Generic Types and Methods

October 24, 2005

Outline

Outline

Stack Holding Integers

```java
public class StackInt {
    private int[] values;
    private int counter; // the next position to insert value
    public StackInt(int size) {
        values = new int[size];
        counter = 0;
    }
    public void push(int value) {
        values[counter++] = value;
    }
    public int pop() {
        return values[--counter];
    }
    public int top() {
        return values[counter - 1];
    }
    public boolean isEmpty(){
        return (counter <= 0);
    }
}
```

Stack Holding Integers, cont.

```
public static void main(String[] args) {
    StackInt s = new StackInt(100);
    s.push(42);
    s.push(3);
    int i = s.pop();
    System.out.println(i);
    System.out.println(s.pop());
}
```

Stack Holding Strings

```
public class StackString {
    private String[] values;
    private int counter; // the next position to insert value
    public StackString(int size) {
        values = new String[size];
        counter = 0;
    }
    public void push(String value) {
        values[counter++] = value;
    }
    public String pop() {
        return values[--counter];
    }
    public String top() {
        return values[counter - 1];
    }
    public boolean isEmpty() {
        return (counter <= 0);
    }
}
```

Stack Holding Anything

```
public class StackObject {
    private Object[] values;
    private int counter; // the next position to insert valu

    public StackObject(int size) {
        values = new Object[size];
        counter = 0;
    }
    public void push(Object value) {
        values[counter++] = value;
    }
    public Object pop() {
        return values[--counter];
    }
    public Object top() {
        return values[counter - 1];
    }
    public boolean isEmpty() {
        return (counter <= 0);
    }
}
```

Stack Holding Anything, cont.

```
public static void main(String[] args) {
    StackObject s = new StackObject(100);
    s.push("Hello");
    s.push(3); // auto-boxing
    int i = (Integer)s.pop(); //down case + auto unboxing
    System.out.println(i);
    String str = (String)s.pop();
    System.out.println(str);
    s.push(s); // push myself onto myself
}
```

Problems

Problems

- Similar code is repeated
- Names of classes are not intuitive (StackInt and StackString)

Observations

- Difference between StackInt and StackString is the type
- Can use Object
  - Requires explicit downcasts
  - Can mix “apples” and “bananas”
- The only approach possible before Java 5.0

Outline

A Generic Stack

```
public class Stack<T> {
    private T[] values;
    private int counter; // the next position to insert value
    public Stack(int size) {
        values = (T[]) new Object[size];
        counter = 0;
    }
    public void push(T value) {
        values[counter++] = value;
    }
    public T pop() {
        return values[--counter];
    }
    public T top() {
        return values[counter - 1];
    }
    public boolean isEmpty() {
        return (counter <= 0);
    }
}
```

A Generic Stack, cont.

```
public static void main(String[] args) {
    // Stack using Strings
    Stack<String> s = new Stack<String>(100);
    s.push("hello");
    s.push("world");
    //s.push(3); // compile-time error
    String s1 = s.pop();
    System.out.println(s1);
    String s2 = s.pop();
    System.out.println(s2);

    // Stack using Integers
    Stack<Integer> si = new Stack<Integer>(100);
    si.push(42);
    si.push(142);
    int i = si.pop();
    System.out.println("i " + i);
    int j = si.pop();
    System.out.println("j " + j);
```

Advantages of Generic Stack

- Less code to write!
- Code more robust
  - compiler can type-check code better
- Code more readable
  - type are specified
  - Explicit downcasts are avoided
- Code more versatile and abstract
  - Code can be reused in more contexts
  - We love abstraction!

Note that:

```
Stack s1 = Stack(10); // non-generic or raw type
Stack<Object> s2 = new Stack<Object>(10); // a similar thing
```

Outline

Generics as Parameters and Return Values

```
public class UseStackShape1 {
    public static Stack<Shape> buildShapeStack(){
        Stack<Shape> stack = new Stack<Shape>(100);
        Shape s1 = new Shape(); stack.push(s1);
        Line l1 = new Line();
        stack.push(l1); // implicit upcast to Shape
        return stack;
    }
    public static Stack<Line> buildLineStack(){
        Stack<Line> lines = new Stack<Line>(100);
        lines.push(new Line()); lines.push(new Line());
        return lines;
    }
    public static void emptyAndDraw(Stack<Shape> stack){
        while(!stack.isEmpty()){
            Shape s = stack.pop(); s.draw();}
    }
    public static void main(String[] args) {
        Stack<Shape> s = buildShapeStack();
        emptyAndDraw(s);
        Stack<Line> l = buildLineStack();
        //emptyAndDraw(l); // not allowed
```

Generics as Parameters and Return Values, cont

```
public class UseStackShape2 {
    // ship
    public static void emptyAndDraw(
            Stack<? extends Shape> stack){ // upper-bound
        while(!stack.isEmpty()){
            Shape s = stack.pop(); s.draw();}
    }
    public static void main(String[] args) {
        Stack<Shape> s = buildShapeStack();
        emptyAndDraw(s);
        Stack<Line> l = buildLineStack();
        emptyAndDraw(l);
    }
```

Generics as Parameters and Return Values, cont

- Cannot upcast Stack<Line> to Stack<Shape>
- Stack<Line> is not a subtype of Stack<Shape> (the same as above)

Example that causes problem due to a modification

```
// in main of class  UseStackShape1
Stack<Line> lines = buildLineStack();
Stack<Object> objects = lines; // upcast??
objects.push(new Car()); // this does not make sense
```

Concrete Parameterized Types

Concrete parameterized type = instantiations of a generic type

```
Stack<Line> lines;
Stack<Object> objects = lines;
```

Cannot be used

- In creating arrays, e.g., `T[] a = new T[10];`
- In exception handling
- In instanceof expersions, e.g., `if (c instanceof Stack<String>){}`
- In class literals (see Java reflection)

Raw Type

- A generic type without a type parameter
- Is compatible with all instantiations of the generic type
- Are used to ensure backwards compatibility

```
public class RawType {
    public static void main(String[] args) {
        Stack raw = new Stack(10); // raw type
        raw.push(10); // type unsafe, compiler warning
        Stack<String> strs = new Stack<String>(10);
        strs.push("Hallo"); // type safe
        Stack<Integer> ints = new Stack<Integer>(10);
        ints.push(10); // type safe
        //ints = strs; // not allowed
        //strs = raw; // type unsafe
        raw = strs;
        System.out.println(raw.pop());
    }
}
```

Wildcard Types

- public void myMethod(Stack<?> arg)
  - Method accepts all generic classes of Stack
- public void myMethod(Stack<? extends Shape> arg)
  - Method accepts stack of any subtype of class Shape
- public void myMethod(Stack<? super Shape> arg)
  - Method accepts stack of any supertype of class Shape

This applies to both formal parameters and return types.

```
public static Stack<?> buildStack(){
    Stack<Integer> s = new Stack<Integer>(10);
    s.push(10);
    s.push(20);
    return s;
}
```

Static Variables

What is the output?

```
public class StaticVar<T> {
    public static int counter = 0;
    public StaticVar(){
        counter++;
    }
    public static void main(String[] args) {
        StaticVar<Integer> i = new StaticVar<Integer>();
        System.out.println(i.counter);
        StaticVar<String> s = new StaticVar<String>();
        System.out.println(s.counter);
    }
}
```

Is this legal?

```
class StaticVar<T>
    public static T v;
    // snip
}
```

Outline

Example

```java
public class GenericMethod {
    public void methodAsYouKnowThem(Shape arg){
        System.out.println("I'm pretty normal");
    }
    public <T> void veryGeneric(T arg){
        System.out.println("I'm very generic " + arg);
    }
    public <T> void lessGeneric
            (Stack<? super T> stack, T element){
        stack.push(element);
    }
    public static void main(String[] args) {
        GenericMethod gm = new GenericMethod();
        gm.methodAsYouKnowThem(new Shape());
        gm.veryGeneric(new Shape());
        gm.lessGeneric(new Stack<Shape>(10), new Shape());
        gm.lessGeneric(new Stack<Shape>(10), new Line());
    }
}
```

Notes

Usage

- Methods (static and non-static) and constructors can be generic
- Are called like non-generic method (type is inferred by context)

Purpose

- To create type relationships between formal parameters

Another Example

```
public class Collections {
    public static <A extends Comparable<A>>
                                    A max(Collection<A> xs) {
        Iterator<A> xi = xs.iterator();
        A w = xi.next();
        while (xi.hasNext()) {
            A x = xi.next();
            if (w.compareTo(x) < 0)
                w = x;
        }
        return w;
    }
}
```

[Source: http://www.langer.camelot.de/GenericsFAQ/FAQSections/ParameterizedMethods.html]

Generic Constructors

```
public class GenericCons {
    public <T> GenericCons(T arg) {
        System.out.println("" + arg);
    }
    public <T, S> GenericCons(T arg1, S arg2) {
        System.out.println("" + arg1 + " " + arg2);
    }
    public <T> GenericCons(T arg1, Stack<? extends T> arg2) {
        System.out.println("2" + arg1 + " " + arg2);
    }
    public static void main(String[] args) {
        String s1 = "hello";
        String s2 = "world";
        GenericCons gc = new GenericCons(s1);
        GenericCons gc1 = new GenericCons(s1, s2);
        Stack<String> stack = new Stack<String>(10);
        stack.push(s2);
        GenericCons gc2 = new GenericCons(s1, stack);
    }
}
```

Outline

Summary

- Generics another type of abstraction!
- Both types and methods can be generics
- Generics are heavily used in the collection library
- There are many details and implications
    - Good FAQ: `http://www.langer.camelot.de/GenericsFAQ/FAQSections/ParameterizedMethods.html`