CSE 12:

Basic data structures and object-oriented design

Jacob Whitehill
jake@mplab.ucsd.edu

Lecture Seven
10 Aug 2011

More on Java generics.

Making ArrayList generic

- We've already discussed the benefit to the user of making a collection generic.
- We also saw (in brief) how to instantiate a generic type, and a bit on how to implement a generic data structure.
  - But let's be a bit more precise.
  - We'll keep going with the `ArrayList<T>` example.

Making `List` generic

- Before creating a generic `ArrayList` class, let's go back to the "contract" between implementor and user -- the `List` interface.
- The benefits of a generic interface are exactly analogous to the benefits of a generic class:
  - When you use a variable of the interface type, the compiler will check that the types are consistent:

```
final List<Integer> list = ... // some concrete impl
list.add(new Integer(5));  // ok
list.add("test");  // not ok -- compile-time error
```

Making List generic

- As described before, when writing a generic `List`, we include a type parameter at the start of the class definition.
  - The type parameter tells the generic `List` interface which type of element the list can accept.
- We can define a generic List interface as follows:

```
interface List<T> {
  int size ();
  void add (T element);
  T get (int index);
  void remove (int index);
}
```

Generics syntax

- Let's examine more carefully how the syntax works:

```
interface List<T> {
  int size ();
  void add (T element);
  T get (int index);
  void remove (int index);
}
```

Generics syntax

- Let's examine more carefully how the syntax works:

```
interface List<T> {
  int size ();
  void add (T element);
  T get (int index);
  void remove (int index);
}
```

When we write angled brackets just after the type name, we are *declaring* a *type parameter*. Here, the type parameter name is `T`.

This is analogous, in a Java method signature, to declaring a data parameter and giving it the name `student`.

```
void method (Student student) {
  student.setAge(24);
  student.printAddress();
}
```

Generics syntax

- Let's examine more carefully how the syntax works:

```
interface List<T> {
  int size ();
  void add (T element);
  T get (int index);
  void remove (int index);
}
```

Following the declaration of type parameter T, whenever we write T, we are *using* the type parameter's *value*.

This is analogous, in a Java method signature, to *using* that parameter inside the method body.

```
void method (Student student) {
  student.setAge(24);
  student.printAddress();
}
```

Generics syntax

- Now, suppose we want the **`List`** interface to extend the **`Iterable`** interface. We could write:

```
interface List<T> extends Iterable<T> {
  int size ();
  void add (T element);
  T get (int index);
  void remove (int index);
}
```

Despite the angled brackets, we are actually "using" `T`, not declaring `T`. We are "passing `T` to the generic `Iterable` interface."

This is analogous, in a Java method signature, to passing the parameter to another method

```
void method (Student student) {
  student.setAge(24);
  student.printAddress();
  otherMethod(student);
}
```

Generics syntax

- Bear in mind that *type parameters* are passed to a generic class at *compile-time*, whereas *data parameters* are passed to a method at *run-time*.

Making **`ArrayList`** generic

- Now that we have a generic **`List`**, we can define a generic **`ArrayList`**.
  - As mentioned last lecture, this consists mostly of replacing "**`Object`**" with "**`T`**":

```
class ArrayList<T> implements List<T> {
  T[] _underlyingStorage;
  int _numElements;

  public void add (T element) { ... }
  public T get (int index) { ... }
  public Iterator<T> iterator () {
  }

  private class ArrayListIterator implements Iterator<T> {
      ...
      T next () { ... }
      void remove () { ... }
      boolean hasNext () { ... }
  }
}
```

Making ArrayList generic

- There is one important exception, however:
  - In the constructor `ArrayList()`, we *cannot* write:

    `_underlyingStorage = new T[128];`

  - The Java compiler will give an error: "`generic array creation`".
  - It would also be illegal to try to write:
    `final T element = new T();`
  - Why?
    - It has to do with how generics are implemented "under the hood".

Erasure

- Java generics are implemented based on the principle of *erasure*.
- In one sentence:
  - After the Java compiler checks that the types are ok, it *erases* the type parameters associated with generic classes/methods and replaces them with just "`Object`". *

* Not quite true -- it actually replaces them with the upper bound of the type parameter.

Erasure

- Let's now define erasure more leisurely. Consider:

```
final List<String> list = new ArrayList<String>();
```

- The `List` was instantiated with a type parameter set to `String`.

- This means that `list.add(o)` now expects `o` to be of `String` type. It will then verify that variables passed to `add(o)` have the correct type:

```
list.add("yup");  // ok
list.add(new Object());  // will not compile
```

Erasure

- Now, after verifying that all type parameters are compatible with the generic `List`, the compiler proceeds to compile your code.
  - The compile strips away ("erases") all of the type parameters.
  - The code

    ```
    final List<String> list = new List<String>();
    ```

    is essentially replaced by:

    ```
    final List list = new List();
    ```

  - We're right back where we started -- an `List` of `Objects`!

Erasure

- Actually, not quite -- we still get two big benefits:
  - The compiler already verified that in all calls to `add(o)`, `o` was compatible with the `list`'s type.
    - No possibility of adding non-`String`s to a list that's supposed to contain only `String`s.
  - We didn't have to cast the result of `get(index)` to be `String`.

Erasure

- However, the erasure does have some suboptimal side effects:
  - We cannot instantiate an object of generic type T:
    ```
    final T t = new T();  // won't compile
    ```
    - Reason: After stripping away the type information T, *the JVM wouldn't know which constructor to call.*
  - We also cannot instantiate arrays of generic type:
    ```
    final T[] array = new T[];  // won't compile
    ```

Arrays of generic type

```
final T[] array = new T[];  // won't compile
```

- As a work-around, we have to instantiate an array of a *particular* (non-generic) type. An array of `Objects` will actually be sufficient for `ArrayList`:

```
final T[] array = (T[]) new Object[128];
```

  - The ugly downcast is back.
  - However, we only have to do this once in all of `ArrayList`.
  - Since this one line of code is an implementation of `ArrayList`, the user need never be bothered by it.

Arrays of generic type

```
final T[] array = (T[]) new Object[128];
```

- In this “workaround” solution, we are once again “promising” the Java compiler that the `Object[]` we instantiate is really of type `T[]`.
  - If it’s not, we’ll end up with a `ClassCastException`.
  - Because we’re downcasting from `Object[]` to a generic type `T`, the compiler will issue a warning that the types are “unchecked”.
  - In this particular instance, we can safely ignore this warning.

Erasure

- FYI: C++ offers “templates” (analogous to generics).
  - Templates are not implemented using erasure.
    - Instead, the compiler essentially compiles a separate version of your generic class *for every type parameter you use*.
  - In C++, it is legal to write **`new T()`**;

Data structures: a quantitative perspective.

Data structures so far

- Up to now, we've focused on data structures from a software construction perspective:
  - Data structures as ADTs.
  - Separation of implementation from interface.
  - Encoding of the user's data in a sequence of bits.

Data structures: a quantitative analysis

- Just as important is the quantitative performance of those structures, e.g.:
  - **Time cost**: If I have a linked list of 100 elements, how long will it take to find a particular element? What if the list is 1000 elements long? 10,000?
  - **Space cost**: How much overhead (e.g., in **`Node`**s) is there in a **`DoublyLinkedList12`** versus an **`ArrayList`**?

Data structures: a quantitative analysis

- In the remainder of this lecture we will discuss *algorithmic analysis*, in particular, methods of estimating the time cost of algorithms.
- Data structures and algorithms are invariably coupled:
  - Without an algorithm, the data are useless.
  - Without a data structure, the algorithm can't accomplish anything -- they need "space" to execute.

Measuring time cost

- Instead of measuring time cost in terms of seconds, milliseconds, etc., we will count the "number of abstract operations".
- Examples of "abstract operations" include:
  - `i = i + 1;  // Assignment and/or arithmetic`
  - `if (i > 5) {  // Comparison`
- On the other hand, calling another method -- i.e., *another algorithm* -- would *not* be considered a single, abstract operation:
  - `otherMethod();  // Have to look inside otherMethod!`

Measuring time cost

- As with all algorithmic analysis, we are interested in *how the time cost grows as the size of the input* to the algorithm grows:
  - For instance, if we want to sort a list of numbers, and the size of the list is *N*, then we want to describe, as a function of *N*, how many operation the sort procedure will take.
- For the case of analyzing data structures and their associated storage/retrieval/removal algorithms, the input size *N* will often be the *number of data already stored* in the ADT.

Measuring time cost

- We are interested in asymptotic analysis:
  - We don't care if the time cost is n, or $3n$, or $0.1n$ -- the main thing is that it's "something times n".
  - We *do* care whether it's n or n^2 or 2^n.
- Despite the fact that asymptotic analysis hides a lot of detail, it is still a very useful tool for comparing and selecting algorithms and data structures.

Unit testing.

Testing is (obviously) important

- Anyone who has written >0 programs knows that they rarely work the first time.
- There are different kinds of approaches to testing software:
  - "Macro-level" testing -- after the software is finished, test its functionality by running it as any "normal user" would.
    - E.g., hire 20 people to run the newest version of Microsoft Word for 1 day and use it normally.
    - We could conceivably *automate* the testing process by using a keyboard-and-mouse simulator.

Testing is (obviously) important

- In this course, however, we are more interested in "micro-level" testing:
  - "Micro-level" testing tests individual classes and methods to make sure they behave as intended.
    - Example: Given a DoublyLinkedList12 class, let's test whether the boolean remove() method works properly.
  - Micro-level testing is sometimes known as *unit-testing* -- test the "units" of a program's code.

Testing a List implementation

```
public interface List {
  // Adds o to the "back" of the list, i.e.,
  // o becomes the element with the highest
  // index in the List.
  void add (Object o);

  // Returns the element stored at the specified
  // index.
  Object get (int index)
    throws IndexOutOfBoundsException;

  // Removes the element stored at the specified
  // index.
  void remove (int index)
   throws IndexOutOfBoundsException;

  // Returns the number of elements stored in
  // the List.
  int size ();
}
```