CHAPTER 7

General Programming

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

Item 29: Minimize the scope of local variables

This item is similar in nature to Item 12, "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.

The C programming language mandates that local variables must be declared at the head of a block, and programmers continue to do this out of habit; it's a habit worth breaking. As a reminder, the Java programming language lets you declare variables anywhere a statement is legal.

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 is 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. If the program evolves and the variable is no longer used, it is easy to forget to remove the declaration if it's far removed from the point of first use.

Not only can declaring a local variable prematurely cause its scope to extend too early, but also too late. The scope of a local variable extends from the point of its declaration 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 by a method that throws a checked exception, it must be initialized inside a `try` block. 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 example, see page 159.

Loops present a special opportunity to minimize the scope of variables. The `for` loop 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 as well as the initialization, test, and update preceding the body.) Therefore **prefer `for` loops to `while` loops**, assuming the contents of the loop variable(s) aren't needed after the loop terminates.

For example, here is the preferred idiom for iterating over a collection:

```
for (Iterator i = c.iterator(); i.hasNext(); ) {
    doSomething(i.next());
}
```

To see why this `for` loop is preferable to the more obvious `while` loop, consider the following code fragment, which contains two `while` loops and one bug:

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

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

The second loop contains a cut-and-paste error: It initializes a new loop variable, `i2`, but uses the old one, `i`, which unfortunately is 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 the analogous cut-and-paste error were made in conjunction with the preferred `for` loop idiom, the resulting code wouldn't even compile. The loop vari-

able from the first loop would not be in scope at the point where the second loop occurred:

```
for (Iterator i = c.iterator(); i.hasNext(); ) {
    doSomething(i.next());
}
    ...

// Compile-time error - the symbol i cannot be resolved
for (Iterator i2 = c2.iterator(); i.hasNext(); ) {
    doSomething(i2.next());
}
```

Moreover, if you use the `for` loop idiom, it's much less likely that you'll make the cut-and-paste error, as there's no incentive to use a different variable name in the two loops. The loops are completely independent, so there's no harm in reusing the loop variable name. In fact, it's stylish to do so.

The `for` loop idiom has one other advantage over the `while` loop idiom, albeit a minor one. The `for` loop idiom is one line shorter, which helps the containing method fit in a fixed-size editor window, enhancing readability.

Here is another loop idiom for iterating over a list that minimizes the scope of local variables:

```
// High-performance idiom for iterating over random access lists
for (int i = 0, n = list.size(); i < n; i++) {
    doSomething(list.get(i));
}
```

This idiom is useful for random access `List` implementations such as `ArrayList` and `Vector` because it is likely to run faster than the "preferred idiom" above for such lists. 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 use of the second variable is essential to the performance of the idiom. Without it, the loop would have to call the `size` method once per iteration, which would negate the performance advantage of the idiom. Using this idiom is acceptable when you're sure the list really does provide random access; otherwise, it displays quadratic performance.

Similar idioms exist for other looping tasks, for example,

```
for (int i = 0, n = expensiveComputation(); i < n; i++) {
    doSomething(i);
}
```

Again, this idiom uses two loop variables, and the second variable, `n`, is used to avoid the cost of performing redundant computation on every iteration. As a rule, you should use this idiom if the loop test involves a method invocation and the method invocation 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 30: Know and use the libraries

Suppose you want to generate random integers between 0 and some upper bound. Faced with this common task, many programmers would write a little method that looks something like this:

```
static Random rnd = new Random();

// Common but flawed!
static int random(int n) {
    return Math.abs(rnd.nextInt()) % n;
}
```

This method isn't bad, but it isn't perfect, either—it has three flaws. The first flaw is that if `n` is a small power of two, the sequence of random numbers it generates 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 flaw can be quite pronounced. This is graphically 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 occasion, fail catastrophically, returning a number outside the specified range. This is so because the method attempts to map the value returned by `rnd.nextInt()` into a nonnegative integer with `Math.abs`. If `nextInt()` returns `Integer.MIN_VALUE`, `Math.abs` will also return `Integer.MIN_VALUE`, and the remainder operator (`%`) will return a neg-

ative 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 `random` that corrects these three flaws, you'd have to know a fair amount about linear congruential pseudorandom number generators, number theory, and two's complement arithmetic. Luckily, you don't have to do this—it's already been done for you. It's called `Random.nextInt(int)`, and it was added to the standard library package `java.util` in release 1.2.

You don't have to concern yourself with the details of how `nextInt(int)` does its job (although you can study the documentation or the source code if you're morbidly 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 experts in the field to make sure it was right. Then the library was beta tested, released, and used extensively by thousands of programmers for several years. No flaws have yet been found in the method, but if a flaw were to be discovered, it would get 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.**

A second advantage of using the libraries is that you don't have to waste your time writing ad hoc solutions to problems 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. For example, the standard multiprecision arithmetic library, `java.math`, was rewritten in release 1.3, resulting in dramatic performance improvements.

Libraries also tend to gain new functionality over time. If a library class is missing some important functionality, the developer community will make this shortcoming known. The Java platform has always been developed with substantial input from this community. Previously the process was informal; now there is a formal process in place called the Java Community Process (JCP). Either way, missing features tend to get added over time.

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 a significant fraction of programmers

don't. Why? Perhaps they don't know that the library facilities exist. **Numerous features are added to the libraries in every major release, and it pays to keep abreast of these additions.** You can peruse the documentation online or read about the libraries in any number of books [J2SE-APIs, Chan00, Flanagan99, Chan98]. The libraries are too big to study all the documentation, but **every programmer should be familiar with the contents of `java.lang`, `java.util`, and, to a lesser extent, `java.io`**. Knowledge of other libraries can be acquired on an as-needed basis.

It is beyond the scope of this item to summarize all the facilities in the libraries, but a few bear special mention. In the 1.2 release, a *Collections Framework* was added to the `java.util` package. It should be part of every programmer's basic toolkit. The Collections Framework is a unified architecture for representing and manipulating collections, allowing them to be manipulated independently of the details of their representation. It reduces programming effort while increasing performance. It allows for interoperability among unrelated APIs, reduces effort in designing and learning new APIs, and fosters software reuse.

The framework is based on six collection interfaces (`Collection`, `Set`, `List`, `Map`, `SortedList`, and `SortedMap`). It includes implementations of these interfaces and algorithms to manipulate them. The legacy collection classes, `Vector` and `Hashtable`, were retrofitted to participate in the framework, so you don't have to abandon them to take advantage of the framework.

The Collections Framework substantially reduces the amount of code necessary to do many mundane tasks. For example, suppose you have a vector of strings, and you want to sort it alphabetically. This one-liner does the job:

```
Collections.sort(v);
```

If you want to do the same thing ignoring case distinctions, use the following:

```
Collections.sort(v, String.CASE_INSENSITIVE_ORDER);
```

Suppose you want to print out all of the elements in an array. Many programmers use a `for` loop, but there's no need if you use the following idiom:

```
System.out.println(Arrays.asList(a));
```

Finally, suppose you want to know all of the keys for which two `Hashtable` instances, `h1` and `h2`, contain the same mappings. Before the Collections

Framework was added, this would have required a fair amount of code, but now it takes three lines:

```
Map tmp = new HashMap(h1);
tmp.entrySet().retainAll(h2.entrySet());
Set result = tmp.keySet();
```

The foregoing examples barely scratch the surface of what you can do with the Collections Framework. If you want to know more, see the documentation on Sun's Web site [Collections] or read the tutorial [Bloch99].

A third-party library worthy of note is Doug Lea's `util.concurrent` [Lea01], which provides high-level concurrency utilities to simplify the task of multithreaded programming.

There are many additions to the libraries in the 1.4 release. Notable additions include the following:

- **`java.util.regex`**—A full-blown Perl-like regular expression facility.
- **`java.util.prefs`**—A facility for the persistent storage of user preferences and program configuration data.
- **`java.nio`**—A high-performance I/O facility, including *scalable I/O* (akin to the Unix `poll` call) and *memory-mapped I/O* (akin to the Unix `mmap` call).
- **`java.util.LinkedHashSet`, `LinkedHashMap`, `IdentityHashMap`**—New collection implementations.

Occasionally, a library facility may 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, use an alternate implementation. There will always be holes in the functionality provided by any finite set of libraries. If the functionality that you need is missing, 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 reasonably common, there may already be a class 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 the average developer could afford to devote to the same functionality.

Item 31: 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 - .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*.10);
```

According to this program fragment, you get $0.09999999999999995. 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 = .10; funds >= price; price += .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`:

```
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)) {
        itemsBought++;
        funds = funds.subtract(price);
    }
    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 less convenient than using a primitive arithmetic type, and its 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 pennies instead of dollars. Here's a straightforward transformation of the program just shown that takes this approach:

```
public static void main(String[] args) {
    int itemsBought = 0;
    int funds = 100;
    for (int price = 10; funds >= price; price += 10) {
        itemsBought++;
        funds -= price;
    }
    System.out.println(itemsBought + " items bought.");
    System.out.println("Money 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 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, if you don't mind keeping track of the decimal point yourself, and if 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 exceed eighteen digits, you must use `BigDecimal`.

Item 32: 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 it 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 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 enumerated types. As discussed in Item 21, both typesafe enums and `int` values 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 18).

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. When confronted with designing such a facility several

years ago, several people independently came up with the same design in which client-provided string keys grant access to the contents of a 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 keys represent a shared global namespace. If two independent clients of the package decide to use the same name for their thread-local variable, they unintentionally share the variable, which will generally cause both clients to fail. Also, the security is poor; a malicious client could intentionally use the same 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 {
        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 better. You don't really need the static methods any more. They can instead become instance methods on the key, at which point the key is no longer a key: it is a thread-local variable. At this point, the noninstantiable top-level class isn't

doing anything for you any more, so you might as well get rid of it and rename the nested class to `ThreadLocal`:

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

This is, roughly speaking, the API that `java.util.ThreadLocal` provides. In addition to solving the problems with the string-based API, it's 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, enumerated types, and aggregate types.

Item 33: 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 for 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*.** It is an unfortunate consequence of the fact that strings are *immutable* (Item 13). When two strings are concatenated, the contents of both are copied.

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

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

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

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

The difference in performance is dramatic. If `numItems` returns 100 and `lineForItem` returns a constant 80-character string, the second method is ninety times faster on my machine than the first. Because the first method is quadratic in the number of items and the second is linear, the performance difference is even more dramatic for larger numbers of items. Note that the second method preallocates a `StringBuffer` large enough to hold the result. Even if it is detuned to use a default-sized `StringBuffer`, it is still forty-five times faster than the first.

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

Item 34: Refer to objects by their interfaces

Item 25 contains the advice that you should use interfaces rather than classes as parameter types. More generally, you should favor the use of interfaces rather than classes to refer to objects. **If appropriate interface types exist, 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. To make this concrete, consider the case of `Vector`, which is an implementation of the `List` interface. Get in the habit of typing this:

```
// Good - uses interface as type
List subscribers = new Vector();
```

rather than this:

```
// Bad - uses class as type!
Vector subscribers = new Vector();
```

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

```
List subscribers = new ArrayList();
```

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 the fact that `Vector` is synchronized, then it would be incorrect to substitute `ArrayList` for `Vector` in the declaration.

So why would you want to change implementations? Because the new implementation offers better performance or because it offers desirable extra functionality. A real-world example concerns the `ThreadLocal` class. Internally, this class uses a package-private `Map` field in `Thread` to associate per-thread values with `ThreadLocal` instances. In the 1.3 release, this field was initialized to a `HashMap` instance. In the 1.4 release, a new, special-purpose `Map` implementation, called

`IdentityHashMap`, was added to the platform. By changing a single line of code to initialize the field to an `IdentityHashMap` instead of a `HashMap`, the `ThreadLocal` facility was made faster.

Had the field been declared as a `HashMap` instead of a `Map`, there is no guarantee that a single-line change would have been sufficient. If the client code had used `HashMap` operations outside of the `Map` interface or passed the map to a method that demanded a `HashMap`, the code would no longer compile if the field were changed to an `IdentityHashMap`. Declaring the field 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 a value class as a parameter, variable, field, or return type. More generally, if a concrete class has no associated interface, then you have no choice but to refer to it by its class whether or not it represents a value. The `Random` class falls into this category.

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 typically abstract, rather than by its implementation class. The `java.util.TimerTask` class falls into this category.

A final case in which there is no appropriate interface type is that of classes that implement an interface but provide extra methods not found in the interface—for example, `LinkedList`. Such a class should be used only to refer to its instances if the program relies on the extra methods: it should never be used as a parameter type (Item 25).

These 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 if you use the interface to refer to the object; if not, just use the highest class in the class hierarchy that provides the required functionality.

Item 35: Prefer interfaces to reflection

The reflection facility, `java.lang.reflect`, offers programmatic access to information about loaded classes. Given a `Class` instance, 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`, `Field`, and `Method` 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 run time 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**. As of release 1.3, reflective method invocation was forty times slower on my machine than normal method invocation. Reflection was rearchitected in release 1.4 for greatly improved performance, but it is still twice as slow as normal access, and the gap is unlikely to narrow.

The reflection facility was originally designed for component-based application builder tools. Such tools generally load classes on demand and use reflection to find out what methods and constructors they support. The tools let their users interactively construct applications that access these classes, but the generated applications access the classes normally, not reflectively. Reflection is used only at *design time*. **As a rule, objects should not be accessed reflectively in normal applications at run time.**

There are a few sophisticated applications that demand the use of reflection. Examples include class browsers, object inspectors, code analysis tools, and interpretive embedded systems. Reflection is also appropriate for use in RPC systems to eliminate the need for stub compilers. If you have any doubts as to whether your application falls into one of these categories, 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 unavailable at compile time, there exists at compile time an appropriate interface or superclass by which to refer to the class (Item 34). If this is the case, you can **create instances reflectively and access them normally via their interface or superclass**. If the appropriate constructor has no parameters, as is usually the case, then you don't even need to use the `java.lang.reflect` package; the `Class.newInstance` method provides the required functionality.

For example, here's a program that creates a `Set` 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 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, as 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 cl = null;
    try {
        cl = Class.forName(args[0]);
    } catch(ClassNotFoundException e) {
        System.err.println("Class not found.");
        System.exit(1);
    }

    // Instantiate the class
    Set s = null;
    try {
        s = (Set) cl.newInstance();
    } catch(IllegalAccessException e) {
        System.err.println("Class not accessible.");
        System.exit(1);
    } catch(InstantiationException e) {
        System.err.println("Class not instantiable.");
        System.exit(1);
    }

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

While this program is just a toy, the technique that it demonstrates is very 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, the technique that it demonstrates is sufficient to implement a full-blown *service provider framework* (Item 1). Most of the time, this technique is all that you need in the way of reflection.

You can see two disadvantages of reflection in the example. First, the example is capable of generating three run-time errors, all of which would have been compile-time errors if reflective instantiation were not used. Second, it takes twenty 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. These disadvantages are, however, restricted to the part of the program that instantiates the object. Once instantiated, it 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.

A legitimate, if rare, use of reflection is to break a class's dependencies on other classes, methods, or fields that may be absent at run time. 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 run time. 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 36: Use native methods judiciously

The Java Native Interface (JNI) allows Java applications to call *native methods*, which are special methods written in *native programming languages* such as C or C++. Native methods can perform arbitrary computation in native languages before returning to the Java programming language.

Historically, native methods have had three main uses. They provided access to platform-specific facilities such as registries and file locks. They provided access to libraries of legacy code, which could in turn provide access to legacy data. Finally, native methods were 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 as the Java platform matures, it provides more and more features previously found only in host platforms. For example, the `java.util.prefs` package, added in release 1.4, offers the functionality of a registry. It is also legitimate to use native methods to access legacy code, but there are better ways to access some legacy code. For example, the JDBC API provides access to legacy databases.

As of release 1.3, it is rarely advisable to use native methods for improved performance. In early releases, it was often necessary, but JVM implementations have gotten much faster. For most tasks, it is now possible to obtain comparable performance without resorting to native methods. By way of example, when `java.math` was added to the platform in release 1.1, `BigInteger` was implemented atop a fast multiprecision arithmetic library written in C. At the time, this was necessary for adequate performance. In release 1.3, `BigInteger` was rewritten entirely in Java and carefully tuned. The new version is faster than the original on all of Sun's 1.3 JVM implementations for most operations and operand sizes.

The use of native methods has serious disadvantages. Because native languages are not *safe* (Item 24), applications using native methods are no longer immune to memory corruption errors. Because native languages are platform dependent, applications using native methods are no longer freely portable. Native code must be recompiled for each target platform and may require modification as well. There is a high fixed cost associated with going into and out of native code, so native methods can *decrease* performance if they do only a small amount of work. Finally, native methods are tedious to write and difficult to read.

In summary, think twice before using native methods. Rarely, if ever, use them for improved performance. If you must use native methods to access low-level resources or legacy 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 37: Optimize judiciously

There are three aphorisms concerning optimization that everyone should know. They are perhaps beginning to suffer from overexposure, but in case you aren't yet familiar with them, here they are:

> 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 modules, so individual decisions can be changed without affecting the remainder of the system (Item 12).

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 nearly 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 should 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 modules 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 24). 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 14). 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 34).

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, as of release 1.3, allocating small objects is relatively inexpensive, allocating millions of objects needlessly can do real harm to performance.

In this case, several alternatives existed. Ideally, `Dimension` should have been immutable (Item 13); alternatively, the `getSize` method could have been replaced by two methods returning the individual primitive components of a `Dimension` object. In fact, two such methods were added to the `Component` API in the 1.2 release 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 it causes will be with you for life.

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 reveals that programs spend 80 percent of their time in 20 percent of their code.

Profiling tools can help you decide where to focus your optimization efforts. Such tools give you run-time 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's 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. The Java 2 SDK comes with a simple profiler, and several more sophisticated profiling tools are available commercially.

The need to measure the effects of optimization is even greater on the Java platform than on more traditional platforms, as the Java programming language does not have a strong *performance model*. The relative costs of the various primitive operations are not well defined. The "semantic gap" between what the programmer writes and what the CPU executes is far greater than in traditional compiled languages which makes it very difficult to reliably predict the performance consequences of any optimization. There are plenty of performance myths floating around that turn out to be half-truths or outright lies.

Not only is the performance model ill-defined, but it varies from JVM implementation to JVM implementation and from release to release. If you will be running your program on multiple JVM implementations, 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 JVM implementations.

To summarize, do not strive to write fast programs—strive to write good ones; speed will follow. Do think about performance issues while you're designing systems and 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 problems 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 38: 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, 6.8]. 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, and fields. 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 names should be hierarchical with the parts separated by periods. Parts 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 top-level domain first, for example, `edu.cmu`, `com.sun`, `gov.nsa`. The standard libraries and optional packages, whose names begin with `java` and `javax`, are exceptions to this rule. Users must not create packages whose names begin with `java` or `javax`. Detailed rules for converting Internet domain names to package name prefixes can be found in *The Java Language Specification* [JLS, 7.7].

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

Many packages have names with just one part in addition to the internet domain name. Additional parts are appropriate for large facilities whose size demands that they be broken up into an informal hierarchy. For example, the `javax.swing` package has a rich hierarchy of packages with names such as `javax.swing.plaf.metal`. Such packages are often referred to as subpackages, although they are subpackages by convention only; there is no linguistic support for package hierarchies.

Class and interface names should consist of one or more words, with the first letter of each word capitalized, for example, `Timer` or `TimerTask`. Abbreviations are to be avoided, except for acronyms and certain common abbreviations like `max` and `min`. There is little consensus as to whether acronyms should be uppercase or have only their first letter capitalized. While uppercase is more common, 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`, `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 13), then it is a constant field. If the type is potentially mutable, it can still be a constant field if the referenced object is immutable. For example, a typesafe enum can export its universe of enumeration constants in an immutable `List` constant (page 107). 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 the local variable occurs, for example, `i`, `xref`, `houseNumber`.

For quick reference, Table 7.1 shows examples of typographical conventions.

Table 7.1: Examples of Typographical Conventions

Identifier Type	Examples
Package	`com.sun.medialib`, `com.sun.jdi.event`
Class or Interface	`Timer`, `TimerTask`, `KeyFactorySpi`, `HttpServlet`
Method or Field	`remove`, `ensureCapacity`, `getCrc`
Constant Field	`VALUES`, `NEGATIVE_INFINITY`
Local Variable	`i`, `xref`, `houseNumber`

The grammatical naming conventions are more flexible and more controversial than the typographical conventions. There are no grammatical naming conventions to speak of for packages. Classes are generally named with a noun or noun phrase, for example, `Timer` or `BufferedWriter`. Interfaces are named like

classes, for example, `Collection` or `Comparator`, or with an adjective ending in "-able" or "-ible," for example, `Runnable` or `Accessible`.

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

Methods that return a nonboolean 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 only the third form (beginning with "get") is acceptable, but there is no 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" is mandatory if the class containing the method is a *Bean* [JavaBeans], and it's advisable if you're considering turning the class into a Bean at a later time. Also, there is strong precedent for this form if the class contains a method to set the same attribute. In this case, the two methods should be named `get`*Attribute* and `set`*Attribute*.

A few method names deserve special mention. Methods that convert the type of an object, returning an independent object of a different type, are often called `to`*Type*, for example, `toString`, `toArray`. Methods that return a *view* (Item 4) 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 are `valueOf` and `getInstance` (Item 1, page 9).

Grammatical conventions for field names are less well established and less important than those for class, interface, and method names, as well-designed APIs contain few if any exposed fields. Fields of type `boolean` are typically 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 are 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, 6.8], "These conventions should not be followed slavishly if long-held conventional usage dictates otherwise." Use common sense.