C H A P T E R 9

General Programming

THIS chapter is devoted to the nuts and bolts of the language. It discusses local variables, control structures, libraries, data types, and two extralinguistic facilities: *reflection* and *native methods*. Finally, it discusses optimization and naming conventions.

Item 57: Minimize the scope of local variables

This item is similar in nature to Item 15, "Minimize the accessibility of classes and members." By minimizing the scope of local variables, you increase the readability and maintainability of your code and reduce the likelihood of error.

Older programming languages, such as C, mandated that local variables must be declared at the head of a block, and some programmers continue to do this out of habit. It's a habit worth breaking. As a gentle reminder, Java lets you declare variables anywhere a statement is legal (as does C, since C99).

The most powerful technique for minimizing the scope of a local variable is to declare it where it is first used. If a variable is declared before it is used, it's just clutter—one more thing to distract the reader who is trying to figure out what the program does. By the time the variable is used, the reader might not remember the variable's type or initial value.

Declaring a local variable prematurely can cause its scope not only to begin too early but also to end too late. The scope of a local variable extends from the point where it is declared to the end of the enclosing block. If a variable is declared outside of the block in which it is used, it remains visible after the program exits that block. If a variable is used accidentally before or after its region of intended use, the consequences can be disastrous.

Nearly every local variable declaration should contain an initializer. If you don't yet have enough information to initialize a variable sensibly, you should postpone the declaration until you do. One exception to this rule concerns `try-catch` statements. If a variable is initialized to an expression whose evaluation can throw a checked exception, the variable must be initialized inside a `try` block (unless the enclosing method can propagate the exception). If the value must be used outside of the `try` block, then it must be declared before the `try` block, where it cannot yet be "sensibly initialized." For an example, see page 283.

Loops present a special opportunity to minimize the scope of variables. The `for` loop, in both its traditional and for-each forms, allows you to declare *loop variables*, limiting their scope to the exact region where they're needed. (This region consists of the body of the loop and the code in parentheses between the `for` keyword and the body.) Therefore, **prefer `for` loops to `while` loops**, assuming the contents of the loop variable aren't needed after the loop terminates.

For example, here is the preferred idiom for iterating over a collection (Item 58):

```
// Preferred idiom for iterating over a collection or array
for (Element e : c) {
    ... // Do Something with e
}
```

If you need access to the iterator, perhaps to call its `remove` method, the preferred idiom uses a traditional `for` loop in place of the for-each loop:

```
// Idiom for iterating when you need the iterator
for (Iterator<Element> i = c.iterator(); i.hasNext(); ) {
    Element e = i.next();
    ... // Do something with e and i
}
```

To see why these `for` loops are preferable to a `while` loop, consider the following code fragment, which contains two `while` loops and one bug:

```
Iterator<Element> i = c.iterator();
while (i.hasNext()) {
    doSomething(i.next());
}
...

Iterator<Element> i2 = c2.iterator();
while (i.hasNext()) {             // BUG!
    doSomethingElse(i2.next());
}
```

The second loop contains a copy-and-paste error: it initializes a new loop variable, `i2`, but uses the old one, `i`, which is, unfortunately, still in scope. The resulting code compiles without error and runs without throwing an exception, but it does the wrong thing. Instead of iterating over c2, the second loop terminates immediately, giving the false impression that c2 is empty. Because the program errs silently, the error can remain undetected for a long time.

If a similar copy-and-paste error were made in conjunction with either of the for loops (for-each or traditional), the resulting code wouldn't even compile. The element (or iterator) variable from the first loop would not be in scope in the second loop. Here's how it looks with the traditional for loop:

```
for (Iterator<Element> i = c.iterator(); i.hasNext(); ) {
    Element e = i.next();
    ... // Do something with e and i
}
...

// Compile-time error - cannot find symbol i
for (Iterator<Element> i2 = c2.iterator(); i.hasNext(); ) {
    Element e2 = i2.next();
    ... // Do something with e2 and i2
}
```

Moreover, if you use a for loop, it's much less likely that you'll make the copy-and-paste error because there's no incentive to use different variable names in the two loops. The loops are completely independent, so there's no harm in reusing the element (or iterator) variable name. In fact, it's often stylish to do so.

The for loop has one more advantage over the while loop: it is shorter, which enhances readability.

Here is another loop idiom that minimizes the scope of local variables:

```
for (int i = 0, n = expensiveComputation(); i < n; i++) {
    ... // Do something with i;
}
```

The important thing to notice about this idiom is that it has *two* loop variables, i and n, both of which have exactly the right scope. The second variable, n, is used to store the limit of the first, thus avoiding the cost of a redundant computation in every iteration. As a rule, you should use this idiom if the loop test involves a method invocation that is guaranteed to return the same result on each iteration.

A final technique to minimize the scope of local variables is to **keep methods small and focused.** If you combine two activities in the same method, local variables relevant to one activity may be in the scope of the code performing the other activity. To prevent this from happening, simply separate the method into two: one for each activity.

Item 58: Prefer for-each loops to traditional `for` loops

As discussed in Item 45, some tasks are best accomplished with streams, others with iteration. Here is a traditional `for` loop to iterate over a collection:

```
// Not the best way to iterate over a collection!
for (Iterator<Element> i = c.iterator(); i.hasNext(); ) {
    Element e = i.next();
    ... // Do something with e
}
```

and here is a traditional `for` loop to iterate over an array:

```
// Not the best way to iterate over an array!
for (int i = 0; i < a.length; i++) {
    ... // Do something with a[i]
}
```

These idioms are better than `while` loops (Item 57), but they aren't perfect. The iterator and the index variables are both just clutter—all you need are the elements. Furthermore, they represent opportunities for error. The iterator occurs three times in each loop and the index variable four, which gives you many chances to use the wrong variable. If you do, there is no guarantee that the compiler will catch the problem. Finally, the two loops are quite different, drawing unnecessary attention to the type of the container and adding a (minor) hassle to changing that type.

The for-each loop (officially known as the "enhanced `for` statement") solves all of these problems. It gets rid of the clutter and the opportunity for error by hiding the iterator or index variable. The resulting idiom applies equally to collections and arrays, easing the process of switching the implementation type of a container from one to the other:

```
// The preferred idiom for iterating over collections and arrays
for (Element e : elements) {
    ... // Do something with e
}
```

When you see the colon (`:`), read it as "in." Thus, the loop above reads as "for each element *e* in *elements*." There is no performance penalty for using for-each loops, even for arrays: the code they generate is essentially identical to the code you would write by hand.

The advantages of the for-each loop over the traditional `for` loop are even greater when it comes to nested iteration. Here is a common mistake that people make when doing nested iteration:

```
// Can you spot the bug?
enum Suit { CLUB, DIAMOND, HEART, SPADE }
enum Rank { ACE, DEUCE, THREE, FOUR, FIVE, SIX, SEVEN, EIGHT,
            NINE, TEN, JACK, QUEEN, KING }
...
static Collection<Suit> suits = Arrays.asList(Suit.values());
static Collection<Rank> ranks = Arrays.asList(Rank.values());

List<Card> deck = new ArrayList<>();
for (Iterator<Suit> i = suits.iterator(); i.hasNext(); )
    for (Iterator<Rank> j = ranks.iterator(); j.hasNext(); )
        deck.add(new Card(i.next(), j.next()));
```

Don't feel bad if you didn't spot the bug. Many expert programmers have made this mistake at one time or another. The problem is that the `next` method is called too many times on the iterator for the outer collection (`suits`). It should be called from the outer loop so that it is called once per suit, but instead it is called from the inner loop, so it is called once per card. After you run out of suits, the loop throws a `NoSuchElementException`.

If you're really unlucky and the size of the outer collection is a multiple of the size of the inner collection—perhaps because they're the same collection—the loop will terminate normally, but it won't do what you want. For example, consider this ill-conceived attempt to print all the possible rolls of a pair of dice:

```
// Same bug, different symptom!
enum Face { ONE, TWO, THREE, FOUR, FIVE, SIX }
...
Collection<Face> faces = EnumSet.allOf(Face.class);

for (Iterator<Face> i = faces.iterator(); i.hasNext(); )
    for (Iterator<Face> j = faces.iterator(); j.hasNext(); )
        System.out.println(i.next() + " " + j.next());
```

The program doesn't throw an exception, but it prints only the six "doubles" (from "`ONE ONE`" to "`SIX SIX`"), instead of the expected thirty-six combinations.

To fix the bugs in these examples, you must add a variable in the scope of the outer loop to hold the outer element:

```
// Fixed, but ugly - you can do better!
for (Iterator<Suit> i = suits.iterator(); i.hasNext(); ) {
    Suit suit = i.next();
    for (Iterator<Rank> j = ranks.iterator(); j.hasNext(); )
        deck.add(new Card(suit, j.next()));
}
```

If instead you use a nested for-each loop, the problem simply disappears. The resulting code is as succinct as you could wish for:

```
// Preferred idiom for nested iteration on collections and arrays
for (Suit suit : suits)
    for (Rank rank : ranks)
        deck.add(new Card(suit, rank));
```

Unfortunately, there are three common situations where you *can't* use for-each:

- **Destructive filtering**—If you need to traverse a collection removing selected elements, then you need to use an explicit iterator so that you can call its `remove` method. You can often avoid explicit traversal by using `Collection`'s `removeIf` method, added in Java 8.
- **Transforming**—If you need to traverse a list or array and replace some or all of the values of its elements, then you need the list iterator or array index in order to replace the value of an element.
- **Parallel iteration**—If you need to traverse multiple collections in parallel, then you need explicit control over the iterator or index variable so that all iterators or index variables can be advanced in lockstep (as demonstrated unintentionally in the buggy card and dice examples above).

If you find yourself in any of these situations, use an ordinary `for` loop and be wary of the traps mentioned in this item.

Not only does the for-each loop let you iterate over collections and arrays, it lets you iterate over any object that implements the `Iterable` interface, which consists of a single method. Here is how the interface looks:

```
public interface Iterable<E> {
    // Returns an iterator over the elements in this iterable
    Iterator<E> iterator();
}
```

It is a bit tricky to implement `Iterable` if you have to write your own `Iterator` implementation from scratch, but if you are writing a type that represents a group of elements, you should strongly consider having it implement `Iterable`, even if you choose not to have it implement `Collection`. This will allow your users to iterate over your type using the for-each loop, and they will be forever grateful.

In summary, the for-each loop provides compelling advantages over the traditional `for` loop in clarity, flexibility, and bug prevention, with no performance penalty. Use for-each loops in preference to `for` loops wherever you can.

Item 59: Know and use the libraries

Suppose you want to generate random integers between zero and some upper bound. Faced with this common task, many programmers would write a little method that looks something like this:

```
// Common but deeply flawed!
static Random rnd = new Random();

static int random(int n) {
    return Math.abs(rnd.nextInt()) % n;
}
```

This method may look good, but it has three flaws. The first is that if `n` is a small power of two, the sequence of random numbers will repeat itself after a fairly short period. The second flaw is that if `n` is not a power of two, some numbers will, on average, be returned more frequently than others. If `n` is large, this effect can be quite pronounced. This is powerfully demonstrated by the following program, which generates a million random numbers in a carefully chosen range and then prints out how many of the numbers fell in the lower half of the range:

```
public static void main(String[] args) {
    int n = 2 * (Integer.MAX_VALUE / 3);
    int low = 0;
    for (int i = 0; i < 1000000; i++)
        if (random(n) < n/2)
            low++;
    System.out.println(low);
}
```

If the `random` method worked properly, the program would print a number close to half a million, but if you run it, you'll find that it prints a number close to 666,666. Two-thirds of the numbers generated by the `random` method fall in the lower half of its range!

The third flaw in the `random` method is that it can, on rare occasions, fail catastrophically, returning a number outside the specified range. This is so because the method attempts to map the value returned by `rnd.nextInt()` to a non-negative `int` by calling `Math.abs`. If `nextInt()` returns `Integer.MIN_VALUE`, `Math.abs` will also return `Integer.MIN_VALUE`, and the remainder operator (`%`) will return a negative number, assuming `n` is not a power of two. This will almost certainly cause your program to fail, and the failure may be difficult to reproduce.

To write a version of the `random` method that corrects these flaws, you'd have to know a fair amount about pseudorandom number generators, number theory, and two's complement arithmetic. Luckily, you don't have to do this—it's been done for you. It's called `Random.nextInt(int)`. You needn't concern yourself with the details of how it does its job (although you can study the documentation or the source code if you're curious). A senior engineer with a background in algorithms spent a good deal of time designing, implementing, and testing this method and then showed it to several experts in the field to make sure it was right. Then the library was beta tested, released, and used extensively by millions of programmers for almost two decades. No flaws have yet been found in the method, but if a flaw were to be discovered, it would be fixed in the next release. **By using a standard library, you take advantage of the knowledge of the experts who wrote it and the experience of those who used it before you.**

As of Java 7, you should no longer use `Random`. For most uses, **the random number generator of choice is now `ThreadLocalRandom`.** It produces higher quality random numbers, and it's very fast. On my machine, it is 3.6 times faster than `Random`. For fork join pools and parallel streams, use `SplittableRandom`.

A second advantage of using the libraries is that you don't have to waste your time writing ad hoc solutions to problems that are only marginally related to your work. If you are like most programmers, you'd rather spend your time working on your application than on the underlying plumbing.

A third advantage of using standard libraries is that their performance tends to improve over time, with no effort on your part. Because many people use them and because they're used in industry-standard benchmarks, the organizations that supply these libraries have a strong incentive to make them run faster. Many of the Java platform libraries have been rewritten over the years, sometimes repeatedly, resulting in dramatic performance improvements.

A fourth advantage of using libraries is that they tend to gain functionality over time. If a library is missing something, the developer community will make it known, and the missing functionality may get added in a subsequent release.

A final advantage of using the standard libraries is that you place your code in the mainstream. Such code is more easily readable, maintainable, and reusable by the multitude of developers.

Given all these advantages, it seems only logical to use library facilities in preference to ad hoc implementations, yet many programmers don't. Why not? Perhaps they don't know the library facilities exist. **Numerous features are added to the libraries in every major release, and it pays to keep abreast of these additions.** Each time there is a major release of the Java platform, a web page is published describing its new features. These pages are well worth reading [Java8-feat, Java9-feat]. To reinforce this point, suppose you wanted to write a program to print the contents of a URL specified on the command line (which is roughly what the Linux `curl` command does). Prior to Java 9, this code was a bit tedious, but in Java 9 the `transferTo` method was added to `InputStream`. Here is a complete program to perform this task using this new method:

```
// Printing the contents of a URL with transferTo, added in Java 9
public static void main(String[] args) throws IOException {
    try (InputStream in = new URL(args[0]).openStream()) {
        in.transferTo(System.out);
    }
}
```

The libraries are too big to study all the documentation [Java9-api], but **every programmer should be familiar with the basics of `java.lang`, `java.util`, and `java.io`, and their subpackages.** Knowledge of other libraries can be acquired on an as-needed basis. It is beyond the scope of this item to summarize the facilities in the libraries, which have grown immense over the years.

Several libraries bear special mention. The collections framework and the streams library (Items 45–48) should be part of every programmer's basic toolkit, as should parts of the concurrency utilities in `java.util.concurrent`. This package contains both high-level utilities to simplify the task of multithreaded programming and low-level primitives to allow experts to write their own higher-level concurrent abstractions. The high-level parts of `java.util.concurrent` are discussed in Items 80 and 81.

Occasionally, a library facility can fail to meet your needs. The more specialized your needs, the more likely this is to happen. While your first impulse should be to use the libraries, if you've looked at what they have to offer in some area and it doesn't meet your needs, then use an alternate implementation. There will always be holes in the functionality provided by any finite set of libraries. If you can't find what you need in Java platform libraries, your next choice should be to look in high-quality third-party libraries, such as Google's excellent, open source Guava library [Guava]. If you can't find the functionality that you need in any appropriate library, you may have no choice but to implement it yourself.

To summarize, don't reinvent the wheel. If you need to do something that seems like it should be reasonably common, there may already be a facility in the libraries that does what you want. If there is, use it; if you don't know, check. Generally speaking, library code is likely to be better than code that you'd write yourself and is likely to improve over time. This is no reflection on your abilities as a programmer. Economies of scale dictate that library code receives far more attention than most developers could afford to devote to the same functionality.

Item 60: Avoid `float` and `double` if exact answers are required

The `float` and `double` types are designed primarily for scientific and engineering calculations. They perform *binary floating-point arithmetic*, which was carefully designed to furnish accurate approximations quickly over a broad range of magnitudes. They do not, however, provide exact results and should not be used where exact results are required. **The `float` and `double` types are particularly ill-suited for monetary calculations** because it is impossible to represent 0.1 (or any other negative power of ten) as a `float` or `double` exactly.

For example, suppose you have $1.03 in your pocket, and you spend 42¢. How much money do you have left? Here's a naive program fragment that attempts to answer this question:

```
System.out.println(1.03 - 0.42);
```

Unfortunately, it prints out `0.6100000000000001`. This is not an isolated case. Suppose you have a dollar in your pocket, and you buy nine washers priced at ten cents each. How much change do you get?

```
System.out.println(1.00 - 9 * 0.10);
```

According to this program fragment, you get $`0.09999999999999998`.

You might think that the problem could be solved merely by rounding results prior to printing, but unfortunately this does not always work. For example, suppose you have a dollar in your pocket, and you see a shelf with a row of delicious candies priced at 10¢, 20¢, 30¢, and so forth, up to a dollar. You buy one of each candy, starting with the one that costs 10¢, until you can't afford to buy the next candy on the shelf. How many candies do you buy, and how much change do you get? Here's a naive program designed to solve this problem:

```
// Broken - uses floating point for monetary calculation!
public static void main(String[] args) {
    double funds = 1.00;
    int itemsBought = 0;
    for (double price = 0.10; funds >= price; price += 0.10) {
        funds -= price;
        itemsBought++;
    }
    System.out.println(itemsBought + " items bought.");
    System.out.println("Change: $" + funds);
}
```

If you run the program, you'll find that you can afford three pieces of candy, and you have `$0.3999999999999999` left. This is the wrong answer! The right way to solve this problem is to **use `BigDecimal`, `int`, or `long` for monetary calculations**.

Here's a straightforward transformation of the previous program to use the `BigDecimal` type in place of `double`. Note that `BigDecimal's String` constructor is used rather than its `double` constructor. This is required in order to avoid introducing inaccurate values into the computation [Bloch05, Puzzle 2]:

```
public static void main(String[] args) {
    final BigDecimal TEN_CENTS = new BigDecimal(".10");

    int itemsBought = 0;
    BigDecimal funds = new BigDecimal("1.00");
    for (BigDecimal price = TEN_CENTS;
            funds.compareTo(price) >= 0;
            price = price.add(TEN_CENTS)) {
        funds = funds.subtract(price);
        itemsBought++;
    }
    System.out.println(itemsBought + " items bought.");
    System.out.println("Money left over: $" + funds);
}
```

If you run the revised program, you'll find that you can afford four pieces of candy, with `$0.00` left over. This is the correct answer.

There are, however, two disadvantages to using `BigDecimal`: it's a lot less convenient than using a primitive arithmetic type, and it's a lot slower. The latter disadvantage is irrelevant if you're solving a single short problem, but the former may annoy you.

An alternative to using `BigDecimal` is to use `int` or `long`, depending on the amounts involved, and to keep track of the decimal point yourself. In this example, the obvious approach is to do all computation in cents instead of dollars. Here's a straightforward transformation that takes this approach:

```
public static void main(String[] args) {
    int itemsBought = 0;
    int funds = 100;
    for (int price = 10; funds >= price; price += 10) {
        funds -= price;
        itemsBought++;
    }
    System.out.println(itemsBought + " items bought.");
    System.out.println("Cash left over: " + funds + " cents");
}
```

In summary, don't use `float` or `double` for any calculations that require an exact answer. Use `BigDecimal` if you want the system to keep track of the decimal point and you don't mind the inconvenience and cost of not using a primitive type. Using `BigDecimal` has the added advantage that it gives you full control over rounding, letting you select from eight rounding modes whenever an operation that entails rounding is performed. This comes in handy if you're performing business calculations with legally mandated rounding behavior. If performance is of the essence, you don't mind keeping track of the decimal point yourself, and the quantities aren't too big, use `int` or `long`. If the quantities don't exceed nine decimal digits, you can use `int`; if they don't exceed eighteen digits, you can use `long`. If the quantities might exceed eighteen digits, use `BigDecimal`.

Item 61: Prefer primitive types to boxed primitives

Java has a two-part type system, consisting of *primitives*, such as `int`, `double`, and `boolean`, and *reference types*, such as `String` and `List`. Every primitive type has a corresponding reference type, called a *boxed primitive*. The boxed primitives corresponding to `int`, `double`, and `boolean` are `Integer`, `Double`, and `Boolean`.

As mentioned in Item 6, autoboxing and auto-unboxing blur but do not erase the distinction between the primitive and boxed primitive types. There are real differences between the two, and it's important that you remain aware of which you are using and that you choose carefully between them.

There are three major differences between primitives and boxed primitives. First, primitives have only their values, whereas boxed primitives have identities distinct from their values. In other words, two boxed primitive instances can have the same value and different identities. Second, primitive types have only fully functional values, whereas each boxed primitive type has one nonfunctional value, which is `null`, in addition to all the functional values of the corresponding primitive type. Last, primitives are more time- and space-efficient than boxed primitives. All three of these differences can get you into real trouble if you aren't careful.

Consider the following comparator, which is designed to represent ascending numerical order on `Integer` values. (Recall that a comparator's `compare` method returns a number that is negative, zero, or positive, depending on whether its first argument is less than, equal to, or greater than its second.) You wouldn't need to write this comparator in practice because it implements the natural ordering on `Integer`, but it makes for an interesting example:

```
// Broken comparator - can you spot the flaw?
Comparator<Integer> naturalOrder =
    (i, j) -> (i < j) ? -1 : (i == j ? 0 : 1);
```

This comparator looks like it ought to work, and it will pass many tests. For example, it can be used with `Collections.sort` to correctly sort a million-element list, whether or not the list contains duplicate elements. But the comparator is deeply flawed. To convince yourself of this, merely print the value of `naturalOrder.compare(new Integer(42), new Integer(42))`. Both `Integer` instances represent the same value (42), so the value of this expression should be 0, but it's 1, which indicates that the first `Integer` value is greater than the second!

So what's the problem? The first test in `naturalOrder` works fine. Evaluating the expression `i < j` causes the `Integer` instances referred to by `i` and `j` to be *auto-unboxed*; that is, it extracts their primitive values. The evaluation proceeds to check if the first of the resulting `int` values is less than the second. But suppose it is not. Then the next test evaluates the expression `i == j`, which performs an *identity comparison* on the two object references. If `i` and `j` refer to distinct `Integer` instances that represent the same `int` value, this comparison will return `false`, and the comparator will incorrectly return 1, indicating that the first `Integer` value is greater than the second. **Applying the == operator to boxed primitives is almost always wrong.**

In practice, if you need a comparator to describe a type's natural order, you should simply call `Comparator.naturalOrder()`, and if you write a comparator yourself, you should use the comparator construction methods, or the static compare methods on primitive types (Item 14). That said, you could fix the problem in the broken comparator by adding two local variables to store the primitive `int` values corresponding to the boxed `Integer` parameters, and performing all of the comparisons on these variables. This avoids the erroneous identity comparison:

```
Comparator<Integer> naturalOrder = (iBoxed, jBoxed) -> {
    int i = iBoxed, j = jBoxed; // Auto-unboxing
    return i < j ? -1 : (i == j ? 0 : 1);
};
```

Next, consider this delightful little program:

```
public class Unbelievable {
    static Integer i;

    public static void main(String[] args) {
        if (i == 42)
            System.out.println("Unbelievable");
    }
}
```

No, it doesn't print `Unbelievable`—but what it does is almost as strange. It throws a `NullPointerException` when evaluating the expression `i == 42`. The problem is that `i` is an `Integer`, not an `int`, and like all nonconstant object reference fields, its initial value is `null`. When the program evaluates the expression `i == 42`, it is comparing an `Integer` to an `int`. In nearly every case **when you mix primitives and boxed primitives in an operation, the boxed primitive is auto-unboxed.** If a null object reference is auto-unboxed, you get a `NullPointerException`. As this program demonstrates, it can happen almost anywhere. Fixing the problem is as simple as declaring `i` to be an `int` instead of an `Integer`.

Finally, consider the program from page 24 in Item 6:

```
// Hideously slow program! Can you spot the object creation?
public static void main(String[] args) {
    Long sum = 0L;
    for (long i = 0; i < Integer.MAX_VALUE; i++) {
        sum += i;
    }
    System.out.println(sum);
}
```

This program is much slower than it should be because it accidentally declares a local variable (`sum`) to be of the boxed primitive type `Long` instead of the primitive type `long`. The program compiles without error or warning, and the variable is repeatedly boxed and unboxed, causing the observed performance degradation.

In all three of the programs discussed in this item, the problem was the same: the programmer ignored the distinction between primitives and boxed primitives and suffered the consequences. In the first two programs, the consequences were outright failure; in the third, severe performance problems.

So when should you use boxed primitives? They have several legitimate uses. The first is as elements, keys, and values in collections. You can't put primitives in collections, so you're forced to use boxed primitives. This is a special case of a more general one. You must use boxed primitives as type parameters in parameterized types and methods (Chapter 5), because the language does not permit you to use primitives. For example, you cannot declare a variable to be of type `ThreadLocal<int>`, so you must use `ThreadLocal<Integer>` instead. Finally, you must use boxed primitives when making reflective method invocations (Item 65).

In summary, use primitives in preference to boxed primitives whenever you have the choice. Primitive types are simpler and faster. If you must use boxed primitives, be careful! **Autoboxing reduces the verbosity, but not the danger, of using boxed primitives.** When your program compares two boxed primitives with the == operator, it does an identity comparison, which is almost certainly *not* what you want. When your program does mixed-type computations involving boxed and unboxed primitives, it does unboxing, and **when your program does unboxing, it can throw a `NullPointerException`.** Finally, when your program boxes primitive values, it can result in costly and unnecessary object creations.

Item 62: Avoid strings where other types are more appropriate

Strings are designed to represent text, and they do a fine job of it. Because strings are so common and so well supported by the language, there is a natural tendency to use strings for purposes other than those for which they were designed. This item discusses a few things that you shouldn't do with strings.

Strings are poor substitutes for other value types. When a piece of data comes into a program from a file, from the network, or from keyboard input, it is often in string form. There is a natural tendency to leave it that way, but this tendency is justified only if the data really is textual in nature. If it's numeric, it should be translated into the appropriate numeric type, such as `int`, `float`, or `BigInteger`. If it's the answer to a yes-or-no question, it should be translated into an appropriate enum type or a `boolean`. More generally, if there's an appropriate value type, whether primitive or object reference, you should use it; if there isn't, you should write one. While this advice may seem obvious, it is often violated.

Strings are poor substitutes for enum types. As discussed in Item 34, enums make far better enumerated type constants than strings.

Strings are poor substitutes for aggregate types. If an entity has multiple components, it is usually a bad idea to represent it as a single string. For example, here's a line of code that comes from a real system—identifier names have been changed to protect the guilty:

```
// Inappropriate use of string as aggregate type
String compoundKey = className + "#" + i.next();
```

This approach has many disadvantages. If the character used to separate fields occurs in one of the fields, chaos may result. To access individual fields, you have to parse the string, which is slow, tedious, and error-prone. You can't provide `equals`, `toString`, or `compareTo` methods but are forced to accept the behavior that `String` provides. A better approach is simply to write a class to represent the aggregate, often a private static member class (Item 24).

Strings are poor substitutes for capabilities. Occasionally, strings are used to grant access to some functionality. For example, consider the design of a thread-local variable facility. Such a facility provides variables for which each thread has its own value. The Java libraries have had a thread-local variable facility since release 1.2, but prior to that, programmers had to roll their own. When confronted with the task of designing such a facility many years ago, several people independently came up with the same design, in which client-provided string keys are used to identify each thread-local variable:

```
// Broken - inappropriate use of string as capability!
public class ThreadLocal {
    private ThreadLocal() { } // Noninstantiable

    // Sets the current thread's value for the named variable.
    public static void set(String key, Object value);

    // Returns the current thread's value for the named variable.
    public static Object get(String key);
}
```

The problem with this approach is that the string keys represent a shared global namespace for thread-local variables. In order for the approach to work, the client-provided string keys have to be unique: if two clients independently decide to use the same name for their thread-local variable, they unintentionally share a single variable, which will generally cause both clients to fail. Also, the security is poor. A malicious client could intentionally use the same string key as another client to gain illicit access to the other client's data.

This API can be fixed by replacing the string with an unforgeable key (sometimes called a *capability*):

```
public class ThreadLocal {
    private ThreadLocal() { }  // Noninstantiable

    public static class Key {  // (Capability)
        Key() { }
    }

    // Generates a unique, unforgeable key
    public static Key getKey() {
        return new Key();
    }

    public static void set(Key key, Object value);
    public static Object get(Key key);
}
```

While this solves both of the problems with the string-based API, you can do much better. You don't really need the static methods anymore. They can instead become instance methods on the key, at which point the key is no longer a key for a thread-local variable: it *is* a thread-local variable. At this point, the top-level class isn't doing anything for you anymore, so you might as well get rid of it and rename the nested class to `ThreadLocal`:

```
public final class ThreadLocal {
    public ThreadLocal();
    public void set(Object value);
    public Object get();
}
```

This API isn't typesafe, because you have to cast the value from `Object` to its actual type when you retrieve it from a thread-local variable. It is impossible to make the original `String`-based API typesafe and difficult to make the `Key`-based API typesafe, but it is a simple matter to make this API typesafe by making `ThreadLocal` a parameterized class (Item 29):

```
public final class ThreadLocal<T> {
    public ThreadLocal();
    public void set(T value);
    public T get();
}
```

This is, roughly speaking, the API that `java.lang.ThreadLocal` provides. In addition to solving the problems with the string-based API, it is faster and more elegant than either of the key-based APIs.

To summarize, avoid the natural tendency to represent objects as strings when better data types exist or can be written. Used inappropriately, strings are more cumbersome, less flexible, slower, and more error-prone than other types. Types for which strings are commonly misused include primitive types, enums, and aggregate types.

Item 63: Beware the performance of string concatenation

The string concatenation operator (+) is a convenient way to combine a few strings into one. It is fine for generating a single line of output or constructing the string representation of a small, fixed-size object, but it does not scale. **Using the string concatenation operator repeatedly to concatenate *n* strings requires time quadratic in *n*.** This is an unfortunate consequence of the fact that strings are *immutable* (Item 17). When two strings are concatenated, the contents of both are copied.

For example, consider this method, which constructs the string representation of a billing statement by repeatedly concatenating a line for each item:

```
// Inappropriate use of string concatenation - Performs poorly!
public String statement() {
    String result = "";
    for (int i = 0; i < numItems(); i++)
        result += lineForItem(i);  // String concatenation
    return result;
}
```

The method performs abysmally if the number of items is large. **To achieve acceptable performance, use a `StringBuilder` in place of a `String`** to store the statement under construction:

```
public String statement() {
    StringBuilder b = new StringBuilder(numItems() * LINE_WIDTH);
    for (int i = 0; i < numItems(); i++)
        b.append(lineForItem(i));
    return b.toString();
}
```

A lot of work has gone into making string concatenation faster since Java 6, but the difference in the performance of the two methods is still dramatic: If `numItems` returns 100 and `lineForItem` returns an 80-character string, the second method runs 6.5 times faster than the first on my machine. Because the first method is quadratic in the number of items and the second is linear, the performance difference gets much larger as the number of items grows. Note that the second method preallocates a `StringBuilder` large enough to hold the entire result, eliminating the need for automatic growth. Even if it is detuned to use a default-sized `StringBuilder`, it is still 5.5 times faster than the first method.

The moral is simple: **Don't use the string concatenation operator to combine more than a few strings** unless performance is irrelevant. Use `StringBuilder`'s `append` method instead. Alternatively, use a character array, or process the strings one at a time instead of combining them.

Item 64: Refer to objects by their interfaces

Item 51 says that you should use interfaces rather than classes as parameter types. More generally, you should favor the use of interfaces over classes to refer to objects. **If appropriate interface types exist, then parameters, return values, variables, and fields should all be declared using interface types.** The only time you really need to refer to an object's class is when you're creating it with a constructor. To make this concrete, consider the case of `LinkedHashSet`, which is an implementation of the `Set` interface. Get in the habit of typing this:

```
// Good - uses interface as type
Set<Son> sonSet = new LinkedHashSet<>();
```

not this:

```
// Bad - uses class as type!
LinkedHashSet<Son> sonSet = new LinkedHashSet<>();
```

If you get into the habit of using interfaces as types, your program will be much more flexible. If you decide that you want to switch implementations, all you have to do is change the class name in the constructor (or use a different static factory). For example, the first declaration could be changed to read:

```
Set<Son> sonSet = new HashSet<>();
```

and all of the surrounding code would continue to work. The surrounding code was unaware of the old implementation type, so it would be oblivious to the change.

There is one caveat: if the original implementation offered some special functionality not required by the general contract of the interface and the code depended on that functionality, then it is critical that the new implementation provide the same functionality. For example, if the code surrounding the first declaration depended on `LinkedHashSet`'s ordering policy, then it would be incorrect to substitute `HashSet` for `LinkedHashSet` in the declaration, because `HashSet` makes no guarantee concerning iteration order.

So why would you want to change an implementation type? Because the second implementation offers better performance than the original, or because it offers desirable functionality that the original implementation lacks. For example, suppose a field contains a `HashMap` instance. Changing it to an `EnumMap` will provide better performance and iteration order consistent with the natural order of the keys, but you can only use an `EnumMap` if the key type is an enum type.

Changing the `HashMap` to a `LinkedHashMap` will provide predictable iteration order with performance comparable to that of `HashMap`, without making any special demands on the key type.

You might think it's OK to declare a variable using its implementation type, because you can change the declaration type and the implementation type at the same time, but there is no guarantee that this change will result in a program that compiles. If the client code used methods on the original implementation type that are not also present on its replacement or if the client code passed the instance to a method that requires the original implementation type, then the code will no longer compile after making this change. Declaring the variable with the interface type keeps you honest.

It is entirely appropriate to refer to an object by a class rather than an interface if no appropriate interface exists. For example, consider *value classes,* such as `String` and `BigInteger`. Value classes are rarely written with multiple implementations in mind. They are often final and rarely have corresponding interfaces. It is perfectly appropriate to use such a value class as a parameter, variable, field, or return type.

A second case in which there is no appropriate interface type is that of objects belonging to a framework whose fundamental types are classes rather than interfaces. If an object belongs to such a *class-based framework*, it is preferable to refer to it by the relevant *base class*, which is often abstract, rather than by its implementation class. Many `java.io` classes such as `OutputStream` fall into this category.

A final case in which there is no appropriate interface type is that of classes that implement an interface but also provide extra methods not found in the interface—for example, `PriorityQueue` has a `comparator` method that is not present on the `Queue` interface. Such a class should be used to refer to its instances *only* if the program relies on the extra methods, and this should be very rare.

These three cases are not meant to be exhaustive but merely to convey the flavor of situations where it is appropriate to refer to an object by its class. In practice, it should be apparent whether a given object has an appropriate interface. If it does, your program will be more flexible and stylish if you use the interface to refer to the object. **If there is no appropriate interface, just use the least specific class in the class hierarchy that provides the required functionality.**

Item 65: Prefer interfaces to reflection

The *core reflection facility*, `java.lang.reflect`, offers programmatic access to arbitrary classes. Given a `Class` object, you can obtain `Constructor`, `Method`, and `Field` instances representing the constructors, methods, and fields of the class represented by the `Class` instance. These objects provide programmatic access to the class's member names, field types, method signatures, and so on.

Moreover, `Constructor`, `Method`, and `Field` instances let you manipulate their underlying counterparts *reflectively*: you can construct instances, invoke methods, and access fields of the underlying class by invoking methods on the `Constructor`, `Method`, and `Field` instances. For example, `Method.invoke` lets you invoke any method on any object of any class (subject to the usual security constraints). Reflection allows one class to use another, even if the latter class did not exist when the former was compiled. This power, however, comes at a price:

- **You lose all the benefits of compile-time type checking,** including exception checking. If a program attempts to invoke a nonexistent or inaccessible method reflectively, it will fail at runtime unless you've taken special precautions.
- **The code required to perform reflective access is clumsy and verbose.** It is tedious to write and difficult to read.
- **Performance suffers.** Reflective method invocation is much slower than normal method invocation. Exactly how much slower is hard to say, as there are many factors at work. On my machine, invoking a method with no input parameters and an `int` return was eleven times slower when done reflectively.

There are a few sophisticated applications that require reflection. Examples include code analysis tools and dependency injection frameworks. Even such tools have been moving away from reflection of late, as its disadvantages become clearer. If you have any doubts as to whether your application requires reflection, it probably doesn't.

You can obtain many of the benefits of reflection while incurring few of its costs by using it only in a very limited form. For many programs that must use a class that is unavailable at compile time, there exists at compile time an appropriate interface or superclass by which to refer to the class (Item 64). If this is the case, you can **create instances reflectively and access them normally via their interface or superclass.**

For example, here is a program that creates a `Set<String>` instance whose class is specified by the first command line argument. The program inserts the remaining command line arguments into the set and prints it. Regardless of the first argument, the program prints the remaining arguments with duplicates eliminated. The order in which these arguments are printed, however, depends on the class specified in the first argument. If you specify `java.util.HashSet`, they're printed in apparently random order; if you specify `java.util.TreeSet`, they're printed in alphabetical order because the elements in a `TreeSet` are sorted:

```
// Reflective instantiation with interface access
public static void main(String[] args) {
    // Translate the class name into a Class object
    Class<? extends Set<String>> cl = null;
    try {
        cl = (Class<? extends Set<String>>)  // Unchecked cast!
                Class.forName(args[0]);
    } catch (ClassNotFoundException e) {
        fatalError("Class not found.");
    }

    // Get the constructor
    Constructor<? extends Set<String>> cons = null;
    try {
        cons = cl.getDeclaredConstructor();
    } catch (NoSuchMethodException e) {
        fatalError("No parameterless constructor");
    }

    // Instantiate the set
    Set<String> s = null;
    try {
        s = cons.newInstance();
    } catch (IllegalAccessException e) {
        fatalError("Constructor not accessible");
    } catch (InstantiationException e) {
        fatalError("Class not instantiable.");
    } catch (InvocationTargetException e) {
        fatalError("Constructor threw " + e.getCause());
    } catch (ClassCastException e) {
        fatalError("Class doesn't implement Set");
    }

    // Exercise the set
    s.addAll(Arrays.asList(args).subList(1, args.length));
    System.out.println(s);
}

private static void fatalError(String msg) {
    System.err.println(msg);
    System.exit(1);
}
```

While this program is just a toy, the technique it demonstrates is quite powerful. The toy program could easily be turned into a generic set tester that validates the specified `Set` implementation by aggressively manipulating one or more instances and checking that they obey the `Set` contract. Similarly, it could be turned into a generic set performance analysis tool. In fact, this technique is sufficiently powerful to implement a full-blown *service provider framework* (Item 1). Usually, this technique is all that you need in the way of reflection.

This example demonstrates two disadvantages of reflection. First, the example can generate six different exceptions at runtime, all of which would have been compile-time errors if reflective instantiation were not used. (For fun, you can cause the program to generate each of the six exceptions by passing in appropriate command line arguments.) The second disadvantage is that it takes twenty-five lines of tedious code to generate an instance of the class from its name, whereas a constructor invocation would fit neatly on a single line. The length of the program could be reduced by catching `ReflectiveOperationException`, a superclass of the various reflective exceptions that was introduced in Java 7. Both disadvantages are restricted to the part of the program that instantiates the object. Once instantiated, the set is indistinguishable from any other `Set` instance. In a real program, the great bulk of the code is thus unaffected by this limited use of reflection.

If you compile this program, you'll get an unchecked cast warning. This warning is legitimate, in that the cast to `Class<? extends Set<String>>` will succeed even if the named class is not a `Set` implementation, in which case the program with throw a `ClassCastException` when it instantiates the class. To learn about suppressing the warning, read Item 27.

A legitimate, if rare, use of reflection is to manage a class's dependencies on other classes, methods, or fields that may be absent at runtime. This can be useful if you are writing a package that must run against multiple versions of some other package. The technique is to compile your package against the minimal environment required to support it, typically the oldest version, and to access any newer classes or methods reflectively. To make this work, you have to take appropriate action if a newer class or method that you are attempting to access does not exist at runtime. Appropriate action might consist of using some alternate means to accomplish the same goal or operating with reduced functionality.

In summary, reflection is a powerful facility that is required for certain sophisticated system programming tasks, but it has many disadvantages. If you are writing a program that has to work with classes unknown at compile time, you should, if at all possible, use reflection only to instantiate objects, and access the objects using some interface or superclass that is known at compile time.

Item 66: Use native methods judiciously

The Java Native Interface (JNI) allows Java programs to call *native methods*, which are methods written in *native programming languages* such as C or C++. Historically, native methods have had three main uses. They provide access to platform-specific facilities such as registries. They provide access to existing libraries of native code, including legacy libraries that provide access to legacy data. Finally, native methods are used to write performance-critical parts of applications in native languages for improved performance.

It is legitimate to use native methods to access platform-specific facilities, but it is seldom necessary: as the Java platform matured, it provided access to many features previously found only in host platforms. For example, the process API, added in Java 9, provides access to OS processes. It is also legitimate to use native methods to use native libraries when no equivalent libraries are available in Java.

It is rarely advisable to use native methods for improved performance. In early releases (prior to Java 3), it was often necessary, but JVMs have gotten *much* faster since then. For most tasks, it is now possible to obtain comparable performance in Java. For example, when `java.math` was added in release 1.1, `BigInteger` relied on a then-fast multiprecision arithmetic library written in C. In Java 3, `BigInteger` was reimplemented in Java, and carefully tuned to the point where it ran faster than the original native implementation.

A sad coda to this story is that `BigInteger` has changed little since then, with the exception of faster multiplication for large numbers in Java 8. In that time, work continued apace on native libraries, notably GNU Multiple Precision arithmetic library (GMP). Java programmers in need of truly high-performance multiprecision arithmetic are now justified in using GMP via native methods [Blum14].

The use of native methods has *serious* disadvantages. Because native languages are not *safe* (Item 50), applications using native methods are no longer immune to memory corruption errors. Because native languages are more platform-dependent than Java, programs using native methods are less portable. They are also harder to debug. If you aren't careful, native methods can *decrease* performance because the garbage collector can't automate, or even track, native memory usage (Item 8), and there is a cost associated with going into and out of native code. Finally, native methods require "glue code" that is difficult to read and tedious to write.

In summary, think twice before using native methods. It is rare that you need to use them for improved performance. If you must use native methods to access low-level resources or native libraries, use as little native code as possible and test it thoroughly. A single bug in the native code can corrupt your entire application.

Item 67: Optimize judiciously

There are three aphorisms concerning optimization that everyone should know:

> More computing sins are committed in the name of efficiency (without necessarily achieving it) than for any other single reason—including blind stupidity.
>
> —William A. Wulf [Wulf72]

> We *should* forget about small efficiencies, say about 97% of the time: premature optimization is the root of all evil.
>
> —Donald E. Knuth [Knuth74]

> We follow two rules in the matter of optimization:
>
> Rule 1. Don't do it.
>
> Rule 2 (for experts only). Don't do it yet—that is, not until you have a perfectly clear and unoptimized solution.
>
> —M. A. Jackson [Jackson75]

All of these aphorisms predate the Java programming language by two decades. They tell a deep truth about optimization: it is easy to do more harm than good, especially if you optimize prematurely. In the process, you may produce software that is neither fast nor correct and cannot easily be fixed.

Don't sacrifice sound architectural principles for performance. **Strive to write good programs rather than fast ones.** If a good program is not fast enough, its architecture will allow it to be optimized. Good programs embody the principle of *information hiding*: where possible, they localize design decisions within individual components, so individual decisions can be changed without affecting the remainder of the system (Item 15).

This does *not* mean that you can ignore performance concerns until your program is complete. Implementation problems can be fixed by later optimization, but pervasive architectural flaws that limit performance can be impossible to fix without rewriting the system. Changing a fundamental facet of your design after the fact can result in an ill-structured system that is difficult to maintain and evolve. Therefore you must think about performance during the design process.

Strive to avoid design decisions that limit performance. The components of a design that are most difficult to change after the fact are those specifying interactions between components and with the outside world. Chief among these design components are APIs, wire-level protocols, and persistent data formats. Not only are these design components difficult or impossible to change after the fact, but all of them can place significant limitations on the performance that a system can ever achieve.

Consider the performance consequences of your API design decisions. Making a public type mutable may require a lot of needless defensive copying (Item 50). Similarly, using inheritance in a public class where composition would have been appropriate ties the class forever to its superclass, which can place artificial limits on the performance of the subclass (Item 18). As a final example, using an implementation type rather than an interface in an API ties you to a specific implementation, even though faster implementations may be written in the future (Item 64).

The effects of API design on performance are very real. Consider the `getSize` method in the `java.awt.Component` class. The decision that this performance-critical method was to return a `Dimension` instance, coupled with the decision that `Dimension` instances are mutable, forces any implementation of this method to allocate a new `Dimension` instance on every invocation. Even though allocating small objects is inexpensive on a modern VM, allocating millions of objects needlessly can do real harm to performance.

Several API design alternatives existed. Ideally, `Dimension` should have been immutable (Item 17); alternatively, `getSize` could have been replaced by two methods returning the individual primitive components of a `Dimension` object. In fact, two such methods were added to `Component` in Java 2 for performance reasons. Preexisting client code, however, still uses the `getSize` method and still suffers the performance consequences of the original API design decisions.

Luckily, it is generally the case that good API design is consistent with good performance. **It is a very bad idea to warp an API to achieve good performance.** The performance issue that caused you to warp the API may go away in a future release of the platform or other underlying software, but the warped API and the support headaches that come with it will be with you forever.

Once you've carefully designed your program and produced a clear, concise, and well-structured implementation, *then* it may be time to consider optimization, assuming you're not already satisfied with the performance of the program.

Recall that Jackson's two rules of optimization were "Don't do it," and "(for experts only). Don't do it yet." He could have added one more: **measure performance before and after each attempted optimization.** You may be surprised by what you find. Often, attempted optimizations have no measurable effect on performance; sometimes, they make it worse. The main reason is that it's difficult to guess where your program is spending its time. The part of the program that you think is slow may not be at fault, in which case you'd be wasting your time trying to optimize it. Common wisdom says that programs spend 90 percent of their time in 10 percent of their code.

Profiling tools can help you decide where to focus your optimization efforts. These tools give you runtime information, such as roughly how much time each method is consuming and how many times it is invoked. In addition to focusing your tuning efforts, this can alert you to the need for algorithmic changes. If a quadratic (or worse) algorithm lurks inside your program, no amount of tuning will fix the problem. You must replace the algorithm with one that is more efficient. The more code in the system, the more important it is to use a profiler. It's like looking for a needle in a haystack: the bigger the haystack, the more useful it is to have a metal detector. Another tool that deserves special mention is jmh, which is not a profiler but a *microbenchmarking framework* that provides unparalleled visibility into the detailed performance of Java code [JMH].

The need to measure the effects of attempted optimization is even greater in Java than in more traditional languages such as C and C++, because Java has a weaker *performance model*: The relative cost of the various primitive operations is less well defined. The "abstraction gap" between what the programmer writes and what the CPU executes is greater, which makes it even more difficult to reliably predict the performance consequences of optimizations. There are plenty of performance myths floating around that turn out to be half-truths or outright lies.

Not only is Java's performance model ill-defined, but it varies from implementation to implementation, from release to release, and from processor to processor. If you will be running your program on multiple implementations or multiple hardware platforms, it is important that you measure the effects of your optimization on each. Occasionally you may be forced to make trade-offs between performance on different implementations or hardware platforms.

In the nearly two decades since this item was first written, every component of the Java software stack has grown in complexity, from processors to VMs to libraries, and the variety of hardware on which Java runs has grown immensely. All of this has combined to make the performance of Java programs even less predictable now than it was in 2001, with a corresponding increase in the need to measure it.

To summarize, do not strive to write fast programs—strive to write good ones; speed will follow. But do think about performance while you're designing systems, especially while you're designing APIs, wire-level protocols, and persistent data formats. When you've finished building the system, measure its performance. If it's fast enough, you're done. If not, locate the source of the problem with the aid of a profiler and go to work optimizing the relevant parts of the system. The first step is to examine your choice of algorithms: no amount of low-level optimization can make up for a poor choice of algorithm. Repeat this process as necessary, measuring the performance after every change, until you're satisfied.

Item 68: Adhere to generally accepted naming conventions

The Java platform has a well-established set of *naming conventions*, many of which are contained in *The Java Language Specification* [JLS, 6.1]. Loosely speaking, naming conventions fall into two categories: typographical and grammatical.

There are only a handful of typographical naming conventions, covering packages, classes, interfaces, methods, fields, and type variables. You should rarely violate them and never without a very good reason. If an API violates these conventions, it may be difficult to use. If an implementation violates them, it may be difficult to maintain. In both cases, violations have the potential to confuse and irritate other programmers who work with the code and can cause faulty assumptions that lead to errors. The conventions are summarized in this item.

Package and module names should be hierarchical with the components separated by periods. Components should consist of lowercase alphabetic characters and, rarely, digits. The name of any package that will be used outside your organization should begin with your organization's Internet domain name with the components reversed, for example, `edu.cmu`, `com.google`, `org.eff`. The standard libraries and optional packages, whose names begin with `java` and `javax`, are exceptions to this rule. Users must not create packages or modules whose names begin with `java` or `javax`. Detailed rules for converting Internet domain names to package name prefixes can be found in the JLS [JLS, 6.1].

The remainder of a package name should consist of one or more components describing the package. Components should be short, generally eight or fewer characters. Meaningful abbreviations are encouraged, for example, `util` rather than `utilities`. Acronyms are acceptable, for example, `awt`. Components should generally consist of a single word or abbreviation.

Many packages have names with just one component in addition to the Internet domain name. Additional components are appropriate for large facilities whose size demands that they be broken up into an informal hierarchy. For example, the `javax.util` package has a rich hierarchy of packages with names such as `java.util.concurrent.atomic`. Such packages are known as *subpackages*, although there is almost no linguistic support for package hierarchies.

Class and interface names, including enum and annotation type names, should consist of one or more words, with the first letter of each word capitalized, for example, `List` or `FutureTask`. Abbreviations are to be avoided, except for acronyms and certain common abbreviations like `max` and `min`. There is some disagreement as to whether acronyms should be uppercase or have only their first letter capitalized. While some programmers still use uppercase, a strong argument can be made in favor of capitalizing only the first letter: even if multiple acronyms occur back-to-back, you can still tell where one word starts and the next word ends. Which class name would you rather see, `HTTPURL` or `HttpUrl`?

Method and field names follow the same typographical conventions as class and interface names, except that the first letter of a method or field name should be lowercase, for example, `remove` or `ensureCapacity`. If an acronym occurs as the first word of a method or field name, it should be lowercase.

The sole exception to the previous rule concerns "constant fields," whose names should consist of one or more uppercase words separated by the underscore character, for example, `VALUES` or `NEGATIVE_INFINITY`. A constant field is a static final field whose value is immutable. If a static final field has a primitive type or an immutable reference type (Item 17), then it is a constant field. For example, enum constants are constant fields. If a static final field has a mutable reference type, it can still be a constant field if the referenced object is immutable. Note that constant fields constitute the *only* recommended use of underscores.

Local variable names have similar typographical naming conventions to member names, except that abbreviations are permitted, as are individual characters and short sequences of characters whose meaning depends on the context in which they occur, for example, `i`, `denom`, `houseNum`. Input parameters are a special kind of local variable. They should be named much more carefully than ordinary local variables, as their names are an integral part of their method's documentation.

Type parameter names usually consist of a single letter. Most commonly it is one of these five: `T` for an arbitrary type, `E` for the element type of a collection, `K` and `V` for the key and value types of a map, and `X` for an exception. The return type of a function is usually `R`. A sequence of arbitrary types can be `T`, `U`, `V` or `T1`, `T2`, `T3`.

For quick reference, the following table shows examples of typographical conventions.

Identifier Type	**Examples**
Package or module	`org.junit.jupiter.api`, `com.google.common.collect`
Class or Interface	`Stream`, `FutureTask`, `LinkedHashMap`, `HttpClient`
Method or Field	`remove`, `groupingBy`, `getCrc`
Constant Field	`MIN_VALUE`, `NEGATIVE_INFINITY`
Local Variable	`i`, `denom`, `houseNum`
Type Parameter	`T`, `E`, `K`, `V`, `X`, `R`, `U`, `V`, `T1`, `T2`

Grammatical naming conventions are more flexible and more controversial than typographical conventions. There are no grammatical naming conventions to speak of for packages. Instantiable classes, including enum types, are generally named with a singular noun or noun phrase, such as `Thread`, `PriorityQueue`, or `ChessPiece`. Non-instantiable utility classes (Item 4) are often named with a plural noun, such as `Collectors` or `Collections`. Interfaces are named like classes, for example, `Collection` or `Comparator`, or with an adjective ending in `able` or `ible`, for example, `Runnable`, `Iterable`, or `Accessible`. Because annotation types have so many uses, no part of speech predominates. Nouns, verbs, prepositions, and adjectives are all common, for example, `BindingAnnotation`, `Inject`, `ImplementedBy`, or `Singleton`.

Methods that perform some action are generally named with a verb or verb phrase (including object), for example, `append` or `drawImage`. Methods that return a `boolean` value usually have names that begin with the word `is` or, less commonly, `has`, followed by a noun, noun phrase, or any word or phrase that functions as an adjective, for example, `isDigit`, `isProbablePrime`, `isEmpty`, `isEnabled`, or `hasSiblings`.

Methods that return a non-`boolean` function or attribute of the object on which they're invoked are usually named with a noun, a noun phrase, or a verb phrase beginning with the verb `get`, for example, `size`, `hashCode`, or `getTime`. There is a vocal contingent that claims that only the third form (beginning with `get`) is acceptable, but there is little basis for this claim. The first two forms usually lead to more readable code, for example:

```
if (car.speed() > 2 * SPEED_LIMIT)
    generateAudibleAlert("Watch out for cops!");
```

The form beginning with `get` has its roots in the largely obsolete *Java Beans* specification, which formed the basis of an early reusable component architecture. There are modern tools that continue to rely on the Beans naming convention, and you should feel free to use it in any code that is to be used in conjunction with these tools. There is also a strong precedent for following this naming convention if a class contains both a setter and a getter for the same attribute. In this case, the two methods are typically named `get`*Attribute* and `set`*Attribute*.

A few method names deserve special mention. Instance methods that convert the type of an object, returning an independent object of a different type, are often called `to`*Type*, for example, `toString` or `toArray`. Methods that return a *view* (Item 6) whose type differs from that of the receiving object are often called `as`*Type*, for example, `asList`. Methods that return a primitive with the same value as the object on which they're invoked are often called *type*`Value`, for example, `intValue`. Common names for static factories include `from`, `of`, `valueOf`, `instance`, `getInstance`, `newInstance`, `get`*Type*, and `new`*Type* (Item 1, page 9).

Grammatical conventions for field names are less well established and less important than those for class, interface, and method names because well-designed APIs contain few if any exposed fields. Fields of type `boolean` are often named like `boolean` accessor methods with the initial `is` omitted, for example, `initialized`, `composite`. Fields of other types are usually named with nouns or noun phrases, such as `height`, `digits`, or `bodyStyle`. Grammatical conventions for local variables are similar to those for fields but even weaker.

To summarize, internalize the standard naming conventions and learn to use them as second nature. The typographical conventions are straightforward and largely unambiguous; the grammatical conventions are more complex and looser. To quote from *The Java Language Specification* [JLS, 6.1], "These conventions should not be followed slavishly if long-held conventional usage dictates otherwise." Use common sense.