Generic Types

and the Java Collections Framework

Lecture 15
CS2110 – Fall 2011

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 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

//removes all 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();
    }
}

Another Example

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();

Map 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

The Java compiler determines that the cast is not necessary, based on the declared type.

In this example, grades.get("John") is known at compile time always to be an Integer.

Map grades = new HashMap<String, Integer>(){
grades.put("John", new Integer(67));
grades.put("Fred", new Integer(72));

Integer x = grades.get("John");
sum = sum + x.intValue();

Autoboxing

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

Map grades = new HashMap<String, Integer>(){
grades.put("John", new Integer(67));
grades.put("Fred", new Integer(72));

Integer x = grades.get("John");
sum = sum + x.intValue();
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 are of type T
- Specify type in declaration, can be checked at compile time
  - Can eliminate explicit casts
  - No need for the runtime check

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 would be necessary
  - 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>`

```java
Stack<Integer> s = new Stack<Integer>();
s.push(new Integer(7));
Stack<Object> t = s; // gives compiler error
// cannot cast
Stack<Integer> is not a subtype of Stack<Object>
```

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

```java
Stack<Integer> s = new Stack<Integer>();
s.push(new Integer(7));
t = s; // compiler allows this
// produces a warning
```

Programming with Generic Types

- 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

```java
// old
void printCollection(Collection c) {
    for (Object e : c) {
        System.out.println(e);
    }
}

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

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

Bounded Wildcards

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

- Note that if we declared the parameter c to be of type `List<Comparable>`, then we could 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 `Integer` in our `List<String>`
- Wildcards let us specify exactly what types are allowed
Generic Methods

- Adding all elements of an array to a Collection

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

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

- See the online Java tutorial for more info on generics
  http://download.oracle.com/javase/tutorial/java/generics/

Generic Classes

```java
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();
    }
}
```

```
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)
  - SortedMap
  - Map (i.e., Dictionary)
- Classes
  - HashSet
  - TreeSet
  - ArrayList
  - LinkedList
  - HashMap
  - TreeMap
  - Iterator
  - Iterable
  - ListIterator

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

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

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

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

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

```java
public Iterator<E> iterator();
```
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<? super E> c);
  - public boolean addAll(Collection<? extends E> c);
  - public boolean removeAll(Collection<? super E> c);
  - public boolean retainAll(Collection<? super E> 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.compareTo(x) < 0
    - (= 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);
  - public E set(int index, E x);
  - public void add(int index, E x);
  - public E remove(int index);
  - public int indexOf(Object x);

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