A Comparison of Generics in Major Imperative Programming Languages

Joe Einertson

University of Minnesota, Morris

April 28, 2012

Languages Used

C++

- Started as “C with Classes”
- Compiled
- Efficiency was important design goal

Java and C♯

- Syntax similar to C
- Interpreted
- Compile to *bytecode* (Java) or *intermediate language* (C♯)
- Bytecode interpreted by a *virtual machine*

Generic Types

Generic types, commonly known as generics, are a form of *parametric polymorphism*

- Define data structure generically
- Use data structure in type-dependent way

Example (Generic data structures)

```
class List<T> { ... }
List<String> strList = ...
List<Car> carList = ...
```

Generic Types

Generic types support formal parameters

- Formal parameters are symbols which denote a type
- Instances replace formal parameters with actual parameters
- Only one type per instance
- Single, consistent implementation

Formal and Actual Parameters

In the following code, T is a formal parameter, and String and Car are actual parameters.

Example

```
class List<T> { ... }
List<String> strList = ...
List<Car> carList = ...
```

Outline

- Introduction
- Fundamentals of Generics
  - Advantages of Generics
  - Generics in Java, C++ and C♯
- Advanced Generic Features
- Implementation of Generics
- Generics Adoption
- Evaluating Claims About Generics
- Conclusion and References

Basic Advantages

Generics have many advantages to programmers

- Code reuse
- Code not bound to a single type
- Increase in type safety
- Easier to read and understand
- Lower cognitive load

Non-Generic Code

Non-generic code is long and convoluted

Example (Non-generic list of integers)

```
ArrayList list = new ArrayList();
... // We will retrieve our element in a bit
list.add(3);
Object element = list.get(0);
if (!(element instanceof Integer)) {
    throw new InvalidTypeException("Expected an
      Integer, but received a different type.");
}
Integer integerElement = (Integer) element;
```

Generic Java Code

Same functionality utilizing generics

Example (Generic list of integers)

```
ArrayList<Integer> list = new ArrayList<Integer>();
list.add(3);
... // We will retrieve our element in a bit
Integer element = list.get(0);
```

- Five lines of code eliminated
- ArrayList is *parametrized* over Integers

Basic C♯ Generics

C♯ syntax is very similar to Java

Example (Generic list of integers)

```
List<int> intList = new List<int>();
intList.Add(3);
int element = intList[0];
```

- C♯ List equivalent to Java ArrayList
- C♯ collections use array access syntax

Basic C++ Generics

C++ generic classes are called templates

Example (Generic list of integers)

```
list<int> intlist;
intlist.push_front(3);
int element = intlist.front();
```

- Uses list class from Standard Template Library

Primitive and Object Types

Java distinguishes between *primitive types* and *object types*

- Primitive types are low-level types, e.g., int, float, boolean
- Object types are everything else, e.g., String, ArrayList, Integer

What is the difference between int and Integer?

Boxing and Unboxing

Java only allows object types as generic parameters

- How can primitive types, e.g. ints, be placed in generic objects?

Java provides wrapper object types for primitive types

Example (Boxing and Unboxing)

```
int x = 3;
Integer y = new Integer(x);
int z = y.intValue();
```

Auto-boxing and auto-unboxing is performed by the compiler

Boxing and Unboxing

C♯ also performs auto-boxing and auto-unboxing, but it is hidden by syntax

- Object methods may be called on primitive types
- Primitive types may be type parameters of generic classes
- C♯ generates specific implementations of generic classes for each primitive type

C++ allows any type to be used in a generic object

- There is no practical distinction between primitive and object types, as related to generics

Why Are Type Constraints Needed?

Example (Generic PriorityList definition)

```
class PriorityList<T> {
    T getMaxPriority() { ... }
    T getMinPriority() { ... }
}
```

What if type T cannot be compared?
What if it can be compared multiple ways?

Java Type Constraints

Example (Generic Comparable interface)

```
interface Comparable<T> {
    int compareTo(T other);
}
```

With type constraints, we can rewrite PriorityList to require objects of type T to be Comparable

Example (PriorityQueue enforcing Comparable)

```
class PriorityList<T extends Comparable<T>> { ... }
```

C♯ Type Constraints

C♯ type constraints are very similar to Java, but with extra constraints

Example (C♯ Constraints)

```
class PriorityList<T> where T: Comparable<T>
class Bar<T, U>
    where T: struct
    where U: class, Comparable<T>
```

- struct requires primitive type
- class requires object type
- new() requires default constructor

C++ Type Constraints

C++ does not provide type constraints as a language feature

- Constraints can be emulated through tricks using function pointers
- Bjarne Stroustrup, designer of C++, advocates for these tricks
- Function pointers are ugly and beyond the scope of this talk

Instantiation in C++

C++ implements generics via instantiation

- Separate class generated for each distinct instantiation
- Results in duplication of machine code, but not source code
- Duplication is minimal in practice
- Chosen for efficiency

Example

Two instantiations of a class list, one for ints and one for chars, will result in the compiler producing two separate classes: one exclusive to ints and one exclusive to chars.

Type Erasure in Java

Java introduced generics in Java 1.5, but wanted to maintain backwards compatibility

- No changes to bytecode
- Allow developers to program generically

Type erasure was chosen as method of implementation

- At compilation time, type parameters are erased, leaving only the class name

Example

List<String> erases to List
List<Map<String, Integer>> erases to List

Reification in C♯

Microsoft was OK with breaking backwards compatibility

- C♯ bytecode rewritten, adding generics
- More flexible implementation
- Allows C♯ to perform *reflection* on generic type parameters

Claims About Generics

Many claims have been made about the effects of generics

- Reduce code duplication
- Reduce or eliminate runtime type errors
- Encourage and enable standardization and consistency
- Lower programmers' cognitive load

Are these claims substantiated by empirical studies?

Generics in Standard Libraries

Study conducted in 2005 rewrote Java Buffer library utilizing generics

- Pre-generics, 68% of code was duplicated somewhere else
- By adding generics, 40% of duplicate code was removed

Separate 2005 study refactored parts of major Java libraries, adding generics

- 91% of compiler warnings eliminated
- Nearly half the type casts removed

This shows potential benefits – if generics are used. So, are they?

Why Open Source Projects?

Open source software (OSS) projects have several advantages in evaluating generics adoption

- Code is freely available
- Nearly all OSS projects use version control
- Lack of central leadership means decisions about code are primarily made by the people writing the code

Analyzing Generics Adoption in Java OSS Projects

In 2011, Parnin et al conducted an in-depth study of generics adoption

- Analyzed top 20 most used Java OSS projects
- Nearly 550 million lines of code
- Millions of separate commits

Patterns in OSS Projects

Some patterns emerge in Parnin's analysis

- 8/20 projects contained more parametrized (generic) type declarations than raw (non-generic) type declarations
- 5 projects did not use any generic declarations at all
- Adoption rate seems unrelated to size of the code base
- Only 14% of developers created or modified generic code
- This number rises to 27% for the most active developers

Patterns in OSS Projects

Patterns also emerge in specific instantiations

- List<String> was most common instantiation in every project
- 25% of instantiations were of the type List<String> or Map<String, String>
- One-third of generic types were parametrized over a single type
- 80% were parametrized over fewer than 5 types

These patterns are interesting to note given C++'s use of instantiation

Claim 1: Reduced Code Duplication

To measure potential duplication, Parnin et al devised a formula to calculate potential number of clones

- Specific classes tailored to a single type, e.g. IntegerList, are referred to as *clones*
- Top ten genericized classes would have resulted in 4000 clones
- Of the remaining classes, 5.8 clones per class
- Generics did not supplant clones in every instance, or even a majority of instances
- Generics eliminated 107,000 lines of duplicated code from top 27 classes alone

Claim 2: Reduced Runtime Type Errors

It is difficult to measure the incidence of errors in any software project

- Parnin et al devised a formula to estimate errors as a function of duplicated lines and the number of revisions to those lines
- They cited a previous study indicating .01 errors per commit and .0074 errors per line of code
- Using these estimates, around 400 errors may have been eliminated

Claim 3: Standardization and Consistency

Very difficult to empirically measure standardization in a project

- Generics inherently standardize syntax
- Generics unify implementations of the same concept for different type parameters
- Non-generic code can be written many ways, with varying degrees of correctness

Claim 4: Lower Cognitive Load

Cognitive load can be inferred from code complexity

- Generic code is simpler to read and intentions are clear
- Developers no longer have to think about whether an item in a collection is the correct type as long as their generic declaration is correct
- Parnin et al found no correlation between generics adoption and a decrease in typecasts
- In one project, they found an *increase* in typecasts

Conclusions

It is obvious that no single language of implementation is perfect. That said, they all have benefits:

- Type parametrization
- Code simplification
- Increased type safety

Conclusions

We can say with relative certainty that generics

- Reduce code duplication
- Reduce runtime type errors

We also believe that generics

- Increase standardization
- Decrease cognitive load, although to what degree is unclear

Conclusions

To conclude,

- Generics are not a panacea
- Generics do increase type safety and reduce errors
- Generics simplify code and reduce duplication

Therefore, we believe generics are a beneficial addition to a language, and that even a generic system with some flaws is better than no generic system at all.

Acknowledgements

I would like to thank my faculty advisor Elena Machkasova for her tireless proofreading and revising. Additionally, I would like to thank my external reviewer Matthew Justin for his keen eye for even the smallest mistakes.

References

1 C. Parnin, C. Bird, and E. Murphy-Hill. Java generics adoption: how new features are introduced, championed, or ignored. 2011.

2 R. Garcia, J. Jarvi, A. Lumsdaine, J. G. Siek, and J. Willcock. A comparative study of language support for generic programming. 2003.

3 D. Ghosh. Generics in Java and C++: a comparative model. 2004.

4 A. Kennedy and S. Don. Design and implementation of generics for the .NET Common Language Runtime. 2001.