Chapter 1

Optimizing Java Code

Many Android application developers have a good practical knowledge of the Java language from previous experience. Since its debut in 1995, Java has become a very popular programming language. While some surveys show that Java lost its luster trying to compete with other languages like Objective-C or C#, some of these same surveys rank Java as the number 1 language popularity-wise. Naturally, with mobile devices outselling personal computers and the success of the Android platform (700,000 activations per day in December 2011) Java is becoming more relevant in today's market than ever before.

Developing applications for mobile devices can be quite different from developing applications for personal computers. Today's portable devices can be quite powerful, but in terms of performance, they lag behind personal computers. For example, some benchmarks show a quad-core Intel Core i7 processor running about 20 times faster than the dual-core Nvidia Tegra 2 that is found in the Samsung Galaxy Tab 10.1.

> **NOTE:** Benchmark results are to be taken with a grain of salt since they often measure only part of a system and do not necessarily represent a typical use-case.

This chapter shows you how to make sure your Java applications perform well on Android devices, whether they run the latest Android release or not. First, we take a look at how Android executes your code. Then, we review several techniques to optimize the implementation of a famous mathematical series, including how to take advantage of the latest APIs Android offers. Finally, we review a few techniques to improve your application's responsiveness and to use databases more efficiently.

Before you jump in, you should realize code optimization is not the first priority in your application development. Delivering a good user experience and focusing on code maintainability should be among your top priorities. In fact, code optimization should be one of your last priorities, and may not even be part of the process altogether. However, good practices can help you reach an acceptable level of performance without having you go back to your code, asking yourself "what did I do wrong?" and having to spend additional resources to fix it.

H. Guihot, *Pro Android Apps Performance Optimization*

How Android Executes Your Code

While Android developers use Java, the Android platform does not include a Java Virtual Machine (VM) for executing code. Instead, applications are compiled into Dalvik bytecode, and Android uses its Dalvik VM to execute it. The Java code is still compiled into Java bytecode, but this Java bytecode is then compiled into Dalvik bytecode by the dex compiler, dx (an SDK tool). Ultimately, your application will contain only the Dalvik bytecode, not the Java bytecode.

For example, an implementation of a method that computes the n^{th} term of the Fibonacci series is shown in Listing 1–1 together with the class definition. The Fibonacci series is defined as follows:

$F_0 = 0$
$F_1 = 1$
$F_n = F_{n-2} + F_{n-1}$ for n greater than 1

Listing 1–1. *Naïve Recursive Implementation of Fibonacci Series*

```
public class Fibonacci {
    public static long computeRecursively (int n)
    {
        if (n > 1) return computeRecursively(n-2) + computeRecursively(n-1);
        return n;
    }
}
```

> **NOTE:** A trivial optimization was done by returning n when n equals 0 or 1 instead of adding another "if" statement to check whether n equals 0 or 1.

An Android application is referred to as an APK since applications are compiled into a file with the apk extension (for example, APress.apk), which is simply an archive file. One of the files in the archive is classes.dex, which contains the application's bytecode. The Android toolchain provides a tool, dexdump, which can convert the binary form of the code (contained in the APK's classes.dex file) into human-readable format.

> **TIP:** Because an apk file is simply a ZIP archive, you can use common archive tools such as WinZip or 7-Zip to inspect the content of an apk file..

Listing 1–2 shows the matching Dalvik bytecode.

Listing 1–2. *Human-Readable Dalvik Bytecode of Fibonacci.computeRecursively*

```
002548:                |[002548] com.apress.proandroid.Fibonacci.computeRecursively:(I)J
002558: 1212           |0000: const/4 v2, #int 1 // #1
00255a: 3724 1100      |0001: if-le v4, v2, 0012 // +0011
00255e: 1220           |0003: const/4 v0, #int 2 // #2
002560: 9100 0400      |0004: sub-int v0, v4, v0
```

```
002564: 7110 3d00 0000  |0006: invoke-static {v0},
    Lcom/apress/proandroid/Fibonacci;.computeRecursively:(I)J
00256a: 0b00            |0009: move-result-wide v0
00256c: 9102 0402       |000a: sub-int v2, v4, v2
002570: 7110 3d00 0200  |000c: invoke-static {v2},
    Lcom/apress/proandroid/Fibonacci;.computeRecursively:(I)J
002576: 0b02            |000f: move-result-wide v2
002578: bb20            |0010: add-long/2addr v0, v2
00257a: 1000            |0011: return-wide v0
00257c: 8140            |0012: int-to-long v0, v4
00257e: 28fe            |0013: goto 0011 // -0002
```

The first number on each line specifies the absolute position of the code within the file. Except on the very first line (which shows the method name), it is then followed by one or more 16-bit bytecode units, followed by the position of the code within the method itself (relative position, or label), the opcode mnemonic and finally the opcode's parameter(s). For example, the two bytecode units 3724 1100 at address 0x00255a translate to "if-le v4, v2, 0012 // +0011", which basically means "if content of virtual register v4 is less than or equal to content of virtual register v2 then go to label 0x0012 by skipping 17 bytecode units" (17_{10} equals 11_{16}). The term "virtual register" refers to the fact that these are not actual hardware registers but instead the registers used by the Dalvik virtual machine.

Typically, you would not need to look at your application's bytecode. This is especially true with Android 2.2 (codename Froyo) and later versions since a Just-In-Time (JIT) compiler was introduced in Android 2.2. The Dalvik JIT compiler compiles the Dalvik bytecode into native code, which can execute significantly faster. A JIT compiler (sometimes referred to simply as a JIT) improves performance dramatically because:

- Native code is directly executed by the CPU without having to be interpreted by a virtual machine.
- Native code can be optimized for a specific architecture.

Benchmarks done by Google showed code executes 2 to 5 times faster with Android 2.2 than Android 2.1. While the results may vary depending on what your code does, you can expect a significant increase in speed when using Android 2.2 and later versions.

The absence of a JIT compiler in Android 2.1 and earlier versions may affect your optimization strategy significantly. If you intend to target devices running Android 1.5 (codename Cupcake), 1.6 (codename Donut), or 2.1 (codename Éclair), most likely you will need to review more carefully what you want or need to provide in your application. Moreover, devices running these earlier Android versions are older devices, which are less powerful than newer ones. While the market share of Android 2.1 and earlier devices is shrinking, they still represent about 12% as of December 2011). Possible strategies are:

- Don't optimize at all. Your application could be quite slow on these older devices.

- Require minimum API level 8 in your application, which can then be installed only on Android 2.2 or later versions.
- Optimize for older devices to offer a good user experience even when no JIT compiler is present. This could mean disabling features that are too CPU-heavy.

> **TIP:** Use `android:vmSafeMode` in your application's manifest to enable or disable the JIT compiler. It is enabled by default (if it is available on the platform). This attribute was introduced in Android 2.2.

Now it is time to run the code on an actual platform and see how it performs. If you are familiar with recursion and the Fibonacci series, you might guess that it is going to be slow. And you would be right. On a Samsung Galaxy Tab 10.1, computing the thirtieth Fibonacci number takes about 370 milliseconds. With the JIT compiler disabled, it takes about 440 milliseconds. If you decide to include that function in a Calculator application, users will become frustrated because the results cannot be computed "immediately." From a user's point of view, results appear instantaneous if they can be computed in 100 milliseconds or less. Such a response time guarantees a very good user experience, so this is what we are going to target.

Optimizing Fibonacci

The first optimization we are going to perform eliminates a method call, as shown in Listing 1–3. As this implementation is recursive, removing a single call in the method dramatically reduces the total number of calls. For example, `computeRecursively(30)` generated 2,692,537 calls while computeRecursivelyWithLoop(30) generated "only" 1,346,269. However, the performance of this method is still not acceptable considering the response-time criteria defined above, 100 milliseconds or less, as `computeRecursivelyWithLoop(30)` takes about 270 milliseconds to complete.

Listing 1–3. *Optimized Recursive Implementation of Fibonacci Series*

```
public class Fibonacci {
    public static long computeRecursivelyWithLoop (int n)
    {
        if (n > 1) {
            long result = 1;
            do {
                result += computeRecursivelyWithLoop(n-2);
                n--;
            } while (n > 1);
            return result;
        }
        return n;
    }
}
```

NOTE: This is not a true tail-recursion optimization.

From Recursive To Iterative

For the second optimization, we switch from a recursive implementation to an iterative one. Recursive algorithms often have a bad reputation with developers, especially on embedded systems without much memory, because they tend to consume a lot of stack space and, as we just saw, can generate too many method calls. Even when performance is acceptable, a recursive algorithm can cause a stack overflow and crash an application. An iterative implementation is therefore often preferred whenever possible. Listing 1–4 shows what is considered a textbook iterative implementation of the Fibonacci series.

Listing 1–4. *Iterative Implementation of Fibonacci Series*

```
public class Fibonacci {
    public static long computeIteratively (int n)
    {
        if (n > 1) {
            long a = 0, b = 1;
            do {
                long tmp = b;
                b += a;
                a = tmp;
            } while (--n > 1);
            return b;
        }
        return n;
    }
}
```

Because the n^{th} term of the Fibonacci series is simply the sum of the two previous terms, a simple loop can do the job. Compared to the recursive algorithms, the complexity of this iterative algorithm is also greatly reduced because it is linear. Consequently, its performance is also much better, and `computeIteratively(30)` takes less than 1 millisecond to complete. Because of its linear nature, you can use such an algorithm to compute terms beyond the 30^{th}. For example, `computeIteratively(50000)` takes only 2 milliseconds to return a result and, by extrapolation, you could guess `computeIteratively(500000)` would take between 20 and 30 milliseconds to complete.

While such performance is more than acceptable, it is possible to to achieve even faster results with a slightly modified version of the same algorithm, as showed in Listing 1–5. This new version computes two terms per iteration, and the total number of iterations is halved. Because the number of iterations in the original iterative algorithm could be odd, the initial values for a and b are modified accordingly: the series starts with a=0 and b=1 when n is odd, and it starts with a=1 and b=1 (Fib(2)=1) when n is even.

Listing 1–5. *Modified Iterative Implementation of Fibonacci Series*

```
public class Fibonacci {
    public static long computeIterativelyFaster (int n)
    {
        if (n > 1) {
            long a, b = 1;
            n--;
            a = n & 1;
            n /= 2;
            while (n-- > 0) {
                a += b;
                b += a;
            }
            return b;
        }
        return n;
    }
}
```

Results show this modified iterative version is about twice as fast as the original one.

While these iterative implementations are fast, they do have one major problem: they don't return correct results. The issue lies with the return value being stored in a long value, which is 64-bit. The largest Fibonacci number that can fit in a signed 64-bit value is 7,540,113,804,746,346,429 or, in other words, the 92nd Fibonacci number. While the methods will still return without crashing the application for values of n greater than 92, the results will be incorrect because of an overflow: the 93rd Fibonacci number would be negative! The recursive implementations actually have the same limitation, but one would have to be quite patient to eventually find out.

> **NOTE:** Java specifies the size of all primitive types (except boolean): long is 64-bit, int is 32-bit, and short is 16-bit. All integer types are signed.

BigInteger

Java offers just the right class to fix this overflow problem: `java.math.BigInteger`. A `BigInteger` object can hold a signed integer of arbitrary size and the class defines all the basic math operations (in addition to some not-so-basic ones). Listing 1–6 shows the `BigInteger` version of `computeIterativelyFaster`.

> **TIP:** The `java.math` package also defines `BigDecimal` in addition to `BigInteger`, while `java.lang.Math` provides math constant and operations. If your application does not need double precision, use Android's `FloatMath` instead of `Math` for performance (although gains may vary depending on platform).

Listing 1–6. *BigInteger Version of Fibonacci.computeIterativelyFaster*

```
public class Fibonacci {
    public static BigInteger computeIterativelyFasterUsingBigInteger (int n)
    {
        if (n > 1) {
            BigInteger a, b = BigInteger.ONE;
            n--;
            a = BigInteger.valueOf(n & 1);
            n /= 2;
            while (n-- > 0) {
                a = a.add(b);
                b = b.add(a);
            }
            return b;
        }
        return (n == 0) ? BigInteger.ZERO : BigInteger.ONE;
    }
}
```

That implementation guarantees correctness as overflows can no longer occur. However, it is not without problems because, again, it is quite slow: a call to `computeIterativelyFasterUsingBigInteger(50000)` takes about 1.3 seconds to complete. The lackluster performance can be explained by three things:

- `BigInteger` is immutable.
- `BigInteger` is implemented using `BigInt` and native code.
- The larger the numbers, the longer it takes to add them together.

Since `BigInteger` is immutable, we have to write "a = a.add(b)" instead of simply "a.add(b)". Many would assume "a.add(b)" is the equivalent of "a += b" and many would be wrong: it is actually the equivalent of "a + b". Therefore, we have to write "a = a.add(b)" to assign the result. That small detail is extremely significant as "a.add(b)" creates a new `BigInteger` object that holds the result of the addition.

Because of `BigInteger`'s current internal implementation, an additional `BigInt` object is created for every `BigInteger` object that is allocated. This results in twice as many objects being allocated during the execution of `computeIterativelyFasterUsingBigInteger`: about 100,000 objects are created when calling `computeIterativelyFasterUsingBigInteger (50000)` (and all of them but one will become available for garbage collection almost immediately). Also, `BigInt` is implemented using native code and calling native code from Java (using JNI) has a certain overhead.

The third reason is that very large numbers do not fit in a single, long 64-bit value. For example, the 50,000th Fibonacci number is 34,7111–bit long.

> **NOTE:** `BigInteger`'s internal implementation (`BigInteger.java`) may change in future Android releases. In fact, internal implementation of any class can change.

For performance reasons, memory allocations should be avoided whenever possible in critical paths of the code. Unfortunately, there are some cases where allocations are needed, for example when working with immutable objects like `BigInteger`. The next optimization focuses on reducing the number of allocations by switching to a different algorithm. Based on the Fibonacci *Q*-matrix, we have the following:

$$F_{2n-1} = F_n^2 + F_{n-1}^2$$

$$F_{2n} = (2F_{n-1} + F_n) * F_n$$

This can be implemented using `BigInteger` again (to guarantee correct results), as shown in Listing 1–7.

Listing 1–7. *Faster Recursive Implementation of Fibonacci Series Using BigInteger*

```
public class Fibonacci {
    public static BigInteger computeRecursivelyFasterUsingBigInteger (int n)
    {
        if (n > 1) {
            int m = (n / 2) + (n & 1); // not obvious at first - wouldn't it be great to
have a better comment here?
            BigInteger fM = computeRecursivelyFasterUsingBigInteger(m);
            BigInteger fM_1 = computeRecursivelyFasterUsingBigInteger(m - 1);
            if ((n & 1) == 1) {
                // F(m)^2 + F(m-1)^2
                return fM.pow(2).add(fM_1.pow(2)); // three BigInteger objects created
            } else {
                // (2*F(m-1) + F(m)) * F(m)
                return fM_1.shiftLeft(1).add(fM).multiply(fM); // three BigInteger
objects created
            }
        }
        return (n == 0) ? BigInteger.ZERO : BigInteger.ONE; // no BigInteger object
created
    }

    public static long computeRecursivelyFasterUsingBigIntegerAllocations(int n)
    {
        long allocations = 0;
        if (n > 1) {
            int m = (n / 2) + (n & 1);
            allocations += computeRecursivelyFasterUsingBigIntegerAllocations(m);
            allocations += computeRecursivelyFasterUsingBigIntegerAllocations(m - 1);

            // 3 more BigInteger objects allocated
            allocations += 3;
        }
        return allocations; // approximate number of BigInteger objects allocated when
computeRecursivelyFasterUsingBigInteger(n) is called
    }
}
```

A call to `computeRecursivelyFasterUsingBigInteger(50000)` returns in about 1.6 seconds. This shows this latest implementation is actually slower than the fastest iterative implementation we have so far. Again, the number of allocations is the culprit as

around 200,000 objects were allocated (and almost immediately marked as eligible for garbage collection).

> **NOTE:** The actual number of allocations is less than what `computeRecursivelyFasterUsingBigIntegerAllocations` would return. Because `BigInteger`'s implementation uses preallocated objects such as `BigInteger.ZERO`, `BigInteger.ONE`, or `BigInteger.TEN`, there may be no need to allocate a new object for some operations. You would have to look at Android's `BigInteger` implementation to know exactly how many objects are allocated.

This implementation is slower, but it is a step in the right direction nonetheless. The main thing to notice is that even though we need to use `BigInteger` to guarantee correctness, we don't have to use `BigInteger` for every value of n. Since we know the primitive type `long` can hold results for n less than or equal to 92, we can slightly modify the recursive implementation to mix `BigInteger` and primitive type, as shown in Listing 1–8.

Listing 1–8. *Faster Recursive Implementation of Fibonacci Series Using BigInteger and long Primitive Type*

```
public class Fibonacci {
    public static BigInteger computeRecursivelyFasterUsingBigIntegerAndPrimitive(int n)
    {
        if (n > 92) {
            int m = (n / 2) + (n & 1);
            BigInteger fM = computeRecursivelyFasterUsingBigIntegerAndPrimitive(m);
            BigInteger fM_1 = computeRecursivelyFasterUsingBigIntegerAndPrimitive(m -
1);
            if ((n & 1) == 1) {
                return fM.pow(2).add(fM_1.pow(2));
            } else {
                return fM_1.shiftLeft(1).add(fM).multiply(fM); // shiftLeft(1) to
multiply by 2
            }
        }
        return BigInteger.valueOf(computeIterativelyFaster(n));
    }

    private static long computeIterativelyFaster(int n)
    {
        // see Listing 1-5 for implementation
    }
}
```

A call to `computeRecursivelyFasterUsingBigIntegerAndPrimitive(50000)` returns in about 73 milliseconds and results in about 11,000 objects being allocated: a small modification in the algorithm yields results about 20 times faster and about 20 times fewer objects being allocated. Quite impressive! It is possible to improve the performance even further by reducing the number of allocations, as shown in Listing 1–9. Precomputed results can be quickly generated when the Fibonacci class is first loaded, and these results can later be used directly.

Listing 1–9. *Faster Recursive Implementation of Fibonacci Series Using BigInteger and Precomputed Results*

```
public class Fibonacci {
    static final int PRECOMPUTED_SIZE= 512;
    static BigInteger PRECOMPUTED[] = new BigInteger[PRECOMPUTED_SIZE];

    static {
        PRECOMPUTED[0] = BigInteger.ZERO;
        PRECOMPUTED[1] = BigInteger.ONE;
        for (int i = 2; i < PRECOMPUTED_SIZE; i++) {
            PRECOMPUTED[i] = PRECOMPUTED[i-1].add(PRECOMPUTED[i-2]);
        }
    }

    public static BigInteger computeRecursivelyFasterUsingBigIntegerAndTable(int n)
    {
        if (n > PRECOMPUTED_SIZE - 1) {
            int m = (n / 2) + (n & 1);
            BigInteger fM = computeRecursivelyFasterUsingBigIntegerAndTable (m);
            BigInteger fM_1 = computeRecursivelyFasterUsingBigIntegerAndTable (m - 1);
            if ((n & 1) == 1) {
                return fM.pow(2).add(fM_1.pow(2));
            } else {
                return fM_1.shiftLeft(1).add(fM).multiply(fM);
            }
        }
        return PRECOMPUTED[n];
    }
}
```

The performance of this implementation depends on `PRECOMPUTED_SIZE`: the bigger, the faster. However, memory usage may become an issue since many `BigInteger` objects will be created and remain in memory for as long as the Fibonacci class is loaded. It is possible to merge the implementations shown in Listing 1–8 and Listing 1–9, and use a combination of precomputed results and computations with primitive types. For example, terms 0 to 92 could be computed using `computeIterativelyFaster`, terms 93 to 127 using precomputed results and any other term using recursion. As a developer, you are responsible for choosing the best implementation, which may not always be the fastest. Your choice will be based on various factors, including:

- What devices and Android versions your application target
- Your resources (people and time)

As you may have already noticed, optimizations tend to make the source code harder to read, understand, and maintain, sometimes to such an extent that you would not recognize your own code weeks or months later. For this reason, it is important to carefully think about what optimizations you really need and how they will affect your application development, both in the short term and in the long term. It is always recommended you first implement a working solution before you think of optimizing it (and make sure you save a copy of the working solution). After all, you may realize optimizations are not needed at all, which could save you a lot of time. Also, make sure you include comments in your code for everything that is not obvious to a person with ordinary skill in the art. Your coworkers will thank you, and you may give yourself a pat

on the back as well when you stumble on some of your old code. My poor comment in Listing 1–7 is proof.

> **NOTE:** All implementations disregard the fact that n could be negative. This was done intentionally to make a point, but your code, at least in all public APIs, should throw an IllegalArgumentException whenever appropriate.

Caching Results

When computations are expensive, it may be a good idea to remember past results to make future requests faster. Using a cache is quite simple as it typically translates to the pseudo-code shown in Listing 1–10.

Listing 1–10. *Using a Cache*

```
result = cache.get(n); // input parameter n used as key
if (result == null) {
    // result was not in the cache so we compute it and add it
    result = computeResult(n);
    cache.put(n, result); // n is the key, result is the value
}
return result;
```

The faster recursive algorithm to compute Fibonacci terms yields many duplicate calculations and could greatly benefit from memoization. For example, computing the 50,000th term requires computing the 25,000th and 24,999th terms. Computing the 25,000th term requires computing the 12,500th and 12,499th terms, while computing the 24,999th term requires computing… the same 12,500th and 12,499th terms again! Listing 1–11 shows a better implementation using a cache.

If you are familiar with Java, you may be tempted to use a `HashMap` as your cache, and it would work just fine. However, Android defines `SparseArray`, a class that is intended to be more efficient than `HashMap` when the key is an integer value: `HashMap` would require the key to be of type `java.lang.Integer`, while `SparseArray` uses the primitive type int for keys. Using `HashMap` would therefore trigger the creation of many `Integer` objects for the keys, which `SparseArray` simply avoids.

Listing 1–11. *Faster Recursive Implementation Using BigInteger, long Primitive TypeAnd Cache*

```
public class Fibonacci {
    public static BigInteger computeRecursivelyWithCache (int n)
    {
        SparseArray<BigInteger> cache = new SparseArray<BigInteger>();
        return computeRecursivelyWithCache(n, cache);
    }

    private static BigInteger computeRecursivelyWithCache (int n,
SparseArray<BigInteger> cache)
    {
        if (n > 92) {
            BigInteger fN = cache.get(n);
```

```
            if (fN == null) {
                int m = (n / 2) + (n & 1);
                BigInteger fM = computeRecursivelyWithCache(m, cache);
                BigInteger fM_1 = computeRecursivelyWithCache(m - 1, cache);
                if ((n & 1) == 1) {
                    fN = fM.pow(2).add(fM_1.pow(2));
                } else {
                    fN = fM_1.shiftLeft(1).add(fM).multiply(fM);
                }
                cache.put(n, fN);
            }
            return fN;
        }
        return BigInteger.valueOf(iterativeFaster(n));
    }

    private static long iterativeFaster (int n) { /* see Listing 1-5 for implementation
*/ }
}
```

Measurements showed `computeRecursivelyWithCache(50000)` takes about 20 milliseconds to complete, or about 50 fewer milliseconds than a call to `computeRecursivelyFasterUsingBigIntegerAndPrimitive(50000)`. Obviously, the difference is exacerbated as n grows: when n equals 200,000 the two methods complete in 50 and 330 milliseconds respectively.

Because many fewer `BigInteger` objects are allocated, the fact that `BigInteger` is immutable is not as big of a problem when using the cache. However, remember that three `BigInteger` objects are still created (two of them being very short-lived) when fN is computed, so using mutable big integers would still improve performance.

Even though using `HashMap` instead of `SparseArray` may be a little slower, it would have the benefit of making the code Android-independent, that is, you could use the exact same code in a non-Android environment (without `SparseArray`).

> **NOTE:** Android defines multiple types of sparse arrays: `SparseArray` (to map integers to objects), `SparseBooleanArray` (to map integers to booleans), and `SparseIntArray` (to map integers to integers).

android.util.LruCache<K, V>

Another class worth mentioning is `android.util.LruCache<K, V>`, introduced in Android 3.1 (codename Honeycomb MR1), which makes it easy to define the maximum size of the cache when it is allocated. Optionally, you can also override the `sizeOf()` method to change how the size of each cache entry is computed. Because it is only available in Android 3.1 and later, you may still end up having to use a different class to implement a cache in your own application if you target Android revisions older than 3.1. This is a very likely scenario considering Android 3.1 as of today represents only a very small portion of the Android devices in use. An alternative solution is to extend

`java.util.LinkedHashMap` and override `removeEldestEntry`. An LRU cache (for Least Recently Used) discards the least recently used items first. In some applications, you may need exactly the opposite, that is, a cache that discards the most recently used items first. Android does not define such an `MruCache` class for now, which is not surprising considering MRU caches are not as commonly used.

Of course, a cache can be used to store information other than computations. A common use of a cache is to store downloaded data such as pictures and still maintain tight control over how much memory is consumed. For example, override LruCache's `sizeOf` method to limit the size of the cache based on a criterion other than simply the number of entries in the cache. While we briefly discussed the LRU and MRU strategies, you may want to use different replacement strategies for your own cache to maximize cache hits. For example, your cache could first discard the items that are not costly to recreate, or simply randomly discard items. Follow a pragmatic approach and design your cache accordingly. A simple replacement strategy such as LRU can yield great results and allow you to focus your resources on other, more important problems.

We've looked at several different techniques to optimize the computation of Fibonacci numbers. While each technique has its merits, no one implementation is optimal. Often the best results are achieved by combining multiple various techniques instead of relying on only one of them. For example, an even faster implementation would use precomputations, a cache mechanism, and maybe even slightly different formulas. (Hint: what happens when n is a multiple of 4?) What would it take to compute $F_{\text{Integer.MAX\_VALUE}}$ in less than 100 milliseconds?

API Levels

The LruCache class mentioned above is a good example of why you need to know what API level you are going to target. A new version of Android is released approximately every six months, with new APIs only available from that release. Any attempt to call an API that does not exist results in a crash, bringing not only frustration for the user but also shame to the developer. For example, calling Log.wtf(TAG, "Really?") on an Android 1.5 device crashes the application, as Log.wtf was introduced in Android 2.2 (API level 8). What a terrible failure indeed that would be. Table 1–1 shows the performance improvements made in the various Android versions.

Table 1–1. *Android Versions*

API level	Version	Name	Significant performance improvements
1	1.0	Base	
2	1.1	Base 1.1	
3	1.5	Cupcake	Camera start-up time, image capture time, faster acquisition of GPS location, NDK support
4	1.6	Donut	
5	2.0	Éclair	Graphics
6	2.0.1	Éclair 0.1	
7	2.1	Éclair MR1	
8	2.2	Froyo	V8 Javascript engine (browser), JIT compiler, memory management
9	2.3.0 2.3.1 2.3.2	Gingerbread	Concurrent garbage collector, event distribution, better OpenGL drivers
10	2.3.3 2.3.4	Gingerbread MR1	
11	3.0	Honeycomb	Renderscript, animations, hardware-accelerated 2D graphics, multicore support
12	3.1	Honeycomb MR1	LruCache, partial invalidates in hardware-accelerated views, new Bitmap.setHasAlpha() API
13	3.2	Honeycomb MR2	
14	4.0	Ice Cream Sandwich	Media effects (transformation filters), hardware-accelerated 2D graphics (required)

However, your decision to support a certain target should normally not be based on which API you want to use, but instead on what market you are trying to reach. For example, if your target is primarily tablets and not cell phones, then you could target Honeycomb. By doing so, you would limit your application's audience to a small subset of Android devices, because Honeycomb represents only about 2.4% as of December 2011, and not all tablets support Honeycomb. (For example, Barnes & Noble's Nook

uses Android 2.2 while Amazon's Kindle Fire uses Android 2.3.) Therefore, supporting older Android versions could still make sense.

The Android team understood that problem when they released the Android Compatibility package, which is available through the SDK Updater. This package contains a static library with some of the new APIs introduced in Android 3.0, namely the fragment APIs. Unfortunately, this compatibility package contains only the fragment APIs and does not address the other APIs that were added in Honeycomb. Such a compatibility package is the exception, not the rule. Normally, an API introduced at a specific API level is not available at lower levels, and it is the developer's responsibility to choose APIs carefully.

To get the API level of the Android platform, you can use Build.VERSION.SDK_INT. Ironically, this field was introduced in Android 1.6 (API level 4), so trying to retrieve the version this way would also result in a crash on Android 1.5 or earlier. Another option is to use Build.VERSION.SDK, which has been present since API level 1. However, this field is now deprecated, and the version strings are not documented (although it would be pretty easy to understand how they have been created).

> **TIP:** Use reflection to check whether the SDK_INT field exists (that is, if the platform is Android 1.6 or later). See Class.forName("android.os.Build$VERSION").getField("SDK").

Your application's manifest file should use the `<uses-sdk>` element to specify two important things:

- The minimum API level required for the application to run (android:minSdkVersion)
- The API level the application targets (android:targetSdkVersion)

It is also possible to specify the maximum API level (android:maxSdkVersion), but using this attribute is not recommended. Specifying maxSdkVersion could even lead to applications being uninstalled automatically after Android updates. The target API level is the level at which your application has been explicitly tested.

By default, the minimum API level is set to 1 (meaning the application is compatible with all Android versions). Specifying an API level greater than 1 prevents the application from being installed on older devices. For example, android:minSdkVersion="4" guarantees Build.VERSION.SDK_INT can be used without risking any crash. The minimum API level does not have to be the highest API level you are using in your application as long as you make sure you call only a certain API when the API actually exists, as shown in Listing 1–12.

Listing 1–12. *Calling a SparseArray Method Introduced in Honeycomb (API Level 11)*

```
if (Build.VERSION.SDK_INT >= Build.VERSION_CODES.HONEYCOMB) {
    sparseArray.removeAt(1); // API level 11 and above
} else {
    int key = sparseArray.keyAt(1); // default implementation is slower
    sparseArray.remove(key);
}
```

This kind of code is more frequent when trying to get the best performance out of your Android platform since you want to use the best API for the job while you still want your application to be able to run on an older device (possibly using slower APIs).

Android also uses these attributes for other things, including determining whether the application should run in screen compatibility mode. Your application runs in screen compatibility mode if minSdkVersion is set to 3 or lower, and targetSdkVersion is not set to 4 or higher. This would prevent your application from displaying in full screen on a tablet, for example, making it much harder to use. Tablets have become very popular only recently, and many applications have not been updated yet, so it is not uncommon to find applications that do not display properly on a big screen.

> **NOTE:** Android Market uses the minSdkVersion and maxSdkVersion attributes to filter applications available for download on a particular device. Other attributes are used for filtering as well. Also, Android defines two versions of screen compatibility mode, and their behaviors differ. Refer to "Supporting Multiple Screens" on `http://d.android.com/guide` for a complete description.

Instead of checking the version number, as shown in Listing 1–12, you can use reflection to find out whether a particular method exists on the platform. While this is a cleaner and safer implementation, reflection can make your code slower; therefore you should try to avoid using reflection where performance is critical. One possible approach is to call `Class.forName()` and `Class.getMethod()`to find out if a certain method exists in the static initialization block, and then only call `Method.invoke()` where performance is important.

Fragmentation

The high number of Android versions, 14 API levels so far, makes your target market quite fragmented, possibly leading to more and more code like the one in Listing 1–12. However, in practice, a few Android versions represent the majority of all the devices. As of December 2011, Android 2.x versions represent more than 95% of the devices connecting to Android Market. Even though Android 1.6 and earlier devices are still in operation, today it is quite reasonable not to spend additional resources to optimize for these platforms.

The number of available devices running Android is even greater, with currently close to 200 phones listed on `www.google.com/phone`, including 80 in the United States alone. While the listed devices are all phones or tablets, they still differ in many ways: screen resolutions, presence of physical keyboard, hardware-accelerated graphics, processors. Supporting the various configurations, or even only a subset, makes application development more challenging. Therefore it is important to have a good knowledge of the market you are targeting in order to focus your efforts on important features and optimizations.

NOTE: Not all existing Android devices are listed on `www.google.com/phone` as some countries are not listed yet, for example India and its dual-SIM Spice MI270 running Android 2.2.

Google TV devices (first released in 2010 by Logitech and Sony in the United States) are technically not so different from phones or tablets. However, the way people interact with these devices differs. When supporting these TV devices, one of your main challenges will be to understand how your application could be used on a TV. For example, applications can provide a more social experience on a TV: a game could offer a simultaneous multiplayer mode, a feature that would not make much sense on a phone.

Data Structures

As the various Fibonacci implementations demonstrated, good algorithms and good data structures are keys to a fast application. Android and Java define many data structures you should have good knowledge of to be able to quickly select the right ones for the right job. Consider choosing the appropriate data structures one of your highest priorities.

The most common data structures from the java.util package are shown in Figure 1.1.

To those data structures Android adds a few of its own, usually to solve or improve performance of common problems.

- LruCache
- SparseArray
- SparseBooleanArray
- SparseIntArray
- Pair

NOTE: Java also defines the `Arrays` and `Collections` classes. These two classes contain only static methods, which operate on arrays and collections respectively. For example, use `Arrays.sort` to sort an array and `Arrays.binarySearch` to search for a value in a sorted array.

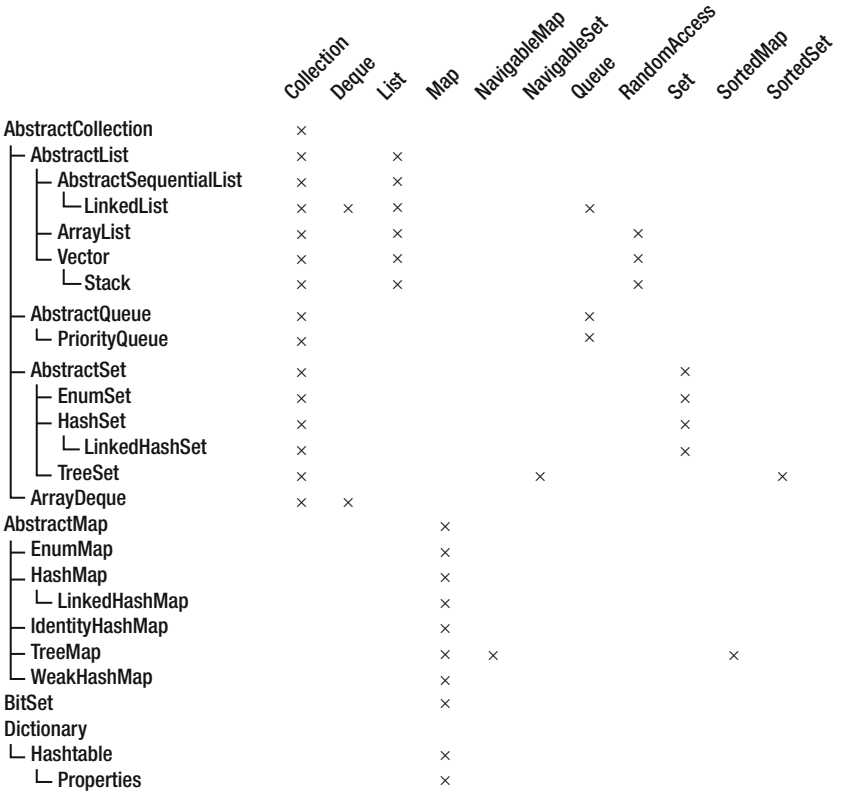

Figure 1–1. *Data structures in the java.util package*

While one of the Fibonacci implementations used a cache internally (based on a sparse array), that cache was only temporary and was becoming eligible for garbage collection immediately after the end result was computed. It is possible to also use an LruCache to save end results, as shown in Listing 1–13.

Listing 1–13. *Using an LruCache to Remember Fibonacci Terms*

```
int maxSize = 4 * 8 * 1024 * 1024; // 32 megabits
LruCache<Integer, BigInteger> cache = new LruCache<Integer, BigInteger> (maxSize) {
    protected int sizeOf (Integer key, BigInteger value) {
        return value.bitLength(); // close approximation of object's size, in bits
    }
};
…
int n = 100;
BigInteger fN = cache.get(n);
if (fN == null) {
    fN = Fibonacci. computeRecursivelyWithCache(n);
    cache.put(n, fN);
}
```

Whenever you need to select a data structure to solve a problem, you should be able to narrow down your choice to only a few classes since each class is usually optimized for

a specific purpose or provides a specific service. For example, choose `ArrayList` over `Vector` if you don't need the operations to be synchronized. Of course, you may always create your own data structure class, either from scratch (extending `Object`) or extending an existing one.

> **NOTE:** Can you explain why LruCache is not a good choice for computeRecursivelyWithCache's internal cache as seen in Listing 1–11?

If you use one of the data structures that rely on hashing (e.g. HashMap) and the keys are of a type you created, make sure you override the equal and hashCode methods. A poor implementation of hashCode can easily nullify the benefits of using hashing.

> **TIP:** Refer to `http://d.android.com/reference/java/lang/Object.html` for a good example of an implementation of `hashCode()`.

Even though it is often not natural for many embedded application developers, don't hesitate to consider converting one data structure into another in various parts of your application: in some cases, the performance increase can easily outweigh the conversion overhead as better algorithms can be applied. A common example is the conversion of a collection to an array, possibly sorted. Such a conversion would obviously require memory as a new object needs to be created. On memory-constrained devices, such allocation may not always be possible, resulting in an OutOfMemoryError exception. The Java Language Specification says two things:

- The class `Error` and its subclasses are exceptions from which ordinary programs are not ordinarily expected to recover.
- Sophisticated programs may wish to catch and attempt to recover from Error exceptions.

If your memory allocation is only part of an optimization and you, as a sophisticated application developer, can provide a fallback mechanism (for example, an algorithm, albeit slower, using the original data structure) then catching the `OutOfMemoryError` exception can make sense as it allows you to target more devices. Such optional optimizations make your code harder to maintain but give you a greater reach.

> **NOTE:** Counterintuitively, not all exceptions are subclasses of Exception. All exceptions are subclasses of `Throwable` (from which Exception and Error are the only direct subclasses).

In general, you should have very good knowledge of the java.util and android.util packages since they are the toolbox virtually all components rely on. Whenever a new Android version is released, you should pay special attention to the modifications in these packages (added classes, changed classes) and refer to the *API Differences*

Report on `http://d.android.com/sdk`. More data structures are discussed in java.util.concurrent, and they will be covered in Chapter 5.

Responsiveness

Performance is not only about raw speed. Your application will be perceived as being fast as long as it appears fast to the user, and to appear fast your application must be responsive. As an example, to appear faster, your application can defer allocations until objects are needed, a technique known as lazy initialization, and during the development process you most likely want to detect when slow code is executed in performance-sensitive calls.

The following classes are the cornerstones of most Android Java applications:

- `Application`
- `Activity`
- `Service`
- `ContentProvider`
- `BroadcastReceiver`
- `Fragment` (Android 3.0 and above)
- `View`

Of particular interest in these classes are all the `onSomething()` methods that are called from the main thread, such as `onStart()` and `onFocusChanged()`. The main thread, also referred to as the UI thread, is basically the thread your application runs in. It is possible, though not recommended, to run all your code in the main thread. The main thread is where, among other things:

- Key events are received (for example, `View.onKeyDown()` and `Activity.onKeyLongPress()`).
- Views are drawn (`View.onDraw()`).
- Lifecycle events occur (for example, `Activity.onCreate()`).

> **NOTE:** Many methods are called from the main thread by design. When you override a method, verify how it will be called. The Android documentation does not always specify whether a method is called from the main thread.

In general, the main thread keeps receiving notifications of what is happening, whether the events are generated from the system itself or from the user. Your application has only one main thread, and all events are therefore processed sequentially. That being said, it becomes easy now to see why responsiveness could be negatively affected: the first event in the queue has to be processed before the subsequent events can be

handled, one at a time. If the processing of an event takes too long to complete, then other events have to wait longer for their turn.

An easy example would be to call `computeRecursivelyWithCache` from the main thread. While it is reasonably fast for low values of n, it is becoming increasingly slower as n grows. For very large values of n you would most certainly be confronted with Android's infamous Application Not Responding (ANR) dialog. This dialog appears when Android detects your application is unresponsive, that is when Android detects an input event has not been processed within 5 seconds or a BroadcastReceiver hasn't finished executing within 10 seconds. When this happens, the user is given the option to simply wait or to "force close" the application (which could be the first step leading to your application being uninstalled).

It is important for you to optimize the startup sequence of all the activities, which consists of the following calls:

- `onCreate`
- `onStart`
- `onResume`

Of course, this sequence occurs when an activity is created, which may actually be more often than you think. When a configuration change occurs, your current activity is destroyed and a new instance is created, resulting in the following sequence of calls:

- `onPause`
- `onStop`
- `onDestroy`
- `onCreate`
- `onStart`
- `onResume`

The faster this sequence completes, the faster the user will be able to use your application again. One of the most common configuration changes is the orientation change, which signifies the device has been rotated.

> **NOTE:** Your application can specify which configuration changes each of its activities wants to handle itself with the activity element's android:configChanges attribute in its manifest. This would result in onConfigurationChanged() being called instead of having the activity destroyed.

Your activities' `onCreate()` methods will most likely contain a call to `setContentView` or any other method responsible for inflating resources. Because inflating resources is a relatively expensive operation, you can make the inflation faster by reducing the complexity of your layouts (the XML files that define what your application looks like). Steps to simplify the complexity of your layouts include:

- Use RelativeLayout instead of nested LinearLayouts to keep layouts as "flat" as possible. In addition to reducing the number of objects allocated, it will also make processing of events faster.
- Use ViewStub to defer creation of objects (see the section on lazy initialization).

> **NOTE:** Pay special attention to your layouts in ListView as there could be many items in the list. Use the SDK's layoutopt tool to analyze your layouts.

The basic rule is to keep anything that is done in the main thread as fast as possible in order to keep the application responsive. However, this often translates to doing as little as possible in the main thread. In most cases, you can achieve responsiveness simply by moving operations to another thread or deferring operations, two techniques that typically do not result in code that is much harder to maintain. Before moving a task to another thread, make sure you understand why the task is too slow. If this is due to a bad algorithm or bad implementation, you should fix it since moving the task to another thread would merely be like sweeping dust under a carpet.

Lazy initializations

Procrastination does have its merits after all. A common practice is to perform all initializations in a component's `onCreate()` method. While this would work, it means `onCreate()` takes longer to return. This is particularly important in your application's activities: since `onStart()` won't be called until after `onCreate()` returns (and similarly `onResume()` won't be called until after `onStart()` returns), any delay will cause the application to take longer to start, and the user may end up frustrated.

For example, Android uses the lazy initialization concept with android.view.ViewStub, which is used to lazily inflate resources at runtime. When the view stub is made visible, it is replaced by the matching inflated resources and becomes eligible for garbage collection.

Since memory allocations take time, waiting until an object is really needed to allocate it can be a good option. The benefits of lazily allocating an object are clear when an object is unlikely to be needed at all. An example of lazy initialization is shown in Listing 1–14, which is based on Listing 1–13. To avoid always having to check whether the object is null, consider the factory method pattern.

Listing 1–14. *Lazily Allocating the Cache*

```
int n = 100;
if (cache == null) {
    // createCache allocates the cache object, and may be called from many places
    cache = createCache();
}
BigInteger fN = cache.get(n);
if (fN == null) {
    fN = Fibonacci. computeRecursivelyWithCache(n);
    cache.put(n, fN);
}
```

Refer to Chapter 8 to learn how to use and android.view.ViewStub in an XML layout and how to lazily inflate a resource.

StrictMode

You should always assume the following two things when writing your application:

- The network is slow (and the server you are trying to connect to may not even be responding).
- File system access is slow.

As a consequence, you should always try not to perform any network or file system access in your application's main thread as slow operations may affect responsiveness. While your development environment may never experience any network issue or any file system performance problem, your users may not be as lucky as you are.

> **NOTE:** SD cards do not all have the same "speed". If your application depends heavily on the performance of external storage then you should make sure you test your application with various SD cards from different manufacturers.

Android provides a utility to help you detect such defects in your application. StrictMode is a tool that does its best to detect bad behavior. Typically, you would enable StrictMode when your application is starting, i.e when its `onCreate()` method is called, as shown in Listing 1–15.

Listing 1–15. *Enabling StrictMode in Your Application*

```
public class MyApplication extends Application {
    @Override
    public void onCreate ()
    {
        super.onCreate();

        StrictMode.setThreadPolicy(new StrictMode.ThreadPolicy.Builder()
        .detectCustomSlowCalls() // API level 11, to use with StrictMode.noteSlowCode
        .detectDiskReads()
        .detectDiskWrites()
        .detectNetwork()
        .penaltyLog()
        .penaltyFlashScreen() // API level 11
        .build());

        // not really performance-related, but if you use StrictMode you might as well
define a VM policy too
        StrictMode.setVmPolicy(new StrictMode.VmPolicy.Builder()
        .detectLeakedSqlLiteObjects()
        .detectLeakedClosableObjects() // API level 11
        .setClassInstanceLimit(Class.forName("com.apress.proandroid.SomeClass"), 100) //
API level 11
        .penaltyLog()
        .build());
```

```
    }
}
```

StrictMode was introduced in Android 2.3, with more features added in Android 3.0, so you should make sure you target the correct Android version and make sure your code is executed only on the appropriate platforms, as shown in Listing 1–12.

Noteworthy methods introduced in Android 3.0 include `detectCustomSlowCall()` and `noteSlowCall()`, both being used to detect slow, or potentially slow, code in your application. Listing 1–16 shows how to mark your code as potentially slow code.

Listing 1–16. *Marking Your Own Code as Potentially Slow*

```
public class Fibonacci {
    public static BigInteger computeRecursivelyWithCache(int n)
    {
        StrictMode.noteSlowCall("computeRecursivelyWithCache"); // message can be
anything
        SparseArray<BigInteger> cache = new SparseArray<BigInteger>();
        return computeRecursivelyWithCache(n, cache);
    }
    …
}
```

A call to `computeRecursivelyWithCache` from the main thread that takes too long to execute would result in the following log if the StrictMode Thread policy is configured to detect slow calls:

```
StrictMode policy violation; ~duration=21121 ms:
android.os.StrictMode$StrictModeCustomViolation: policy=31 violation=8 msg=
computeRecursivelyWithCache
```

Android provides some helper methods to make it easier to allow disk reads and writes from the main thread temporarily, as shown in Listing 1–17.

Listing 1–17. *Modifying the Thread Policy to Temporarily Allow Disk Reads*

```
    StrictMode.ThreadPolicy oldPolicy = StrictMode.allowThreadDiskReads();
    // read something from disk
    StrictMode.setThreadPolicy(oldPolicy);
```

There is no method for temporarily allowing network access, but there is really no reason to allow such access even temporarily in the main thread as there is no reasonable way to know whether the access will be fast. One could argue there is also no reasonable way to know the disk access will be fast, but that's another debate.

> **NOTE:** Enable StrictMode only during development, and remember to disable it when you deploy your application. This is always true, but even more true if you build the policies using the `detectAll()` methods as future versions of Android may detect more bad behaviors.

SQLite

Most applications won't be heavy users of SQLite, and therefore you very likely won't have to worry too much about performance when dealing with databases. However, you need to know about a few concepts in case you ever need to optimize your SQLite-related code in your Android application:

- SQLite statements
- Transactions
- Queries

> **NOTE:** This section is not intended to be a complete guide to SQLite but instead provides you with a few pointers to make sure you use databases efficiently. For a complete guide, refer to `www.sqlite.org` and the Android online documentation.

The optimizations covered in this section do not make the code harder to read and maintain, so you should make a habit of applying them.

SQLite Statements

At the origin, SQL statements are simple strings, for example:

- CREATE TABLE cheese (name TEXT, origin TEXT)
- INSERT INTO cheese VALUES ('Roquefort', 'Roquefort-sur-Soulzon')

The first statement would create a table named "cheese" with two columns named "name" and "origin". The second statement would insert a new row in the table. Because they are simply strings, the statements have to be interpreted, or compiled, before they can be executed. The compilation of the statements is performed internally when you call for example SQLiteDatabase.execSQL, as shown in Listing 1–18.

Listing 1–18. *Executing Simple SQLite Statements*

```
SQLiteDatabase db = SQLiteDatabase.create(null); // memory-backed database
db.execSQL("CREATE TABLE cheese (name TEXT, origin TEXT)");
db.execSQL("INSERT INTO cheese VALUES ('Roquefort', 'Roquefort-sur-Soulzon')");
db.close(); // remember to close database when you're done with it
```

> **NOTE:** Many SQLite-related methods can throw exceptions.

As it turns out, executing SQLite statements can take quite some time. In addition to the compilation, the statements themselves may need to be created. Because `String` is also immutable, this could lead to the same performance issue we had with the high number

of BigInteger objects being allocated in computeRecursivelyFasterUsingBigInteger. We are now going to focus on the performance of the insert statement. After all, a table should be created only once, but many rows could be added, modified, or deleted.

If we want to build a comprehensive database of cheeses (who wouldn't?), we would end up with many insert statements, as shown in Listing 1–19. For every insert statement, a String would be created and execSQL would be called, the parsing of the SQL statement being done internally for every cheese added to the database.

Listing 1–19. *Building a Comprehensive Cheeses Database*

```
public class Cheeses {
    private static final String[] sCheeseNames = {
        “Abbaye de Belloc”,
        “Abbaye du Mont des Cats”,
        …
        “Vieux Boulogne”
    };
    private static final String[] sCheeseOrigins = {
        “Notre-Dame de Belloc”,
        “Mont des Cats”,
        …
        “Boulogne-sur-Mer”
    };
    private final SQLiteDatabase db;
    public Cheeses () {
        db = SQLiteDatabase.create(null); // memory-backed database
        db.execSQL(“CREATE TABLE cheese (name TEXT, origin TEXT)”);
    }
    public void populateWithStringPlus () {
        int i = 0;
        for (String name : sCheeseNames) {
            String origin = sCheeseOrigins[i++];
            String sql = “INSERT INTO cheese VALUES(\”” + name + ”\”,\”” + origin +
”\”)”;
            db.execSQL(sql);
        }
    }
}
```

Adding 650 cheeses to the memory-backed database took 393 milliseconds on a Galaxy Tab 10.1, or 0.6 microsecond per row.

An obvious improvement is to make the creation of the sql string, the statement to execute, faster. Using the + operator to concatenate strings is not the most efficient method in this case, and it is possible to improve performance by either using a StringBuilder object or calling String.format. The two new methods are shown in Listing 1–20. As they simply optimize the building of the string to pass to execSQL, these two optimizations are not SQL-related per se.

Listing 1–20. *Faster Ways to Create the SQL Statement Strings*

```
    public void populateWithStringFormat () {
        int i = 0;
        for (String name : sCheeseNames) {
            String origin = sCheeseOrigins[i++];
```

```
            String sql = String.format("INSERT INTO cheese VALUES(\"%s\",\"%s\")", name,
origin);
            db.execSQL(sql);
        }
    }

    public void populateWithStringBuilder () {
        StringBuilder builder = new StringBuilder();
        builder.append("INSERT INTO cheese VALUES(\"");
        int resetLength = builder.length();
        int i = 0;
        for (String name : sCheeseNames) {
            String origin = sCheeseOrigins[i++];
            builder.setLength(resetLength); // reset position
            builder.append(name).append("\",\"").append(origin).append("\")"); // chain
calls
            db.execSQL(builder.toString());
        }
    }
```

The `String.format` version took 436 milliseconds to add the same number of cheeses, while the `StringBuilder` version returned in only 371 milliseconds. The `String.format` version is therefore slower than the original one, while the `StringBuilder` version is only marginally faster.

Even though these three methods differ in the way they create Strings, they all have in common the fact that they call `execSQL`, which still has to do the actual compilation (parsing) of the statement. Because all the statements are very similar (they only differ by the name and origin of the cheese), we can use `compileStatement` to compile the statement only once, outside the loop. This implementation is shown in Listing 1–21.

Listing 1–21. *Compilation of SQLite Statement*

```
    public void populateWithCompileStatement () {
        SQLiteStatement stmt = db.compileStatement("INSERT INTO cheese VALUES(?,?)");
        int i = 0;
        for (String name : sCheeseNames) {
            String origin = sCheeseOrigins[i++];
            stmt.clearBindings();
            stmt.bindString(1, name); // replace first question mark with name
            stmt. bindString(2, origin); // replace second question mark with origin
            stmt.executeInsert();
        }
    }
```

Because the compilation of the statement is done only once instead of 650 times and because the binding of the values is a more lightweight operation than the compilation, the performance of this method is significantly faster as it builds the database in only 268 milliseconds. It also has the advantage of making the code easier to read.

Android also provides additional APIs to insert values in a database using a `ContentValues` object, which basically contains the binding information between column names and values. The implementation, shown in Listing 1–22, is actually very close to `populateWithCompileStatement`, and the "`INSERT INTO cheese VALUES`" string does not even appear as this part of the insert statement is implied by the call to `db.insert()`.

However, the performance of this implementation is below what we achieved with `populateWithCompileStatement` since it takes 352 milliseconds to complete.

Listing 1–22. *Populating the Database Using ContentValues*

```
public void populateWithContentValues () {
    ContentValues values = new ContentValues();
    int i = 0;
    for (String name : sCheeseNames) {
        String origin = sCheeseOrigins[i++];
        values.clear();
        values.put("name", name);
        values.put("origin", origin);
        db.insert("cheese", null, values);
    }
}
```

The fastest implementation is also the most flexible one as it allows more options in the statement. For example, you could use "INSERT OR FAIL" or "INSERT OR IGNORE" instead of simply "INSERT".

> **NOTE:** Many changes were made in Android 3.0's android.database and android.database.sqlite packages. For instance, the `managedQuery`, `startManagingCursor`, and `stopManagingCursor` methods in the `Activity` class are all deprecated in favor of `CursorLoader`.

Android also defines a few classes that can improve performance. For example, you can use `DatabaseUtils.InsertHelper` to insert multiple rows in a database while compiling the SQL insert statement only once. It is currently implemented the same way we implemented `populateWithCompileStatement` although it does not offer the same flexibility as far as options are concerned (for example, `FAIL` or `ROLLBACK`).

Not necessarily related to performance, you may also use the static methods in the `DatabaseUtils` class to simplify your implementation.

Transactions

The examples above did not explicitly create any transaction, however one was automatically created for every insertion and committed immediately after each insertion. Creating a transaction explicitly allows for two basic things:

- Atomic commit
- Better performance

The first feature is important but not from a performance point of view. Atomic commit means either all or none of the modifications to the database occur. A transaction cannot be only partially committed. In our example, we can consider the insertion of all 650 cheeses as one transaction. Either we succeed at building the complete list of

cheeses or we don't, but we are not interested in a partial list. The implementation is shown in Listing 1–23.

Listing 1–23. *Insertion of All Cheeses in a Single Transaction*

```
    public void populateWithCompileStatementOneTransaction () {
        try {
            db.beginTransaction();
            SQLiteStatement stmt = db.compileStatement("INSERT INTO cheese
VALUES(?,?)");
            int i = 0;
            for (String name : sCheeseNames) {
                String origin = sCheeseOrigins[i++];
                stmt.clearBindings();
                stmt.bindString(1, name); // replace first question mark with name
                stmt. bindString(2, origin); // replace second question mark with origin
                stmt.executeInsert();
            }
            db.setTransactionSuccessful(); // remove that call and none of the changes
will be committed!
        } catch (Exception e) {
            // handle exception here
        } finally {
            db.endTransaction(); // this must be in the finally block
        }
    }
```

This new implementation took 166 milliseconds to complete. While this is quite an improvement (about 100 milliseconds faster), one could argue both implementations were probably acceptable for most applications as it is quite unusual to insert so many rows so quickly. Indeed, most applications would typically access rows only once in a while, possibly as a response to some user action. The most important point is that the database was memory-backed and not saved to persistent storage (SD card or internal Flash memory). When working with databases, a lot of time is spent on accessing persistent storage (read/write), which is much slower than accessing volatile memory. By creating the database in internal persistent storage, we can verify the effect of having a single transaction. The creation of the database in persistent storage is shown in Listing 1–24.

Listing 1–24. *Creation of Database On Storage*

```
    public Cheeses (String path) {
        // path could have been created with getDatabasePath("fromage.db")

        // you could also make sure the path exists with a call to mkdirs
        // File file = new File(path);
        // File parent = new File(file.getParent());
        // parent.mkdirs();

        db = SQLiteDatabase.openOrCreateDatabase(path, null);
        db.execSQL("CREATE TABLE cheese (name TEXT, origin TEXT)");
    }
```

When the database is on storage and not in memory, the call to `populateWithCompileStatement` takes almost 34 seconds to complete (52 milliseconds per row), while the call to `populateWithCompileStatementOneTransaction` takes less than

200 milliseconds. Needless to say, the one-transaction approach is a much better solution to our problem. These figures obviously depend on the type of storage being used. Storing the database on an external SD card would make it even slower and therefore would make the one-transaction approach even more appealing.

> **NOTE:** Make sure the parent directory exists when you create a database on storage. See `Context.getDatabasePath` and `File.mkdirs` for more information. For convenience, use SQLiteOpenHelper instead of creating databases manually.

Queries

The way to make queries faster is to also limit the access to the database, especially on storage. A database query simply returns a `Cursor` object, which can then be used to iterate through the results. Listing 1–25 shows two methods to iterate through all the rows. The first method creates a cursor that gets both columns in the database whereas the second method's cursor retrieves only the first column.

Listing 1–25. *Iterating Through All the Rows*

```
    public void iterateBothColumns () {
        Cursor c = db.query("cheese", null, null, null, null, null, null);
        if (c.moveToFirst()) {
            do {
            } while (c.moveToNext());
        }
        c.close(); // remember to close cursor when you are done (or else expect an
exception at some point)
    }

    public void iterateFirstColumn () {
        Cursor c = db.query("cheese", new String[]{"name"}, null, null, null, null,
null); // only difference
        if (c.moveToFirst()) {
            do {
            } while (c.moveToNext());
        }
        c.close();
    }
```

As expected, because it does not have to read data from the second column at all, the second method is faster: 23 milliseconds vs. 61 milliseconds (when using multiple transactions). Iterating through all the rows is even faster when all the rows are added as one transaction: 11 milliseconds for `iterateBothColumns` vs. 7 milliseconds for `iterateFirstColumn`. As you can see, you should only read the data you care about. Choosing the right parameters in calls to query can make a substantial difference in performance. You can reduce the database access even more if you only need up to a certain number of rows, and specify the limit parameter in the call to query.

TIP: Consider using the FTS (full-text search) extension to SQLite for more advanced search features (using indexing). Refer to `www.sqlite.org/fts3.html` for more information.

Summary

Years ago, Java had a bad reputation for performance, but today this is no longer true. The Dalvik virtual machine, including its Just-In-Time compiler, improves with every new release of Android. Your code can be compiled into native code that takes advantage of the latest CPU architectures without you having to recompile anything. While implementation is important, your highest priority should be to carefully select data structures and algorithms. Good algorithms can be pretty forgiving and perform quite well even without you optimizing anything. On the other hand, a bad algorithm almost always gives poor results, no matter how hard you work on its implementation.

Finally, never sacrifice responsiveness. It may make your application development a little bit more difficult, but the success of your application depends on it.