Java Generics vs. C++ Templates

Michael R. Head

Grid Computing Research Laboratory
Binghamton University

March 26, 2007
Outline

1. Introduction
   - Generic Programming in General
   - Generic Programming in C++
   - Generic Programming in Java

2. Architectural Differences
   - Type Erasure
   - Parameterized Type Bounds
   - Metaprogramming
Outline

1. **Introduction**
   - Generic Programming in General
     - Generic Programming in C++
     - Generic Programming in Java

2. **Architectural Differences**
   - Type Erasure
   - Parameterized Type Bounds
   - Metaprogramming
What is Generic Programming?

- Programming without named types
- “Programming with concepts” – David Musser
Why Program Generically?

- Type-safe containers without specialized coding
  - `std::vector<std::string>`
- Generic algorithms
  - Iteration, `std::max()`
- Achieve higher performance than with other mechanisms (C++ only)
  - vs. function pointers
  - vs. object hierarchies
- Template metaprogramming (C++ only)
Outline

1. Introduction
   - Generic Programming in General
   - Generic Programming in C++
   - Generic Programming in Java

2. Architectural Differences
   - Type Erasure
   - Parameterized Type Bounds
   - Metaprogramming
Client Template Code in C++

```cpp
#include <iostream>
#include <vector>
#include <string>
using namespace std;
int main()
{
    vector<string> strings;
    strings.push_back("Hello");
    strings.push_back("world");

    vector<string>::const_iterator it;
    for (it = strings.begin(); it != strings.end(); it++)
    {
        cout << *it << endl;
    }
}
```

Michael R. Head
Java Generics vs. C++ Templates
Features of C++ Templates

- Implemented in the compiler
  - No runtime overhead
  - Requires template source to be in headers
  - Latent typing means template instantiator does no type checking

- Glorified macro facility
  - “Macros done right”/“Macros that look like classes”

- Can use template arguments for both classes and straight functions

- Template specialization
  - Specific implementation of a templated type or method

- Pattern matching and text replacement
  - Declarative model (like Prolog)
Outline

1. Introduction
   - Generic Programming in General
   - Generic Programming in C++
   - Generic Programming in Java

2. Architectural Differences
   - Type Erasure
   - Parameterized Type Bounds
   - Metaprogramming
import java.util.List;
import java.util.ArrayList;
public class Hello {
    public static void main(String[] args) {
        List<String> strings = new LinkedList<String>();
        strings.add("Hello");
        strings.add("world");

        for(String s: strings) {
            System.out.println(s);
        }
    }
}
Features of Java Generics

- Based on Pizza project’s GJ compiler
- Implemented in the compiler
  - Does not require source of generic type to be available
  - Compiled code can theoretically run on older JVMs
- Applies to classes and methods within classes
- Type parameter bounds "<X extends Widget>"
- Mostly used to eliminate downcasts
  - “End-run around the type checker”
Outline

1. Introduction
   - Generic Programming in General
   - Generic Programming in C++
   - Generic Programming in Java

2. Architectural Differences
   - Type Erasure
   - Parameterized Type Bounds
   - Metaprogramming
Some kind of retro 80s band?

Actually designed to maintain backward compatibility with pre-Java5 code

- Also reduces runtime memory load from multiple specializations of a generic type

After statically analyzing to ensure type safety, the compiler removes all references to the parameters of generic type

Internally, type parameter $X$ is converted to $\text{Object}$ (or its bound) in the byte code
Type Erasure

- Some kind of retro 80s band?
- Actually designed to maintain backward compatibility with pre-Java5 code
  - Also reduces runtime memory load from multiple specializations of a generic types
- After statically analyzing to ensure type safety, the compiler removes all references to the parameters of generic type
- Internally, type parameter $X$ is converted to `Object` (or its bound) in the byte code
Type Erasure

- Some kind of retro 80s band?
- Actually designed to maintain backward compatibility with pre-Java5 code
  - Also reduces runtime memory load from multiple specializations of a generic types
- After statically analyzing to ensure type safety, the compiler removes all references to the parameters of generic type
- Internally, type parameter \( X \) is converted to \texttt{Object} (or its bound) in the byte code
What is the erased type of each of these? *Hint: The first one is Object*

```java
<T>
<T extends Number>
<T extends Comparable<T>>
<T extends Cloneable & Comparable<T>>
<T extends Object & Comparable<T>>
<S, T extends S>
```
“The type erasure of its leftmost bound, or type $\text{Object}$ if no bound was specified.”

\[
\begin{align*}
<T> & : \text{Object} \\
<T \text{ extends } \text{Number}> & : \text{Number} \\
<T \text{ extends } \text{Comparable}<T>> & : \text{Comparable} \\
<T \text{ extends } \text{Cloneable \& Comparable}<T>> & : \text{Cloneable} \\
<T \text{ extends } \text{Object \& Comparable}<T>> & : \text{Object} \\
<S, T \text{ extends } S> & : \text{Object, Object}
\end{align*}
\]
Implications

- No new classes are created when instantiating a generic type.
- Instantiating a generic class has no runtime overhead over using an equivalent class that uses `Object` rather than `X`.
  - Indeed, the two classes would compile to almost identical byte code.
  - Also, no potential performance benefit from using a specially compiled version (say for `ints` to eliminate autoboxing).
- You can’t do `X.class.newInstance()` or anything related to the actual class of the type parameter.
- Parameterized Exception classes are right out.
- Leads to so-called “raw types” `List` for `List<T>`.
Outline

1. Introduction
   - Generic Programming in General
   - Generic Programming in C++
   - Generic Programming in Java

2. Architectural Differences
   - Type Erasure
   - Parameterized Type Bounds
   - Metaprogramming
Java allows the parameterized type to be specified in terms of other types

- This can mean `<X extends Widget>`
- Or `<X extends Comparable<T>>`
- Or `<X extends Comparable<T> & Serializable>`

Allows the use of extra or more specialized methods on `X`

Also allows “wildcard” bounds `<? extends Foo>` or `<? super Bar>`

- Useful for `Foo<List<?>>`
- Why is “super” not legal for normal type bounds?
#FAQ102 from Angelika Langer’s Generics FAQ

class X0 <T extends int> { ... }
class X1 <T extends Object[]> { ... }
class X2 <T extends Number> { ... }
class X3 <T extends String> { ... }
class X4 <T extends Runnable> { ... }
class X5 <T extends Thread.State> { ... }
class X6 <T extends List> { ... }
class X7 <T extends List<String>> { ... }
class X8 <T extends List<? extends Number>> { ... }
class X9 <T extends Comparable<? super Number>> { ... }
class X10<T extends Map.Entry<?>,???> { ... }

Michael R. Head
Java Generics vs. C++ Templates
#FAQ102 from Angelika Langer’s Generics FAQ

class X0 <T extends int> { ... }
class X1 <T extends Object[]> { ... }

class X2 <T extends Number> { ... }
class X3 <T extends String> { ... }
class X4 <T extends Runnable> { ... }
class X5 <T extends Thread.State> { ... }
class X6 <T extends List> { ... }
class X7 <T extends List<String>> { ... }
class X8 <T extends List<? extends Number>> { ... }
class X9 <T extends Comparable<? super Number>> { ... }
class X10<T extends Map.Entry<?,?>> { ... }
Implications

- System is actually quite complex to implement in the compiler
- Compiler checks type before instantiation: implementing type-safe generics
  - C++ templates know nothing about types ("concepts" notwithstanding) leading to confusing compile errors
  - C++ lets you do anything you like with the type parameter (latent typing), without a bound in Java, only Object’s method are allowed
- Leads to types like `Enum<E extends Enum<E>>`
Outline

1. Introduction
   - Generic Programming in General
   - Generic Programming in C++
   - Generic Programming in Java

2. Architectural Differences
   - Type Erasure
   - Parameterized Type Bounds
   - Metaprogramming
C++ provides the ability to provide multiple implementations of the same template signature. The different implementations may reference each other recursively. The compiler implements “template specialization” to resolve this at compile time. The resolution process is recursive and Turing-complete, so this effectively lets the programmer “program the compiler.”

Java Generics offers none of this.
Implications

- “Abusing your compiler for extremely early binding”
- Can do really powerful computations at compile time
  - Compute constants such as $\pi$ to any precision required
  - Let the compiler generate lookup tables (for example, for base64 codecs)
  - Loop unrolling
- Extremely difficult to write, even harder to communicate to someone else
- Conflation with generic types makes both tasks slightly worse
  - Consider Lisp macros for metaprogramming
  - A more specialized generic typing facility would probably involve a type system
We’ve looked at some of the differences between Java Generics and C++ Templates

- Java Generics handles type checking during instantiation and generates byte-code equivalent to non-generic code
- C++ has “latent typing” and template metaprogramming and generates a new class for each instantiation
On Java Generics...
- Using and Programming Generics in J2SE 5.0
- Java Generics FAQs

On C++ Templates...
- C++ Templates FAQ
- A gentle introduction to Template Metaprogramming with C++
- C++ FAQ Lite: Templates

Other Comparisons...
- Bruce Eckel on Templates vs. Generics
- Generics in C#, Java, and C++ (A Conversation with Anders Hejlsberg)