

Generic Types and the Java Collections Framework

Lecture 14
CS2110 – Fall 2008



Generic Types in Java 5

- When using a collection (e.g., `LinkedList`, `HashSet`, `HashMap`), we generally have a single type `T` of elements that we store in it (e.g., `Integer`, `String`)
- Before Java 5, when extracting an element, had to cast it to `T` before we could invoke `T`'s methods
- Compiler could not check that the cast was correct at `compile-time`, since it didn't know what `T` was
- Inconvenient and unsafe, could fail at `runtime`
- Generics in Java 5 provide a way to communicate `T`, the type of elements in a collection, to the compiler
 - Compiler can check that you have used the collection consistently
 - Result: safer and more-efficient code

Example

old

```
//removes 4-letter words from c
//elements must be Strings
static void purge(Collection c) {
    Iterator i = c.iterator();
    while (i.hasNext()) {
        if (((String)i.next()).length() == 4)
            i.remove();
    }
}
```

new

```
//removes 4-letter words from c
static void purge(Collection<String> c) {
    Iterator<String> i = c.iterator();
    while (i.hasNext()) {
        if (i.next().length() == 4)
            i.remove();
    }
}
```

Another Example

old

```
Map grades = new HashMap();  
grades.put("John", new Integer(67));  
grades.put("Jane", new Integer(88));  
grades.put("Fred", new Integer(72));  
Integer x = (Integer)grades.get("John");  
sum = sum + x.intValue();
```

new

```
Map<String, Integer> grades = new HashMap<String, Integer>();  
grades.put("John", new Integer(67));  
grades.put("Jane", new Integer(88));  
grades.put("Fred", new Integer(72));  
Integer x = grades.get("John");  
sum = sum + x.intValue();
```

Type Casting

- In effect, Java inserts the correct cast automatically, based on the declared type
- In this example, `grades.get("John")` is automatically cast to `Integer`

```
Map<String, Integer> grades = new HashMap<String, Integer>();  
grades.put("John", new Integer(67));  
grades.put("Jane", new Integer(88));  
grades.put("Fred", new Integer(72));  
Integer x = grades.get("John");  
sum = sum + x.intValue();
```

An Aside: Autoboxing

- Java 5 also has autoboxing and auto-unboxing of primitive types, so the example can be further simplified

```
Map<String,Integer> grades = new HashMap<String,Integer>();  
grades.put("John",new Integer(67));  
grades.put("Jane",new Integer(88));  
grades.put("Fred",new Integer(72));  
Integer x = grades.get("John");  
sum = sum + x.intValue());
```

```
Map<String,Integer> grades = new HashMap<String,Integer>();  
grades.put("John", 67);  
grades.put("Jane", 88);  
grades.put("Fred", 72);  
sum = sum + grades.get("John");
```

Using Generic Types

- `<T>` is read, “of T”
 - For example: `Stack<Integer>` is read, “Stack of Integer”
- The type annotation `<T>` informs the compiler that all extractions from this collection should be automatically cast to T
- Specify type in declaration, can be checked at compile time
 - Can eliminate explicit casts

Advantage of Generics

- Declaring `Collection<String> c` tells us something about the variable `c` (i.e., `c` holds only Strings)
 - This is true wherever `c` is used
 - The compiler checks this and won't compile code that violates this
- Without use of generic types, explicit casting must be used
 - A cast tells us something the programmer **thinks** is true at a single point in the code
 - The Java virtual machine **checks** whether the programmer is right only at runtime

Subtypes

`Stack<Integer>` is *not* a subtype of `Stack<Object>`

```
Stack<Integer> s = new Stack<Integer>();  
s.push(new Integer(7));  
Stack<Object> t = s; // Gives compiler error  
t.push("bad idea");  
System.out.println(s.pop().intValue());
```

However, `Stack<Integer>` *is* a subtype of `Stack` (for backward compatibility with previous Java versions)

```
Stack<Integer> s = new Stack<Integer>();  
s.push(new Integer(7));  
Stack t = s; // Compiler allows this  
t.push("bad idea"); // Produces a warning  
System.out.println(s.pop().intValue()); //Runtime error!
```

Programming with Generic Types

```
public interface List<E> { // E is a type variable
    void add(E x);
    Iterator<E> iterator();
}

public interface Iterator<E> {
    E next();
    boolean hasNext();
    void remove();
}
```

- To use the interface `List<E>`, supply an actual type argument, e.g., `List<Integer>`
- All occurrences of the formal type parameter (`E` in this case) are replaced by the actual type argument (`Integer` in this case)

Wildcards

old

```
void printCollection(Collection c) {  
    Iterator i = c.iterator();  
    while (i.hasNext()) {  
        System.out.println(i.next());  
    }  
}
```

bad

```
void printCollection(Collection<Object> c) {  
    for (Object e : c) {  
        System.out.println(e);  
    }  
}
```

good

```
void printCollection(Collection<?> c) {  
    for (Object e : c) {  
        System.out.println(e);  
    }  
}
```

Bounded Wildcards

```
static void sort (List<? extends Comparable> c) {  
    ...  
}
```

- Note that if we declared the parameter `c` to be of type `List<Comparable>` then we could not sort an object of type `List<String>` (even though `String` is a subtype of `Comparable`)
 - Suppose Java treated `List<String>` and `List<Integer>` as a subtype of `List<Comparable>`
 - Then, for instance, a method passed an object of type `List<Comparable>` would be able to store `Integers` in our `List<String>`
- Wildcards let us specify exactly what types are allowed

Generic Methods

- Adding all elements of an array to a **Collection**

bad

```
static void a2c(Object[] a, Collection<?> c) {  
    for (Object o : a) {  
        c.add(o); // compile time error  
    }  
}
```

good

```
static <T> void a2c(T[] a, Collection<T> c) {  
    for (T o : a) {  
        c.add(o); // ok  
    }  
}
```

- See the online Java Tutorial for more information on generic types and generic methods

Generic Classes

```
public class Queue<T> extends AbstractBag<T> {  
  
    private java.util.LinkedList<T> queue  
        = new java.util.LinkedList<T>();  
  
    public void insert(T item) {  
        queue.add(item);  
    }  
  
    public T extract() throws java.util.NoSuchElementException {  
        return queue.remove();  
    }  
  
    public void clear() {  
        queue.clear();  
    }  
  
    public int size() {  
        return queue.size();  
    }  
}
```

Generic Classes

```
public class InsertionSort<T extends Comparable<T>> {  
  
    public void sort(T[] x) {  
  
        for (int i = 1; i < x.length; i++) {  
            // invariant is: x[0],...,x[i-1] are sorted  
            // now find rightful position for x[i]  
            T tmp = x[i];  
            int j;  
            for (j = i; j > 0 && x[j-1].compareTo(tmp) > 0; j--)  
                x[j] = x[j-1];  
            x[j] = tmp;  
        }  
    }  
}
```

Java Collections Framework

- **Collections**: holders that let you store and organize objects in useful ways for efficient access
- Since Java 1.2, the package `java.util` includes interfaces and classes for a general collection framework
- Goal: conciseness
 - A few concepts that are broadly useful
 - Not an exhaustive set of useful concepts
- The collections framework provides
 - Interfaces (i.e., ADTs)
 - Implementations

JCF Interfaces and Classes

- Interfaces

- Collection
- Set (no duplicates)
- SortedSet
- List (duplicates OK)

- Map (i.e., Dictionary)
- SortedMap

- Iterator
- Iterable
- ListIterator

- Classes

- HashSet
- TreeSet
- ArrayList
- LinkedList

- HashMap
- TreeMap

java.util.Collection<E> (an interface)

```
public int size();
```

- Return number of elements in collection

```
public boolean isEmpty();
```

- Return true iff collection holds no elements

```
public boolean add(E x);
```

- Make sure the collection includes x; returns true if collection has changed (some collections allow duplicates, some don't)

```
public boolean contains(Object x);
```

- Returns true iff collection contains x (uses equals() method)

```
public boolean remove(Object x);
```

- Removes a single instance of x from the collection; returns true if collection has changed

```
public Iterator<E> iterator();
```

- Returns an Iterator that steps through elements of collection

`java.util.Iterator<E>` (an interface)

```
public boolean hasNext();
```

- Returns true if the iteration has more elements

```
public E next();
```

- Returns the next element in the iteration
- Throws `NoSuchElementException` if no next element

```
public void remove();
```

- The element most recently returned by `next()` is removed from the underlying collection
- Throws `IllegalStateException` if `next()` not yet called or if `remove()` already called since last `next()`
- Throws `UnsupportedOperationException` if `remove()` not supported

Additional Methods of Collection<E>

```
public Object[] toArray()
```

- Returns a new array containing all the elements of this collection

```
public <T> T[] toArray(T[] dest)
```

- Returns an array containing all the elements of this collection; uses dest as that array if it can

Bulk Operations:

- `public boolean containsAll(Collection<?> c);`
- `public boolean addAll(Collection<? extends E> c);`
- `public boolean removeAll(Collection<?> c);`
- `public boolean retainAll(Collection<?> c);`
- `public void clear();`

java.util.Set<E> (an interface)

- **Set** extends **Collection**
 - **Set** inherits all its methods from **Collection**
- A **Set** contains no duplicates
 - If you attempt to **add()** an element twice then the second **add()** will return false (i.e., the **Set** has not changed)
- Write a method that checks if a given word is within a **Set** of words
- Write a method that removes all words longer than 5 letters from a **Set**
- Write methods for the union and intersection of two **Sets**

Set Implementations

- `java.util.HashSet<E>` (a hashtable)

- Constructors

```
public HashSet();  
public HashSet(Collection<? extends E> c);  
public HashSet(int initialCapacity);  
public HashSet(int initialCapacity, float loadFactor);
```

- `java.util.TreeSet<E>`

(a balanced BST [red-black tree])

- Constructors

```
public TreeSet();  
public TreeSet(Collection<? extends E> c);  
...
```

`java.util.SortedSet<E>` (an interface)

- `SortedSet` *extends* `Set`
- For a `SortedSet`, the `iterator()` returns the elements in sorted order
- Methods (in addition to those inherited from `Set`):
 - `public E first();`
 - ◆ Returns the first (lowest) object in this set
 - `public E last();`
 - ◆ Returns the last (highest) object in this set
 - `public Comparator<? super E> comparator();`
 - ◆ Returns the `Comparator` being used by this sorted set if there is one; returns null if the natural order is being used
 - ...

java.lang.Comparable<T> (an interface)

- `public int compareTo(T x) ;`
 - Returns a value (< 0), (= 0), or (> 0)
 - ♦ (< 0) implies `this` is before `x`
 - ♦ (= 0) implies `this.equals(x)` is true
 - ♦ (> 0) implies `this` is after `x`
- Many classes implement `Comparable`
 - `String`, `Double`, `Integer`, `Char`, `java.util.Date`,...
 - If a class implements `Comparable` then that is considered to be the class's *natural ordering*

java.util.Comparator<T> (an interface)

- `public int compare(T x1, T x2);`
 - Returns a value (< 0), (= 0), or (> 0)
 - ♦ (< 0) implies `x1` is before `x2`
 - ♦ (= 0) implies `x1.equals(x2)` is true
 - ♦ (> 0) implies `x1` is after `x2`
- Can often use a `Comparator` when a class's natural order is not the one you want
 - `String.CASE_INSENSITIVE_ORDER` is a predefined `Comparator`
 - `java.util.Collections.reverseOrder()` returns a `Comparator` that reverses the natural order

SortedSet Implementations

- `java.util.TreeSet<E>`

- constructors:

```
public TreeSet();  
public TreeSet(Collection<? extends E> c);  
public TreeSet(Comparator<? super E> comparator);  
...
```

- Write a method that prints out a **SortedSet** of words in order
- Write a method that prints out a **Set** of words in order

java.util.List<E> (an interface)

- List extends Collection
- Items in a list can be accessed via their index (position in list)
- The add() method always puts an item at the end of the list
- The iterator() returns the elements in list-order
- Methods (in addition to those inherited from Collection):
 - public E get(int index);
 - ◆ Returns the item at position index in the list
 - public E set(int index, E x);
 - ◆ Places x at position index, replacing previous item; returns the previous item
 - public void add(int index, E x);
 - ◆ Places x at position index, shifting items to make room
 - public E remove(int index);
 - ◆ Remove item at position index, shifting items to fill the space;
 - ◆ Returns the removed item
 - public int indexOf(Object x);
 - ◆ Return the index of the first item in the list that equals x (x.equals())
 - ...

List Implementations

- `java.util.ArrayList<E>` (an array; uses array-doubling)
 - Constructors
 - ♦ `public ArrayList();`
 - ♦ `public ArrayList(int initialCapacity);`
 - ♦ `public ArrayList(Collection<? extends E> c);`
- `java.util.LinkedList <E>` (a doubly-linked list)
 - Constructors
 - ♦ `public LinkedList();`
 - ♦ `public LinkedList(Collection<? extends E> c);`
- Both include some additional useful methods specific to that class

Efficiency Depends on Implementation

- `Object x = list.get(k) ;`
 - $O(1)$ time for `ArrayList`
 - $O(k)$ time for `LinkedList`

- `list.remove(0) ;`
 - $O(n)$ time for `ArrayList`
 - $O(1)$ time for `LinkedList`

- `if (set.contains(x)) ...`
 - $O(1)$ expected time for `HashSet`
 - $O(\log n)$ for `TreeSet`