CS 112 – Introduction to Computing II

Lab 07 – Java Generic Types

Today in Lab:

Polymorphic ADTs and Java Generics

Computer Science

Polymorphic ADTs

Computer Science

One big problem with "reusing" code such as we have written is that we have to create an ADT (e.g., stack or queue) for a **specific type**:

```
class ArrayIntStack {
  private int [] A = new int[8];
  private int next = 0;

  public void push(int key) {
      A[next] = key;
      ++next;
  }
  public int pop() {
      int temp = A[next-1];
      --next;
      return temp;
  }

  public boolean isEmpty() {
    return (next == 0);
  }

  public int size() {
    return next;
  }
}
```

Polymorphic ADTs

Computer Science

If we need a stack for a different type of data, we have to create another whole class, although we can do this easily by using the editor to replace int by, e.g., double:

```
class ArrayIntStack {
  private int [] A = new int[8];
  private int next = 0;

  public void push(int key) {
    A[next] = key;
    ++next;
  }
  public int pop() {
    int temp = A[next-1];
    --next;
    return temp;
  }

  public boolean isEmpty() {
    return (next == 0);
  }

  public int size() {
    return next;
  }
}
```

➔ Find/Replace int by double ➔

```
class ArrayDoubleStack {
  private double [] A = new double[8];
  private int next = 0;

  public void push(double key) {
    A[next] = key;
    ++next;
  }
  public double pop() {
    double temp = A[next-1];
    --next;
    return temp;
  }

  public boolean isEmpty() {
    return (next == 0);
  }

  public int size() {
    return next;
  }
}
```

Polymorphic ADTs

But this is a really terrible idea: not only might we make a mistake and change something that should be an int into a double, we now have m**ultiple versions** of the same code that have to be **stored** and **maintained**. Or more than two! If we change one, we have to change the others! **It's a massive version of the "Multiple Update Problem."**

```
class ArrayIntStack {
  private int [] A = new int[8];
  private int next = 0;

  public void push(int key) {          ;
    A[next] = key;
    ++next;
  }
  public int pop() {
    int temp = A[next-1];
    --next;
    return temp;
  }

  public boolean isEmpty() {
    return (next == 0);
  }

  public int size() {
    return next;
  }
}
```

```
class ArrayDoubleStack {
  private double [] A = new double[8];
  private int next = 0;

  public void push(double key) {          ;
    A[next] = key;
    ++next;
  }
  public double pop() {
    double temp = A[next-1];
    --next;
    return temp;
  }

  public boolean isEmpty() {
    return (next == 0);
  }

  public int size() {
    return next;
  }
}
```

```
class ArrayStringStack {
  private String [] A = new String[8];
  private int next = 0;

  public void push(String key) {          ;
    A[next] = key;
    ++next;
  }
  public String pop() {
    String temp = A[next-1];
    --next;
    return temp;
  }

  public boolean isEmpty() {
    return (next == 0);
  }

  public int size() {
    return next;
  }
}
```

```
class ArrayCharStack {
  private char [] A = new char[8];
  private int next = 0;

  public void push(char key) {
    A[next] = key;
    ++next;
  }
  public char pop() {
    char temp = A[next-1];
    --next;
    return temp;
  }

  public boolean isEmpty() {
    return (next == 0);
  }

  public int size() {
    return next;
  }
}
```

Polymorphic ADTs

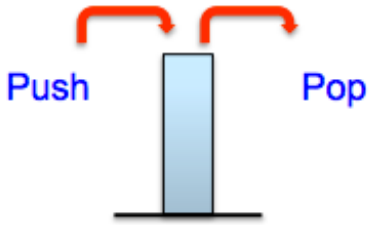
Interesting question: is there anything essential about the data and methods for a Stack that really depends on the type of the elements stored?

```
class ArrayIntStack {
  private int [] A = new int[8];
  private int next = 0;

  public void push(int key) {
    A[next] = key;
    ++next;
  }
  public int pop() {
    int temp = A[next-1];
    --next;
    return temp;
  }

  public boolean isEmpty() {
    return (next == 0);
  }

  public int size() {
    return next;
  }
}
```




```
class ArrayDoubleStack {
  private double [] A = new double[8];
  private int next = 0;

  public void push(double key) {
    A[next] = key;
    ++next;
  }
  public double pop() {
    double temp = A[next-1];
    --next;
    return temp;
  }

  public boolean isEmpty() {
    return (next == 0);
  }

  public int size() {
    return next;
  }
}
```

Polymorphic ADTs

Interesting question: is there anything essential about the data and methods for a Stack that really depends on the type of the elements stored?

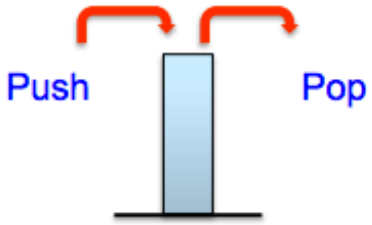
NO! You want proof? How about this: we made the change using Find/Replace.....

```
class ArrayIntStack {
  private int [] A = new int[8];
  private int next = 0;

  public void push(int key) {        ;
    A[next] = key;
    ++next;
  }
  public int pop() {
    int temp = A[next-1];
    --next;
    return temp;
  }

  public boolean isEmpty() {
    return (next == 0);
  }

  public int size() {
    return next;
  }
}
```




```
class ArrayDoubleStack {
  private double [] A = new double[8];
  private int next = 0;

  public void push(double key) {
    A[next] = key;
    ++next;
  }
  public double pop() {
    double temp = A[next-1];
    --next;
    return temp;
  }

  public boolean isEmpty() {
    return (next == 0);
  }

  public int size() {
    return next;
  }
}
```

Java Generic Types

Java's Generic Type system solves this problem.

The idea is to parameterize a class as if it were a method.

Consider the following (silly) methods:

```
void repeat2(string s) {
        System.out.println(s); System.out.println(s);
}

void repeat3(string s) {
        System.out.println(s); System.out.println(s);
        System.out.println(s);
}

// to use:

     repeat3("I am sick of cold weather");
```

Java Generic Types

Computer Science

To parameterize something, we take an essential part out and make it a variable that we can instantiate; then the same code can be used in a variety of ways. It avoids the multiple update problem and makes it possible to reuse code later.

Obviously, instead of writing repeat2(), repeat3(), ….., repeat100000()…..
we should just make the number of repetitions a parameter:

```
void repeat (string s, int N) {
     for(int I = 0; i < N; ++i)
         System.out.println(s);
}

// to use:

Repeat("I am sick of cold weather", 2);
```

Recall: Parameter passing is an assignment:

N = 2;

Java Generic Types

Computer Science

The EXACT SAME THING can be done with a Java class: we can take out one or more type names and make it a parameter.

The only thing to get straight is the syntax, which is a little different than parameters in methods.

Original code for a Stack holding Strings:

```
Public class Stack {
  private String[] A = new String[10];
  private int next = 0;

  public void push(String key) {
      A[next] = key; ++next;
  }
  public int pop() {
      String temp = A[next-1]; --next; return temp;
  }

  public boolean isEmpty() { return (next == 0); }

  public int size() {return next; }
}
```

Java Generic Types

Computer Science

Original code for an Stack holding Strings, with type of item stored in red:

```
public class Stack {
  private String[] A = new String[10];
  private int next = 0;

  public void push(String key) {
      A[next] = key; ++next;
  }
  public String pop() {
      String temp = A[next-1]; --next; return temp;
  }

  public boolean isEmpty() { return (next == 0); }

  public int size() {return next; }
}
```

Java Generic Types

Now we replace the type name String with a variable (can be any name but usually it is called Item or T or something descriptive, and capitalized):

```
public class Stack {
  private Item[] A = new Item[10];
  private int next = 0;

  public void push(Item key) {
      A[next] = key; ++next;
  }
  public Item pop() {
      Item temp = A[next-1]; --next; return temp;
  }

  public boolean isEmpty() { return (next == 0); }

  public int size() {return next; }
}
```

Java Generic Types

Computer Science

Now we replace the type name int with a variable (can be any name but usually it is called Item or T or something descriptive, and capitalized);

Next we add the parameter list to the name of the class in angle brackets: < >

```
public class Stack<Item> {
  private Item[] A = new Item[10];
  private int next = 0;

  public void push(Item key) {
      A[next] = key; ++next;
  }
  public Item pop() {
      Item temp = A[next-1]; --next; return temp;
  }

  public boolean isEmpty() { return (next == 0); }

  public int size() {return next; }
}
```

Note that there is no type (as in “int N”) because Item is itself a type!

Java Generic Types

Computer Science

Now we replace the type name int with a variable (can be any name but usually it is called Item or T or something descriptive, and capitalized);

Next we add the parameter list to the name of the class in angle brackets: < >

To use the class, we must supply an actual parameter whenever we use the name IntStack:

```
public class Stack<Item> {
  private Item[] A = new Item[10];
  private int next = 0;

  public void push(Item key) {
      A[next] = key; ++next;
  }
  public Item pop() {
      Item temp = A[next-1]; --next; return temp;
  }

  …    etc.    …..
}
```

```
Stack<String> S = new Stack<String>();
```

Java Generic Types

Computer Science

Now we replace the type name int with a variable (can be any name but usually it is called Item or T or something descriptive, and capitalized);

Next we add the parameter list to the name of the class in angle brackets: < >

To use the class, we must supply an actual parameter whenever we use the name IntStack;

The effect of this is exactly as if we had written the code with the actual parameter Integer substituted for the formal parameter Item---which is the exact same code we started with!

```
public class Stack {
  private String[] A = new String[10];
  private int next = 0;

  public void push(String key) {
      A[next] = key; ++next;
  }
  public String pop() {
      String temp = A[next-1]; --next; return temp;
  }

  public boolean isEmpty() { return (next == 0); }

  public int size() {return next; }
```

Java Generic Types

Computer Science

There is one more thing you need to know:

You can ONLY use reference types (e.g. Strings and classes), to instantiate Java Generics, and if you want to use primitive types, you have to use the Wrapper Types instead:

Integer = int Double = double Boolean = boolean Character = char

There is not much difference between these, except that variables are now references.

So, if you were thinking of doing this:

```
Stack<int> S = new Stack<int>();
```

You have to do this:

```
Stack<Integer> S = new Stack<Integer>()
```