A Comparison of Generic Template Support: Ada, C++, C#, and Java™

Benjamin M. Brosgol

AdaCore; 104 Fifth Ave., New York NY 10011 USA
brosgol@adacore.com

Abstract. *Generics* (also known as *templates*) have become a standard feature of modern programming languages, offering parameterization by data types and possibly other entities. Generics support efficient type-safe container data structures, general-purpose algorithms, and other reusable components. However, the approaches in different languages vary widely in syntax, semantics, and usage. This paper summarizes the design of generics in Ada, C++, C#, and Java and compares them with respect to expressiveness, implementation model / run-time efficiency, and interaction with object-oriented programming and other features.

Keywords: generic programming, templates, Ada, C++, C#, Java.

1 Introduction

One of the fundamental software engineering principles is *abstraction*: the ability to generalize from specific instances and identify a common pattern that can be parameterized. Run-time parameterization (subroutine definition and invocation) has been a staple of programming languages since the earliest days of computing, and translation-time parameterization (macro definition and instantiation) followed soon thereafter. Macros, however, are rather unstructured, since instantiation simply involves text substitution and expansion. It was only in the 1970s, with the groundbreaking work on languages such as CLU [1] and Alphard [2], that researchers recognized that translation-time parameterization could be generalized and made more reliable and robust. Instead of a mechanism based on text substitution, the language could enforce syntactic and semantic checks on the content to be expanded, and allow parameters to be specified in terms of program entities such as types. A well-known example is a "container" data structure (list, stack, queue, etc.) that can be parameterized by element type. The resulting mechanism came to be known as *generics* or *templates*, and it is sometimes referred to more fancily as *parametric polymorphism*.

Generics resemble macros in that each instantiation replicates the template, either conceptually or actually, with formals replaced by actuals. However, generics are distinct from macros in at least two important ways.

- The generic template is checked for syntactic and semantic correctness; it is not simply raw source text that gets expanded. Thus all uses of generic formal parameters must be consistent with their declaration, or compilation will fail.

J. Real and T. Vardanega (Eds.): Ada-Europe 2010, LNCS 6106, pp. 222–237, 2010.

- Instantiations are checked to ensure that their arguments (actual generic parameters) match their corresponding formal parameters.

Generics offer several advantages.[1] An obvious benefit is *reuse*; without generics, the developer of a component has two main choices:

1. Use macros and a preprocessor. But text substitution and expansion are at the wrong semantic level: names in the generic template would be interpreted based on the scope of the instantiation rather than the scope of the generic's definition, contrary to standard block structure conventions.
2. For container-like data structures, define the component to take a general element type (`Object` in many object-oriented languages, `void*` in C) and apply casts when extracting elements. This approach adds run-time cost for type checking and means that errors are detected late.

This second point brings up another benefit of generics: *type safety*. With generics, a data structure such as a queue can be guaranteed to only contain elements of some specific type. It is not possible to insert an integer and then remove it and treat it as having some other type. A corollary benefit is thus *efficiency*: with an appropriate implementation model, type safety can be guaranteed at compile time. There should be no need for run-time checks to ensure that the data in the queue has the correct type.

Generics are heavily used in the definition of standard libraries. Examples are the I/O and containers packages in Ada, the Standard Template Library in C++, and the containers class libraries in C# and Java.

Different languages, however, take different approaches to realizing generics. The main distinctions stem from design decisions in several areas:

- *Expressiveness / Basic Semantics*
  - Which kinds of entities can be made generic? Does instantiation need to be explicit, or can it be implicit?
  - Which kinds of formal parameters are allowed for a generic entity, and what are the rules for matching a formal parameter by an actual parameter at an instantiation?
  - What establishes an instantiation's legality and how it may be used?
  - If a formal parameter to a generic is a type, how may it be used in the template? Can a formal type parameter be constrained so that the matching actual type parameter needs to supply specific operations?
  - Are recursive instantiations permitted?
- *Implementation Model*
  - Does each instantiation of a generic yield a separate expansion, or can different instantiations (in particular with different arguments) share common code?
  - Are there run-time costs associated with generics?
  - When are errors detected?

[1] There are also some drawbacks. Generics introduce syntactic and semantic complexity, and, if misused, may lead to "code bloat" in implementations that expand the template separately for each instantiation. They also generally require special tool support, for example in a debugger.

- How is separate compilation handled?

- *Feature Interactions*
  - How do generics interact with Object-Oriented Programming, overload resolution, name binding, and other language features?

The remainder of this paper will summarize how these issues are addressed in practice by four specific languages: Ada [3], C++ [4], C# [5], and Java [6]. These are not the only languages that support generics (others include Modula-3 and Delphi) but they are used heavily and illustrate the breadth of approaches.

Length constraints prevent inclusion of complete examples. Please refer to the companion web site [7] for full listings of examples in the four languages.

Earlier work comparing generic programming features in different languages includes [8], which omits Ada, and [9], which omits C#.

2 Ada

Among the languages covered in this paper, Ada is the only one that supported generics from its earliest release. Ada's model is significantly different from the other languages, principally because Ada has distinct features for modularization and data typing. A class in C++, C#, or Java would be modeled in Ada by a (tagged) type declared in a package.

2.1 Expressiveness / Basic Semantics

• *Which kinds of entities can be made generic? Does instantiation need to be explicit, or can it be implicit?*

Ada allows packages and subprograms to be generic. Types themselves are not generic, but a type declared in a generic package has some of the properties of generic types in other languages.

Instantiation must be explicit and has the effect of declaring a non-generic package or subprogram. Implicit instantiation was rejected because of semantic complexity and readability concerns [10]. Also, implicit instantiation is most useful for generic types (classes) but in Ada types are not generic.

• *Which kinds of formal parameters are allowed for a generic entity, and what are the rules for matching a formal parameter by an actual parameter at an instantiation?*

Ada has a rich facility for generic parameterization. Formal generic parameters may be types, subprograms (including operator symbols), objects (values or variables), and instances of generic packages. Formal types come in a variety of flavors, reflecting the fact that different categories of types (integer types, access types, ...) have different operations available. Instantiations may make use of default values and named associations. Unlike C++ (which requires constants), Ada allows run-time evaluable expressions in actual parameters for formal objects.

If a unit needs to be parameterized by type, the designer chooses the appropriate category (for example, `digits <>` for floating-point types) and can provide additional formal subprogram parameters for operations that are not automatically available based on the type's category but that are needed within the generic template.

• *What establishes an instantiation's legality and how it may be used?*

An instantiation's legality may depend on the full generic specification rather than just whether the parameters match. However, embodying a principle that has come to be known as the *contract model*, an instantiation's legality does not depend on the generic body. In general, Ada prohibits usage of features in a generic body if such features could make some instantiations illegal.

An instance of a generic unit can be used in the same manner as a non-generic unit. Unavailability problems with an instance's subprograms, which are possible in C++ (see below), do not arise in Ada.

• *If a formal parameter to a generic is a type, how may it be used in the template? Can a formal type parameter be constrained so that the matching actual type parameter needs to supply specific operations?*

A formal type may only be used in ways that are allowed for every type in its category, unless the required operations are provided as additional generic formal parameters.

• *Are recursive instantiations permitted?*

No. The error will be detected at compile time.

2.2 Implementation Model

• *Does each instantiation of a generic yield a separate expansion, or can different instantiations share common code?*

The Ada rules allow both expansion-based and code-sharing implementations. During Ada's early years, code sharing (for instantiations of the same generic with different actual parameters) was especially useful because generics were the only way to pass subprograms as parameters. The subject attracted considerable attention (e.g., [11]), and the code sharing optimization was provided by several compilers.

Code sharing in general, however, is very difficult to implement. With the provision of subprograms as run-time parameters in Ada 95 (and also the emergence of hardware with greater code space capacity), this space optimization became less critical, and today nearly all Ada compilers use the expansion model.

• *Are there run-time costs associated with generics?*

Generic instantiation in Ada is purely a compile-time activity, but in some cases run-time checks may be generated (for example to ensure that an expression supplied as an actual parameter satisfies the subtype constraints of a formal generic object parameter).

• *When are errors detected?*

All static semantic errors in generic templates, and all mismatch errors at instantiations, are caught at compile time. However – and this is an issue also in C++, C#, and Java – it is possible for an instantiation to be legal, but to generate an illegal overloading of some subprogram `P` that the compiler only detects at points of `P`'s invocation. One example (with analogs in C++ and C#) is a generic package that declares subprograms `P(X1:T1)` and `P(X2:T2)` where `T1` and `T2` are formal generic parameters, and the generic is instantiated with the same type as actual parameter for both.

As noted above, run-time checking may be needed for formal object parameters. Also, a generic body must be elaborated before any instantiation, or else an "access before elaboration" exception will be raised at run time.

• *How is separate compilation handled?*

Generic units may be separately compiled, as may their instantiations. Ada's requirement for explicit instantiation has the benefit of the programmer controlling code sharing. A simple conceptual model is that each instantiation expands to a separate body of code. A linker optimization can eliminate those subprograms that are not called.

2.3 Feature Interactions

• *How do generics interact with Object-Oriented Programming, overload resolution, and other language features?*

Object-Oriented Programming. The main interaction with OOP concerns the extension of tagged types declared in generic packages. Ada addresses this issue through generic child packages: a type declared in a generic child can extend a tagged type declared in the generic parent package.

Any of the generic packages in the hierarchy may contain a dynamically bound invocation for a primitive subprogram of one of the tagged types, and it will work with the standard effect in the instantiated package.

With non-generic OOP (a tagged type in a non-generic package specification), a derived type may appear in either a child or a client ("with"ing) unit. If a tagged type is declared in a generic package, then the derived type may only be declared in a child.

Overload Resolution / Name Binding. Generics themselves are not overloadable, but instances of a generic subprogram may be overloaded.

An important issue is how to resolve a (statically bound) subprogram invocation in the generic template when some of the actual parameters of the subprogram are of a generic formal type `T`. When the subprogram can be resolved as one of the other generic formal parameters, then that is the interpretation. Otherwise, if it is an operation defined for `T` based on `T`'s type category (e.g., "=" for a formal private type) then the built-in (primitive) version of the operation is used. If neither of these resolutions is possible, the subprogram invocation is illegal.

3 C++

Although not in the initial version of C++, templates were always considered essential for parameterization of container classes. A minimal facility was added in 1990, based on experience with prototype implementations of early design proposals, and several enhancements have appeared subsequently. The key goals for C++'s template facility were "notational convenience, run-time efficiency, and type safety. The main constraints were portability and reasonably efficient compilation and linkage" [12, p. 339].

3.1 Expressiveness / Basic Semantics

• *Which kinds of entities can be made generic? Does instantiation need to be explicit, or can it be implicit?*

Function and class (or struct) templates are allowed. Instantiation may be either explicit or implicit. Explicit instantiation must be managed carefully, since multiple

instantiations of the same template with the same arguments can cause linker errors due to duplicate definitions.

When a function template's implicit instantiation is invoked, the type arguments to the instantiation may be omitted if they can be deduced from the run-time parameters:

```
template <typename T> void foo(T t) ...
...
foo(100); // Equivalent to foo<int>(100)
```

Member functions of templates must be defined as function templates themselves, and they are only instantiated if called. Implicitly instantiating a class template does not automatically instantiate all member function definitions.

• *Which kinds of formal parameters are allowed for a generic entity, and what are the rules for matching a formal parameter by an actual parameter at an instantiation?*

Templates may have type parameters (identified by the keyword `typename` or `class`), nontype parameters, and template parameters. At an instantiation, a template's type parameter is matched by a type argument, a nontype parameter is matched by a constant value, and a template parameter is matched by a template argument.

Any type (built-in, class, struct, etc.), except class types defined within functions, may be supplied as an argument for a type parameter.

The type of a template's nontype parameter must be an integral or enumeration type, a pointer type (for example a pointer-to-function type), or a reference type. The matching argument must be a constant value of the corresponding type. Since instantiating a class template with an argument produces a specific type, restricting nontype arguments to constants ensures that the compiler can determine when two instantiations denote the same type. However, a side effect is that string literals are not allowed as arguments for templates, since different occurrences of the same literal may be at different addresses.

A template's template parameter `TT` is matched by a template argument whose parameters correspond to (i.e. have the same types as) the parameters of `TT`. Such parameters facilitate template composition.

• *What establishes an instantiation's legality and how it may be used?*

A class template instantiation's legality depends only on whether the arguments match the formal parameters. However, instantiating a member function from the class instance will fail if the member function invokes a function that is not available for the argument type. For example, consider a class template with a type parameter `T` and a member template function `Foo()` that uses some operation for `T`. Unless the template takes an additional parameter (a pointer-to-function for the required operation), instantiating `Foo` with an argument type `TT` will fail if `TT` does not have the needed operation.

The absence of a generic "contract model" makes software maintenance difficult, since there is no way for the author of a template to know whether an implementation change will cause errors when existing instantiations are recompiled.

• *If a formal parameter to a generic is a type, how may it be used in the template? Can a formal type parameter be constrained so that the matching actual type parameter needs to supply specific operations?*

A template's type parameter may be used as a regular type inside the template. C++ has no syntax for establishing constraints on a type parameter; if additional operations are needed this will either be part of the informal contract of the template (documented in comments) or else supplied as explicit pointer-to-function parameters to the template.

• *Are recursive instantiations permitted?*

Yes, for example using non-type parameters such as `int`s. Templates in C++ in effect provide a compile-time functional language, with iteration realized through recursive instantiation (template specialization [12, p. 373] terminates the recursion). This style, referred to as *metaprogramming*, can be used to obtain highly-optimized performance (see for example [13, p. 314]) but can be rather opaque.

3.2 Implementation Model

• *Does each instantiation of a generic yield a separate expansion, or can different instantiations share common code?*

C++'s design philosophy encourages sharing of separate instantiations with the same arguments, whether within the same translation unit or across translation units, but this is not consistent in practice. Because of the lack of a "contract model", sharing instantiations with different arguments is more difficult than with Ada.

• *Are there run-time costs associated with generics?*

Template expansion and argument/parameter matching are purely compile-time (or link-time) activities. Run-time efficiency was a major goal for the C++ design, and techniques such as template specialization can be used to optimize performance (e.g. by specifying pass-by-value parameters rather than `const` references for particular types).

• *When are errors detected?*

Errors, especially instantiations with types that do not provide needed operations, are detected during compilation (or linking), at the point of instantiation or function invocation. Thus, unlike Ada, C#, and Java, such errors are not detected during compilation of the template definition.

• *How is separate compilation handled?*

It is natural for the template declaration (the interface, without function bodies) to be placed in a header file (`.hpp`). An issue is whether the template body (the "template definition") should be in the same header file, similar to an inline function, or in an implementation file (`.cpp`).

Placing the template definition in the header file, referred to as the "inclusion model" in [13], is common practice. A program that needs the template can simply `#include` the header file. However, template definitions typically require further `#include`s, and the closure of all of these `#include`s can result in considerable code bloat and/or excessive compilation time. Also, the definition needs to be bracketed by `#ifndef`/`#define` and `#endif` brackets to avoid duplication of definition in `#include`ing translation units.

An alternative is to place the template declaration in a `.hpp` file and the definition in a `.cpp` file (the "separation model"). The translation unit using the template `#include`s the `.hpp` file. This raises the issue of how the implementation locates

the corresponding `.cpp` file. One technique is to mark the declaration(s) and definition with the `export` keyword. However, this mechanism is not uniformly supported by current compilers, and in general the inclusion model is used. Precompiled headers can help reduce compilation cost.

3.3 Feature Interactions

• *How do generics interact with Object-Oriented Programming, overload resolution, and other language features?*

Object-Oriented Programming. Template classes can form inheritance hierarchies, in the same way as regular classes. A template class may derive from non-template classes and/or from template classes. Here's an example adapted from [12, p. 346]:

```
template <typename T> class Vector  /* ... */ ;
template <typename T> class OrderedVector : Vector<T>  /* ... */ ;
```

However, if `TC<T>` is a template class with a class parameter, and `T2` derives from `T1`, it does not necessarily follow that `TC<T2>` derives from `TC<T1>`. I.e., template instantiations are not *covariant*. The rationale is straightforward. If a `List<Subclass>` reference `ref` were assignable to a `List<Superclass>` variable `vbl`, then through `vbl` it would be possible to add `Superclass` objects to `ref`.

Overload Resolution / Name Binding. Function templates may be overloaded, and the names used inside template definitions may likewise be overloaded.

The name binding rules are complex [12, pp. 368ff], since three different contexts are involved: the template definition, the argument type declaration, and the template use / instantiation. The complications are especially intense for the names that depend on a template argument.

Other Features. There are a number of "corner case" pathologies. For example, an instantiation `SomeTemplate<int, Vector<int>>` is illegal because the ">>" at the end is parsed as a shift operator. The programmer must insert a space between the brackets, e.g. `SomeTemplate<int, Vector<int> >`.

4 C#

Generics were added to C# (and to the .NET infrastructure) in Version 2, with the goals of type safety, time efficiency (no boxing /unboxing or casts) and space efficiency (avoidance of code duplication).

4.1 Expressiveness / Basic Semantics

• *Which kinds of entities can be made generic? Does instantiation need to be explicit, or can it be implicit?*

C# allows declarations of generic types (classes, interfaces, structs, and delegates) and generic methods. It does not permit generic properties, events, indexers, operators, constructors, or finalizers, but these entities may be declared within generic types.

Instantiation is implicit, but a "using alias" directive can simulate an explicit instantiation (for example to create a shorthand name).

• *Which kinds of formal parameters are allowed for a generic entity, and what are the rules for matching a formal parameter by an actual parameter at an instantiation?*

The only kind of formal parameter that a generic entity can take is a type.

Unless otherwise constrained (see below), instantiation with both value types (such as the predefined type `int`) and reference types is permitted. However, instantiation with pointer types is not allowed.

• *What establishes an instantiation's legality and how it may be used?*

Although an instantiation's legality is based only on the matching of parameters (including satisfying constraints on the formal), ambiguous overloadings may result that are only detected at method invocation. This is similar to Ada's semantics, described above.

• *If a formal parameter to a generic is a type, how may it be used in the template? Can a formal type parameter be constrained so that the matching actual type parameter needs to supply specific operations?*

Type parameters may be constrained so that they can only be instantiated with types having particular characteristics. Type constraints include whether a type is a reference type or a value type, whether it derives from a specific class or implements a particular interface (or interfaces), and whether it supplies an accessible no-arg constructor.

If the generic requires a type parameter to have a specific method available, then the type parameter can be constrained to derive from some specific type that includes that method, or to implement an interface that declares that method.

Constraints that are not enforceable at compile time can be checked at run time through the use of reflection in a static constructor. An example (checking that a type parameter is an enum) appears in [5, p. 390].

• *Are recursive instantiations permitted?*

Since C# generics can only take type parameters, "metaprogramming" in the style of C++ is not supported. However, the definition of a generic type or method may contain instantiations of the same generic. The caching / code sharing mechanism prevents unbounded expansion.

4.2 Implementation Model

• *Does each instantiation of a generic yield a separate expansion, or can different instantiations share common code?*

The language rules were designed to allow code sharing. All instantiations with reference types can share a single code body. Each instantiation with a specific value type has its own expansion (instantiations with the same value type share the same body).

• *Are there run-time costs associated with generics?*

With the standard C# implementation model, instantiation itself is performed at run time, through the "just-in-time" (*JIT*) compiler, when control reaches the point of instantiation. Instantiations are cached to perform code sharing as described above.

Alternatively, instantiations can be precompiled to native code through Microsoft's NGEN utility, also with code sharing. NGEN will process all instantiations, since it does not know which ones are required at run time and which ones are not.

With either approach there is no run-time overhead (boxing, casts, etc).

• *When are errors detected?*

Errors are detected during compilation, potentially in three different contexts: an illegal construct in the generic itself, an illegal instantiation (non-matching argument), or usage of an ambiguous construct resulting from an instantiation

• *How is separate compilation handled?*

No special rules. The compiler produces special intermediate language for generics, and also outputs generic-specific information in the metadata.

4.3 Feature Interactions

• *How do generics interact with Object-Oriented Programming, overload resolution, and other language features?*

Object-Oriented Programming. Generic classes can participate in inheritance hierarchies. As with C++, generics are not covariant.

Overload Resolution / Name Binding. Method overloading within a generic type definition results in some complex semantics. For example:

```
class Gen<T>{
  Foo(T t){...}
  Foo(int i){...}
}
...
Gen<int> x = new Gen<int>();
x.Foo(0); // Which Foo?
```

The C# rules (similar to C++) dictate that the "more specific" method (here the one with an explicit `int` parameter) is selected. Such preferencing rules can cause maintenance problems. If the version with an explicit `int` parameter was introduced during program maintenance, the invocation of `Foo` will silently change meaning when recompiled. In the analogous examples in Ada and Java, the call on `Foo` would be ambiguous.

Other Features. Since information about generics is retained in the compiled assembly, run-time reflection and introspection are permitted.

5 Java

Generics were introduced in Java 5 to support type-safe collections and general-purpose methods in an upwards-compatible fashion (i.e., not requiring any changes to Java Virtual Machine implementations, and easing the migration path for developers who had previously used non-generic solutions). Upwards compatibility comes at a price, however; as observed in [14, p. 1], generics are "sometimes controversial" and have left "a few rough edges". They also introduce a run-time cost (casts implicitly inserted by the compiler). These issues result from Java's model of "type erasure" where all traces of genericity in the source disappear in the generated byte codes.

5.1 Expressiveness / Basic Semantics

• *Which kinds of entities can be made generic? Does instantiation need to be explicit, or can it be implicit?*

Classes, interfaces, methods, and constructors can be defined as generic.

Instantiation is implicit, similarly to C++ and C#. Instantiating a generic class or interface is referred to as *type invocation* and results in a *parameterized type*.

A unique aspect of Java's generic model is that instantiations of the same generic class `Gen<T>` with different type arguments, say `Gen<T1>` and `Gen<T2>`, do not produce different classes. Instead they yield the same *raw type* `Gen` as described below. Consequently all of the instantiations share the same static fields. This effect is generally undesirable (e.g. preventing the common style of maintaining an instantiation-specific counter to keep track of the number of constructed objects).

• *Which kinds of formal parameters are allowed for a generic entity, and what are the rules for matching a formal parameter by an actual parameter at an instantiation?*

The only formal parameter permitted is a type (Java refers to this as a *type variable*.) A matching actual parameter is a reference type that satisfies all specified constraints (see below).

Although primitive types are not allowed, Java's "boxing" rules allow uses of generic instantiations with primitive values; the compiler will implicitly allocate an actual object of the relevant type (for example an `Integer` for an `int` value).

• *What establishes an instantiation's legality and how it may be used?*

Instantiation legality is based only on parameter matching. Problems with unavailability of operations, as in C++, do not arise. However, as with the other languages, instantiations may yield ambiguous overloadings that make certain method invocations illegal.

• *If a formal parameter to a generic is a type, how may it be used in the template? Can a formal type parameter be constrained so that the matching actual type parameter needs to supply specific operations?*

Java allows the specification of type constraints (*bounds*) to ensure that the needed operations on the formal type parameter are available. This is accomplished by specifying that a formal type extends some other type (possibly one of the other generic formal parameters) and/or implements any number of interfaces.

• *Are recursive instantiations permitted?*

Analogous to C#, Java does not support metaprogramming in the style of C++, but a generic class may contain an instantiation of itself.

5.2 Implementation Model

• *Does each instantiation of a generic yield a separate expansion, or can different instantiations share common code?*

Java's type erasure model ensures that all instantiations share the same code. The erasure of a generic type yields the *raw* type, which is obtained by replacing each formal parameter (type variable) by `Object` if it does not have an "extends" bound, and by its first bound otherwise.

• *Are there run-time costs associated with generics?*

The compiler generates implicit casts, since the erasure of a generic class loses instantiation-specific information. Thus:

```
Stack<String> ss = new Stack<String>();
ss.Push("Hello");      // compiled as ss.Push((String)"Hello");
String s = ss.Pop();   // compiled as String s = (String)(ss.Pop());
```

Also, since type parameters must be reference types, using primitive values such as `int`s incurs boxing/unboxing overhead.

• *When are errors detected?*

Static semantic errors in the definition of the generic are caught at compile time. These errors include invocation of operations that are not consistent with the formal parameter's constraints, and declaring potentially ambiguous method overloadings (see below). Some ambiguities are detected (at compile time) when an attempt is made to invoke an ambiguous method from an instantiation.

For upwards compatibility, Java permits converting from a parameterized type to its underlying non-generic (*raw*) type. However, type errors might then remain undetected until run time (for an example see [15, p.165]).

• *How is separate compilation handled?*

Compilation of a generic produces a (non-generic) raw type. Run-time reflection will not recover the generic-related information.

5.3 Feature Interactions

• *How do generics interact with Object-Oriented Programming, overload resolution, and other language features?*

Object-Oriented Programming. Similarly to C++ and C#, Java allows generic types to extend other types (either non-generic or generic), subject to the usual rule that a class can extend only one superclass but can implement an arbitrary number of interfaces.

Also like C++ and C#, generic classes are not covariant: if `C2` extends `C1`, and `Gen<T>` is a generic class, then it does not follow that `Gen<C2>` extends `Gen<C1>`. Thus a `Gen<C2>` reference cannot be assigned to a `Gen<C1>` variable, nor may it be passed as an argument to a method taking a `Gen<C1>` parameter. Sometimes, however, these operations make sense and should be allowed. For example, consider the following hypothetical generic collection class:

```
public class Mob<T>{
   public addAll( Mob<T> m){...} // Add m's elements to this
}
...
Mob<C1> m1 = new Mob<C1>();
Mob<C2> m2 = new Mob<C2>();
m1.addAll(m2);  // Illegal
}
```

The method invocation is illegal since `m2` is not a `Mob<C1>`. In order to allow such invocations, the declaration of `addAll`'s parameter must be changed so that it accepts

not only a Mob<T> but also a Mob<U> for any class U in the type hierarchy rooted at T. The form is called a *wildcard* and has the following syntax:

```
public class Mob<T>{
   public addAll( Mob<? extends T> m){...}
// Add m's elements to this
}
```

The example above – `m1.addAll(m2);` – will now succeed. As a shorthand, the common case `<? extends Object>` can be abbreviated as `<?>`.

Wildcards are also useful in the other direction. If we are "consuming" elements from a collection Mob<C2>, then the target may a Mob<C1> where C1 is either C2 or any of its superclasses. Java has a wildcard form for this purpose also, with the syntax `<? super T>`.

A wildcard may be used for a formal parameter and/or result type of a method in a generic class or interface, and also as the type of a declared variable.

Overload Resolution / Name Binding. Outside of generics, Java's overloading rules forbid declaring two methods with the same name and the same signature. Java's erasure model requires strengthening this rule for generics; for example the following is illegal:

```
public class Gen<T1, T2>{
  public void Foo(T1 : t1){...}
  public void Foo(T2 : t2){...}
}
```

Although the two declarations of Foo have different signatures, they have the same "erasure signatures" (obtained by replacing the type name by its corresponding bound), and declaring two methods in the same generic type with the same erasure signature is prohibited. (Here it is as though each declaration of Foo had a parameter of type Object.)

The example above would be legal in Ada, C++, and C#, as would an instantiation with the same type argument for both parameters. The illegality would be an attempt to invoke Foo from such an instantiation.

Java's rules are somewhat analogous to Ada's model of "assuming the worst" in a generic declaration, although motivated by different factors. For Ada the issue is enforcement of the "contract model", and for Java the issue is consistency with type erasure.

Other Features. Java's erasure model results in a number of restrictions. A formal type parameter of a generic type cannot be used as the type of a static field, nor may it be referenced within a static method or static constructor. It is also prohibited in constructors for objects or arrays. Other restrictions are listed in [16, p. 268].

6 Conclusions

Each of the language designs has approached generics in a unique fashion, yielding both advantages and drawbacks. Tables 1 through 3 summarize the comparison.

Table 1. Generics Comparison Summary, Part 1

	Generic entities	*Generic parameters*	*Instantiation*	*Constraints allowed on formal types?*
Ada	• Packages • Subprograms	• Types • Subprograms • Objects • Package instances	• Explicit	• Yes
C++	• Classes, structs • Functions	• Types • Constant values • Templates	• Implicit	• No
C#	• Types • Methods	• Types	• Implicit	• Yes
Java	• Classes, interfaces • Methods	• Reference types	• Implicit	• Yes

Table 2. Generics Comparison Summary, Part 2

	Contract model?	*Recursive intantiation / "metaprogramming"?*	*Code sharing? (same generic, different arguments)*
Ada	• Yes	• Neither	• Implementation dependent, but generally no
C++	• No	• Both	• Implementation dependent, but generally no
C#	• No	• Recursion: yes • Metaprogramming: no	• Yes for reference types, no for value types
Java	• Yes	• Recursion: yes • Metaprogramming: no	• Yes for all types

Table 3. Generics Comparison Summary, Part 3

	Avoidance of run-time overhead	*Error detection*	*Instantiation covariance*
Ada	• Yes	• Compile time, generally early (at point of generic declaration or instantiation) but some only at uses of instantiation	• Not applicable
C++	• Yes	• Compile or link time, late (at uses of instantiation)	• No
C#	• Yes (after JIT compilation of instantiation)	• Compile time, generally early but some only at uses of instantiation	• No
Java	• No (compiler generates implicit casts)	• Mixed. Some caught at compile time, analogous to Ada; others at run time (class cast exceptions)	• No, but may be modeled through wildcards

In brief:

- *Ada* provides the most precise and explicit interface for a generic unit, and the requirement for explicit instantiations makes the programmer's intent clear. But the separation of the class concept into two features (package and tagged type) complicates the interaction between generics and OOP (generic hierarchies), and the number and variety of generic formal types can be confusing.
- *C++* provides considerable expressive power and flexibility, and allows the programmer to fine-tune performance. But the interface for a generic template is broad and implicit, allowing errors in a template definition to remain undetected until triggered by an instantiation. And the semantics for name resolution in templates is complex.
- *C#* allows expression of constraints on a type's operations, and it offers a straightforward approach for code sharing. But arguments to a generic are restricted to types (no constants, for example), and the preferencing rules for name resolution in generics are complicated.
- *Java* provides a solution that is upwards compatible with earlier versions of Java and the JVM, and its "erasure" model ensures code sharing. But performance is compromised (implicit casts are generated to ensure type safety, requiring run-time checks), and the semantics can be counterintuitive (e.g., different instantiations sharing static data).

Given the tradeoffs among different goals (generality, early error detection, run-time efficiency, code space compactness, upwards compatibility) it is not surprising that none of the languages is uniformly superior to the others. However, languages evolve over time, and new languages will undoubtedly be designed. Perhaps the "lessons learned" from user and implementer experience will lead to new insights and approaches that bring reuse with fewer tears.

Acknowledgements

The author is grateful to Bob Duff, Ed Falis, and Ed Schonberg from AdaCore for their review of an early version of the paper, and to the anonymous referees for their commments and suggestions.

References

1. Liskov, B., et al.: CLU Reference Manual. Springer, Heidelberg (1983)
2. Wulf, W.A., London, R.L., Shaw, M.: Abstraction and verification in Alphard: Introduction to language and methodology. USC Information Sciences Institute (1976)
3. ISO/IEC JTC1/SC 22/WG 9. Ada Reference Manual – ISO/IEC 8652:2007(E) with Technical Corrigendum 1 and Amendment 1 – Language and Standard Libraries (2007)
4. International Organization for Standardization. ISO/IEC 14882:2003, The C++ Standard Incorporating Technical Corrigendum 1 (2003)
5. Ecma International. C# Language Specification – ECMA-334, 4th edn. (June 2006)
6. Gosling, J., Joy, B., Steele, G., Bracha, G.: The Java Language Specification, 3rd edn. Addison-Wesley, Reading (2005)
7. Brosgol, B.M.: Companion Examples: A Comparison of Generic Template Support (November 2009), `http://www1.adacore.com/ brosgol/ae2010/examples.html`
8. Garcia, R., et al.: A comparative study of language support for generic programming. In: Proc. OOPSLA 2003. ACM, New York (2003)
9. Khalifa, A.A.: Generics: Ada 95 vs C++ vs Java 1.5. Master's thesis, Univ. of Jyväskylä, Finland (2005), `https://jyx.jyu.fi/dspace/handle/123456789/12351?show=full`
10. Ichbiah, J.D., Barnes, J.G.P., Firth, R.J., Woodger, M.: Rationale for the Design of the Ada Programming Language (1983)
11. Bray, G.: Implementation implications of Ada generics. ACM SIGAda Ada Letters III(2) (1983)
12. Stroustrup, B.: The Design and Evolution of C++. Addison-Wesley, Reading (1995)
13. Vandevoorde, D., Josuttis, N.M.: C++ Templates: The Complete Guide. Addison-Wesley, Reading (2007)
14. Naftalin, M., Wadler, P.: Java Generics and Collections. O'Reilly, Sebastopol (2007)
15. Flanagan, D.: Java in a Nutshell, 5th edn. O'Reilly, Sebastopol (2005)
16. Arnold, K., Gosling, J., Holmes, D.: The Java Programming Language, 4th edn. Addison-Wesley, Reading (2006)