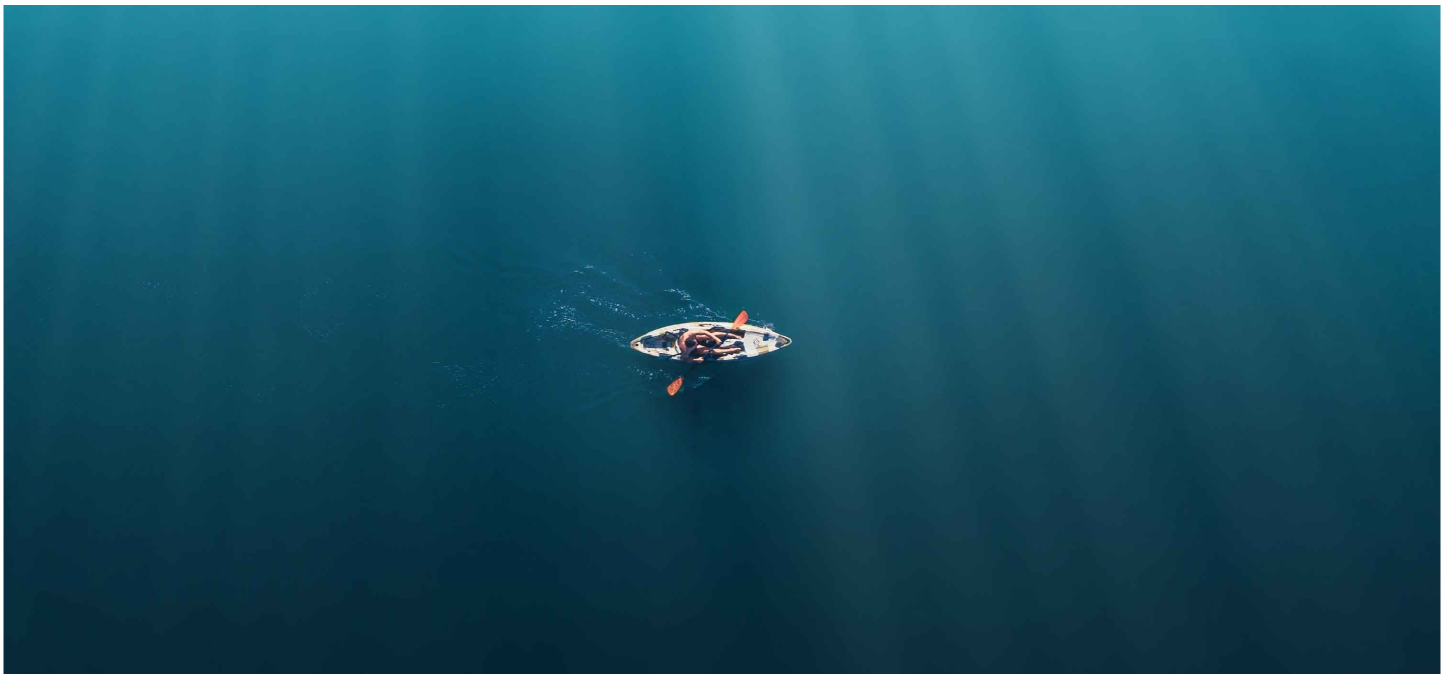


17-Polymorphism

Author: Vincent Lau

Learning Objectives

> Understand the concept of Polymorphism

> Describe the two types of polymorphism

> Chain overloaded constructors

> Describe the things to watch out for when casting primitives and objects

> Understand the difference between upcasting and downcasting

> Understand the risk of downcasting and use *instanceof* keyword to check type

Overview

- The dictionary definition of *polymorphism* refers to a principle in biology in which an organism or species can have many different forms.

- Polymorphism is one of the characteristics of Object-Oriented Programming (OOP).
- Subclasses can define their own unique behaviors while sharing some of the same functionality of the parent class.
- In Java, there are two core types of polymophism - **static/ compile-time polymorphism** AND **dynamic/ runtime polymorphism**.

Compile-time/ Static Polymorphism (Overloading)

- Static polymorphism is a type of polymorphism which is resolved at compile time. In Java, this is displayed through the use of `overloading`.
- *Overloading* is implemented by **having multiple methods with the same name but different method signatures (return type does not count)**. That means the compiler knows at compile time which method is going to be invoked automatically, hence the name **Static Polymorphism**.
- The benefit of *overloading* is that we can define a group of closely related methods with the same name, but different behaviors.

```java
public class OverloadingClass {
    public void doSomething() {
        System.out.println("do something");
    }

//    // Having a different return type is NOT overloading
//    public String doSomething() {
//        return "do something";
//    }

    public void doSomething(String thing) {
        System.out.println("do " + thing);
    }

    public void doSomething(int n) {
        for (int i = 1; i <= n; i++) {
            System.out.println("do something " + i + " times");
        }
    }

    public void doSomething(String thing, int n) {
        for (int i = 1; i <= n; i++) {
            System.out.println("do " + thing + " " + i + " times");
        }
    }

    public static void main(String[] args) {
        OverloadingClass c = new OverloadingClass();
        c.doSomething();
        c.doSomething("something else");
        c.doSomething(3);
        c.doSomething("something else", 3);
    }
}

/*
Output:
do something
do something else
do something 1 times
do something 2 times
do something 3 times
do something else 1 times
do something else 2 times
do something else 3 times
*/
```

- We can also **chain overloaded constructors**, also known as **horizontal constructor chaining**.
- Constructors with different parameters will call each other for constructing instance.

```java
public class ClassWithConstructors {
    private int x;
    private int y;

    public ClassWithConstructors() {
        this(100, 100); // call another constructor
    }

    public ClassWithConstructors(int x) {
        this(x, 100);
    }

    public ClassWithConstructors(int x, int y) {
        this.x = x;
        this.y = y;
    }
}
```

Run-time/ Dynamic Polymorphism 動態多型 (Overriding)

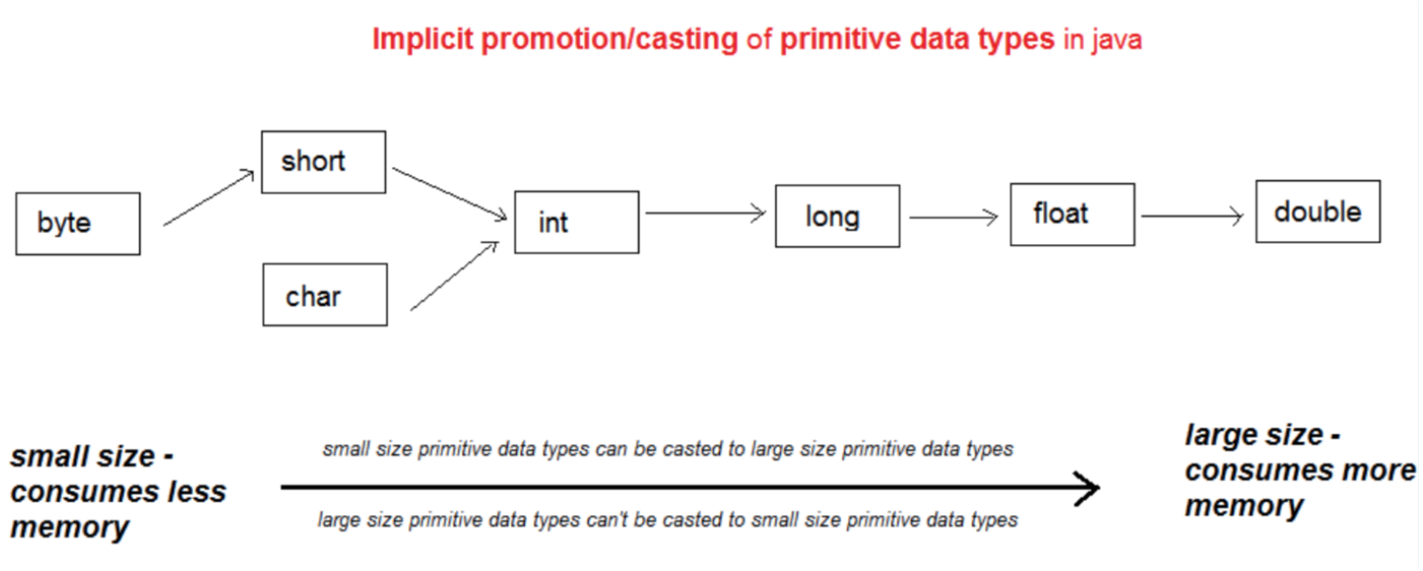
- **一個訊息 (message or event or stimulus) 的意義是由接收者 (接收到這個訊息的物件) 來解釋，而不是由訊息發出者 (sender) 來解釋。**所以，在runtime時只要接受者換成不同的物件或是instance，系統的行為就會改變。具有這樣的特性就稱之為Dynamic polymorphism
- With dynamic polymorphism, the **JVM** handles the detection of the appropriate method to invoke when a subclass is assigned to its parent form. This is necessary because the subclass may override some or all of the methods defined in the parent class. This is also known as `overriding`.
- The JVM calls the appropriate method for the object that is referred to in each variable (e.g. SubclassA). It does not call the method that is defined by the variable's type (e.g. ParentClass). This behavior is referred to as `virtual method invocation` and demonstrates an aspect of the important **polymorphism** features in the Java language.

```java
public class ParentClass {
    public void doSomething() {
        System.out.println("do something from ParentClass");
    }
}

public class SubclassA extends ParentClass {

    @Override
    public void doSomething() {
        System.out.println("do something from SubClassA");
    }

    public String getString() {
        return "abc";
    }
}

public class SubclassB extends ParentClass {
    @Override
    public void doSomething() {
        System.out.println("do something from SubClassB");
    }
}

public class OverridingDemo {
    public static void main(String[] args) {
        ParentClass o1 = new ParentClass();
        ParentClass o2 = new SubclassA();
        ParentClass o3 = new SubclassB();

        o1.doSomething();
        o2.doSomething();
        o3.doSomething();
        // o2.getString(); // Compile Error, ParentClass has NO getString
method

        SubclassA s1 = (SubclassA) o1; // downcast, risky
    }
}

/*
Output:
do something from ParentClass
do something from SubClassA
do something from SubClassB
*/
```

Casting

- Casting is a way of explicitly informating the compiler that you intend to **convert a primitive or object into a specific type**, with the **risk of failing at runtime**.
- To perform a cast, we **specify the target type in parentheses** in front of the variable to be converted.

Casting Primitives


Implicit promotion/casting of primitive data types in java
byte
short
char
int
long
float
double
small size - consumes less memory
small size primitive data types can be casted to large size primitive data types
large size primitive data types can't be casted to small size primitive data types
large size - consumes more memory

- ***Type promotion (implicit conversion)***, also known as *widening conversion,* means casting a primitive variable to another primitive type that has a larger capacity, e.g. from an int to a long. In this case, we **do not need to specify the target type** in parentheses.
- **Type Casting *(explicit conversion)***, also known as *narrowing conversion, b*e careful when casting a primitive variable (e.g. *long*) to a type with smaller capacity (e.g. *int*), **data loss** will occur.

```java
int i1 = 1;
long i2 = i1; // type promotion, no need to specify the target type in parentheses
int i3 = (int) i2; // type casting, narrowing conversion
System.out.println("i1: " + i1);
System.out.println("i2: " + i2);
System.out.println("i3: " + i3);

int i4 = Integer.MAX_VALUE;
long i5 = i4 + 1; // i) add two integers -> overflow of integer, ii) store the overflow value to long.
long i6 = ((long) i4) + 1;  // i) cast an int to a long, ii) add 1 and store in long
int i7 = (int) i6; // cast a bigger type to a smaller type
System.out.println("i4: " + i4);
System.out.println("i5: " + i5);
System.out.println("i6: " + i6);
System.out.println("i7: " + i7);

/*
Output:
i1: 1
i2: 1
i3: 1
i4: 2147483647
i5: -2147483648
i6: 2147483648
i7: -2147483648
*/
```

Casting Objects

- In Java, we can use `instanceof` to check the type of a particular object before `explicit casting` it to a specific type.

```java
// This equals() method is in Class Cat
```

```java
 2 // and let's assume Class cat extends Animal Class
 3 // Overriding the default *equals* method in the *Object* class
 4 // This e
 5 @Override
 6 public boolean equals(Object o) {
 7     if (this == o) {
 8         return true;
 9     }
10     // if o is created from Animal Class or Cat class, we let it to downcast
11     // if o is from Dog Class, return false in line 13 directly
12     if (!(o instanceof Cat)) {  // o is an object, where Cat is a Class
13         return false;
14     }
15     Cat cat = (Cat) o; // downcasting
16     return this.name.equals(cat.getName()) // string
17             && this.weight == cat.getWeight() // int
18             && this.height == cat.getHeight() // int
19             && this.hkid == cat.getHkid();
20 }
```

Upcasting

- Upcasting means **casting a child class object to a parent class type**.
- Upcasting can be done **implicitly**.
- Upcasting gives us the flexibility to access the parent class members, but we **lose access to some subtype-specific methods (and fields)** after performing an upcast.
- With upcasting, we can still access invoke the child class's overridden methods at runtime.

Downcasting

- Similarly, downcasting means **casting a parent class object to a child class type**.
- Downcasting cannot be implicit. We must **specify the target type with parentheses**.
- If an object being referred to by the parent class type variable is **not the child class type to begin with**, downcasting it from the parent type to the child type can lead to **runtime errors**. Therefore, it is important to perform type checks, such as ***instanceof***, before downcasting.

```java
public class ParentClass {
    public void doSomething() {
        System.out.println("do something from ParentClass");
    }
}
```

```java

public class SubclassC extends ParentClass {
    @Override
    public void doSomething() {
        System.out.println("do something from SubclassC");
    }

    public void doSomethingElse() {
        System.out.println("do something else from SubclassC");
    }
}

public class SomeOtherClass extends ParentClass {

    public void otherMethod() {
        System.out.println("print something");
    }
}

// Factory Pattern
public static ParentClass produceChild(int count) {
    switch (count) {
        case 1:
            return new SubclassC();
        case 2:
            return new SomeOtherClass();
        default:
            return null;
    }
}

public class CastingDemo {
    public static void main(String[] args) {

        // Scenario 1
        ParentClass parentObject = new SubclassC(); // upcasting
        parentObject.doSomething(); // can still access child's overridden method
        // parentObject.doSomethingElse();  // Compile error, can no longer access child's method

        SubclassC childObject2 = (SubclassC) parentObject; // downcasting
        childObject2.doSomethingElse(); // can access child's method again

        // Scenario 2
        // Cause Runtime error - class SubclassC cannot be cast to class SomeOtherClass
        // To solve, add if(parentObject instanceOf SomeOtherClass)
        // SomeOtherClass childObject3 = (SomeOtherClass) parentObject;

        // if we got Factory Pattern (method produceChild())
        ParentClass child = produceChild(2);
    }
}
```

- For *method(2)* in the main, compiler will **upcast 2 to float** and call method with float as argument.

```
public class MyClass {

    static void method(float x) {
            System.out.println("hello");
    }

    //static void method(int x) {
    //        System.out.println("hello");
    //}

    public static void main(String args[]) {
            method(2); // cast 2 to float (2 is int)
    }
}
/*
Output:
float
*/
```

Questions

- What is Polymorphism?
- What are the two types of polymorphism?
- How to chain overloaded constructors?
- What are the things to watch out for when casting primitives and objects?
- What is the difference between upcasting and downcasting?
- What is the risk of downcasting?
- What is *instanceOf* used for ?

[illegible]