

## Class 24: Java Generics

**Held:** Friday, March 16, 2007

**Summary:** Today we consider Java's *generics*. Introduced in Java 1.5, generics permit you to parameterize class and interface declarations.

### Related Pages:

- EBoard.

### Due

- Mid-Semester Exam.

### Notes:

- Thanks for your patience on Wednesday. Eldest son came in first, so we're off to state in a month.
- Have a great break!

### Overview:

- Limitations to Java's Strong Type Checking.
- A (Partial) Solution: Java Generics.
- Restricting Parameter Classes.
- Subtleties.
- Behind the Scenes.

## Limitations to Java's Strong Type Checking

- One of Java's key "features" is that it does relatively strong compile-time type checking to limit the number of silly errors programmers make.
  - Yes, I know that we've also read people, like Paul Graham, who believe that compile-time type checking burdens the programmer. Let's accept this model for now.
- Unfortunately, as you've observed, in Java 1.4 (the version of Java most of you learned), the type checking does not extend to a number of common situations. For example,
  - You can't use type checking to ensure that a `Vector` (or any similar collection) is homogenous.
  - You can't use type checking to ensure that a `Comparator` can be applied to the values in a list (e.g., for `smallest` or `sorting`)
- This lack also requires us to write more verbose code than we'd like. For example, even if we know that that `Vector stuff` only contains `String` values, we still need to cast any value coming out of it.

```

public String smallestString(Vector stuff)
{
    String guess = (String) stuff.get(0);
    int len = stuff.size();
    for (int i = 1; i < len; i++) {
        String candidate = (String) stuff.get(i);
        if (guess.compareTo(candidate) > 0)
            guess = candidate;
    } // for
    return guess;
} // smallestString(Vector)

```

- The lack also means that we can accidentally call `smallestString` with an incorrect vector.

```

Vector vec = new Vector();
vec.add(new BigInteger(23));
System.out.println(smallestString(vec));

```

## A (Partial) Solution: Java Generics

- Java 1.5 introduced a partial solution to this problem, called generics. Generics permit you to add type parameters to class, interface, and method declarations.
- For classes and interfaces, you can add a list of type variables, surrounded by angle brackets, to the class name.
  - Tradition is that we name types with singleton capital letters.
- You can then use those type variables within the class definition.
- For example, you might define linear structures of certain kinds of values with

```

public interface LinearStructure<T>
{
    public void put(T val);
    public T get();
    public boolean isEmpty();
} // interface LinearStructure<T>

```

- The folks at Sun have conveniently parameterized all of the collection types (and more).
- When referring to a homogeneous linear structure, you fill in the type.

```

Vector<String> names = new Vector<String>();

```

- We can now rewrite our `smallestString` as

```

public String smallestString(Vector<String> stuff)
{
    String guess = stuff.get(0);
    int len = stuff.size();
    for (int i = 1; i < len; i++) {
        String candidate = stuff.get(i);
        if (guess.compareTo(candidate) > 0)
            guess = candidate;
    } // for
    return guess;
} // smallestString(Vector<String>)

```

- Method can also take type parameters. For methods, you precede the return type with any type parameters in angle brackets.

## Restricting Parameter Classes

- Consider the problem of representing a point generically, so that the client can specify the type of the X and Y coordinates. We might begin with

```
public class Point<T>
{
    T x;
    T y;
    ...
} // class Point<T>
```

- However, it might make sense to restrict the parameter types. For example, generic points might only have X and Y coordinates that are numbers.
- For such restrictions, we use `extends Type`.
- For example,

```
public class Point<T extends Number>
```

## Subtleties

- Subclassing can be difficult with generics. Suppose B is a subclass of A. Which of the following assignments should we be able to do, and why or why not?

```
A a = ...;
B b = ...;
Vector<A> va = ...;
Vector<B> vb = ...;
a = b; // Valid?
b = a; // Valid?
va = vb; // Valid?
vb = va; // Valid?
```

- Only `a = b` is acceptable.
- Why isn't `va = vb` acceptable? Because when we try to call `va.put(...)`, the type checker only knows that `va` contains A's. However, for type safety, we need to ensure that we can only add B's.
- This issue carries over into arrays. Hence, it is nearly impossible to declare an array of a parameter type.
- You will find that some of your `extends` clauses are more complicated than you'd expect.
- We'll rewrite `smallestString` generically to see those complications.

## Behind the Scenes

- The designers of Java 1.5 decided not to change the underlying VM. That means that the Java 1.5 VM has no direct support for generics (since the Java 1.4 VM lacked such support).
- Then how do we get generics? Behind the scenes, every generic uses `Object` (or the specified superclass/interface) for each type parameter.
- The compiler's type checker makes sure that the typing is valid.
- Type coercion operators may also be inserted.