Advanced Exercise on Computer and Information Science B

Collection Framework

Generics

Java Collection Framework

- What is Collection?
  - Collection is a class libraries that enables to store and manipulate groups of objects
- Before JDK1.2
  - Vector
    - enables to store objects while keeping the order of the objects inserted. Duplication is allowed.
  - Hashtable
    - enables to store object by using key. Duplication is not allowed.

Java Collection Framework

- Introduced from JDK1.2
  - Vector, Hashtable are included.
  - Map – Objects are stored by key. Duplication is NOT allowed.
  - List – Objects are stored with the order. Duplication is allowed
  - Set – Objects are stored with the order. Duplication is NOT allowed

Main classes

- List
  - ArrayList
  - LinkedList
  - Vector

```
List list = new ArrayList();
list.add("test")
Object obj = list.get(1);
Iterator ite = list.iterator();
while(ite.hasNext() {…}
```

- Set
  - HashSet
  - TreeSet

```
Set set = new HashSet();
set.add("test");
if(set.contains("test")) {…}
```

- Map
  - HashMap
  - TreeMap

```
Map map = new HashMap();
map.put("test", 1);
map.get("test");
```

How to differentiate the usages of Vector and ArrayList

- Vector
  - Does not synchronized
- ArrayList
  - Synchronized
- Summary
  - Vector should not be used
  - ArrayList with synchronized mechanism
    - Collections.synchronizedList can be used.

Foundations of Collection Framework

- Any types can be accepted.
  - All classes extend a type(class) of Object
- Object is a only type that can be put and got
  - (Down) case is required to manipulate arbitrary types

```
String str = (String)list.get(10);
```

- Is it OK?
  - NO! it cannot be accepted at all!
    - To be continued…

Generics(from Java5)

- Objective
  - Eliminate down cast
  - Evaluate types at NOT runtime but compiling
- Extends Java language syntax
  - Introduce type parameter
  - Guarantees types at compile time
  - But what is type?
    - Is it safe if introducing type?

Concepts of generics

Source code

```
Class Sample<E> {
  void method(E e);
}
```

Compile

Type is undefined when this class is defined. This type at least extends Object type

```
Sample<String> ss =
    new Sample<String>();
```

At compile time, ss becomes an instance of Sample with type String

```
Sample<Integer> si =
    new Sample<Integer>();
```

At compile time, si becomes an instance of Sample with type Integer

Use of generics

- Use a class with generics
  - Usually use collection framework such as List
  - Generally, easy
- Design a class with generics
  - Design our own classes with generics
  - Requires correct understanding of generics
  - Difficult!

Use of generics – in case of List

- List is defined by List<E>
  - Compare List of JDK1.4 with that of JDK 1.5
  - See real code

A kind of usage of generics

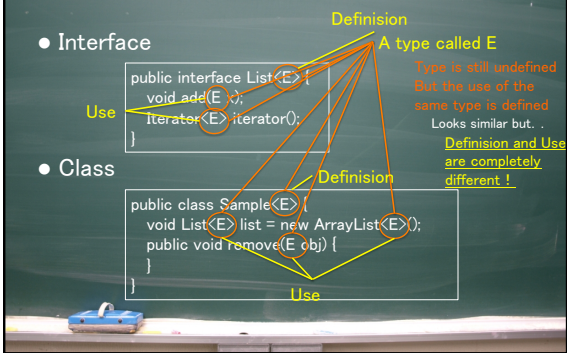
- Generics in interface and class
- Generics in method
- Boundaries of generics
  - Upper/Bottom boundary
- Wild card
  - Type erasure
  - Usage of upper bound and bottom bound

A kind of usage of generics

- Generics in interface and class
- Generics in method
- Boundaries of generics
  - Upper/Bottom boundary
- Wild card
  - Type erasure
  - Usage of upper bound and bottom bound

Definision of generics –Class and Interface

- Interface
- Class



A kind of usage of generics

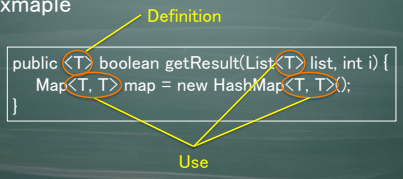
- Generics in interface and class
- Generics in method
- Boundaries of generics
  - Upper/Bottom boundary
- Wild card
  - Type erasure
  - Usage of upper bound and bottom bound

Definition of generics – method

- Definition of method

modifier <type parameter> return type method name (arguments)

- Exmaple



A kind of usage of generics

- Generics in interface and class
- Generics in method
- Boundaries of generics
  - Upper/Bottom boundary
- Wild card
  - Type erasure
  - Usage of upper bound and bottom bound

Boundaries of generics

- Type parameter (e.g., T,V) cannot be used until concrete types are assigned.
- Upper boundary
  - T extends Number
    - T at least extends Number
- Type parameter with upper boundary

public <? extends T> void xxx(T t) ⇒NG

public <T extends Number> void xxx(T t)

<T> void xxx(T extends String arg)⇒NG

<T extends String> void xxx(T arg)

Covariant

- Intuitive

```
Integer i = new Integer(1);
Number n = new Number(2);
n = i;
```

→ Covariant

- However, in generics

```
List<Integer> ilist = new ArrayList<Integer>();
List<Number> nlist = new ArrayList<Number>();
nlist = ilist;
```

NG → NOT Covariant

```
nlist.add(new Integer(10));
```

→ Possible

Wildcard(?)の登場

- Non-covariant means…
  - If a set of parameters has parent-child relationship, classes that has the type parameters are not the same.
- Then. .
  - We want「Please ignore type parameters」
- We now have Wildcard!

A kind of usage of generics

- Generics in interface and class
- Generics in method
- Boundaries of generics
  - Upper/Bottom boundary
- Wild card
  - Type erasure
  - Usage of upper bound and bottom bound

Usage of Wildcard

- Advantage
  - Can assign an object without a care of type parameter

```
List<?> list1 = new ArrayList<Integer>();
List<?> list2 = new ArrayList<String>();
```

- Disadvantage
  - Cannot add any objects
  - Type information is missing when getting an value

```
List<?> list1 = new ArrayList<Integer>();
list1.add(new Integer(1));
```

NG

Type Erasure

- Class with type parameter in which the wildcard is specified

List Just list　　List<?> Erasure of List

- Difference?
  - Erasure
    - Type is undefined, however, a certain type should be specified and it is guaranteed.
  - abbreviated
    - For compatible of up to JDK1.4

What is the wildcard for ?

- Does the list that does not accept insert operation has meanings?
  - No meaning when doing assignment
- Use as an argument of a method
  - A concrete type should be specified when creating
  - Only get operations should be written in a method

```
List<String> list = new ArrayList<String>();
List.add("Hello World");
method(list);
void method(List<?> list) {
    ……
}
```

What is the wildcard for ?

- What operations do we have to write in a method ?
  - Type of object is undefined
  - The object should be handled as Object type
  - Introduce boundary!
- Usage of Upper/Bottom boundary

example)

```
public void xxx(List<? extends Number> list)
public <T extends Number> void xxx(List<? extends T> list)
public <T extends U> void xxx(List<? extends T> list)
```

Usage of erasure

- Used as an argument of a method, then getting objects
  - Type checking is not applied
  - Use cast operation
  - Erasure may be deprecated

```
public void process (DataHolder<?, ?> data) {
    String str = (String)data.getE1();
    List<Number> list =
                =(List<Number>)data.getE2();
}
```

```
Public class DataHolder<E1, E2> {
    private E1 element1;
    public void setE1(E1 element1);
    public E1 getE1();
}
```

Tips

- Wildcard should be used in the argument

  List<? Extends String> list = new ArrayList<String>()

  Syntax is accepted but no meaning…
- A class that uses type parameter should be used at least as erasure
- Specify a concrete type when inserting, relax when getting
  - Concrete type needs to be decided when inserting
  - Use upper boundary when getting