18 Java Collections and Generics Best Practices

Last Updated on 26 November 2017 | Print Email

By applying the following best practices with regard to collections and generics, you will know how to use collections and generics effectively in the right ways rather than “just work”.

The best practices about collections and generics which I’m going to share with you today come from my own experiences over many years of working as a Java developer. Perhaps you will see some practices are new, some you already knew but the key point is that, you should consider these useful practices and apply them quickly into your daily coding.

Here’s the list:

Now, let’s start with the first one.

1. Choosing the right collections

This is the first and most important step before using a collection. Depending on the problem you’re trying to solve, choose the most appropriate collection. If you choose a wrong one, your program might still work but work inefficiently; and if you choose a right collection, your solution may be much simpler and your program works much faster.

Choosing the right collection is like choosing a vehicle to travel from Tokyo to New York. If you choose a boat, perhaps you will reach the destination after few months. If you choose an airplane, you will be at the Times square within the day. And the journey is impossible if you choose a train.

To know which kind of collection (`List`, `Set`, `Map`, `Queue`, etc) is appropriate to solve the problem, you should figure out the characteristics and behaviors of each and the differences among them. You also need to know the pros and cons of each concrete implementation (`ArrayList`, `LinkedList`, `HashSet`, `TreeSet`, etc).

Basically, you decide to choose a collection by answering the following questions:

- Does it allow duplicate elements?
- Does it accept null?

- Does it allow accessing elements by index?

- Does it offer fast adding and fast removing elements?

- Does it support concurrency?

- Etc

So let consult these tutorials to understand each kind of collection and their concrete implementation:

- Java List Collection Tutorial and Examples
- Java Map Collection Tutorial and Examples
- Java Queue Collection Tutorial and Examples
- Java Set Collection Tutorial and Examples

Also refer to their Javadocs whenever you're not sure something about a specific collection.

2. Always using interface type when declaring a collection

It's better to declare a list like this:

```
1  List<String> listNames = new ArrayList<String>();   // (1)
```

instead of:

```
1  ArrayList<String> listNames = new ArrayList<String>();  // (2)
```

What's the difference between (1) and (2)?

In (1), the type of the variable `listNames` is List, and in (2) `listNames` has type of `ArrayList`. By declaring a collection using an interface type, the code would be more flexible as you can change the concrete implementation easily when needed, for example:

```
1  List<String> listNames = new LinkedList<String>();
```

When your code is designed to depend on the `List` interface, then you can swap among `List`'s implementations with ease, without modifying the code that uses it.

The flexibility of using interface type for a collection is more visible in case of method's parameters. Consider the following method:

```
1  public void foo(Set<Integer> numbers) {
2  }
```

Here, by declaring the parameter `numbers` as of type `Set`, the client code can pass any implementations of `Set` such as `HashSet` or `TreeSet`:

```
1  foo(treeSet);
2  foo(hashSet);
```

This makes your code more flexible and more abstract.

In contrast, if you declare the parameter `numbers` as of type `HashSet`, the method cannot accept anything except `HashSet` (and its subtypes), which makes the code less flexible.

It's also recommended to use interface as return type of a method that returns a collection, for example:

```
1  public Collection listStudents() {
2      List<Student> listStudents = new ArrayList<Student>();
3
4      // add students to the list
5
6      return listStudents;
7  }
```

This definitely increases the flexibility of the code, as you can change the real implementation inside the method without affecting its client code.

So this 2nd best practice encourages you to favor abstract types over concrete types.

3. Use generic type and diamond operator

Of course you should declare a collection of a generic type like this:

```
List<Student> listStudents = new ArrayList<Student>();
```

Since Java 7, the compiler can infer the generic type on the right side from the generic type declared on the left side, so you can write:

```
List<Student> listStudents = new ArrayList<>();
```

The **<>** is informally called the diamond operator. This operator is quite useful. Imagine if you have to declare a collection like this:

```
Map<Integer, Map<String, Student>> map = new HashMap<Integer, Map<String, Student>>();
```

You see, without the diamond operator, you have to repeat the same declaration twice, which make the code un-necessarily verbose. So the diamond operator saves you:

```
Map<Integer, Map<String, Student>> map = new HashMap<>();
```

4. Specify initial capacity of a collection if possible

Almost concrete collection classes have an overloaded constructor that specifies the initial capacity of the collection (the number of elements which the collection can hold when it is created). That means if you're pretty sure how many elements will be added to the collection, let specify the initial capacity when creating a new instance of that collection. For example:

```
List<String> listNames = new ArrayList<String>(5000);
```

This creates an array list that can hold 5000 elements initially. If you don't specify this number, than the array list itself will have to grow its internal array each time the current capacity is exceeded, which is inefficient. Therefore, consult Javadocs of each collection to know its default initial capacity so you can know whether you should explicitly specify the initial capacity or not.

5. Prefer isEmpty() over size()

Avoid checking the emptiness of a collection like this:

```
if (listStudents.size() > 0) {
    // dos something if the list is not empty
}
```

Instead, you should use the `isEmpty()` method:

```
if (!listStudents.isEmpty()) {
    // dos something if the list is not empty
}
```

There's no performance difference between `isEmpty()` and `size()`. The reason is for the readability of the code.

6. Do not return null in a method that returns a collection

If a method is designed to return a collection, it should not return null in case there's no element in the collection. Consider the following method:

```
public List<Student> findStudents(String className) {
    List<Student> listStudents = null;

    if (//students are found//) {
        // add students to the lsit
    }
```

Here, the method returns null if no student are found. The key point here is, a null value should not be used to indicate no result. The best practice is, returning an empty collection to indicate no result. The above code can be easily corrected by initializing the collection:

```
List<Student> listStudents = new ArrayList<>;
```

Therefore, always check the logic of the code to avoid returning null instead of an empty collection.

7. Do not use the classic for loop

There's nothing wrong if you write code to iterate a list collection like this:

```
for (int i = 0; i < listStudents.size(); i++) {
    Student aStudent = listStudents.get(i);

    // do something with aStudent
}
```

However, this is considered as bad practice because using the counter variable `i` may lead to potential bugs if it is altered somewhere inside the loop. Also this kind of loop is not object-oriented, since every collection has its own iterator. So it's recommended to use an iterator like the following code:

```
Iterator<Student> iterator = listStudents.iterator();

while (iterator.hasNext()) {
    Student nextStudent = iterator.next();

    // do something with nextStudent
}
```

Also the iterator may throw `ConcurrentModificationException` if the collection is modified by another thread after the iterator is created, which eliminates potential bugs.

Now, it's better to use the enhanced for loop like this:

```
for (Student aStudent : listStudents) {
    // do something with aStudent
}
```

As you can see, the enhanced for loop is more succinct and readable though it uses an iterator behind the scenes.

8. Favor using forEach() with Lambda expressions

Since Java 8, every collection now provides the `forEach()` method that encapsulates the iteration code inside the collection itself (internal iteration), and you just pass a Lambda expression to this method. This make the iteration code even more compact, more flexible and more powerful. Here's an example:

```
List<String> fruits = Arrays.asList("Banana", "Lemon", "Orange", "Apple");

fruits.forEach(fruit -> System.out.println(fruit));
```

This is equivalent to the following enhanced for loop:

```
for (String fruit : fruits) {
```

So I encourage you to use the `forEach()` method for iterating a collection in a way that helps you focus on your code, not on the iteration.

9. Overriding equals() and hashCode() properly

When you use a collection of a custom type, e.g. a list of `Student` objects, remember to override the `equals()` and `hashCode()` methods in the custom type properly, in order to allow the collection manages the elements efficiently and properly, especially in `Set` collections which organize elements based on their hash code values.

See the article Understanding equals() and hashCode() in Java to understand the constraints between `equals()` and `hashCode()` and how to override them in your class.

10. Implementing the Comparable interface properly

Remember to have your custom types implemented the `Comparable` interface properly when their elements are added to collections that sort elements by natural ordering, such as `TreeSet` and `TreeMap`. It also helps to sort elements in a list collection based on the natural ordering of the elements.

See the article Understanding Object Ordering in Java with Comparable and Comparator to understand how this practice works in details.

11. Using Arrays and Collections utility classes

Be aware that the Java Collections Framework provides two utility classes named **`Arrays`**and **`Collections`**which give us many useful functionalities. For example, the `Arrays.asList()` method returns a list collection containing the given elements, as you can see I used this method in many examples:

```
List<String> listFruits = Arrays.asList("Apple", "Banana", "Orange");
List<Integer> listIntegers = Arrays.asList(1, 2, 3, 4, 5, 6);
List<Double> listDoubles = Arrays.asList(0.1, 1.2, 2.3, 3.4);
```

And the `Collections` class provides various useful methods for searching, sorting, modifying elements in a collection (almost on lists). Therefore, remember to look at these two utility classes for reusable methods, before looking for other libraries or writing your own code.

.

12. Using the Stream API on collections

Since Java 8, every collection now has the `stream()` method that returns a stream of elements so you can take advantages of the Stream API to perform aggregate functions with ease. For example, the following code uses the Stream API to calculate sum of a list of integers:

```
List<Integer> numbers = Arrays.asList(1, 2, 3, 4, 5, 6);
int sum = numbers.stream().reduce(0, (x, y) -> x + y);
System.out.println("sum = " + sum);
```

The key point here is, always take advantages of the Stream API on collections to write code that performs aggregate functions quickly and easily.

See the tutorial Understanding Java 8 Stream API for more useful examples using the Stream API.

13. Prefer concurrent collections over synchronized wrappers

When you have to use collections in multi-threaded applications, consider using concurrent collections in the `java.util.concurrent` package instead of using the synchronized collections generated by the `Collections.synchronizedXXX()` methods. It's because the concurrent collections are designed to provide maximum performance in concurrent applications, by implementing different synchronization mechanisms like copy-on-write, compare-and-swap and special locks. The following list shows you how to choose some concurrent collections (on the right) which are equivalent to the normal ones (on the left):

- `HashMap -> ConcurrentHashMap`
- `ArrayList -> CopyOnWriteArrayList`
- `TreeMap -> ConcurrentSkipListMap`
- `PriorityQueue -> PriorityBlockingQueue`

See the article Understanding Collections and Thread Safety in Java to understand in-depth about collections and thread safety.

14. Using third-party collections libraries

The Java Collections Framework is not always sufficient for all demands, so third-party collections libraries emerge to fulfill the needs. There are a lot of third-party collections out there, however there are 4 notable ones:

- **Fastutil**: This library is a great choice for collections of primitive types like `int` or `long`. It's also able to handle big collections with more than 2.1 billion elements (2^31) very well.

- **Guava**: This is Google core libraries for Java 6+. It contains a magnitude of convenient methods for creating collections, like fluent builders, as well as advanced collection types like `HashBiMap`, `ArrayListMultimap`, etc.

- **Eclipse Collections**: this library includes almost any collection you might need: primitive type collections, multimaps, bidirectional maps and so on.

- **JCTools**: this library provides Java concurrency tools for the JVM. It offers some concurrent data structures currently missing from the JDK such as advanced concurrent queues.

Having said that, don't lock yourself to Java Collections Framework provided by the JDK, and always take advantages of the third-party collections libraries.

Recommended Book: **Java Generics and Collections**

15. Eliminate unchecked warnings

When the Java compiler issues unchecked warnings, do not ignore them. The best practice is, you should eliminate unchecked warnings. Consider the following code:

```
1  List list1 = new ArrayList();
2  List<String> list2 = new ArrayList<>(list1);
```

The compiler issues the following warning although the code is still compiled:

```
1  Note: ClassNam.java uses unchecked or unsafe operations
```

Unchecked warnings are important, so don't ignore them. It's because every unchecked warning represent a potential `ClassCastException` at runtime. In the above code, if `list1` contains an Integer element rather than String, than the code that uses `list2` will throw `ClassCastException` at runtime.

Let do your best to eliminate these warnings. The above code can be corrected like this:

```
1  List<String> list1 = new ArrayList<>();
2  List<String> list2 = new ArrayList<>(list1);
```

However, not every warning can be easily eliminated like this. In cases you cannot eliminate unchecked warnings, let prove that the code is typesafe and then suppress the warning with an `@SuppressWarnings("unchecked")` annotation in the narrowest possible scope. Also write comments explaining why you suppress the warning.

16. Favor generic types

Consider to generify your existing types when possible, as generic types are safer and easier to use then non-generic ones. When you design new types, also consider if they can be generified.

See the tutorial How to write generic classes and methods in Java to know details about how to write generic classes.

17. Favor generic methods

Like generic types, you are encouraged to write new methods with generic parameters in mind, and convert your existing methods to make use of type parameters, as generic methods are safer and easier to use than non-generic ones. Generic methods also help you write highly general and reusable APIs.

18. Using bounded wildcards to increase API flexibility

When writing new generic methods, consider using wildcard types on input parameters for maximum flexibility. Consider the following method:

```
1  public double sum(Collection<Number> col) {
2      double sum = 0;
3
4      for (Number num : col) {
5          sum += num.doubleValue();
6      }
7
8      return sum;
9  }
```

A limitation of this method is that it can accept only `List<Number>`, `Set<Number>` but not `List<Integer>`, `List<Long>` or `Set<Double>`. So to maximize the flexibility, update the method to use the bounded wildcard as shown below:

```
1  public double sum(Collection<? extends Number> col)
```

Now, this method can accept a collection of any types which are subtypes of `Number` like `Integer`, `Double`, `Long`, etc.

See the article Generics with extends and super Wildcards and the Get and Put Principle to understand details about generic wildcards.