Chapter: 9

**Generics**

Generic Class, Wildcards, Generic Methods, Generic Constructors, Generic Interfaces

**9.1 Generics : Fundamentals**

At its core, the term generics means parameterized types ***(No exact types, type varies)***. Parameterized types are important because they enable you to create classes, interfaces, and methods in which the type of data upon which they operate is specified as a parameter. A class, interface, or method that operates on a type parameter is called generic, as in generic class or generic method.

* Java’s generics are similar to templates in C++ : **Java** generics are similar to templates in C++. What Java calls a *parameterized* *type*, C++ calls a *template*. However, Java ***generics*** and C++ ***templates*** are ***not the same***, and there are some fundamental differences between the two approaches to generic types. For the most part, Java’s approach is simpler to use.
* Warning: If you have a background in C++, it is important not to jump to conclusions about how generics work in Java. The two approaches to generic code differ in subtle but fundamental ways.
* *Generic code* automatically work with the type of data passed to its *type parameter*. Many algorithms are logically the same no matter what type of data they are being applied to. Eg: Quicksort is the same for int, str, Object, or Thread (similar C++ idea).
* Example 1: Following program defines two classes. The first is the generic class Gen, and the second is GenDemo, which uses Gen.

*// Here, T is a type parameter that will be replaced by a real type when an object of type Gen is created.*

**class** Gen<T> { **T** ob; *// declare an object of type T*

Gen(**T** o) { ob = o; } *// Pass the constructor a reference to an object of type T.*

**T** getob() { **return** ob; } *// Return ob.*

**void** showType() { **System.out.println**("Type of T is " + ob.**getClass**().**getName**()); } *// Show type of T.*

}

*// Demonstrate the generic class.*

**class** GenDemo { **public static void main(String args[])** {

**Gen<Integer>** iOb; *// Create a reference to an object of type Gen<Integer>.*

|  |  |
| --- | --- |
| Output:  **Type of T is java.lang.Integer**  **value: 88**  **Type of T is java.lang.String**  **value: Generics Test** | *// Notice the use of autoboxing to encapsulate the value 88 within an Integer object.*  iOb = **new** Gen<**Integer**>(88); *// Create a Gen<Integer> object and assign its reference to iOb.*  iOb.showType(); *// Show the type of data used by iOb*  **int** v = iOb.getob(); *// Get the value in iOb. Notice that no cast is needed*  **System.out.println**("value: " + v + "\n");  *// Create a reference and an object of type Gen<String>.*  **Gen<String>** strOb = **new** Gen<**String**>("Generics Test");  strOb.showType(); *// Show the type of data used by strOb.*  **String** str = strOb.getob(); *// Get the value of strOb. Again, notice that no cast is needed.*  **System.out.println**("value: " + str); }} |

* Notice how ***Gen*** is declared by the line: ***class Gen<T> {***
* Here, **T** is the name of a type parameter. (Which is a placeholder for the actual type that will be passed to Gen when an object is created. )
* Notice,***T*** is contained within ***< >***. Whenever a type parameter is being declared, it is specified within ***angle brackets***.

[In the declaration of Gen, there is no special significance to the name T. Any valid identifier could have been used, but T is traditional. Furthermore, it is recommended that type parameter names be single-character, capital letters. Other commonly used type parameter names are V and E.]

* Next, **" T ob; "** **T** is used to declare an object called ***ob***. ***ob*** will be an object of the type passed to ***T***. For example, if type ***String*** is passed to ***T***, then in that instance, ***ob*** will be of ***type*** ***String***.
* Now consider ***Gen’s*** constructor: ***Gen(T o) { ob = o; }*** Notice that its parameter, ***o***, is of type ***T***.
* The type parameter ***T*** can also be used to specify the return type of method, as is the case with the ***getob()*** method,:

***T getob() { return ob; }***

Because ***ob*** is also of type ***T***, its type is compatible with the return type specified by ***getob( )***.

* ***showType()*** displays the type of **T**. By calling ***getName()*** on the ***Class*** object returned by the call to ***getClass()*** on ***ob***.
* Haven’t used this feature before: Recall Chapter 4, the ***Object*** class defines the method ***getClass()***. Thus, ***getClass()*** is a member of all class types. It returns a ***Class*** object that corresponds to the *class type* of the object on which it is called.
* ***Class*** is a class defined within ***java.lang*** that encapsulates information about a class. ***Class*** defines several methods that can be used to obtain information about a class at ***run time***. Among these is the ***getName()*** method, which returns a *string representation* of the *class name.*
* A version of ***Gen*** for integers is created first: ***Gen<Integer> iOb;***
* Notice that the type ***Integer*** is specified within the **< >** after ***Gen***. ***Integer*** is a *type argument* that is passed to ***Gen***’s*type parameter*, ***T***. This effectively *creates a version* of ***Gen*** in which all *references* to ***T*** are translated into *references* to ***Integer***. Thus, for this declaration, ***ob*** is of type ***Integer***, and the return type of ***getob()*** is of type ***Integer***.
* ***iOb = new Gen<Integer>(88);***  assigns to ***iOb*** a reference to an instance of an ***Integer*** version of the ***Gen*** class.

Notice when ***Gen*** constructor is called, the *type argument* ***Integer*** is also specified. Since the type of ***iOb*** is Gen<Integer>. Thus, the reference returned by ***new*** must also be of type Gen<Integer>. Otherwise *compile-time error* will result.

* For example, ***iOb = new Gen<Double>(88.0);*** will cause a *compile-time error*. Because ***iOb*** is of type ***Gen<Integer>***, it can’t be used to refer to an object of ***Gen<Double>***. *[This type of checking is one of the advantages of generics.]*
* ***iOb = new Gen<Integer>(88);*** makes use of autoboxing to encapsulate the value 88, which is an int, into an Integer.
* This works because ***Gen<Integer>*** creates a constructor that takes an ***Integer*** argument. Since an Integer is expected, Java autoboxes 88 inside one. Of course, the assignment could also have been written explicitly, like following:

iOb = **new** Gen<**Integer**>(**new** Integer(88));

* ***int v = iOb.getob();*** The return type of ***getob()*** is also ***Integer***, which *auto-unboxes* into ***int*** when assigned to ***v*** (which is an ***int***). [Because the return type of getob() is T, which was replaced by Integer when iOb was declared] Thus, there is no need to *cast the return type* of getob() to Integer.
* ***Gen<String> strOb = new Gen<String>("Generics Test");***
* Because the type argument is ***String***, ***String*** is substituted for ***T*** inside ***Gen***. This creates a ***String*** version of ***Gen***.

Note

***Java compiler does not actually create different versions*** of Gen, or of any other generic class. Instead, the compiler removes all generic type information, substituting the necessary casts, to make your code behave as if a specific version of Gen was created.

* Thus, there is really only one version of ***Gen*** that actually exists in your program.
* The process of removing generic type information is called ***erasure***, which is discussed later in this chapter.

**9.2 Generics : Details**

* Generics Work Only with Reference Types: When declaring an *instance* of a generic type, the ***type argument*** passed to the ***type*** ***parameter*** must be a reference type. You cannot use a primitive type, such as ***int*** or ***char***. For example, with ***Gen***, it is possible to pass any class type to **T**, but you cannot pass a primitive type to **T**. i.e. ***Gen<int> intOb = new Gen<int>(53);*** is illegal.
* You can use the type-wrappers/ autoboxing-autounboxing to encapsulate a ***primitive type***.
* Generic Types Differ Based on Their Type Arguments: A reference of ***one specific version of a*** generic type is not type-compatible with ***another version*** of the same generic type. For example, iOb = strOb; of Example 1 is in *error* and will *not compile*:
* In Example 1: Even though both ***iOb*** and ***strOb*** are of type ***Gen<T>***, they are references to *different types* because their type arguments differ. *[That’s how generics add type safety and prevent errors.]*
* A Generic Class with Two Type Parameters: To specify two or more type parameters, simply use a comma-separated list.
* Example 2: Following ***TwoGen*** class is a variation of the ***Gen*** class that has two type parameters:

|  |  |
| --- | --- |
| **class** TwoGen<T, V> { **T** ob1; **V** ob2;  TwoGen(**T** o1, **V** o2) { ob1 = o1; ob2 = o2; }  **void** showTypes() {  **System.out.println**("Type of T is " + ob1.**getClass().getName()**);  **System.out.println**("Type of V is " + ob2.**getClass().getName()**); }  T getob1() { return ob1; }  V getob2() { return ob2; } } | **class** SimpGen { **public static void main(String args[])** {  **TwoGen<Integer, String>** tgObj =  **new** TwoGen<**Integer**, **String**>(88, "Generics");  tgObj.showTypes(); *// Show the types.*  int v = tgObj.getob1(); *// Obtain and show values.*  **String** str = tgObj.getob2();  **System.out.println**("value: " + v + "value: " + str); }} |

* Notice how TwoGen is declared: **class TwoGen<T, V> {** *//…* **}** It specifies two type parameters, ***T*** and ***V***, separated by a ***comma***. Since it has two ***type parameters***, **two** ***type arguments*** must be passed to TwoGen when an object is created, as:

**TwoGen<Integer, String>** tgObj = **new** TwoGen<**Integer**, **String**>(88, "Generics");

* In this case, Integer is substituted for T, and String is substituted for V.
* Both *type arguments* can be the same. Eg: ***TwoGen<String, String> x = new TwoGen<String, String>("A", "B");***
* In this case, both T and V would be of type String. If the ***type arguments*** are the same, then two type parameters (i.e. T and V ) would be unnecessary.
* General Form of a Generic Class: The syntax for declaring a generic class: ***class class-name<type-param-list>{*** ***}***
* Here is the full syntax for ***declaring a*** reference ***to a generic class*** and creating a ***generic*** instance:

***class-name<type-arg-list> var-name = new class-name<type-arg-list>(cons-arg-list);***

* *Bounded Types:* In the preceding examples, the *type parameters* could be replaced by *any class type*. But sometimes it is useful to limit the types that can be passed to a *type parameter*.
* Example 3: To create a ***generic class*** that stores a ***numeric value*** and is capable of performing various mathematical functions using any type of number, including ***integers***, ***floats***, and ***doubles***:

**class** NumericFns<T> { **T** num;

NumericFns(**T** n) { num = n; }

**double** reciprocal() { **return** 1 / num.doubleValue(); */\* Error! \*/* } *// Return the reciprocal*.

**double** fraction() { **return** num.**doubleValue**() - num.**intValue**(); */\* Error!\*/* } *// Return the fractional component.*

}

* Unfortunately, NumericFns will not compile as written because *both methods* will generate *compile-time errors*.
* In ***reciprocal()*** method, the value of ***num*** is obtained by calling ***doubleValue()***, which obtains the double version of the numeric object stored in ***num***.
* All ***numeric wrapper classes***, such as ***Integer*** and ***Double***, are subclasses of ***Number***, and ***Number*** defines the ***doubleValue()*** method, this ***method*** is available to all ***numeric wrapper classes***.
* The trouble is that the compiler has no way to know that you are intending to create NumericFns objects using only numeric types. Thus, when you try to compile NumericFns, an error is reported that indicates that the doubleValue( ) method is unknown. The same type of error occurs twice in fraction(), which needs to call both doubleValue( ) and intValue( ). Both calls result in error messages stating that these methods are unknown.
* To solve this problem, you need some way to tell the compiler that you intend to pass only numeric types to T. Furthermore, you need some way to ensure that only numeric types are actually passed.
* *Bounded Types:* To handle such situations, Java provides bounded types. When specifying a *type parameter*, you can create an *upper bound* that ***declares the*** superclass ***from which all*** type arguments ***must be derived***. This is accomplished through the use of an ***extends*** clause when specifying the *type parameter* : ***<T extends superclass>***
* specifies that T can be replaced only by superclass, or subclasses of superclass. [i.e. superclass defines an inclusive, upper limit.]
* Example 4: We can use an upper bound to fix the ***NumericFns*** class shown earlier by specifying ***Number*** as an upper bound:

|  |  |
| --- | --- |
| *// T must be either Number, or a class derived from Number.*  **class** NumericFns<T **extends** **Number**> { */\* all as in Example 3 \*/* }  **class** BoundsDemo { **public static void main(String args[])** {  **NumericFns<Integer>** iOb = **new** NumericFns<**Integer**>(5);  **System.out.println**("Reciprocal: " + iOb.reciprocal());  **System.out.println**("Fractional component : " + iOb.fraction());  **System.out.println**(); | **NumericFns<Double>** dOb = **new** NumericFns<**Double**>(5.25);  **System.out.println**("Reciprocal : " + dOb.reciprocal());  **System.out.println**("Fractional component : " + dOb.fraction());  *// Following won't compile because String is not a subclass of Number.*  *// NumericFns<String> strOb = new NumericFns<String>("Err");*  }} |

* Notice how ***NumericFns*** is now declared: **class** NumericFns<T **extends Number**> {}
* Because the type ***T*** is now bounded by ***Number***, Java compiler knows that all objects of type ***T*** can call ***doubleValue()*** because it is a method declared by ***Number***.
* The *bounding* of ***T*** also prevents *nonnumeric* ***NumericFns*** objects from being created. For example, you will receive *compile-time errors* because ***String*** is not a subclass of ***Number***.
* Parameter compatibility: *Bounded types* are especially useful when you need to ensure that *one type parameter is compatible* *with another*. For example, consider the following class called Pair

**class** Pair<T, V **extends** T>{ *// Here, V must be either the same type as T, or a subclass of T.*

**T** first; **V** second;

Pair(**T** a, **V** b) { first = a; second = b; }

*/\* ... \*/* }

* Notice that ***Pair*** uses two type parameters, ***T*** and ***V***, and that ***V*** ***extends T***. This means that ***V*** will either be the *same* as ***T*** or a *subclass* of ***T***. This ensures that the two arguments to ***Pair***’s constructor ***will be objects of the same type or of related types***. For example, the following constructions are valid:

**Pair<Integer, Integer> x = new Pair<Integer, Integer>(1, 2);** *// This is OK because both T and V are Integer.*

**Pair<Number, Integer> y = new Pair<Number, Integer>(10.4, 12);** *// OK because Integer is a subclass of Number.*

* *Following is invalid:* In this case, ***String*** is not a subclass of ***Number***, which violates the *bound* specified by ***Pair***.

**Pair<Number, String> z = new Pair<Number, String>(10.4, "12");** *// error because String is not a subclass of Number*

**9.3 Wildcard Arguments**

Consider Example 4, in ***NumericFns*** we want to create a method called ***absEqual()*** that returns ***true*** if two NumericFns objects contain numbers ***whose absolute values are the same***.

* Now we want to make ***absEqual()*** generic, For example, if one object contains the ***Double*** value ***1.25*** and the other object contains the ***Float*** value ***–1.25***, then ***absEqual( )*** would return ***true***. And we want to be able to call ***absEqual()***, as:

*/\* typical call to generic absEqual() \*/*

**NumericFns<Double>** dOb = **new** NumericFns<**Double**>(1.25);

**NumericFns<Float>** fOb = **new** NumericFns<**Float**>(-1.25);

**if**(dOb.absEqual(fOb)) **System.out.println**("Absolute values are the same.");

**else** **System.out.println**("Absolute values differ.");

* But how can we define **absEqual()** method ? Normally we can define as follows (which won't work actually):

**boolean** absEqual(**NumericFns<T>** ob) { *// This won't work!*

**if**(**Math**.**abs**(num.**doubleValue**()) == **Math.abs**(ob.num.**doubleValue**()) **return** **true**;

**return** **false**; }

* This won't work because what ***type*** do you specify for ***NumericFns***’ type parameter? The problem is that, ***Math.abs( )*** will work only with other ***NumericFns*** objects whose type is the ***same as the invoking object***. For example, if the ***invoking object*** is of type ***NumericFns<Integer>***, then the parameter ***ob*** must also be of type ***NumericFns<Integer>*** (i.e. it can’t be used to compare an object of type ***NumericFns<Double>***).
* Wildcard " **?** " Argument : *Wildcard argument* is another feature of Java *generics*. ***Wildcard argument*** is specified by the ***?***, and it represents an unknown type. Using a ***wildcard***, here is one way to write the ***absEqual()*** method:

**boolean** absEqual(**NumericFns**<**?**> ob) { *//wildcard argument is used*

**if**(**Math.abs**(num.d**oubleValu**e()) == **Math.abs**(ob.num.**doubleValue**())) **return true**;

**return false**; }

* Here, ***NumericFns<?>*** matches any type of ***NumericFns object***, allowing any two ***NumericFns*** ***objects*** to have their absolute values compared.
* Example 5: The following program demonstrates ***wildcard***.

|  |  |
| --- | --- |
| **class** NumericFns<T **extends** **Number**> {  **T** num;  NumericFns(**T** n) { num = n; } *// constructor*  **double** reciprocal() { **return** 1 / num.**doubleValue**(); }  **double** fraction() { **return** num.**doubleValue**() - num.**intValue**(); }  **boolean** absEqual(**NumericFns<?>** ob) { *// absolute values check*  **if**(**Math.abs**(num.**doubleValue**()) == **Math.abs**(ob.num.**doubleValue**())) **return** **true**;  **return** **false**; } } | **class** WildcardDemo {  **public static void main(String args[])** {  **NumericFns<Integer>** iOb = **new** NumericFns<**Integer**>(3);  **NumericFns<Float>** fOb = **new** NumericFns<**Float**>(-3.00);  **if**(iOb.absEqual(fOb)) **System.out.println**("Abs values are same.");  **else** **System.out.println**("Absolute values differ."); }} |

* Notice, ***iOb*** is an object of type ***NumericFns<Integer>*** and ***fOb*** is an object of type ***NumericFns<Float>***. However, through the use of a wildcard, it is possible for ***iOb*** to pass ***fOb*** in the call to ***absEqual( )***.
* Type **remain unaffected:** wildcard doesn’t affect the type (i.e. doesn’t affect what type of ***NumericFns*** ***objects*** can be created). This is ***governed by the*** extends ***clause*** in the ***NumericFns*** declaration. The wildcard simply matches any valid ***NumericFns*** ***object***.
* Bounded Wildcards: Wildcard arguments can be bounded in much the same way that a *type parameter* can be *bounded*. A *bounded wildcard* is especially important when you are *creating a method* that is designed to operate only on ***objects that are subclasses of a specific superclass***.
* Upper bound: To establish an *upper bound* for a ***wildcard***, use this *wildcard expression*: ***< ? extends superclass>***
* Where superclass is the name of the class that serves as the upper bound.
* It is an inclusive clause because the class forming the upper bound (specified by *superclass*) is also within bounds.
* Lower bound: To specify a *lower bound* for a ***wildcard***, add a ***super*** to a *wildcard declaration*: ***<? super subclass>***
* In this case, only classes that are *superclasses* of ***subclass*** are acceptable arguments. This is an *inclusive clause*.
* Example 6: Let’s work through a simple example. Consider the following set of classes:

|  |  |
| --- | --- |
| **class** A { */\* ... \*/* }  **class** B **extends** A { */\* ... \*/* }  **class** C **extends** A { */\* ... \*/* }  *// Note that D does NOT extend A.*  **class** D { */\* ... \*/* }   * Next, consider the following very simple generic class:   *// A simple generic class.*  **class** Gen<T>{ **T** ob;  Gen(**T** o) { ob = o; } }   * ***Gen*** takes one type parameter, which specifies the type of object stored in ***ob***. Because **T** is ***unbounded***, the type of T is ***unrestricted***. That is, **T** can be of any *class type*. * Now, suppose that you want to create a method that ***operates only on objects*** of ***Gen<type>***, where ***type*** is either **A** or a subclass of **A**. To accomplish this, you must use a bounded wildcard. For example, here is a method called ***test()*** that accepts as an argument only ***Gen*** objects whose type parameter is ***A*** or a subclass of ***A***:   *// Here, the ? will match A or any class type that extends A.*  **static** **void** test(Gen<? extends A> o){ */\* ... \*/* } | The following class demonstrates the *types* of ***Gen*** objects that can be passed to ***test()***.  **class** UseBoundedWildcard {  **static void** test(**Gen**<**?** **extends** A> o) { */\* . . . \*/* } *// bounded wildcard.*  **public static void main(String args[])** { **A** a = **new** A();  **B** b = **new** B();  **C** c = **new** C();  **D** d = **new** D();  **Gen<A>** w = **new** Gen<A>(a);  **Gen<B>** w2 = **new** Gen<B>(b);  **Gen<C>** w3 = **new** Gen<C>(c);  **Gen<D>** w4 = **new** Gen<D>(d);  test(w); *// call to test() is OK.*  test(w2); *// call to test() is OK.*  test(w3); *// call to test() is OK.*  *// Can't call test() with w4 because it is not an object of a class that inherits A.*  *//* test(w4); *// Error!!! Unacceptable call to test()*  }} |

* In ***main()***, objects of type ***A***, ***B***, ***C***, and ***D*** are created. These are then used to create four ***Gen*** objects, one for each type.
* Finally, four calls to ***test()*** are made, with ***the last call commented*** out.
* The *first three calls* are valid because ***w***, ***w2***, and ***w3*** are ***Gen*** objects whose type is either ***A*** or a *subclass* of ***A***.
* The *last call* to test( ) is *illegal* because ***w4*** is an *object* of *type* ***D***, which is *not* *derived* from ***A***. Thus, the *bounded wildcard* in ***test()*** *will not accept* ***w4*** as an argument.

Note :

CAST one INSTANCE of a GENERIC CLASS into another: you can cast one instance of a generic class into another, but only if the two are otherwise compatible and their type arguments are the same. For example, assume a *generic class* called ***Gen*** :

**class** Gen<T> { */\* . . . \*/* }

Next, assume that **x** is declared as: **Gen<Integer>** x = **new** Gen<**Integer**>();

* Then, this cast is legal because ***x*** is an instance of ***Gen<Integer>*** : (**Gen<Integer>**) x *// legal*
* But, this cast is illegal because ***x*** is not an instance of ***Gen<Long>*** : (**Gen<Long>**) x *// illegal*

**9.4 Generic Methods and Generic Constructors**

Generic Methods: Methods inside a ***generic******class*** are automatically generic relative to the ***type******parameter***. However, it is possible to declare a generic method ***that uses one or more type parameters of its*** own. Furthermore, it is possible to create a generic method that is enclosed within a nongeneric class. The syntax for a generic method:

***<type-param-list> ret-type meth-name(param-list) {*** */\* ... \*/* ***}***

* In all cases, ***type-param-list*** is a ***comma-separated list*** of type parameters.
* Notice that for a generic method, the ***type parameter list*** precedes the ***return*** ***type***.
* Example 7: The following program declares a nongeneric class called ***GenericMethodDemo*** and a static generic method within that class called ***arraysEqual()***. This method determines if two arrays contain the same elements, in the same order. It can be used to compare any two arrays as long as the arrays are of the *same* or *compatible* *types* and the array *elements* are *comparable*.

class GenericMethodDemo { */\* Determine if the contents of two arrays are the same. \*/ //A generic method*

**static** <T **extends Comparable<T>**, V **extends** T> **boolean** arraysEqual(T[] x, V[] y) { *// Comparable<T> is a standard interface*

*/\* If array lengths differ, then the arrays differ.\*/* **if**(x.**length** != y.**length**) **return** **false**;

**for**(**int** i=0; i < x.**length**; i++) **if**(!x[i].**equals**(y[i])) **return false**; *// arrays differ*

*/\* contents of arrays are equivalent \*/* **return** **true**; }

|  |  |
| --- | --- |
| ***public static void main(String args[])*** { **Integer** nums[] = { 1, 2, 3, 4, 5 };  **Integer** nums2[] = { 1, 2, 3, 4, 5 };  **Integer** nums3[] = { 1, 2, 7, 4, 5 };  **Integer** nums4[] = { 1, 2, 7, 4, 5, 6 };  **if**(arraysEqual(nums, nums)) **System.out.println**("nums equals nums");  **if**(arraysEqual(nums, nums2)) **System.out.println**("nums equals nums2");  **if**(arraysEqual(nums, nums3)) **System.out.println**("nums equals nums3");  **if**(arraysEqual(nums, nums4)) **System.out.println**("nums equals nums4"); | **Double** dvals[] = { 1.1, 2.2, 3.3, 4.4, 5.5 }; *//array of Doubles*  *// This won't compile because nums and dvals are not of the same type.*  *//* **if**(arraysEqual(nums, dvals))  *//* **System.out.println**("nums equals dvals");  } } |
| output: **nums equals nums**  **nums equals nums2** |

* The generic method ***arraysEqual()*** is declared by following line:

**static** <T **extends** **Comparable<T>**, V **extends** T> **boolean** arraysEqual(T[] x, V[] y) {

* Here ***static <T extends Comparable<T>, V extends T>*** are the type parameters
* ***boolean*** is return type of the method.
* Also note that ***T*** ***extends*** ***Comparable<T>***. [See NOTE below]
* Next, notice that the type ***V*** is upper-bounded by ***T***. Thus, ***V*** must be either the *same* as type ***T*** or a *subclass* of ***T***. This relationship enforces that ***arraysEqual()*** can be called only with *arguments that are comparable with each other.*
* Also notice that ***arraysEqual()*** is ***static***, enabling it to be called independently of any object. *[Generic methods can be either static or nonstatic. There is no restriction in this regard.]*
* Notice ***arraysEqual()*** is called within ***main()*** by use of the normal call syntax, ***without specify type arguments***.
* This is because the types of the arguments are ***automatically discerned***, and the types of ***T*** and ***V*** are adjusted accordingly.

[Eg: In ***if(arraysEqual(nums, nums))*** the element type of the first argument is Integer, which causes Integer to be substituted for T. The element type of the second argument is also Integer, which makes Integer a substitute for V, too. Thus, the call to arraysEqual() is legal, and the two arrays can be *compared*.]

|  |  |
| --- | --- |
| * Now, notice the commented-out code: | *//* **if**(arraysEqual(nums, dvals))  *//* **System.out.println**("nums equals dvals"); |

* Since ***V*** *is bounded by* ***T*** in the *extends clause* in ***V*** ’s declaration, then ***V*** must be either type ***T*** or a subclass of ***T***. In this case, the *first* *argument* is of type ***Integer***, making T into Integer, but the *second* *argument* is of type ***Double***, which is not a subclass of Integer. This makes the call to ***arraysEqual()*** illegal, and results a ***compile-time type-mismatch error***.
* Generic Constructors: A constructor can be generic, even if its class is not.
* Example 8: For example, in the following program, the class ***Summation*** is *not generic*, but its *constructor is*.

|  |  |
| --- | --- |
| **class** Summation { **private** **int** sum;  <T **extends** **Number**> Summation(T arg){ */\* generic constructor \*/*  sum = 0;  **for**(**int** i=0; i <= arg.**intValue**(); i++) sum += i; }  int getSum() { return sum; } } | **class** GenConsDemo {  **public static void main(String args[])** {  **Summation** ob = **new** Summation(4.0);  **System.out.println**("Summation of 4.0 is " + ob.getSum());  }} |

* Because Summation( ) specifies a type parameter that is bounded by Number, a Summation object can be constructed using any numeric type, including Integer, Float, or Double. No matter what numeric type is used, its value is converted to Integer by calling intValue( ), and the summation is computed. Therefore, it is not necessary for the class Summation to be generic; only a generic constructor is needed.

Note

Comparable: ***Comparable*** is an interface declared in ***java.lang***. A class that implements ***Comparable*** defines objects that can be ordered. ***Thus, requiring an*** upper bound ***of Comparable ensures that arraysEqual() can be used only with objects that are*** capable of being compared***. Comparable is generic, and its*** type parameter ***specifies the*** type ofobjects ***that it compares.***

**9.5 Generic Interfaces**

In previous example we used standard generic interface ***Comparable<T>***. However, you can also define your own generic interface. ***Generic interfaces*** are specified just like ***generic classes***.

* The generalized syntax for a generic interface: ***interface interface-name<type-param-list> {*** // ...
* Here, ***type-param-list*** is a comma-separated list of *type parameters*.
* When a *generic interface* is implemented, you must specify the *type arguments* :

**class** *class-name***<type-param-list> implements interface-name<type-param-list> {**

* Example 9: Following creates an interface called ***Containment***, which can be implemented by classes that store one or more values. It declares a method called ***contains()*** that determines if a specified value is *contained* by the *invoking object*.

**interface** Containment<T> { **boolean** contains(**T** o); } *//A generic interface*

*// The contains() method tests if a specific item is contained within an object that implements Containment.*

*// Implement Containment using an array to hold the values.* ***Any class that implements a generic interface must itself be generic.***

**class** MyClass<T> **implements** **Containment<T>** { T[ ] arrayRef;

MyClass(T[] o) { arrayRef = o; }

*/\* Implement contains( ) \*/* **public** **boolean** contains(T o){ **for**(T x : arrayRef) **if**(x.**equals**(o)) **return** **true**;

**return** **false**; } }

|  |  |  |
| --- | --- | --- |
| **class** GenIFDemo { **public static void main(String args[])** {  **Integer** x[] = { 1, 2, 3 };  **MyClass<Integer>** ob = **new** MyClass<Integer>(x); | | *// The following is illegal because ob is an Integer Containment and 9.25 is a Double value.*  *//* **if**(ob.contains(9.25)) *// Illegal!*  *//* **System.out.println**("9.25 is in ob");  }} |
| output :  **2 is in ob**  **5 is NOT in ob** | **if**(ob.contains(2)) **System.out.println**("2 is in ob");  **else** **System.out.println**("2 is NOT in ob");  **if**(ob.contains(5)) **System.out.println**("5 is in ob");  **else** **System.out.println**("5 is NOT in ob"); |

* Notice that ***Containment*** is declared like this: ***interface Containment<T> {***
* The type parameter ***T*** specifies the *type of objects* that are *contained*.
* Next, ***Containment*** is implemented by ***MyClass***. Notice the declaration of ***MyClass*** :

***class MyClass<T> implements Containment<T> {***

* In general, if a class implements a *generic interface*, then that *class must also be generic*, at least to the extent that ***it takes a type parameter that is passed to the interface***. Eg: The following attempt to declare ***MyClass*** is in error :

**class** MyClass **implements** **Containment<T>** { *// Wrong!*

* It is wrong because ***MyClass*** doesn't declare a *type parameter*, i.e. there is no way to pass one to ***Containment***. Here, the *identifier* ***T*** is simply ***unknown*** and the ***compiler*** reports an ***error***.
* If a class implements a ***specific type of*** generic interface, then the implementing class does not need to be generic. Eg:

**class** MyClass **implements Containment<Double>** { *// OK: Double type of generic interface*

* Bounding a generic interface: The ***type parameter(s)*** specified by a generic interface can be bounded. This lets you ***limit the type of data for which the interface can be implemented***. For example, to limit ***Containment*** to numeric types, then declare:

**interface** Containment<T **extends Number**> {

* Remember, *any implementing class must pass* to ***Containment*** a type argument also having the same bound. For example, ***MyClass*** must be declared as:

**class** MyClass<**T extends Number**> **implements** **Containment**<**T**> {

* Notice the way the *type parameter* ***T*** is declared by ***MyClass*** and then passed to ***Containment***. Because ***Containment*** now requires a type that extends ***Number***, the implementing class (i.e. ***MyClass***) must specify the same bound.
* Furthermore, once this bound has been established, there is *no need to specify it again* in the *implements clause*. In fact, it would be wrong to do so. Once the *type parameter* has been established, it is simply passed to the *interface* without further modification. For example, this declaration is incorrect and won’t compile:

**class** MyClass<**T extends Number**> **implements** **Containment**<**T extends Number**>{ *// Wrong!*

**9.6 Raw Types and Legacy Code**

Prior to JDK 5, *generics code* was not supported. There need a way that *pre-generics code* must be able to work with ***generics***, and *generic code* must be able to work with ***pre-generics*** code.

* To handle the ***transition to generics***, Java allows a generic class to be ***used without any type arguments***. This creates a ***raw type*** for the class. This *raw type* is compatible with legacy (older-pre-generic) code, ***which has no knowledge of generics***.
* When **no** type argument is supplied to a generic class, a raw type is created.
* The main drawback to using the raw type is that the ***type safety of generics*** is lost.
* Example 10: Here is an example that shows a ***raw type*** in action:

|  |  |
| --- | --- |
| **class** Gen<T> { T ob; *// declare an object of type T*  Gen(T o) { ob = o; } *// Pass the constructor a reference to an object of type T*  T getob() { return ob; } *// Return ob.*  }  *// Demonstrate raw type.*  **class** RawDemo { **public static void main(String args[])** {  **Gen<Integer>** iOb = **new** Gen<Integer>(88); *// Create a Gen object for Integers.*  **Gen<String>** strOb = **new** Gen<String>("Generics Test"); *// Gen object for Strings.*  *// When no type argument is supplied to a generic class, a raw type is created.*  **Gen** raw = **new** Gen(**new** **Double**(8.6)); *// Create a raw-type Gen object & give it a Double.*  **double** d = (**Double**) raw.getob(); *// Cast here is necessary because type is unknown.*  **System.out.println**("value: " + d); | */\* The use of a raw type can lead to run-time exceptions. Here are some examples. \*/*  *//* **int** i = (**Integer**) raw.getob(); *// run-time error*  *// In this assignment raw types overrides type safety.*  strOb = raw; *// OK, but potentially wrong*  *//* **String** str = strOb.getob(); *// run-time error*  *// This assignment also overrides type safety.*  raw = iOb; *// OK, but potentially wrong*  *//* d = (**Double**) raw.getob(); *// run-time error*  }} |

* A raw type of the generic ***Gen*** class is created by the declaration: ***Gen raw = new Gen(new Double(98.6));***
* Notice that no type arguments are specified. In essence, this creates a ***Gen*** object whose type ***T*** is replaced by Object.
* BYPASSING **the** ***type-safety mechanism*** **using** Raw: A raw type is not ***type safe***. Thus, a variable of a *raw type* can be assigned a *reference* to any type of ***Gen*** *object*. The assignment of a raw reference to a generic reference bypasses the ***type-safety mechanism***.
* The *reverse* is also allowed, in which a variable of a specific ***Gen*** *type* can be assigned a *reference* to a raw ***Gen*** *object*.

[However, both operations are potentially unsafe because the type checking mechanism of generics is circumvented. Notice the COMMENTED lines at the end.]

* *//* **int** i = (**Integer**) raw.getob(); *// run-time error*
* In this statement, the value of ob inside raw is obtained, and this value is cast to Integer. The trouble is that raw contains a Double value, not an integer value. However, this cannot be detected at compile time because the type of raw is unknown. Thus, this statement fails at run time.
* strOb = raw; *// OK, but potentially wrong* and *//* **String** str = strOb.getob(); *// run-time error*
* The assignment itself is syntactically correct, but questionable. It assigns to ***strOb*** (a ***reference*** of type ***Gen<String>***) a ***reference*** to a raw ***Gen*** object. Because strOb is of type Gen<String>, it is assumed to contain a String. However, after the assignment, the object referred to by strOb contains a Double. At run time, when an attempt is made to assign the contents of strOb to str, a run-time error results because strOb now contains a Double.
* raw = iOb; *// OK, but potentially wrong* and *//* d = (**Double**) raw.getob(); *// run-time error*
* This sequence inverts the preceding case. Here, a generic reference is assigned to a raw reference variable. Although this is syntactically correct, it can lead to problems, as illustrated by the second line. In this case, raw now refers to an object that contains an Integer object, but the cast assumes that it contains a Double. This error cannot be prevented at compile time. Rather, it causes a run-time error.
* javac displays *unchecked warnings* when a raw type is used in a way that might violate type safety. Eg: following generate *unchecked warnings*:

|  |  |
| --- | --- |
| **Gen raw = new Gen(new Double(98.6));** | Use of Gen without a type argument that causes the warning. |
| **strOb = raw;** *// OK, but potentially wrong* | Assignment of a raw *reference* to a generic *variable* that generates the warning. |

One final point: You should limit the use of raw types to those cases in which you must mix legacy code with newer, generic code. Raw types are simply a transitional feature and not something that should be used for new code.

**9.7 TYPE INFERENCE using DIAMOND Operator <>**

Beginning with JDK 7, it is possible to shorten the syntax used to create an instance of a *generic* *type*. When *type inference* is used, the declaration syntax for a generic reference and instance creation has this general form [ Here **<>** is called the *diamond operator* ]:

***class\_name<type-arg-list> var\_name = new class\_name< >(cons-arg-list);***

* Here, the ***type argument list*** of the ***new*** clause is empty i.e. **<>** which is an *empty type argument list*.
* Consider the following portion of ***TwoGen*** class shown earlier. Notice that it uses *two generic types*.

**class** TwoGen<T, V>{ **T** ob1; **V** ob2;

TwoGen(**T** o1, **V** o2){ob1 = o1; ob2 = o2;}*// Pass the constructor a reference to an object of type T.*

*/\* . . . \*/* }

* To create an instance of ***TwoGen*** using the full-length syntax, use a statement similar to the following:

**TwoGen<Integer, String>** tgOb = **new** TwoGen<**Integer**, **String**>(42, "testing");

* Here, the *type arguments* (which are ***Integer*** and ***String***) are specified twice: first, when ***tgOb*** is declared, and second, when a ***TwoGen*** instance is created via ***new***.

[Since, in the new clause, the type of the type arguments can be readily inferred, there is really no reason that they need to be specified a second time. To address this situation, JDK 7 added a syntactic element that lets you avoid the second specification.]

* Using the Diamond operator '<>' : The preceding declaration can be rewritten as:

**TwoGen<Integer, String>** tgOb = **new** TwoGen**<>**(42, "testing"); *// Diamond operator used*

* Notice that the *instance creation* portion simply uses **< >**, which is an ***empty type argument list***. This is referred to as the diamond operator. It tells the ***compiler*** to ***infer*** the ***type arguments*** needed by the constructor in the ***new*** expression.

[Type-inference syntax is especially helpful for generic types that specify bounds.]

* Parameter passing using < > : ***Type inference*** can also be applied to *parameter passing*. For example, if the following method is added to ***TwoGen***:

**boolean** isSame(**TwoGen<T, V>** o) { **if**(ob1 == o.ob1 **&&** ob2 == o.ob2) **return** **true**;

**else** **return** **false**; }

Then this call is legal: **if**(tgOb.isSame(**new TwoGen<>**(42, "testing"))) **System.out.println**("Same");

[ In this case, the *type arguments* for the arguments passed to ***isSame()*** can be inferred from the parameters’ types. They don’t need to be specified again. ]

**9.8 Erasure**

Usually, it is not necessary for the programmer to know the details about how the Java compiler transforms a source code into object code. However, in the case of generics, some *general understanding* *of the process* is important because it explains why the *generic* *features* work as they do—and why their *behavior* is sometimes a bit *surprising*.

* ***Generic code*** had to be compatible with preexisting, ***nongeneric*** ***code***. Thus, any changes to the syntax of the *Java language*, or to the *JVM*, had to *avoid breaking older code*. This *issue is resolved* by the use of erasure when implements generics.
* How erasure works: When your Java code is compiled, all generic type information is removed (erased). This means replacing *type* *parameters* with their *bound* *type*, which is ***Object*** if no explicit bound is specified, and then applying the appropriate casts (as determined by the *type arguments*) to maintain ***type compatibility*** with the types specified by the ***type arguments***.
* The compiler also enforces this type compatibility. This approach to generics means that no type parameters exist at run time. They are simply a source-code mechanism.

**9.9 AMBIGUITY Errors and RESTRICTIONS on Generic Classes**

Ambiguity errors occur when erasure causes two ***seemingly (apparently) distinct generic declarations*** to resolve to the ***same erased type***, causing a conflict. Here is an example of ambiguity that involves method overloading:

**class** MyGenClass<T, V> { **T** ob1; **V** ob2;

*// Following two overloaded methods are ambiguous and will not compile.*

*/\* These two methods are inherently ambiguous.\*/* **void** set(T o) { ob1 = o; }

**void** set(V o) { ob2 = o; } }

* Notice, ***MyGenClass*** declares two generic types: ***T*** and ***V***. Inside ***MyGenClass***, an attempt is made to overload ***set()*** based on parameters of type ***T*** and ***V***. Looks reasonable because ***T*** and ***V*** appear to be *different types*. But, there are two ambiguity problems:
* As ***MyGenClass*** is written there is no requirement that **T** and **V** actually be different types. Eg: Construct a ***MyGenClass*** object as: **MyGenClass<String, String>** obj = **new** MyGenClass<**String**, **String**>()
* Here, both ***T*** and ***V*** will be replaced by ***String***. This makes both versions of ***set()*** *identical*, which is, of course, an *error*.
* The ***type erasure*** of ***set()*** effectively reduces both versions to: **void** set(**Object** o) { *// ...*
* The SOLUTION in this case is to use two separate method names rather than trying to overload ***set()***.
* Some Generic Restrictions: There are a few restrictions which involve creating objects of a *type parameter*, *static members*, *exceptions*, and *arrays*.
* Type Parameters Can’t Be Instantiated: It is not possible to create an instance of a type parameter. For example:

class Gen<T>{ **T** ob;

Gen() { ob = **new** T(); */\* Can't create an instance of T : Illegal!!!\*/* }

}

* Here, it is illegal to attempt to create an *instance* of **T**. The reason is: the *compiler* has no way to know *what type* *of object* to create. ***T*** is simply a ***placeholder***.
* Generic Exception Restriction: A generic class cannot extend ***Throwable***. i.e. Creation of generic exception classes isn't possible.
* Restrictions on Static Members: No ***static*** *member* can use a type parameter declared by the enclosing class. For example, both of the ***static*** members of this class are illegal:

**class** Wrong<T> { **static** **T** ob; *// Wrong, no static variables of type T.*

**static** **T** getob() { **return** ob; } *// Wrong, no static method can use T.*

}

* Although you can’t declare *static members* that use a *type parameter* declared by the enclosing class, you can declare static generic methods, which define their own *type parameters*.
* Generic Array Restrictions: There are two important generics restrictions that apply to arrays.
* First, you ***cannot*** instantiate ***an array*** whose element type is a ***type parameter***.
* Second, you ***cannot create an array*** of type-specific generic references. The following shows both situations:

|  |  |
| --- | --- |
| **class** Gen<T **extends** **Number**> {  **T** ob;  **T** vals[]; // OK  Gen(**T** o, **T[]** nums) { ob = o;  *//* vals = new T[10]; */\* ILLEGAL: can't create an array of T \*/*  vals = nums; */\* OK: can assign reference to existent array \*/* }  } | **class** GenArrays { **public static void main(String args[])** {  **Integer** n[] = { 1, 2, 3, 4, 5 };  **Gen<Integer>** iOb = **new** Gen<**Integer**>(50, n);  *// Can't create an array of type-specific generic references.*  *//* **Gen<Integer>** gens[] = new Gen<**Integer**>[10]; *// Wrong!*    **Gen<?>** gens[] = **new** Gen<**?**>[10]; *// OK: Leagal*  } } |

* It’s valid to declare a reference to an *array of type* ***T***, as this line does: **T vals[];** *// OK*
* But, cannot instantiate an array of T, as this commented-out line attempts: *//* vals = **new** T[10]; *// can't create an array of T*
* Reason you can’t create an array of T is: there is no way for the *compiler* to know what *type of array* to actually create.
* However, can pass a reference to a type-compatible array to ***Gen()*** when an object is created and assign that reference to ***vals*** as this line: **vals = nums;** *// OK to assign reference to existent array*
* This works because the array passed to ***Gen()*** has a *known type*, which will be the same type as ***T*** at the time of object creation.
* Inside ***main()***, notice that you can’t declare an array of references to a ***specific generic type***. i.e, following won’t compile.

*//* **Gen<Integer>** gens[] = **new** Gen<**Integer**>[10]; *// Wrong!*

* Advanced Study of Generics: To learn about how generics affect ***class hierarchies***, ***run-time type comparisons***, and ***overriding***, for example. Discussions of these and other topics are found in Java: The Complete Reference, Ninth Edition (Oracle Press/McGraw-Hill Professional, 2014).