沒有泛型以前

1. 類別定義時邏輯完全一樣，只是當中宣告的成員型態不同
2. 有點小聰明的程式設計人員會將第一個類別內容複製至 另一個 檔案中，然後用編輯器「取代 」功能一次取代所有的型態名稱

```java
class Foo1 {

    private int foo;

    public void setFoo(int foo) {
        this.foo = foo;
    }

    public int getFoo() {
        return foo;
    }
}
```

```java
class Foo2 {

    private double foo;

    public void setFoo(double foo) {
        this.foo = foo;
    }

    public double getFoo() {
        return foo;
    }
}
```

3. 如果類別中的邏輯要修改，您要修改兩個檔 案，泛型 (Generics) 的需求就 在此 產生，當 您定義類別時，發現到好幾個類別的邏輯其實都相同，就只是當中所 涉及的型態不一樣時 ，使用複製、貼上 、取代的功能來撰寫程式只是讓 您增加不必要的 檔案管理困擾，有沒有辦 法只寫一個檔案就好，畢竟 它們 的邏輯是相同的
4. 泛型的第一步➔把基本資料型態改為 包裝型別➔泛型只能使用在參考資料型別上

```java
class Foo3 {

    private Integer foo;

    public void setFoo(Integer foo) {
        this.foo = foo;
    }

    public Integer getFoo() {
        return foo;
    }
}
```

```java
class Foo4 {

    private Double foo;

    public void setFoo(Double foo) {
        this.foo = foo;
    }

    public Double getFoo() {
        return foo;
    }
}
```

```java
public static void 沒有泛型() {
    Foo3 a = new Foo3();
    a.setFoo(123);
    Integer a1 = a.getFoo();
    System.out.println("a1=" + a1); //123

    Foo4 b = new Foo4();
    b.setFoo(123.45);
    Double b1 = b.getFoo();
    System.out.println("b1=" +b1); //123.45
}
```

5. Java 中所有的類別都擴充自 Object，但由於傳回的是 Object，您必須轉換 它的介面，不轉換就會 Compiler 失敗
   萬一型態轉換有錯，還會發生 ClassCastException 當掉的例外

```
class Foo5 {

    private Object foo;

    public void setFoo(Object foo) {
        this.foo = foo;
    }

    public Object getFoo() {
        return foo;
    }
}
```

```
public static void 使用Object_需轉型() {
    Foo5 a = new Foo5();
    a.setFoo(123); //自動包裝
    Integer a1 = (Integer) a.getFoo();
    System.out.println(a1);

    Foo5 b = new Foo5();
    b.setFoo(123.45); //自動包裝
    Double b1 = (Double) b.getFoo();
    System.out.println(b1);

    Foo5 c = new Foo5();
    c.setFoo("abc");
    String c1 = (String) c.getFoo();
    System.out.println(c1);
}
```

```
public static void 使用Object_需轉型() {
    Foo5 a = new Foo5();
    a.setFoo(123); //自動包裝
    Integer a1 = (Integer) a.getFoo();
    System.out.println(a1);

    Foo5 b = n
    b.setFoo(1
    Double b1 = b.getFoo();
    System.out.println(b1);

    Foo5 c = new Foo5();
    c.setFoo("abc");
    String c1 = (String) c.getFoo();
    System.out.println(c1);
}
```

incompatible types: Object cannot be converted to Double

(Alt-Enter shows hints)

```
public static void 使用Object_型態轉換不正確_當掉() {

    Foo5 a = new Foo5();
    a.setFoo(123); //自動包裝
    Boolean a1 = (Boolean) a.getFoo(); //不小心轉成布林
    System.out.println(a1);
}
```

```
Exception in thread "main" java.lang.ClassCastException: java.lang.Integer cannot be cast to java.lang.Boolean
        at java17_泛型.Test17.使用Object_型態轉換不正確_當掉(Test17.java:44)
        at java17_泛型.Main.main(Main.java:8)
Java Result: 1
```

使用泛型

1. 由於 Java 中所有定義的類別，都以 Object 為最上層的父類別
2. 在 J2SE 5.0 之前，Java 程式設計人員可以使用 Object 來解決上面這樣的需求，為了讓定義出來的類別可以更加通用（Generic），傳入的值或傳回的物件都是以 Object 為 主，當您要取出這些物件來使用時，必須記得將介面轉換為原來的類型，這樣才可以操作物件上的方法。
3. 然而使用 Object 來撰寫泛型類別（ Generic Class ）留下了一個問題，因 為必須要轉換介面，粗心的程式設計人員往 往會忘了要作這個動作，或者是轉換介面時用錯了型態（像是該用 Boolean 卻用了 Integer），要命的是，語法上是可 以的，所以編譯器檢查不出錯誤，真正的錯誤要在執行時期才會發生，這時惱人的 ClassCastException 就會出來搞怪，
4. 在 J2SE 5.0 之後，提出了針對泛型（Generics ）設計的解決方案

```java
class GenericFoo1<T> {

    private T foo;

    public void setFoo(T foo) {
        this.foo = foo;
    }

    public T getFoo() {
        return foo;
    }
}
```

```java
//class GenericFoo1<Integer> {
//
//    private Integer foo;
//
//    public void setFoo(Integer foo) {
//        this.foo = foo;
//    }
//
//    public Integer getFoo() {
//        return foo;
//    }
//}
```

```java
//class GenericFoo1 <Double>{
//
//    private Double foo;
//
//    public void setFoo(Double foo) {
//        this.foo = foo;
//    }
//
//    public Double getx() {
//        return foo;
//    }
//}
```

```java
//class GenericFoo1<Boolean> {
//
//    private Boolean foo;
//
//    public void setFoo(Boolean foo) {
//         this.foo = foo;
//    }
//
//    public Boolean getx() {
//        return foo;
//    }
//}
```

```java
//class GenericFoo1 {
//
//    private Object foo;
//
//    public void setFoo(Object foo) {
//        this.foo = foo;
//    }
//
//    public Object getx() {
//       return foo;
//    }
//}
```

```java
//泛型定義的方式
public static void 泛型定義_有傳入型態時() {

    //GenericFoo1<Integer> a = new GenericFoo1<Integer>();
    //GenericFoo1<Integer> a = new GenericFoo1<>();
    GenericFoo1<Integer> a = new GenericFoo1();
    a.setFoo(4);
    Integer a1 = a.getFoo();
    System.out.println("a1=" + a1);

    //GenericFoo1<Double> b = new GenericFoo1<Double>();
    GenericFoo1<Double> b = new GenericFoo1<>();
    b.setFoo(4.5);
    Double b1 = b.getFoo();
    System.out.println("b1=" + b1);

    //GenericFoo1<Boolean> c = new GenericFoo1<Boolean>();
    GenericFoo1<Boolean> c = new GenericFoo1<>();
    c.setFoo(true);
    Boolean c1 = c.getFoo();
    System.out.println("c1=" + c1);
}
```

```
//                                                    ();
Ge  incompatible types: Boolean cannot be converted to String
c.  ----
    (Alt-Enter shows hints)
String c1 = c.getFoo();
System.out.println("c1=" + c1);
```

```
a1=4
b1=4.5
c1=true
```

如果使用泛型類別，但宣告時不一併指定型態呢？那　麼預設會使用 Object，要自己轉換物件的介面型態

```
public static void 泛型定義_沒有傳入型態時1() {
    //沒有傳入型態時 ，一律當成 Object
    GenericFoo1 a = new GenericFoo1();
    a.setFoo(123);
    Integer a1 = (Integer) a.getFoo(); //記得轉型 ，否則會Compiler 錯
    System.out.println("a1=" + a1);
}
```

```
public static void 泛型定義_沒有傳入型態時2() {
    //沒有傳入型態時 ，一律當成 Object
    GenericFoo1 a = new GenericFoo1();
    a.setFoo(123);
    Object a1 = a.getFoo();
    System.out.println("a1=" + a1);
}
```

限制泛型可用類型

1. 在定義泛型類別時，預設您可以使用任何的型態來實例化泛型類別中的型態持有者，但假設您想要限制使用泛型類別時，只能用某個特定型態或其子類別才能實例化型態持有者的話呢？
2. 可以在定義型態持有者時，使用"extends"指定這個型態持有者必須是擴充某個類型

```
class GenericFoo2<T extends Number> {

    private T foo;

    public void setFoo(T foo) {
        this.foo = foo;
    }

    public T getFoo() {
        return foo;
    }
}
```

```
public static void 限制泛型可用類型() {

    GenericFoo2<Integer> foo1 = new GenericFoo2<>();
    GenericFoo2<Double> foo2 = new GenericFoo2<>();
    GenericFoo2<String> foo3 = new GenericFoo2<>(); //字串不能 ，有限制型態
}
```

```
}

publ
```

```
type argument String is not within bounds of type-variable T
  where T is a type-variable:
    T extends Number declared in class GenericFoo2

incompatible types: cannot infer type arguments for GenericFoo2< >
    reason: inferred type does not conform to upper bound(s)
      inferred: String
      upper bound(s): Number
----
(Alt-Enter shows hints)
```

```
    GenericFoo2<String> foo3 = new GenericFoo2<>(); //字串不能 ，有限制型態
}
```

型態通配字元

```java
class GenericFoo1<T> {

    private T foo;

    public void setFoo(T foo) {
        this.foo = foo;
    }

    public T getFoo() {
        return foo;
    }
}
```

```java
class GenericFoo3<T extends String> {

    private T foo;

    public void setFoo(T foo) {
        this.foo = foo;
    }
    public T getFoo() {
        return foo;
    }
}
```

1. foo1 就只接受 GenericFoo1<Integer>的實例，而 foo2 只接受 GenericFoo1<Double>的實例
2. 希望有一個參考名稱 foo 可以接受所有下面的實例使用 “?” 通配字元，並使用 "extends" 關鍵字限定型態持有者的型態
3. 如果指定了不是實作 Number 的類別或其子類別，則編譯器會回報錯誤

```java
public static void 型態通配字元_兩個參考名稱() {

    GenericFoo1<Integer> foo1 = null;
    foo1 = new GenericFoo1<Integer>();
    GenericFoo1<Double> foo2 = null;
    foo2 = new GenericFoo1<Double>();
}
```

```java
public static void 型態通配字元_一個參考名稱() {

    GenericFoo1<? extends Number> foo1 = null;
    foo1 = new GenericFoo1<Integer>();
    foo1 = new GenericFoo1<Double>();
    foo1 = new GenericFoo1<String>();
    foo1 = new GenericFoo1(); //不傳入型態也 可以

    GenericFoo1<?> foo2 = null;
    foo2 = new GenericFoo1<Integer>();
    foo2 = new GenericFoo1<Double>();
    foo2 = new GenericFoo1<String>();
    foo2 = new GenericFoo1(); //不傳入型態也 可以

    GenericFoo1<? super Number> foo3 = null;
    foo3 = new GenericFoo1<Number>(); //只能接受 Number 上層
    foo3 = new GenericFoo1(); //不傳入型態也 可以
}
```

4. 如果您想要自訂一個showFoo()方法，方法的內容實作是針對Number而制定的，例如：

public void showFoo(GenericFoo1 foo) {

// 針對List而制定的內容

}

您當然不希望任何的型態都可以傳入showFoo()方法中，您可以使用以下的方式來限定，例如：

public void showFoo(GenericFoo1<? Extends Number> foo) {

//

}

5. 這麼一來，如果有粗心的程式設計人員傳入了您不想要的型態，例如GenericFoo1<String>型態的實例，則編譯器都會告訴它這是不可行的
6. 在宣告名稱時如果指定了**<?>**而不使用"extends"，則預設是允許 Object 及其下的子類，也就是所有的Java 物件了，那為什麼不直接使用GenericFoo1宣告就好了，何必要用GenericFoo1<?> 來宣告？因為 使用通配字元要注意的是，透過使用通配字元宣告的名稱所參考的物件，您沒辦法再對它加入新的資訊，您只能取得它的資訊或是移除它的資訊
7. 使用<?>或是<? extends SomeClass>的宣告方式，意味著您只能透過該名稱來取得所參考實例的資訊，或者是移除某些資訊，但不能增加它的資訊，因為只知道當中放置的是 SomeClass 的子類，但不確定是什麼類的實例，編譯器不讓您加入物件，理由是，如果可以加入物件的話，那麼您就得記得取回的物件實例是什麼型態，然後轉換為原來的型態方可進行操作，這就失去了使用泛型的意義
8. 除了可以向下限制，您也可以向上限制，只要使用"super"關鍵字，例如：GenericFoo1<? super StringBuilder> foo; 如此，foo 就只接受 StringBuilder 及其上層的父類型態之物件

```java
public static void 型態通配字元的限制() {
    //1.傳入型態
    GenericFoo1<Integer> a = new GenericFoo1<Integer>();
    a.setFoo(123); //可以加入
    a.setFoo(null); //可以移除
    Integer a1 = a.getFoo(); //可以取回
    System.out.println(a1);

    //2.傳入通配型態 Number
    GenericFoo1<? extends Number> b = null;
    b = new GenericFoo1<Integer>();
    // b.setFoo(123);  //不能加入
    b.setFoo(null); //可以移除
    Number b1 = b.getFoo(); //可以取回
    System.out.println(b1);
    b = new GenericFoo1<Double>();
    // b.setFoo(4.5); //不能加入 , 只能取回  或 移除
    b.setFoo(null);  //可以移除
    Number b2 = b.getFoo();  //可以取回要轉型
    System.out.println(b2);

    //3.傳入通配型態 Object
    GenericFoo1<?> c = null;
    c = new GenericFoo1<String>();
    // c.setFoo("abc");  //不能加入
    c.setFoo(null); //可以移除
    String c1 = (String) c.getFoo(); //可以取回
    System.out.println(c1);
    //4.傳入通配型態 String
    GenericFoo1<? super String> d = null; //向上繼承
    d = new GenericFoo1<String>();
    d.setFoo("abc");          //能加入
    d.setFoo(null);           //可以清除
    String f = (String) d.getFoo();    //可以取回

    //5.向上限制
    GenericFoo1 e = new GenericFoo1(); //沒有傳入型態
    e.setFoo(123); //可以加入
    e.setFoo(null); //可以移除
    Integer e1 = (Integer) e.getFoo(); //可以取回要轉型
    System.out.println(e1);
}
```

9. 這裡的 Fruit 是一個 Apple 的父類別的 List。同樣地，出於對類型安全的考慮，我們可以加入 Apple 對象或者其任何子類（如 RedApple）對象，但由於編譯器並不知道 List 的內容究竟是 Apple 的哪個父類別，因此不允許加入特定的任何超類型。
而當我們讀取的時候，編譯器在不知道是什麼類型的情況下只能返回 Object 對象，因為 Object 是任何 Java 類的最終祖先類。

```java
public class Fruit {

}

class Apple extends Fruit{

}

class RedApple extends Apple{

}
```

```java
public static void 向下限制() {

    List<Fruit> 水果 = new ArrayList<Fruit>();
    List<Apple> 蘋果 = new ArrayList<Apple>();
    List<RedApple> 紅蘋果 = new ArrayList<RedApple>();
    List<? extends Apple> 水果盤子 = null;
    水果盤子 = 水果; //不可以是蘋果的父親
    水果盤子 = 蘋果; //只要是蘋果或蘋果的小孩都可以
    水果盤子 = 紅蘋果; //只要是蘋果或蘋果的小孩都可以
    水果盤子.add(new Apple());     //compile error
    水果盤子.add(new RedApple()); //compile error
    水果盤子.add(new Fruit());     //compile error
    水果盤子.add(new Object());    //compile error
}
```

```java
public static void 向上限制() {

    List<Fruit> 水果 = new ArrayList<Fruit>();
    List<Apple> 蘋果 = new ArrayList<Apple>();
    List<RedApple> 紅蘋果 = new ArrayList<RedApple>();
    List<? super Apple> 水果盤子 = null;
    水果盤子 = 水果; //只要是蘋果或蘋果的父親都可以
    水果盤子 = 蘋果; //只要是蘋果或蘋果的父親都可以
    水果盤子 = 紅蘋果; //不可以是蘋果的小孩
    水果盤子.add(new Apple());     //work
    水果盤子.add(new RedApple()); //work
    水果盤子.add(new Fruit());     //compile error
    水果盤子.add(new Object());    //compile error
}
```

```java
public static void 型態通配字元_參數傳遞() {

    List<Object> x = new ArrayList<Object>();
    List y = new ArrayList();
    List<String> z = new ArrayList<String>();

    foo1(x);
    foo1(y);
    foo1(z);  //不行

    foo2(x);
    foo2(y);
    foo2(z);

    foo3(x);
    foo3(y);
    foo3(z);

    foo4(x);
    foo4(y);
    foo4(z);

    foo5(x); //不行
    foo5(y);
    foo5(z);
}
```

```java
public static void foo1(List<Object> list) {
    list.add(2);
    list.add("abc");
}

public static void foo2(List list) {
    list.add(2);
    list.add("abc");
}

public static void foo3(List<? extends Object> list) {
    list.add(2);
    list.add("abc");
}

public static void foo4(List<?> list) {
    list.add(2);
    list.add("abc");
}

public static void foo5(List<String> list) {
    list.add("xyz");
    list.add("abc");
}
```

```
public static void 泛型應用_型態通配字元1() {

    List<Integer> a = new ArrayList();
    a.add(123);
    a.add(456);
    印集合11(a);
    印集合12(a);

    List<Double> b = new ArrayList();
    b.add(12.34);
    b.add(45.98);
    印集合13(b);
    印集合14(b);
}
public static void 印集合11(List obj) {
    for (Object x : obj) {
        System.out.println((Integer) x + " ");
    }
}
public static void 印集合12(List<Integer> obj) {
    for (Integer x : obj) {
        System.out.println(x + " ");
    }
}
public static void 印集合13(List obj) {
    for (Object x : obj) {
        System.out.println((Double) x + " ");
    }
}
public static void 印集合14(List<Double> obj) {
    for (Double x : obj) {
        System.out.println(x + " ");
    }
}
```

```
public static void 泛型應用_型態通配字元2() {

    List<Integer> a = new ArrayList();
    a.add(123);
    a.add(456);
    印集合2(a);
    List<Double> b = new ArrayList();
    b.add(12.34);
    b.add(45.98);
    印集合2(b);
}

public static void 印集合2(List<? extends Number> obj) {
    for (Number x : obj) {
        System.out.println(x + " ");
    }
}
```

系統 API 的泛型定義與應用

```
//HashSet集合
//public class HashSet<E> extends AbstractSet<E>
//        implements Set<E>, Cloneable, java.io.Serializable {//
//
//    public HashSet() {
//        map = new HashMap<>();
//    }
//}
```

```
public static void 泛型應用_HashSet() {
    HashSet<Integer> aSet = new HashSet<Integer>();
    aSet.add(1);
    aSet.add(new Integer(2));
    aSet.add(new Integer(3));
    //aSet.add(new Double(3.5));
    System.out.println(aSet);
}
```

```
[1, 2, 3]
```

定義泛型的例子

```
//宣告多個類型持有者
class GenericFoo5<T1, T2> {

    private T1 foo1;
    private T2 foo2;

    public void setFoo1(T1 foo1) {
        this.foo1 = foo1;
    }

    public T1 getFoo1() {
        return foo1;
    }

    public void setFoo2(T2 foo2) {
        this.foo2 = foo2;
    }

    public T2 getFoo2() {
        return foo2;
    }
}
```

```
public static void 泛型應用_兩個型態() {
    GenericFoo5<Integer, Boolean> foo = new GenericFoo5<>();
    foo.setFoo1(123);
    Integer a = foo.getFoo1();
    foo.setFoo2(true);
    Boolean b = foo.getFoo2();
    System.out.println("a=" + a);
    System.out.println("b=" + b);
}
```

```
a=123
b=true
```

```
//陣列
class GenericFoo6<T> {

    private T[] fooArray;

    public void setFooArray(T[] fooArray) {
        this.fooArray = fooArray;
    }

    public T[] getFooArray() {
        return fooArray;
    }
}
```

```
public static void 泛型應用_陣列() {
    String[] str1 = {"caterpillar", "momor", "bush"};
    String[] str2;
    GenericFoo6<String> foo = new GenericFoo6<String>();
    foo.setFooArray(str1);
    str2 = foo.getFooArray();
    for (String s : str2) {
        System.out.println(s + " ");
    }
}
```

```
caterpillar
momor
bush
```

```java
//特定類型物件的容器
class SimpleCollection<T> {

    private T[] objArr;
    private int index = 0;

    public SimpleCollection() {
        objArr = (T[]) new Object[10];
    }
    public SimpleCollection(int capacity) {
        objArr = (T[]) new Object[capacity];
    }
    public void add(T t) {
        objArr[index] = t;
        index++;
    }
    public int getLength() {
        return index;
    }
    public T get(int i) {
        return (T) objArr[i];
    }
    public String toString() {

        StringBuilder s = new StringBuilder();
        s.append("[");

        for (int i = 0; i < index; i++) {
            if (i < index-1) {
                s.append(objArr[i]).append(",");
            } else {
                s.append(objArr[i]);
            }

        }
        s.append("]");
        return s.toString();
    }
}
```

```java
public static void 泛型應用_特定物件容器() {
    SimpleCollection<String> c = new SimpleCollection<>();
    c.add("aa");
    c.add("bb");
    c.add("cc");
    c.add("dd");
    System.out.println(c.toString());
}
```

```
[aa,bb,cc,dd]
```