CSE 326: Data Structures

Java Generics & JUnit 4

Section notes, 1/15/2009

slides by Hal Perkins

Type-Safe Containers

- Idea – a class or interface can have a type parameter:

```
public class Bag<E> {
    private E item;
    public void setItem(E x) { item = x; }
    public E getItem( ) { return item; }
}
```

- Given such a type, we can create & use instances:

```
Bag<String> b = new Bag<String>();
b.setItem("How about that?");
String contents = b.getItem();
```

Why?

- Main advantage is compile-time type checking:
  - Ensure at compile time that items put in a generic container have the right type
  - No need for a cast to check the types of items returned; guaranteed by type system
- Underneath, everything is a raw object, but we don’t have to write the casts explicitly or worry about type failures

Type Erasure

- Type parameters are a compile-time-only artifact. At runtime, only the raw types are present
- So, at runtime, the compile-time class Bag<E> is just a Bag (only one instance of class Bag), and everything added or removed is just an Object, not a particular E
  - Casts, etc. are inserted by compiler as needed, but guaranteed to succeed if generics rules are obeyed
  - Underlying code and JVM is pre-generics Java
- Ugly, but necessary design decision
  - Makes it possible for new code that uses generics to interoperate with old code that doesn't
  - Not how you would do it if you could start over

Type Erasure Consequences

- Code in a class cannot depend on the actual value of a type parameter at runtime. Examples of problems:

```
public class Bag<E> {

   private E item; // OK
   private E[ ] array; // also OK
   public Bag() {
      item = new E(); // error – new E() not allowed
      array = new E[10 ]; // error – new E[] also not allowed
   }
}
```

But I Need to Make an E[ ]!!!!

- Various solutions. For simple case, we can use an unchecked cast of an Object array (which is what it really is underneath anyway)

```
E[ ] stuff = (E[ ])new Object[size];
```

  - All the other code that uses stuff[ ] and its elements will work and typecheck just fine
- Be sure you understand the cause of *all unchecked* cast warnings & limit to “safe” situations like this
- More complex solutions if you want more type safety or have more general requirements – see references for detailed discussions

Example with “Generic” Array

```
public class Bag<E> {
   // instance variable
   E[ ] items;
   // constructor

   public Bag() { items = (E[ ]) new Object[10]; }
   // methods

   public void store(E item) { items[0] = item; }
   public E get( ) { return items[0]; }
}
```

References

- Textbook (Weiss), sec. 1.5.3
- Sun online Java tutorial

  java.sun.com/docs/books/tutorial/extra/generics/index.html

- For the truly hard-core:

  Java Generics and Collections,
  Maurice Naftalin & Philip Wadler, O'Reilly, 2006

  The Java Programming Language, **4**th ed.,
  Arnold, Gosling & Holmes, A-W, 2006

- And for the Language Lawyers in the crowd:

  The Java Language Specification, **3**rd ed.,
  Gosling, Joy, Steele & Bracha, A-W, 2005

Testing & Debugging

- Testing Goals
  - Verify that software behaves as expected
  - Be able to recheck this as the software evolves
- Debugging
  - A controlled experiment to discover what is wrong
  - Strategies and questions:
    - What's wrong?
    - What do we know is working? How far do we get before something isn't right?
    - What changed?

      (Even if the changed code didn't produce the bug, it's fairly likely that some interaction between the changed code and other code did.)

Unit Tests

- Idea: create ***small tests that verify individual*** properties or operations of objects
  - Do constructors and methods do what they are supposed to?
  - Do variables and value-returning methods have the expected values?
  - Is the right output produced?
- Lots of small unit tests, each of which test something specific; not big, complicated tests
  - *If something breaks, the broken test should be a great clue about where the problem is*

JUnit 4

- Test framework for Java Unit tests
- Idea: implement classes that have JUnit tests
- Each test in the class has the `@Test` annotation
- Each test performs some computation and then checks the result
- Optional: method with `@Before` tag to initialize instance variables or otherwise prepare for each test
- Optional: method with `@After` to clean up after each test
  - Less commonly used than `@Before`

Example

```
import static org.junit.Assert.assertEquals;
import org.junit.Test;
public class CalculatorTest {

   @Test
   public void testAddition() {
      Calculator calc = new Calculator();
      int expected = 7;
      int actual = calc.add(3, 4);
      assertEquals("adding 3 and 4", expected, actual);
   }
   ...
}
```

Exceptions

```
@Test
public void testDivisionByZero() {
   Calculator calc = new Calculator();
   try { // verify exception thrown
      calc.divide(2, 0);
      fail("should have thrown an exception");
   } catch (ArithmeticException e) {
      // do nothing - this is what we expect
   }
}
```

What Kinds of Checks are Available

- Need to include `import static org.junit.Assert.*;`
- Look in junit.framework.Assert (JavaDocs on www.junit.org)

```
assertEquals(expected, actual);
  //works on any type except double; uses .equals() for objects

assertEquals(messsage, expected, actual);
  //all have variations with messages

assertEquals(expected, actual, delta);
     // for doubles to test "close enough"

assertFalse(condition);
assertTrue(condition);

assertNotNull(object);
assertNull(object);

fail();
```

@Before

- If the tests require some common initial setup, we can write this once and it is automatically executed before each test (i.e., each test starts with a fresh setUp)

```
import org.junit.Before;
public class CalculatorTest extends TestCase {
   private Calculator calc; // calculator object for tests
   /** initialize: repeated before each test */
   @Before
   protected void setUp() {
      calc = new Calculator();
   }
   // tests as before, but no local declaration of calc
```