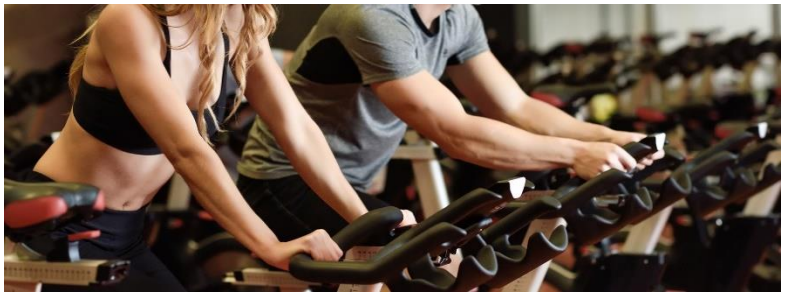
IECS273/274 [1111 3670/3671]
物件導向設計與實習

12 : Generic Class Collections,Maps,Iterators



01 Generic Class

Generic Class in Java

- Generics let you parameterize types. With this capability, you can define a class or a method with generic types that the compiler can replace with concrete types.
- In Java, a class with type parameters is called a **generic class**.

  public class GenClass <T> {…}

  - The definition of a generic class will contain a type parameter <T>, and will instantiate T as an existing type, such as Integer, String, Matrix, etc., to create an object of a specific type.
  - Java introduced generics in J2SE-5 to deal with collection-type-safe objects, which makes the code more stable by checking whether the type conforms at compile time. ***Generics enable you to detect errors at compile time rather than at runtime.***
  - Before generics, we could store any type of object in a collection, i.e. non-generic objects. Generics now force Java programmers to store specific types of objects.

Generic Class in Java

- Java has allowed you to define generic classes, interfaces, and methods since JDK 1.5.
- Several interfaces and classes in the Java API were modified using generics. The **java.lang.Comparable** interface was modified in JDK 1.5 with a generic type.
  - Here, <T> represents a formal generic type, which can be replaced later with an actual concrete type. Replacing a generic type is called a generic instantiation.
  - By convention, a single capital letter such as E or T is used to denote a formal generic type.

```
package java.lang;

public interface Comparable {
  public int compareTo(Object o)
}
```

(a) Prior to JDK 1.5

```
package java.lang;

public interface Comparable<T> {
  public int compareTo(T o)
}
```

(b) JDK 1.5

Generic1.java

```
public class Generic1<T> { // T is a type parameter.
  private T elem; // Data member of type T.

  void set(T a) {this.elem=a;} // Set method.

  T get() {return elem;} // Get method
}
```

Generic1_App.java

```
import java.lang.Float;

public class Generic1_App {
  public static void main(String args[]) {
    Generic1<Integer> int1=new Generic1<Integer>(); // Instantiate as Integer class.
    int1.set(100); // Set an integer value.
    //int1.set("test"); // Adding string elements will cause compilation errors.
    System.out.println("Integer type:\t" + int1.get()); // Output an integer.

    Generic1<Float> flt1 = new Generic1<Float>(); // Instantiate as Float class.
    flt1.set(Float.parseFloat("3.14159f")); // Set a floating point number value.
    System.out.println("Float type:\t" + flt1.get()); // Output an floating point number.

    Generic1<String> str1 = new Generic1<String>(); // Instantiate as String class.
    str1.set("This is a string"); // Set a string.
    System.out.println("String type:\t" + str1.get()); // Output a string.
  }
}
```

Generic1_App.class

```
Integer type:    100
Float type:      3.14159
String type:     This is a string
```

GenericStack.java

```
public class GenericStack<E> {
  private java.util.ArrayList<E> list = new java.util.ArrayList<>();
  public int getSize() {
                    return list.size();
                   }
  public E peek() {
                    return list.get(getSize() − 1);
                   }
  public void push(E o) {
                   list.add(o);
                   }
  public E pop() {
                   E o = list.get(getSize() − 1);
                   list.remove(getSize() − 1);
                   return o;
                   }
  public boolean isEmpty() {
                    return list.isEmpty();
                   }
  @Override
  public String toString() {
                   return "stack: " + list.toString();
                   }
}
```

Generic Class in Java

- Advantages of Java Generics:
  - Type safety: we can only store objects of a single type in a generic; it does not allow storing other objects.
  - No cast required: no need to cast object type.
  - Compile-time checking: Types are checked at compile time, so there will be no problems at runtime. A good programming strategy shows that it is much better to deal with problems at compile time than at runtime.

Generic Class in Java

- The type parameter naming convention is very important for learning about generics. Common type parameters are as follows:
  - T - Type,
  - E - Element,
  - K - Key,
  - N - Number,
  - V - Value,

Generic2.java

```
import java.util.ArrayList;
import java.util.List;

public class Generic2 {
  @SuppressWarnings("unchecked")
  public static void main(String[] args) {
    @SuppressWarnings("rawtypes")
    List list1 = new ArrayList(); // not using generics
    list1.add(10); // add integer element
    list1.add("10"); // add string element
    System.out.println("Output integer value list1.get(0):\t" + list1.get(0)); // Output integer value
    System.out.println("Output string  value list1.get(1):\t" + list1.get(1) + "\n"); // Output string value

    List<Integer> list2 = new ArrayList<Integer>(); // Use generics
    list2.add(10); // Only integer elements can be added
    System.out.println("Output integer valuelist2.get(0):\t" + list2.get(0)); // Output integer value
    // list2.add("10"); // Adding string elements will cause compilation errors
  }
}
```

Generic2.class

```
Output integer value list1.get(0): 10
Output string    value list1.get(1): 10

Output integer valuelist2.get(0):  10
```

Generic3java

```
import java.util.ArrayList;
import java.util.List;

public class Generic3 {
   @SuppressWarnings("unchecked")
   public static void main(String[] args) {
     @SuppressWarnings("rawtypes")
     List list1 = new ArrayList(); // not using generics
     list1.add("list1[0]: This a string."); // add string element
     String str1 = (String) list1.get(0); // get a string element, need casting
     System.out.println("Output string str1:\t" + str1); // output string

     List<String> list2 = new ArrayList<String>(); // using generics
     list2.add("list2[0]: This a string."); // add string element
     String str2 = list2.get(0); // get a string element, need no casting
     System.out.println("Output string str2:\t" + str2); // output string
   }
}
```

Generic3.class

```
Output string str1: list1[0]: This a string.
Output string str2: list2[0]: This a string.
```

02 Collections

Collection Classes

- ***A collection is a data structure for holding elements.*** For example, an ArrayList<T> object is a collection. Java has a repertoire of interfaces and classes that give a uniform treatment of collections.
- A Java **collection** is a *framework* that provides an architecture for *storing* and *manipulating groups of objects*.
- Java collections can implement all operations performed on data, such as *searching*, *sorting*, *insertion*, *manipulation*, and *deletion*.
- A Java collection means *a unit of objects*. The main collection interfaces are Collection<T>, Set<T>, and List<T>.
  - The Set<T> and List<T> interfaces extend the Collection<T> interface. The library classes that are standard to use and that implement these interfaces are HashSet<T>, which implements the Set<T> interface, and ArrayList<T>, which implements the List<T> interface.
  - A Set<T> does not allow repeated elements and does not order its elements. A List<T> allows repeated elements and orders its elements.

ArrayList Class

- In Java, the ArrayList class uses a *dynamic array* to store elements. The ArrayList class inherits the AbstractList class and implements the List interface.
- Key features of the ArrayList class :
  - The ArrayList class can contain duplicate elements.
  - The ArrayList class maintains insertion order.
  - The ArrayList class is *not synchronized*. (**Synchronization** in Java is the ability to control access to any *shared resource* by *multiple threads*.)
  - The ArrayList class allows *random access* because arrays work on an index basis.
  - In the ArrayList class, the operation is *slow*; because if any element is *removed* from the array list, a lot of *shifting is required*.

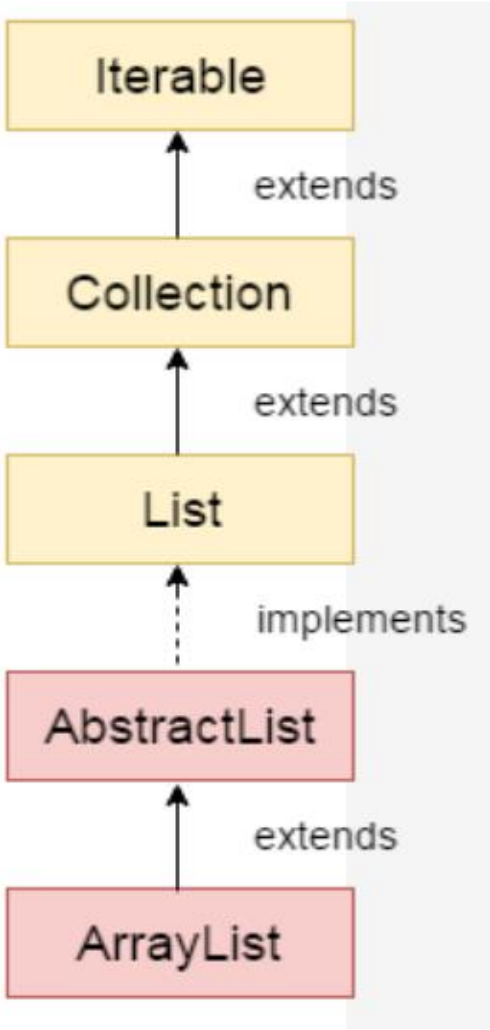
ArrayList Class

- Hierarchy of the ArrayList class:
  - ArrayList class inherits AbstractList class;
  - AbstractList class implements List interface;
  - List interface inherits Collection and Iterable interface.
  - java.util.ArrayList class declaration:

    public class ArrayList<E> **extends** AbstractList<E>
    implements List<E>, RandomAccess,
    Cloneable, Serializable

    <E> is the *parameterized* type of
    a Java **generic type**.

    <E> is the element type of ArrayList,
    such as: **byte**, **char**, **int**, **double**,
    string, complex, etc.。



ArrayList Class

- ArrayList class constructors:
  - ArrayList(): create empty array list.
  - ArrayList(Collection<? extends E> c): create an array list and initializes the array list elements with collection c.
  - ArrayList(**int** capacity): create an array list and set its initial size as capacity.
- ArrayList class methods:
  - **void** add(**int** index, E element): insert element at position index in an array list.
  - **boolean** add(E e): add element e to the end of the array list.
  - **boolean** addAll(Collection<? extends E> c): append all elements of collection c to the end of the array list, in the order of the *iterator* of collection c.
  - Object[] toArray(): return an array of all elements in the order of the array list.
  - **boolean** contains(Object o): return **true** if the array list contains element o.
  - E remove(**int** index): remove the element at index in the array list.

ArrayList Class

- **boolean** remove(Object o): remove the first occurrence of element o and return **true**; if element o does not appear, return false.
- **boolean** removeAll(Collection<?> c): remove all elements from an array list.
- Eset(**int** index, E element): replace the element at position index in the array list with element.
- **void** trimToSize(): resize the capacity of the array list instance to the current size of the list.
- **void** clear(): clear all elements of an array list.
- **void** ensureCapacity(**int** requiredCapacity): increase capacity of array list instance.
- **int** size(): increase capacity of array list instance.
- E get(**int** index): get the element with index from the array list.
- **boolean** isEmpty(): return **true** if the list is empty; otherwise, return **false**.
- **int** indexOf(Object o): return the index in the array list of the first occurrence of element o, or -1 if the array list does not contain the element.
- **int** lastIndexOf(Object o): return the index of the last occurrence of the specified element o in the list, or -1 if the list does not contain the element.

Wildcard of Collections Classes

- Classes and interfaces in the collection framework use some parameter type specifications that we have not seen before. For example, they allow you to say things such as, “The argument must be a ArrayList<T>, but it can have any base type.”
- More generally these new parameter type specifications use generic classes but do not fully specify the type plugged in for the type parameter. Because they specify a wide range of argument types, they are known as **wildcards**.
- The easiest wildcard to understand is <?>, which says that you can use any type in place of the type parameter. For example,

  public void sampleMethod(String arg1, ArrayList<?> arg2)

  is invoked with two arguments.

Sorting of Collections Classes

- The Collections framework represents a unified architecture for storing and manipulating a set of objects. It has interfaces and their implementations (i.e. classes) and algorithms.

```
public class Collections {
    // Suppress default constructors to ensure non-instantiable.
    private Collections() { }
… }
```

  - Algorithms that define various subclasses of Collections, such as sort, search, iterator, minimum, maximum, etc.
- Sorts the elements of an array list according to the specified comparator.

```
public static <T extends Comparable<? super T>> void sort(List<T> list)
```

  - Example: Sort the elements of Arraylist list，

```
ArrayList<String> list = new ArrayList<String>();
…
Collections.sort(list);
```

CollectionArrayListEnglish.java

```
import java.util.ArrayList;
import java.util.Collections;
import java.util.Iterator;

public class CollectionArrayListEnglish {

   public static void main(String[] args) {
      ArrayList<String> list = new ArrayList<String>(); // Create an empty string array list.
      list.add("London"); // Add element to array list.
      list.add("Moscow");

      System.out.println(">>> List of the original capital cities: " + list);
      System.out.println();

      System.out.println("*** Add Amsterdam to the index 1 position.");
      list.add(1, "Amsterdam");
      System.out.println(">>> New list: " + list);
      System.out.println();

      ArrayList<String> rest = new ArrayList<String>(); // Creates an empty temporary array list of stringst.
      rest.add("Paris"); // Add two capital cities to the temporary list.
      rest.add("Bangkok");
```

CollectionArrayListEnglishjava

```
System.out.println("*** Add the two capital cities at the end of the list.");
list.addAll(rest);
System.out.println(">>> Capital cities: " + list);
System.out.println();

rest.clear(); // Clear temporary list.
System.out.println("*** Add two capital cities after Paris.");
rest.add("Manila"); // Add two capital cities.
rest.add("Tokyo");
list.addAll(4, rest);
System.out.println(">>> Capital cities: " + list);
System.out.println();

list.ensureCapacity(9); // Expand the list to 9 city names.

System.out.println("*** Add the last two cities:list.add(7, \"Seoul\");list.add(7, \"Jakarta\");");
list.add(7, "Seoul");
list.add(7, "Jakarta");
System.out.println(">>> Number of capital citis: " + list.size());
System.out.println(">>> The 8th capital city: " + list.get(7));
System.out.println(">>> The 9th capital city: " + list.get(8));
```

CollectionArrayListEnglish.java

```
        System.out.println(">>> Current capital cities: " + list);
        System.out.println();

        Collections.sort(list);
        System.out.println(">>> Sorted capital cities: " + list);
        System.out.println();
        }
    }
}
```

CollectionArrayListEnglish.class

```
>>> List of the original capital cities: [London, Moscow]

*** Add Amsterdam to the index 1 position.
>>> New list: [London, Amsterdam, Moscow]

*** Add the two capital cities at the end of the list.
>>> Capital cities: [London, Amsterdam, Moscow, Paris, Bangkok]

*** Add two capital cities after Paris.
>>> Capital cities: [London, Amsterdam, Moscow, Paris, Manila, Tokyo, Bangkok]

*** Add the last two cities:list.add(7, "Seoul");list.add(7, "Jakarta");
>>> Number of capital citis: 9
>>> The 8th capital city: Jakarta
>>> The 9th capital city: Seoul
>>> Current capital cities: [London, Amsterdam, Moscow, Paris, Manila, Tokyo, Bangkok, Jakarta, Seoul]

>>> Sorted capital cities: [Amsterdam, Bangkok, Jakarta, London, Manila, Moscow, Paris, Seoul, Tokyo]
```

03 Maps

Java Map

- The Java **Map** framework is similar in character to the collection framework, except that it ***deals with collections of ordered pairs***.
- Objects in the map framework can implement mathematical functions and relations and so can be used to construct database classes. Think of the pair as consisting of a key K (to search for) and an associated value V.
- The Map<K,V> interface is used to store a mapping between a key K and a value V. It is commonly used to store databases in memory. The HashMap<K,V> class is a standard library class that implements a map.

Hash Map

- The **Map interface** stores a set of **key-value** objects, providing a mapping from *key* to *value*.
  - HashMap is a *hash table* that stores a mapping of key-value pairs (key, value).
  - HashMap implements the Map interface, stores data according to the HashCode value of the key, has fast access speed, allows the key of at most one record to be null, and does not support thread synchronization.
  - HashMap is unordered, i.e. the order of insertion is not recorded.
  - The key and value types of HashMap can be the same or different, they can be the key and value of String type, or the key of Integer type and the value of String type.

Generic4java

```
import java.util.HashMap;
import java.util.Iterator;
import java.util.Map;
import java.util.Set;

public class Generic4 {

   public static void main(String[] args) {
     // Create a city hash table with keys of integer class and values of strings。
     Map<Integer, String> city = new HashMap<Integer, String>();
     // Create a population hash table with keys of  integer class and values of double class.
     Map<Integer, Double> popu = new HashMap<Integer, Double>();

     city.put(1, "New York"); // Add multiple cities in key order。
      city.put(2, "Boston");
      city.put(3, "Chicago");
      city.put(4, "Miami");
      city.put(5, "Denver");
      city.put(6, "Houston");
      city.put(7, "Dallas");
      city.put(8, "Los Angeles");
      city.put(9, "Seattle");
```

Generic4java

```
    popu.put(1, 8.80); // Add multiple population integer values out of key order.
    popu.put(2, 0.70);
    popu.put(4, 6.14);
    popu.put(5, 0.72);
    popu.put(8, 3.97);
    popu.put(7, 1.30);
    popu.put(9, 0.78);
    popu.put(3, 2.75);
    popu.put(6, 2.30);
    // Use Map.Entry to implement Set and Iterator.
    Set<Map.Entry<Integer, String>> set1 = city.entrySet();
    Set<Map.Entry<Integer, Double>> set2 = popu.entrySet();
    Iterator<Map.Entry<Integer, String>> itr1 = set1.iterator();
    Iterator<Map.Entry<Integer, Double>> itr2 = set2.iterator();
    System.out.println("    City\tPopulation");
    while (itr1.hasNext()) { // Get mapping of the set
      Map.Entry<Integer, String> e = itr1.next(); // No casting
      Map.Entry<Integer, Double> p = itr2.next(); // No casting
      System.out.println(e.getKey() + "  " + e.getValue() + "\t" + p.getValue() + " million");
    }
  }
}
```

Generic4.class

	City	Population
1	New York	8.8 million
2	Boston	0.7 million
3	Chicago	2.75 million
4	Miami	6.14 million
5	Denver	0.72 million
6	Houston	2.3 million
7	Dallas	1.3 million
8	Los Angeles	3.97 million
9	Seattle	0.78 million

04 Iterators

Java Iterator

- An **iterator** is an object that is used with a collection to provide ***sequential access*** to the elements in the collection. That is, an iterator is something that allows you to examine and possibly modify the elements in a collection in ***some sequential order***.
- An iterator is something that allows you to examine and possibly modify the elements in a collection in some sequential order. Java formalizes this concept with the two interfaces Iterator<T> and ListIterator<T>.
  - An Iterator<T> (with only the required methods implemented) goes through the elements of the collection in one direction only, from the beginning to the end.
  - A ListIterator<T> can move through the collection list in both directions, forward and back. A ListIterator<T> has a set method; the Iterator<T> interface does not require a set method.

Iterator Interface

- Iterator interface is the root interface of all Collection classes. The Collection interface extends the Iterator interface, so all subclasses of the Collection interface also implement the Iterable interface.
  - Iterator is a way to visit the all elements of a collection:

    Iterator<E> iterator();

    Example: Iterator <String> iter = (ArrayList L).iterator();
- Methods of Iterator interface:
  - **boolean** hasNext(): **true** if the iterator has more elements; **false** otherwise.
  - Object next(): return element Object of the collection and moves the iterator pointer to the next element.
  - **void** remove(): remove the previous element returned by the iterator. (less used)

CollectionArrayListChinese.java

```
import java.util.ArrayList;
import java.util.Collections;
import java.util.Iterator;

public class CollectionArrayListChinese {

   public static void main(String[] args) {
      ArrayList<String> list = new ArrayList<String>(); // Create an empty string array list.
      list.add("台北"); // Add elements to array list.
      list.add("台中");

      System.out.println(">>> Original list: " + list);
      System.out.println();

      System.out.println("*** Add 新北 to the city at position 1.");
      list.add(1, "新北");
      System.out.println(">>> New list: " + list);
      System.out.println();

      ArrayList<String> rest = new ArrayList<String>(); // Creates an empty temporary array list of strings.
      rest.add("基隆"); // Add to more cities.
```

CollectionArrayListChinese.java

```
rest.add("桃園");
System.out.println("*** Add two cities at the end of the list.");
list.addAll(rest);
System.out.println(">>> Current list of five cities: " + list);
System.out.println();

rest.clear(); // Clear the temporary array list.
System.out.println("*** Add two cities after 桃園.");
rest.add("新竹"); // 加上其他八闽的两个市。
rest.add("苗栗");
list.addAll(4, rest);
System.out.println(">>> Current list of seven cities: " + list);
System.out.println();

list.ensureCapacity(9); // Expand the list to nine cities。

System.out.println("*** Add two more cities: list.add(7, \"台南\");list.add(7, \"高雄\");");
list.add(7, "台南");
list.add(7, "高雄");
System.out.println(">>> The number of cities: " + list.size());
System.out.println(">>> The 8th city: " + list.get(7));
```

CollectionArrayListChinese.java

```
        System.out.println(">>> The 9th city: " + list.get(8));
        System.out.println(">>> The final list of nine cities: " + list);
        System.out.println();

        Collections.sort(list);
        System.out.println("*** Java may not follow Unicode to sort data.");
        System.out.println(">>> The sorted list of nine cities: " + list);
        System.out.println();

        Iterator<String> iter = list.iterator();
        while (iter.hasNext()) {
           String city = iter.next().toString();
           byte code[] = city.getBytes();
           System.out.printf("%s：%02X %02X %02X %02X\n", city, code[0], code[1], code[2], code[3]);
        }
     }
}
```

CollectionArrayListChinese.class

```
>>> Original list: [台北, 台中]

*** Add 新北 to the city at position 1.
>>> New list: [台北, 新北, 台中]

*** Add two cities at the end of the list.
>>> Current list of five cities: [台北, 新北, 台中, 基隆, 桃園]

*** Add two cities after 桃園.
>>> Current list of seven cities: [台北, 新北, 台中, 基隆, 新竹, 苗栗, 桃園]

*** Add two more cities: list.add(7, "台南");list.add(7, "高雄");
>>> The number of cities: 9
>>> The 8th city: 台南
>>> The 9th city: 高雄
>>> The final list of nine cities: [台北, 新北, 台中, 基隆, 新竹, 苗栗, 桃園, 台南, 高雄]

*** Java may not follow Unicode to sort data.
>>> The sorted list of nine cities: [台中, 台北, 台南, 基隆, 新北, 新竹, 桃園, 苗栗, 高雄]
```

CollectionArrayListChinese.class

台中：CC A8 D6 D0
台北：CC A8 B1 B1
台南：CC A8 C4 CF
基隆：BB F9 C2 A1
新北：D0 C2 B1 B1
新竹：D0 C2 D6 F1
桃園：CC D2 88 40
苗栗：C3 E7 C0 F5
高雄：B8 DF D0 DB