CHAPTER 7

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 5, this chapter focuses on usability, robustness, and flexibility.

Item 38: 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.

For public methods, use the Javadoc `@throws` tag to document the exception that will be thrown if a restriction on parameter values is violated (Item 62). Typically the exception will be `IllegalArgumentException`, `IndexOutOfBounds-`

Exception, or NullPointerException (Item 60). 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
}
```

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 should generally 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 interpreter. For more information on assertions, see Sun's tutorial [Asserts].

It is particularly important to check the validity of parameters that are not used by a method but are stored away for later use. For example, consider the static factory method on page 95, which takes an int array and returns a List view of the array. If a client of this method were to pass in null, the method

would throw a `NullPointerException` because the method contains an explicit check. 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 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 validity 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 a loss of failure atomicity (Item 64).

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 61, 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 39: Make defensive copies when needed

One thing that makes Java such 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 remain true, 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 may actually be true if someone tries to break the security of your system, but more likely your class will have to cope with unexpected behavior resulting from honest mistakes on the part of programmers using your API. 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!
```

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(start +" after "+ 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 38), 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 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 reign 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 above 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. 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 potentially untrusted subclass. That said, you are generally better off using a constructor or static factory, for reasons outlined in Item 11.

Defensive copying of parameters is not just for immutable classes. Anytime you write a method or constructor that enters a client-provided object into 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 destroyed 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 13.

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 15). In the case of our `Period` example, it is worth pointing out that experienced programmers often use the primitive `long` returned by `Date.getTime()` as an internal time representation instead of using a `Date` reference. They do this primarily because `Date` is mutable.

Defensive copying can have a performance penalty associated with it and 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 must 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 the 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 16). 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 40: 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 56). 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. 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're 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` methods, 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 22). 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 52). 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 ever to write a method that takes `HashMap` on input—use `Map` instead. This lets you pass in a `Hashtable`, a `HashMap`, a `TreeMap`, 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. It makes your code easier to read and to write, especially if you're using an IDE that supports autocompletion. Also, it makes it easy to add more options later. For example, you might have a `Thermometer` type with a static factory that takes a value of 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 30). For example, each scale constant could have a method that took a `double` value and normalized it to Celsius.

Item 41: 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) {
        Wine[] wines = {
            new Wine(), new SparklingWine(), new Champagne()
        };
        for (Wine wine : wines)
            System.out.println(wine.name());
    }
}
```

The `name` method is declared in class `Wine` and overridden in classes `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 program is to replace all three overloadings of `classify` with a single method that does an explicit `instanceof` test:

```
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 be unable to diagnose 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 42. If you adhere to these restrictions, programmers will never be in doubt as to which overloading applies to any set of actual parameters. The restrictions are not terribly onerous because you can always give methods different names instead of overloading them.

For example, consider the class `ObjectOutputStream`. 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 have signatures like `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 an instance of either type to the other. 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 the 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 release 1.5, all primitive types were radically different from all reference types, but this is no longer 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<Integer>();
        List<Integer> list = new ArrayList<Integer>();

        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);
    }
}
```

The program adds the integers from -3 through 2 to a sorted set and to a list, and then makes three identical calls to `remove` on both 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* from a 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 release 1.5 when it was "generified," the `List` interface 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 it is even more important to overload with care now that autoboxing and generics are part of the language.

Array types and classes other than `Object` are radically different. Also, array types and interfaces 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.

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. They take up *thirty-three* pages in the language specification [JLS, 15.12.1-3], and 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, the `String` class has had a `contentEquals(StringBuffer)` method since release 1.4. In release 1.5, a new interface called `CharSequence` was added to provide a common interface for `StringBuffer`, `StringBuilder`, `String`, `CharBuffer`, and other similar types, all of which were retrofitted to implement this interface. At the same time that `CharSequence` was added to the platform, `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 as long as both overloaded methods always 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:

```
public boolean contentEquals(StringBuffer sb) {
    return contentEquals((CharSequence) sb);
}
```

While the Java platform libraries largely adhere to the spirit of the advice in this item, there are a number of classes that violate it. For example, the `String` class 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. You should generally 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 that case, you should at least avoid situations where the same set of parameters can be passed to different overloadings by the addition of casts. If such a situation 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 42: Use varargs judiciously

In release 1.5, varargs methods, formally known as *variable arity methods* [JLS, 8.4.1], were added to the language. Varargs methods 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 compute the minimum of a number of `int` 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 really do want a method with a variable number of arguments. Varargs were designed for `printf`, which was added to the platform in release 1.5, and for the core reflection facility (Item 53), which was retrofitted to take advantage of varargs in that release. Both `printf` and reflection benefit enormously from varargs.

You can retrofit an existing method that takes an array as its final parameter to take a varargs parameter instead with no effect on existing clients. But just because you can doesn't mean that you should! Consider the case of `Arrays.asList`. This method was never designed to gather multiple arguments into a list, but it seemed like a good idea to retrofit it to do so when varargs were added to the platform. As a result, it became possible to do things like this:

```
List<String> homophones = Arrays.asList("to", "too", "two");
```

This usage works, but it was a big mistake to enable it. Prior to release 1.5, this was a common idiom to print the contents of an array:

```
// Obsolete idiom to print an array!
System.out.println(Arrays.asList(myArray));
```

The idiom was necessary because arrays inherit their `toString` implementation from `Object`, so calling `toString` directly on an array produces a useless string such as `[Ljava.lang.Integer;@3e25a5`. The idiom worked only on arrays of

object reference types, but if you accidentally tried it on an array of primitives, the program wouldn't compile. For example, this program:

```
public static void main(String[] args) {
    int[] digits = { 3, 1, 4, 1, 5, 9, 2, 6, 5, 4 };
    System.out.println(Arrays.asList(digits));
}
```

would generate this error message in release 1.4:

```
Va.java:6: asList(Object[]) in Arrays can't be applied to (int[])
        System.out.println(Arrays.asList(digits));
                                         ^
```

Because of the unfortunate decision to retrofit `Arrays.asList` as a varargs method in release 1.5, this program now compiles without error or warning. Running the program, however, produces output that is both unintended and useless: `[[I@3e25a5]`. The `Arrays.asList` method, now "enhanced" to use varargs, gathers up the object reference to the `int` array `digits` into a one-element array of arrays and dutifully wraps it into a `List<int[]>` instance. Printing this list causes `toString` to be invoked on the list, which in turn causes `toString` to be invoked on its sole element, the `int` array, with the unfortunate result described above.

On the bright side, the `Arrays.asList` idiom for translating arrays to strings is now obsolete, and the current idiom is far more robust. Also in release 1.5, the `Arrays` class was given a full complement of `Arrays.toString` methods (not varargs methods!) designed specifically to translate arrays of any type into strings. If you use `Arrays.toString` in place of `Arrays.asList`, the program produces the intended result:

```
// The right way to print an array
System.out.println(Arrays.toString(myArray));
```

Instead of retrofitting `Arrays.asList`, it would have been better to add a new method to `Collections` specifically for the purpose of gathering its arguments into a list:

```
public static <T> List<T> gather(T... args) {
    return Arrays.asList(args);
}
```

Such a method would have provided the capability to gather without compromising the type-checking of the existing `Arrays.asList` method.

The lesson is clear. **Don't retrofit every method that has a final array parameter; use varargs *only* when a call really operates on a variable-length sequence of values.**

Two method signatures are particularly suspect:

```
ReturnType1 suspect1(Object... args) { }
<T> ReturnType2 suspect2(T... args) { }
```

Methods with either of these signatures will accept *any* parameter list. Any compile-time type-checking that you had prior to the retrofit will be lost, as demonstrated by what happened to `Arrays.asList`.

Exercise care when using the varargs facility 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 `EnumSet` class uses this technique for its static factories to reduce the cost of creating enum sets to a bare minimum. It was appropriate to do this because it was critical that enum sets provide performance-competitive replacements for bit fields (Item 32).

In summary, varargs methods are a convenient way to define methods that require a variable number of arguments, but they should not be overused. They can produce confusing results if used inappropriately.

Item 43: Return empty arrays or collections, not nulls

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

```
private final List<Cheese> cheesesInStock = ...;

/**
 * @return an array containing all of the cheeses in the shop,
 *     or null if no cheeses are available for purchase.
 */
public Cheese[] getCheeses() {
    if (cheesesInStock.size() == 0)
        return null;
    ...
}
```

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

```
Cheese[] cheeses = shop.getCheeses();
if (cheeses != null &&
    Arrays.asList(cheeses).contains(Cheese.STILTON))
    System.out.println("Jolly good, just the thing.");
```

instead of:

```
if (Arrays.asList(shop.getCheeses()).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 (zero-length) array or collection. 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, as such methods usually return one or more objects. Less significant, but still worthy of note, returning `null` in place of an empty array also complicates the method that returns the array or collection.

It is sometimes argued that a null return value is preferable to an empty array because it avoids the expense of allocating the array. This argument fails on two counts. First, it is inadvisable to worry about performance at this level unless profiling has shown that the method in question is a real contributor to performance problems (Item 55). Second, it is possible to return the same zero-length array

from every invocation that returns no items because zero-length arrays are immutable and immutable objects may be shared freely (Item 15). In fact, this is exactly what happens when you use the standard idiom for dumping items from a collection into a typed array:

```
// The right way to return an array from a collection
private final List<Cheese> cheesesInStock = ...;

private static final Cheese[] EMPTY_CHEESE_ARRAY = new Cheese[0];

/**
 * @return an array containing all of the cheeses in the shop.
 */
public Cheese[] getCheeses() {
    return cheesesInStock.toArray(EMPTY_CHEESE_ARRAY);
}
```

In this idiom, an empty-array constant is passed to the `toArray` method to indicate the desired return type. Normally the `toArray` method allocates the returned array, but if the collection is empty, it fits in the zero-length input array, and the specification for `Collection.toArray(T[])` guarantees that the input array will be returned if it is large enough to hold the collection. Therefore the idiom never allocates an empty array.

In similar fashion, a collection-valued method can be made to return the same immutable empty collection every time it needs to return an empty collection. The `Collections.emptySet`, `emptyList`, and `emptyMap` methods provide exactly what you need, as shown below:

```
// The right way to return a copy of a collection
public List<Cheese> getCheeseList() {
  if (cheesesInStock.isEmpty())
    return Collections.emptyList(); // Always returns same list
  else
    return new ArrayList<Cheese>(cheesesInStock);
}
```

In summary, **there is no reason ever to return `null` from an array- or collection-valued method instead of returning an empty array or collection.** The null-return idiom is likely a holdover from the C programming language, in which array lengths are returned separately from actual arrays. In C, there is no advantage to allocating an array if zero is returned as the length.

Item 44: 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*.

If you are not familiar with the doc comment conventions, you should learn them. While these conventions are not officially part of the language, they constitute a de facto API that every programmer should know. These conventions are described on Sun's *How to Write Doc Comments* Web page [Javadoc-guide]. While this page has not been updated since release 1.4, it is still an invaluable resource. Two important Javadoc tags were added to Javadoc in release 1.5, `{@literal}` and `{@code}` [Javadoc-5.0]. These tags 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 75). 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. To write maintainable code, you should also write doc comments for most unexported classes, interfaces, constructors, methods, and fields.

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 17), 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. Finally, documentation comments should describe the *thread safety* of a class or method, as discussed in Item 70.

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 62). 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. 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. Occasionally, arithmetic expressions are used in place of noun phrases. 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 short 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 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. Prior to release 1.5, code fragments were included in doc comments by using HTML tags and HTML escapes. **It is no longer necessary to use the HTML `<code>` or `<tt>` tags in doc comments: the Javadoc `{@code}` tag is preferable because it eliminates the need to escape HTML metacharacters.** 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 multiline code example with the characters <pre>{@code and follow it with the characters }</pre>.

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

Don't forget that you must take special action to generate documentation containing 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:

```
* The triangle inequality is {@literal |x + y| < |x| + |y|}.
```

produces the documentation: "The triangle inequality is |x + y| < |x| + |y|." 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 in both the source code and in the generated documentation. If you can't achieve both, generated documentation readability trumps source code readability.

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 204 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 in a prose description (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]. In this case, 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 a {@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.
 * College is a fountain of knowledge where many go to drink.
 */
public class Degree { ... }
```

It is somewhat 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 full 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.

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,

- `TimerTask`—A task that can be scheduled for one-time or repeated execution by a `Timer`.
- `Math.PI`—The `double` value that is closer than any other to pi, the ratio of the circumference of a circle to its diameter.

Three features added to the language in release 1.5 require special care in doc comments: generics, enums, and annotations. **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> {
    ... // Remainder omitted
}
```

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 succeed.
 */
@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 Exception> value();
}
```

As of release 1.5, package-level doc comments should be placed in a file called `package-info.java` instead of `package.html`. In addition to package-level doc comments, `package-info.java` can (but is not required to) contain a package declaration and package annotations.

Two aspects of a class's exported API that are often neglected are thread-safety and serializability. Whether or not a class is thread-safe, you should document its thread-safety level, as described in Item 70. If a class is serializable, you should document its serialized form, as described in Item 75.

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.

A simple way to reduce the likelihood of errors in documentation comments is to run the HTML files generated by Javadoc through an *HTML validity checker*. This will detect many incorrect uses of HTML tags, as well as HTML metacharacters that should have been escaped. Several HTML validity checkers are available for download and you can validate HTML online [W3C-validator].

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.

The conventions described in this item cover the basics. The definitive guide to writing doc comments is Sun's *How to Write Doc Comments* [Javadoc-guide]. There are IDE plug-ins that check for adherence to many of these rules [Burn01].

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 within documentation comments and that HTML metacharacters must be escaped.