

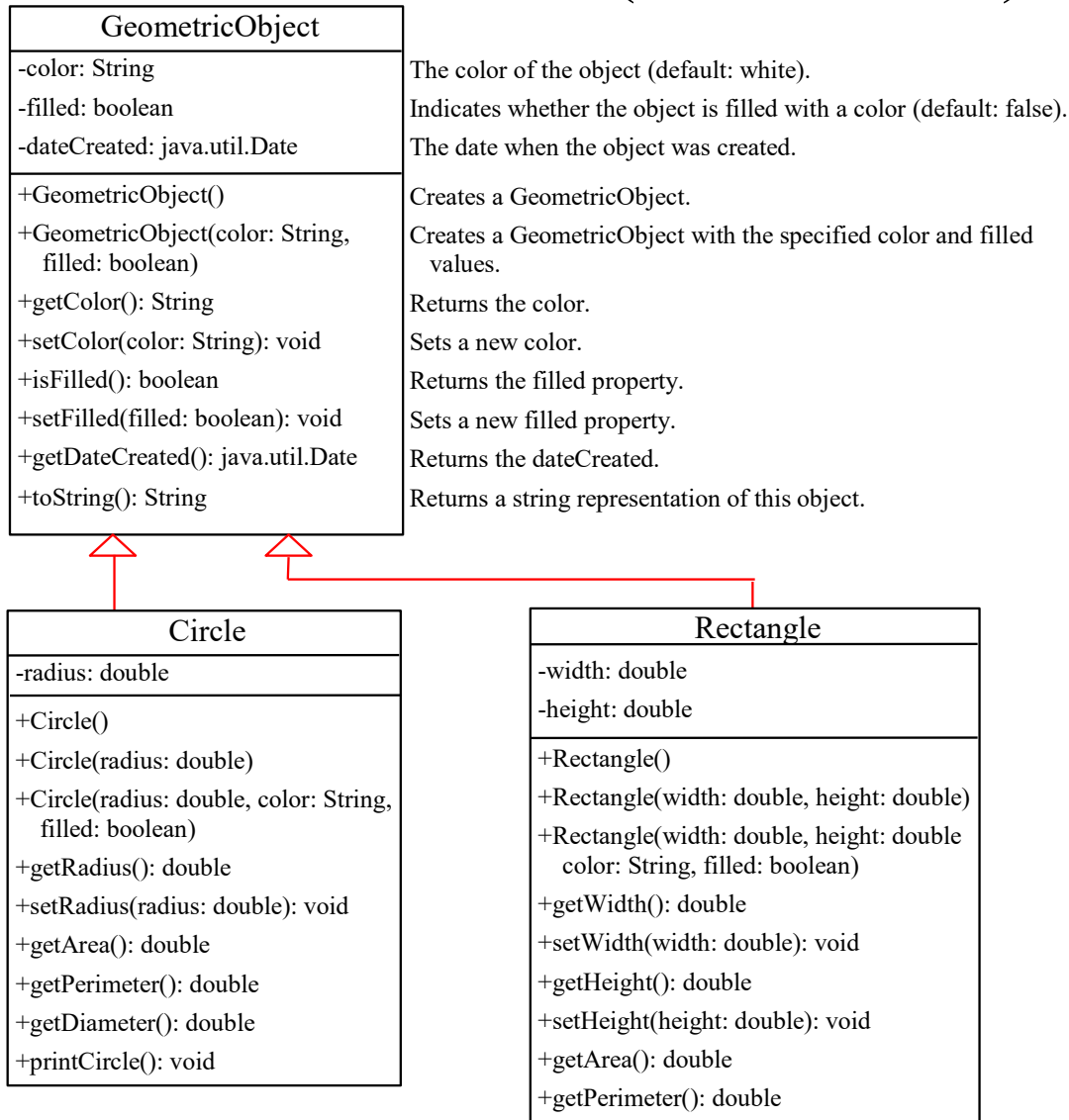
Chương 6

Hướng đối tượng

(Kế thừa – Đa hình)



Lớp cha (Superclasses) và Lớp con (Subclasses)



Phương thức khởi tạo của lớp cha có được kế thừa không?

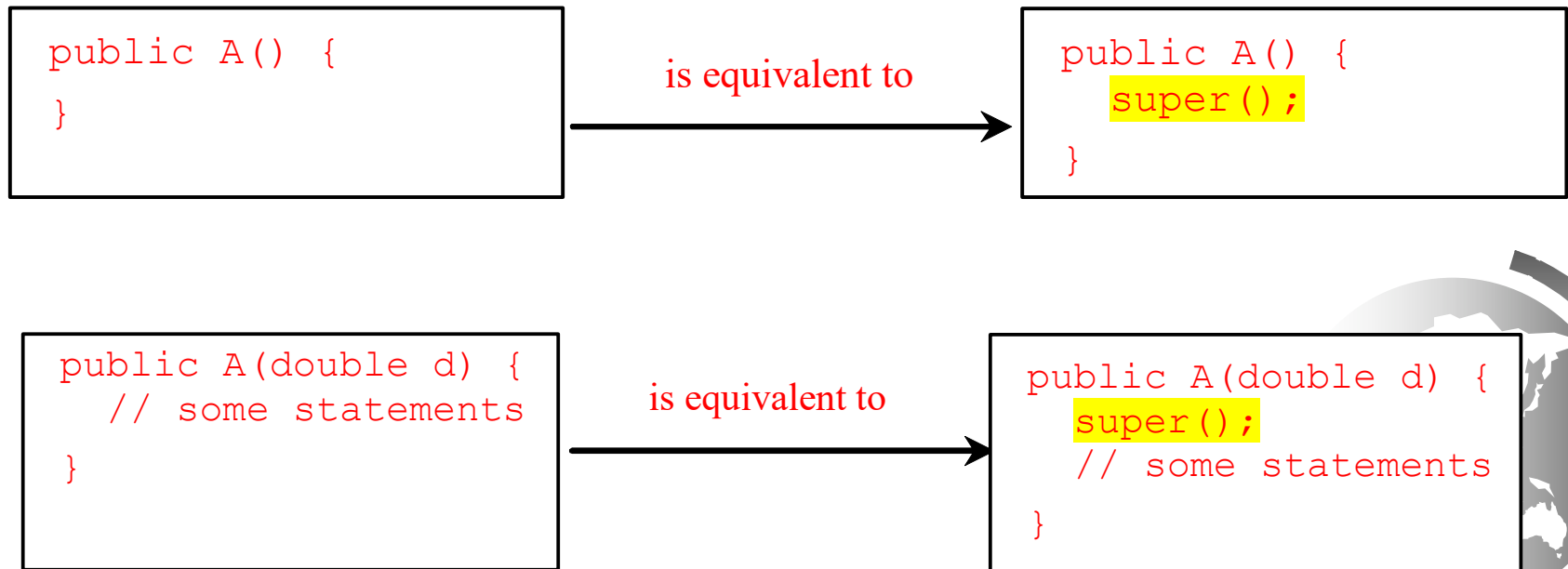
Không. Chúng không được kế thừa.

Chúng được gọi thực thi một cách tường minh (*sử dụng từ khóa **super***) hoặc ngầm định.

Một phương thức khởi tạo được sử dụng để tạo một thể hiện của một lớp. Không giống như các thuộc tính và phương thức, các phương thức khởi tạo của một lớp cha không được kế thừa trong lớp con. Chúng chỉ có thể được gọi thực thi từ các phương thức khởi tạo của lớp con bằng cách sử dụng từ khóa **super**. *Nếu từ khóa super không được sử dụng một cách tường minh thì phương thức khởi tạo không có tham số đầu vào của lớp cha sẽ được thực thi một cách tự động.*

Phương thức khởi tạo của lớp cha luôn luôn được thực thi

Một phương thức khởi tạo có thể gọi thực thi một nạp chồng phương thức khởi tạo hoặc phương thức khởi tạo của lớp cha của nó. Nếu không có phương thức khởi tạo nào được gọi thực thi một cách tường minh thì trình biên dịch sẽ thêm câu lệnh super() thành câu lệnh đầu tiên trong phương thức khởi tạo, ví dụ:



Sử dụng từ khóa `super`

Từ khóa `super` tham chiếu đến lớp cha. Có thể sử dụng từ khóa này theo 2 cách:

- ◆ Gọi một phương thức khởi tạo của lớp cha
- ◆ Gọi một phương thức của lớp cha



LƯU Ý

Cần phải sử dụng từ khóa **super** để gọi một phương thức khởi tạo của lớp cha. Việc gọi thực thi một phương thức khởi tạo bằng tên của lớp cha trong lớp con sẽ gây ra lỗi cú pháp. Trong Java, câu lệnh sử dụng từ khóa **super** là câu lệnh đầu tiên trong phương thức khởi tạo.



Chuỗi phương thức khởi tạo

Việc khởi tạo một thể hiện của một lớp gọi thực thi tất cả các phương thức khởi tạo của các lớp cha trong chuỗi kế thừa. Đó là *chuỗi phương thức khởi tạo*.

```
public class Faculty extends Employee {
    public static void main(String[] args) {
        new Faculty();
    }

    public Faculty() {
        System.out.println("(4) Faculty's no-arg constructor is invoked");
    }
}

class Employee extends Person {
    public Employee() {
        this("(2) Invoke Employee's overloaded constructor");
        System.out.println("(3) Employee's no-arg constructor is invoked");
    }

    public Employee(String s) {
        System.out.println(s);
    }
}

class Person {
    public Person() {
        System.out.println("(1) Person's no-arg constructor is invoked");
    }
}
```



Ví dụ minh họa

```
public class Faculty extends Employee {
    public static void main(String[] args) {
        new Faculty();
    }

    public Faculty() {
        System.out.println("(4) Faculty's no-arg constructor is invoked");
    }
}

class Employee extends Person {
    public Employee() {
        this("(2) Invoke Employee's overloaded constructor");
        System.out.println("(3) Employee's no-arg constructor is invoked");
    }

    public Employee(String s) {
        System.out.println(s);
    }
}

class Person {
    public Person() {
        System.out.println("(1) Person's no-arg constructor is invoked");
    }
}
```

1. Bắt đầu phương thức main



Ví dụ minh họa

```
public class Faculty extends Employee {  
    public static void main(String[] args) {  
        new Faculty();  
    }  
  
    public Faculty() {  
        System.out.println("(4) Faculty's no-arg constructor is invoked");  
    }  
}  
  
class Employee extends Person {  
    public Employee() {  
        this("(2) Invoke Employee's overloaded constructor");  
        System.out.println("(3) Employee's no-arg constructor is invoked");  
    }  
  
    public Employee(String s) {  
        System.out.println(s);  
    }  
}  
  
class Person {  
    public Person() {  
        System.out.println("(1) Person's no-arg constructor is invoked");  
    }  
}
```

2. Gọi phương thức khởi tạo của lớp Faculty



Ví dụ minh họa

```
public class Faculty extends Employee {
    public static void main(String[] args) {
        new Faculty();
    }

    public Faculty() {
        System.out.println("(4) Faculty's no-arg constructor is invoked");
    }
}

class Employee extends Person {
    public Employee() {
        this("(2) Invoke Employee's overloaded constructor");
        System.out.println("(3) Employee's no-arg constructor is invoked");
    }

    public Employee(String s) {
        System.out.println(s);
    }
}

class Person {
    public Person() {
        System.out.println("(1) Person's no-arg constructor is invoked");
    }
}
```

3. Gọi phương thức khởi tạo không tham số của lớp Employee



Ví dụ minh họa

```
public class Faculty extends Employee {  
    public static void main(String[] args) {  
        new Faculty();  
    }  
  
    public Faculty() {  
        System.out.println("(4) Faculty's no-arg constructor is invoked");  
    }  
}  
  
class Employee extends Person {  
    public Employee() {  
        this("(2) Invoke Employee's overloaded constructor");  
        System.out.println("(3) Employee's no-arg constructor is invoked");  
    }  
  
    public Employee(String s) {  
        System.out.println(s);  
    }  
}  
  
class Person {  
    public Person() {  
        System.out.println("(1) Person's no-arg constructor is invoked");  
    }  
}
```

4. Gọi phương thức khởi tạo
Employee(String)



Ví dụ minh họa

```
public class Faculty extends Employee {
    public static void main(String[] args) {
        new Faculty();
    }

    public Faculty() {
        System.out.println("(4) Faculty's no-arg constructor is invoked");
    }
}

class Employee extends Person {
    public Employee() {
        this("(2) Invoke Employee's overloaded constructor");
        System.out.println("(3) Employee's no-arg constructor is invoked");
    }

    public Employee(String s) {
        System.out.println(s);
    }
}

class Person {
    public Person() {
        System.out.println("(1) Person's no-arg constructor is invoked");
    }
}
```

5. Gọi phương thức khởi tạo
Person()

Ví dụ minh họa

```
public class Faculty extends Employee {  
    public static void main(String[] args) {  
        new Faculty();  
    }  
  
    public Faculty() {  
        System.out.println("(4) Faculty's no-arg constructor is invoked");  
    }  
}  
  
class Employee extends Person {  
    public Employee() {  
        this("(2) Invoke Employee's overloaded constructor");  
        System.out.println("(3) Employee's no-arg constructor is invoked");  
    }  
  
    public Employee(String s) {  
        System.out.println(s);  
    }  
}  
  
class Person {  
    public Person() {  
        System.out.println("(1) Person's no-arg constructor is invoked");  
    }  
}
```

6. In ra màn hình

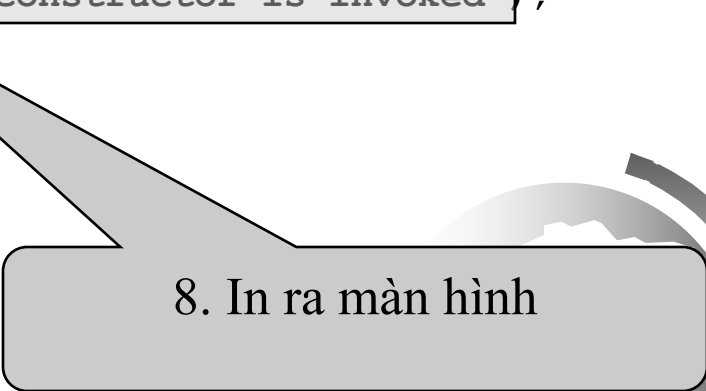
Ví dụ minh họa

```
public class Faculty extends Employee {  
    public static void main(String[] args) {  
        new Faculty();  
    }  
  
    public Faculty() {  
        System.out.println("(4) Faculty's no-arg constructor is invoked");  
    }  
}  
  
class Employee extends Person {  
    public Employee() {  
        this("(2) Invoke Employee's overloaded constructor");  
        System.out.println("(3) Employee's no-arg constructor is invoked");  
    }  
  
    public Employee(String s) {  
        System.out.println(s);  
    }  
}  
  
class Person {  
    public Person() {  
        System.out.println("(1) Person's no-arg constructor is invoked");  
    }  
}
```

7. In ra màn hình

Ví dụ minh họa

```
public class Faculty extends Employee {  
    public static void main(String[] args) {  
        new Faculty();  
    }  
  
    public Faculty() {  
        System.out.println("(4) Faculty's no-arg constructor is invoked");  
    }  
}  
  
class Employee extends Person {  
    public Employee() {  
        this("(2) Invoke Employee's overloaded constructor");  
        System.out.println("(3) Employee's no-arg constructor is invoked");  
    }  
  
    public Employee(String s) {  
        System.out.println(s);  
    }  
}  
  
class Person {  
    public Person() {  
        System.out.println("(1) Person's no-arg constructor is invoked");  
    }  
}
```



8. In ra màn hình

Ví dụ minh họa

```
public class Faculty extends Employee {  
    public static void main(String[] args) {  
        new Faculty();  
    }  
  
    public Faculty() {  
        System.out.println("(4) Faculty's no-arg constructor is invoked");  
    }  
}  
  
class Employee extends Person {  
    public Employee() {  
        this("(2) Invoke Employee's overloaded constructor");  
        System.out.println("(3) Employee's no-arg constructor is invoked");  
    }  
  
    public Employee(String s) {  
        System.out.println(s);  
    }  
}  
  
class Person {  
    public Person() {  
        System.out.println("(1) Person's no-arg constructor is invoked");  
    }  
}
```

9. In ra màn hình



Ví dụ

Tìm lỗi trong đoạn chương trình sau:

```
public class Apple extends Fruit {  
}  
  
class Fruit {  
    public Fruit(String name) {  
        System.out.println("Fruit's constructor is invoked");  
    }  
}
```



Định nghĩa lớp con

Một lớp con kế thừa từ một lớp cha. Chúng ta có thể:

- ◆ Thêm các thuộc tính mới
- ◆ Thêm các phương thức mới
- ◆ Ghi đè (Override) các phương thức của lớp cha



Gọi các phương thức của lớp cha

Chúng ta có thể viết lại phương thức printCircle() trong lớp Circle như sau:

```
public void printCircle() {  
    System.out.println("The circle is created " +  
        super.getDateCreated() + " and the radius is " + radius);  
}
```



Ghi đè phương thức

Một lớp con kế thừa các phương thức từ một lớp cha. Đôi khi lớp con cần cài đặt lại một phương thức đã được định nghĩa trong lớp cha. Đó được gọi là ghi đè phương thức (*method overriding*).

```
public class Circle extends GeometricObject {  
    // Other methods are omitted  
  
    /** Override the toString method defined in GeometricObject */  
    public String toString() {  
        return super.toString() + "\nradius is " + radius;  
    }  
}
```



LƯU Ý

Một phương thức private thì không thể bị ghi đè.

Nếu một phương thức được định nghĩa trong lớp con và lớp cha có cùng tên nhưng trong lớp cha là private thì 2 phương thức này *không liên quan* với nhau.



LƯU Ý

Một phương thức tĩnh cũng có thể được kế thừa. Tuy nhiên một phương thức tĩnh không thể bị ghi đè.

Nếu một phương thức tĩnh được định nghĩa trong lớp cha và được định nghĩa lại trong một lớp con thì phương thức trong lớp cha sẽ bị *ấn*.



Ghi đè (Overriding) vs. Nạp chồng (Overloading)

```
public class Test {  
    public static void main(String[] args) {  
        A a = new A();  
        a.p(10);  
        a.p(10.0);  
    }  
}  
  
class B {  
    public void p(double i) {  
        System.out.println(i * 2);  
    }  
}  
  
class A extends B {  
    // This method overrides the method in B  
    public void p(double i) {  
        System.out.println(i);  
    }  
}
```

```
public class Test {  
    public static void main(String[] args) {  
        A a = new A();  
        a.p(10);  
        a.p(10.0);  
    }  
}  
  
class B {  
    public void p(double i) {  
        System.out.println(i * 2);  
    }  
}  
  
class A extends B {  
    // This method overloads the method in B  
    public void p(int i) {  
        System.out.println(i);  
    }  
}
```

Lớp Object

Mọi lớp trong Java đều được dẫn xuất từ lớp `java.lang.Object`. Nếu không có kế thừa được chỉ định khi một lớp được định nghĩa thì lớp cha của lớp đó sẽ là lớp `Object`.

```
public class Circle {  
    ...  
}
```

Equivalent

```
public class Circle extends Object {  
    ...  
}
```


Phương thức toString() trong lớp Object

Phương thức toString() trả về một chuỗi biểu diễn cho một đối tượng. Cài đặt mặc định trả về một chuỗi chứa tên của lớp của đối tượng, dấu @, và một con số biểu diễn cho đối tượng.

```
Loan loan = new Loan();  
System.out.println(loan.toString());
```

Kết quả: Loan@15037e5 .

Thông điệp này không hữu ích và mang nhiều thông tin. Chúng ta thường ghi đè phương thức toString để trả về một chuỗi digestible biểu diễn cho đối tượng.



Đa hình

Đa hình nghĩa là một biến của một supertype có thể tham chiếu đến một đối tượng subtype.

Một lớp định nghĩa một kiểu dữ liệu. Một kiểu dữ liệu được định nghĩa bởi một lớp con gọi là *subtype*, và một kiểu dữ liệu được định nghĩa bởi lớp cha gọi là *supertype*. Do đó, có thể nói rằng **Circle** là một subtype của **GeometricObject** và **GeometricObject** là một supertype cho **Circle**.

Đa hình, Gắn kết động và Lập trình tổng quát

```
public class PolymorphismDemo {
    public static void main(String[] args) {
        m(new GraduateStudent());
        m(new Student());
        m(new Person());
        m(new Object());
    }

    public static void m(Object x) {
        System.out.println(x.toString());
    }
}

class GraduateStudent extends Student {
}

class Student extends Person {
    public String toString() {
        return "Student";
    }
}

class Person extends Object {
    public String toString() {
        return "Person";
    }
}
```

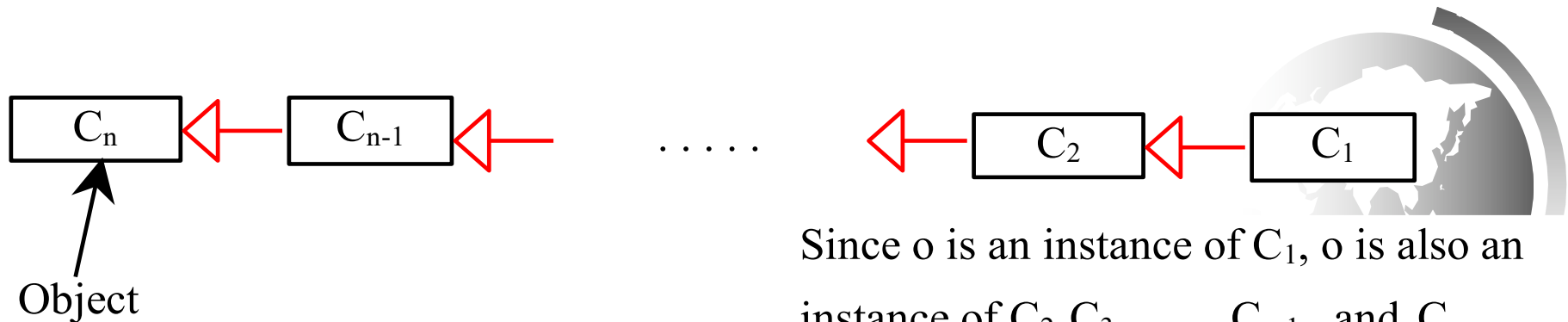
Phương thức m có 1 tham số có kiểu Object. Chúng ta có thể gọi nó với bất kỳ đối tượng nào.

Một đối tượng của một subtype có thể được sử dụng ở bất kỳ chỗ nào yêu cầu giá trị supertype của nó. Đó gọi là Đa hình (*polymorphism*).

Khi phương thức m(Object x) được thực thi, phương thức toString của tham số x được gọi. x có thể là một thể hiện của lớp GraduateStudent, Student, Person, hoặc Object. Các lớp GraduateStudent, Student, Person, và Object có cài đặt riêng phương thức toString. Cài đặt này thường được xác định một cách tự động bởi máy ảo Java lúc thực thi. Khả năng đó gọi là gắn kết động (*dynamic binding*).

Gắn kết động

Gắn kết động làm việc như sau: Giả sử đối tượng o là một thể hiện của các lớp C_1, C_2, \dots, C_{n-1} , và C_n , trong đó C_1 là một lớp con của C_2 , C_2 là một lớp con của C_3 , ..., và C_{n-1} là một lớp con của C_n . Do đó, C_n là lớp tổng quát nhất, và C_1 là lớp cụ thể nhất. Trong Java, C_n là lớp *Object*. Nếu o gọi một phương thức p , máy ảo JVM tìm kiếm cài đặt của phương thức p trong các lớp C_1, C_2, \dots, C_{n-1} và C_n , một cách tuần tự, cho đến khi tìm thấy. Khi một cài đặt được tìm thấy, quá trình tìm kiếm sẽ dừng lại và cài đặt được tìm thấy đầu tiên sẽ được gọi thực thi.



Method Matching vs. Binding

Việc khớp 1 chữ ký phương thức và việc gắn kết 1 cài đặt phương thức là 2 vấn đề. Trình biên dịch tìm 1 phương thức trùng khớp thông qua kiểu tham số, số lượng tham số và thứ tự của các tham số trong thời gian biên dịch. Một phương thức có thể được cài đặt trong nhiều lớp con. Máy ảo Java gắn kết động cài đặt của phương thức lúc thực thi.



Lập trình tổng quát

```
public class PolymorphismDemo {
    public static void main(String[] args) {
        m(new GraduateStudent());
        m(new Student());
        m(new Person());
        m(new Object());
    }

    public static void m(Object x) {
        System.out.println(x.toString());
    }
}

class GraduateStudent extends Student {
}

class Student extends Person {
    public String toString() {
        return "Student";
    }
}

class Person extends Object {
    public String toString() {
        return "Person";
    }
}
```

Đa hình cho phép các phương thức được sử dụng một cách tổng quát cho một phạm vi rộng của các đối số. Đó là lập trình tổng quát (generic programming). Nếu một kiểu tham số của phương thức là một lớp cha (v.d., Object), chúng ta có thể truyền một đối tượng của bất kỳ lớp con vào phương thức này (v.d., Student hoặc String). Khi một đối tượng (v.d., một đối tượng Student hoặc String) được sử dụng trong phương thức, một cài đặt đặc biệt của phương thức của đối tượng đó được gọi (v.d., toString) được determined một cách tự động.



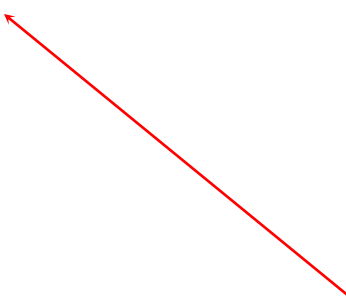
Ép kiểu Objects

Ép kiểu được sử dụng để chuyển một đối tượng của một lớp này sang một lớp khác theo phân cấp kế thừa. Câu lệnh trong ví dụ trước:

```
m(new Student());
```

Gán đối tượng `new Student()` cho một tham số kiểu `Object`. Tương đương với câu lệnh:

```
Object o = new Student(); // Ép kiểu ngầm định  
m(o);
```



Câu lệnh `Object o = new Student()`, là ép kiểu ngầm định, là câu lệnh đúng bởi vì 1 thể hiện của lớp `Student` tự động là 1 thể hiện của lớp `Object`.

Tại sao ép kiểu là cần thiết

Giả sử cần gán một đối tượng tham chiếu đến một biến của kiểu `Student` bằng câu lệnh sau:

```
Student b = o;
```

Một lỗi biên dịch xuất hiện. Tại sao câu lệnh **`Object o = new Student()`** đúng và câu lệnh **`Student b = o`** thì không đúng?

Câu lệnh đúng là:

```
Student b = (Student)o; // Ép kiểu tường minh
```



Ép kiểu từ Lớp cha sang Lớp con

Ép kiểu tường minh cần được sử dụng khi ép một đối tượng của lớp cha sang lớp con. Loại ép kiểu này có thể không thành công.

```
Apple x = (Apple) fruit;
```

```
Orange x = (Orange) fruit;
```



Toán tử instanceof

Sử dụng toán tử `instanceof` để kiểm tra đối tượng đó là thể hiện của lớp nào:

```
Object myObject = new Circle();  
... // Some lines of code  
/** Perform casting if myObject is an instance of  
    Circle */  
if (myObject instanceof Circle) {  
    System.out.println("The circle diameter is " +  
        ((Circle)myObject).getDiameter());  
    ...  
}
```



Phương thức equals

Phương thức equals () so sánh nội dung của 2 đối tượng. Mặc định, phương thức equals trong lớp Object được cài đặt như sau:

```
public boolean equals(Object obj) {  
    return this == obj;  
}
```

VD: phương thức equals được ghi đè trong lớp Circle.

```
public boolean equals(Object o) {  
    if (o instanceof Circle) {  
        return radius == ((Circle)o).radius;  
    }  
    else  
        return false;  
}
```



Lớp ArrayList

Chúng ta có thể tạo một mảng lưu trữ các đối tượng. Nhưng kích thước của mảng là cố định khi tạo mảng. Java cung cấp lớp ArrayList có thể được sử dụng để lưu trữ số lượng không giới hạn các đối tượng.

java.util.ArrayList<E>

```
+ArrayList()  
+add(o: E) : void  
+add(index: int, o: E) : void  
+clear(): void  
+contains(o: Object): boolean  
+get(index: int) : E  
+indexOf(o: Object) : int  
+isEmpty(): boolean  
+lastIndexOf(o: Object) : int  
+remove(o: Object): boolean  
+size(): int  
+remove(index: int) : boolean  
+set(index: int, o: E) : E
```

Creates an empty list

Appends a new element *o* at the end of this list.

Adds a new element *o* at the specified index in this list.

Removes all the elements from this list.

Returns true if this list contains the element *o*.

Returns the element from this list at the specified index.

Returns the index of the first matching element in this list.

Returns true if this list contains no elements.

Returns the index of the last matching element in this list.

Removes the element *o* from this list.

Returns the number of elements in this list.

Removes the element at the specified index.

Sets the element at the specified index.

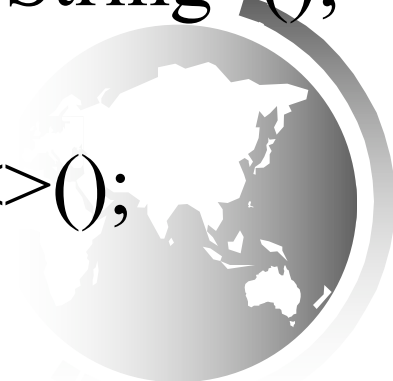
Kiểu tổng quát (Generic Type)

ArrayList là một lớp tổng quát với kiểu tổng quát E. Chúng ta có thể chỉ định một kiểu rời rạc (concrete type) để thay thế cho E khi tạo một ArrayList.

Ví dụ:

```
ArrayList<String> cities = new ArrayList<String>();
```

```
ArrayList<String> cities = new ArrayList<>();
```



Arrays vs ArrayList

<i>Operation</i>	<i>Array</i>	<i>ArrayList</i>
Creating an array/ArrayList	<code>String[] a = new String[10]</code>	<code>ArrayList<String> list = new ArrayList<>();</code>
Accessing an element	<code>a[index]</code>	<code>list.get(index);</code>
Updating an element	<code>a[index] = "London";</code>	<code>list.set(index, "London");</code>
Returning size	<code>a.length</code>	<code>list.size();</code>
Adding a new element		<code>list.add("London");</code>
Inserting a new element		<code>list.add(index, "London");</code>
Removing an element		<code>list.remove(index);</code>
Removing an element		<code>list.remove(Object);</code>
Removing all elements		<code>list.clear();</code>



Array Lists from/to Arrays

Tạo một ArrayList từ một mảng:

```
String[] array = {"red", "green", "blue"};
```

```
    ArrayList<String> list = new  
ArrayList<>(Arrays.asList(array));
```

Tạo một mảng từ một ArrayList:

```
String[] array1 = new String[list.size()];  
list.toArray(array1);
```



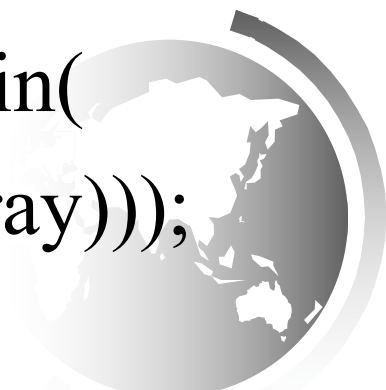
max và min trong Array List

```
String[] array = {"red", "green", "blue"};
```

```
System.out.println(java.util.Collections.max(  
    new ArrayList<String>(Arrays.asList(array))));
```

```
String[] array = {"red", "green", "blue"};
```

```
System.out.println(java.util.Collections.min(  
    new ArrayList<String>(Arrays.asList(array))));
```



Trộn ngẫu nhiên một Array List

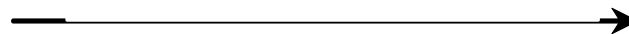
```
Integer[] array = {3, 5, 95, 4, 15, 34, 3, 6, 5};  
ArrayList<Integer> list = new  
    ArrayList<>(Arrays.asList(array));  
java.util.Collections.shuffle(list);  
System.out.println(list);
```



Chỉ định `protected`

- ♦ Chỉ định `protected` có thể được áp dụng trên dữ liệu và các phương thức trong một lớp. Một trường dữ liệu hay một phương thức `protected` trong một lớp `public` có thể được truy cập bởi bất kỳ lớp nào trong cùng `package` hoặc các lớp con của nó, hay thậm chí các lớp con trong một `package` khác.
- ♦ `private`, `default`, `protected`, `public`

Visibility increases



`private`, `none` (if no modifier is used), `protected`, `public`

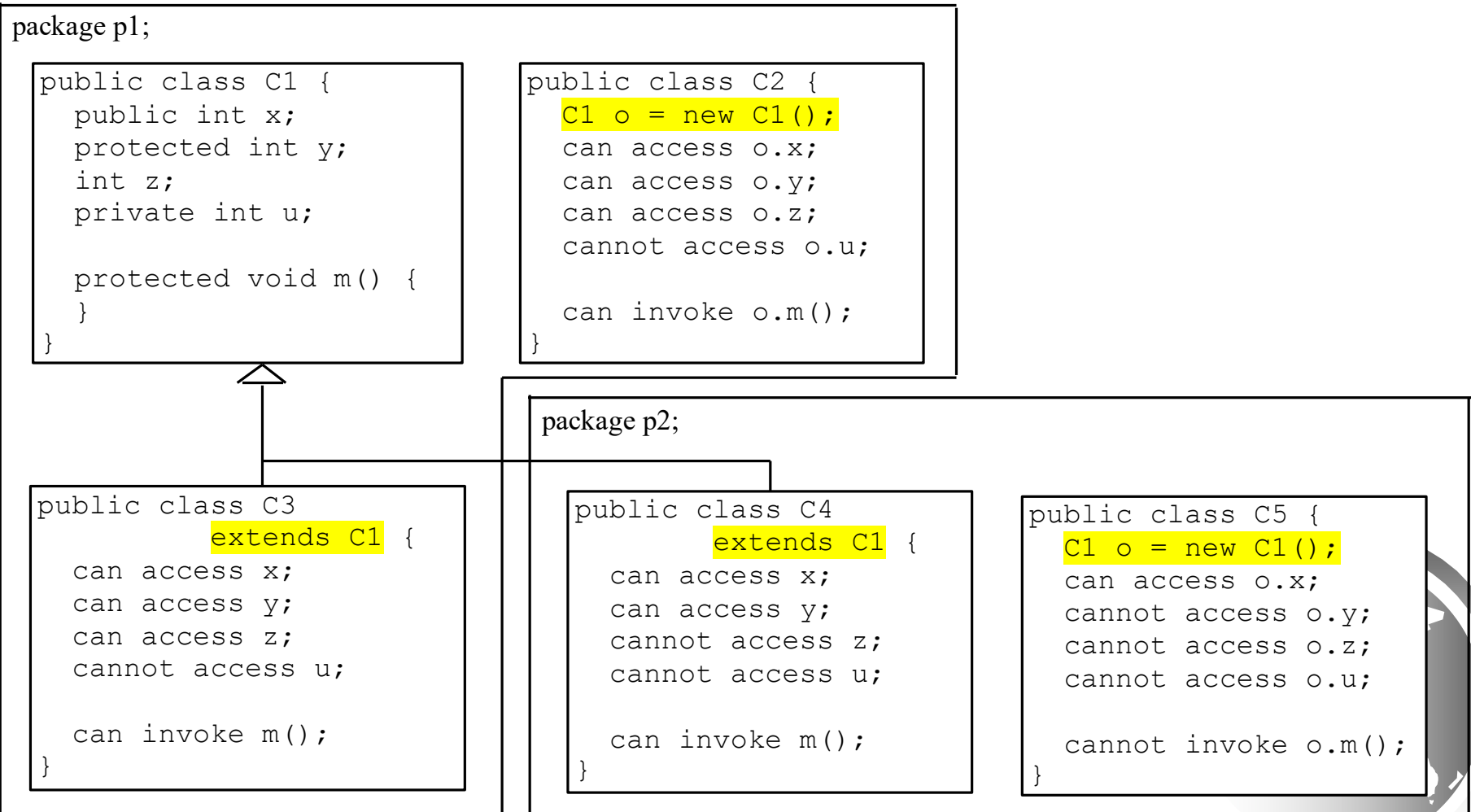


Chỉ định truy cập

Modifier on members in a class	Accessed from the same class	Accessed from the same package	Accessed from a subclass	Accessed from a different package
public	✓	✓	✓	✓
protected	✓	✓	✓	—
default	✓	✓	—	—
private	✓	—	—	—



Chỉ định truy cập



Một lớp con không thể làm suy yếu khả năng truy cập

Một lớp con có thể ghi đè một phương thức protected trong lớp cha của nó và có thể đổi chỉ định truy cập là public.

Tuy nhiên, một lớp con không thể làm suy yếu khả năng truy cập của một phương thức trong lớp cha.

Ví dụ, nếu 1 phương thức được định nghĩa với chỉ định public trong lớp cha thì nó cũng cần được định nghĩa public trong lớp con.

LƯU Ý

Các chỉ định (*modifiers*) thường được sử dụng trên các lớp và các thành viên của lớp (*dữ liệu và phương thức*), ngoại trừ chỉ định final có thể được sử dụng trên các biến cục bộ trong một phương thức.

Một biến cục bộ final là một hằng số bên trong một phương thức.



Chỉ định `final`

- ♦ Lớp `final` không thể được kế thừa:

```
final class Math {  
    ...  
}
```

- ♦ Biến `final` là một hằng số:

```
final static double PI = 3.14159;
```

- ♦ Phương thức `final` không thể bị ghi đè.

