Generics

Agenda

1 **Generics**

Generics

What are and Why Generics?

Mechanism by which a single piece of code can manipulate many different data types without explicitly having a separate entity for each data type

- **Before Generics**

```
List myIntegerList = new LinkedList(); // 1
myIntegerList.add(new Integer(0)); // 2
Integer x = (Integer) myIntegerList.iterator().next(); // 3
```

Line no 3 if not properly typecasted will throw runtime exception

- **After Generics**

```
List<Integer> myIntegerList = new LinkedList<Integer>(); // 1
myIntegerList.add(new Integer(0)); //2
Integer x = myIntegerList.iterator().next(); // 3
```

No need for typecasting here

What problems does Generics solve?

- **Problem: Collection element types**
  - Compiler is unable to verify types
  - Assignment must have type casting
  - *ClassCastException* can occur during runtime
- **Solution: Generics**
  - Tell the compiler type of the collection
  - Let the compiler fill in the cast
  - Example: Compiler will check if you are adding Integer type entry to a String type collection (compile time detection of type mismatch)

Using Generic class

- **Instantiate a generic class to create type specific object**
- **In J2SE 5.0, all collection classes are rewritten to be generic classes**

```
Vector<String> vs = new Vector<String>();
vs.add(new Integer(5)); // Compile error!
vs.add(new String("hello"));
String s = vs.get(0); // No casting needed
```

Using Generic class (Contd.).

- **Generic class can have multiple type parameters**
- **Type argument can be a custom type**

```
HashMap<String, Mammal> map = new HashMap<String, Mammal>();
map.put(“monkey”, new Mammal("monkey"));
Mammal w = map.get(“monkey”);
```

Raw Type

- **Generic type instantiated with no type arguments**
- **Pre-J2SE 5.0 classes continue to function over J2SE 5.0 JVM as raw type**

```
// Generic type instantiated with type argument
List<String> ls = new LinkedList<String>();
// Generic type instantiated with no type
// argument – Raw type
List lraw = new LinkedList();
```

Type Erasure

- Enables Java applications that uses generics to maintain binary compatibility with Java libraries and applications that were created before generics
- So generic type information does not exist during runtime
  - During runtime, the class that represents ArrayList<String>, ArrayList<Integer> is the same class that represents ArrayList

Type Erasure Example code: True / False

```
ArrayList<Integer> ai = new ArrayList<Integer>();
ArrayList<String> as = new ArrayList<String>();
Boolean b1 = (ai.getClass() == as.getClass());
System.out.println(“ Do ArrayList<Integer> and ArrayList<String>
share same class ? ” + b1);
```

O/P:

Do ArrayList<Integer> and ArrayList<String> share same class? true

Interoperability with pre J2SE 5 code

- **For raw type, compiler does not have enough type information (for type checking) so it just generates “unchecked” or “unsafe” warning**
- **If you ignore them, ClassCastException can still occur during runtime**

```
List<String> ls = new LinkedList<String>();

List lraw = ls;

lraw.add(new Integer(4)); // Compiler Warning

String s = ls.iterator().next(); // Runtime error
```

Recommendations

- Do not use raw type whenever possible
- If you have to use raw type or have to use pre-J2SE 5.0 compiled classes or libraries, make sure the "*unsafe*" compile warnings are really just warnings

AutoBoxing with Collections

- **Boxing conversion converts primitive values to objects of corresponding wrapper types**

// prior to Java 5, Explicit Boxing

```
int i = 11;
Integer iReference = new Integer(i);
// In Java 5, Automatic Boxing
iReference = i;
```

- **Unboxing conversion converts objects of wrapper types to values of corresponding primitive types**

// prior to Java 5,Explicit Unboxing

```
int j = iReference.intValue();
j = iReference; // In Java 5, Automatic Unboxing
```

AutoBoxing with Collections

- J2SE 5 adds to the java language autoboxing and auto-unboxing.
- Primitive types and their corresponding wrapper classes can now be used interchangeably. For example the following lines of code are legitimate in Java 5 : int a = 0;
- Integer b = a; //
- int c = new Integer(b);
- This is often referred to as automatic ***boxing*** or ***unboxing***.

AutoBoxing with Collections

- If an *int* is passed where an Integer is expected, the compiler will automatically insert a call to the Integer constructor. Conversely, if an Integer is provided where an *int* is required, there will be an automatic call to the IntValue method.
- Autoboxing is the process by which a primitive type is automatically encapsulated into its equivalent type wrapper whenever an object of that type is needed.
- Auto-unboxing is the process by which the value of a boxed object is automatically extracted (unboxed) from type wrapper when its value is needed.

Quiz

1. Which of the following are true regarding Generics?

   a. It lets you enforce compile-time safety on collections

   b. When using collections, a cast is needed to get elements out of the Collection

   c. Both option a and option b are true

   d. None of the above are true

2. Will the following code compile?

   List<Animal> aList = new ArrayList<Animal>();

   a. True

   b. False

Summary

- Generics

Thank You