CS 211

Computers and Programming

http://www.cs.cornell.edu/courses/cs211/2005su

Generics and the For-Each loop
(From Sun's online Java tutorials)

The For-Each loop

- Over Collections and Arrays, Java 5 supports enhanced **`for`** loop functionality.

Instead of

```
void cancelAll(Collection<TimerTask> c) {
        for (Iterator<TimerTask> i = c.iterator();
        i.hasNext(); ) i.next().cancel();
}
```

We can write

```
void cancelAll(Collection<TimerTask> c) {
        for (TimerTask t : c)
                t.cancel();
}
```

Read the lower for loop as “for each TimerTask t in c”

Restrictions on For-Each

- Only works for collections in the Java Collections Framework (and arrays)

- There are many cases where you'll need to use a normal for loop (like when you need access to the index, or want to call the remove method from the iterator

Why Generics?

- The motivation for generics is for code to be cleaner, easier to write, and safer. The code below isn't pretty:

```
//Removes 4-letter words from c.
//Elements must be strings
static void expurgate(Collection c) {
  for (Iterator i = c.iterator(); i.hasNext();)
       if (((String) i.next()).length() == 4)
       i.remove();
}
```

- A `Collection` contains elements of type `Object`
- The cast to `String` require that the *programmer* know the dynamic type of `i.next()`. We'll only find out if the cast works at runtime.

Using Generics

- Here's the same method using generics:

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

- The compiler now knows the type of element in the Collection, so it knows that **`i.next()`** returns a **`String`**.

Defining Generics

```
public interface List<E> {
   void add(E x);
   Iterator<E> iterator();
}
public interface Iterator<E> {
   E next();
   boolean hasNext();
}
```

- Here **`E`** is the the formal type parameter (or just parameter) for the interfaces **`List`** and **`Iterator`**
- When you define a generic class or interface in this way, you can then use the parameter just as you would a normal type in your definitions.

Generics and Subtypes

- Subtyping with generics is harder than you think:

```
List<String> ls = new ArrayList<String>();
List<Object> lo = ls; // This is illegal
```

- Because **`String`** is a subtype of **`Object`**, you may be tempted to think **`List<String>`** is a subtype of **`List<Object>.`** This is wrong!

- From the tutorial:

  "In general, if Foo is a subtype (subclass or subinterface) of Bar, and G is some generic type declaration, it is **not** the case that G<Foo> is a subtype of G<Bar>. This is probably the hardest thing you need to learn about generics, because it goes against our deeply held intuitions."

Wildcards

- Sometimes the parameter will be too restrictive:

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

- The problem is the **`Collection<Object>`** isn't a supertype of every **`Collection<E>.`** The above method only works on a collection of **`Objects.`**
- We can fix this using Wildcards:

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

Using Wildcards

- Think of the <?> as denoting any possible type. G<?> is a supertype of every G<T>.

- Since we don't actually know the parameter when using wildcards, typically we have to use **`Object`** in the method definition.

- Sometimes you want to be slightly more restrictive:

```
G<? extends Foo>// restricts to G<T> where T must
                  //extend FOO
G<? super Bar>// restricts to G<T> where Bar must
                  //extend T
```

Generic Methods

- This won't work:

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

- The problem is that we don't know what kind of Collection c is, so we can add and o of type Object.
- The solution: use parameters in method declaration:

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

- The method now can refer to a parameter T anywhere in the definition.