Principles of Computer Science

Java Generics

Topics

- Benefits of generics
- Using generic classes and interfaces
- Declaring generic classes and interfaces
- To understand why generic types can improve reliability and readability
- To declare and use generic methods and bounded generic types
- To know certain restrictions on generic types caused by type erasure.
- Packages in Java

Why Do You Get a Warning?

```
public class ShowUncheckedWarning {
  public static void main(String[] args) {
    java.util.ArrayList list =
      new java.util.ArrayList();
    list.add("Java Programming");
  }
}
```

Why Do You Get a Warning?

```
public class ShowUncheckedWarning {
  public static void main(String[] args) {
    java.util.ArrayList list =
      new java.util.ArrayList();
    list.add("Java Programming");
  }
}
```

compile warning

To understand the compile warning on this line, you need to learn JDK 1.5 generics.

Fix the Warning

```
public class ShowUncheckedWarning {
  public static void main(String[] args) {
    java.util.ArrayList<String> list =
      new java.util.ArrayList<String>();
    list.add("Java Programming");
  }
}
```

No compile warning on this line.

```
public class ShowUncheckedWarning {
  public static void main(String[] args)
  {
    java.util.ArrayList list =
      new java.util.ArrayList();

    list.add("Java Programming");
  }
}
```

```
public class ShowUncheckedWarning {
  public static void main(String[] args)
  {
    java.util.ArrayList<String> list =
      new java.util.ArrayList<String>();
    list.add("Java Programming");
  }
}
```

What is Generics?

- *Generics* is the capability to parameterize types.
- For example, you may define a generic stack class that stores the elements of a generic type.
- From this generic class, you may create a stack object for holding strings and a stack object for holding numbers.

Why Generics?

- The key benefit of generics is to enable errors to be detected at compile time rather than at runtime.
- A generic class or method permits you to specify allowable types of objects that the class or method may work with. If you attempt to use the class or method with an incompatible object, the compile error occurs.

Generic Classes

- Lets consider a very simple data structure: Pair
  - This data structure stores pairs of things, e.g.
    - (“Toronto”, “Ontario”)
    - (“Ken”, “Pu”)
    - (3, 4)
    - (“Toronto”, 2.5)
    - (“Toronto”, (43, 79))
- Consider different operations:
  - **first():** Get the first element
  - **second():** Get the second element
  - **swap():** Swap the first and second elements

Generic Classes

- How do we implement something like **Pair** in Java so we can store all sorts of data into it?
- Say we want to store pairs of ints and pairs of strings, we need to implement the Pair class twice

```
class Pair {
  protected int x;
  protected int y;
  public int first() { ... };
  public int second() { ... };
  public Pair swap() { ... };
}
```

Pair for ints

```
class Pair {
  protected String x;
  protected String y;
  public String first() { … };
  public String second() { … };
  public Pair swap() { ... };
}
```

Pair for strings

These classes are restricted to the specific data types. Generic classes removes this restriction.

Generic Classes: Syntax

- To make a class generic:

```
class MyClass<T> { ... }
```

- This makes MyClass generic to whatever class specified by the type variable T.
- There can be many type variables:
  - `class MyClass<S, T, ...>`
  - The convention is to use upper letters – just a convention, not a rule.

Example: Pair

```
class Pair<U, V> {
  protected U x;
  protected V y;
  public Pair(U x, V y) {
    this.x = x;
    this.y = y;
  }
  public U first() {
    return x;
  }
  public V second() {
    return y;
  }
  public Pair<V, U> swap() {
    return new Pair<V,U>(y, x);
  }
  public void debug() {
    System.out.println("<" + x.getClass().getName() + "> x = " + x);
    System.out.println("<" + y.getClass().getName() + "> y = " + y);
  }
}
```

Generic Classes: Usage

- To declare a variable of a generic class, you must specify the type variables in terms of classes (or existing type variables) after the class name.

```
MyClass<String> x1;
```

- To invoke a generic method, you must specify the type variables before the method name. This is rare. Do not confuse this with invoking regular methods of a generic class.

Example: Pair

```
class Pair<U, V> {
  protected U x;
  protected V y;
  public Pair(U x, V y) {
    this.x = x;
    this.y = y;
  }
  public U first() {
    return x;
  }
  public V second() {
    return y;
  }
  public Pair<V, U> swap() {
    return new Pair<V,U>(y, x);
  }
  public void debug() {
    System.out.println("<" + x.getClass().getName() + "> x = " + x);
    System.out.println("<" + y.getClass().getName() + "> y = " + y);
  }
}
```

Usage

```
Pair<String, Float> p1 =
    new Pair<String, Float>("Toronto", 2.5f);
p1.debug();

Pair<Float, String> p2 = p1.swap();
p2.debug();

Pair<String, Pair<Integer, Integer>> city =
    new Pair<String, Pair<Integer, Integer>> (
      "Toronto",
      new Pair<Integer, Integer>(41, 79)
    );
```

Generic Methods: syntax

- To make a method generic:

```
public <U> void doSomething( … )
```

- This is rarely used because all methods of a generic class is already generic.

Generic Methods

```
public static <E> void print(E[] list) {
   for (int i = 0; i < list.length; i++)
     System.out.print(list[i] + " ");
   System.out.println();
 }
```

Generic Version

```
public static void print(Object[] list) {
   for (int i = 0; i < list.length; i++)
     System.out.print(list[i] + " ");
   System.out.println();
 }
```

Generic Type: Improves Reliability

```
package java.lang;

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

(a) Prior to JDK 1.5

```
package java.lang;

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

(b) JDK 1.5

Runtime error

```
Comparable c = new Date();
System.out.println(c.compareTo("red"));
```

(a) Prior to JDK 1.5

Generic Instantiation

```
Comparable<Date> c = new Date();
System.out.println(c.compareTo("red"));
```

(b) JDK 1.5

Compile error

No Casting Needed

```
// Declare a list to be ArrayList<Double>
ArrayList<Double> list = new ArrayList<Double>();

// 5.5 is automatically converted to new Double(5.5)
list.add(5.5);

// 3.0 is automatically converted to new Double(3.0)
list.add(3.0);

// No casting is needed
Double doubleObject = list.get(0);

// Automatically converted to double
double d = list.get(1);
```

No Casting Needed

```
// Declare a list to be ArrayList<Double>
ArrayList<Double> list = new ArrayList<Double>();

// 5.5 is automatically converted to new Double(5.5)
list.add(5.5);

// 3.0 is automatically converted to new Double(3.0)
list.add(3.0);

// No casting is needed
Double doubleObject = list.get(0);

// Automatically converted to double
double d = list.get(1);
```

Not using generics

Needs casting

```
// Old style ArrayList, not using Generics
ArrayList list = new ArrayList();

// Have to explicitly create an object of type Double
list.add((Object) new Double(5.5));

// Have to use casting, since list.get(0) returns an Object
Double doubleObject = (Double) list.get(0);
```

Raw Type and Backward Compatibility

```
// Raw type
ArrayList list = new ArrayList();
```

This is roughly equivalent to

```
ArrayList<Object> list = new ArrayList<Object>();
```

Raw Type is Unsafe

```
// Max.java: Find a maximum object
public class Max {
  /** Return the maximum between two objects */
  public static Comparable max(Comparable o1, Comparable o2) {
    if (o1.compareTo(o2) > 0)
      return o1;
    else
      return o2;
  }
}
```

Runtime Error:

```
Max.max("Welcome", 23);
```

Make it Safe

```
// Max1.java: Find a maximum object
public class Max1 {
  /** Return the maximum between two objects */
  public static <E extends Comparable<E>> E max(E o1, E o2) {
    if (o1.compareTo(o2) > 0)
      return o1;
    else
      return o2;
  }
}
```

Compile Time Error: (much better than runtime errors)

```
Max.max("Welcome", 23);
```

Avoiding Unsafe Raw Types

- Use

```
new ArrayList<ConcreteType>()
```

- Instead of

```
new ArrayList();
```

Generic ArrayList in JDK 1.5

java.util.ArrayList
+ArrayList()
+add(o: Object) : void
+add(index: int, o: Object) : void
+clear(): void
+contains(o: Object): boolean
+get(index: int) : Object
+indexOf(o: Object) : int
+isEmpty(): boolean
+lastIndexOf(o: Object) : int
+remove(o: Object): boolean
+size(): int
+remove(index: int) : boolean
+set(index: int, o: Object) : Object

(a) ArrayList before JDK 1.5

java.util.ArrayList<E>
+ArrayList()
+add(o: E) : void
+add(index: int, o: E) : void
+clear(): void
+contains(o: Object): boolean
+get(index: int) : E
+indexOf(o: Object) : int
+isEmpty(): boolean
+lastIndexOf(o: Object) : int
+remove(o: Object): boolean
+size(): int
+remove(index: int) : boolean
+set(index: int, o: E) : E

(b) ArrayList in JDK 1.5

Compile Time Checking

- For example, the compiler checks whether generics is used correctly for the following code in (a) and translates it into the equivalent code in (b) for runtime use. The code in (b) uses the raw type.

```
ArrayList<String> list = new ArrayList<String>();
list.add("Oklahoma");
String state = list.get(0);
```

(a)

```
ArrayList list = new ArrayList();
list.add("Oklahoma");
String state = (String)(list.get(0));
```

(b)

Generic Classes: Features and Limitations

- Type variables can be constraint using

```
<T extends AreaShape>
```

  - AreaShape can be an interface, abstract class or a class.
- We can also do

```
<T extends AreaShape & ShapeShifter>
```

- Wildecard, upperbound
  - Beyond the scope of this course – refer to online tutorial.
- **Limitation: (type erasure)**
  - Cannot create an array involving type variables. The following is not allowed

```
T[] array = new T[100];
```

Generic Classes: Solution to Type Erasure

- A different way of creating an array in Java – using the Java **reflection** API. Details are beyond the scope of this course.

```
import java.lang.reflect.Array;
class MyClass {
  public static <T> T[] makeArray(T sample, int length) {
    Class javaClass = sample.getClass() // get the class reference of x
    return (T[])Array.newInstance(javaClass, length);
  }
}
```

```
String x = "Hello";
String[] array = MyClass.<String>makeArray(x, 100);
```

Erasure and Restrictions

- Generics are implemented using an approach called *type erasure*.
- The compiler uses the generic type information to compile the code, but erases it afterwards.
- So the generic information is not available at run time. This approach enables the generic code to be backward-compatible with the legacy code that uses raw types.

Important Facts

- It is important to note that a generic class is shared by all its instances regardless of its actual generic type:

```
GenericStack<String> stack1 = new GenericStack<String>();
GenericStack<Integer> stack2 = new GenericStack<Integer>();
```

- Although GenericStack<String> and GenericStack<Integer> are two types, but there is only **one** class GenericStack loaded into the JVM.

Restrictions on Generics

- Restriction 1
  - Cannot create an instance of a generic type. (i.e., new E()).
- Restriction 2
  - Generic array creation is not allowed. (i.e., new E[100]).
- Restriction 3
  - A generic type parameter of a class is not allowed in a atatic context.
- Restriction 4
  - Exception classes cannot be generic.

Java Packages

Packages

- When there are too many classes, the class names may conflict with one another.
  - E.g. UOIT students are modeled by **Student** class, but Durham college students may be modeled by a different class, but also named **Student**.
- Solution: group classes into packages.
  - E.g. we may have a package called **uoit** and another called **durham.college**. Then the two student classes are:
    - uoit.Student
    - durham.college.Student

Packages in Java

- Packages are not created explicitly (unlike classes).
- Each class can declare which package they belong to.

```
package uoit;

public class Student {
   ...
}
```

```
package durham.college;

public class Student {
   ...
}
```

Packages in Java

- Packages are not created explicitly (unlike classes).
- Each class can declare which package they belong to.

```
package uoit;

public class Student {
   ...
}
```

```
package durham.college;

public class Student {
   ...
}
```

Packages in Java

- Packages are not created explicitly (unlike classes).
- Each class can declare which package they belong to.

```
package uoit;

public class Student {
   ...
}
```

```
package durham.college;

public class Student {
   ...
}
```

These are two different Student classes

Referring to classes by their fully qualified names

- A class is accessible outside its own package only if it is declared as a public class. By default, classes are not public.
- A fully qualified class name is of the form:

```
package-name.class-name
```

- To avoid using fully qualified names to refer to classes, use **import**.
- A class can be referred by its own name under two conditions:
  - It is a class in the current package.
  - Its package has been imported using the **import** keyword.

Summary

- Generics
  - Benefits, usage, restrictions
- Packages in Java

Readings

- Text
  - Chapter 14
  - Chapter 3 (Sec. 3.3)
  - Chapter 11 (Section 11.2)
- Online
  - Generics: http://docs.oracle.com/javase/tutorial/java/generics/index.html