Generic Programming

David Rabinowitz

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.
- 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?

Uses

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

C++ Templates

- Most famous use of generic programming
- STL – Standard Template Library
  - Containers
    - vector, set, hash_map
  - Algorithms
    - for_each, swap, binary_search, min, max
- Very high performance library
  - Compile time, Inlining, Specializations, …

What about Java?

- Until 1.5, Java had a large collection set
  - Set, List, Map, Iterator, and more
  - sort(), search(), fill(), copy(), max(), min()
- The collections were not type safe
  - No problem to do:

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


Java Generics

- JSR 14, add in Java 1.5 ("Tiger")
- Done in the compiler only
  - Converts:

```
String s = vector<String>.get(3)
```

    to:

```
String s = (String)vector.get(3)
```

- "Forward Compatible" with earlier JDKs
- For objects only (not simple types)

Using Java Generics

```
List<Integer> myIntList =
    new LinkedList<Integer>();
myIntList.add(new Integer(0));
Integer x =
    myIntList.iterator().next();
```


Generic Collections

```
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;
lo.add(new Object());
// attempts to assign an Object
// to a String!
String s = ls.get(0);
```


Subtyping (cont.)

- `Foo` is subtype of `Bar`
  - `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 a 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)
  {…}
```


More About Wildcards

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

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

C# Generics

- Were added to .NET 2.0
- Generics designed in advance
  - Visible to reflection, bytecode manipulation
- Very similar to Java:

```
public struct Point<T> {
   public T X; public T Y; }
Point<int> point;
point.X = 1; point.Y = 2;
```

- More at: http://msdn.microsoft.com/

C++ versus Java/C# Implementation

- The C++ Approach to wildcards
  - Use any required method/operator
  - The compiler verifies that all methods exist
  - For example, STL assumes default constructor
- More flexible, yet undocumented
  - Weird compiler errors
  - Templates only compiled when instantiated

C++ Template Instantiation

- Multiple executable code segments may be compiled from one template
  - Stack<int>, Stack<char>, Stack<Person*>
  - This doesn't happen in C#/Java
- Specialization may define optimizations

```
template<class T> class Vector {…}
template<> Vector<void*> { …}
template<class T> class Vector<T*> : private
 Vector<void*> { …}
```


Summary

- Generics are a required feature
  - Containers, Reusable algorithms
- Different Approaches
  - C++ - Performance first, non-OO
  - Java – Safety first, Forward Compatible
  - C# - Safety first, integrate with OOP