AHK SHORTCUT: **[?templates] or [?generics]**  
TODO: Get a hold on difference between generic and template types so you can avoid re-writing code.

|  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| https://docs.oracle.com/javase/tutorial/extra/generics/methods.html QUESTION: Difference between <T> and T without angle braces?   |  | | --- | | **static <T> void fromArrayToCollection(T[] a, Collection<T> c) {**  **for (T o : a) {**  **c.add(o); // Correct**  **}**  **}** |   https://docs.oracle.com/javase/tutorial/extra/generics/methods.html   |  | | --- | | *One question that arises is: when should I use generic methods, and when should I use wildcard types? To understand the answer, let's examine a few methods from the Collection libraries.* | | **interface Collection<E> {**  **public boolean containsAll(Collection<?> c);**  **public boolean addAll(Collection<? extends E> c);**  **}** | | We could have used generic methods here instead: | | **interface Collection<E> {**  **public <T> boolean containsAll(Collection<T> c);**  **public <T extends E> boolean addAll(Collection<T> c);**  **// Hey, type variables can have bounds too!**  **}** | | However, in both containsAll and addAll, the type parameter T is used only once. The return type doesn't depend on the type parameter, nor does any other argument to the method (in this case, there simply is only one argument). This tells us that the type argument is being used for polymorphism; its only effect is to allow a variety of actual argument types to be used at different invocation sites. If that is the case, one should use wildcards. Wildcards are designed to support flexible subtyping, which is what we're trying to express here. |   https://docs.oracle.com/javase/tutorial/extra/generics/methods.html   |  | | --- | | Generic methods allow type parameters to be used to express **dependencies** among the types of one or more arguments to a method and/or its return type. If there isn't such a dependency, a generic method should not be used. | | It is possible to use both generic methods and wildcards in tandem. Here is the method Collections.copy(): | | **class Collections {**  **public static <T> void copy(List<T> dest, List<? extends T> src) {**  **...**  **}** | | Note the dependency between the types of the two parameters. Any object copied from the source list, src, must be assignable to the element type T of the destination list, dst. So the element type of src can be any subtype of T—we don't care which. The signature of copy expresses the dependency using a type parameter, but uses a wildcard for the element type of the second parameter. |   https://docs.oracle.com/javase/tutorial/extra/generics/methods.html |

AHK SHORTCUT: **[?templates] or [?generics]**

AHK SHORTCUT: **[?templates] or [?generics]**

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| http://programmers.stackexchange.com/questions/258556/what-are-the-main-differences-between-c-templates-and-java-generics   |  | | --- | | In Java you can only use class-types as type arguments, while in C++ any type fits. I don't think this is significant because Java autoboxing is good. | | A more significant difference: C++ templates are duck typed. Meaning you can call any operation on a generic type T. If it doesn't support the operation, a compile time error is raised. Java generics however aren't duck typed. By default all type arguments are of type Object, and you can specify something more specific by the syntax T extends Something, which allows us to call on T methods defined in Something. |   http://programmers.stackexchange.com/questions/258556/what-are-the-main-differences-between-c-templates-and-java-generics   |  | | --- | | Implementation of the generics is night and day | | In java the compiled code removes all references to the generic type and adds casts where necessary. This is called type erasure and lets you do **List<String> list; ((List)list).add(new Object());** which will only throw an error when you try to get the value as a string. All java generics actually are is syntactic sugar with extra compile-time type checking. | | In C++ when you use a template the compiler will emit the template code again after replacing the generic parameter in it with the type you used. This is more powerful in several ways but can lead to bloated executables. |   http://programmers.stackexchange.com/questions/258556/what-are-the-main-differences-between-c-templates-and-java-generics   |  |  |  |  | | --- | --- | --- | --- | | |  |  |  | | --- | --- | --- | | |  | | --- | |  | | So type erasure means: 'replace all the generic type references with references to Object, and add casts down to the generic type everywhere'? – [Aviv Cohn](http://programmers.stackexchange.com/users/121368/aviv-cohn) [Oct 9 '14 at 10:39](http://programmers.stackexchange.com/questions/258556/what-are-the-main-differences-between-c-templates-and-java-generics#comment524096_258557) | | | @AvivCohn just about – ratchet freak Oct 9 '14 at 10:40 | | |  |  |  | | --- | --- | --- | | |  | | --- | |  | | As opposed to C++ templates (or C# generics, I think) which means that for every 'instantiation' of the class/function with a different generic parameter, an actual new instance of the function/class is put into the executable, using the appropriate type? – [Aviv Cohn](http://programmers.stackexchange.com/users/121368/aviv-cohn) [Oct 9 '14 at 10:45](http://programmers.stackexchange.com/questions/258556/what-are-the-main-differences-between-c-templates-and-java-generics#comment524099_258557) | | | |  |  |  | | --- | --- | --- | | |  | | --- | |  | | @AvivCohn C++ templates will function this way, but C# generics do not. C# generics keep the type, but utilize the type at runtime instead of creating multiple methods like C++ does. – [Peter Smith](http://programmers.stackexchange.com/users/14294/peter-smith) [Dec 31 '14 at 16:03](http://programmers.stackexchange.com/questions/258556/what-are-the-main-differences-between-c-templates-and-java-generics#comment545911_258557) | | |

AHK SHORTCUT: **[?templates] or [?generics]**

AHK SHORTCUT: **[?templates] or [?generics]**

|  |  |  |  |
| --- | --- | --- | --- |
| http://tutorials.jenkov.com/java-generics/methods.html   |  | | --- | | It is possible to generify methods in Java. Here is an example: | | **public static <T> T addAndReturn(T element, Collection<T> collection){**  **collection.add(element);**  **return element;**  **}**  **//JMIM QUESTION: Why is <T> next to T??? Typo???** | | This method specifies a type T which is used both as type for the element parameter and the generic type of the Collection. Notice how it is now possible to add elements to the collection. This was not possible if you had used a wildcard in the Collection parameter definition. |   Check my understanding. <T> == Generic Collection. T == Generic Type. http://tutorials.jenkov.com/java-generics/methods.html |

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| http://tutorials.jenkov.com/java-generics/class-objects-as-type-literals.html   |  | | --- | | Class objects can be used as type specifications too, at runtime. For instance, you can create a generified method like this: | | public static <T> T getInstance(Class<T> theClass)  throws IllegalAccessException, InstantiationException {  return theClass.newInstance();  } |   JMIM THOUGHTS: Wait... So..  <T> == Generic Type T == Generic variable of generic type?   |  | | --- | | Here are a few examples of calls to the getInstance() method: | | String string = getInstance(String.class);  MyClass myClass = getInstance(MyClass.class); |   http://tutorials.jenkov.com/java-generics/class-objects-as-type-literals.html |

<http://stackoverflow.com/questions/2189370/how-is-generic-function-implemented-in-java>  
Answer: Type Erasure.  
Type Erasure: https://en.wikipedia.org/wiki/Type\_erasure  
Reification (Opposite of Erasure): <https://en.wikipedia.org/wiki/Reification_%28computer_science%29>

|  |
| --- |
| http://stackoverflow.com/questions/529085/how-to-create-a-generic-array-in-java |
| import java.lang.reflect.Array;  class Stack<T> {  public Stack(Class<T> clazz, int capacity) {  array = (T[])Array.newInstance(clazz, capacity);  }  private final T[] array;  } |

|  |
| --- |
| http://stackoverflow.com/questions/28464614/collection-extends-t-vs-collectiont |
| **class Animal { }**  **class Horse extends Animal { }**  **private static void specific(List<Animal> param) { }**  **private static void wildcard(List<? extends Animal> param) { }** |
| Without the extends syntax you can only use the exact class in the signature |
| **specific(new ArrayList<Horse>()); // <== compiler error** |
| With the wildcard extends you can allow any subclasses of Animal |
| **wildcard(new ArrayList<Horse>()); // <== OK** |
| It's generally better to use the ? extends syntax as it makes your code more reusable and future-proof. |