Theory: Wildcards

51 minutes 0 / 5 problems solved

Earlier, when we were discussing type bounds, we've mentioned *Wildcards* as a feature that does the same trick and has wide application.
Wildcards are a specific Java tool that allows the implementation of some compatibility between different generic objects. The wildcard is basically "?" sign used to indicate that a class, a method, or a field is compatible with different type parameters.

§1. Why Wildcards?

Since Java is an object-oriented language, the concept of inheritance is essential. However, since generics are type-safe structures it is impossible to introduce inheritance for Generic objects.
To illustrate the problem, let's consider two classes:

```
class Book{}
class Album extends Book {}
```

Logically we assume that a list of albums can be treated as a list of books, because `Album` is a subclass of `Book`. However, the compiler thinks in a different way:

```
List<Album> albums = new ArrayList<>();
List<Book> books = albums; // compile-time error
```

The root cause of the problem lies In the fact that `List<Album>` is not a subclass of `List<Book>`: a usual inheritance rule of Java does not work this way with generic classes. Such behavior is known as **invariance**. It doesn't matter that `Album` extends `Book`, their containers like a `List<T>`, `Set<T>` and others are treated like independent classes. It is extremely important to mind this fact every time you use generic classes.

The example above is exactly where wildcards could help. A generic class or a method declared with wildcards can take any type parameter and there won't be any collisions with inheritance. To implement wildcards, use "?" inside angle brackets (`<?>`). Let's use it to make the compiler error go away in the example above:

```
List<Album> albums = new ArrayList<>();
List<? extends Book> albumsAndBooks = albums; // it is ok
```

or

```
List<Album> albums = new ArrayList<>();
List<? super Album> albumsAndBooks = albums; // it is ok as well
```

Wildcards are commonly used with some limitations that we called type bounds before: there we used only an `extends` keyword. Now we will take a look at another keyword `super`. Since wildcards are used for type bounding, they can be divided into three groups: unbounded wildcards, upper bounded wildcards, and lower bounded ones.

§2. Upper Bounded Wildcards

Upper Bounded Wildcards are used when we want to set an upper bound. It is done with the `extends` keyword, like this:

```
? extends ReferenceType
```

It can be read as "any type that is a subtype of `ReferenceType`". In other words, if `S` is a subtype of `T` then type `List<S>` is considered to be a subtype of `List<? extends T>`. That feature is known as **covariance**.

Current topic:

Wildcards

Topic depends on:

- ✕ Generics and Object
- ✕ Generic methods

Topic is required for:

- Type Erasure

Table of contents:

Now imagine again that we are dealing with a library, where we have books of different types (normal books, booklets, albums and so on). We also may have some other media files like audio recordings. Let's introduce two classes:

```
public class Booklet extends Book {}
public class AudioFile {}
```

Now say we want to create storage for all types of books:

```
List<? extends Book> storage = new ArrayList<>();

List<Album> albums = new ArrayList<>();
storage = albums; // it works, Album is a subtype of Book

List<Booklet> booklets = new ArrayList<>();
storage = booklets; // it works, Booklet is a subtype of Book

List<AudioFile> recordings = new ArrayList<>();

storage = recordings; // compile-time error, AudioFile is not a subtype of Book
```

This way we made sure that only subtypes of the `Book` can be put to the storage.

Now let's consider another limitation of upper bounding.

```
/**
* Hierarchy: Book -> Album
*                 -> Booklet
* Allowed types: List<Book>, List<Album>, List<Booklet>
*/
public void upperBoundedMethod(List<? extends Book> books) {
    Book book = books.get(0); // It is fine

    books.add(new Album()); // compile-time error

    books.add(new Book());  // compile-time error

    books.add(null); // also fine, because of null is a special type-independent value

}
```

It may be surprising, but some lines of `upperBoundedMethod` won't compile. Upper bounded wildcards are completely fine with reading content as `Book` type, but writing is prohibited except a `null` value.

Let's explain the logic. The compiler doesn't know which type of argument will be passed to the method in runtime. As we already learned, the method accepts list parameterized by `Book` or any of its subtypes: `List<Books>`, `List<Album>` or `List<Booklet>`. This is a reason why any object from `books` argument can be read as `Book`. However, writing is prohibited to avoid future runtime errors. Imagine the case when `List<Album>` was passed, but then we try to add an instance of `Book`. It can potentially lead to a runtime error because an added object will be treated as `Album` in the future.

§3. Lower Bounded Wildcards

Lower Bounded Wildcards are introduced with the `super` keyword followed by the lower bound:

```
? super ReferenceType
```

It means "any type that is a supertype of `ReferenceType`" and that if `S` is a supertype of `T` then `List<S>` is considered to be a supertype of `List<? super T>`. The feature is called **contravariance**.

Let's think of books again. Now we would like to write a code that will enable `List` of `Albums` and its superclasses to be added to a general library.

Take a look at the following code:

```
List<? super Album> storage = new ArrayList<>();

List<Album> albums = new ArrayList<>();
storage = albums; // it works

List<Book> books = new ArrayList<>();
storage = books; // it works, Book is a supertype for Album

List<Booklet> booklets = new ArrayList<>();

storage = booklets; // compile-
time error, Booklet is not a supertype for Album
```

Here we made sure that only supertypes of the `Album` class can be put to the storage.

Now let's consider another limitation of lower bounding.

```
/**
* Hierarchy: Album <- Book <- Object
* Allowed types: List<Album>, List<Book>, List<Object>
*/
public void lowerBoundedMethod(List<? super Album> albums) {

 Object object = albums.get(0); // it is ok. Object is upper bound of Album
       Book book = albums.get(0);     // compile-time error
       Album album = albums.get(0);   // compile-time error


       albums.add(new Object()); // compile-time error

       albums.add(new Book());   // compile-time error

       albums.add(new Album());  // OK

       albums.add(null);         // OK, null is type-independent

 }
```

There are also some compile-time errors as well as for upper bounded wildcards. Let's explain why the compiler suspects these lines as potential danger.

Since any of `List<Album>`, `List<Book>`, `List<Object>` can be passed to the `lowerBoundedMethod`, we can't assert that read object has a certain type `Album` or `Book`. We can only assume its type as `Object` for sure.

On the other hand, only an instance of `Album` can be treated as `Book` and `Object` simultaneously, that is why we are allowed to add only `Album`. Otherwise, if we pass `List<Album>` to the method and add an instance of `Book`, it will lead to the instance of `Book` being treated as `Album` in the future. Such errors are prevented by the compiler.

§4. Get and Put Principle

To detect and memorize whether `extends` or `super` should be used it is worth remembering the *Get and Put principle*:

Use Upper Bounded Wildcards (i.e., **<? extends Number>**) when you only get values out of a structure (when you use only getters or similar methods), use Lower Bounded Wildcards (i.e., **<? super Integer>**) when you only put values into a structure (when you use only setters or similar methods) and do use Unbounded Wildcards (simple **<?>**) when you both get and put (when it is essential for you to use all kind of methods).

To memorize this principle, you can also use PECS: Producer Extends, Consumer Super. This means that if you get a value from a generic class, method or any other object (it can *produce* for you what you need), you use `extends`. And vice versa, if you put or set a value into a generic class, method or any other object (it can *consume* what you put in it), you use `super`.

Remember, that it is not possible to put anything into a type declared with an `extends` wildcard except for the `null` value since it can represent any reference type. Similarly, it is not possible to get anything from a type declared with `super` wildcard except for a value of an `Object` type: a super type for every reference type.

> You cannot use a lower and an upper bound simultaneously in wildcards in particular and in type bounds in Java in general.

Note, that a class or an interface that is used after an “extends” or a “super” keyword itself is included in the inheritance. For example, `Box<T>` is absolutely compatible and covariant with `Box<? extends T>` or `Box<? super T>`.

In the end, it is important to note that a frequently used unbounded wildcard `?` is equivalent to: `? extends Object`.

> It is interesting that an inheritance prohibition in generics is made specifically to prevent run-time errors: otherwise, generics would lose their type safety feature.

§5. Wildcard Capture

Let's consider the example:

```
public static void reverse(List<?> list) {
  List<Object> tmp = new ArrayList<Object>(list);
  for (int i = 0; i < list.size(); i++) {
    list.set(i, tmp.get(list.size() - i - 1)); // compile-time error
  }
}
```

On the first look, this example may seem ok to you, but compile-error hints us it is not. As you know `<?>` equivalent to `<? extends Object>`, so by PECS principle, we cannot mutate the content of `list`, just read it. The scenario is known as **wildcard capture** problem and can be solved by the trick:

```
public static void reverse(List<?> list) {
  reverseCaptured(list);
}

private static <T> void reverseCaptured(List<T> list) {
  List<T> tmp = new ArrayList<T>(list);
  for (int i = 0; i < list.size(); i++) {
    list.set(i, tmp.get(list.size() - i - 1));
  }

}
```

Here we introduced a helper method `reverseCaptured` which has a parameter of a certain type `T` for all elements of list. The method is completely fine from the compiler point of view because it is a merely **generic method**.

§6. Conclusion

Wildcards are a very convenient and safe way of implementing an equivalent of inheritance in Generics. They are declared as a “?” in angle brackets and are widely used with upper or lower bounds to restrict type parameters. Wildcards are mainly used inside different libraries and frameworks, as well as generics themselves.

Report a typo

67 users liked this theory. **1** didn't like it. **What about you?**

Start practicing

Comments (10) Hints (0) Useful links (1) Show discussion