CSE 341 - Programming Languages - Autumn 2012
Java Generics & Nested Classes

Java's type rules

A brief recap: remember Java's rules for type compatibility from 142/143. If B is a subtype of A, then we can use an object of type B anywhere that we expect something of type A. For example:

```
Object a;
Point p = new Point(10,20);
Point q;
a = p;  // legal since a point is a kind of an object
q = a;  // NOT LEGAL (since the compiler only knows that a is of type Object)
q = (Point) a;  // legal (since we put in a runtime type check using a cast)
```

Similarly, if we had a method that expects an object as a parameter, we can pass it a Point, a String, or any other object.

Both classes and interfaces can be used as types in Java. For example, in the code above we used the class `Point` as a type.

The Subtype Relation

If some code needs something of type A, and B is a subtype of A, then it should always be sound to give it something of type B instead (substitutability).

If C is a subtype of B, and B is a subtype of A, then C is a subtype of A (transitivity).

- If A and B are classes, and B extends A, then (considering A and B as types) B is a subtype of A
- If A and B are interfaces, and B extends A, then (considering A and B as types) B is a subtype of A
- If A is an interface and B is a class that implements A, then (considering A and B as types) B is a subtype of A

In Java, a class B extends a class A if B is explicitly declared to be a subclass of A (and analogously for interfaces). This is called *nominal subtyping* (i.e. we are basing the subtype relation on the names of the types). An alternative -- not commonly used in statically typed languages -- is *structural subtyping*; in this case we determine the subtype relation based on the structure of the types. For example, is `Octopus1` a subtype of `Octopus2`? (So if we have a variable declared to be of type `Octopus1`, will the compiler let us assign something of type `Octopus2` to it?)

```
public class Octopus1 {
    public int tentacles() {
        return 8;
    }
}

public class Octopus2 {
    public int tentacles() {
        return 8;
    }
}
```

Nominal subtyping (and Java) says no; structural subtyping says yes.

Parametric Polymorphism

Recall that Haskell's type system allows one to declare types such as

```
(a -> b) -> [a] -> [b]
```

(the type of the `map` function).

In early versions of Java, programmers couldn't declare types of this sort. This led to less precise type declarations, less machine-checkable documentation, and more runtime casts. For example, the built-in collection classes in Java (Set, ArrayList, etc) all had Object as the type of the contents -- you couldn't declare that x is a set of strings, or a set of points.

For example, consider the following Java code fragment in before version 5.

```
ArrayList a = new ArrayList();
Point p = new Point(10,20);
Point q;
// add the point p to the arraylist
a.add(0,p);
// now get the point out of the arraylist and assign it to q
// bleah!  I know perfectly well that I've only put points into a, but
// Java's type system says that the type of a.get(i) is Object.  So I
// need to put in a cast and have it do a runtime check.
q = (Point) a.get(0);
```

Java Arrays

Java arrays have always been an exception to this: rather than just being able to declare arrays of Object, one can declare an array of ints, Points, etc. (Note that this works both for primitive types and for objects.) Java checks that you are putting the right kind of object or type into the array, and knows that you're getting the same type out. For example:

```
Point[] a = new Point[10];
Point p = new Point(10,20);
Point q;
a[0] = p;
q = a[0]; // no cast needed - we know a[0] is a Point
```

Java regards an array of points as a subtype of array of objects ... which sounds reasonable, but alas isn't true! So Java's type checking for arrays is **unsound**! However, the Java designers were well aware of this problem, and so Java includes a runtime check (so you get an exception, not a crash). Consider:

```
import java.awt.Point;

class ArrayException {
    public static void main(String[] args) {
        String[] s;
        s = new String[10];
        s[0] = "hi there";
        test(s);
    }

    public static void test(Object[] a) {
        System.out.println("in test - before storing into a");
        a[1] = new Point(10,20);
        System.out.println("in test - after storing into a");
    }
}
```

This gives the following result:

```
in test - before storing into a
Exception in thread "main" java.lang.ArrayStoreException: java.awt.Point
at ArrayTest.test(ArrayTest.java:23)
at ArrayTest.main(ArrayTest.java:15)
```

So there is even a specialized kind of exception for this situation.

The technical term for the kind of type checking rule that Java uses for arrays (and that has this unsoundness problem) is *covariant typing*.

Java Generics

Integrating parametric polymorphism and inheritance was not easy, and for many years there was academic work in this area and various experimental languages prior to it becoming mainstream.

Code examples

OldLinkedListExample.java - Linked list example without generics

```
// CSE 341
// Example of collection code written without generics.
// Make a linked list, add a couple of integers to it,
// get an iterator over the list, and get the integers out.
// Since the type of the LinkedList and of the iterator
// doesn't use a parameterized type, we need to use a cast
// when extracting the integers.

import java.util.LinkedList;
import java.util.Iterator;

class OldLinkedListExample {

    public static void main(String[] args) {
        LinkedList myIntList = new LinkedList();
        myIntList.add(new Integer(3));
        myIntList.add(new Integer(4));
        Iterator i = myIntList.iterator();
        Integer x = (Integer) i.next();
        Integer y = (Integer) i.next();
        System.out.println(x);
        System.out.println(y);
    }
}
```

LinkedListExample.java - Linked list of integers with generics

```
// CSE 341
// Simple example of using Java generics.
// Make a linked list of integers, add a couple of integers to it,
// get an iterator over the list, and get the integers out.
// Notice that the LinkedList uses a parameterized type, as does the
// iterator.

import java.util.LinkedList;
import java.util.Iterator;

class LinkedListExample {

    public static void main(String[] args) {
        LinkedList<Integer> myIntList = new LinkedList<Integer>();
        myIntList.add(new Integer(3));
        myIntList.add(new Integer(4));
        Iterator<Integer> i = myIntList.iterator();
        Integer x = i.next();
        Integer y = i.next();
        System.out.println(x);
        System.out.println(y);
    }
}
```

ObjectLinkedListExample.java - Linked list of objects with generics

```
// CSE 341
// Linked list of Object (using generics)
// Make a linked list, add an integer and a point to it.  Then
// get an iterator over the list, and get the integer and point out.
// Use casts.

import java.util.LinkedList;
import java.util.Iterator;
import java.awt.Point;

class ObjectLinkedListExample {
    public static void main(String[] args) {
        LinkedList<Object> list = new LinkedList<Object>();
        list.add(new Integer(3));
        list.add(new Point(10,20));
```

```
        Iterator<Object> itr = list.iterator();
        // living dangerously -- we happen to know that the first thing
        // in the list is an integer and the second is a point.
        Integer i = (Integer) itr.next();
        Point p = (Point) itr.next();
        System.out.println(i);
        System.out.println(p);
    }
}
```

Wild1.java - simple example of using a wildcard

How can we write a method that works for any kind of linked list? Is there a type that is a supertype of any kind of linked list? We know that `LinkedList<Integer>` isn't a subtype of `LinkedList<Object>`, so `LinkedList<Object>` won't work. `LinkedList<?>` is what we're looking for.

```
// CSE 341
// Simple example of using a wildcard.
// Make a linked list of ?
// The only value we can add to it is null (since this is a legal value
// of any type).  We can get an iterator for the linked list, but we
// don't know what kind of types it produces.  (But they must be objects.)

import java.util.LinkedList;
import java.util.Iterator;

class Wild1 {

    public static void main(String[] args) {
        LinkedList<?> s = new LinkedList<Integer>();
        s.add(null);
        Iterator<?> x = s.iterator();
        Object o = x.next();
        System.out.println(o);

        // note that LinkedList<?> is NOT the same type as LinkedList<Object>
        // here are some statements that wouldn't compile:
        // s.add(new Integer(3));
        // Iterator<Object> x = s.iterator();
    }
}
```

Wild2.java - method with a wildcard

```
// CSE 341
// Example of using a wildcard.  Define a method that takes
// a linked list of ? as a parameter.

import java.util.LinkedList;
import java.util.Iterator;
import java.awt.Point;

class Wild2 {

    // note that we couldn't declare the parameter as LinkedList<Object> !

    public static void printAll(LinkedList<?> s) {
        for (Object e : s) {
            System.out.println(e);
        }
    }

    // another version using an iterator
    public static void printAll2(LinkedList<?> s) {
        Iterator<?> it = s.iterator();
        while (it.hasNext()) {
            System.out.println(it.next());
        }
    }

    public static void main(String[] args) {
        LinkedList<Integer> ilist = new LinkedList<Integer>();
        ilist.add(new Integer(3));
        ilist.add(new Integer(5));
        ilist.add(null);
        printAll(ilist);
        printAll2(ilist);

        LinkedList<Point> plist = new LinkedList<Point>();
        plist.add(new Point(10,20));
```

```
        printAll(plist);
        printAll2(plist);
    }
}
```

Wild3.java - method with a bounded wildcard

```
// CSE 341
// Example of using a bounded wildcard.  Define a method that finds
// the maximum width of a list of rectangular shapes

import java.util.LinkedList;
import java.util.Iterator;
import java.awt.Point;
import java.awt.geom.RectangularShape;
import java.awt.geom.Rectangle2D;
import java.awt.geom.Ellipse2D;


class Wild3 {

    // version 1 of maxWidth -- note that we need a cast to get
    // rectangular shapes out of the list
    public static double maxWidth1 (LinkedList<?> s) {
        double m = 0.0;
        for (Object e : s) {
            RectangularShape r = (RectangularShape) e;
            if (r.getWidth() > m) {
                m = r.getWidth();
            }
        }
        return m;
    }

    // version 2 of maxWidth, using a bounded wildcard
    public static double maxWidth2 (LinkedList<? extends RectangularShape> s) {
        double m = 0.0;
        for (RectangularShape e : s) {
            // no cast needed!
            RectangularShape r = e;
            if (r.getWidth() > m) {
                m = r.getWidth();
            }
        }
        return m;
    }

    public static void main(String[] args) {
        // demonstrate the use of the maxWidth method by calling it
        // with lists of different kinds of rectangular shapes
        // first make a linked list of rectangles
        LinkedList<Rectangle2D> rlist =
            new LinkedList<Rectangle2D>();
        rlist.add(new Rectangle2D.Double(0.0, 0.0, 50.0, 100.0));
        rlist.add(new Rectangle2D.Double(0.0, 0.0, 12.0, 18.0));
        double ans1 = maxWidth1(rlist);
        System.out.println(ans1);
        double ans2 = maxWidth2(rlist);
        System.out.println(ans2);
        // now do the same thing, but with ellipses
        LinkedList<Ellipse2D> elist =
            new LinkedList<Ellipse2D>();
        elist.add(new Ellipse2D.Double(0.0, 0.0, 20.0, 30.0));
        elist.add(new Ellipse2D.Double(0.0, 0.0, 24.0, 30.0));
        double ans1e = maxWidth1(elist);
        System.out.println(ans1e);
        double ans2e = maxWidth2(elist);
        System.out.println(ans2e);


    }
}
```

MyArray.java - an array-like class (written with generics)

```
// An array-like class (written with generics)

import java.util.Iterator;
import java.util.NoSuchElementException;
import java.awt.Point;

public class MyArray<E> {
    /* internal array to actually hold the data */
    private E [] internalArray;
```

```
    // the constructor
    public MyArray (int n) {
        internalArray = (E[]) new Object[n];
        // unfortunately this doesn't work due to current Java limitations:
        // internalArray = new E[n];
    }

    public int length() {
        return internalArray.length;
    }

    // get an element
    public E at(int i) {
        return internalArray[i];
    }

    // set an element
    public void set(int i, E value) {
        internalArray[i] = value;
    }

    /**
     * inner class for the iterator
     */
    class MyIterator implements Iterator<E> {
        int index;
        MyIterator() {
            index = 0;
        }
        public E next() {
            E temp;
            if (!hasNext())
                throw new NoSuchElementException("no next element available");
            temp = internalArray[index];
            index++;
            return temp;
        }
        public boolean hasNext() {
            return index < internalArray.length;
        }
        public void remove() {
            /* this is an optional operation - we don't support it */
            throw new UnsupportedOperationException("remove operation not supported");
        }
    }

    /**
     * return an iterator for this array
     */
    public Iterator<E> iterator() {
        return new MyIterator();
    }

    public static void main(String[] args) {
        MyArray<Integer> a = new MyArray<Integer>(2);
        a.set(0, new Integer(50));
        a.set(1, new Integer(100));
        Iterator<Integer> it = a.iterator();
        while(it.hasNext()) {
            System.out.println(it.next());
        }
    }
}
```

Conventions for Type Parameter Names in Java

By convention, type parameter names are single, uppercase letters. This makes it easier for programmers to spot the difference between a type variable and a class or interface name. (The compiler can tell this from the context, so this is just for the benefit of programmers.) According to the Java Generics tutorial the most commonly used type parameter names are:

- E - Element (used extensively by the Java Collections Framework)
- K - Key
- N - Number
- T - Type
- V - Value
- S,U,V etc. - 2nd, 3rd, 4th types