Covariance and contravariance (computer science)

From Wikipedia, the free encyclopedia

The type system of many programming languages support subtyping. For instance, if `Cat` is subtype of `Animal`, then an expression of type `Cat` can be used whenever an expression of type `Animal` could. **Variance** refers to how subtyping between more complex types (list of `Cat`s versus list of `Animal`s, function returning `Cat` versus function returning `Animal`, ...) relates to subtyping between their components. Depending on the variance of the type constructor, the subtyping relation may be either preserved, reversed, or ignored. For example, in C#:

- `IEnumerable<Cat>` is a subtype of `IEnumerable<Animal>`. The subtyping is preserved because `IEnumerable<T>` is **covariant** on `T`.
- `Action<Animal>` is a subtype of `Action<Cat>`. The subtyping is reversed because `Action<T>` is **contravariant** on `T`.
- Neither `IList<Cat>` nor `IList<Animal>` is a subtype of the other, because `IList<T>` is **invariant** on `T`.

The variance of a C# interface is determined by `in`/`out` annotations on its type parameters; the above interfaces are declared as `IEnumerable<out T>`, `Action<in T>`, and `IList<T>`. Types with more than one type parameter may specify different variances on each type parameter. For example, the delegate type `Func<in T, out TResult>` represents a function with a **contravariant** input parameter of type `T` and a **covariant** return value of type `TResult`.[1]

The typing rules for interface variance ensure type safety. For example, an `Action<T>` represents a first-class function expecting an argument of type `T`, and a function that can handle any type of animal can always be used instead of one that can only handle cats.

A programming language designer will consider variance when devising typing rules for e.g. arrays, inheritance, and generic datatypes. By making type constructors covariant or contravariant instead of invariant, more programs will be accepted as well-typed. On the other hand, programmers often find contravariance unintuitive, and accurately tracking variance to avoid runtime type errors can lead to complex typing rules. In order to keep the type system simple and allow useful programs, a language may treat a type constructor as invariant even if it would be safe to consider it variant, or treat it as covariant even when that can violate type safety.

Contents

Formal definition

Within the type system of a programming language, a typing rule or a type constructor is:

- *covariant* if it preserves the ordering of types (≤), which orders types from more specific to more generic;
- *contravariant* if it reverses this ordering;
- *bivariant* if both of these apply (i.e., both `I<A>` ≤ `I<B>` and `I<B>` ≤ `I<A>` at the same time);
- *invariant* or *nonvariant* if neither of these applies.

The article considers how this applies to some common type constructors.

Arrays

First consider the array type constructor: from the type `Animal` we can make the type `Animal[]` ("array of animals"). Should we treat this as

- Covariant: a `Cat[]` is an `Animal[]`
- Contravariant: an `Animal[]` is a `Cat[]`
- Invariant: an `Animal[]` is not a `Cat[]` and a `Cat[]` is not an `Animal[]`?

If we wish to avoid type errors, and the array supports both reading and writing elements, then only the third choice is safe. Clearly, not every `Animal[]` can be treated as if it were a `Cat[]`, since a client reading from the array will expect a `Cat`, but an `Animal[]` may contain e.g. a `Dog`. So the contravariant rule is not safe.

Conversely, a `Cat[]` cannot be treated as an `Animal[]`. It should always be possible to put a `Dog` into an `Animal[]`. With covariant arrays this cannot be guaranteed to be safe, since the backing store might actually be an array of cats. So the covariant rule is also not safe—the array constructor should be *invariant*. Note that this is only an issue for mutable arrays; the covariant rule is safe for immutable (read-only) arrays.

This illustrates a general phenomenon. Read-only data types (sources) can be covariant; write-only data types (sinks) can be contravariant. Mutable data types which act as both sources and sinks should be invariant.

Covariant arrays in Java and C#

Early versions of Java and C# did not include generics, also termed parametric polymorphism. In such a setting, making arrays invariant rules out useful polymorphic programs.

For example, consider writing a function to shuffle an array, or a function that tests two arrays for equality using the `Object (https://docs.oracle.com/javase/8/docs/api/java/lang/Object.html).equals` method on the elements. The implementation does not depend on the exact type of element stored in the array, so it should be possible to write a single function that works on all types of arrays. It is easy to implement functions of type

```
boolean equalArrays(Object[] a1, Object[] a2);
void shuffleArray(Object[] a);
```

However, if array types were treated as invariant, it would only be possible to call these functions on an array of exactly the type `Object[]`. One could not, for example, shuffle an array of strings.

Therefore, both Java and C# treat array types covariantly. For instance, in C# `string[]` is a subtype of `object[]`, and in Java `String[]` is a subtype of `Object[]`.

As discussed above, covariant arrays lead to problems with writes into the array. Java and C# deal with this by marking each array object with a type when it is created. Each time a value is stored into an array, the execution environment will check that the run-time type of the value is equal to the run-time type of the

array. If there is a mismatch, an `ArrayStoreException` (Java) or `ArrayTypeMismatchException` (C#) is thrown:

```
// a is a single-element array of String
String[] a = new String[1];

// b is an array of Object
Object[] b = a;

// Assign an Integer to b. This would be possible if b really were
// an array of Object, but since it really is an array of String,
// we will get a java.lang.ArrayStoreException.
b[0] = 1;
```

In the above example, one can *read* from the array (b) safely. It is only trying to *write* to the array that can lead to trouble.

One drawback to this approach is that it leaves the possibility of a run-time error that a stricter type system could have caught at compile-time. Also, it hurts performance because each write into an array requires an additional run-time check.

With the addition of generics, Java and C# now offer ways to write this kind of polymorphic functions without relying on covariance. The array comparison and shuffling functions can be given the parameterized types

```
<T> boolean equalArrays(T[] a1, T[] a2);
<T> void shuffleArray(T[] a);
```

Alternatively, to enforce that a C# method accesses a collection in a read-only way, one can use the interface `IEnumerable<object>` instead of passing it an array `object[]`.

Function types

Languages with first-class functions have function types like "a function expecting a Cat and returning an Animal" (written `Cat -> Animal` in OCaml syntax or `Func<Cat,Animal>` in C# syntax).

Those languages also need to specify when one function type is a subtype of another—that is, when it is safe to use a function of one type in a context that expects a function of a different type. It is safe to substitute a function *f* for a function *g* if *f* accepts a more general type of arguments and returns a more specific type than *g*. For example, a function of type `Cat->Cat` can safely be used wherever a `Cat->Animal` was expected, and likewise a function of type `Animal->Animal` can be used wherever a `Cat->Animal` was expected. (One can compare this to the robustness principle of communication: "be liberal in what you accept and conservative in what you produce"). The general rule is

$$S_1 \to S_2 \leq T_1 \to T_2 \text{ if } T_1 \leq S_1 \text{ and } S_2 \leq T_2.$$

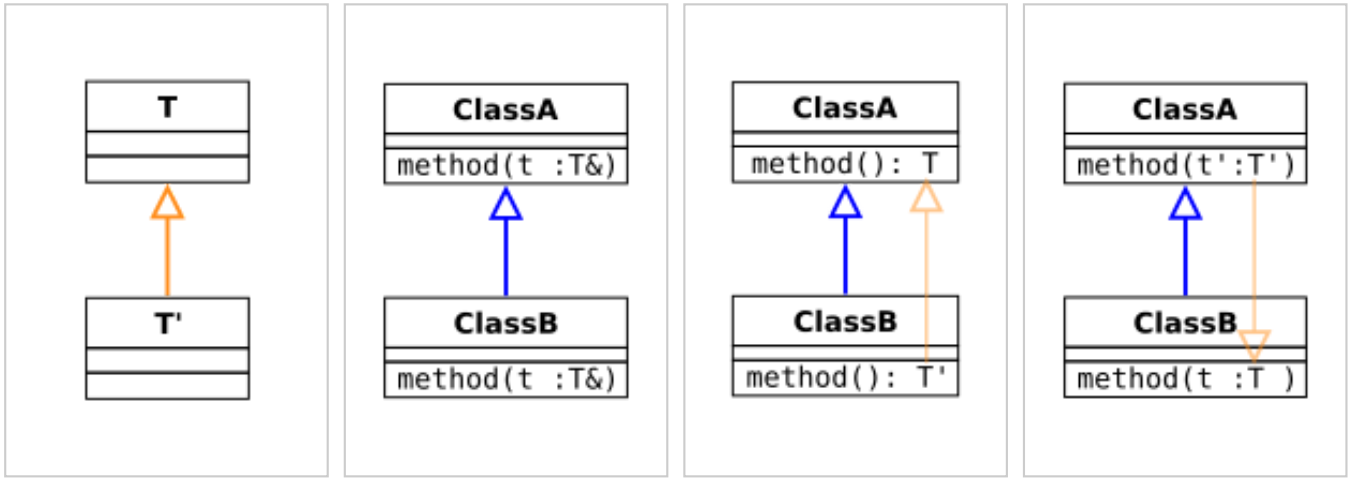
In other words, the → type constructor is *contravariant in the input type* and *covariant in the output type*. This rule was first stated formally by John C. Reynolds,[2] and further popularized in a paper by Luca Cardelli.[3]

When dealing with functions that take functions as arguments, this rule can be applied several times. For example, by applying the rule twice, we see that $(A' \to B) \to B \leq (A \to B) \to B$ if $A' \leq A$. In other words, the type $(A \to B) \to B$ is *covariant* in the A position. For complicated types it can be confusing to mentally trace why a given type specialization is or isn't type-safe, but it is easy to calculate which positions are co- and contravariant: a position is covariant if it is on the left side of an even number of arrows applying to it.

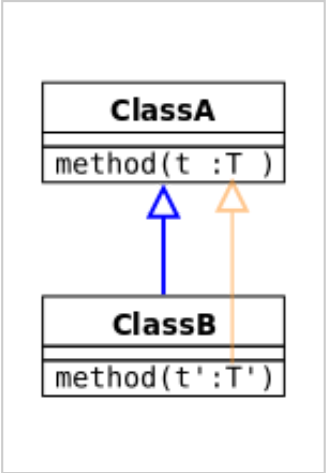
Inheritance in object-oriented languages

When a subclass overrides a method in a superclass, the compiler must check that the overriding method has the right type. While some languages require that the type exactly matches the type in the superclass (invariance), it is also type safe to allow the overriding method to have a "better" type. By the usual subtyping rule for function types, this means that the overriding method should return a more specific type (return type covariance), and accept a more general argument (argument type contravariance). In UML notation, the possibilities are as follows:

Variance and method overriding: overview



Subtyping of the argument/return type of the method.	*Invariance*. The signature of the overriding method is unchanged.	*Covariant return type*. The subtyping relation is in the same direction as the relation between ClassA and ClassB.	*Contravariant argument type*. The subtyping relation is in the opposite direction to the relation between ClassA and ClassB.



Covariant argument type. Not type safe.

For a concrete example, suppose we are writing a class to model an animal shelter. We assume that `Cat` is a subclass of `Animal`, and that we have a base class (using Java syntax)

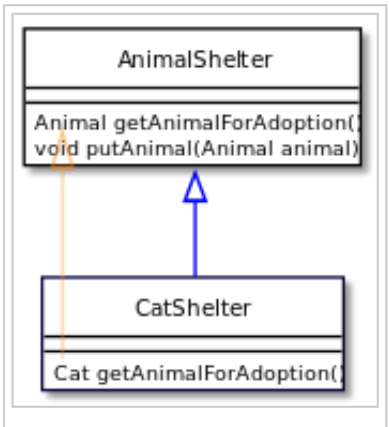
```
class AnimalShelter {
    Animal getAnimalForAdoption() {
      ...
    }

    void putAnimal(Animal animal) {
      ...
    }
}
```

Now the question is: if we subclass `AnimalShelter`, what types are we allowed to give to `getAnimalForAdoption` and `putAnimal`?

Covariant method return type

In a language which allows covariant return types, a derived class can override the `getAnimalForAdoption` method to return a more specific type:

```
class CatShelter extends AnimalShelter {
    Cat getAnimalForAdoption() {
        return new Cat();
    }
}
```

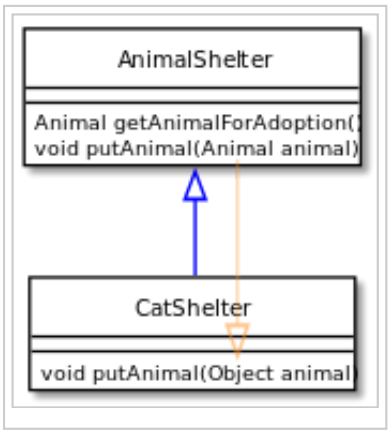


Among mainstream OO languages, Java and C++ support covariant return types, while C# does not. Adding the covariant return type was one of the first modifications of the C++ language approved by the standards committee in 1998.[4] Scala and D also support covariant return types.

Contravariant method argument type

Similarly, it is type safe to allow an overriding method to accept a more general argument than the method in the base class:

```
class CatShelter extends AnimalShelter {
    void putAnimal(Object animal) {
        ...
    }
}
```



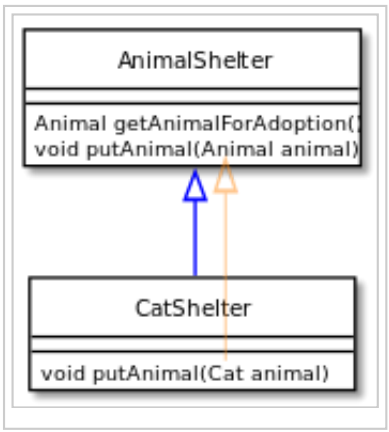
Not many object-oriented languages actually allow this. C++ and Java would interpret this as an unrelated method with an overloaded name.

However, Sather supports both covariance and contravariance. Calling convention for overridden methods are covariant with *out* arguments and return values, and contravariant with normal arguments (with the mode *in*).

Covariant method argument type

Uniquely among mainstream languages, Eiffel allows the arguments of an overriding method to have a *more* specific type than the method in the superclass (argument type covariance). Thus, the Eiffel version of the following code would type check, with `putAnimal` overriding the method in the base class:

```
class CatShelter extends AnimalShelter {
    void putAnimal(Cat animal) {
        ...
    }
}
```



This is not type safe. By up-casting a `CatShelter` to an `AnimalShelter`, one can try to place a dog in a cat shelter. That does not meet `CatShelter` argument restrictions. The lack of type safety (known as the "catcall problem" in the Eiffel community) has been a long-standing issue. Over the years, various combinations of global static analysis, local static analysis, and new language features have been proposed to remedy it,[5] [6] and these have been implemented in some Eiffel compilers.

Despite the type safety problem, the Eiffel designers consider covariant argument types crucial for modeling real world requirements.[6] The cat shelter illustrates a common phenomenon: it is *a kind of* animal shelter but has *additional restrictions*, and it seems reasonable to use inheritance and restricted argument types to model this. In proposing this use of inheritance, the Eiffel designers reject the Liskov substitution principle, which states that objects of subclasses should always be less restricted than objects of their superclass.

Another example where covariant arguments seem helpful is so-called binary methods, i.e. methods where the argument is expected to be of the same type as the object the method is called on. An example is the `compareTo` method: `a.compareTo(b)` checks whether `a` comes before or after `b` in some ordering, but the

way to compare, say, two rational numbers will be different from the way to compare two strings. Other common examples of binary methods include equality tests, arithmetic operations, and set operations like subset and union.

In older versions of Java, the comparison method was specified as an interface `Comparable`:

```
interface Comparable {
    int compareTo(Object o);
}
```

The drawback of this is that the method is specified to take an argument of type `Object`. A typical implementation would first down-cast this argument (throwing an error if it is not of the expected type):

```
class RationalNumber implements Comparable {
    int numerator;
    int denominator;

    ...

    public int compareTo(Object other) {
        RationalNumber otherNum = (RationalNumber)other;
        return Integer.compare(numerator*otherNum.denominator,
                               otherNum.numerator*denominator);
    }
}
```

In a language with covariant arguments, the argument to compareTo could be directly given the desired type `RationalNumber`, hiding the typecast. (Of course, this would still give a runtime error if `compareTo` was then called on e.g. a `String`).

Avoiding the need for covariant argument types

Other language features can provide the apparent benefits of covariant arguments while preserving Liskov substitutability.

In a language with *generics* (a.k.a. parametric polymorphism) and bounded quantification, the previous examples can be written in a type-safe way .[7] Instead of defining `AnimalShelter`, we define a parameterized class `Shelter<T>`. (One drawback of this is that the implementer of the base class needs to foresee which types will need to be specialized in the subclasses).

```
class Shelter<T extends Animal> {
    T getAnimalForAdoption() {
      ...
    }

    void putAnimal(T animal) {
      ...
    }
}


class CatShelter extends Shelter<Cat> {
```

```
    Cat getAnimalForAdoption() {
      ...
    }

    void putAnimal(Cat animal) {
      ...
    }
}
```

Similarly, in recent versions of Java the `Comparable` interface has been parameterized, which allows the downcast to be omitted in a type-safe way:

```
class RationalNumber implements Comparable<RationalNumber> {
    int numerator;
    int denominator;

    ...

    public int compareTo(RationalNumber otherNum) {
        return Integer.compare(numerator*otherNum.denominator,
                               otherNum.numerator*denominator);
    }
}
```

Another language feature that can help is *multiple dispatch*. One reason that binary methods are awkward to write is that in a call like `a.compareTo(b)`, selecting the correct implementation of `compareTo` really depends on the runtime type of both `a` and `b`, but in a conventional OO language only the runtime type of `a` is taken into account. In a language with Common Lisp Object System (CLOS)-style multiple dispatch, the comparison method could be written as a generic function where both arguments are used for method selection.

Giuseppe Castagna[8] observed that in a typed language with multiple dispatch, a generic function can have some arguments which control dispatch and some "left-over" arguments which do not. Because the method selection rule chooses the most specific applicable method, if a method overrides another method, then the overriding method will have more specific types for the controlling arguments. On the other hand, to ensure type safety the language still must require the left-over arguments to be at least as general. Using the previous terminology, types used for runtime method selection are covariant while types not used for runtime method selection of the method are contravariant. Conventional single-dispatch languages like Java also obey this rule: there only one argument is used for method selection (the receiver object, passed along to a method as the hidden argument **`this`**), and indeed the type of **`this`** is more specialized inside overriding methods than in the superclass.

Castagna suggests that examples where covariant argument types are superior, particularly binary methods, should be handled using multiple dispatch which is naturally covariant. Unfortunately, most programming languages do not support multiple dispatch.

Summary of variance and inheritance

The following table summarizes the rules for overriding methods in the languages discussed above.

	Argument type	Return type
C++ (since 1998), Java (since J2SE 5.0), Scala, D	Invariant	Covariant
C#	Invariant	Invariant
Sather	Contravariant	Covariant
Eiffel	Covariant	Covariant

Generic types

In programming languages that support generics (a.k.a. parametric polymorphism), the programmer can extend the type system with new constructors. For example, a C# interface like `IList<T>` makes it possible to construct new types like `IList<Animal>` or `IList<Cat>`. The question then arises what the variance of these type constructors should be.

There are two main approaches. In languages with *declaration-site variance annotations* (e.g., C#), the programmer annotates the definition of a generic type with the intended variance of its type parameters. With *use-site variance annotations* (e.g., Java), the programmer instead annotates the places where a generic type is instantiated.

Declaration-site variance annotations

The most popular languages with declaration-site variance annotations are C# (using the keywords **`out`** and **`in`**), and Scala and OCaml (using the keywords `+` and `-`). C# only allows variance annotations for interface types, while Scala and OCaml allows them for both interface types and concrete data types.

Interfaces

In C#, each type parameter of a generic interface can be marked covariant (**`out`**), contravariant (**`in`**), or invariant (no annotation). For example, we can define an interface `IEnumerator<T>` of read-only iterators, and declare it to be covariant (out) in its type parameter.

```
interface IEnumerator<out T>
{
    T Current { get; }
    bool MoveNext();
}
```

With this declaration, `IEnumerator` will be treated as covariant in its type argument, e.g. `IEnumerator<Cat>` is a subtype of `IEnumerator<Animal>`.

The typechecker enforces that each method declaration in an interface only mentions the type parameters in a way consistent with the **`in`**/**`out`** annotations. That is, a parameter that was declared covariant must not occur in any contravariant positions (where a position is contravariant if it occurs under an odd number of contravariant type constructors). The precise rule[9][10] is that the return types of all methods in the interface must be *valid covariantly* and all the method argument types must be *valid contravariantly*, where *valid S-ly* is defined as follows:

- Non-generic types (classes, structs, enums, etc.) are valid both co- and contravariantly.
- A type argument `T` is valid covariantly if it was not marked **`in`**, and valid contravariantly if it was not marked **`out`**
- An array type `A[]` is valid S-ly if `A` is. (This is because C# has covariant arrays).
- A generic type `G<A1,A2,...,An>` is valid S-ly if for each argument `Ai`,
  - Ai is valid S-ly, and the *i*th parameter to `G` is declared covariant, or
  - Ai is valid (not S)-ly, and the *i*th parameter to `G` is declared contravariant, or
  - Ai is valid both covariantly and contravariantly, and the *i*th parameter to `G` is declared invariant.

As an example of how these rules apply, consider the `IList<T>` interface.

```
interface IList<T>
{
void Insert(int index, T item);
IEnumerator<T> GetEnumerator();
}
```

The argument type `T` of `Insert` must be valid contravariantly, i.e. the type parameter `T` must not be tagged **`out`**. Similarly, the result type `IEnumerator<T>` of `GetEnumerator` must be valid covariantly, i.e. (since `IEnumerator` is a covariant interface) the type `T` must be valid covariantly, i.e. the type parameter `T` must not be tagged **`in`**. This shows that the interface `IList` is not allowed to be marked either co- or contravariant.

In the common case of a generic data structure such as `IList`, these restrictions mean that an **`out`** parameter can only be used for methods getting data out of the structure, and an **`in`** parameter can only be used for methods putting data into the structure, hence the choice of keywords.

Data

C# allows variance annotations on the parameters of interfaces, but not the parameters of classes. Because fields in C# classes are always mutable, variantly parameterized classes in C# would not be very useful. But languages which emphasize immutable data can make good use of covariant data types. For example, both in Scala and OCaml the immutable list type is covariant: **`List`**`[Cat]` is a subtype of **`List`**`[Animal]`.

Scala's rules for checking variance annotations are essentially the same as C#'s. However, there are some idioms that apply to immutable datastructures in particular. They are illustrated by the following (excerpt from the) definition of the **`List`**`[A]` class.

```
sealed abstract class List[+A] extends AbstractSeq[A] {
  def head: A
  def tail: List[A]

  /** Adds an element at the beginning of this list. */
  def ::[B >: A] (x: B): List[B] =
    new scala.collection.immutable.::(x, this)

  ...
}
```

First, class members that have a variant type must be immutable. Here, `head` has the type `A`, which was declared covariant (`+`), and indeed `head` was declared as a method (**`def`**). Trying to declare it as a mutable field (**`var`**) would be rejected as type error.

Second, even if a data structure is immutable, it will often have methods where the parameter type occurs contravariantly. For example, consider the method `::` which adds an element to the front of a list. (The implementation works by creating a new object of the similarly-named *class* `::`, the class of nonempty lists). The most obvious type to give it would be

```
def :: (x: A): List[A]
```

However, this would be a type error, because the covariant parameter `A` appears in a contravariant position (as a function argument). But there is a trick to get around this problem. We give `::` a more general type, which allows adding an element of any type `B` as long as `B` is a supertype of `A`. Note that this relies on **`List`** being covariant, since **`this`** has type **`List`**`[A]` and we treat it as having type **`List`**`[B]`. At first glance it may not be obvious that the generalized type is sound, but if the programmer starts out with the simpler type declaration, the type errors will point out the place that needs to be generalized.

Inferring variance

It is possible to design a type system where the compiler automatically infers the best possible variance annotations for all datatype parameters.[11] However, the analysis can get complex for several reasons. First, the analysis is nonlocal since the variance of an interface `I` depends the variance of all interfaces that `I` mentions. Second, in order to get unique best solutions the type system must allow *bivariant* parameters (which are simultaneously co- and contravariant). And finally, the variance of type parameters should arguably be a deliberate choice by the designer of an interface, not something that just happens.

For these reasons[12] most languages do very little variance inference. C# and Scala do not infer any variance annotations at all. OCaml can infer the variance of parameterized concrete datatypes, but the programmer must explicitly specify the variance of abstract types (interfaces).

For example, consider an OCaml datatype **`T`** which wraps a function

```
type ('a, 'b) t = T of ('a -> 'b)
```

The compiler will automatically infer that **`T`** is contravariant in the first parameter, and covariant in the second. The programmer can also provide explicit annotations, which the compiler will check are satisfied. Thus the following declaration is equivalent to the previous one:

```
type (-'a, +'b) t = T of ('a -> 'b)
```

Explicit annotations in OCaml become useful when specifying interfaces. For example, the standard library interface **`Map.S`** for association tables include an annotation saying that the map type constructor is covariant in the result type.

```
module type S =
  sig
    type key
    type (+'a) t
    val empty: 'a t
    val mem: key -> 'a t -> bool
    ...
  end
```

This ensures that e.g. `cat IntMap.t` is a subtype of `animal IntMap.t`.

Use-site variance annotations (wildcards)

One drawback of the declaration-site approach is that many interface types must be made invariant. For example, we saw above that `IList` needed to be invariant, because it contained both `Insert` and `GetEnumerator`. In order to expose more variance, the API designer could provide additional interfaces which provide subsets of the available methods (e.g. an "insert-only list" which only provides `Insert`). However this quickly becomes unwieldy.

Use-site variance annotations aim to give users of a class more opportunities for subtyping without requiring the designer of the class to define multiple interfaces with different variance. Instead, each time a class or interface is used in a type declaration, the programmer can indicate that only a subset of the methods will be used. In effect, each definition of a class also makes available interfaces for the covariant and contravariant *parts* of that class. Therefore the designer of the class no longer needs to take variance into account, increasing re-usability.

Java provides use-site variance annotations through wildcards, a restricted form of bounded existential types. A parameterized type can be instantiated by a wildcard `?` together with an upper or lower bound, e.g. `List<? extends Animal>` or `List<? super Animal>`. (A an unbounded wildcard like `List<?>` is equivalent to `List<? extends Object>`). Such a type represents `List<X>` for some unknown type `X` which satisfies the bound. For example, if `l` has type `List<? extends Animal>`, then the typechecker will accept

```
   Animal a = l.get(3);
```

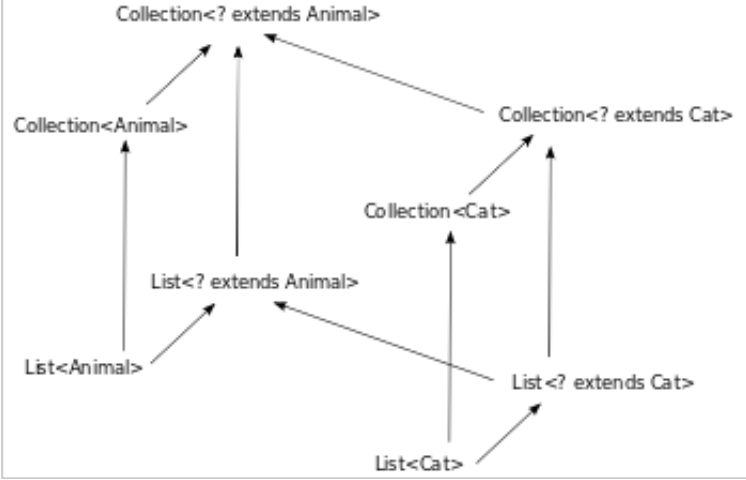
because the type `X` is known to be a subtype of `Animal`, but

```
   l.add(new Animal())
```

will be rejected as a type error since an `Animal` is not necessarily an `X`. In general, given some interface `I<T>`, a reference to a `I<? extends A>` forbids using methods from the interface where `T` occurs contravariantly in the type of the method. Conversely, if `l` had type `List<? super Animal>` one could call `l.add` but not `l.get`.

While plain generic types in Java are invariant (e.g. there is no subtyping relationship between `List<Cat>` and `List<Animal>`), wildcard types can be made more specific by specifying a tighter bound, for example `List<? extends Cat>` is a subtype of `List<? extends Animal>`. This shows that wildcard types are

covariant in their upper bounds (and also *contravariant in their lower bounds*). In total, given a wildcard type like `C<? extends T>`, there are three ways to form a subtype: by specializing the class `C`, by specifying a tighter bound `T`, or by replacing the wildcard `?` by a specific type (see figure).



Wildcard subtyping in Java can be visualized as a cube.

By combining two steps of subtyping, it is therefore possible to e.g. pass an argument of type `List<Cat>` to a method expecting a `List<? extends Animal>`. This is exactly the kind of programs that covariant interface types allow. The type `List<? extends Animal>` acts as an interface type containing only the covariant methods of `List<T>`, but the implementer of `List<T>` did not have to define it ahead of time. This is use-site variance.

In the common case of a generic data structure `IList`, covariant parameters are used for methods getting data out of the structure, and contravariant parameters for methods putting data into the structure. The mnemonic for Producer Extends, Consumer Super (PECS), from the book *Effective Java* by Joshua Bloch gives an easy way to remember when to use covariance and contravariance.

Wildcards are flexible, but there is a drawback. While use-site variance means that API designers need not consider variance of type parameters to interfaces, they must often instead use more complicated method signatures. A common example involves the `Comparable (https://docs.oracle.com/javase/8/docs/api/java/lang/Comparable.html)` interface. Suppose we want to write a function that finds the biggest element in a collection. The elements need to implement the `compareTo` method, so a first try might be

```
<T extends Comparable<T>>  T max(Collection<T> coll);
```

However, this type is not general enough—one can find the max of a `Collection<Calendar>`, but not a `Collection<GregorianCalendar>`. The problem is that `GregorianCalendar (https://docs.oracle.com/javase/8/docs/api/java/util/GregorianCalendar.html)` does not implement `Comparable<GregorianCalendar>`, but instead the (better) interface `Comparable<Calendar>`. In Java, unlike in C#, `Comparable<Calendar>` is not considered a subtype of `Comparable<GregorianCalendar>`. Instead the type of `max` has to be modified:

```
<T extends Comparable<? super T>> T max(Collection<T> coll);
```

The bounded wildcard `? super T` conveys the information that `max` calls only contravariant methods from the `Comparable` interface. This particular example is frustrating because *all* the methods in `Comparable` are contravariant, so that condition is trivially true. A declaration-site system could handle this example with less clutter by annotating only the definition of `Comparable`.

Comparing declaration-site and use-site annotations

Use-site variance annotations provide additional flexibility, allowing more programs to type-check. However, they have been criticized for the complexity they add to the language, leading to complicated type signatures and error messages.

One way to assess whether the extra flexibility is useful is to see if it is used in existing programs. A survey of a large set of Java libraries[11] found that 39% of wildcard annotations could have been directly replaced by a declaration-site annotations. Thus the remaining 61% is an indication on places where Java benefits from having the use-site system available.

In a declaration-site language, libraries must either expose less variance, or define more interfaces. For example, the Scala Collections library defines three separate interfaces for classes which employ covariance: a covariant base interface containing common methods, an invariant mutable version which adds side-effecting methods, and a covariant immutable version which may specialize the inherited implementations to exploit structural sharing.[13] This design works well with declaration-site annotations, but the large number of interfaces carry a complexity cost for clients of the library. And modifying the library interface may not be an option—in particular, one goal when adding generics to Java was to maintain binary backwards compatibility.

On the other hand, Java wildcards are themselves complex. In a conference presentation[14] Joshua Bloch criticized them as being too hard to understand and use, stating that when adding support for closures "we simply cannot afford another *wildcards*". Early versions of Scala used use-site variance annotations but programmers found them difficult to use in practice, while declaration-site annotations were found to be very helpful when designing classes.[15] Later versions of Scala added Java-style existential types and wildcards; however, according to Martin Odersky, if there were no need for interoperability with Java then these would probably not have been included.[16]

Ross Tate argues[17] that part of the complexity of Java wildcards is due to the decision to encode use-site variance using a form of existential types. The original proposals[18] [19] used special-purpose syntax for variance annotations, writing `List<+Animal>` instead of Java's more verbose `List<? extends Animal>`.

Since wildcards are a form of existential types they can be used for more things than just variance. A type like `List<?>` ("some type of list") lets objects be passed to methods or stored in fields without exactly specifying their type parameters. This is particularly valuable for classes such as `Class (https://docs.oracle.com/javase/8/docs/api/java/lang/Class.html)` where most of the methods do not mention the type parameter.

However, type inference for existential types is a difficult problem. For the compiler implementer, Java wildcards raise issues with type checker termination, type argument inference, and ambiguous programs.[20] For the programmer, it leads to complicated type error messages. Java typechecks wildcard types by replacing the wildcards with fresh type variables (so-called *capture conversion*). This can make error messages harder to read, because they refer to type variables that the programmer did not directly write. For example, trying to add a `Cat` to a `List<? extends Animal>` will give an error like

```
method List.add(capture#1) is not applicable
```

```
  (actual argument Cat cannot be converted to capture#1 by method invocation conversion)
where capture#1 is a fresh type-variable:
  capture#1 extends Animal from capture of ? extends Animal
```

Since both declaration-site and use-site annotations can be useful, some type system provide both.[11][17]

Origin of the term *covariance*

These terms come from the notion of covariant and contravariant functors in category theory. Consider the category C whose objects are types and whose morphisms represent the subtype relationship ≤. (This is an example of how any partially ordered set can be considered as a category). Then for example the function type constructor takes two types *p* and *r* and creates a new type $p \to r$; so it takes objects in C^2 to objects in C. By the subtyping rule for function types this operation reverses ≤ for the first argument and preserves it for the second, so it is a contravariant functor in the first argument and a covariant functor in the second.

See also

- Polymorphism (computer science)
- Inheritance (computer science)

References

1. Func<T, TResult> Delegate (http://msdn.microsoft.com/en-us/library/bb549151.aspx) - MSDN Documentation
2. John C. Reynolds (1981). *The Essence of Algol*. Symposium on Algorithmic Languages. North-Holland.
3. Luca Cardelli (1984). *A semantics of multiple inheritance* (PDF). Semantics of Data Types (International Symposium Sophia-Antipolis, France, June 27 – 29, 1984). Lecture Notes in Computer Science. Springer. doi:10.1007/3-540-13346-1_2.(Longer version in Information and Computation, 76(2/3): 138-164, February 1988.)
4. Allison, Chuck. "What's New in Standard C++?".
5. Bertrand Meyer (October 1995). "Static Typing" (PDF). *OOPSLA 95 (Object-Oriented Programming, Systems, Languages and Applications), Atlanta, 1995*.
6. Howard, Mark; Bezault, Eric; Meyer, Bertrand; Colnet, Dominique; Stapf, Emmanuel; Arnout, Karine; Keller, Markus (April 2003). "Type-safe covariance: Competent compilers can catch all catcalls" (PDF). Retrieved 23 May 2013.
7. Franz Weber (1992). "Getting Class Correctness and System Correctness Equivalent - How to Get Covariance Right". *TOOLS 8 (8th conference on Technology of Object-Oriented Languages and Systems), Dortmund, 1992*.
8. Giuseppe Castagna, Covariance and contravariance: conflict without a cause (http://portal.acm.org/citation.cfm?id=203096&dl=ACM&coll=&CFID=15151515&CFTOKEN=6184618), ACM Transactions on Programming Languages and Systems (TOPLAS), Volume 17, Issue 3, May 1995, pages 431-447.
9. Eric Lippert (3 December 2009). "Exact rules for variance validity". Retrieved July 2013.
10. Section II.9.7 in ECMA International Standard ECMA-335 Common Language Infrastructure (CLI) 6th edition (June 2012); available online (http://www.ecma-international.org/publications/standards/Ecma-335.htm)
11. John Altidor; Huang Shan Shan; Yannis Smaragdakis (2011). "Taming the wildcards: combining definition- and use-site variance" (PDF). *Proceedings of the 32nd ACM SIGPLAN conference on Programming language design and implementation (PLDI'11)*.
12. Eric Lippert (October 29, 2007). "Covariance and Contravariance in C# Part Seven: Why Do We Need A Syntax At All?". Retrieved October 2013.
13. Marin Odersky; Lex Spoon (September 7, 2010). "The Scala 2.8 Collections API". Retrieved May 2013.
14. Joshua Bloch (November 2007). "The Closures Controversy [video]". Presentation at Javapolis'07. Retrieved

May 2013.
15. Martin Odersky; Matthias Zenger (2005). "Scalable component abstractions" (PDF). *Proceedings of the 20th annual ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications (OOPSLA '05)*.
16. Bill Venners and Frank Sommers (May 18, 2009). "The Purpose of Scala's Type System: A Conversation with Martin Odersky, Part III". Retrieved May 2013.
17. Ross Tate (2013). "Mixed-Site Variance". *FOOL '13: Informal Proceedings of the 20th International Workshop on Foundations of Object-Oriented Languages*.
18. Atsushi Igarashi; Mirko Viroli (2002). "On Variance-Based Subtyping for Parametric Types" (PDF). *Proceedings of the 16th European Conference on Object-Oriented Programming (ECOOP '02)*.
19. Kresten Krab Thorup; Mads Torgersen (1999). "Unifying Genericity: Combining the Benefits of Virtual Types and Parameterized Classes" (PDF). *Object-Oriented Programming (ECOOP '99)*.
20. Tate, Ross; Leung, Alan; Lerner, Sorin (2011). "Taming wildcards in Java's type system". *Proceedings of the 32nd ACM SIGPLAN conference on Programming language design and implementation (PLDI '11)*.

External links

- Fabulous Adventures in Coding (http://blogs.msdn.com/ericlippert/archive/tags/Covariance+and+Contravariance/default.aspx): An article series about implementation concerns surrounding co/contravariance in C#
- Contra Vs Co Variance (http://c2.com/cgi/wiki?ContraVsCoVariance) (note this article is not updated about C++)
- Closures for the Java 7 Programming Language (v0.5) (http://www.javac.info/closures-v05.html)

Categories: Object-oriented programming | Type theory | Polymorphism (computer science)

Covariance and Contravariance In Java

by Edwin Dalorzo · Apr. 08, 13 · Java Zone

The Java Zone is brought to you in partnership with ZeroTurnaround. Check out this 8-step guide to see how you can increase your productivity by skipping slow application redeploys and by implementing application profiling, as you code!

I have found that in order to understand covariance and contravariance a few examples with Java arrays are always a good start.

Arrays Are Covariant

Arrays are said to be covariant which basically means that, given the subtyping rules of Java, an array of type `T[]` may contain elements of type `T` or any subtype of `T`. For instance:

```
Number[] numbers = newNumber[3];
numbers[0] = newInteger(10);
numbers[1] = newDouble(3.14);
numbers[2] = newByte(0);
```

But not only that, the subtyping rules of Java also state that an array `S[]` is a subtype of the array `T[]` if `S` is a subtype of `T`, therefore, something like this is also valid:

```
Integer[] myInts = {1,2,3,4};
Number[] myNumber = myInts;
```

Because according to the subtyping rules in Java, an array `Integer[]` is a subtype of an array `Number[]` because `Integer` is a subtype of `Number`.

But this subtyping rule can lead to an interesting question: what would happen if we try to do this?

```
myNumber[0] = 3.14; //attempt of heap pollution
```

This last line would compile just fine, but if we run this code, we would get an `ArrayStoreException` because we're trying to put a double into an integer array. The fact that we are accessing the array through a `Number` reference is irrelevant here, what matters is that the array is an array of integers.

This means that we can fool the compiler, but we cannot fool the run-time type system. And this is so because arrays are what we call a *reifiable type*. This means that at run-time Java knows that this array was actually instantiated as an array of integers which simply happens to be accessed through a reference of type `Number[]`.

So, as we can see, one thing is the actual type of the object, an another thing is the type of the reference that we use to access it, right?

The Problem with Java Generics

Now, the problem with *generic types* in Java is that the type information for type parameters is discarded by the compiler after the compilation of code is done; therefore this type information is not available at run time. This process is called type erasure. There are good reasons for implementing generics like this in Java, but that's a long story, and it has to do with binary compatibility with pre-existing code.

The important point here is that since at run-time there is no type information, there is no way to ensure that we are not committing heap pollution.

Let's consider now the following unsafe code:

```
List<Integer> myInts = newArrayList<Integer>();
myInts.add(1);
myInts.add(2);
List<Number> myNums = myInts; //compiler error
myNums.add(3.14); //heap polution
```

If the Java compiler does not stop us from doing this, the run-time type system cannot stop us either, because there is no way, at run time, to determine that this list was supposed to be a list of integers only. The Java run-time would let us put whatever we want into this list, when it should only contain integers, because when it was created, it was declared as a list of integers. That's why the compiler rejects line number 4 because it is unsafe and if allowed could break the assumptions of the type system.

As such, the designers of Java made sure that we cannot fool the compiler. If we cannot fool the compiler (as we can do with arrays) then we cannot fool the run-time type system either.

As such, we say that generic types are *non-reifiable,* since at run time we cannot determine the true nature of the generic type.

Evidently this property of generic types in Java would have a negative impact on polymorphism. Let's consider now the following example:

```
staticlongsum(Number[] numbers) {
```

```
longsummation = 0;
for(Number number : numbers) {
summation += number.longValue();
}
returnsummation;
}
```

Now we could use this code as follows:

```
Integer[] myInts = {1,2,3,4,5};
Long[] myLongs = {1L, 2L, 3L, 4L, 5L};
Double[] myDoubles = {1.0, 2.0, 3.0, 4.0, 5.0};
System.out.println(sum(myInts));
System.out.println(sum(myLongs));
System.out.println(sum(myDoubles));
```

But if we attempt to implement the same code with generic collections, we would not succeed:

```
staticlongsum(List<Number> numbers) {
longsummation = 0;
for(Number number : numbers) {
summation += number.longValue();
}
returnsummation;
}
```

Because we we would get compiler errors if you try to do the following:

```
List<Integer> myInts = asList(1,2,3,4,5);
List<Long> myLongs = asList(1L, 2L, 3L, 4L, 5L);
List<Double> myDoubles = asList(1.0, 2.0, 3.0, 4.0, 5.0);
System.out.println(sum(myInts)); //compiler error
System.out.println(sum(myLongs)); //compiler error
System.out.println(sum(myDoubles)); //compiler error
```

The problem is that now we cannot consider a list of integers to be subtype of a list of numbers, as we saw above, that

would be considered unsafe for the type system and compiler rejects it immediately.

Evidently, this is affecting the power of polymorphism and it needs to be fixed. The solution consists in learning how to use two powerful features of Java generics known as covariance and contravariance.

Covariance

For this case, instead of using a type `T` as the type argument of a given generic type, we use a wildcard declared as `? extends T`, where `T` is a known base type.

With covariance we can read items from a structure, but we cannot write anything into it. All these are valid covariant declarations.

```
List<? extendsNumber> myNums = newArrayList<Integer>();
List<? extendsNumber> myNums = newArrayList<Float>();
List<? extendsNumber> myNums = newArrayList<Double>();
```

And we can read from our generic structure `myNums` by doing:

```
Number n = myNums.get(0);
```

Because we can be sure that whatever the actual list contains, it can be upcasted to a `Number` (after all anything that extends `Number` is a `Number`, right?)

However, we are not allowed to put anything into a covariant structure.

```
myNumst.add(45L); //compiler error
```

This would not be allowed because the compiler cannot determine what is the actual type of the object in the generic structure. It can be anything that extends `Number` (like `Integer`, `Double`, `Long`), but the compiler cannot be sure what, and therefore any attempt to retrieve a generic value is considered an unsafe operation and it is immediately rejected by the compiler. So we can read, but not write.

Contravariance

For contravariance we use a different wildcard called `? super T`, where `T` is our base type. With contravariance we can do the opposite. We can put things into a generic structure, but we cannot read anything out of it.

In this case, the actual nature of the object is `List` of `Object`, and through contravariance, we can put a `Number` in it, basically because a `Number` has `Object` as its common ancestor. As such, all numbers are also objects, and therefore this is valid.

However, we cannot safely read anything from this contravariant structure assuming that we will get a number.

```
Number myNum = myNums.get(0); //compiler-error
```

As we can see, if the compiler allowed us to write this line, we would get a `ClassCastException` at run time. So, once again, the compiler does not run the risk of allowing this unsafe operation and rejects it immediately.

Get/Put Principle

In summary, we use covariance when we only intend to take generic values out of a structure. We use contravariance when we only intend to put generic values into a structure and we use an invariant when we intend to do both.

The best example I have is the following that copies any kind of numbers from one list into another list. It only *gets* items from the source, and it only *puts* items in the destiny.

```
publicstaticvoidcopy(List<? extendsNumber> source,
List<? superNumber> destiny) {
for(Number number : source) {
destiny.add(number);
}
}
```

Thanks to the powers of covariance and contravariance this works for a case like this:

```
List<Integer> myInts = asList(1,2,3,4);
List<Integer> myDoubles = asList(3.14, 6.28);
List<Object> myObjs = newArrayList<Object>();
copy(myInts, myObjs);
copy(myDoubles, myObjs);
```

Further Reading

Most of my insights in this topic actually come from my reading of an excellent book:

- Java Generics and Collections (by Naftaling and Wadler)

Topics:

Covariant Vs Contravariant

http://stackoverflow.com/questions/8481301/covariance-invariance-and-contravariance-explained-in-plain-english

Some say it is about relationshop between types and subtypes, other say it is about type conversion and others say it is used to decide whether a method is overwritten or overloaded.

All of the above.

At heart, these terms describe how the subtype relation is affected by type transformations. That is, if `A` and `B` are types, f a type transformation, and ≤ the subtype relation (i.e. A ≤ B means that A is a subtype of B), we have

- f is covariant if A ≤ B implies that f(A) ≤ f(B)
- f is contravariant if A ≤ B implies that f(B) ≤ f(A)
- f is invariant if neither of the above holds

Let's consider an example. Let f(A) = `List<A>` where `List` is declared by

```
class List<T> { ... }
```

Is f covariant, contravariant, or invariant? Covariant would mean that a `List<String>` is a subtype of `List<Object>`, contravariant that a `List<Object>` is a subtype of `List<String>` and invariant that neither is a subtype of the other, i.e. `List<String>` and `List<Object>` are inconvertible types. In Java, the latter is true, we say (somewhat informally) that *generics* are invariant.

Another example. Let f(A) = `A[]`. Is f covariant, contravariant, or invariant? That is, is String[] a subtype of Object[], Object[] a subtype of String[], or is neither a subtype of the other? (Answer: In Java, arrays are covariant)

This was still rather abstract. To make it more concrete, let's look at which operations in Java are defined in terms of the subtype relation. The simplest example is assignment. The statement

```
x = y;
```

will compile only if typeof(y) ≤ typeof(x). That is, we have just learned that the statements

```
ArrayList<String> strings = new ArrayList<Object>();
ArrayList<Object> objects = new ArrayList<String>();
```

will not compile in Java, but

```
Object[] objects = new String[1];
```

will.

Another example where the subtype relation matters is a method invocation expression:

```
result = method(a);
```

Informally speaking, this statement is evaluated by assigning the value of `a` to the method's first parameter, then executing the body of the method, and then assigning the methods return value to `result`. Like the plain assignment in the last example, the "right hand side" must be a subtype of the "left hand side", i.e. this statement can only be valid if typeof(a) ≤ typeof(parameter(method)) and returntype(method) ≤ typeof(result). That is, if method is declared by:

```
Number[] method(ArrayList<Number> list) { ... }
```

none of the following expressions will compile:

```
Integer[] result = method(new ArrayList<Integer>());
Number[] result = method(new ArrayList<Integer>());
Object[] result = method(new ArrayList<Object>());
```

but

```
Number[] result = method(new ArrayList<Number>());
Object[] result = method(new ArrayList<Number>());
```

will.

Another example where subtyping matters is overriding. Consider:

```
Super sup = new Sub();
Number n = sup.method(1);
```

where

```
class Super {
    Number method(Number n) { ... }
}

class Sub extends Super {
    @Override
    Number method(Number n);
}
```

Informally, the runtime will rewrite this to:

```
class Super {
    Number method(Number n) {
        if (this instanceof Sub) {
            return ((Sub) this).method(n);  // *
        } else {
            ...
        }
    }
}
```

For the marked line to compile, the method parameter of the overriding method must be a supertype of the method parameter of the overridden method, and the return type a subtype of the overridden method's one. Formally speaking, f(A) = parametertype(method asdeclaredin(A)) must at least be contravariant, and if f(A) = returntype(method asdeclaredin(A)) must at least be covariant.

Note the "at least" above. Those are minimum requirements any reasonable statically type safe object oriented programming language will enforce, but a programming language may elect to be more strict. In the case of Java 1.4, parameter types and method return types must be identical (except for type erasure) when overriding methods, i.e. parametertype(method asdeclaredin(A)) = parametertype(method asdeclaredin(B)) when overriding. Since Java 1.5, covariant return types are permitted when overriding, i.e. the following will compile in Java 1.5, but not in Java 1.4:

```
class Collection {
    Iterator iterator() { ... }
}

class List extends Collection {
    @Override
    ListIterator iterator() { ... }
}
```

I hope I covered everything - or rather, scratched the surface. Still I hope it will help to understand the abstract, but important concept of type variance.

Taking the java type system, and then classes:

Any object of some type T can be substituted with an object of subtype of T.
TYPE VARIANCE - CLASS METHODS HAVE THE FOLLOWING CONSEQUENCES

```
class A {
    public S f(U u) { ... }
}

class B extends A {
    @Override
    public T f(V v) { ... }
}

B b = new B();
t = b.f(v);
A a = ...; // Might have type B
s = a.f(u); // and then do V v = u;
```

It can be seen, that:

- The T must be subtype S (**covariant, as B is subtype of A**).
- The V must be supertype of U (**contravariant**, as contra inheritance direction).

Now co- and contra- relate to B being subtype of A. The following stronger typings may be introduced with more specific knowledge. In the subtype.

Covariance (available in Java) is useful, to say that one returns a more specific result in the subtype; especially seen when A=T and B=S. Contravariance says you are prepared to handle a more general argument.

http://stackoverflow.com/questions/2501023/demonstrate-covariance-and-contravariance-in-java

Covariance:

```
class Super {
  Object getSomething(){}
}
class Sub extends Super {
  String getSomething() {}
}
```

Sub#getSomething is covariant because it returns a subclass of the return type of Super#getSomething (but fullfills the contract of Super.getSomething())

Contravariance

```
class Super{
  void doSomething(String parameter)
}
class Sub extends Super{
  void doSomething(Object parameter)
}
```

Sub#doSomething is contravariant because it takes a parameter of a superclass of the parameter of Super#doSomething (but, again, fullfills the contract of Super#doSomething)

Generics
This is also possible for Generics:

```
List<String> aList...
List<? extends Object> covariantList = aList;
List<? super String> contravariantList = aList;
```

You can now access all methods of `covariantList` that doesn't take a generic parameter (as it *must*be something "extends Object"), but getters will work fine (as the returned object will always be of type "Object")
The opposite is true for `contravariantList`: You can access all methods with generic parameters (you know it must be a superclass of "String", so you can always pass one), but no getters (The returned type may be of any other supertype of String)

Co-variance: Iterable and Iterator. It almost always makes sense to define a co-variant `Iterable`or `Iterator`. `Iterator<? extends T>` can be used just as `Iterator<T>` - the only place where the type parameter appears is the return type from the `next` method, so it can be safely up-cast to `T`. But if you have `S` extends `T`, you can also assign `Iterator<S>` to a variable of type `Iterator<? extends T>`. For example if you are defining a find method:

```
boolean find(Iterable<Object> where, Object what)
```

you won't be able to call it with `List<Integer>` and `5`, so it's better defined as

```
boolean find(Iterable<?> where, Object what)
```

Contra-variance: Comparator. It almost always makes sense to use `Comparator<? super T>`, because it can be used just as `Comparator<T>`. The type parameter appears only as the `compare`method parameter type, so `T` can be safely passed to it. For example if you have a `DateComparator implements Comparator<java.util.Date> { ... }` and you want to sort a `List<java.sql.Date>` with that comparator (`java.sql.Date` is a sub-class of `java.util.Date`), you can do with:

```
<T> void sort(List<T> what, Comparator<? super T> how)
```

but not with

```
<T> void sort(List<T> what, Comparator<T> how)
```

http://codeinventions.blogspot.com/2014/11/covariant-contravariant-and-class-invariant-example-and-difference-in-java.html

Covariant, Contravariant and Invariant (class invariant) in Java

Type variance is another feature which defines the flexibility in code writing every of programming language. Let me show you some of the variant type jargon's which explains the *type variance*.

When ever we are talking about type variance,
we end up with explaining about ***sub-typing which is again inheritance***. Because the above stated jargon's Covariant, Contravariant can clearly explain only with sub-typing and we use there terms in the context of Overloading and overriding Sticking to one language, examples shown below explains Covariance,Contravariance in Java. We will discuss Invariance later. The basic principle of there type variance is (quote from wikipedia)

If Cat is Subtype of Animal, then an expression of type Cat can be used whenever an expression of type Animal could.

Based on the above principle let see how the type has been divided and what actually it is.

Covariance:

Consider you have Parent and Child relationship and trying to override the methods of Parent in Child, *Co-variance*means that the overriding method returning a more specific type. Below example shows you the same, Parent method returning Object and where as the Child method decided to return a specific type (String) where String is child of Object class. Hence *covariance* existed here.

Covariant return type :

```
public class Parent{
  public Object doSomething(){}
 }
 public class Child extends Parent{
  public String doSomething() {}
 }
```

Covariant arg type :

```
public class Codeinventions {
      public static void print(Object[] arg) {
            for (Object object : arg) {
                  System.out.println(object);
            }
      }
      public static void main(String[] args) {
            String[] strs = { "Java", "Codeinventions" };
            print(strs);
      }
 }
 Result :
 Java
 Codeinventions
```

So, it is clear that we can pass a *Sub-type* where a Parent needed, since *Every Child is a Parent*. Yes Java supports it. And the same logic works in Generics as well.

Contravariance:

Contravariance is just reverse to Covariance. Passing exact type to the required place. There will be no flexibility in coding. But useful to avoid run time problem since you are being forced to stick to the exact type.

When you want to pass a Parent where Child is required, that's Contravariance and many languages won't allow you to do it. At least in Java I can confirm you that, it is not allowed to do so. Look at the Contravariance shown in below, and you end up with the error "*The method print(String) in the type ParentClass is not applicable for the arguments (Object)*".

```
public class ParentClass {
      public static void print(String str) {
            System.out.println(str);
      }
      public static void main(String[] args) {
            Object str = "Codeinventions";
            print(str);  //compiler error here
      }
 }
```

Form the compiler error shown above, there is no way of supporting Contravariance in Java.

Invriant (class invariant):

Invariant or variance is something which is always true no matter what the context is.

Variant or Variance is not just a *unchanged variable* it's more of a term which used to represent something which never changed what so ever it's context, Let me explain you the same with respect to Java with help of a Class called as *Class Invariance*. Consider there is a class called *Account* and have a *holder* in it. Now, **Invariance** or **Class Invariance** is, no matter where the *Account* being use in the program the state of the invariant *holder* will be satisfies some conditions(*always true*) at any context. That's the design decision to make which variable is invariant in the program.

```
Class Account {
private AccountHolder holder = new AccountHolder();
}
```

If you see in the above code making *holder* an invariant mean that no matter what the state of *Account*, my *holder*always holds some properties which are never be changed in normal conditions. May change in special conditions but with notifying to the owner.

There invariant's are useful or comes into picture while designing the program and for programmers while debugging the program. Consider at some point you are getting some weird results from a method and trying to debug your program, the basic principle you need to apply is see any invariant's are gone change and proceeding your debug keeping in mind these class *invariant's* so that it makes your debug easy as well.

Covariance = narrowing conversion.
Contravariance = widening conversion.
Invariance (in this context) = not convertible.

https://dzone.com/articles/covariance-and-contravariance

I have found that in order to understand covariance and contravariance a few examples with Java arrays are always a good start.

Arrays Are Covariant

Arrays are said to be covariant which basically means that, given the subtyping rules of Java, an array of type `T[]` may contain elements of type `T` or any subtype of `T`. For instance:

```
Number[] numbers = newNumber[3];
numbers[0] = newInteger(10);
numbers[1] = newDouble(3.14);
numbers[2] = newByte(0);
```

But not only that, the subtyping rules of Java also state that an array `S[]` is a subtype of the array `T[]` if `S` is a subtype of `T`, therefore, something like this is also valid:

```
Integer[] myInts = {1,2,3,4};
```

```
Number[] myNumber = myInts;
```

Because according to the subtyping rules in Java, an array `Integer[]` is a subtype of an array`Number[]` because `Integer` is a subtype of `Number`.

But this subtyping rule can lead to an interesting question: what would happen if we try to do this?

```
myNumber[0] = 3.14; //attempt of heap pollution
```

This last line would compile just fine, but if we run this code, we would get an`ArrayStoreException` because we're trying to put a double into an integer array. The fact that we are accessing the array through a `Number` reference is irrelevant here, what matters is that the array is an array of integers.

This means that we can fool the compiler, but we cannot fool the run-time type system. And this is so because arrays are what we call a *reifiable type*. This means that at run-time Java knows that this array was actually instantiated as an array of integers which simply happens to be accessed through a reference of type `Number[]`.

So, as we can see, one thing is the actual type of the object, an another thing is the type of the reference that we use to access it, right?

The Problem with Java Generics

Now, the problem with *generic types* in Java is that the type information for type parameters is discarded by the compiler after the compilation of code is done; therefore this type information is not available at run time. This process is called type erasure. There are good reasons for implementing generics like this in Java, but that's a long story, and it has to do with binary compatibility with pre-existing code.

The important point here is that since at run-time there is no type information, there is no way to ensure that we are not committing heap pollution.

Let's consider now the following unsafe code:

```
List<Integer> myInts = newArrayList<Integer>();
myInts.add(1);
myInts.add(2);
List<Number> myNums = myInts; //compiler error
```

```
myNums.add(3.14); //heap polution
```

If the Java compiler does not stop us from doing this, the run-time type system cannot stop us either, because there is no way, at run time, to determine that this list was supposed to be a list of integers only. The Java run-time would let us put whatever we want into this list, when it should only contain integers, because when it was created, it was declared as a list of integers. That's why the compiler rejects line number 4 because it is unsafe and if allowed could break the assumptions of the type system.

As such, the designers of Java made sure that we cannot fool the compiler. If we cannot fool the compiler (as we can do with arrays) then we cannot fool the run-time type system either.

As such, we say that generic types are *non-reifiable,* since at run time we cannot determine the true nature of the generic type.

Evidently this property of generic types in Java would have a negative impact on polymorphism. Let's consider now the following example:

```
staticlongsum(Number[] numbers) {
longsummation = 0;
for(Number number : numbers) {
summation += number.longValue();
}
returnsummation;
}
```

Now we could use this code as follows:

```
Integer[] myInts = {1,2,3,4,5};
Long[] myLongs = {1L, 2L, 3L, 4L, 5L};
Double[] myDoubles = {1.0, 2.0, 3.0, 4.0, 5.0};
System.out.println(sum(myInts));
System.out.println(sum(myLongs));
System.out.println(sum(myDoubles));
```

But if we attempt to implement the same code with generic collections, we would not succeed:

```
staticlongsum(List<Number> numbers) {
```

```
longsummation = 0;
for(Number number : numbers) {
summation += number.longValue();
}
returnsummation;
}
```

Because we we would get compiler errors if you try to do the following:

```
List<Integer> myInts = asList(1,2,3,4,5);
List<Long> myLongs = asList(1L, 2L, 3L, 4L, 5L);
List<Double> myDoubles = asList(1.0, 2.0, 3.0, 4.0, 5.0);
System.out.println(sum(myInts)); //compiler error
System.out.println(sum(myLongs)); //compiler error
System.out.println(sum(myDoubles)); //compiler error
```

The problem is that now we cannot consider a list of integers to be subtype of a list of numbers, as we saw above, that would be considered unsafe for the type system and compiler rejects it immediately.

Evidently, this is affecting the power of polymorphism and it needs to be fixed. The solution consists in learning how to use two powerful features of Java generics known as covariance and contravariance.

Covariance

For this case, instead of using a type `T` as the type argument of a given generic type, we use a wildcard declared as `? extends T`, where `T` is a known base type.

With covariance we can read items from a structure, but we cannot write anything into it. All these are valid covariant declarations.

```
List<? extendsNumber> myNums = newArrayList<Integer>();
List<? extendsNumber> myNums = newArrayList<Float>();
List<? extendsNumber> myNums = newArrayList<Double>();
```

And we can read from our generic structure `myNums` by doing:

```
Number n = myNums.get(0);
```

Because we can be sure that whatever the actual list contains, it can be upcasted to a `Number`(after all anything that extends `Number` is a `Number`, right?)

However, we are not allowed to put anything into a covariant structure.

```
myNumst.add(45L); //compiler error
```

This would not be allowed because the compiler cannot determine what is the actual type of the object in the generic structure. It can be anything that extends `Number` (like `Integer`, `Double`,`Long`), but the compiler cannot be sure what, and therefore any attempt to retrieve a generic value is considered an unsafe operation and it is immediately rejected by the compiler. So we can read, but not write.

Contravariance

For contravariance we use a different wildcard called `? super T`, where `T` is our base type. With contravariance we can do the opposite. We can put things into a generic structure, but we cannot read anything out of it.

In this case, the actual nature of the object is `List` of `Object`, and through contravariance, we can put a `Number` in it, basically because a `Number` has `Object` as its common ancestor. As such, all numbers are also objects, and therefore this is valid.

However, we cannot safely read anything from this contravariant structure assuming that we will get a number.

```
Number myNum = myNums.get(0); //compiler-error
```

As we can see, if the compiler allowed us to write this line, we would get a `ClassCastException` at run time. So, once again, the compiler does not run the risk of allowing this unsafe operation and rejects it immediately.

Get/Put Principle

In summary, we use covariance when we only intend to take generic values out of a structure. We use contravariance when we only intend to put generic values into a structure and we use an invariant when we intend to do both.

The best example I have is the following that copies any kind of numbers from one list into another list. It only *gets* items from the source, and it only *puts* items in the destiny.

```
publicstaticvoidcopy(List<? extendsNumber> source,
List<? superNumber> destiny) {
for(Number number : source) {
destiny.add(number);
}
}
```

Thanks to the powers of covariance and contravariance this works for a case like this:

```
List<Integer> myInts = asList(1,2,3,4);
List<Integer> myDoubles = asList(3.14, 6.28);
List<Object> myObjs = newArrayList<Object>();
copy(myInts, myObjs);
copy(myDoubles, myObjs);
```

Covariance and contravariance rules in Java

September 2014 programming

This post is a go-to quick reference for exactly how covariance and contravariance work in Java. Different programming languages do this in different ways, so sometimes it can be tricky to switch between languages and keep the rules straight. When you switch over to Java, use this guide to refresh your memory.

Type conversion

Type conversion in Java is **covariant** (unlike Dart). That means that if `SubClazz` is a subtype of `Clazz` then a `SubClazz` reference can be cast to a `Clazz`.

```
public class Clazz { }
public class SubClazz extends Clazz { }

(Clazz)new SubClazz(); // OK
(SubClazz)new Clazz(); // Error
```

Conversion can occur implictly during assignment:

```
Clazz instance = new SubClazz();
```

Conversion can also occur implicitly when returning from a method or when passing arguments.

```
public Clazz makeClazz() {
    return new SubClazz();
}

public Clazz takeClazz(Clazz foo) { }

takeClazz(new SubClazz());
```

Arrays

Arrays in Java are **covariant** in the type of the objects they hold. In other words, `Clazz[]` can hold `SubClazz` objects.

```
Clazz[] array = new Clazz[10];
array[0] = new SubClazz();
```

They are also **covariant** in the type of the array itself. You can directly assign a `SubClazz[]` type to a `Clazz[]`.

```
Clazz[] array = new SubClazz[10];
```

Be careful though; the above line is dangerous. Although the type of the `array` variable is `Clazz[]`, the actual array object on the heap is a `SubClazz[]`. For that reason, the following code compiles fine but throws a `java.lang.ArrayStoreException` at runtime:

```
Clazz[] array = new SubClazz[10];
array[0] = new Clazz();
```

Overriding methods

The overriding method is **covariant** in the return type and **invariant** in the argument types. That means that the return type of the overriding method can be a subclass of the return type of the overridden method, but the argument types must match exactly.

```
public interface Parent {
    public Clazz act(Clazz argument);
}

public interface Child extends Parent {
    @Override
    public SubClazz act(Clazz argument);
}
```

If the argument types aren't identical in the subclass then the method will be *overloaded* instead of overridden. You should always use the `@Override` annotation to ensure that this doesn't happen accidentally.

Generics

Unless bounds are involved, generic types are **invariant** with respect to the parameterized type. So you can't do covariant ArrayLists like this:

```
ArrayList<Clazz> ary = new ArrayList<SubClazz>(); // Error!
```

The normal rules apply to the type being parameterized:

```
List<Clazz> list = new ArrayList<Clazz>();
```

Unbounded wildcards allow assignment with any type parameter:

```
List<?> list = new ArrayList<Clazz>();
```

Bounded wildcards affect assignment like you might expect:

```
List<? extends Clazz> list = new ArrayList<SubClazz>();
List<? super Clazz> list2 = new ArrayList<Object>();
```

Java is smart enough that *more* restrictive type bounds are commensurable with *less* restrictive type bounds when appropriate:

```
List<? super Clazz> clazzList;
List<? super SubClazz> subClazzList;


subClazzList = clazzList;
```

Type parameter bounds work the same way, although they cannot be lower-bounded. If you have multiple upper bounds on a type parameter, you can upcast to any of them, as expected:

```
interface A {}
interface B {}
interface C extends A, B {}

public class Holder<T extends A & B> {
    T member;
}


A member1 = new Holder<C>().member;
B member2 = new Holder<C>().member;
C member3 = new Holder<C>().member;
```

You can add or remove the type parameters from the return type of an overriding method and it will still compile:

```
public interface Parent {
    public List echo();
}

public interface Child extends Parent {
    @Override
    public List<String> echo();
}


public interface Parent {
    public List<String> echo();
}

public interface Child extends Parent {
    @Override
    public List echo();
}
```

Wildcards can be present in the types of method arguments. If you want to override a method with a wildcard-typed argument, the overriding method must have an identical type parameter. You cannot be

"more specific" with the overriding method:

```
public interface Parent {
    public void act(List<? extends List> a);
}

public interface Child extends Parent {
    @Override
    public void act(List<? extends ArrayList> a); // Error!
}
```

Also, you can replace any type-parameterized method argument with a non-type-parameterized method argument in the subclass and it will still be considered an override:

```
public interface Parent {
    public void act(List<? extends Number> a);
}

public interface Child extends Parent {
    @Override
    public void act(List a);
}
```

Related Posts

- 31 Aug 2014 » The skyline problem
- 02 Jun 2014 » Square roots and fixed points
- 05 Mar 2014 » MagicalTux's EVE Online pathfinder
