CHAPTER 6 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 23: 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 nonnegative 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, and 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 44). Typically the exception will be `IllegalArgumentException`, `IndexOutOfBoundsException`, or `NullPointerException` (Item 42). 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
 * nonnegative BigInteger.
 *
 * @param  m the modulus, which must be positive.
 * @return this mod m.
 * @throws ArithmeticException if m <= 0.
 */
public BigInteger mod(BigInteger m) {
    if (m.signum() <= 0)
        throw new ArithmeticException("Modulus not positive");

    ... // 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* rather than normal checks. If you are using a release of the platform that supports assertions (1.4 or later), you should use the `assert` construct; otherwise you should use a makeshift assertion mechanism.

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 86, 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. If the check had 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, unfortunately, the origin of the `List` instance might be very 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 very important to check the validity of parameters to constructors to prevent the construction of an object that violates 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 application of this technique can result in a loss of failure atomicity (Item 46).

Occasionally, a computation implicitly performs the required validity check on some parameter but throws the wrong exception if the check fails. That is to say, the exception that the computation would naturally throw as the result of an invalid parameter value does not match the exception that you have documented the method to throw. Under these circumstances, you should use the *exception translation* idiom described in Item 43 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 24: Make defensive copies when needed

One thing that makes the Java programming language 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 the programmer 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 23), 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.

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 access to 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 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 (Date) start.clone();
}

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

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 he can 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.

Note that the new accessors, unlike the new constructor, do use the `clone` method to make defensive copies. This is acceptable (although not required), as we know with certainty that the class of `Period`'s internal `Date` objects is `java.util.Date` rather than some potentially untrusted subclass.

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 were 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 be returning a defensive copy. Also, it is critical to 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 to the user. Both of these techniques are shown in Item 12.

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 13). 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 rather than using a `Date` object reference. They do this primarily because `Date` is mutable.

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 control 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 14). 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.

Item 25: 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 38). 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 the libraries—there is also consensus. An invaluable resource is Patrick Chan's *The Java Developers Almanac* [Chan00], which contains the method declarations for every single method in the Java platform libraries, indexed alphabetically. If, for example, you were wondering whether to name a method `remove` or `delete`, a quick look at the index of this book would tell you that `remove` was the obvious choice. There are hundreds of methods whose names begin with `remove` and a small handful whose names begin with `delete`.

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 for users. For each action supported by your type, provide a fully functional method. Consider providing a "shorthand" for an operation only when it will be used frequently. **When in doubt, leave it out.**

Avoid long parameter lists. As a rule, three parameters should be viewed as a practical maximum, and fewer is better. 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. **Long sequences of identically typed parameters are especially harmful.** Not only won't the users of your API be able to remember the order of the parameters, but when they transpose parameters by mistake, their programs will still compile and run. They just won't do what their authors intended.

There are two 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 other 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 overly long parameter lists is to create *helper classes* to hold aggregates of parameters. Typically these helper classes are static member classes (Item 18). 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 be improved 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.

For parameter types, favor interfaces over classes. Whenever an appropriate interface to define a parameter exists, use it in favor of a class that implements the interface. For example, there is no reason ever to write a method that takes `Hashtable` 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 happen to exist in some other form.

Use *function objects* (Item 22) judiciously. There are some languages, notably Smalltalk and the various Lisp dialects, that encourage a style of programming rich in objects that represent functions to be applied to other objects. Programmers with experience in these languages may be tempted to adopt a similar style in the Java programming language, but it isn't a terribly good fit. The easiest way to create a function object is with an anonymous class (Item 18), but even that involves some syntactic clutter and has limitations in power and performance when compared to inline control constructs. Furthermore, the style of programming wherein you are constantly creating function objects and passing them from method to method is out of the mainstream, so other programmers will have a difficult time understanding your code if you adopt this style. This is not meant to imply that function objects don't have legitimate uses; they are essential to many powerful design patterns, such as *Strategy* [Gamma98, p. 315] and *Visitor*

[Gamma98, p. 331]. Rather, function objects should be used only with good reason.

Item 26: Use overloading judiciously

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

```
// Broken - incorrect use of overloading!
public class CollectionClassifier {
    public static String classify(Set s) {
        return "Set";
    }

    public static String classify(List l) {
        return "List";
    }

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

    public static void main(String[] args) {
        Collection[] tests = new Collection[] {
            new HashSet(),          // A Set
            new ArrayList(),        // A List
            new HashMap().values()  // Neither Set nor List
        };

        for (int i = 0; i < tests.length; i++)
            System.out.println(classify(tests[i]));
    }
}
```

You might expect this program to print "`Set`," followed by "`List`" and "`Unknown Collection`," but it doesn't; it prints out "`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 run-time 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 run time, based on the run-time 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 exactly 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 little program:

```
class A {
    String name() { return "A"; }
}

class B extends A {
    String name() { return "B"; }
}

class C extends A {
    String name() { return "C"; }
}

public class Overriding {
    public static void main(String[] args) {
        A[] tests = new A[] { new A(), new B(), new C() };

        for (int i = 0; i < tests.length; i++)
            System.out.print(tests[i].name());
    }
}
```

The `name` method is declared in class `A` and overridden in classes `B` and `C`. As you would expect, this program prints out "`ABC`" even though the compile-time type of the instance is `A` 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 run-time 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 run-time type of the parameter, just as the `name` method did in the "`ABC`" example. Method overloading simply does not provide this functionality. The 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 would not be obvious to the average programmer upon inspection. 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 run time, 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 you adhere to this restriction, programmers will never be in doubt as to which overloading applies to any set of parameters. This restriction is not terribly onerous because you can always give methods different names instead of overloading.

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 read methods with these names.

For constructors, you don't have the option of using different names; multiple constructors for a class are always overloaded. You do, in some cases, have the option of exporting static factories instead of constructors (Item 1), but that isn't always practical. On the bright side, with constructors you don't have to worry about interactions between overloading and overriding, as constructors can't be overridden. Because you'll probably have occasion to export multiple constructors with the same number of parameters, it pays to know when it is safe to do so.

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 run-time types of the parameters and cannot be affected by their compile-time types, so the major source of confusion evaporates.

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 because primitive types and reference types are radically different. Similarly, `BigInteger` has one constructor that takes a `byte` array and another that takes a `String`; this causes no confusion. Array types and classes other than `Object` are radically different. Also, array types and interfaces other than `Serializable` and `Cloneable` are radically different. Finally, `Throwable`, as of release 1.4, has one constructor that takes a `String` and another takes a `Throwable`. The classes `String` and `Throwable` are *unrelated*, which is to say that neither class is a descendant of the other. It is impossible for any object to be an instance of two unrelated classes, so unrelated classes are radically different.

There are a few additional examples of pairs of types that can't be converted in either direction [JLS, 5.1.7], but once you go beyond these simple cases, it can become very difficult for the average programmer to discern which, if any, overloading applies to a set of actual parameters. The specification that determines which overloading is selected is complex, and few programmers understand all of its subtleties [JLS, 15.12.1-3].

Occasionally you may be forced to violate the above guidelines when retrofitting existing classes to implement new interfaces. For example, many of the value types in the Java platform libraries had "self-typed" `compareTo` methods prior to the introduction of the `Comparable` interface. Here is the declaration for `String`'s original self-typed `compareTo` method:

```
public int compareTo(String s);
```

With the introduction of the `Comparable` interface, all of the these classes were retrofitted to implement this interface, which involved adding a more general `compareTo` method with this declaration:

```
public int compareTo(Object o);
```

While the resulting overloading is clearly a violation of the above guidelines, it causes no harm as long as both overloaded methods always do exactly the same thing when they are invoked on the same parameters. The programmer may not know which overloading will be invoked, but it is of no consequence as long as both methods return the same result. The standard way to ensure this behavior is to have the more general overloading forward to the more specific:

```
public int compareTo(Object o) {
    return compareTo((String) o);
}
```

A similar idiom is sometimes used for equals methods:

```
public boolean equals(Object o) {
    return o instanceof String && equals((String) o);
}
```

This idiom is harmless and may result in slightly improved performance if the compile-time type of the parameter matches the parameter of the more specific overloading. That said, it probably isn't worth doing as a matter of course (Item 37).

While the Java platform libraries largely adhere to the advice in this item, there are a number of places where it is violated. For example, the `String` class exports two overloaded static factory methods, `valueOf(char[])` and `val-ueOf(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 not be able to make effective use of the overloaded method or constructor, and they won’t understand why it doesn’t work.

Item 27: Return zero-length arrays, not nulls

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

```
private List 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(shop.getCheeses()).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 a zero length array. It is error prone, as 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 a zero length array also complicates the array-returning method itself.

It is sometimes argued that a null return value is preferable to a zero-length 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 37). Second, it is possible to return the same zero-length array from every invocation that returns no items because zero-length arrays are

methods.fm Page 136 Tuesday, May 15, 2001 9:02 AM

immutable and immutable objects may be shared freely (Item 13). In fact, this is exactly what happens when you use the standard idiom for dumping items from a collection into a typed array:

```
private List cheesesInStock = ...;

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

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

In this idiom, a zero-length 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 input array, and the specification for `Collection.toArray(Object[])` guarantees that the input array will be returned if it is large enough to hold the collection. Therefore the idiom never allocates a zero-length array but instead reuses the “type-specifier constant.”

In summary, **there is no reason ever to return `null` from an array-valued method instead of returning a zero-length array.** This 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 28: 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 documentation in sync with code was a big chore. The Java programming environment eases this task with a utility called *Javadoc*. This utility generates API documentation automatically from source code in conjunction with specially formatted *documentation comments*, more commonly known as *doc comments*. The Javadoc utility provides an easy and effective way to document your APIs, and its use is widespread.

If you are not already familiar with the doc comment conventions, you should learn them. While these conventions are not part of the Java programming language, they constitute a de facto API that every programmer should know. The conventions are defined *The Javadoc Tool Home Page* [Javadoc-b].

To document your API properly, you must precede every exported class, interface, constructor, method, and field declaration with a doc comment, subject to one exception discussed at the end of this item. 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 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 15), 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 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, as discussed in Item 52.

To describe its contract fully, the doc comment for a method should have a `@param` tag for every parameter, a `@return` tag unless the method has a void return type, and a `@throws` tag for every exception thrown by the method, whether checked or unchecked (Item 44). By convention the text following a `@param` tag or `@return` tag should be a noun phrase describing the value represented by the parameter or return value. The text following a `@throws` tag should consist of the word "if," followed by a noun phrase describing the conditions under which the exception is thrown. Occasionally, arithmetic expressions are used in place of noun phrases. All of these conventions are illustrated in the following short doc comment, which comes from the `List` interface:

```
/**
 * Returns the element at the specified position in this list.
 *
 * @param  index index of element to return; must be
 *         nonnegative 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
 *         (<tt>index &lt; 0 || index &gt;= this.size()</tt>).
 */
Object get(int index);
```

Notice the use of HTML metacharacters and tags in this doc comment. The Javadoc utility translates doc comments into HTML, and arbitrary HTML elements contained 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 uncommon. The most commonly used tags are `<p>` to separate paragraphs; `<code>` and `<tt>`, which are used for code fragments; and `<pre>`, which is used for longer code fragments.

The `<code>` and `<tt>` tags are largely equivalent. The `<code>` tag is more commonly used and, according to the HTML 4.01 specification, is generally preferable because `<tt>` is a *font style element.* (The use of font style elements is discouraged in favor of style sheets [HTML401].) That said, some programmers prefer `<tt>` because it is shorter and less intrusive.

Don't forget that escape sequences are required to generate HTML metacharacters, such as the less than sign (<), the greater than sign (>), and the ampersand `(&)`. To generate a less than sign, use the escape sequence "`&lt;`". To generate a greater than sign, use the escape sequence "`&gt;`". To generate an ampersand, use the escape sequence "`&amp;`". The use of escape sequences is demonstrated in the `@throws` tag of the above doc comment.

Finally, notice the use of 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.

The first sentence of each doc comment becomes the *summary description* of the element to which the comment pertains. The summary description must stand on its own to describe the functionality of the entity 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.

Be careful not to include a period within the first sentence of a doc comment. If you do, it will prematurely terminate the summary description. For example, a documentation comment that began with “`A college degree, such as B.S., M.S., or Ph.D.`” would result in a summary description of “A college degree, such as B.” The best way avoid this problem is to avoid the use of abbreviations and decimal fractions in summary descriptions. It is, however, possible to include a period in a summary description by replacing the period with its *numeric encoding*, “`&#46;`”. While this works, it doesn’t make for pretty source code:

```
/**
 * A college degree, such as B&#46;S&#46;, M&#46;S&#46; or
 * Ph&#46;D.
 */
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 verb phrase 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.

The doc comment conventions described in this item are sufficient to get by, but there are many others. There are several style guides for writing doc comments [Javadoc-a, Vermeulen00]. Also, there are utilities to check adherence to these rules [Doclint].

Since release 1.2.2, Javadoc has had the ability to “automatically reuse” or “inherit” method comments. If a method 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 Manual*.

This means that classes can now reuse the doc comments from interfaces they implement, rather than copying these comments. This facility has the potential to reduce or eliminate the burden of maintaining multiple sets of nearly identical doc comments, but it does have a limitation. Doc-comment inheritance is all-or-nothing: the inheriting method cannot modify the inherited doc comment in any way. It is not uncommon for a method to specialize the contract inherited from an interface, in which case the method really does need its own doc comment.

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, such as *weblint* [Weblint].

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.

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 adhering to standard conventions. Remember that arbitrary HTML is permissible within documentation comments and that HTML metacharacters must be escaped.