The UNIVERSITY of VERMONT

Benjamin Fenster
CS-100 (Damon)
3 May 2003

Notes – Parameterized Types

I. "Considered Harmful"
   a. Common phrase in CS now.
   b. It was first introduced int he discussion of structured programming.
      i. Nobody would write unstructured code anymore.
      ii. Called "Goto Considered Harmful"
   c. Object Oriented programming was the next big revolution
      i. Called "Switch Considered Harmful"
      ii. Used to have huge switch statements to handle the stuff that's now handled by objects.
      iii. Now code is grouped by class, not functionality.
   d. Most switch statements are replaced by dispatches now (which method gets called).
   e. Huge improvements in code
      i. Fewer errors
      ii. Lower maintenance costs
   f. Current Problem
      i. Runtime errors
      ii. Bigger code means more complicated logic.
      iii. More complicated logic means more subtle bugs.
      iv. Buffer Overflow
         1. Most common problem
         2. C/C++ makes it a security problem.
         3. Java throws an exception
         4. Both are problems for the user.
      v. Static Checking
         1. Try to find as many problems at compile time as possible.
         2. Type checking is the most common.
         3. Reduces errors significantly.
      vi. Common problem
         1. Downcasting from super to subclass, but to the wrong subclass
         2. Can't downcast to the rwrong type!
         3. Consider using Iterator
            a. Even if you only ever insert Robots, it's not looking for that – it's just looking for Objects.
            b. Thus, when you take something out you can only guarantee that it's an object.
            c. If you assume it's a Robot and it's not, you'll have problems.

II. Parameterized Types
   a. A type that takes some parameter telling it about the type of data it uses.
   b. Array is a parameterized type
   c. Instead o just Vector, you'd have a Vector<Robot>
   d. The power is that no new classes are defined, but you can specialize an otherwise general behavior.
   e. Then the class takes a parameter that can be used anywhere a class name (or object) would be.
   f. Pretty easy to do.
   g. Java does not support this yet. The Tiger release (1.5) should support it.
   h. (Also called Generics)
   i. Will work in November
   j. Restrictions
      i. Can't use the type parameter (EType) in anything static.
      ii. Static methods (etc), by definition, are shared by everybody. They don't get a copy of the parameter because it could vary.

iii. Cannot instantiate an EType since you don't know at compile-time what type it represents.
iv. There are other, more subtle issues with casting that may be fixed in the future.
v. Must be a class, not a primitive.

k. Benefits in the Decorator Pattern
i. You don't lose the functionality of the decorated component.
ii. It needs some new syntax though.
iii. `public ScrollingPane<c extends JComponent>`
iv. That keeps from making a scrolling pane of Fish or something weird – the parameter you use must be a subclass of JCcomponent.

l. Parameterization exists only at compile time.
i. The compiler guarantees that there won't be runtime violations.
ii. For runtime, it then inserts old-fashioned downcasts wherever they're needed.
iii. It's safe now though, because it's been checked.
iv. The performance implications of that downcasting are still there.

III. Other Tiger Features
a. Add enumerations
i. Still have however many distinct values for the type, but can add fields.
ii. Add new information as part of any of them.

b. foreach loops
i. for (string s: myNames)
ii. Uses Iterator()

c. Autoboxing
i. The Integer class is called a box for the int type
ii. Whenever you try to put an int into an Object now, it's boxed automatically

d. Importing Statics
i. More minor
ii. Can say import static java.lang.math.*
iii. Statics won't have to be prefixed with class names anymore.
iv. Works well with enumerations

IV. C++ Templates
a. Essentially parameterized types, but not quite.
b. Simple cases look a lot like Java
c. `template <class T> class myClass {…}`
d. Can include *any* types, including primitives.
e. Very different implementation
i. Creates a new class for each type based on the parameter
ii. Makes for *immense* code at runtime
iii. Common to have thousands of stack classes (or whatever), all from the same template.

f. It's not really parameterized types, it's a scheme to generate new types based on a pattern: a *template*, ya might say.