CSE3040 Java Language

Lecture 14: Generic Programming

Dept. of Computer Engineering,

Sogang University

This material is based on the book "Core JAVA" and "Java의 정석". Do not post it on the Internet.

Generic Programming

- A technique for reusing the same code for various types of objects.

- Suppose you need to define a class BoxA for class A objects.

```
class BoxA {
    A item;
    void setItem(A item) { this.item = item; }
    A getItem() { return item; }
}
```

- Also, you need to define another class BoxB for class B objects.

```
class BoxB {
    B item;
    void setItem(B item) { this.item = item; }
    B getItem() { return item; }
}
```

- These two classes are basically the same, except they deal with different type of objects.

Generic Programming

- In order to avoid defining separate classes, you can do this:

```
class Box {
    Object item;
    void setItem(Object item) { this.item = item; }
    Object getItem() { return item; }
}
```

  - Since class Object is a superclass of every class, you can use class Box for objects of type A and also type B.
  - However, since you are using the same class for different types of objects, you always need to check whether the variable item is referring to a class A object or a class B object.
  - Also, when assigning the item to another variable, you will always need to use type casting.

```
Box b = new Box();
b.setItem(new Object());
b.setItem("ABC");
String item = (String)b.getItem();
System.out.println(item);
```

Generic Classes

- You can make class Box a generic class.
  - T is called a type variable.

```
class Box<T> {
    T item;
    void setItem(T item) { this.item = item; }
    T getItem() { return item; }
}
```

  - When you create an instance of class Box, you can decide the type for type variable T.

```
Box<String> b = new Box<String>();
b.setItem(new Object());    // this line causes error.
b.setItem("ABC");
String item = b.getItem();  // type casting not necessary.
System.out.println(item);
```

  - Then, the type of variable item is fixed to String.
  - You cannot assign other types to variable item.
  - You do not need type casting when assigning item to a String type variable.

Generic Classes

- You can define a class with multiple type variables.

```
class Entry<K, V> {
    private K key;
    private V value;
    public Entry(K key, V value) {
        this.key = key;
        this.value = value;
    }
    public K getKey() { return key; }
    public V getValue() { return value; }
}
```

```
Entry<String, Integer> entry = new Entry<String, Integer>("Fred", 42);
```

- If the compiler can figure out the type variables, you can omit them.

```
Entry<String, Integer> entry = new Entry<>("Fred", 42);
```

Generic Classes: Terms

```
class Box<T> {
    T item;
    void setItem(T item) { this.item = item; }
    T getItem() { return item; }
}
```

- Box<T>: a generic class. called "T Box" or "Box of T"
- T: a type variable
- Box: a raw type

```
Box<String> b = new Box<String>();
```

- String: a parameterized type

Generic Classes: Limitations

- Cannot define a static variable of type T.
  - Because a static variable is shared among all instances, which could be parameterized to different types.

```
class Box<T> {
    static T item;     // Error: cannot define a static variable of type T.
    void setItem(T item) { this.item = item; }
    T getItem() { return item; }
}
```

  - cannot create an array of type T using new.
    - Also, cannot use instanceof with T because of the same reason.

```
class Box<T> {
    T[] itemArr;     // This is OK.
    T[] createArray() {
        T[] tmpArr = new T[10];    // Error: cannot create a generic array of T
        return tmpArr;
    }
}
```

  - At compile time, <T> is removed and T is converted to Object.
    - Creating instances of T using new is disallowed to prevent logical errors.

Generic Classes: Limitations

- Cannot create a generic array.

```
class Box<T> {
    final T x;
    Box(T x) { this.x = x; }
}
```

```
Box<String>[] bsa = new Box<String>()[3];    // Error: cannot create a generic array of type Box<String>
Object[] oa = bsa;
oa[0] = new Box<Integer>(3);
String s = bsa[0].x;
```

- Cannot use a primitive type as a parameterized type.

```
Box<int> intBox = new Box<int>();    // Error
```

Generic Classes: Example 1

- Class definitions

```
import java.util.ArrayList;

class Fruit               { public String toString() { return "Fruit"; } }
class Apple extends Fruit { public String toString() { return "Apple"; } }
class Grape extends Fruit { public String toString() { return "Grape"; } }
class Toy                 { public String toString() { return "Toy"  ; } }

class Box<T> {
    ArrayList<T> list = new ArrayList<T>();
    void add(T item) { list.add(item); }
    T get(int i) { return list.get(i); }
    int size() { return list.size(); }
    public String toString() { return list.toString(); }
}
```

Generic Classes: Example 1

- main method

```
public class Lecture {
    public static void main(String[] args) {
        Box<Fruit> fruitBox = new Box<Fruit>();
        Box<Apple> appleBox = new Box<Apple>();
        Box<Toy>   toyBox   = new Box<Toy>();
//      Box<Grape> grapeBox = new Box<Apple>();   // Error: wrong type

        fruitBox.add(new Fruit());
        fruitBox.add(new Apple());

        appleBox.add(new Apple());
        appleBox.add(new Apple());
//      appleBox.add(new Toy());    // Error: cannot add Toy to Box<Apple>

        toyBox.add(new Toy());
//      toyBox.add(new Apple());    // Error: cannot add Apple to Box<Toy>

        System.out.println(fruitBox);
        System.out.println(appleBox);
        System.out.println(toyBox);
    }
}
```

Generic Classes: Example 1

- When creating an instance, the parameterized type must match that of the constructor.
  - Box<Apple> appleBox = new Box<Apple>();   // OK
  - Box<Apple> appleBox = new Box<Grape>();   // Error

- Even if the parameterized types are super-sub classes, different parameterized types are not allowed.
  - Assume class Apple is a subclass of class Fruit.
  - Box<Fruit> appleBox = new Box<Apple>();   // Error

- If the parameterized type is the same, creating an instance of a subclass raw type is possible
  - Assume class FruitBox is a subclass of class Box.
  - Box<Apple> appleBox = new FruitBox<Apple>();   // OK

Generic Classes: Example 1

- Since the parameterized type must match, you can omit the parameterized type when calling the constructor.
  - Box<Apple> appleBox = new Box<Apple>();
  - Box<Apple> appleBox = new Box<>();   // same as the above statement

- When calling instance method add, the type must match the parameterized type.
  - Box<Apple> appleBox = new Box<Apple>();
  - appleBox.add(new Apple());   // OK
  - appleBox.add(new Grape());   // Error

- However, you can assign a subclass of a parameterized type.
  - Assume class Apple is a subclass of class Fruit.
  - Box<Fruit> fruitBox = new Box<Fruit>();
  - fruitBox.add(new Fruit());   // OK
  - fruitBox.add(new Apple());   // OK

Generic Classes: Limiting Types

- Suppose you want to create a generic class FruitBox.

```
class FruitBox<T> {
    ArrayList<T> list = new ArrayList<T>();
    void add(T item) { list.add(item); }
    T get(int i) { return list.get(i); }
    int size() { return list.size(); }
    public String toString() { return list.toString(); }
}
```

- Then, it is possible to create a FruitBox of Toy.

```
FruitBox<Toy> fruitBox = new FruitBox<Toy>();
fruitBox.add(new Toy());    // OK. We are adding a toy to a fruit box.
```

- If you want to limit the parameterized, you can do this:

```
class FruitBox<T extends Fruit> {
    ArrayList<T> list = new ArrayList<T>();
    void add(T item) { list.add(item); }
    T get(int i) { return list.get(i); }
    int size() { return list.size(); }
    public String toString() { return list.toString(); }
}
```

  - Then, only the subclasses of class Fruit can become T.

Generic Classes: Limiting Types

- T in void add(T item) must also be a subclass of class Fruit.

```
FruitBox<Fruit> fruitBox = new FruitBox<Fruit>();
fruitBox.add(new Apple());    // OK. class Apple is a subclass of Fruit.
fruitBox.add(new Grape());    // OK. class Grape is a subclass of Fruit.
```

- We can limit the type of a generic class to classes that implement a certain interface.
  - In this case, the keyword to use is extends (not implements).

```
interface Eatable {}
class FruitBox<T extends Eatable> { ... }
```

  - To limit the type to a subclass of class Fruit and also a class that implements interface Eatable:

```
class FruitBox<T extends Fruit & Eatable> { ... }
```

Generic Classes: Example 2

- Class definitions

```java
import java.util.ArrayList;

interface Eatable { }
class Fruit implements Eatable { public String toString() { return "Fruit"; } }
class Apple extends Fruit { public String toString() { return "Apple"; } }
class Grape extends Fruit { public String toString() { return "Grape"; } }
class Toy { public String toString() { return "Toy"; } }

class Box<T> {
    ArrayList<T> list = new ArrayList<T>();
    void add(T item) { list.add(item); }
    T get(int i) { return list.get(i); }
    int size() { return list.size(); }
    public String toString() { return list.toString(); }
}

class FruitBox<T extends Fruit & Eatable> extends Box<T> { }
```

Generic Classes: Example 2

- main method

```
public class Lecture {
    public static void main(String[] args) {
        FruitBox<Fruit> fruitBox = new FruitBox<Fruit>();
        FruitBox<Apple> appleBox = new FruitBox<Apple>();
        FruitBox<Grape> grapeBox = new FruitBox<Grape>();
//      FruitBox<Grape> grapeBox = new FruitBox<Apple>();    // Error: Type mismatch
//      FruitBox<Toy>   toyBox   = new FruitBox<Toy>();      // Error: Toy cannot be a type of FruitBox.

        fruitBox.add(new Fruit());
        fruitBox.add(new Apple());
        fruitBox.add(new Grape());
        appleBox.add(new Apple());
//      appleBox.add(new Grape());    // Error: Grape is not a subclass of Apple.
        grapeBox.add(new Grape());

        System.out.println("fruitBox-"+fruitBox);
        System.out.println("appleBox-"+appleBox);
        System.out.println("grapeBox-"+grapeBox);
    }
}
```

Programming Lab #14

14-01. Defining and Using a Generic Class

- Modify the following code to use a generic class Box instead of BoxA, BoxB, and BoxC.
- Additionally try the following and see what happens:
  - Write a constructor for the generic class.
  - Try creating an instance of type T inside the class definition.
  - Try creating an array of generic class.

```
class A { public String toString() { return "Class A Object"; }}
class B { public String toString() { return "Class B Object"; }}
class C { public String toString() { return "Class C Object"; }}

class BoxA {
     A item;
     void setItem(A item) { this.item = item; }
     A getItem() { return item; }
}
class BoxB {
     B item;
     void setItem(B item) { this.item = item; }
     B getItem() { return item; }
}
class BoxC {
     C item;
     void setItem(C item) { this.item = item; }
     C getItem() { return item; }
}
```

14-01. Defining and Using a Generic Class

```java
public class Ex14_01 {
     public static void main(String[] args) {
          BoxA boxa = new BoxA();
          boxa.setItem(new A());
          BoxB boxb = new BoxB();
          boxb.setItem(new B());
          BoxC boxc = new BoxC();
          boxc.setItem(new C());

          System.out.println(boxa.getItem());
          System.out.println(boxb.getItem());
          System.out.println(boxc.getItem());
     }
}
```

14-02. Limiting Parameterized Types

- Write and execute the following code. Understand why certain statements work and certain statements cause errors.

```java
import java.util.ArrayList;

interface Eatable { }
class Fruit implements Eatable { public String toString() { return "Fruit"; } }
class Apple extends Fruit { public String toString() { return "Apple"; } }
class Grape extends Fruit { public String toString() { return "Grape"; } }
class Toy { public String toString() { return "Toy"; } }

class Box<T> {
    ArrayList<T> list = new ArrayList<T>();
    void add(T item) { list.add(item); }
    T get(int i) { return list.get(i); }
    int size() { return list.size(); }
    public String toString() { return list.toString(); }
}

class FruitBox<T extends Fruit & Eatable> extends Box<T> { }
```

14-02. Limiting Parameterized Types

```
public class Ex14_02 {
    public static void main(String[] args) {
        FruitBox<Fruit> fruitBox = new FruitBox<Fruit>();
        FruitBox<Apple> appleBox = new FruitBox<Apple>();
        FruitBox<Grape> grapeBox = new FruitBox<Grape>();
//      FruitBox<Grape> grapeBox = new FruitBox<Apple>();    // Error: Type mismatch
//      FruitBox<Toy>   toyBox   = new FruitBox<Toy>();      // Error: Toy cannot be a type of FruitBox.

        fruitBox.add(new Fruit());
        fruitBox.add(new Apple());
        fruitBox.add(new Grape());
        appleBox.add(new Apple());
//      appleBox.add(new Grape());    // Error: Grape is not a subclass of Apple.
        grapeBox.add(new Grape());

        System.out.println("fruitBox-"+fruitBox);
        System.out.println("appleBox-"+appleBox);
        System.out.println("grapeBox-"+grapeBox);
    }
}
```

End of Class

Instructor office: AS818A

Email: jso1@sogang.ac.kr