Program specialization and metaprogramming

Laboratory work No. 6

Generic programming in Java

1. Aims

To get acquainted with generic programming in Java using Generics.

2. Tasks

Learn how to use Java generics for developing generic software components.

3. Work

When programmers start to design structures that collect values, they hit one immediate design problem: Should the structure contain be homogeneous, and contain only one type of values, or should the structure be heterogeneous and contain multiple types of values? There are many situations in which we want homogeneous data types - e.g., not just any list, but a list of strings or a list of integers. But, once you have decided to have homogeneous data types, how do we write general data types. Once we have implemented all the methods for a list of strings, the implementation for a list of integers or a list of other objects should be essentially the same.

How does the Java language allow the programmer to design generalized homogenous collections so that we can indicate the type of value in the collection? The typical strategy is to allow programmers to design collections (classes, interfaces) that have a type parameter. Such parameterized classes and interfaces are typically called generics. We'll analyse how to write generics in Java.

For further work download the Java class collection from Moodle course page.

A. Box classes for storing objects

a. One approach to generic code is to just store objects. The `BoxedObject` class represents that approach. Write a simple main class that allows you to experiment with `BoxedObject` values. Your experiment should include the following.

- Create a box and have two variables refer to that box. Update the contents of the box through one of the variables and verify that the new value is available through the other.
- Put an integer in a box, have two variables refer to that box, and increment the integer in the box using one of the variables.
- Put an integer in a box, have two variables refer to that box, have the first put an integer in the box, the second put a string in the box, and the first increment the value in the box.

b. Write a simple main class that contains an experiment that uses the generic `Box<T>` class to build boxes with different types and that verifies that this class works as advertised. Your experiment should include the following:

- Create a boxed string object and two variables that refer to that box. Change the contents of one and determine the effect on the other.
- Create a boxed integer object and two variables that refer to that box. Change the contents of one and determine the effect on the other.

- Create a boxed object object and two variables that refer to that box. Determine what happens if you put a string in the box. Determine what happens if you put an integer in the box.

c. Reflect on what, if anything, you've taken from these experiments.

B. Nesting Boxes

a. Boxes seem to be able to contain any kind of object. Can a box contain another box? In particular, do you think we should be able to write something like the following?

```
    String s1 = "Hello";
    Box<String> box1 = new Box<String>(s1);
    Box<Box<String>> bbox1 = new Box<Box<String>>(box1);
```

b. Check your answer experimentally.

c. Write code that changes the contents of `bbox1` directly or indirectly. (Do not refer to `s1` or `box1` in making those changes.)

d. Reflect on what, if anything, you've taken from these experiments.

C. Polymorphism and Generics

a. Determine experimentally whether you can assign an object of type `Box<Integer>` to a variable of type `Box<Object>`. If so, attempt to replicate the troublesome code from the reading. If not, take a note of what the error message says.

b. Do you think the following code should be legal? Why or why not?

```
  Box<Integer> ibox = new Box<Integer>(new Integer(5));
  Box<Object> obox = (Box<Objecdt>) ibox;
```

c. Check your answer experimentally.

d. Reflect on the implications of these experiments.

D. Testing Equality

One deficiency of the current `Box` class is that it lacks an `equals` method. When should a boxed value equal another value? In two cases: If the other contents of the box equals the other value or if the the other value is also a box and the contents of the two boxes ar ethe same.

a. Add an `equals` method to `Box` that follows these guidelines. (If you have trouble writing this method, you can glance ahead to part c and use the first method you find there.)

b. Use something like the following code to check whether your `equals` method works correctly.

```
package taojava.generics;

import java.io.PrintWriter;

/**
 * An experiment that lets us behave the behavior of various
 * equals methods.
```

```
 */
public class BoxEqualsExpt
{
  public static void observeEquals(PrintWriter pen, String prefix,
     Object o1, Object o2)
  {
    pen.print(prefix + ": ");
    pen.println("Does " + o1 + " equal " + o2 + "?: " + o1.equals(o2));
  } // check(PrintWriter, Box, Box)

  public static void main(String args[])
    throws Exception
  {
    PrintWriter pen = new PrintWriter(System.out, true);
    String s1 = "Hello";
    String s2 = "H" + "Yello".substring(1);
    String s3 = "Hello World".substring(0,5);
    String s4 = "hello";
    Box<String> box1 = new Box<String>(s1);
    Box<String> box2 = new Box<String>(s2);
    Box<Box<String>> bbox1 = new Box<Box<String>>(box1);
    Box<Box<String>> bbox2 = new Box<Box<String>>(box2);
    Box<Integer> ibox1 = new Box<Integer>(new Integer(42));

    observeEquals(pen, "s1/s1", s1, s1);
    observeEquals(pen, "s1/s2", s1, s2);
    observeEquals(pen, "s1/s3", s1, s3);
    observeEquals(pen, "s1/s4", s1, s4);

    observeEquals(pen, "b1/s1", box1, s1);
    observeEquals(pen, "b1/s2", box1, s2);
    observeEquals(pen, "b1/s3", box1, s3);
    observeEquals(pen, "b1/s4", box1, s4);
    observeEquals(pen, "b1/b1", box1, box1);
    observeEquals(pen, "b1/b2", box1, box2);

    observeEquals(pen, "b2/s1", box2, s1);
    observeEquals(pen, "b2/s2", box2, s2);
    observeEquals(pen, "b2/s3", box2, s3);
    observeEquals(pen, "b2/s4", box2, s4);
    observeEquals(pen, "b2/b1", box2, box1);
    observeEquals(pen, "b2/b2", box2, box2);

    observeEquals(pen, "s1/b1", s1, box1);

    observeEquals(pen, "bb1/bb1", bbox1, bbox1);
    observeEquals(pen, "bb1/bb2", bbox1, bbox2);
    observeEquals(pen, "bb1/b1", bbox1, box1);
    observeEquals(pen, "bb1/s1", bbox1, s1);
    observeEquals(pen, "b1/bb1", box1, bbox1);
    observeEquals(pen, "s1/bb1", s1, bbox1);

    observeEquals(pen, "b1/i1", box1, ibox1);
    observeEquals(pen, "i1/b1", ibox1, box1);

    pen.close();
  } //  main(String[])
} // BoxEqualsExpt
```

c. Consider the following implementations of the `equals` method. Which do you prefer? Why? Which does Java permit? Why?

```
  public boolean equals(Object other)
  {
    if (other instanceof Box)
      {
        Box that = (Box) other;
        return this.contents.equals(that.contents);
      } // it's a box
    else
      {
        return this.contents.equals(other);
      } // it's not a box
```

```
  } // equals(Object)

  public boolean equals(Object other)
  {
    return (this.contents.equals(other) ||
      (other instanceof Box) && (this.contents.equals(((Box) other).contents)));
  } // equals(Object)

  public boolean equals(Object other)
  {
    if (other instanceof Box<T>)
      {
        Box<T> that = (Box<T>) other;
        return this.contents.equals(that.contents);
      } // it's a box of the same type
    else if (other instanceof T)
      {
        return this.contents.equals((T) other);
      } // it's the same type
    else
      {
        return false;
      } // Nothing sensible
  } // equals(Object)

  public boolean equals(Object other)
  {
    if (other instanceof Box)
      {
        Box that = (Box) other;
        return this.contents.equals(that.contents);
      } // it's a box
    else if (this.contents.equals(other))
      {
        return true;
      } // if our contents equals the other value
    else
      {
        return other.equals(this);
      } // default case
  } // equals(Object)

  public boolean equals(Object other)
  {
    if (other instanceof Box)
      {
        Box that = (Box) other;
        return this.contents.equals(that.contents);
      } // it's a box
    else if (this.contents.equals(other))
      {
        return true;
      } // if our contents equals the other value
    else
      {
        return other.equals(this.contents);
      } // default case
  } // equals(Object)
```

d. Reflect on what you learned from these experiments.

E. Extending Generic Classes

Predict the answer to each of the following and then check your answers experimentally. In your experiments, you will likely need to create a constructor for the class.

a. Can a generic class extend another generic class? For example, can a class `NewBox<T>` extend `Box<T>`?

b. Can a non-generic class extend a generic class? For example, can `NewBox` extend `Box<T>`?

c. Can a non-generic class extend an instantiated generic class? For example,, `BoxedString` extend `Box<String>`?

d. What effect does the type of the instantiated superclass have on the abilty to extend. Can the following `BoxedString` extend `Box<Object>`?

```
public class BoxedString
  extends Box<Object>
{
  public BoxedString(String s)
  {
    super(s);
  } // BoxedString(String)

  public String get()
  {
    return super.get();
  } // get()

  public void put(String val)
  {
    super.put(val);
  } // put(String)
} // class BoxedString
```

e. Can a generic class extend a nongeneric class? `Box`.

```
public class Box<T>
  extends BoxedObject
{
  public Box(T val)
  {
    super(val);
  } // Box(T)

  public T get()
  {
    return super.get();
  } // get()

  public void put(T val)
  {
    super.put(val);
  } // put(T)
} // class Box<T>
```

f. Reflect on what, if anything, you learned from these experiments.

F. Simple Expandable Arrays

Build a generic “expandable array” class. You'll find that generic class in the repository for this lab.

a. Read through `SEAExpt.java` and predict what the output will be.

b. Compile and run `SEAExpt.java` to see what the output is.

c. Create an expandable array of strings, assign some values to it, and print them out. Here's a start.

```
  ExpandableArray<String> strings =
      new SimpleExpandableArray<String>();
  ...
  for (int i = 0; i < 10; i++)
    {
```

```
      pen.println("strings[" + i + "] = " + strings.get(i));
    } // for
```

d. What do you expect to happen if you assign a string to an element of `numbers` or a number to an element of strings??

e. Check your answer experimentally.

f. What do you expect to happen if we leave out the type when we construct `numbers`, as in the following?

```
ExpandableArray<BigInteger> numbers =
    new SimpleExpandableArray();
```

g. Check your answer experimentally.

h. What do you expect to happen if we leave out the type when we declare `strings`, as in the following?

```
ExpandableArray strings =
    new SimpleExpandableArray();
```

i. Check your answer experimentally.

j. Summarize what you've learned in these exercises.

G. Generic search method

Build a generic search method. You'll find that code in the repository.

a. Read through `SearchExpt.java` and predict what the output will be.

b. Compile and run `SearchExpt.java` to see what the output is.

d. What do you expect to happen if you try to search `strings` with `odd` or `numbers` with `small`?

e. Check your answer experimentally.

f. What do you expect to happen if we try to generalize the declaration of `strings`, as in the following?

```
    Object[] strings = new Object[] { ... };
```

g. Check your answer experimentally.

h. Revise the `short` predicate so that it takes an object as a parameter, converts it to a string, and sees if it has fewer than five characters. Do you expect that new predicate to work with the updated `strings`?

i. Check your answer experimentally.

j. Summarize what you've learned in these exercises.

H. Extra 3: Predicates, Continued

a. What do you expect to happen if we restore the original declaration of `strings` and use the new version of `small`?

```
    String[] strings = new String[] { ... };
    ...
    Predicate<Object> small =
      new Predicate<Object>()
        {
          @Override
          public boolean holds(Object val)
          {
            return (val.toString().length() < 5);
          } // holds(Object)
        }; // new Predicate<Object>
    ...
    pen.println("A small string: " + SearchUtils.search(strings, small));
```

b. Check your answer experimentally.

c. What do you expect to happen if we use the new `small` predicate to search `numbers`?

```
    pen.println("A small integer: " + SearchUtils.search(numbers, small));
```

d. Check your answer experimentally.

e. Summarize what you've learned in this exercise.

I. Expanding with Vectors

Finish the following alternate implementation of `ExpandableArray`

```
public class VectorBasedExpandableArray
{
  Vector<T> values;
  ...
} // class VectorBasedExpandableArray
```

4. Report

Prepare a report of your work in this lab using a standard word processing application. The report is a single chapter of your final semester report. Final semester report will have to be uploaded to course Moodle page.

The content of report:

1. Title
2. Aims and tasks
3. Work done, described in steps and illustrated by screenshots and written or modified source code. Provide comments what you have done and what were the results (program outputs). Present answers to the questions given in the lab description.
4. Conclusions – what you have learned?