Wildcard (Java)

From Wikipedia, the free encyclopedia

The **wildcard** `?` in Java is a special actual parameter for the instantiation of generic (parameterized) types. It can be used for the instantiation, not in the definition of a generic unit. Thus, a wildcard is a form of *use-site* variance annotation (contrast this with the *definition-site* variance annotations found in C# and Scala). This article summarizes the most important rules for its use.

Contents

Covariance for generic types

Unlike arrays (which are covariant in Java), different instantiations of a generic type are not compatible with each other, not even explicitly: With the declaration `Generic<Supertype> superGeneric; Generic<Subtype> subGeneric;` the compiler would report a conversion error for both castings `(Generic<Subtype>)superGeneric` and `(Generic<Supertype>)subGeneric`.

This incompatibility may be softened by the wildcard if `?` is used as an actual type parameter: `Generic<?>` is the abstract supertype for all instantiations of the generic type. It means, no objects of this type may be created, only variables. The usage of such a variable is to refer to instantiations of `Generic` with any actual type parameter.

Wildcard as parameter type

In the body of a generic unit, the (formal) type parameter is handled like its upper bound (expressed with **`extends`**; `Object` if not constrained). If the return type of a method is the type parameter, the result (e.g. of type `?`) can be referenced by a variable of the type of the upper bound (or `Object`). In the other direction,

the wildcard fits no other type, not even `Object`: If `?` has been applied as the formal type parameter of a method, no actual parameters can be passed to it. It can be called only by casting the wildcard reference:

```
class Generic <T extends UpperBound> {
    private T t;
    void write(T t) {
        this.t = t;
    }
    T read() {
        return t;
    }
}
...
Generic<?> wildcardReference = new Generic<>();
UpperBound ub = wildcardReference.read(); // Object would also be OK
wildcardReference.write(new Object()); // type error
wildcardReference.write(new UpperBound()); // type error
((Generic<UpperBound>)wildcardReference).write(new UpperBound()); // OK
```

Bounded Wildcards

A bounded wildcard is one with either an upper or a lower constraint. Not only the formal type parameters in the generic unit, but also the wildcard can be (further) constrained if one doesn't want to be compatible with all instantiations:

```
Generic<? extends SubtypeOfUpperBound> referenceConstrainedFromAbove;
```

This reference can hold any instantiation of `Generic` with an actual type parameter of `SubtypeOfUpperBound`'s subtype. A wildcard that does not have a constraint is effectively the same as one that has the constraint `extends Object`, since all types implicitly extend Object. A constraint with a lower bound

```
Generic<? super SubtypeOfUpperBound> referenceConstrainedFromBelow;
```

can hold instantiations of `Generic` with any supertype (e.g. `UpperBound`) of `SubtypeOfUpperBound`. (Such a wildcard still has an implicit upper bound of `Object`.)

It is even possible to constrain a reference's compatibility from both sides: from above by a generic class or method definition (`<SubtypeOfUpperBound extends UpperBound>`), or from below by the reference declaration (`<? super SubtypeOfUpperBound>`).

Object Creation with Wildcard

No objects may be created with a wildcard type parameter: `new Generic<?>()` is forbidden because `Generic<?>` is abstract. In practice, this is unnecessary because if one wanted to create an object that was assignable to a variable of type `Generic<?>`, one could simply use any arbitrary type (that falls within the constraints of the wildcard, if any) as the type parameter.

On the other hand, an array object that is an array of a parameterized type may be created only by an unconstrained (i.e. with a wildcard type parameter) type (and by no other instantiations) as the component type: `new Generic<?>[20]` is correct, while `new Generic<UpperBound>[20]` is prohibited.

An example of using a wildcard in List's instantiation is contained in the article Generics in Java.

Example: Lists

In the Java Collections Framework, the class `List<MyClass>` represents an ordered collection of objects of type `MyClass`. Upper bounds are specified using `extends`: A `List<? extends MyClass>` is a list of objects of some subclass of `MyClass`, i.e. any object in the list is guaranteed to be of type `MyClass`, so one can iterate over it using a variable of type `MyClass`

```
public void doSomething(List<? extends MyClass> list) {
  for(MyClass object : list) { // OK
    // do something
  }
}
```

However, it is not guaranteed that one can add any object of type `MyClass` to that list:

```
public void doSomething(List<? extends MyClass> list) {
  MyClass m = new MyClass();
  list.add(m); // Compile error
}
```

The converse is true for lower bounds, which are specified using `super`: A `List<? super MyClass>` is a list of objects of some superclass of `MyClass`, i.e. the list is guaranteed to be able to contain any object of type `MyClass`, so one can add any object of type `MyClass`:

```
public void doSomething(List<? super MyClass> list) {
  MyClass m = new MyClass();
  list.add(m); // OK
}
```

However, it is not guaranteed that one can iterate over that list using a variable of type `MyClass`:

```
public void doSomething(List<? super MyClass> list) {
  for(MyClass object : list) { // Compile error
    // do something
  }
}
```

In order to be able to do both add objects of type `MyClass` to the list and iterate over it using a variable of type `MyClass`, a `List<MyClass>` is needed, which is the only type of `List` that is both `List<? extends MyClass>` and `List<? super MyClass>`.

The mnemonics PECS (Producer Extends, Consumer Super) from the book **Effective Java** by Joshua Bloch gives an easy way to remember when to use wildcards (corresponding to Covariance and Contravariance) in Java.

See also

- Bounded quantification
- Covariance and contravariance (computer science)
- Generics in Java#Type wildcards section explains lower and upper wildcard bounds

References

- The Java Language Specification, Third Edition (Sun), ISBN 978-0-321-24678-3 http://java.sun.com/docs/books/jls/
- Java Tutorials, Lesson Generics http://download.oracle.com/javase/tutorial/java/generics/index.html
- Capturing Wildcards, http://bayou.io/draft/Capturing_Wildcards.html
- Typkompatibilität in Java http://public.beuth-hochschule.de/~solymosi/veroeff/typkompatibilitaet/Typkompatibilitaet.html#Joker (in German)

Categories: Java (programming language) | Polymorphism (computer science)

What is Reifiable Type in Java ?

http://docs.oracle.com/javase/specs/jls/se7/html/jls-4.html#jls-4.6

Because some type information is erased during compilation, not all types are available at run time. Types that are completely available at run time are known as *reifiable types*.

A type is *reifiable* if and only if one of the following holds:

- It refers to a non-generic class or interface type declaration.
- It is a parameterized type in which all type arguments are unbounded wildcards (§4.5.1).
- It is a raw type (§4.8).
- It is a primitive type (§4.2).
- It is an array type (§10.1) whose element type is reifiable.
- It is a nested type where, for each type T separated by a ".", T itself is reifiable.

  For example, if a generic class X<T> has a generic member class Y<U>, then the type X<?>.Y<?> is reifiable because X<?> is reifiable and Y<?> is reifiable. The type X<?>.Y<Object> is not reifiable because Y<Object> is not reifiable.

An intersection type is not reifiable.

The decision not to make all generic types reifiable is one of the most crucial, and controversial design decisions involving the type system of the Java programming language.

Ultimately, the most important motivation for this decision is compatibility with existing code. In a naive sense, the addition of new constructs such as generics has no implications for pre-existing code. The Java programming language, per se, is compatible with earlier versions as long as every program written in the previous versions retains its meaning in the new version. However, this notion, which may be termed language compatibility, is of purely theoretical interest. Real programs (even trivial ones, such as "Hello World") are composed of several compilation units, some of which are provided by the Java SE platform (such as elements of java.lang or java.util). In practice, then, the minimum requirement is platform compatibility - that any program written for the prior version of the Java SE platform continues to function unchanged in the new version.

One way to provide platform compatibility is to leave existing platform functionality unchanged, only adding new functionality. For example, rather than modify the existing Collections hierarchy in java.util, one might introduce a new library utilizing generics.

The disadvantages of such a scheme is that it is extremely difficult for pre-existing clients of the Collection library to migrate to the new library. Collections are used to exchange data between independently developed modules; if a vendor decides to switch to the new, generic, library, that vendor must also distribute two versions of their code, to be compatible with their clients. Libraries that are dependent on other vendors code cannot be modified to use generics until the supplier's library is updated. If two modules are mutually dependent, the changes must be made simultaneously.

Clearly, platform compatibility, as outlined above, does not provide a realistic path for adoption of a pervasive new feature such as generics. Therefore, the design of the generic type system seeks to support migration compatibiliy allows the

evolution of existing code to take advantage of generics without imposing dependencies between independently developed software modules.

The price of migration compatibility is that a full and sound reification of the generic type system is not possible, at least while the migration is taking place.

http://cs.uno.edu/~jaime/costOfErasure.pdf

REIFIABLE TYPES

Types that do not lose any information during type erasure are called reifiable types. Because some type information is erased during compilation, not all statically defined types are available at run time. Types that are completely available at runtime are known as reifiable types. The following are reifiable types: non-generic types, non-parameterized types, wildcard parameterized types, raw types, primitive types and arrays of reifiable types.

Array component type

The only allowed types for array creation are: primitive types, non-generic (or non-parameterized) reference types, unbounded wildcard instantiations, and raw types resulting from generic types.
Examples:

```
int[] table1 = new int[10];
String[] table2 = new String[10];
List<?>[] table3 = new DefaultList[10];

List [] table4 = new DefaultList[10];
```

What must be noticed is that the component type cannot be parameterized types. Because of type erasure, parameterized types do not have exact runtime type information. Thus, the array store check does not work because it uses the dynamic type information regarding the array's (non-exact) component type. Only arrays with an unbounded wildcard parameterized type as the
component type are permitted. More generally, only reifiable types are permitted as component type of arrays.
Also, although you can declare arrays using a type parameter T, or using parametric classes, instances of such arrays are not allowed and will produce compilation errors. Specifically,

```
1. public class MyContainer<T> {
2. private T[] elements;
3. private java.util.Collection<Integer> table;
4. public Container(int size){
5. elements = new E[size];
6. table = new ArrayList<Integer> [size];
7. }
8. ...
9. }
```

will produce "generic array creation" errors in the lines 5 and 6, where elements and table are created. Since arrays are used to implement collection of objects, we can get around it by creating the arrays having component Object:

```
public class MyContainer<T> {
private T[] elements;
...
public MyComp(int size){
elements = (T[])new Object[size];
}
...
}
```

The cast to T[] will produce an unchecked cast warning due to type erasure; but the specification of the List's add(T element) will guarantee that only elements of type T will be added to the list. Thus, we will need to live with this warning
when using arrays as data components in a generic type implementation.
Why does the code above not generate an ArrayStoreException when elements is accessed? After all, if T is instantiated with String you can't assign an array of Object to an array of String. Well, because generics are implemented by erasure, the type of elements is actually Object[], because Object is the erasure of T. This means that the class is really expecting elements to be an array of Object anyway, but the compiler does extra type checking to ensure that it contains only objects of type T.

http://stackoverflow.com/questions/18848885/why-following-types-are-reifiable-non-reifiable-in-java

Understand the meaning of this two terms.

Reifiable means whose type is fully available at run time means java compiler do not need any process of type erasure.
Non-Reifiable means java compiler needs type erasure process because type is not fully available.
A type is reifiable if it is one of the following:
1. A primitive type (such as int) :
Here think that when you write or use any int as a reference, do you think that compiler needs any process for identification for the type of int? no because int is int.... same for all primitive type

2. A nonparameterized class or interface type (such as Number, String, or Runnable)
same answer as i told in previous answer that compiler do not need any type erasure for Number, String, or Runnable.

3. A parameterized type in which all type arguments are unbounded wildcards (such as List<?>, ArrayList<?>, or Map<?, ?>)
All unbounded wildcard are accepted as reifiable type because it is already mention in definition of reifiable type, now it is up to the API developer why they consider it as a reifiable type.

4. A raw type (such as List, ArrayList, or Map) ::
same answer as first question

5. An array whose component type is reifiable(such as int[], Number[], List<?>[], List[], or int[][]) ::
same answer as first question

A type is not reifiable if it is one of the following:
6. A type variable(such as T) :
Because java can not identify the type of T, Compiler needs type erasure to identify the type.

7. A parameterized type with actual parameters (such as List<Number>, ArrayList<String>, or Map<String, Integer>):
Here all type is a generic type, at runtime compiler see List as List ... so as per definition of Non-refiable all these collection are consider as a non reifiable.

8. A parameterized type with a bound (such as List<? extends Number> or Comparable<? super String>).

http://www.angelikalanger.com/GenericsFAQ/FAQSections/TechnicalDetails.html#FAQ106

What is a reifiable type?

A type whose type information is fully available at runtime, that is, a type that does not lose information in the course of type erasure.

As a side effect of type erasure, some type information that is present in the source code is no longer available at runtime. For instance, parameterized types are translated to their corresponding raw type in a process called *type erasure* and lose the information regarding their type arguments.

For example, types such as `List<String>` or `Pair<? extends Number, ? extends Number>` are available to and used by the compiler in their exact form, including the type argument information. After type erasure, the virtual machine has only the raw types `List` and `Pair` available, which means that part of the type information is lost.

In contrast, non-parameterized types such as `java.util.Date` or `java.lang.Thread.State` are not affected by type erasure. Their type information remains exact, because they do not have type arguments.

Among the instantiations of a generic type only the unbounded wildcard instantiations, such as `Map<?,?>` or `Pair<?,?>`, are unaffected by type erasure. They do lose their type arguments, but since all type arguments are unbounded wildcards, no information is lost.

Types that do NOT lose any information during type erasure are called *reifiable types*. The term reifiable stems from *reification*. Reification means that type parameters and type arguments of generic types and methods are available at runtime. Java does not have such a runtime representation for type arguments because of type erasure. Consequently, the reifiable types in Java are only those types for which reification does not make a difference, that is, the types that do not need any runtime representation of type arguments.

The following types are reifiable:

- primitive types
- non-generic (or non-parameterized) reference types
- unbounded wildcard instantiations
- raw types
- arrays of any of the above

The non-reifiable types, which lose type information as a side effect of type erasure, are:

- instantiations of a generic type with at least one concrete type argument
- instantiations of a generic type with at least one bounded wildcard as type argument

Reifiable types are permitted in some places where non-reifiable types are disallowed. Reifiable types are permitted (and non-reifiable types are prohibited):

- as type in an `instanceof` expression
- as component type of an array