Generics and Collections

Oracle® Certified Professional, Java® SE 7 Programmer

Module 4

Module 4 - Objectives

- Create a generic class
- Use the diamond syntax to create a collection
- Analyze the interoperability of collections that use raw type and generic types
- Use wrapper classes and auto-boxing
- Create and use a List, a Set and a Deque.
- Create and use a Map
- Use java.util.Comparator and java.lang.Comparable.
- Sort and search arrays and lists

Generic Class

- Generics were introduced in Java 5.
- Generics were added to allow compile time checking of Java types. This allows the javac compiler to find errors during compilation, and helps to avoid exceptions at runtime.
- Generics remove the requirement to cast objects.
- A generic class requires a Type. Typically you will see arguments like <T> (type), <K> (key), <V> (value), <E> (element).

Generic Class (cont.)

```
public interface Producer<T> {
   T produce();
}
public class WidgetProducer implements Producer<Widget> {
   @Override
   public Widget produce() {
      return new Widget();
   }
```

Raw and Generic Type Collections

- A "raw" collection is a name for the version of collections prior to the introduction of generics in Java 5.
- A "raw" collection is not type-safe, and requires casting when an element is extracted from the collection.
- A "raw" collection contains Object(s).
- A typed collection declares a type which the collection contains, e.g. ArrayList<Widget> is a list of Widget objects.
- No casting required when elements are are extracted.

Raw and Generic Type Collections (cont.)

- A typed collection can also contain subclasses of a typed object using a syntax like, e.g. ArrayList<? extends Widget>();.
- A wildcard (?) can be used to declare a collection, e.g. Collection<?> c = new ArrayList<>();

Diamond Operator <>

- Introduced in Java 7 as a part of Project Coin.
- Also called "Type Inference" because the type is inferred from the LHS (Left-Hand Side) of the declaration.
- Simplifies the declaration of typed collections.

```
List<List<Map<String,String>>> list =  new
ArrayList<List<Map<String,String>>>(); //pre-Java 7

List<List<Map<String,String>>> list  = new ArrayList<>();//Java 7+
```

Wrapper Classes (primitive wrappers)

- Wrapper classes are used to represent primitive values when an Object is required.
- Wrapper classes are used extensively to wrap primitive classes for use in Collections since a Collection can only store objects. It is also used with a number of reflection API classes.
- The 8 primitive wrapper classes are: Byte (byte), Short (short), Character (char), Integer (int), Long (long), Float (float), Double (double), and Boolean (boolean).

Auto-Boxing and Auto-Unboxing

- Java 5 SE introduced auto-boxing. This is the automatic conversion of a primitive to an Object wrapper if required.
- Auto-unboxing is the automatic conversion of a primitive wrapper class to a primitive.
- An Exception can occur when auto-unboxing if the primitive wrapper is null.

List<T>, Set<T>, and Deque<T>

- List – An ordered collection also known as a sequence. Values are stored in the order in which they are added unless manipulated.
- Set – A collection which contains no duplicate entries, and at most one null value. Objects being added to a set should implement equals, and hashcode methods otherwise it is possible to add multiple objects to a Set which may be intrinsically equivalent.
- Deque – A linear collection which supports adding and removing elements from either end ("double-ended queue").

Map<K,V>

- A Map is not part of the Collections API!
- A map stores values by key.
- The keys must be unique, but the same value can be assigned to multiple keys.
- **Warning:** do not use mutable objects for keys!

Comparable<T>

- The Comparable<T> interface has one method: int compareTo(T o); which is used determine order.
- "Returns a negative integer, zero, or a positive integer as this object is less than, equal to, or greater than the specified object."

Comparator<T>

- The Comparator<T> interface has two methods: int compare(T o1, T o2); and boolean equals(Object o); which are used to determine order, and if two comparators are the same.
- "Compares its two arguments for order. Returns a negative integer, zero, or a positive integer as the first argument is less than, equal to, or greater than the second."
- The equals method DOES not determine equality between compared objects. It only determines if the Object provided to the method is equal to the current Comparator. This includes ordering.

Sorting and Searching Arrays and Lists

- The Collections class is YOUR friend.
- The Collections class has a significant number of methods to make using the Java Collections Framework easier to use.
- Sorting can be done via "natural" sorting using the Collections.sort(Collection<T>) method.
- Sorting can also be accomplished by providing a Comparator<T> to the sort method.

Sorting and Searching Arrays and Lists (cont.)

- The Arrays class is YOUR friend.
- The Arrays class has a number of convenience methods to sort, copy, compare (equals), and search an array.
- The Arrays class has a convenience method to convert an array to a List object.

creative commons

Attribution-NonCommercial-ShareAlike 3.0 Unported