CHAPTER 8

Methods

THIS chapter discusses several aspects of method design: how to treat parameters and return values, how to design method signatures, and how to document methods. Much of the material in this chapter applies to constructors as well as to methods. Like Chapter 4, this chapter focuses on usability, robustness, and flexibility.

Item 49: Check parameters for validity

Most methods and constructors have some restrictions on what values may be passed into their parameters. For example, it is not uncommon that index values must be non-negative and object references must be non-null. You should clearly document all such restrictions and enforce them with checks at the beginning of the method body. This is a special case of the general principle that you should attempt to detect errors as soon as possible after they occur. Failing to do so makes it less likely that an error will be detected and makes it harder to determine the source of an error once it has been detected.

If an invalid parameter value is passed to a method and the method checks its parameters before execution, it will fail quickly and cleanly with an appropriate exception. If the method fails to check its parameters, several things could happen. The method could fail with a confusing exception in the midst of processing. Worse, the method could return normally but silently compute the wrong result. Worst of all, the method could return normally but leave some object in a compromised state, causing an error at some unrelated point in the code at some undetermined time in the future. In other words, failure to validate parameters, can result in a violation of *failure atomicity* (Item 76).

For public and protected methods, use the Javadoc `@throws` tag to document the exception that will be thrown if a restriction on parameter values is violated

(Item 74). Typically, the resulting exception will be `IllegalArgumentException`, `IndexOutOfBoundsException`, or `NullPointerException` (Item 72). Once you've documented the restrictions on a method's parameters and you've documented the exceptions that will be thrown if these restrictions are violated, it is a simple matter to enforce the restrictions. Here's a typical example:

```
/**
 * Returns a BigInteger whose value is (this mod m).  This method
 * differs from the remainder method in that it always returns a
 * non-negative BigInteger.
 *
 * @param  m the modulus, which must be positive
 * @return this mod m
 * @throws ArithmeticException if m is less than or equal to 0
 */
public BigInteger mod(BigInteger m) {
    if (m.signum() <= 0)
        throw new ArithmeticException("Modulus <= 0: " + m);
    ... // Do the computation
}
```

Note that the doc comment does *not* say "`mod` throws `NullPointerException` if `m` is null," even though the method does exactly that, as a byproduct of invoking `m.signum()`. This exception *is* documented in the class-level doc comment for the enclosing `BigInteger` class. The class-level comment applies to all parameters in all of the class's public methods. This is a good way to avoid the clutter of documenting every `NullPointerException` on every method individually. It may be combined with the use of `@Nullable` or a similar annotation to indicate that a particular parameter may be null, but this practice is not standard, and multiple annotations are in use for this purpose.

The `Objects.requireNonNull` method, added in Java 7, is flexible and convenient, so there's no reason to perform null checks manually anymore. You can specify your own exception detail message if you wish. The method returns its input, so you can perform a null check at the same time as you use a value:

```
// Inline use of Java's null-checking facility
this.strategy = Objects.requireNonNull(strategy, "strategy");
```

You can also ignore the return value and use `Objects.requireNonNull` as a freestanding null check where that suits your needs.

In Java 9, a range-checking facility was added to `java.util.Objects`. This facility consists of three methods: `checkFromIndexSize`, `checkFromToIndex`, and `checkIndex`. This facility is not as flexible as the null-checking method. It doesn't let you specify your own exception detail message, and it is designed solely for use on list and array indices. It does not handle closed ranges (which contain both of their endpoints). But if it does what you need, it's a useful convenience.

For an unexported method, you, as the package author, control the circumstances under which the method is called, so you can and should ensure that only valid parameter values are ever passed in. Therefore, nonpublic methods can check their parameters using *assertions,* as shown below:

```
// Private helper function for a recursive sort
private static void sort(long a[], int offset, int length) {
    assert a != null;
    assert offset >= 0 && offset <= a.length;
    assert length >= 0 && length <= a.length - offset;
    ... // Do the computation
}
```

In essence, these assertions are claims that the asserted condition *will* be true, regardless of how the enclosing package is used by its clients. Unlike normal validity checks, assertions throw `AssertionError` if they fail. And unlike normal validity checks, they have no effect and essentially no cost unless you enable them, which you do by passing the `-ea` (or `-enableassertions`) flag to the `java` command. For more information on assertions, see the tutorial [Asserts].

It is particularly important to check the validity of parameters that are not used by a method, but stored for later use. For example, consider the static factory method on page 101, which takes an `int` array and returns a `List` view of the array. If a client were to pass in `null`, the method would throw a `NullPointerException` because the method has an explicit check (the call to `Objects.requireNonNull`). Had the check been omitted, the method would return a reference to a newly created `List` instance that would throw a `NullPointerException` as soon as a client attempted to use it. By that time, the origin of the `List` instance might be difficult to determine, which could greatly complicate the task of debugging.

Constructors represent a special case of the principle that you should check the validity of parameters that are to be stored away for later use. It is critical to check the validity of constructor parameters to prevent the construction of an object that violates its class invariants.

There are exceptions to the rule that you should explicitly check a method's parameters before performing its computation. An important exception is the case

in which the validity check would be expensive or impractical *and* the check is performed implicitly in the process of doing the computation. For example, consider a method that sorts a list of objects, such as `Collections.sort(List)`. All of the objects in the list must be mutually comparable. In the process of sorting the list, every object in the list will be compared to some other object in the list. If the objects aren't mutually comparable, one of these comparisons will throw a `ClassCastException`, which is exactly what the `sort` method should do. Therefore, there would be little point in checking ahead of time that the elements in the list were mutually comparable. Note, however, that indiscriminate reliance on implicit validity checks can result in the loss of *failure atomicity* (Item 76).

Occasionally, a computation implicitly performs a required validity check but throws the wrong exception if the check fails. In other words, the exception that the computation would naturally throw as the result of an invalid parameter value doesn't match the exception that the method is documented to throw. Under these circumstances, you should use the *exception translation* idiom, described in Item 73, to translate the natural exception into the correct one.

Do not infer from this item that arbitrary restrictions on parameters are a good thing. On the contrary, you should design methods to be as general as it is practical to make them. The fewer restrictions that you place on parameters, the better, assuming the method can do something reasonable with all of the parameter values that it accepts. Often, however, some restrictions are intrinsic to the abstraction being implemented.

To summarize, each time you write a method or constructor, you should think about what restrictions exist on its parameters. You should document these restrictions and enforce them with explicit checks at the beginning of the method body. It is important to get into the habit of doing this. The modest work that it entails will be paid back with interest the first time a validity check fails.

Item 50: Make defensive copies when needed

One thing that makes Java a pleasure to use is that it is a *safe language*. This means that in the absence of native methods it is immune to buffer overruns, array overruns, wild pointers, and other memory corruption errors that plague unsafe languages such as C and C++. In a safe language, it is possible to write classes and to know with certainty that their invariants will hold, no matter what happens in any other part of the system. This is not possible in languages that treat all of memory as one giant array.

Even in a safe language, you aren't insulated from other classes without some effort on your part. **You must program defensively, with the assumption that clients of your class will do their best to destroy its invariants.** This is increasingly true as people try harder to break the security of systems, but more commonly, your class will have to cope with unexpected behavior resulting from the honest mistakes of well-intentioned programmers. Either way, it is worth taking the time to write classes that are robust in the face of ill-behaved clients.

While it is impossible for another class to modify an object's internal state without some assistance from the object, it is surprisingly easy to provide such assistance without meaning to do so. For example, consider the following class, which purports to represent an immutable time period:

```
// Broken "immutable" time period class
public final class Period {
    private final Date start;
    private final Date end;

    /**
     * @param  start the beginning of the period
     * @param  end the end of the period; must not precede start
     * @throws IllegalArgumentException if start is after end
     * @throws NullPointerException if start or end is null
     */
    public Period(Date start, Date end) {
        if (start.compareTo(end) > 0)
            throw new IllegalArgumentException(
                start + " after " + end);
        this.start = start;
        this.end   = end;
    }

    public Date start() {
        return start;
    }
```

```
    public Date end() {
        return end;
    }

    ...  // Remainder omitted
}
```

At first glance, this class may appear to be immutable and to enforce the invariant that the start of a period does not follow its end. It is, however, easy to violate this invariant by exploiting the fact that `Date` is mutable:

```
// Attack the internals of a Period instance
Date start = new Date();
Date end = new Date();
Period p = new Period(start, end);
end.setYear(78);  // Modifies internals of p!
```

As of Java 8, the obvious way to fix this problem is to use `Instant` (or `LocalDateTime` or `ZonedDateTime`) in place of a `Date` because `Instant` (and the other `java.time` classes) are immutable (Item 17). **`Date` is obsolete and should no longer be used in new code.** That said, the problem still exists: there are times when you'll have to use mutable value types in your APIs and internal representations, and the techniques discussed in this item are appropriate for those times.

To protect the internals of a `Period` instance from this sort of attack, **it is essential to make a *defensive copy* of each mutable parameter to the constructor** and to use the copies as components of the `Period` instance in place of the originals:

```
// Repaired constructor - makes defensive copies of parameters
public Period(Date start, Date end) {
    this.start = new Date(start.getTime());
    this.end   = new Date(end.getTime());

    if (this.start.compareTo(this.end) > 0)
      throw new IllegalArgumentException(
          this.start + " after " + this.end);
}
```

With the new constructor in place, the previous attack will have no effect on the `Period` instance. Note that **defensive copies are made *before* checking the validity of the parameters (Item 49), and the validity check is performed on the copies rather than on the originals.** While this may seem unnatural, it is necessary. It protects the class against changes to the parameters from another

thread during the *window of vulnerability* between the time the parameters are checked and the time they are copied. In the computer security community, this is known as a *time-of-check/time-of-use* or *TOCTOU* attack [Viega01].

Note also that we did not use `Date`'s `clone` method to make the defensive copies. Because `Date` is nonfinal, the `clone` method is not guaranteed to return an object whose class is `java.util.Date`: it could return an instance of an untrusted subclass that is specifically designed for malicious mischief. Such a subclass could, for example, record a reference to each instance in a private static list at the time of its creation and allow the attacker to access this list. This would give the attacker free rein over all instances. To prevent this sort of attack, **do not use the `clone` method to make a defensive copy of a parameter whose type is subclassable by untrusted parties.**

While the replacement constructor successfully defends against the previous attack, it is still possible to mutate a `Period` instance, because its accessors offer access to its mutable internals:

```
// Second attack on the internals of a Period instance
Date start = new Date();
Date end = new Date();
Period p = new Period(start, end);
p.end().setYear(78);  // Modifies internals of p!
```

To defend against the second attack, merely modify the accessors to **return defensive copies of mutable internal fields:**

```
// Repaired accessors - make defensive copies of internal fields
public Date start() {
    return new Date(start.getTime());
}

public Date end() {
    return new Date(end.getTime());
}
```

With the new constructor and the new accessors in place, `Period` is truly immutable. No matter how malicious or incompetent a programmer, there is simply no way to violate the invariant that the start of a period does not follow its end (without resorting to extralinguistic means such as native methods and reflection). This is true because there is no way for any class other than `Period` itself to gain access to either of the mutable fields in a `Period` instance. These fields are truly encapsulated within the object.

In the accessors, unlike the constructor, it would be permissible to use the `clone` method to make the defensive copies. This is so because we know that the class of `Period`'s internal `Date` objects is `java.util.Date`, and not some untrusted subclass. That said, you are generally better off using a constructor or static factory to copy an instance, for reasons outlined in Item 13.

Defensive copying of parameters is not just for immutable classes. Any time you write a method or constructor that stores a reference to a client-provided object in an internal data structure, think about whether the client-provided object is potentially mutable. If it is, think about whether your class could tolerate a change in the object after it was entered into the data structure. If the answer is no, you must defensively copy the object and enter the copy into the data structure in place of the original. For example, if you are considering using a client-provided object reference as an element in an internal `Set` instance or as a key in an internal `Map` instance, you should be aware that the invariants of the set or map would be corrupted if the object were modified after it is inserted.

The same is true for defensive copying of internal components prior to returning them to clients. Whether or not your class is immutable, you should think twice before returning a reference to an internal component that is mutable. Chances are, you should return a defensive copy. Remember that nonzero-length arrays are always mutable. Therefore, you should always make a defensive copy of an internal array before returning it to a client. Alternatively, you could return an immutable view of the array. Both of these techniques are shown in Item 15.

Arguably, the real lesson in all of this is that you should, where possible, use immutable objects as components of your objects so that you that don't have to worry about defensive copying (Item 17). In the case of our `Period` example, use `Instant` (or `LocalDateTime` or `ZonedDateTime`), unless you're using a release prior to Java 8. If you are using an earlier release, one option is to store the primitive `long` returned by `Date.getTime()` in place of a `Date` reference.

There may be a performance penalty associated with defensive copying and it isn't always justified. If a class trusts its caller not to modify an internal component, perhaps because the class and its client are both part of the same package, then it may be appropriate to dispense with defensive copying. Under these circumstances, the class documentation should make it clear that the caller must not modify the affected parameters or return values.

Even across package boundaries, it is not always appropriate to make a defensive copy of a mutable parameter before integrating it into an object. There are some methods and constructors whose invocation indicates an explicit *handoff* of the object referenced by a parameter. When invoking such a method, the client

promises that it will no longer modify the object directly. A method or constructor that expects to take ownership of a client-provided mutable object must make this clear in its documentation.

Classes containing methods or constructors whose invocation indicates a transfer of control cannot defend themselves against malicious clients. Such classes are acceptable only when there is mutual trust between a class and its client or when damage to the class's invariants would harm no one but the client. An example of the latter situation is the wrapper class pattern (Item 18). Depending on the nature of the wrapper class, the client could destroy the class's invariants by directly accessing an object after it has been wrapped, but this typically would harm only the client.

In summary, if a class has mutable components that it gets from or returns to its clients, the class must defensively copy these components. If the cost of the copy would be prohibitive *and* the class trusts its clients not to modify the components inappropriately, then the defensive copy may be replaced by documentation outlining the client's responsibility not to modify the affected components.

Item 51: Design method signatures carefully

This item is a grab bag of API design hints that don't quite deserve items of their own. Taken together, they'll help make your API easier to learn and use and less prone to errors.

Choose method names carefully. Names should always obey the standard naming conventions (Item 68). Your primary goal should be to choose names that are understandable and consistent with other names in the same package. Your secondary goal should be to choose names consistent with the broader consensus, where it exists. Avoid long method names. When in doubt, look to the Java library APIs for guidance. While there are plenty of inconsistencies—inevitable, given the size and scope of these libraries—there is also a fair amount of consensus.

Don't go overboard in providing convenience methods. Every method should "pull its weight." Too many methods make a class difficult to learn, use, document, test, and maintain. This is doubly true for interfaces, where too many methods complicate life for implementors as well as users. For each action supported by your class or interface, provide a fully functional method. Consider providing a "shorthand" only if it will be used often. **When in doubt, leave it out.**

Avoid long parameter lists. Aim for four parameters or fewer. Most programmers can't remember longer parameter lists. If many of your methods exceed this limit, your API won't be usable without constant reference to its documentation. Modern IDEs help, but you are still much better off with short parameter lists. **Long sequences of identically typed parameters are especially harmful.** Not only won't users be able to remember the order of the parameters, but when they transpose parameters accidentally, their programs will still compile and run. They just won't do what their authors intended.

There are three techniques for shortening overly long parameter lists. One is to break the method up into multiple methods, each of which requires only a subset of the parameters. If done carelessly, this can lead to too many methods, but it can also help *reduce* the method count by increasing orthogonality. For example, consider the `java.util.List` interface. It does not provide methods to find the first or last index of an element in a sublist, both of which would require three parameters. Instead it provides the `subList` method, which takes two parameters and returns a *view* of a sublist. This method can be combined with the `indexOf` or `lastIndexOf` method, each of which has a single parameter, to yield the desired functionality. Moreover, the `subList` method can be combined with *any* method that operates on a `List` instance to perform arbitrary computations on sublists. The resulting API has a very high power-to-weight ratio.

A second technique for shortening long parameter lists is to create *helper classes* to hold groups of parameters. Typically these helper classes are static member classes (Item 24). This technique is recommended if a frequently occurring sequence of parameters is seen to represent some distinct entity. For example, suppose you are writing a class representing a card game, and you find yourself constantly passing a sequence of two parameters representing a card's rank and its suit. Your API, as well as the internals of your class, would probably benefit if you added a helper class to represent a card and replaced every occurrence of the parameter sequence with a single parameter of the helper class.

A third technique that combines aspects of the first two is to adapt the Builder pattern (Item 2) from object construction to method invocation. If you have a method with many parameters, especially if some of them are optional, it can be beneficial to define an object that represents all of the parameters and to allow the client to make multiple "setter" calls on this object, each of which sets a single parameter or a small, related group. Once the desired parameters have been set, the client invokes the object's "execute" method, which does any final validity checks on the parameters and performs the actual computation.

For parameter types, favor interfaces over classes (Item 64). If there is an appropriate interface to define a parameter, use it in favor of a class that implements the interface. For example, there is no reason to ever write a method that takes `HashMap` on input—use `Map` instead. This lets you pass in a `HashMap`, a `TreeMap`, a `ConcurrentHashMap`, a submap of a `TreeMap`, or any `Map` implementation yet to be written. By using a class instead of an interface, you restrict your client to a particular implementation and force an unnecessary and potentially expensive copy operation if the input data happens to exist in some other form.

Prefer two-element enum types to `boolean` parameters, unless the meaning of the boolean is clear from the method name. Enums make your code easier to read and to write. Also, they make it easy to add more options later. For example, you might have a `Thermometer` type with a static factory that takes this enum:

```
public enum TemperatureScale { FAHRENHEIT, CELSIUS }
```

Not only does `Thermometer.newInstance(TemperatureScale.CELSIUS)` make a lot more sense than `Thermometer.newInstance(true)`, but you can add `KELVIN` to `TemperatureScale` in a future release without having to add a new static factory to `Thermometer`. Also, you can refactor temperature-scale dependencies into methods on the enum constants (Item 34). For example, each scale constant could have a method that took a `double` value and converted it to Celsius.

Item 52: Use overloading judiciously

The following program is a well-intentioned attempt to classify collections according to whether they are sets, lists, or some other kind of collection:

```
// Broken! - What does this program print?
public class CollectionClassifier {
    public static String classify(Set<?> s) {
        return "Set";
    }

    public static String classify(List<?> lst) {
        return "List";
    }

    public static String classify(Collection<?> c) {
        return "Unknown Collection";
    }

    public static void main(String[] args) {
        Collection<?>[] collections = {
            new HashSet<String>(),
            new ArrayList<BigInteger>(),
            new HashMap<String, String>().values()
        };

        for (Collection<?> c : collections)
            System.out.println(classify(c));
    }
}
```

You might expect this program to print `Set`, followed by `List` and `Unknown Collection`, but it doesn't. It prints `Unknown Collection` three times. Why does this happen? Because the `classify` method is *overloaded*, and **the choice of which overloading to invoke is made at compile time.** For all three iterations of the loop, the compile-time type of the parameter is the same: `Collection<?>`. The runtime type is different in each iteration, but this does not affect the choice of overloading. Because the compile-time type of the parameter is `Collection<?>`, the only applicable overloading is the third one, `classify(Collection<?>)`, and this overloading is invoked in each iteration of the loop.

The behavior of this program is counterintuitive because **selection among overloaded methods is static, while selection among overridden methods is dynamic.** The correct version of an *overridden* method is chosen at runtime,

based on the runtime type of the object on which the method is invoked. As a reminder, a method is overridden when a subclass contains a method declaration with the same signature as a method declaration in an ancestor. If an instance method is overridden in a subclass and this method is invoked on an instance of the subclass, the subclass's *overriding method* executes, regardless of the compile-time type of the subclass instance. To make this concrete, consider the following program:

```
class Wine {
    String name() { return "wine"; }
}

class SparklingWine extends Wine {
    @Override String name() { return "sparkling wine"; }
}

class Champagne extends SparklingWine {
    @Override String name() { return "champagne"; }
}

public class Overriding {
    public static void main(String[] args) {
        List<Wine> wineList = List.of(
            new Wine(), new SparklingWine(), new Champagne());

        for (Wine wine : wineList)
            System.out.println(wine.name());
    }
}
```

The `name` method is declared in class `Wine` and overridden in subclasses `SparklingWine` and `Champagne`. As you would expect, this program prints out `wine`, `sparkling wine`, and `champagne`, even though the compile-time type of the instance is `Wine` in each iteration of the loop. The compile-time type of an object has no effect on which method is executed when an overridden method is invoked; the "most specific" overriding method always gets executed. Compare this to overloading, where the runtime type of an object has no effect on which overloading is executed; the selection is made at compile time, based entirely on the compile-time types of the parameters.

In the `CollectionClassifier` example, the intent of the program was to discern the type of the parameter by dispatching automatically to the appropriate method overloading based on the runtime type of the parameter, just as the `name` method did in the `Wine` example. Method overloading simply does not provide this

functionality. Assuming a static method is required, the best way to fix the `CollectionClassifier` program is to replace all three overloadings of `classify` with a single method that does explicit `instanceof` tests:

```
public static String classify(Collection<?> c) {
    return c instanceof Set  ? "Set" :
           c instanceof List ? "List" : "Unknown Collection";
}
```

Because overriding is the norm and overloading is the exception, overriding sets people's expectations for the behavior of method invocation. As demonstrated by the `CollectionClassifier` example, overloading can easily confound these expectations. It is bad practice to write code whose behavior is likely to confuse programmers. This is especially true for APIs. If the typical user of an API does not know which of several method overloadings will get invoked for a given set of parameters, use of the API is likely to result in errors. These errors will likely manifest themselves as erratic behavior at runtime, and many programmers will have a hard time diagnosing them. Therefore you should **avoid confusing uses of overloading.**

Exactly what constitutes a confusing use of overloading is open to some debate. **A safe, conservative policy is never to export two overloadings with the same number of parameters.** If a method uses varargs, a conservative policy is not to overload it at all, except as described in Item 53. If you adhere to these restrictions, programmers will never be in doubt as to which overloading applies to any set of actual parameters. These restrictions are not terribly onerous because **you can always give methods different names instead of overloading them.**

For example, consider the `ObjectOutputStream` class. It has a variant of its `write` method for every primitive type and for several reference types. Rather than overloading the `write` method, these variants all have different names, such as `writeBoolean(boolean)`, `writeInt(int)`, and `writeLong(long)`. An added benefit of this naming pattern, when compared to overloading, is that it is possible to provide read methods with corresponding names, for example, `readBoolean()`, `readInt()`, and `readLong()`. The `ObjectInputStream` class does, in fact, provide such read methods.

For constructors, you don't have the option of using different names: multiple constructors for a class are *always* overloaded. You do, in many cases, have the option of exporting static factories instead of constructors (Item 1). Also, with constructors you don't have to worry about interactions between overloading and overriding, because constructors can't be overridden. You will probably have

occasion to export multiple constructors with the same number of parameters, so it pays to know how to do it safely.

Exporting multiple overloadings with the same number of parameters is unlikely to confuse programmers *if* it is always clear which overloading will apply to any given set of actual parameters. This is the case when at least one corresponding formal parameter in each pair of overloadings has a "radically different" type in the two overloadings. Two types are radically different if it is clearly impossible to cast any non-null expression to both types. Under these circumstances, which overloading applies to a given set of actual parameters is fully determined by the runtime types of the parameters and cannot be affected by their compile-time types, so a major source of confusion goes away. For example, `ArrayList` has one constructor that takes an `int` and a second constructor that takes a `Collection`. It is hard to imagine any confusion over which of these two constructors will be invoked under any circumstances.

Prior to Java 5, all primitive types were radically different from all reference types, but this is not true in the presence of autoboxing, and it has caused real trouble. Consider the following program:

```
public class SetList {
    public static void main(String[] args) {
        Set<Integer> set = new TreeSet<>();
        List<Integer> list = new ArrayList<>();

        for (int i = -3; i < 3; i++) {
            set.add(i);
            list.add(i);
        }
        for (int i = 0; i < 3; i++) {
            set.remove(i);
            list.remove(i);
        }
        System.out.println(set + " " + list);
    }
}
```

First, the program adds the integers from −3 to 2, inclusive, to a sorted set and a list. Then, it makes three identical calls to `remove` on the set and the list. If you're like most people, you'd expect the program to remove the non-negative values (0, 1, and 2) from the set and the list and to print `[-3, -2, -1] [-3, -2, -1]`. In fact, the program removes the non-negative values from the set and the odd values from the list and prints `[-3, -2, -1] [-2, 0, 2]`. It is an understatement to call this behavior confusing.

Here's what's happening: The call to `set.remove(i)` selects the overloading `remove(E)`, where `E` is the element type of the set (`Integer`), and autoboxes `i` from `int` to `Integer`. This is the behavior you'd expect, so the program ends up removing the positive values from the set. The call to `list.remove(i)`, on the other hand, selects the overloading `remove(int i)`, which removes the element at the specified *position* in the list. If you start with the list `[-3, -2, -1, 0, 1, 2]` and remove the zeroth element, then the first, and then the second, you're left with `[-2, 0, 2]`, and the mystery is solved. To fix the problem, cast `list.remove`'s argument to `Integer`, forcing the correct overloading to be selected. Alternatively, you could invoke `Integer.valueOf` on `i` and pass the result to `list.remove`. Either way, the program prints `[-3, -2, -1] [-3, -2, -1]`, as expected:

```
for (int i = 0; i < 3; i++) {
    set.remove(i);
    list.remove((Integer) i);  // or remove(Integer.valueOf(i))
}
```

The confusing behavior demonstrated by the previous example came about because the `List<E>` interface has two overloadings of the `remove` method: `remove(E)` and `remove(int)`. Prior to Java 5 when the `List` interface was "generified," it had a `remove(Object)` method in place of `remove(E)`, and the corresponding parameter types, `Object` and `int`, were radically different. But in the presence of generics and autoboxing, the two parameter types are no longer radically different. In other words, adding generics and autoboxing to the language damaged the `List` interface. Luckily, few if any other APIs in the Java libraries were similarly damaged, but this tale makes it clear that autoboxing and generics increased the importance of caution when overloading.

The addition of lambdas and method references in Java 8 further increased the potential for confusion in overloading. For example, consider these two snippets:

```
new Thread(System.out::println).start();

ExecutorService exec = Executors.newCachedThreadPool();
exec.submit(System.out::println);
```

While the `Thread` constructor invocation and the `submit` method invocation look similar, the former compiles while the latter does not. The arguments are identical (`System.out::println`), and both the constructor and the method have an overloading that takes a `Runnable`. What's going on here? The surprising answer is that the `submit` method has an overloading that takes a `Callable<T>`, while the `Thread` constructor does not. You might think that this shouldn't make any

difference because all overloadings of `println` return `void`, so the method reference couldn't possibly be a `Callable`. This makes perfect sense, but it's not the way the overload resolution algorithm works. Perhaps equally surprising is that the `submit` method invocation would be legal if the `println` method weren't also overloaded. It is the combination of the overloading of the referenced method (`println`) and the invoked method (`submit`) that prevents the overload resolution algorithm from behaving as you'd expect.

Technically speaking, the problem is that `System.out::println` is an *inexact method reference* [JLS, 15.13.1] and that "certain argument expressions that contain implicitly typed lambda expressions or inexact method references are ignored by the applicability tests, because their meaning cannot be determined until a target type is selected [JLS, 15.12.2]." Don't worry if you don't understand this passage; it is aimed at compiler writers. The key point is that overloading methods or constructors with different functional interfaces in the same argument position causes confusion. Therefore, **do not overload methods to take different functional interfaces in the same argument position.** In the parlance of this item, different functional interfaces are not radically different. The Java compiler will warn you about this sort of problematic overload if you pass the command line switch `-Xlint:overloads`.

Array types and class types other than `Object` are radically different. Also, array types and interface types other than `Serializable` and `Cloneable` are radically different. Two distinct classes are said to be *unrelated* if neither class is a descendant of the other [JLS, 5.5]. For example, `String` and `Throwable` are unrelated. It is impossible for any object to be an instance of two unrelated classes, so unrelated classes are radically different, too.

There are other pairs of types that can't be converted in either direction [JLS, 5.1.12], but once you go beyond the simple cases described above, it becomes very difficult for most programmers to discern which, if any, overloading applies to a set of actual parameters. The rules that determine which overloading is selected are extremely complex and grow more complex with every release. Few programmers understand all of their subtleties.

There may be times when you feel the need to violate the guidelines in this item, especially when evolving existing classes. For example, consider `String`, which has had a `contentEquals(StringBuffer)` method since Java 4. In Java 5, `CharSequence` was added to provide a common interface for `StringBuffer`, `StringBuilder`, `String`, `CharBuffer`, and other similar types. At the same time that `CharSequence` was added, `String` was outfitted with an overloading of the `contentEquals` method that takes a `CharSequence`.

While the resulting overloading clearly violates the guidelines in this item, it causes no harm because both overloaded methods do exactly the same thing when they are invoked on the same object reference. The programmer may not know which overloading will be invoked, but it is of no consequence so long as they behave identically. The standard way to ensure this behavior is to have the more specific overloading forward to the more general:

```
// Ensuring that 2 methods have identical behavior by forwarding
public boolean contentEquals(StringBuffer sb) {
    return contentEquals((CharSequence) sb);
}
```

While the Java libraries largely adhere to the spirit of the advice in this item, there are a number of classes that violate it. For example, `String` exports two overloaded static factory methods, `valueOf(char[])` and `valueOf(Object)`, that do completely different things when passed the same object reference. There is no real justification for this, and it should be regarded as an anomaly with the potential for real confusion.

To summarize, just because you can overload methods doesn't mean you should. It is generally best to refrain from overloading methods with multiple signatures that have the same number of parameters. In some cases, especially where constructors are involved, it may be impossible to follow this advice. In these cases, you should at least avoid situations where the same set of parameters can be passed to different overloadings by the addition of casts. If this cannot be avoided, for example, because you are retrofitting an existing class to implement a new interface, you should ensure that all overloadings behave identically when passed the same parameters. If you fail to do this, programmers will be hard pressed to make effective use of the overloaded method or constructor, and they won't understand why it doesn't work.

Item 53: Use varargs judiciously

Varargs methods, formally known as *variable arity* methods [JLS, 8.4.1], accept zero or more arguments of a specified type. The varargs facility works by first creating an array whose size is the number of arguments passed at the call site, then putting the argument values into the array, and finally passing the array to the method.

For example, here is a varargs method that takes a sequence of `int` arguments and returns their sum. As you would expect, the value of `sum(1, 2, 3)` is `6`, and the value of `sum()` is `0`:

```
// Simple use of varargs
static int sum(int... args) {
    int sum = 0;
    for (int arg : args)
        sum += arg;
    return sum;
}
```

Sometimes it's appropriate to write a method that requires *one* or more arguments of some type, rather than *zero* or more. For example, suppose you want to write a function that computes the minimum of its arguments. This function is not well defined if the client passes no arguments. You could check the array length at runtime:

```
// The WRONG way to use varargs to pass one or more arguments!
static int min(int... args) {
    if (args.length == 0)
        throw new IllegalArgumentException("Too few arguments");
    int min = args[0];
    for (int i = 1; i < args.length; i++)
        if (args[i] < min)
            min = args[i];
    return min;
}
```

This solution has several problems. The most serious is that if the client invokes this method with no arguments, it fails at runtime rather than compile time. Another problem is that it is ugly. You have to include an explicit validity check on `args`, and you can't use a for-each loop unless you initialize `min` to `Integer.MAX_VALUE`, which is also ugly.

Luckily there's a much better way to achieve the desired effect. Declare the method to take two parameters, one normal parameter of the specified type and

one varargs parameter of this type. This solution corrects all the deficiencies of the previous one:

```
// The right way to use varargs to pass one or more arguments
static int min(int firstArg, int... remainingArgs) {
    int min = firstArg;
    for (int arg : remainingArgs)
        if (arg < min)
            min = arg;
    return min;
}
```

As you can see from this example, varargs are effective in circumstances where you want a method with a variable number of arguments. Varargs were designed for `printf`, which was added to the platform at the same time as varargs, and for the core reflection facility (Item 65), which was retrofitted. Both `printf` and reflection benefited enormously from varargs.

Exercise care when using varargs in performance-critical situations. Every invocation of a varargs method causes an array allocation and initialization. If you have determined empirically that you can't afford this cost but you need the flexibility of varargs, there is a pattern that lets you have your cake and eat it too. Suppose you've determined that 95 percent of the calls to a method have three or fewer parameters. Then declare five overloadings of the method, one each with zero through three ordinary parameters, and a single varargs method for use when the number of arguments exceeds three:

```
public void foo() { }
public void foo(int a1) { }
public void foo(int a1, int a2) { }
public void foo(int a1, int a2, int a3) { }
public void foo(int a1, int a2, int a3, int... rest) { }
```

Now you know that you'll pay the cost of the array creation only in the 5 percent of all invocations where the number of parameters exceeds three. Like most performance optimizations, this technique usually isn't appropriate, but when it is, it's a lifesaver.

The static factories for `EnumSet` use this technique to reduce the cost of creating enum sets to a minimum. This was appropriate because it was critical that enum sets provide a performance-competitive replacement for bit fields (Item 36).

In summary, varargs are invaluable when you need to define methods with a variable number of arguments. Precede the varargs parameter with any required parameters, and be aware of the performance consequences of using varargs.

Item 54: Return empty collections or arrays, not nulls

It is not uncommon to see methods that look something like this:

```
// Returns null to indicate an empty collection. Don't do this!
private final List<Cheese> cheesesInStock = ...;

/**
 * @return a list containing all of the cheeses in the shop,
 *     or null if no cheeses are available for purchase.
 */
public List<Cheese> getCheeses() {
    return cheesesInStock.isEmpty() ? null
        : new ArrayList<>(cheesesInStock);
}
```

There is no reason to special-case the situation where no cheeses are available for purchase. Doing so requires extra code in the client to handle the possibly null return value, for example:

```
List<Cheese> cheeses = shop.getCheeses();
if (cheeses != null && cheeses.contains(Cheese.STILTON))
    System.out.println("Jolly good, just the thing.");
```

This sort of circumlocution is required in nearly every use of a method that returns `null` in place of an empty collection or array. It is error-prone, because the programmer writing the client might forget to write the special-case code to handle a `null` return. Such an error may go unnoticed for years because such methods usually return one or more objects. Also, returning `null` in place of an empty container complicates the implementation of the method returning the container.

It is sometimes argued that a `null` return value is preferable to an empty collection or array because it avoids the expense of allocating the empty container. This argument fails on two counts. First, it is inadvisable to worry about performance at this level unless measurements have shown that the allocation in question is a real contributor to performance problems (Item 67). Second, it *is* possible to return empty collections and arrays without allocating them. Here is the typical code to return a possibly empty collection. Usually, this is all you need:

```
//The right way to return a possibly empty collection
public List<Cheese> getCheeses() {
    return new ArrayList<>(cheesesInStock);
}
```

In the unlikely event that you have evidence suggesting that allocating empty collections is harming performance, you can avoid the allocations by returning the

same *immutable* empty collection repeatedly, as immutable objects may be shared freely (Item 17). Here is the code to do it, using the `Collections.emptyList` method. If you were returning a set, you'd use `Collections.emptySet`; if you were returning a map, you'd use `Collections.emptyMap`. But remember, this is an optimization, and it's seldom called for. If you think you need it, measure performance before and after, to ensure that it's actually helping:

```
// Optimization - avoids allocating empty collections
public List<Cheese> getCheeses() {
    return cheesesInStock.isEmpty() ? Collections.emptyList()
        : new ArrayList<>(cheesesInStock);
}
```

The situation for arrays is identical to that for collections. Never return null instead of a zero-length array. Normally, you should simply return an array of the correct length, which may be zero. Note that we're passing a zero-length array into the `toArray` method to indicate the desired return type, which is `Cheese[]`:

```
//The right way to return a possibly empty array
public Cheese[] getCheeses() {
    return cheesesInStock.toArray(new Cheese[0]);
}
```

If you believe that allocating zero-length arrays is harming performance, you can return the same zero-length array repeatedly because all zero-length arrays are immutable:

```
// Optimization - avoids allocating empty arrays
private static final Cheese[] EMPTY_CHEESE_ARRAY = new Cheese[0];

public Cheese[] getCheeses() {
    return cheesesInStock.toArray(EMPTY_CHEESE_ARRAY);
}
```

In the optimized version, we pass *the same* empty array into every `toArray` call, and this array will be returned from `getCheeses` whenever `cheesesInStock` is empty. Do *not* preallocate the array passed to `toArray` in hopes of improving performance. Studies have shown that it is counterproductive [Shipilëv16]:

```
// Don't do this - preallocating the array harms performance!
return cheesesInStock.toArray(new Cheese[cheesesInStock.size()]);
```

In summary, **never return `null` in place of an empty array or collection.** It makes your API more difficult to use and more prone to error, and it has no performance advantages.

Item 55: Return optionals judiciously

Prior to Java 8, there were two approaches you could take when writing a method that was unable to return a value under certain circumstances. Either you could throw an exception, or you could return `null` (assuming the return type was an object reference type). Neither of these approaches is perfect. Exceptions should be reserved for exceptional conditions (Item 69), and throwing an exception is expensive because the entire stack trace is captured when an exception is created. Returning `null` doesn't have these shortcomings, but it has its own. If a method returns `null`, clients must contain special-case code to deal with the possibility of a null return, unless the programmer can *prove* that a null return is impossible. If a client neglects to check for a null return and stores a null return value away in some data structure, a `NullPointerException` may result at some arbitrary time in the future, at some place in the code that has nothing to do with the problem.

In Java 8, there is a third approach to writing methods that may not be able to return a value. The `Optional<T>` class represents an immutable container that can hold either a single non-null `T` reference or nothing at all. An optional that contains nothing is said to be *empty*. A value is said to be *present* in an optional that is not empty. An optional is essentially an immutable collection that can hold at most one element. `Optional<T>` does not implement `Collection<T>`, but it could in principle.

A method that conceptually returns a `T` but may be unable to do so under certain circumstances can instead be declared to return an `Optional<T>`. This allows the method to return an empty result to indicate that it couldn't return a valid result. An `Optional`-returning method is more flexible and easier to use than one that throws an exception, and it is less error-prone than one that returns `null`.

In Item 30, we showed this method to calculate the maximum value in a collection, according to its elements' natural order.

```
// Returns maximum value in collection - throws exception if empty
public static <E extends Comparable<E>> E max(Collection<E> c) {
    if (c.isEmpty())
        throw new IllegalArgumentException("Empty collection");

    E result = null;
    for (E e : c)
        if (result == null || e.compareTo(result) > 0)
            result = Objects.requireNonNull(e);

    return result;
}
```

This method throws an `IllegalArgumentException` if the given collection is empty. We mentioned in Item 30 that a better alternative would be to return `Optional<E>`. Here's how the method looks when it is modified to do so:

```
// Returns maximum value in collection as an Optional<E>
public static <E extends Comparable<E>>
        Optional<E> max(Collection<E> c) {
    if (c.isEmpty())
        return Optional.empty();

    E result = null;
    for (E e : c)
        if (result == null || e.compareTo(result) > 0)
            result = Objects.requireNonNull(e);

    return Optional.of(result);
}
```

As you can see, it is straightforward to return an optional. All you have to do is to create the optional with the appropriate static factory. In this program, we use two: `Optional.empty()` returns an empty optional, and `Optional.of(value)` returns an optional containing the given non-null value. It is a programming error to pass `null` to `Optional.of(value)`. If you do this, the method responds by throwing a `NullPointerException`. The `Optional.ofNullable(value)` method accepts a possibly null value and returns an empty optional if `null` is passed in. **Never return a null value from an `Optional`-returning method:** it defeats the entire purpose of the facility.

Many terminal operations on streams return optionals. If we rewrite the `max` method to use a stream, `Stream`'s `max` operation does the work of generating an optional for us (though we do have to pass in an explicit comparator):

```
// Returns max val in collection as Optional<E> - uses stream
public static <E extends Comparable<E>>
        Optional<E> max(Collection<E> c) {
    return c.stream().max(Comparator.naturalOrder());
}
```

So how do you choose to return an optional instead of returning a `null` or throwing an exception? **Optionals are similar in spirit to checked exceptions** (Item 71), in that they *force* the user of an API to confront the fact that there may be no value returned. Throwing an unchecked exception or returning a `null` allows the user to ignore this eventuality, with potentially dire consequences. However, throwing a checked exception requires additional boilerplate code in the client.

If a method returns an optional, the client gets to choose what action to take if the method can't return a value. You can specify a default value:

```
// Using an optional to provide a chosen default value
String lastWordInLexicon = max(words).orElse("No words...");
```

or you can throw any exception that is appropriate. Note that we pass in an exception factory rather than an actual exception. This avoids the expense of creating the exception unless it will actually be thrown:

```
// Using an optional to throw a chosen exception
Toy myToy = max(toys).orElseThrow(TemperTantrumException::new);
```

If you can *prove* that an optional is nonempty, you can get the value from the optional without specifying an action to take if the optional is empty, but if you're wrong, your code will throw a `NoSuchElementException`:

```
// Using optional when you know there's a return value
Element lastNobleGas = max(Elements.NOBLE_GASES).get();
```

Occasionally you may be faced with a situation where it's expensive to get the default value, and you want to avoid that cost unless it's necessary. For these situations, `Optional` provides a method that takes a `Supplier<T>` and invokes it only when necessary. This method is called `orElseGet`, but perhaps it should have been called `orElseCompute` because it is closely related to the three `Map` methods whose names begin with `compute`. There are several `Optional` methods for dealing with more specialized use cases: `filter`, `map`, `flatMap`, and `ifPresent`. In Java 9, two more of these methods were added: `or` and `ifPresentOrElse`. If the basic methods described above aren't a good match for your use case, look at the documentation for these more advanced methods and see if they do the job.

In case none of these methods meets your needs, `Optional` provides the `isPresent()` method, which may be viewed as a safety valve. It returns `true` if the optional contains a value, `false` if it's empty. You can use this method to perform any processing you like on an optional result, but make sure to use it wisely. Many uses of `isPresent` can profitably be replaced by one of the methods mentioned above. The resulting code will typically be shorter, clearer, and more idiomatic.

For example, consider this code snippet, which prints the process ID of the parent of a process, or `N/A` if the process has no parent. The snippet uses the `ProcessHandle` class, introduced in Java 9:

```
Optional<ProcessHandle> parentProcess = ph.parent();
System.out.println("Parent PID: " + (parentProcess.isPresent() ?
    String.valueOf(parentProcess.get().pid()) : "N/A"));
```

The code snippet above can be replaced by this one, which uses `Optional`'s `map` function:

```
System.out.println("Parent PID: " +
  ph.parent().map(h -> String.valueOf(h.pid())).orElse("N/A"));
```

When programming with streams, it is not uncommon to find yourself with a `Stream<Optional<T>>` and to require a `Stream<T>` containing all the elements in the nonempty optionals in order to proceed. If you're using Java 8, here's how to bridge the gap:

```
streamOfOptionals
    .filter(Optional::isPresent)
    .map(Optional::get)
```

In Java 9, `Optional` was outfitted with a `stream()` method. This method is an adapter that turns an `Optional` into a `Stream` containing an element if one is present in the optional, or none if it is empty. In conjunction with `Stream`'s `flatMap` method (Item 45), this method provides a concise replacement for the code snippet above:

```
streamOfOptionals.
    .flatMap(Optional::stream)
```

Not all return types benefit from the optional treatment. **Container types, including collections, maps, streams, arrays, and optionals should not be wrapped in optionals.** Rather than returning an empty `Optional<List<T>>`, you should simply return an empty `List<T>` (Item 54). Returning the empty container will eliminate the need for client code to process an optional. The `ProcessHandle` class does have the `arguments` method, which returns `Optional<String[]>`, but this method should be regarded as an anomaly that is not to be emulated.

So when should you declare a method to return `Optional<T>` rather than `T`? As a rule, **you should declare a method to return `Optional<T>` if it might not be able to return a result *and* clients will have to perform special processing if no result is returned.** That said, returning an `Optional<T>` is not without cost.

An `Optional` is an object that has to be allocated and initialized, and reading the value out of the optional requires an extra indirection. This makes optionals inappropriate for use in some performance-critical situations. Whether a particular method falls into this category can only be determined by careful measurement (Item 67).

Returning an optional that contains a boxed primitive type is prohibitively expensive compared to returning a primitive type because the optional has two levels of boxing instead of zero. Therefore, the library designers saw fit to provide analogues of `Optional<T>` for the primitive types `int`, `long`, and `double`. These optional types are `OptionalInt`, `OptionalLong`, and `OptionalDouble`. They contain most, but not all, of the methods on `Optional<T>`. Therefore, **you should never return an optional of a boxed primitive type,** with the possible exception of the "minor primitive types," `Boolean`, `Byte`, `Character`, `Short`, and `Float`.

Thus far, we have discussed returning optionals and processing them after they are returned. We have not discussed other possible uses, and that is because most other uses of optionals are suspect. For example, you should never use optionals as map values. If you do, you have two ways of expressing a key's logical absence from the map: either the key can be absent from the map, or it can be present and map to an empty optional. This represents needless complexity with great potential for confusion and errors. More generally, **it is almost never appropriate to use an optional as a key, value, or element in a collection or array.**

This leaves a big question unanswered. Is it ever appropriate to store an optional in an instance field? Often it's a "bad smell": it suggests that perhaps you should have a subclass containing the optional fields. But sometimes it may be justified. Consider the case of our `NutritionFacts` class in Item 2. A `NutritionFacts` instance contains many fields that are not required. You can't have a subclass for every possible combination of these fields. Also, the fields have primitive types, which make it awkward to express absence directly. The best API for `NutritionFacts` would return an optional from the getter for each optional field, so it makes good sense to simply store those optionals as fields in the object.

In summary, if you find yourself writing a method that can't always return a value and you believe it is important that users of the method consider this possibility every time they call it, then you should probably return an optional. You should, however, be aware that there are real performance consequences associated with returning optionals; for performance-critical methods, it may be better to return a `null` or throw an exception. Finally, you should rarely use an optional in any other capacity than as a return value.

Item 56: Write doc comments for all exposed API elements

If an API is to be usable, it must be documented. Traditionally, API documentation was generated manually, and keeping it in sync with code was a chore. The Java programming environment eases this task with the *Javadoc* utility. Javadoc generates API documentation automatically from source code with specially formatted *documentation comments*, more commonly known as *doc comments*.

While the doc comment conventions are not officially part of the language, they constitute a de facto API that every Java programmer should know. These conventions are described in the *How to Write Doc Comments* web page [Javadoc-guide]. While this page has not been updated since Java 4 was released, it is still an invaluable resource. One important doc tag was added in Java 9, `{@index}`; one in Java 8, `{@implSpec}`; and two in Java 5, `{@literal}` and `{@code}`. These tags are missing from the aforementioned web page, but are discussed in this item.

To document your API properly, you must precede *every* exported class, interface, constructor, method, and field declaration with a doc comment. If a class is serializable, you should also document its serialized form (Item 87). In the absence of a doc comment, the best that Javadoc can do is to reproduce the declaration as the sole documentation for the affected API element. It is frustrating and error-prone to use an API with missing documentation comments. Public classes should not use default constructors because there is no way to provide doc comments for them. To write maintainable code, you should also write doc comments for most unexported classes, interfaces, constructors, methods, and fields, though these comments needn't be as thorough as those for exported API elements.

The doc comment for a method should describe succinctly the contract between the method and its client. With the exception of methods in classes designed for inheritance (Item 19), the contract should say *what* the method does rather than *how* it does its job. The doc comment should enumerate all of the method's *preconditions*, which are the things that have to be true in order for a client to invoke it, and its *postconditions*, which are the things that will be true after the invocation has completed successfully. Typically, preconditions are described implicitly by the `@throws` tags for unchecked exceptions; each unchecked exception corresponds to a precondition violation. Also, preconditions can be specified along with the affected parameters in their `@param` tags.

In addition to preconditions and postconditions, methods should document any *side effects*. A side effect is an observable change in the state of the system that is not obviously required in order to achieve the postcondition. For example, if a method starts a background thread, the documentation should make note of it.

To describe a method's contract fully, the doc comment should have an `@param` tag for every parameter, an `@return` tag unless the method has a void return type, and an `@throws` tag for every exception thrown by the method, whether checked or unchecked (Item 74). If the text in the `@return` tag would be identical to the description of the method, it may be permissible to omit it, depending on the coding standards you are following.

By convention, the text following an `@param` tag or `@return` tag should be a noun phrase describing the value represented by the parameter or return value. Rarely, arithmetic expressions are used in place of noun phrases; see `BigInteger` for examples. The text following an `@throws` tag should consist of the word "if," followed by a clause describing the conditions under which the exception is thrown. By convention, the phrase or clause following an `@param`, `@return`, or `@throws` tag is not terminated by a period. All of these conventions are illustrated by the following doc comment:

```
/**
 * Returns the element at the specified position in this list.
 *
 * <p>This method is <i>not</i> guaranteed to run in constant
 * time. In some implementations it may run in time proportional
 * to the element position.
 *
 * @param  index index of element to return; must be
 *         non-negative and less than the size of this list
 * @return the element at the specified position in this list
 * @throws IndexOutOfBoundsException if the index is out of range
 *         ({@code index < 0 || index >= this.size()})
 */
E get(int index);
```

Notice the use of HTML tags in this doc comment (`<p>` and `<i>`). The Javadoc utility translates doc comments into HTML, and arbitrary HTML elements in doc comments end up in the resulting HTML document. Occasionally, programmers go so far as to embed HTML tables in their doc comments, although this is rare.

Also notice the use of the Javadoc `{@code}` tag around the code fragment in the `@throws` clause. This tag serves two purposes: it causes the code fragment to be rendered in `code font`, and it suppresses processing of HTML markup and nested Javadoc tags in the code fragment. The latter property is what allows us to use the less-than sign (`<`) in the code fragment even though it's an HTML metacharacter. To include a multiline code example in a doc comment, use a Javadoc `{@code}` tag wrapped inside an HTML `<pre>` tag. In other words, precede the code example with the characters `<pre>{@code` and follow it with `}</pre>`. This preserves line

breaks in the code, and eliminates the need to escape HTML metacharacters, but *not* the at sign (@), which must be escaped if the code sample uses annotations.

Finally, notice the use of the words "this list" in the doc comment. By convention, the word "this" refers to the object on which a method is invoked when it is used in the doc comment for an instance method.

As mentioned in Item 15, when you design a class for inheritance, you must document its *self-use patterns,* so programmers know the semantics of overriding its methods. These self-use patterns should be documented using the `@implSpec` tag, added in Java 8. Recall that ordinary doc comments describe the contract between a method and its client; `@implSpec` comments, by contrast, describe the contract between a method and its subclass, allowing subclasses to rely on implementation behavior if they inherit the method or call it via `super`. Here's how it looks in practice:

```
/**
 * Returns true if this collection is empty.
 *
 * @implSpec
 * This implementation returns {@code this.size() == 0}.
 *
 * @return true if this collection is empty
 */
public boolean isEmpty() { ... }
```

As of Java 9, the Javadoc utility still ignores the `@implSpec` tag unless you pass the command line switch `-tag "implSpec:a:Implementation Requirements:"`. Hopefully this will be remedied in a subsequent release.

Don't forget that you must take special action to generate documentation that contains HTML metacharacters, such as the less-than sign (<), the greater-than sign (>), and the ampersand (&). The best way to get these characters into documentation is to surround them with the `{@literal}` tag, which suppress processing of HTML markup and nested Javadoc tags. It is like the `{@code}` tag, except that it doesn't render the text in code font. For example, this Javadoc fragment:

```
* A geometric series converges if {@literal |r| < 1}.
```

generates the documentation: "A geometric series converges if |r| < 1." The `{@literal}` tag could have been placed around just the less-than sign rather than the entire inequality with the same resulting documentation, but the doc comment would have been less readable in the source code. This illustrates the general principle that **doc comments should be readable both in the source code and in the generated documentation.** If you can't achieve both, the readability of the generated documentation trumps that of the source code.

The first "sentence" of each doc comment (as defined below) becomes the *summary description* of the element to which the comment pertains. For example, the summary description in the doc comment on page 255 is "Returns the element at the specified position in this list." The summary description must stand on its own to describe the functionality of the element it summarizes. To avoid confusion, **no two members or constructors in a class or interface should have the same summary description.** Pay particular attention to overloadings, for which it is often natural to use the same first sentence (but unacceptable in doc comments).

Be careful if the intended summary description contains a period, because the period can prematurely terminate the description. For example, a doc comment that begins with the phrase "`A college degree, such as B.S., M.S. or Ph.D.`" will result in the summary description "A college degree, such as B.S., M.S." The problem is that the summary description ends at the first period that is followed by a space, tab, or line terminator (or at the first block tag) [Javadoc-ref]. Here, the second period in the abbreviation "M.S." is followed by a space. The best solution is to surround the offending period and any associated text with an `{@literal}` tag, so the period is no longer followed by a space in the source code:

```
/**
 * A college degree, such as B.S., {@literal M.S.} or Ph.D.
 */
public class Degree { ... }
```

It is a bit misleading to say that the summary description is the first *sentence* in a doc comment. Convention dictates that it should seldom be a complete sentence. For methods and constructors, the summary description should be a verb phrase (including any object) describing the action performed by the method. For example:

- `ArrayList(int initialCapacity)`—Constructs an empty list with the specified initial capacity.
- `Collection.size()`—Returns the number of elements in this collection.

As shown in these examples, use the third person declarative tense ("returns the number") rather than the second person imperative ("return the number").

For classes, interfaces, and fields, the summary description should be a noun phrase describing the thing represented by an instance of the class or interface or by the field itself. For example:

- `Instant`—An instantaneous point on the time-line.
- `Math.PI`—The `double` value that is closer than any other to pi, the ratio of the circumference of a circle to its diameter.

In Java 9, a client-side index was added to the HTML generated by Javadoc. This index, which eases the task of navigating large API documentation sets, takes the form of a search box in the upper-right corner of the page. When you type into the box, you get a drop-down menu of matching pages. API elements, such as classes, methods, and fields, are indexed automatically. Occasionally you may wish to index additional terms that are important to your API. The `{@index}` tag was added for this purpose. Indexing a term that appears in a doc comment is as simple as wrapping it in this tag, as shown in this fragment:

```
* This method complies with the {@index IEEE 754} standard.
```

Generics, enums, and annotations require special care in doc comments. **When documenting a generic type or method, be sure to document all type parameters:**

```
/**
 * An object that maps keys to values.  A map cannot contain
 * duplicate keys; each key can map to at most one value.
 *
 * (Remainder omitted)
 *
 * @param <K> the type of keys maintained by this map
 * @param <V> the type of mapped values
 */
public interface Map<K, V> { ... }
```

When documenting an enum type, be sure to document the constants as well as the type and any public methods. Note that you can put an entire doc comment on one line if it's short:

```
/**
 * An instrument section of a symphony orchestra.
 */
public enum OrchestraSection {
    /** Woodwinds, such as flute, clarinet, and oboe. */
    WOODWIND,

    /** Brass instruments, such as french horn and trumpet. */
    BRASS,

    /** Percussion instruments, such as timpani and cymbals. */
    PERCUSSION,

    /** Stringed instruments, such as violin and cello. */
    STRING;
}
```

When documenting an annotation type, be sure to document any members as well as the type itself. Document members with noun phrases, as if they were fields. For the summary description of the type, use a verb phrase that says what it means when a program element has an annotation of this type:

```
/**
 * Indicates that the annotated method is a test method that
 * must throw the designated exception to pass.
 */
@Retention(RetentionPolicy.RUNTIME)
@Target(ElementType.METHOD)
public @interface ExceptionTest {
    /**
     * The exception that the annotated test method must throw
     * in order to pass. (The test is permitted to throw any
     * subtype of the type described by this class object.)
     */
    Class<? extends Throwable> value();
}
```

Package-level doc comments should be placed in a file named `package-info.java`. In addition to these comments, `package-info.java` must contain a package declaration and may contain annotations on this declaration. Similarly, if you elect to use the module system (Item 15), module-level comments should be placed in the `module-info.java` file.

Two aspects of APIs that are often neglected in documentation are thread-safety and serializability. **Whether or not a class or static method is thread-safe, you should document its thread-safety** level, as described in Item 82. If a class is serializable, you should document its serialized form, as described in Item 87.

Javadoc has the ability to "inherit" method comments. If an API element does not have a doc comment, Javadoc searches for the most specific applicable doc comment, giving preference to interfaces over superclasses. The details of the search algorithm can be found in *The Javadoc Reference Guide* [Javadoc-ref]. You can also inherit *parts* of doc comments from supertypes using the `{@inheritDoc}` tag. This means, among other things, that classes can reuse doc comments from interfaces they implement, rather than copying these comments. This facility has the potential to reduce the burden of maintaining multiple sets of nearly identical doc comments, but it is tricky to use and has some limitations. The details are beyond the scope of this book.

One caveat should be added concerning documentation comments. While it is necessary to provide documentation comments for all exported API elements, it is not always sufficient. For complex APIs consisting of multiple interrelated classes, it is often necessary to supplement the documentation comments with an external document describing the overall architecture of the API. If such a document exists, the relevant class or package documentation comments should include a link to it.

Javadoc automatically checks for adherence to many of the recommendations in this item. In Java 7, the command line switch `-Xdoclint` was required to get this behavior. In Java 8 and 9, checking is enabled by default. IDE plug-ins such as checkstyle go further in checking for adherence to these recommendations [Burn01]. You can also reduce the likelihood of errors in doc comments by running the HTML files generated by Javadoc through an *HTML validity checker.* This will detect many incorrect uses of HTML tags. Several such checkers are available for download, and you can validate HTML on the web using the W3C markup validation service [W3C-validator]. When validating generated HTML, keep in mind that as of Java 9, Javadoc is capable of generating HTML5 as well as HTML 4.01, though it still generates HTML 4.01 by default. Use the `-html5` command line switch if you want Javadoc to generate HTML5.

The conventions described in this item cover the basics. Though it is fifteen years old at the time of this writing, the definitive guide to writing doc comments is still *How to Write Doc Comments* [Javadoc-guide].

If you adhere to the guidelines in this item, the generated documentation should provide a clear description of your API. The only way to know for sure, however, is to **read the web pages generated by the Javadoc utility.** It is worth doing this for every API that will be used by others. Just as testing a program almost inevitably results in some changes to the code, reading the documentation generally results in at least a few minor changes to the doc comments.

To summarize, documentation comments are the best, most effective way to document your API. Their use should be considered mandatory for all exported API elements. Adopt a consistent style that adheres to standard conventions. Remember that arbitrary HTML is permissible in documentation comments and that HTML metacharacters must be escaped.