Generic Programming

Amit Shabtay

The Problem

- Assume we have a nice **`Stack`** implementation.
  - Our stack receives only `Objects`
- The problem:
  - We need different stacks for **`int`**, **`String`** and **`Method`**.
  - We don't want the problems of:

```
Stack.add("clearly not a method");
…
Method m = (Method)stack.pop();
```

Solution (?)

- Let's create **`IntStack`** , **`StringStack`** and **`MethodStack`**.
- All of them will rely on the original **`Stack`** as internal implementation.
- Are there any problems?
  - A lot of code duplication
  - It's hard to add new types

Another Problem

- We want to use a swap function between two **`int`**s.

```
void swap(int& a, int&b) {
   int temp = a; a = b; b = temp;
}
```

- What about swap(double&, double&) and swap(Method&, Method&) ?

The Actual Solution

- Generic programming.
  (The ability to have type parameters on your type)
- Write the code once, and worry about the type at compile time
  - The code is suitable to all types
  - Easy to add types later
  - No code duplication
  - Demands from types

So How Do We Do It?

```
swap<Class T>(T& a, T& b) {
  T temp = a;
  a = b;
  b = temp;
}
```

- Looks simple?

Java 1.4.2 vs. 1.5 and Autoboxing

-
  ```
  ArrayList list = new ArrayList(); //1.4.2
  list.add(0, new Integer(42));
  int total = ((Integer)list.get(0)).intValue();
  ```

-
  ```
  ArrayList<Integer> list = new ArrayList<Integer>();//1.5
  list.add(0, new Integer(42));
  int total = list.get(0).intValue();
  ```

-
  ```
  ArrayList<Integer> list =
                new ArrayList<Integer>(); //1.5 auto-boxing
  list.add(0, 42);
  int total = list.get(0);
  ```

Uses

- Containers
  - list
  - set
  - vector
  - map
- Algorithms
  - sort
  - search
  - copy

C++ Templates

- The most known use of generic programming
- STL – Standard Template Library
  - Containers
    - vector, set, hash_map
  - Algorithms
    - for_each, swap, binary_search, min, max

What about Java?

- Until now Java had large collection set
  - Set, List, Map, Iterator, and more
  - sort(), search(), fill(), copy(), max(), min()
- One major problem – the collections are not type safe
  - No problem to do

```
Map.put("key", "4");
Integer i = (Integer)map.get("key");
```

Java Generics

- Added as one of the new features of Java 1.5 ("Tiger")
- Done in the compiler only
  - Converts `String s = vector<String>.get(3)` to `String s = (String)vector.get(3)`

How to Use Generics ?

```
List<Integer> myIntList = new LinkedList<Integer>();

myIntList.add(new Integer(0));

Integer x = myIntList.iterator().next();
```

And What About the Collection ?

```
public interface List<E> {
  void add(E x);
  Iterator<E> iterator();
}
public interface Iterator<E> {
  E next();
  boolean hasNext();
}
```

Subtyping

```
List<String> ls =
  new ArrayList<String>();
List<Object> lo = ls; //Compiler will not permit
lo.add(new Object());
//Attempts to assign an Object to a String!
String s = ls.get(0);
```

Subtyping (cont.)

- Foo is subtype of Bar if:
  - Foo extends Bar
  - Foo implements Bar
- C is a generic container C<E>
- Results that C<Foo> is not subtype of C<Bar>

Generic Algorithms (1)

- How to print entire Collection?
- Do we have to use `Collection<Object>` ?
- Use wildcard

```
void printCollection(Collection<?> c){
  for (Object e : c) {
     System.out.println(e);
  }
}
```

Generic Algorithms (2)

- What happens when we want to use specific method?

```
public void
drawAll(List<Shape> shapes) {
  for (Shape s: shapes) {
     s.draw(this);
  }
}
```

- What about subtyping?
  - List<Circle>

Generic Algorithms (3)

- The solution

```
public void
drawAll(List<? extends Shape> shapes)
  {…} //Called bounded wildcard.
```

More About Wildcards

```
Collection<?> c = new ArrayList<String>();
c.add(new Object());
```

```
public void addRectangle(List<? extends
   Shape> shapes) {
   shapes.add(0, new Rectangle());
}
```

Super vs. Extends

- The syntax **? super T** denotes an unknown type that is a supertype of T.
- It is the dual of the **? extends T** to denote an unknown type that is a subtype of T.

Collection<Object> *vs* Collection<?> *vs* Collection

- Collection<Object> is a collection of *heterogeneous* instances of potentially no common type
- Collection<?> is a collection of *homogeneous* instances of some common types – we just don't know what that common type is
- Collection is a *raw type* – we should avoid it

Many More Features

- Java Generics are one of the important language features of Java 1.5
- More information in http://java.sun.com/developer/technicalArticles/J2SE/generics/
- J2SE 5.0 in a Nutshell http://java.sun.com/developer/technicalArticles/releases/j2se15/

Java Generics Summary

- Java Generics use a technique known as *type erasure* which is the process of translating or rewriting code that uses generics into non-generic code
- all information between angle brackets is erased.

C# Generics

- Very similar to Java
- public struct Point<T>
  { public T X; public T Y; }
- Point<int> point;
  point.X = 1; point.Y = 2;
- See http://msdn.microsoft.com/
  for more information

Java Generics *vs.* C++ Templates vs. C#

- Java borrowed C++ syntax and made it mean something very different
- Type erasure *vs.* code generation in C++.
- Evaluated in compile time in java and c++, run time in c#