Java Programming II

Generic Types and Inner Classes

Contents

- Generic Types
  - Generic Type Declarations
  - Generic Methods and Constructors
  - Bounded Type Parameters
  - Subtyping
  - Wildcards
  - Type Erasure
- Nested Types
  - Static Nested Types
  - Inner Classes
  - Extended Inner Classes
  - Anonymous Inner Classes

Generic Types

- In Java the class Object forms the root of the class hierarchy
- This is good because all objects can be cast up to the Object
- This is bad because:
  - we can potentially insert objects of the wrong type into a set or similar
  - we must cast explicitly back down to the actual class
- **Type**: Describes characteristics of data [primitive types, reference types(class or interface), annotation types]
- Generic types allow to store only a particular type of objects and they make code type safe
- Any type mismatch is detected at compile time

An Example: Generic Store

```
class Store {                // before java 1.5
    private int count;
    private Object[] arr = new Object[10];

    public Object get(final int i) {
            if (i < arr.length)
                return arr[i];
            return null;
    }
    public boolean set(final Object obj) {
            if (count < arr.length) {
                arr[count++] = obj;
                return true;
            }
            return false;
    }
}
// this Store takes any type, not only Strings
final Store store = new Store();

store.set(“a string”);
store.set(new Integer(4));
String str = (String)store.get(1); // a runtime error
```

```
class Store<T> {          // java 1.5
    private int count;
    private T[] arr = (T[])new Object[10];

    public T get(final int i) {
            if (i < arr.length)
                return arr[i];
            return null;
    }
    public boolean set(final T obj) {
            if (count < arr.length) {
                arr[count++] = obj;
                return true;
            }
            return false;
    }
}
// this Store takes only Strings
final Store<String> store = new Store<String>();

store.set(“a string”);
store.set(new Integer(4)); // a compile time error
String str = store.get(1); // cast not needed
```

Another Example: Box

```
public class Box {

    private Object object;

    public void add(Object object) {
      this.object = object;
    }

    public Object get() {
      return object;
    }
}

public class BoxDemo1 {
  public static void main(String[] args) {
    // ONLY place Integer objects into this box!
    Box integerBox = new Box();

    integerBox.add(new Integer(10));
    Integer someInteger =
   (Integer)integerBox.get();
    System.out.println(someInteger);
  }
}
```

```
public class BoxDemo2 {

 public static void main(String[] args) {

  // ONLY place Integer objects into this box!
     Box integerBox = new Box();

  // Imagine this is one part of a large application
  // modified by one programmer.

   integerBox.add("10"); // note how the type is
     now String

  //  ... and this is another, perhaps written
  //  by a different programmer
 Integer someInteger = (Integer)integerBox.get();
 System.out.println(someInteger);
 }  // end of main
}
```

If the Box class had been designed with generics in mind, this mistake would have been caught by the compiler.

Exception!

Another Example: Generic Type Box

```
**
 * Generic version of the Box class.
 */
public class Box<T> {

    private T t; // T stands for "Type"

    public void add(T t) {
        this.t = t;
    }

    public T get() {
        return t;
    }
}
```

- Generic Type Invocation

  Box<Integer> integerBox;

- Instantiation of the class

  integerBox = new Box<Integer>();
- The entire statement on one line

  Box<Integer> integerBox = new Box<Integer>();

- Type Parameter Naming Conventions
  - E - Element (used extensively by the Java Collections Framework)
  - K - Key
  - N - Number
  - T - Type
  - V - Value
  - S,U,V etc. - 2nd, 3rd, 4th types

Generic Type Declarations

- The declaration *Store<T>* is a *generic type declaration*
- *Store* is a generic class, and *T* is the *type parameter*
- *Store<String>* is a specific i.e. parameterized type and *String* is a specific *type argument*
- The use of a parameterized type is known as a *generic type invocation*
- A generic type declaration can contain multiple type parameters separated by commas (e.g. Store<A,B>, Store<A,B,C>)
- “***SingleLinkQueue***” Example of Generic Type

/home/java2/code/ByTopics/GenericType/SingleLinkQueue.java

Generic Methods

```
public class Box<T> {
    private T t; // T stands for "Type"
    public void add(T t) {
        this.t = t;
    }
 public T get() {
        return t;
    }
 public <U> void inspect(U u) {
        System.out.println("T: " +
      t.getClass().getName());
        System.out.println("U: " +
      u.getClass().getName());
    }
public static void main(String[] args) {
        Box<Integer> integerBox = new
      Box<Integer>();
        integerBox.add(new Integer(10));
        integerBox.inspect("some text");
 }
}
```

Generic Method

Result:

```
T: java.lang.Integer
U: java.lang.String
```

Generic method can allow the String type.

- It does not be restrictive to the type of the generic class. The generic method can accept any type of parameter.
- Another example

Look at the “/home/course/java2/code/GenericType/GenericMethods.java”.

Generic Methods

```
class AA<E> {
// restrictive method
 E[] method1(E[] arr) {
  for (E e: arr)
    out.println("m1: " + e.toString());
  return arr;
}

// generic method
<T> T[] method2(T[] arr) {
  for (T e: arr)
    out.println("m2: " + e.toString());
  return arr;
}
}
```

- The restrictive method (m1) just allows the type of the type parameter of the class.
- The generic method (m2) allows any type of parameter.

```
public class GenericMethods {

  public static void main(String[] args) {
    AA<Number> aa = new AA<Number>();
    Object[] a1 = aa.method1(new Integer[]
       { 1,2 });
//  Object[] a2 = aa.method1(new Object[] { 1,2 });
//  Object[] a2 = aa.method1(new String[]
       { "First","Second" });
    Object[] a3 = aa.method2(new Object[] { 1,2 });
    Object[] a4 = aa.method2(new Object[]
       { "First","Second" });

    for(Object i: a1)
      out.println("In main, a1 element: " + i);

    for(Object i: a3)
      out.println("In main, a3 element: " + i);

    for(Object i: a4)
      out.println("In main, a4 element: " + i);
  }
}
```

Error!

Bounded Type Parameters

- **When restrict the kinds of types that are allowed to be passed to a type parameter**

```
public class Box<T> {
    private T t;
    public void add(T t) {   this.t = t;     }
    public T get() {  return t;    }

    public <U extends Number> void inspect(U u){
        System.out.println("T: " + t.getClass().getName());
        System.out.println("U: " + u.getClass().getName());
    }

    public static void main(String[] args) {
        Box<Integer> integerBox = new Box<Integer>();
        integerBox.add(new Integer(10));
        integerBox.inspect("some text"); // error: this is still String!
    }
}
```

- Compilation will now fail, since our invocation of inspect still includes a String:

```
Box.java:21: <U>inspect(U) in
  Box<java.lang.Integer> cannot
be applied to (java.lang.String)
          integerBox.inspect("10");
                    ^
1 error
```

- **To specify additional interfaces that must be implemented, use the '&' character**

```
<U extends Number & MyInterface>
```

Subtyping

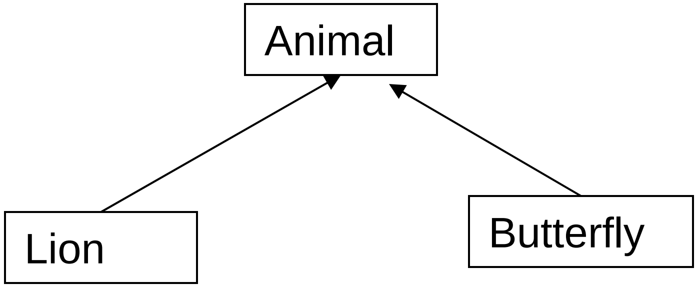
```
class Animal {    // Animal Super Class
   private String name;
   private int age;
    Animal(String n, int a) {   name = n;
      age = a; }
   public String toString() {
      return new String("Name= " + name +
      ", age= " + age);
    }
}

class Lion extends Animal {   // Lion Class
   public Lion(String n, int a)
      {   super(n,a);    }
}

class Butterfly extends Animal {     //
      Butterfly Class
   public Butterfly(String n, int a)
      {  super(n,a);    }


}
```

- Subtyping



```
Animal animal = new Animal();
Lion lion1 = new Lion();
Butterfly butterfly1 = new Butterfly();

animal = lion1;
animal = butterfly1;
```

- **However, can it be applied to the generic type either?**

Subtyping

- **It's possible to assign an object of one type to an object of another type provided that the types are compatible.**

```
public void someMethod(Number n){
    // method body omitted
}
someMethod(new Integer(10)); // OK
someMethod(new Double(10.1)); // OK
```

- **The same is also true with generics**

```
Box<Number> box = new Box<Number>();
box.add(new Integer(10)); // OK
box.add(new Double(10.1)); // OK
```

- **Now consider the following method:**

```
public void boxTest(Box<Number> n){
    // method body omitted
}
```

Is it possible to pass Box<Integer> or Box<Double> ? → NO! Why?

- **Cage interface**

```
interface Cage<E> extends Collection<E>;
```

- Let's consider the followings:

```
interface Lion extends Animal {}
Lion king = ...;

Animal a = king; // OK
```

- A lion can, of course, be put into a lion cage:

```
Cage<Lion> lionCage = ...;
lionCage.add(king); // OK
```

- and,

```
interface Butterfly extends Animal {}
Butterfly monarch = ...;
Cage<Butterfly> butterflyCage = ...;
butterflyCage.add(monarch); // OK
```

- What about an "animal cage? " :

```
Cage<Animal> animalCage = …;
animalCage.add(king); // OK
animalCage.add(monarch); // OK
```

- But, does this mean that Cage<Lion> a subtype of Cage<Animal>?" → No!
- **Neither cage can be considered an "all-animal" cage:**

```
animalCage = lionCage;        // compile-time error
animalCage = butterflyCage; // compile-time error
```

Wildcards

- **Not for any kind of animal, but rather for some kind of animal whose type is unknown. In generics, an unknown type is represented by the wildcard character "?"**

```
Cage<? extends Animal> someCage = ...;
```

- **A bounded wildcard**

  Read "? extends Animal" as "an unknown type that is a subtype of Animal, possibly Animal itself", which boils down to "some kind of animal".

```
someCage = lionCage; // OK
someCage = butterflyCage; // OK
```

- **"Can you add butterflies and lions directly to someCage?" → NO**

```
someCage.add(king);          // compiler-time error
someCage.add(monarch); // compiler-time error
```

- A way for putting some cage:

```
void feedAnimals(Cage<? extends Animal> someCage) {
    for (Animal a : someCage)
        a.feedMe();
}
```

- Invocation:

```
feedAnimals(lionCage);
feedAnimals(butterflyCage);
```

- Arrays of elements that are of a specific type produced from a generic type are not allowed.

```
Vector<String>[] v = new Vector<String>[10];
```

- But, we can define arrays of elements of a generic type where the element type is the result of an unbounded wildcard type argument.

```
Vector<?>[] vs= {new Vector<String>(), new Vector<Integer>()};  or
Vector<?>[] vs= new Vector<?>[5];
```

Type Erasure

- **When a generic type is instantiated, the compiler translates those types by a technique called type erasure — a process where the compiler removes all information related to type parameters and type arguments within a class or method.**

Box<String> -- translate → Box

Raw type

```
public class MyClass<E> {
    public static void myMethod(Object item) {
        if (item instanceof E) {  //Compiler error
            ...
        }
        E item2 = new E();   //Compiler error
        E[] iArray = new E[10]; //Compiler error
        E obj = (E)new Object(); //Unchecked cast
      warning
    }
}
```

- The operations shown in bold are meaningless at runtime because the compiler removes all information about the actual type argument (represented by the type parameter E) at compile time.

- What happens when using an older API that operates on raw types?

```
public class WarningDemo {
    public static void main(String[] args){
        Box<Integer> bi;
        bi = createBox();

/** * Pretend that this method is part of an old library,
     * written before generics. It returns
     * Box instead of Box<T>.  * **/
static Box createBox(){
        return new Box();
    }
}
```

- **What happens when using an older API that**

Recompiling with -Xlint:unchecked reveals the following additional information:

```
WarningDemo.java:4: warning: [unchecked]
      unchecked conversion
found   : Box
required: Box<java.lang.Integer>
        bi = createBox();
                     ^

1 warning
```

Nested Classes

- A class can be defined inside another class
- Benefits:
  - to structure and scope members
  - to connect logically related objects
- A nested class is considered a part of its enclosing class
- They share a trust relationship, i.e. everything is mutually accessible
- Nested types could be:
  - static – allows simple structuring of types
  - nonstatic – defines a special relationship between a nested object and an object of the enclosing class

Inner Class

- Class in the Class
  - Provide the method to define the object type to use in the class
  - Solve the class name conflict to restrict the reference scope of class
  - Information hiding

```
class OuterClass {
          // ...
          class InnerClass {
                    // ...
          }
}
```

Inner Class

- Name Reference
  - OuterClass inside : Use InnerClass Simple name
  - OuterClass outside : OuterClass.InnerClass

```
public static void main(String[] args) {
        OuterClass  outObj = new  OuterClass();
       OuterClass.InnerClass   inObj = outObj.new  InnerClass();
}
```

- Access Modifier
  - public, private, protected

] Inner class cannot have static variable

Static Inner Class

- Able to use static variable

```
class OuterClass {
        // ...
        static class InnerClass {
                static  int staticVariable;
                // ...
         }
}
```

OuterClass.InnerClass

- Can refer without creating object
    - [StaticInnerClass.java]

Static Inner Class

- Can refer without creating object
- Can use static variable

- Result:

```
... without creating Outer-class object
call static method
call static method
```

```
class OuterClass {
    static class InnerClass  {
        static String str;
        InnerClass(String s) {
            str = s;
        }
        void print() {
        staticPrint(str);
        }
        static void staticPrint(String s) {
            str = s;
            System.out.println(s);
        }
    } // end of InnerClass
} // end of OuterClass

public class StaticInnerClass {
    public static void main(String[] args) {
        String s = "... without creating Outer-class object";
        OuterClass.InnerClass p = new
        OuterClass.InnerClass(s);
        p.print();
        OuterClass.InnerClass.staticPrint("call static
        method");
        p.print();
    }
}
```

Inner Classes

- Inner classes are associated with instances of a class
- Inner classes cannot have static members including static nested types
- Inner classes can have final static fields that are constant
- Inner classes can extend any other class, implement any interface, or be extended

- When an inner class object is created inside a method of the enclosing class the current object this is automatically associated with it
- The name of the reference to the enclosing object is this preceded by the enclosing class name – a form known as **qualified–this**

```
public class BankAccount {
      private long number;
      private static long bankID;

      public Permissions permissionsFor(Person who) {
                Permissions p = new Permissions();
                // equal to        this.new Permissions();

                p.method();
                return p;
      }

      public class Permissions {
                public boolean canDeposit;
                public boolean canWithdraw;

                void method() {
                    long bid = bankID;      ]  in scope
                    long num = number;      ]

                   num = BankAccount.this.number; // qualified-this
                }

      }

      public static void main(String[] args) {
                BankAccount ba = new BankAccount();
                Permissions p = ba.new Permissons();
      }
}
```

Extended Inner Classes

- An inner class can be extended
- Objects of the extended class must remain associated with objects of the original enclosing class or a subclass

```
class Outer {
      class Inner {
      }
}

class ExtendOuter extends Outer {
      class ExtendInner extends Inner {

      }

      Inner ref = new ExtendInner();
}

class Unrelated extends Outer.Inner {
      Unrelated(Outer ref) {
            ref.super();
      }
}
```

Local Inner Classes

- Can be defined in code blocks and they are local and accessible to that block
- They do not have access modifiers (e.g. private, public) and cannot be static
- They are not members of the class to which the block belongs
- A local inner class can access all the variables in that block, and static variables, but a local variable or method parameter must be declared as final

```
import static java.lang.System.*;

public class EnclosingClass {
  int x = 55;

  public void enclosingMethod(final int a) {
      int z = 44;
      final int q = 32;

      class LocalInner {
          private int x = 17;

          public void innerMethod() {
             out.println(x);        // → 17
             out.println(q);        // → 32
             out.println(a);        // → 99
             out.println(this.x); // → 17
             out.println(EnclosingClass.this.x); // → 55
          }
      }  // end of LocalInner

      LocalInner li = new LocalInner();
      li.innerMethod();
  }

  public static void main(String[] args) {
       new EnclosingClass().enclosingMethod(99);
  }

}
```

Anonymous Inner Classes

- Extend a class or implement an interface
- They are defined at the same time when instantiated with new, as part of a statement
- They cannot have explicit constructors because they have no name

```
import static java.lang.System.*;

public class EnclosingClass {
    int x = 55;

    public void anotherMethod2() {
      out.println(new Object() {
           private int x = 17;

           public String toString() {
             return "Inner x: " + x +
" Enclosing x: " + EnclosingClass.this.x;
           }
      } );
    }

}
```

```
→ Inner x: 17 Enclosing x: 55
```