

What are the differences between Generics in C# and Java... and Templates in C++? [closed]

Asked 16 years, 3 months ago Modified 11 years, 2 months ago

Viewed 62k times

203

votes



Closed. This question needs to be more [focused](#). It is not currently accepting answers.

Closed 11 years ago.



Locked. This question and its answers are [locked](#) because the question is off-topic but has historical significance. It is not currently accepting new answers or interactions.

I mostly use Java and generics are relatively new. I keep reading that Java made the wrong decision or that .NET has better implementations etc. etc.

So, what are the main differences between C++, C#, Java in generics? Pros/cons of each?

c#

java

c++

generics

templates

Share

edited Apr 11, 2009 at 23:25



Shog9

159k ● 36 ● 235 ● 240

asked Aug 28, 2008 at 5:08



pek

18k ● 28 ● 88 ● 99

Comments disabled on deleted / locked posts / reviews

13 Answers

Sorted by:

Highest score (default)



362

votes



I'll add my voice to the noise and take a stab at making things clear:

C# Generics allow you to declare something like this.

```
List<Person> foo = new List<Person>();
```

and then the compiler will prevent you from putting things that aren't `Person` into the list.

Behind the scenes the C# compiler is just putting

`List<Person>` into the .NET dll file, but at runtime the JIT compiler goes and builds a new set of code, as if you had written a special list class just for containing people - something like `ListOfPerson`.

The benefit of this is that it makes it really fast. There's no casting or any other stuff, and because the dll contains the information that this is a List of `Person`, other code that

looks at it later on using reflection can tell that it contains `Person` objects (so you get intellisense and so on).

The downside of this is that old C# 1.0 and 1.1 code (before they added generics) doesn't understand these new `List<something>`, so you have to manually convert things back to plain old `List` to interoperate with them. This is not that big of a problem, because C# 2.0 binary code is not backwards compatible. The only time this will ever happen is if you're upgrading some old C# 1.0/1.1 code to C# 2.0

Java Generics allow you to declare something like this.

```
ArrayList<Person> foo = new ArrayList<Person>();
```

On the surface it looks the same, and it sort-of is. The compiler will also prevent you from putting things that aren't `Person` into the list.

The difference is what happens behind the scenes. Unlike C#, Java does not go and build a special `ListOfPerson` - it just uses the plain old `ArrayList` which has always been in Java. When you get things out of the array, the usual `Person p = (Person)foo.get(1);` casting-dance still has to be done. The compiler is saving you the key-presses, but the speed hit/casting is still incurred just like it always was.

When people mention "Type Erasure" this is what they're

talking about. The compiler inserts the casts for you, and then 'erases' the fact that it's meant to be a list of `Person` not just `Object`

The benefit of this approach is that old code which doesn't understand generics doesn't have to care. It's still dealing with the same old `ArrayList` as it always has. This is more important in the java world because they wanted to support compiling code using Java 5 with generics, and having it run on old 1.4 or previous JVM's, which microsoft deliberately decided not to bother with.

The downside is the speed hit I mentioned previously, and also because there is no `ListOfPerson` pseudo-class or anything like that going into the .class files, code that looks at it later on (with reflection, or if you pull it out of another collection where it's been converted into `Object` or so on) can't tell in any way that it's meant to be a list containing only `Person` and not just any other array list.

C++ Templates allow you to declare something like this

```
std::list<Person>* foo = new std::list<Person>();
```

It looks like C# and Java generics, and it will do what you think it should do, but behind the scenes different things are happening.

It has the most in common with C# generics in that it builds special `pseudo-classes` rather than just throwing the type information away like java does, but it's a whole different kettle of fish.

Both C# and Java produce output which is designed for virtual machines. If you write some code which has a `Person` class in it, in both cases some information about a `Person` class will go into the .dll or .class file, and the JVM/CLR will do stuff with this.

C++ produces raw x86 binary code. Everything is *not* an object, and there's no underlying virtual machine which needs to know about a `Person` class. There's no boxing or unboxing, and functions don't have to belong to classes, or indeed anything.

Because of this, the C++ compiler places no restrictions on what you can do with templates - basically any code you could write manually, you can get templates to write for you.

The most obvious example is adding things:

In C# and Java, the generics system needs to know what methods are available for a class, and it needs to pass this down to the virtual machine. The only way to tell it this is by either hard-coding the actual class in, or using interfaces. For example:

```
string addNames<T>( T first, T second ) { return first.  
}
```

That code won't compile in C# or Java, because it doesn't know that the type `T` actually provides a method called `Name()`. You have to tell it - in C# like this:

```
interface IHasName{ string Name(); };  
string addNames<T>( T first, T second ) where T : IHasName
```

And then you have to make sure the things you pass to `addNames` implement the `IHasName` interface and so on. The java syntax is different (`<T extends IHasName>`), but it suffers from the same problems.

The 'classic' case for this problem is trying to write a function which does this

```
string addNames<T>( T first, T second ) { return first
```

You can't actually write this code because there are no ways to declare an interface with the `+` method in it. You fail.

C++ suffers from none of these problems. The compiler doesn't care about passing types down to any VM's - if both your objects have a `.Name()` function, it will compile. If they don't, it won't. Simple.

So, there you have it :-)

Share

edited Jun 20, 2020 at 9:12



Community Bot

1 • 1

answered Aug 28, 2008 at 9:50



Orion Edwards

123k ● 66 ● 245 ● 339

-
- 8 The generated pseudoclasses for references types in C# share the same implementation so you won't get exactly `ListOfPeople`. Check out blogs.msdn.com/ericlippert/archive/2009/07/30/... – [Piotr Czapla](#) Aug 13, 2009 at 6:28
-
- 4 No, you can *not* compile Java 5 code using generics, and have it run on old 1.4 VMs (at least the Sun JDK does not implement this. Some 3rd party tools do.) What you can do is use previously-compiled 1.4 JARs from 1.5/1.6 code. – [finnw](#) Feb 3, 2010 at 17:13
-
- 4 I object to the statement that you can't write `int` `addNames<T>(T first, T second) { return first + second; }` in C#. The generic type can be restricted to a class instead of an interface, and there is a way to declare a class with the `+` operator in it. – [Mashmagar](#) Jul 20, 2011 at 13:24
-
- 4 @AlexanderMalakhov it's non idiomatic on purpose. The point was not to educate about the idioms of C++, but to illustrate how the same-looking piece of code is handled differently by each language. This goal would have been harder to achieve the more different the code looks – [Orion Edwards](#) Jun 4, 2012 at 21:15
-
- 3 @phresnel I agree in principle, but if I'd written that snippet in idiomatic C++, it would be far less understandable to C#/Java developers, and therefore (I believe) would have done a worse job at explaining the difference. Let's agree to disagree on this one :-) – [Orion Edwards](#) Mar 25, 2013 at 20:23
-

61

votes



C++ rarely uses the “generics” terminology. Instead, the word “templates” is used and is more accurate. Templates describes one *technique* to achieve a generic design.

C++ templates is very different from what both C# and Java implement for two main reasons. The first reason is that C++ templates don't only allow compile-time type arguments but also compile-time const-value arguments: templates can be given as integers or even function signatures. This means that you can do some quite funky stuff at compile time, e.g. calculations:

```
template <unsigned int N>
struct product {
    static unsigned int const VALUE = N * product<N - 1>
};

template <>
struct product<1> {
    static unsigned int const VALUE = 1;
};

// Usage:
unsigned int const p5 = product<5>::VALUE;
```

This code also uses the other distinguished feature of C++ templates, namely template specialization. The code defines one class template, `product` that has one value argument. It also defines a specialization for that template that is used whenever the argument evaluates to 1. This allows me to define a recursion over template definitions. I believe that this was first discovered by [Andrei Alexandrescu](#).

Template specialization is important for C++ because it allows for structural differences in data structures.

Templates as a whole is a means of unifying an interface across types. However, although this is desirable, all types cannot be treated equally inside the implementation. C++ templates takes this into account. This is very much the same difference that OOP makes between interface and implementation with the overriding of virtual methods.

C++ templates are essential for its algorithmic programming paradigm. For example, almost all algorithms for containers are defined as functions that accept the container type as a template type and treat them uniformly. Actually, that's not quite right: C++ doesn't work on containers but rather on *ranges* that are defined by two iterators, pointing to the beginning and behind the end of the container. Thus, the whole content is circumscribed by the iterators: `begin <= elements < end`.

Using iterators instead of containers is useful because it allows to operate on parts of a container instead of on the whole.

Another distinguishing feature of C++ is the possibility of *partial specialization* for class templates. This is somewhat related to pattern matching on arguments in Haskell and other functional languages. For example, let's consider a class that stores elements:

```
template <typename T>  
class Store { ... }; // (1)
```

This works for any element type. But let's say that we can store pointers more efficiently than other types by applying some special trick. We can do this by *partially* specializing for all pointer types:

```
template <typename T>
class Store<T*> { ... }; // (2)
```

Now, whenever we instance a container template for one type, the appropriate definition is used:

```
Store<int> x; // Uses (1)
Store<int*> y; // Uses (2)
Store<string**> z; // Uses (2), with T = string*.
```

Share

edited Oct 19, 2011 at 1:29



Mankarse

40.6k ● 11 ● 103 ● 144

answered Aug 28, 2008 at 7:11



Konrad Rudolph

545k ● 139 ● 956 ● 1.2k

I've sometimes wished the generics feature in .net could allow things besides types to be used as keys. If value-type arrays were a part of the Framework (I'm surprised they're not, in a way, given the need to interact with older APIs that embed fixed-sized arrays within structures), it would be useful to declare a class which contained a few individual items and then a value-type array whose size was a generic parameter. As it is, the closest one can come is to have a class object which holds the individual items and then also holds a reference to a

separate object holding the array. – [supercat](#) Jan 6, 2013 at 18:20

@supercat If you interact with legacy API the idea is to use marshalling (which can be annotated via attributes). The CLR doesn't have fixed-size arrays anyway so having non-type template arguments would be of no help here.

– [Konrad Rudolph](#) Jan 6, 2013 at 19:27

I guess what I find puzzling is that it would seem like having fixed-sized value-type arrays shouldn't have been hard, and it would have allowed many data types to be marshalled by reference rather than by value. While marshal-by-value can be useful in cases that genuinely can't be handled any other way, I would regard marshal-by-ref to be superior in nearly all cases where it's usable, so allowing such cases to include structs with fixed-sized arrays would have seemed a useful feature.

– [supercat](#) Jan 6, 2013 at 20:35

BTW, another situation where non-type generic parameters would be useful would be with data types that represent dimensioned quantities. One could include dimensional information within instances that represent quantities, but having such information within a type would allow one to specify that a collection is supposed to hold objects representing a particular dimensioned unit. – [supercat](#) Jan 6, 2013 at 20:39

35

votes

Anders Hejlsberg himself described the differences here "[Generics in C#, Java, and C++](#)".



Share



answered Aug 28, 2008 at 5:14



jfs

16.7k ● 13 ● 63 ● 90

i really like that interview. it makes it clear for non-c# guys like me what's going on with c# generics. – [Johannes Schaub - litb](#)
Apr 12, 2009 at 1:38

18
votes



There are already a lot of good answers on *what* the differences are, so let me give a slightly different perspective and add the *why*.

As was already explained, the main difference is *type erasure*, i.e. the fact that the Java compiler erases the generic types and they don't end up in the generated bytecode. However, the question is: why would anyone do that? It doesn't make sense! Or does it?

Well, what's the alternative? If you don't implement generics in the language, where *do* you implement them? And the answer is: in the Virtual Machine. Which breaks backwards compatibility.

Type erasure, on the other hand, allows you to mix generic clients with non-generic libraries. In other words: code that was compiled on Java 5 can still be deployed to Java 1.4.

Microsoft, however, decided to break backwards compatibility for generics. *That's why* .NET Generics are "better" than Java Generics.

Of course, Sun aren't idiots or cowards. The reason why they "chickened out", was that Java was significantly older and more widespread than .NET when they introduced

generics. (They were introduced roughly at the same time in both worlds.) Breaking backwards compatibility would have been a huge pain.

Put yet another way: in Java, Generics are a part of the *Language* (which means they apply *only* to Java, not to other languages), in .NET they are part of the *Virtual Machine* (which means they apply to *all* languages, not just C# and Visual Basic.NET).

Compare this with .NET features like LINQ, lambda expressions, local variable type inference, anonymous types and expression trees: these are all *language* features. That's why there are subtle differences between VB.NET and C#: if those features were part of the VM, they would be the same in *all* languages. But the CLR hasn't changed: it's still the same in .NET 3.5 SP1 as it was in .NET 2.0. You can compile a C# program that uses LINQ with the .NET 3.5 compiler and still run it on .NET 2.0, provided that you don't use any .NET 3.5 libraries. That would *not* work with generics and .NET 1.1, but it *would* work with Java and Java 1.4.

Share

edited Aug 29, 2008 at 1:08

answered Aug 29, 2008 at 0:57



Jörg W Mittag

369k ● 79 ● 453 ● 661

-
- 3 LINQ is primarily a library feature (though C# and VB also added syntax sugar alongside it). Any language that targets the 2.0 CLR can gain full use of LINQ simply by loading the System.Core assembly. – [Richard Berg](#) Jul 30, 2009 at 20:53
-

Yeah, sorry, I should have been more clear wrt. LINQ. I was referring to the query syntax, not the monadic standard query operators, the LINQ extension methods or the IQueryable interface. Obviously, you can use those from any .NET language. – [Jörg W Mittag](#) Jul 30, 2009 at 21:26

- 1 I am thinking another option for Java. Even Oracle do not want to break backward compatibility, they can still make some compiler trick to avoid type information being erased. For example, `ArrayList<T>` can be emitted as a new internally named type with a (hidden) static `Class<T>` field. As long as the new version of generic lib has been deployed with the 1.5+ byte code, it will be able to run on 1.4- JVMs. – [Earth Engine](#) Oct 12, 2012 at 0:14
-

14 Follow-up to my previous posting.

votes



Templates are one of the main reasons why C++ fails so abysmally at intellisense, regardless of the IDE used.



Because of template specialization, the IDE can never be really sure if a given member exists or not. Consider:

```
template <typename T>
struct X {
    void foo() { }
};

template <>
struct X<int> { };
```

```
typedef int my_int_type;

X<my_int_type> a;
a. |
```

Now, the cursor is at the indicated position and it's damn hard for the IDE to say at that point if, and what, members `a` has. For other languages the parsing would be straightforward but for C++, quite a bit of evaluation is needed beforehand.

It gets worse. What if `my_int_type` were defined inside a class template as well? Now its type would depend on another type argument. And here, even compilers fail.

```
template <typename T>
struct Y {
    typedef T my_type;
};

X<Y<int>::my_type> b;
```

After a bit of thinking, a programmer would conclude that this code is the same as the above: `Y<int>::my_type` resolves to `int`, therefore `b` should be the same type as `a`, right?

Wrong. At the point where the compiler tries to resolve this statement, it doesn't actually know `Y<int>::my_type` yet! Therefore, it doesn't know that this is a type. It could be something else, e.g. a member function or a field. This might give rise to ambiguities (though not in the present

case), therefore the compiler fails. We have to tell it explicitly that we refer to a type name:

```
X<typename Y<int>::my_type> b;
```

Now, the code compiles. To see how ambiguities arise from this situation, consider the following code:

```
Y<int>::my_type(123);
```

This code statement is perfectly valid and tells C++ to execute the function call to `Y<int>::my_type`. However, if `my_type` is not a function but rather a type, this statement would still be valid and perform a special cast (the function-style cast) which is often a constructor invocation. The compiler can't tell which we mean so we have to disambiguate here.

Share

edited Sep 1, 2008 at 9:21

answered Aug 28, 2008 at 7:57



Konrad Rudolph

545k ● 139 ● 956 ● 1.2k

-
- 2 I quite agree. There is some hope, though. The autocompletion system and the C++ compiler must interact very closely. I am pretty sure Visual Studio will never have such a feature, but things could happen in Eclipse/CDT or some other IDE based on GCC. HOPE ! :) – [Benoit](#) Feb 19, 2009 at 10:48
-

6

votes



Both Java and C# introduced generics after their first language release. However, there are differences in how the core libraries changed when generics was introduced. **C#'s generics are not just compiler magic** and so it was not possible to *generify* existing library classes without breaking backwards compatibility.

For example, in Java the existing [Collections Framework](#) was *completely genericised*. **Java does not have both a generic and legacy non-generic version of the collections classes.** In some ways this is much cleaner - if you need to use a collection in C# there is really very little reason to go with the non-generic version, but those legacy classes remain in place, cluttering up the landscape.

Another notable difference is the Enum classes in Java and C#. Java's Enum has this somewhat tortuous looking definition:

```
// java.lang.Enum Definition in Java
public abstract class Enum<E> extends Enum<E> implements
Serializable {
```

(see Angelika Langer's very clear [explanation of exactly why](#) this is so. Essentially, this means Java can give type safe access from a string to its Enum value:

```
// Parsing String to Enum in Java
Colour colour = Colour.valueOf("RED");
```

Compare this to C#'s version:

```
// Parsing String to Enum in C#  
Colour colour = (Colour)Enum.Parse(typeof(Colour), "RED")
```

As Enum already existed in C# before generics was introduced to the language, the definition could not change without breaking existing code. So, like collections, it remains in the core libraries in this legacy state.

Share

edited Oct 30, 2008 at 0:50



Adam Bellaire

110k ● 19 ● 152 ● 165

answered Aug 28, 2008 at 8:52



serg10

32.6k ● 16 ● 75 ● 94

Even C#'s generics are not just compiler magic, compiler can do further magic to generify existing library. There is no reason why they need to rename `ArrayList` to `List<T>` and put it into a new namespace. The fact is, if there were a class appear in source code as `ArrayList<T>` it will become a different compiler generated class name in the IL code, so there are no name conflicts can happen. – [Earth Engine](#) Oct 12, 2012 at 0:24

4

votes

11 months late, but I think this question is ready for some Java Wildcard stuff.



This is a syntactical feature of Java. Suppose you have a method:



```
public <T> void Foo(Collection<T> thing)
```

And suppose you don't need to refer to the type T in the method body. You're declaring a name T and then only using it once, so why should you have to think of a name for it? Instead, you can write:

```
public void Foo(Collection<?> thing)
```

The question-mark asks the the compiler to pretend that you declared a normal named type parameter that only needs to appear once in that spot.

There's nothing you can do with wildcards that you can't also do with a named type parameter (which is how these things are always done in C++ and C#).

Share

answered Jul 10, 2009 at 13:49





[Daniel Earwicker](#)

117k ● 38 ● 208 ● 286

-
- 2 Another 11 months late... There are things you can do with Java wildcards that you can't with named type parameters. You can do this in Java: `class Foo<T extends List<?>>` and use `Foo<StringList>` but in C# you have to add that extra type parameter: `class Foo<T, T2> where T : IList<T2>` and use the clunky `Foo<StringList, String>`.

– [R. Martinho Fernandes](#) Jun 4, 2010 at 14:00

2 votes



Wikipedia has great write-ups comparing both [Java/C# generics](#) and [Java generics/C++ templates](#). The [main article on Generics](#) seems a bit cluttered but it does have some good info in it.



Share

answered Aug 28, 2008 at 5:14



travis

36.4k ● 21 ● 72 ● 97

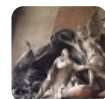
1 vote



The biggest complaint is type erasure. In that, generics are not enforced at runtime. [Here's a link to some Sun docs on the subject.](#)

Generics are implemented by type erasure: generic type information is present only at compile time, after which it is erased by the compiler.



Share

answered Aug 28, 2008 at 5:15



Matt Cummings

2,026 ● 1 ● 20 ● 22

1 vote



C++ templates are actually much more powerful than their C# and Java counterparts as they are evaluated at compile time and support specialization. This allows for Template Meta-Programming and makes the C++ compiler equivalent to a Turing machine (i.e. during the compilation process you can compute anything that is computable with a Turing machine).

Share

answered Aug 28, 2008 at 6:32



On Freund

4,436 ● 2 ● 24 ● 31

1 In Java, generics are compiler level only, so you get:

vote



```
a = new ArrayList<String>()  
a.getClass() => ArrayList
```

Note that the type of 'a' is an array list, not a list of strings. So the type of a list of bananas would equal() a list of monkeys.

So to speak.

Share

answered Aug 28, 2008 at 7:22



izb

51.7k ● 39 ● 118 ● 172

1 Looks like, among other very interesting proposals, there is one about refining generics and breaking backwards compatibility:

vote



Currently, generics are implemented using erasure, which means that the generic type information is not available at runtime, which makes some kind of code hard to write. Generics were implemented this way to support backwards compatibility with older non-generic code. Reified generics would make the

generic type information available at runtime, which would break legacy non-generic code. However, Neal Gafter has proposed making types reifiable only if specified, so as to not break backward compatibility.

at [Alex Miller's article about Java 7 Proposals](#)

Share

edited Jul 23, 2010 at 7:36



Pontus Gagge

17.3k ● 1 ● 42 ● 52

answered Sep 1, 2008 at 18:53



pek

18k ● 28 ● 88 ● 99

0

votes



NB: I don't have enough point to comment, so feel free to move this as a comment to appropriate answer.

Contrary to popular believe, which I never understand where it came from, .net implemented true generics without breaking backward compatibility, and they spent explicit effort for that. You don't have to change your non-generic .net 1.0 code into generics just to be used in .net 2.0. Both the generic and non-generic lists are still available in .Net framework 2.0 even until 4.0, exactly for nothing else but backward compatibility reason. Therefore old codes that still used non-generic ArrayList will still work, and use the same ArrayList class as before. Backward code compatibility is always maintained since 1.0 till now... So

even in .net 4.0, you still have to option to use any non-generics class from 1.0 BCL if you choose to do so.

So I don't think java has to break backward compatibility to support true generics.

Share

edited Aug 3, 2010 at 18:20

answered Aug 3, 2010 at 15:33



Sheepy

676 ● 10 ● 18

That's not the kind of backward compatibility people talk about. The idea is backwards compatibility for the *runtime* : Code written using generics in .NET 2.0 *cannot* be run on older versions of the .NET framework / CLR. Similarly, if Java were to introduce "true" generics, newer Java code wouldn't be able to run on older JVMs (because it requires breaking changes to the bytecode). – [tzaman](#) Aug 3, 2010 at 17:36 ✎

That's .net, not generics. Always requires recompilation to target specific CLR version. There's bytecode compatibility, there's code compatibility. And also, I was replying specifically regarding the need to convert old code that was using old List to use the new generics List, which is not true at all. – [Sheepy](#) Aug 3, 2010 at 18:05 ✎

-
- 1 I think people are talking about *forward compatibility*. I.e. .net 2.0 code to run on .net 1.1, which will break because the 1.1 runtime knows nothing about 2.0 "pseudo-class". Shouldn't it then be that "java doesnt implement true generic because they want to maintain forward compatibility"? (rather than backward) – [Sheepy](#) Aug 3, 2010 at 18:14 ✎
-

Compatibility issues are subtle. I don't think the problem was that adding "real" generics to Java would affect any programs that use older versions of Java, but rather that code which used "new improved" generics would have a hard time exchanging such objects with older code that knew nothing about the new types. Suppose, for example, a program has an `ArrayList<Foo>` that it wants to pass to an older method which is supposed to populate an `ArrayList` with instances of `Foo`. If an `ArrayList<foo>` isn't an `ArrayList`, how does one make that work? – [supercat](#) Jan 6, 2013 at 18:25 