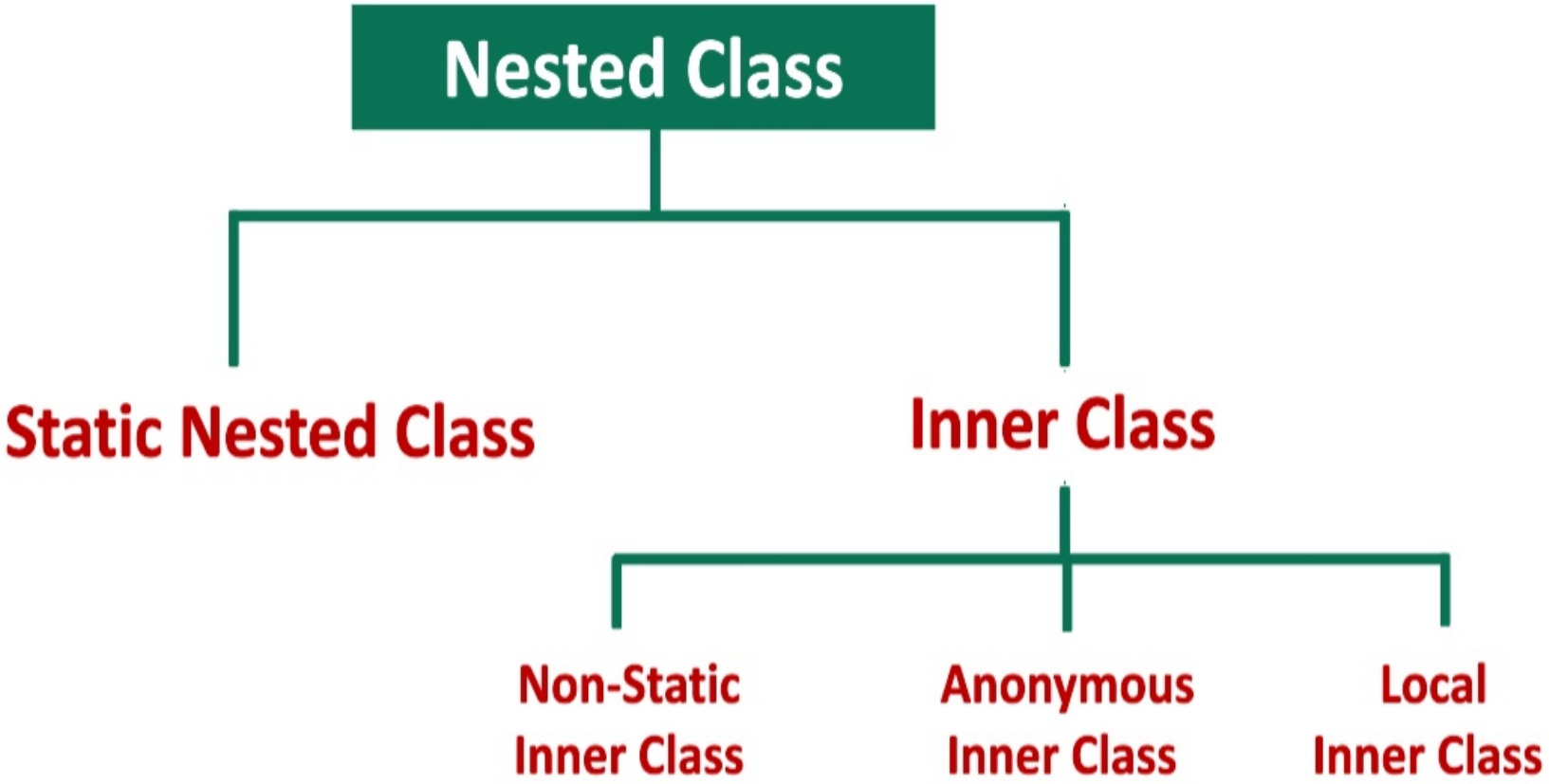
Chương 3:

3. Các khái niệm mở rộng

Inner/Outer class

- **Khái niệm:**
  - Trong Java, cho phép khai báo lớp/interface nằm trong một lớp/interface người ta gọi là lớp lồng nhau ➔ nested class.
  - Lớp chứa được gọi là outer class còn lớp khai báo bên trong được gọi là inner class.
  - Ưu điểm:
    - Nhóm các lớp có cùng logic trong một tệp tin để quản lý.
    - Lớp inner khi đó được coi là một phần của lớp outer nên có thể truy cập đến các thuộc tính/phương thức của lớp outer ngay cả khi các thuộc tính/phương thức là private.
    - Tiết kiệm code trong việc truy xuất đến các thành phần của outer.
  - Nhược điểm:
    - Không sử dụng được lớp inner, các phương thức của nó trong các lớp khác.
    - Inner class tham chiếu đến outer class nên đối tượng tạo ra chỉ được giải phóng khi đối tượng outer được giải phóng ➔ lãng phí bộ nhớ. Không nên dùng các mảng inner.

Inner/Outer class



Inner/Outer class

- **Static nested class**
  - Được coi như một thành viên static của lớp bên ngoài ➔
    - Có thể được truy cập thông qua lớp bên ngoài.
    - Chỉ truy cập vào các thành phần static của lớp bên ngoài.
  - i.e.

```
package week_06;

public class StaticNestedClass {
    private static String data = "I'm static data!";

    // Static inner class
    public static class InnerClass{
        public void show(){
            System.out.println(data);
        }
    }

    public static void main(String[] args) {
        StaticNestedClass.InnerClass s = new StaticNestedClass.InnerClass();
        s.show();
    }
}
```

Inner/Outer class

- **Non static nested class – inner class**
  - Được coi như là 2 lớp khi biên dịch tuy nhiên
    - Được truy cập thông qua lớp bên ngoài.
    - Có thể truy cập vào các thành phần thuộc tính/phương thức của lớp bên ngoài.
  - i.e.

```
package week_06;

public class NonStaticNestedClass {

    private int data = 30;

    public void showOuter(){
        System.out.println("I'm outer class!");
    }

    public class InnerClass {
        private void show() {
            showOuter();
            System.out.println("Data is " + data);
        }
    }

    public static void main(String args[]) {
        NonStaticNestedClass outer = new NonStaticNestedClass();
        NonStaticNestedClass.InnerClass inner = outer.new InnerClass();
        inner.show();
    }
}
```

Inner/Outer class

- **Local inner class**
  - Lớp inner được khai báo trong một phương thức của lớp outer. Khi đó lớp inner sẽ không thể được gọi từ bên ngoài.
  - i.e.

```
package week_06;

public class LocalInnerClass {

    private int data = 30;

    void showOuter(){
        System.out.println("I'm outer class");
    }

    void display() {
        class InnerClass {
            void show() {
                showOuter();
                System.out.println(data);
            }
        }
        InnerClass inner = new InnerClass();
        inner.show();
    }

    public static void main(String args[]) {
        LocalInnerClass obj = new LocalInnerClass();
        obj.display();
    }
}
```

Inner/Outer class

- **Anonymous inner class**
  - Lớp inner được tạo ra trong quá trình cài đặt các phương thức ảo của một abstract class hoặc interface.
  - i.e.

```
package week_06;

public class AnonymousClass {
    public static void main(String[] args) {
        IShape s = new IShape() {
            private double r = 5;

            public void draw() {
                System.out.println("I'm anonymous class - circle.");
            }

            public double area() {
                return Math.PI * r * r;
            }

            public double perimeter() {
                return Math.PI * 2.0 * r;
            }
        };
        s.draw();
        System.out.println(s.area());
        System.out.println(s.perimeter());
    }
}
```

Debug trong Java với Eclipse

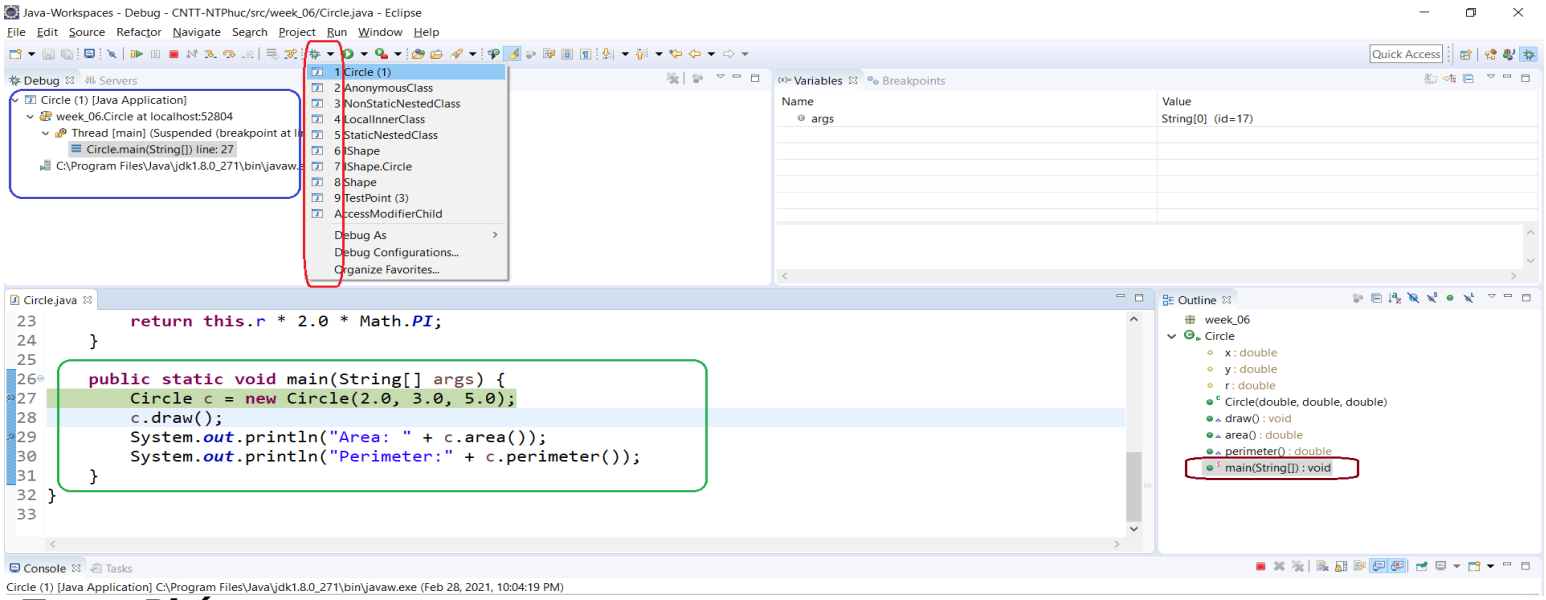
- **Debug**
  - Là chế độ chạy từng bước cho phép lập trình viên có thể theo dõi và gỡ lỗi trong chương trình.
  - Chế độ này cho phép:
    - Chạy từng dòng lệnh theo ý muốn.
    - Theo dõi giá trị của mỗi biến tại mỗi thời điểm.
    - Theo dõi lời gọi hàm tại mỗi vị trí cần gỡ lỗi.
  - Ứng dụng
    - Được sử dụng bởi các lập trình viên khi quan sát và gỡ lỗi thay cho việc dùng câu lệnh System.out.println().
    - Đều được hỗ trợ bởi tất cả các IDE: đặt điểm dừng, chạy chế độ debug, quan sát các giá trị biến, …
    - Eclipse hỗ trợ đầy đủ các thao tác cho phép lập trình viên dễ dàng chạy và kiểm soát trong chế độ debug.

Debug trong Java với Eclipse

1. Đặt điểm breakpoint (Ctrl+Shift+B)

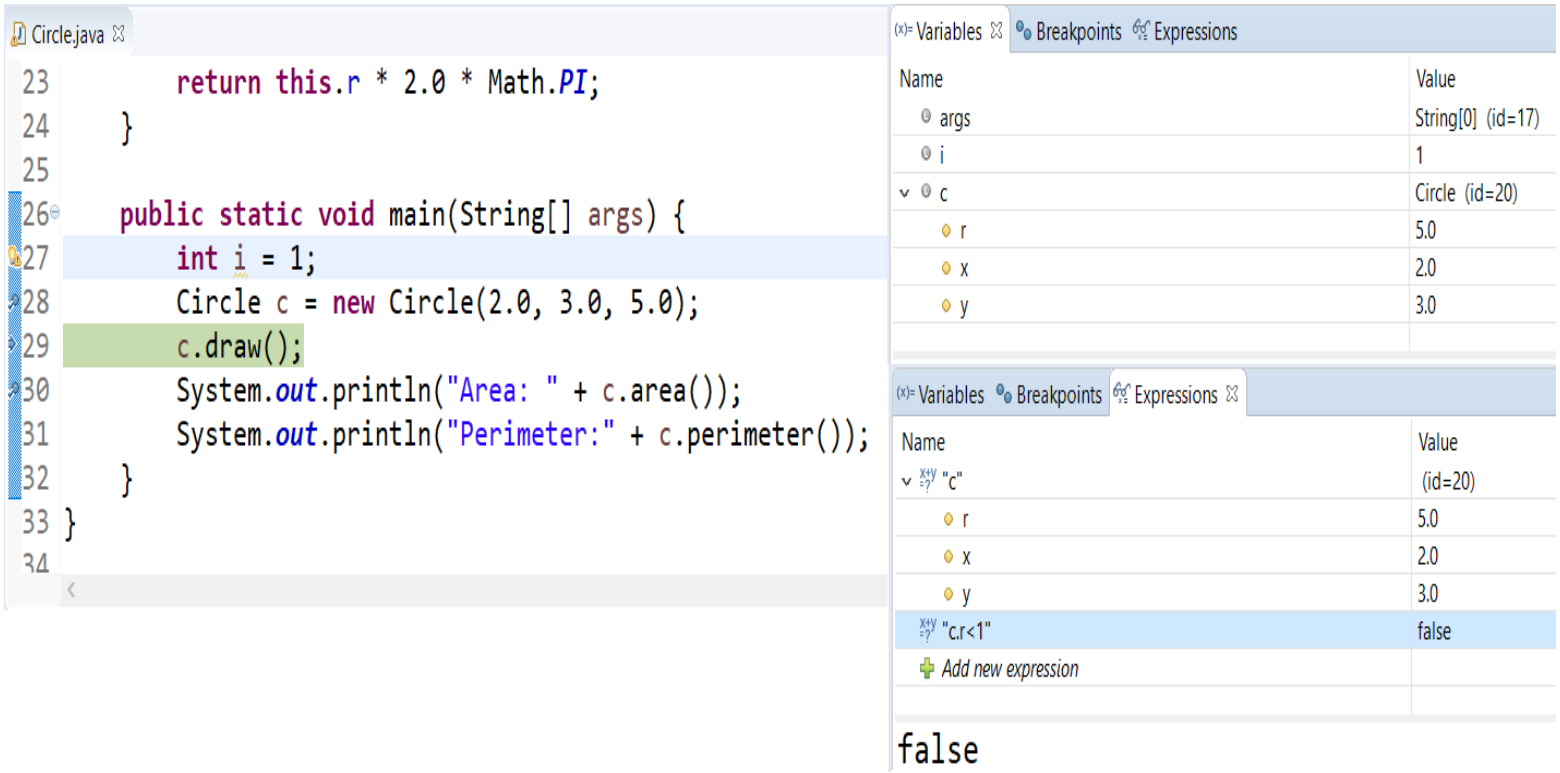
```
26      public static void main(String[] args) {
27          Circle c = new Circle(2.0, 3.0, 5.0);
28          c.draw();
29          System.out.println("Area: " + c.area());
30          System.out.println("Perimeter:" + c.perimeter());
31      }
32  }
33
```

2. Chạy chế độ debug (Alt+Shift+D,...)



Debug trong Java với Eclipse

3. Chạy từng bước để theo dõi
   - F5: chạy vào bên trong của phương thức.
   - F6: bỏ qua lời gọi phương thức.
   - F8: chạy đến điểm breakpoint tiếp theo.
4. Quan sát biến thông qua Variables/Expressions



Bài tập

- **Các bài tập với Sodoku**

1. Từ chương trình Sodoku đã được xây dựng, hãy thiết kế lại chương trình theo cách tiếp cận OOP.
   - Các đối tượng chính của bài toán bao gồm: Node, Game.
   - Node:
     - Thuộc tính: x, y, value
     - Phương thức: getters, setters, …
   - Game:
     - Thuộc tính: Node[][]
     - Phương thức: validate (row, column, zone), validate game.
   - Chương trình chính:
     - Khởi tạo game từ mảng tĩnh.

- **Chú ý**
  - Tách các lớp theo mô hình MVC: model, view, controller.
  - View: thực hiện nhập dữ liệu, hiển thị game.
  - Controller: thực hiện vòng lặp while điều khiển chương trình.

CÔNG NGHỆ JAVA