DOWNLOAD PDF GENERIC PROGRAMMING

Chapter 1 : Generic Programming - ABAP Keyword Documentation

Generic programming is a style of computer programming in which algorithms are written in terms of types to-be-specified-later that are then instantiated when needed for specific types provided as parameters.

Java is a strongly typed language, so a field in a class may be typed like this: Unfortunately, it can be used only with Integer objects. If you want to use the same class in another context with Strings, you have to generalize the type like this: The solution is to use Generics. Generic class[edit] A generic class does not hard code the type of a field, a return value or a parameter. The class only indicates that a generic type should be the same, for a given object instance. The generic type is not specified in the class definition. It is specified during object instantiation. This allows the generic type to be different from an instance to another. So we should write our class this way: Any new identifier can be chosen. Here, we have chosen T, which is the most common choice. The actual type is defined at the object instantiation: Choose a different identifier for each generic type and separate them by a comma: A generic type can be defined for just a method: The type is specific to a method call and different types can be used for the same object instance: Test your knowledge Question 4. Consider the following class. Answer Here is the text: Wildcard Types[edit] As we have seen above, generics give the impression that a new container type is created with each different type parameter. We have also seen that in addition to the normal type checking, the type parameter has to match as well when we assign generics variables. In some cases this is too restrictive. What if we would like to relax this additional checking? What if we would like to define a collection variable that can hold any generic collection, regardless of the parameter type it holds? Any-Type includes Interfaces, not only Classes. So now we can define a collection whose element type matches anything. For example, to create a collection that may only contain "Serializable" objects, specify: Collection of serializable subobjects. Use of a class that is not serializable would cause a compilation error. The added items can be retrieved as Serializable object. You can call methods of the Serializable interface or cast it to String. The following collection can only contain objects that extend the class Animal. For example, to declare a Comparator that can compare Dogs, you use: Comparators for any superclass of Dog can also compare Dog; but comparators for any strict subclass cannot. Use of a class that is not a supertype would cause a compilation error. Unbounded wildcard[edit] The advantage of the unbounded wildcard i. That way, all the operations that implies to know the type are forbidden to avoid unsafe operation. Consider the following code: Integer incompatible with java. With this signature, it is impossible to compile a code that is dependent of the parameterized type. Only independent methods of a collection clear , isEmpty , iterator , remove Object o , size , It is an interesting example of using generics for something other than a container class. For example, the type of String. This can be used to improve the type safety of your reflection code. In particular, since the newInstance method in Class now returns T, you can get more precise types when creating objects reflectively. Now we can use the newInstance method to return a new object with exact type, without casting. An example with generics: There was no way to enforce that a "collection" class contains only one type of object e. This is possible since Java 1. In the first couple of years of Java evolution, Java did not have a real competitor. This has changed by the appearance of Microsoft C. With Generics Java is better suited to compete against C. Similar constructs to Java Generics exist in other languages, see Generic programming for more information. Generics were added to the Java language syntax in version 1. This means that code using Generics will not compile with Java 1. Use of generics is optional. In such a case, when you retrieve an object reference from a generic object, you will have to manually cast it from type Object to the correct type. There are some differences however. All extra code for templates is generated at compiler time. In contrast, Java Generics are built into the language. The same code is used for each generic type. The generic type information is erased during compilation type erasure. ArrayList; 2 import java. Object does not extend String. String is not a superclass of Object. Integer does not extend String.

Chapter 2 : What does generic programming mean?

In the simplest definition, generic programming is a style of computer programming in which algorithms are written in terms of to-be-specified-later types that are then instantiated when needed for specific types provided as parameters.

The second kind of template, a class template, extends the same concept to classes. A class template specialization is a class. Class templates are often used to make generic containers. For example, the STL has a linked list container. This allows alternative implementations to be provided based on certain characteristics of the parameterized type that is being instantiated. Template specialization has two purposes: For example, consider a sort template function. If the values are large in terms of the number of bytes it takes to store each of them , then it is often quicker to first build a separate list of pointers to the objects, sort those pointers, and then build the final sorted sequence. If the values are quite small however it is usually fastest to just swap the values in-place as needed. Furthermore, if the parameterized type is already of some pointer-type, then there is no need to build a separate pointer array. Template specialization allows the template creator to write different implementations and to specify the characteristics that the parameterized type s must have for each implementation to be used. Unlike function templates, class templates can be partially specialized. That means that an alternate version of the class template code can be provided when some of the template parameters are known, while leaving other template parameters generic. This can be used, for example, to create a default implementation the primary specialization that assumes that copying a parameterizing type is expensive and then create partial specializations for types that are cheap to copy, thus increasing overall efficiency. Clients of such a class template just use specializations of it without needing to know whether the compiler used the primary specialization or some partial specialization in each case. Class templates can also be fully specialized, which means that an alternate implementation can be provided when all of the parameterizing types are known. Advantages and disadvantages[edit] Some uses of templates, such as the max function, were previously filled by function-like preprocessor macros a legacy of the C programming language. For example, here is a possible max macro: Macros are always expanded inline; templates can also be expanded as inline functions when the compiler deems it appropriate. Thus both function-like macros and function templates have no run-time overhead. However, templates are generally considered an improvement over macros for these purposes. Templates avoid some of the common errors found in code that makes heavy use of function-like macros, such as evaluating parameters with side effects twice. Perhaps most importantly, templates were designed to be applicable to much larger problems than macros. There are three primary drawbacks to the use of templates: Many compilers historically have poor support for templates, thus the use of templates can make code somewhat less portable. Almost all compilers produce confusing, long, or sometimes unhelpful error messages when errors are detected in code that uses templates. Finally, the use of templates requires the compiler to generate a separate instance of the templated class or function for every permutation of type parameters used with it. So the indiscriminate use of templates can lead to code bloat , resulting in excessively large executables. However, judicious use of template specialization and derivation can dramatically reduce such code bloat in some cases: So, can derivation be used to reduce the problem of code replicated because templates are used? This would involve deriving a template from an ordinary class. This technique proved successful in curbing code bloat in real use. People who do not use a technique like this have found that replicated code can cost megabytes of code space even in moderate size programs. It is a nice approach for creating generic heap-based containers. The extra instantiations generated by templates can also cause debuggers to have difficulty working gracefully with templates. For example, setting a debug breakpoint within a template from a source file may either miss setting the breakpoint in the actual instantiation desired or may set a breakpoint in every place the template is instantiated. Also, because the compiler needs to perform macro-like expansions of templates and generate different instances of them at compile time, the implementation source code for the templated class or function must be available e. Templated classes or functions, including much of the Standard Template Library STL , if not included in header files, cannot be compiled. This is in contrast to non-templated code, which may be compiled to binary, providing only a

declarations header file for code using it. This may be a disadvantage by exposing the implementing code, which removes some abstractions, and could restrict its use in closed-source projects. Template parameters in D are not restricted to just types and primitive values, but also allow arbitrary compile-time values such as strings and struct literals , and aliases to arbitrary identifiers, including other templates or template instantiations. The keyword and the typeof expression allow type inference for variable declarations and function return values, which in turn allows "Voldemort types" types which do not have a global name. If there is only one parameter, the parentheses can be omitted. Conventionally, D combines the above features to provide compile-time polymorphism using trait-based generic programming. For example, an input range is defined as any type that satisfies the checks performed by isInputRange , which is defined as follows: The expression allows reading a file from disk and using its contents as a string expression. Compile-time reflection allows enumerating and inspecting declarations and their members during compilation. User-defined attributes allow users to attach arbitrary identifiers to declarations, which can then be enumerated using compile-time reflection. String mixins allow evaluating and compiling the contents of a string expression as D code that becomes part of the program. Combining the above allows generating code based on existing declarations. User-defined attributes could further indicate serialization rules. The import expression and compile-time function execution also allow efficiently implementing domain-specific languages. For example, given a function that takes a string containing an HTML template and returns equivalent D source code, it is possible to use it in the following way: The foundation publications of Eiffel, [21] [22] use the term genericity to describe the creation and use of generic classes. G -- The item currently pointed to by cursor Constrained genericity[edit] For the list class shown above, an actual generic parameter substituting for G can be any other available class. To constrain the set of classes from which valid actual generic parameters can be chosen, a generic constraint can be specified. Generics in Java Support for the generics, or "containers-of-type-T" was added to the Java programming language in as part of J2SE 5. In Java, generics are only checked at compile time for type correctness. The generic type information is then removed via a process called type erasure , to maintain compatibility with old JVM implementations, making it unavailable at runtime. NET][edit] Generics were added as part of. NET generics do not apply type erasure , but implement generics as a first class mechanism in the runtime using reification. This design choice provides additional functionality, such as allowing reflection with preservation of generic types, as well as alleviating some of the limitations of erasure such as being unable to create generic arrays. When primitive and value types are used as generic arguments, they get specialized implementations, allowing for efficient generic collections and methods. NET allows six varieties of generic type constraints using the where keyword including restricting generic types to be value types, to be classes, to have constructors, and to implement interfaces. This ensures a compile time error, if the method is called if the type does not support comparison. The interface provides the generic method CompareTo T The above method could also be written without generic types, simply using the non-generic Array type. However, since arrays are contravariant , the casting would not be type safe , and compiler may miss errors that would otherwise be caught while making use of the generic types. In addition, the method would need to access the array items as objects instead, and would require casting to compare two elements. A notable behavior of static members in a generic. NET class is static member instantiation per run-time type see example below. NET compiler before being added to the native code in the Delphi release. The semantics and capabilities of Delphi generics are largely modelled on those had by generics in. T ; overload; end; class procedure TUtils. Create 0, 1, 2, 3 ; TUtils. As with C , methods as well as whole types can have one or more type parameters. The available constraints are very similar to the available constraints in C: Multiple constraints act as an additive union. Genericity in Free Pascal[edit] Free Pascal implemented generics before Delphi, and with different syntax and semantics. This allows Free Pascal programmers to use generics in whatever style they prefer. Delphi and Free Pascal example:

Chapter 3 : C# Generics Programming

Generics were added to version of the C# language and the common language runtime (CLR). Generics introduce to blog.quintoapp.com Framework the concept of type parameters, which make it possible to design classes and methods that defer the specification of one or more types until the class or method is.

Generic Programming G eneric programming refers to writing code that will work for many types of data. The source code presented there for working with dynamic arrays of integers works only for data of type int. But the source code for dynamic arrays of double, String, JButton, or any other type would be almost identical, except for the substitution of one type name for another. It seems silly to write essentially the same code over and over. ArrayList is just one class, but the source code works for many different types. This is generic programming. The ArrayList class is just one of several standard classes that are used for generic programming in Java. All the classes and interfaces discussed in these sections are defined in the package java. For example, both java. List exist, so it is often better to import the individual classes that you need. In the final section of this chapter, we will see that it is possible to define new generic classes, interfaces, and methods. It is no easy task to design a library for generic programming. It is almost certainly not the best, and has a few features that in my opinion can only be called bizarre, but in the context of the overall design of Java, it might be close to optimal. To get some perspective on generic programming in general, it might be useful to look very briefly at some other approaches to generic programming. It is still used today, although its use is not very common. In Smalltalk, essentially all programming is generic, because of two basic properties of the language. First of all, variables in Smalltalk are typeless. A data value has a type, such as integer or string, but variables do not have types. Any variable can hold data of any type. Parameters are also typeless, so a subroutine can be applied to parameter values of any type. Similarly, a data structure can hold data values of any type. There is simply no need to write new code for each data type. Secondly, all data values are objects, and all operations on objects are defined by methods in a class. This is true even for types that are "primitive" in Java, such as integers. There is no need to write a different sorting subroutine for each type of data. Put these two features together and you have a language where data structures and algorithms will work for any type of data for which they make sense, that is, for which the appropriate operations are defined. This is real generic programming. Once you have a data structure that can contain data of any type, it becomes hard to ensure that it only holds the type of data that you want it to hold. More particularly, there is no way for a compiler to ensure these things. The problem will only show up at run time when an attempt is made to apply some operation to a data type for which it is not defined, and the program will crash. Every variable has a type, and can only hold data values of that type. It is made possible by a language feature known as templates. Even though it says "class ItemType", you can actually substitute any type for ItemType, including the primitive types. If you substitute "string" for "ItemType", you get a subroutine for sorting arrays of strings. This is pretty much what the compiler does with the template. If your program says "sort list,10 " where list is an array of ints, the compiler uses the template to generate a subroutine for sorting arrays of ints. If you say "sort cards,10 " where cards is an array of objects of type Card, then the compiler generates a subroutine for sorting arrays of Cards. At least, it tries to. If this operator is defined for values of type Card, then the compiler will successfully use the template to generate a subroutine for sorting cards. If you write a template for a binary tree class, you can use it to generate classes for binary trees of ints, binary trees of strings, binary trees of dates, and so on -- all from one template. The STL is quite complex. Early versions of Java did not have parameterized types, but they did have classes to represent common data structures. Those classes were designed to work with Objects; that is, they could hold objects of any type, and there was no way to restrict the types of objects that could be stored in a given data structure. For example, ArrayList was not originally a parameterized type, so that any ArrayList could hold any type of object. This means that if list was an ArrayList, then list. If the programmer was actually using the list to store Strings, the value returned by list. Unfortunately, as in Smalltalk, the result is a category of errors that show up only at run time, rather than at compile time. If a programmer assumes that all the items in a data structure are strings and tries to process

those items as strings, a run-time error will occur if other types of data have inadvertently been added to the data structure. In Java, the error will most likely occur when the program retrieves an Object from the data structure and tries to type-cast it to type String. If the object is not actually of type String, the illegal type-cast will throw an error of type ClassCastException. Furthermore, the return type of list. In this chapter, I will use the parameterized types exclusively, but you should remember that their use is not mandatory. It is still legal to use a parameterized class as a non-parameterized type, such as a plain ArrayList. In that case, any type of object can be stored in the data structure. Every time the template is used with a new type, a new compiled class is created. With a Java parameterized class, there is only one compiled class file. For example, there is only one compiled class file, ArrayList. The type parameter -- String or Integer -- just tells the compiler to limit the type of object that can be stored in the data structure. The type parameter has no effect at run time and is not even known at run time. The type information is said to be "erased" at run time. This type erasure introduces a certain amount of weirdness. Most people who use parameterized types will not encounter the problems, and they will get the benefits of type-safe generic programming with little difficulty. We will spend the next few sections learning about the JFC. The generic data structures in the Java Collection Framework can be divided into two categories: A collection is more or less what it sounds like: A map associates objects in one set with objects in another set in the way that a dictionary associates definitions with words or a phone book associates phone numbers with names. Here, "T" and "S" stand for any type except for the primitive types. In this section and the next, we look at collections only. There are two types of collections: A list is a collection in which the objects are arranged in a linear sequence. A list has a first item, a second item, and so on. For any item in the list, except the last, there is an item that directly follows it. For collections that are "sets," the defining property is that no object can occur more than once in a set; the elements of a set are not necessarily thought of as being in any particular order. Of course, any actual object that is a collection, list, or set must belong to a concrete class that implements the corresponding interface. This means that all the methods that are defined in the list and collection interfaces can be used with an ArrayList. We will look at various classes that implement the list and set interfaces in the next section. Since "collection" is a very general concept, operations that can be applied to all collections are also very general. They are generic operations in the sense that they can be applied to various types of collections containing various types of objects. The parameter must be of type T; if not, a syntax error occurs at compile time. Remember that if T is a class, this includes objects belonging to a subclass of T, and if T is an interface, it includes any object that implements T. The add method returns a boolean value which tells you whether the operation actually modified the collection. For example, adding an object to a Set has no effect if that object was already in the set. Note that object is not required to be of type T, since it makes sense to check whether object is in the collection, no matter what type object has. For testing equality, null is considered to be equal to itself. The test for equality is the same test that is used by contains. The parameter can be any collection. However, it can also be more general. This makes sense because any object of type S is automatically of type T and so can legally be added to coll. It "retains" only the objects that do occur in coll2. Note that the return type is Object[], not T[]! However, there is another version of this method that takes an array of type T[] as a parameter: If the array parameter tarray is large enough to hold the entire collection, then the items are stored in tarray and tarray is also the return value of the collection. If tarray is not large enough, then a new array is created to hold the items; in that case tarray serves only to specify the type of the array. There is a problem with this, however. For example, the size of some collections cannot be changed after they are created. While it is still legal to call the methods, an exception will be thrown when the call is evaluated at run time. The type of the exception is UnsupportedOperationException. This means that the semantics of the methods, as described above, are not guaranteed to be valid for all collection objects; they are valid, however, for classes in the Java Collection Framework. There is also the question of efficiency.

Chapter 4 : Generic programming - Wikipedia

Introduction to Generics (C# Programming Guide) 07/20/; 2 minutes to read Contributors. all; In this article. Generic classes and methods combine reusability, type safety and efficiency in a way that their non-generic counterparts cannot.

Generic Programming Background Generic programming dynamic creation of source code makes programs as dynamic as possible. The following methods can be used to achieve this: Dynamic token specification Dynamic token specification involves specifying individual operands or whole parts of statements clauses in the form of character-like data objects. These are usually enclosed in parentheses and must contain source code with correct syntax at runtime. Dynamic access to attributes of classes Dynamic Access Call procedures dynamically, especially methods Dynamic Invoke Dynamic type specifications when anonymous data objects are created. Dynamic specifications of clauses when internal tables are accessed or in Open SQL. Dynamic token specifications are often used in combination with dynamic access to data objects. Program generation Program generation involves preparing complete programs as content for internal tables and then creating the programs. A distinction is made between the following cases: The generated programs are saved as repository objects. Rule Avoiding Program Generation Program generation should only be used as a last resort for generic programming. Other dynamic methods especially in application programs should be tried first, such as dynamic token specification, runtime type services RTTS and dynamic access to data objects. Details Program generation has a lot of conceptual problems such as checking, testing, and editing the new programs. In addition, programs generated hastily can be a security risk because they cannot be statically checked. Creating programs is usually very intensive in terms of runtime and resources. Due to the above reasons, program generation should be avoided wherever possible and other dynamic methods used instead: The dynamic token specification has the advantage that only parts of the statements are dynamic. The rest can be checked statically. Runtime type services RTTS can be used as follows: These methods, combined with field symbols and data references , are now usually sufficient for most tasks that could only be solved using program generation in older releases. Exception Program generation should only be used as a last resort if the other methods are not sufficient to achieve dynamic program control. Another reason is the processing speed. When program generation is used, the costs incurred due to checking and generation occur less frequently than with the other dynamic methods. However, program generation is usually associated with worse system performance than dynamic token specification. The conceptual problems explained at the start are still applicable here and therefore careful consideration is needed. Unlike application programs, system programs usually frequently rely on program generation and the associated language constructs. Examples include the generation of proxy classes for Web Dynpro or Web Services. Note Even generated programs should always adhere to the predefined guidelines. The generated functions are usually called by means of a single subroutine that is used as an entry point into the generated local classes see the following example. To minimize the risks involved, it can be useful to save templates with correct syntax that adhere to the guidelines in the repository. Subroutines in generated subroutine pools are an exception to the rule , which states that subroutines should no longer be created, and an exception to the rule , which states that subroutines should no longer be called. In addition, absolute type names can be used to access the local classes of a generated subroutine pool. However, this violates the rule Only call suitable procedures externally. Bad Example The following source code demonstrates unnecessary program generation. The only reason for generating the program are dynamic reads on a database table. The name of the database table and the row type of the internal table into which data is read are replaced by a parameter value in the source code of the program to be generated. As recommended, the subroutine of the generated subroutine pool only contains the call for a method of a local class where the actual implementation is located. Instead of filling the program table row by row, it would have also been possible in this case to create a corresponding program in the repository and load this program using READ REPORT.

Chapter 5 : What is the meaning of "generic programming" in c++? - Stack Overflow

Generic programming was introduced in Java and this is basically a way to write reusable software and to avoid Boilerplate code (Boilerplate code - Wikipedia). Basically this allows you to write code even if you do not yet know the type of all members or argument.

Recently, I have played some more with Shapeless. This time with the goal of generating React javascript components for case classes. Getting Started As I mentioned before, I had to define implicits for the simple types that I wanted to be able to handle, and the starting point is, of course, accepting a case class as the input. First of all, the method is parameterized with two types: Next up, we are expecting several implict method arguments. We will ignore the third implicit for now â€" that is an implicit that I am using purely for the react side of things. This can be skipped if the method handles everything itself: Then, we can start dealing with the fields one by one. In other words, this is the first step to converting a case class to a generic list that we can then handle element by element. Straight forward so far? The first implicit argument is a bit more interesting. If not, we will get a compile error. But, things get more interesting if we look more at what the. Path Dependent Types and the Aux Pattern The best way to work out what is going on is to just jump into the Shapeless code and have a dig. That is, the type is dependent on the actual instance of the enclosing trait or class. This is a powerful mechanism in Scala. But, it is also one that can also catch you out, if you are in the habit of defining classes within other classes or traits. But, it would be a real pain to have to actually define that as a type parameter in the class along with our case class type, so it uses this path-dependent type approach. The best way to work out what is going on from it is to take a look at the Shapeless code! They have a pretty decent explanation of what is going on in the comments, so I will reproduce that here: Aux[T,R] by implicit search. So Aux neatly sidesteps this problem. This pattern allows us to essentially promote the result of a type-level computation to the higher level parameter. It can be used for a variety of things where we want to reason about the path dependent types. Besies this, this is a more common use for the pattern. So, thats all I wanted to get into for now. You can see the code here!

Chapter 6 : Elegant Coding: What is Generic Programming?

By generic programming, the definition of algorithms and data structures at an abstract or generic level, thereby accomplishing many related programming tasks simultaneously. The central notion is that of generic algorithms, which are parameterized procedural schemata that are completely independent of the underlying data representation and are.

Chapter 7 : Generics (C# Programming Guide) | Microsoft Docs

â€¢ simple generic code as simple as non-generic code â€¢ more advanced generic code as easy to use and not that much more difficult to write. generic programming.

Chapter 8 : Generic programming in Fortran Wiki

Generic programming is about generalizing software components so that they can be easily reused in a wide variety of situations. In C++, class and function templates are particularly effective mechanisms for generic programming because they make the generalization possible without sacrificing efficiency.

Chapter 9 : Generic Programming

This feature is not available right now. Please try again later.