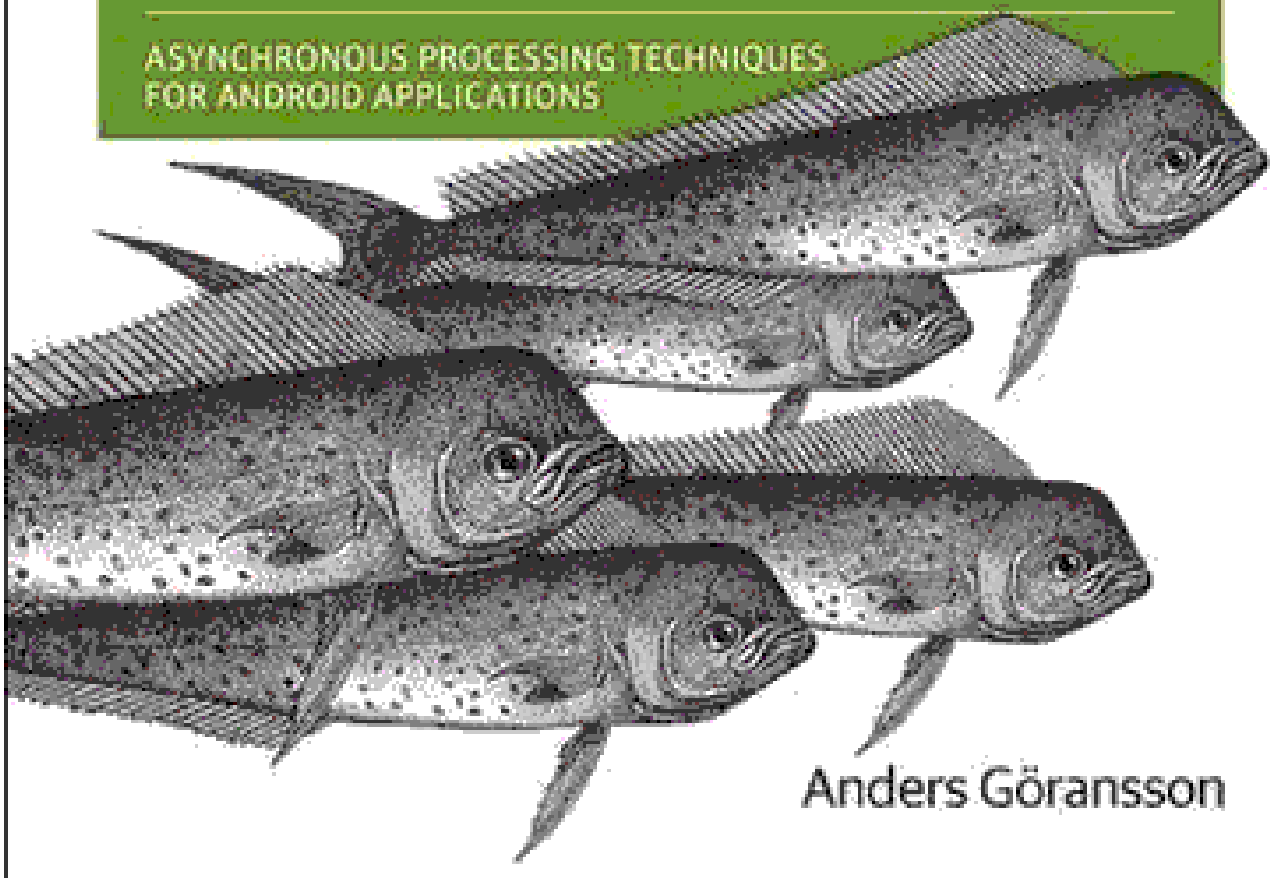
O'REILLY

Efficient Android Threading

ASYNCHRONOUS PROCESSING TECHNIQUES FOR ANDROID APPLICATIONS



Anders Göransson

Contents Overview

IntentService Versus Service

Summary

Efficient Android Threading

Anders Göransson

Editor

Rachel Roumeliotis

Editor

Andy Oram

Revision History	
2014-01-15	Early release revision 1

O'Reilly Media

1005 Gravenstein Highway North

Sebastopol, CA 95472

2014-01-14T10:02:14-08:00

About the Author

Anders Göransson is a Senior Consultant at a leading software technology company in Malmö, Sweden, where he specializes in the Android platform. Anders has been working with applications on Android ever since the first release of the Android platform as developer, architect, mentor and now teacher in threading and concurrency.

Part I. Introduction

Chapter 1. Android Components and the Need for Multi-Processing

Multicore processors are standard in the handheld devices of today and applications should utilize the opportunity to process data in parallel. With parallel data processing comes—in most cases—better performance and responsiveness as more processors can actively take part in running the application. But the other side of the coin is that execution becomes more complex, which can induce indeterministic behavior and timing related bugs that can be hard to reproduce. Every application developer need to manage a multi-threaded programming model and how to steer clear of the problem it induces.

This book focuses on application threading on the Android platform, and how multi-threading cooperates with the Android programming model. It provides an understanding of underlying execution mechanisms and how to utilize the asynchronous techniques efficiently. Multiple-CPU hardware holds many benefits for applications and developers: the application code can be executed on multiple CPUs concurrently to gain performance. Concurrent execution and responsiveness is a key factor for an application to succeed; a snappy application is crucial to preserve the user experience of an otherwise great application. But this book does not stop at making the application fast—it should be fast with minimal implementation effort, a good code design, and minimal risk for unexpected errors (not too uncommon in the world of concurrency...). Choosing the right asynchronous mechanism for the right use case is crucial to achieve simple and robust code execution with coherent code design.

But before we immerse ourselves in the world of threading, we will start with an introduction to the Android platform, the application architecture, and its execution. This chapter provides a baseline of knowledge required for an effective discussion of threading in the rest of the book, but a complete information on the Android platform can be found in the official documentation or in most of the numerous Android programming books on the market.

Android Software Stack

Applications run on top of a software stack that is based on a Linux kernel, native C/C++ libraries, and a runtime that executes the application code (Figure 1-1). The elements are:

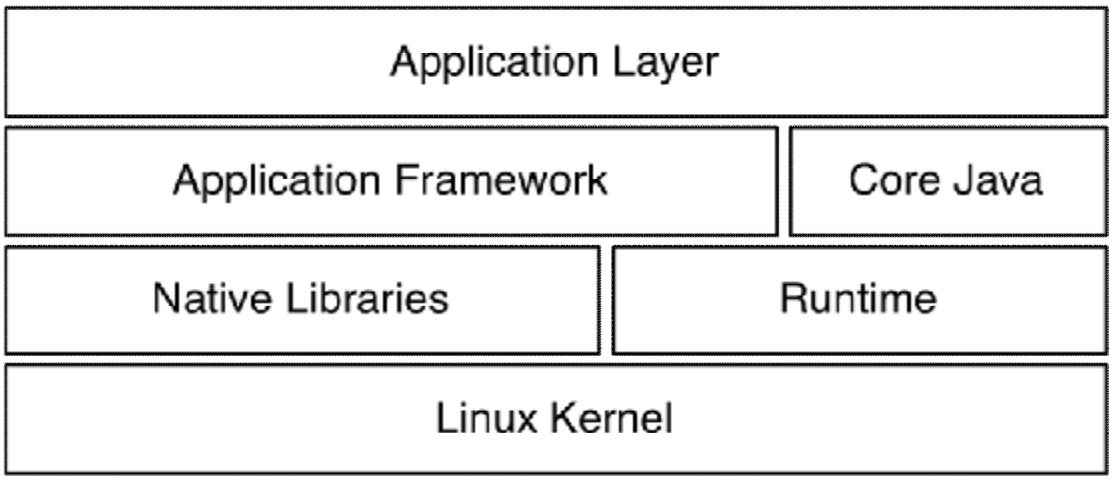


Figure 1-1. Android Software Stack

Applications

Top layer with Java applications, that use the Core Java library and the Android Application Framework classes.

Core Java

The core Java libraries used by applications and the Application Framework. It is not a fully compliant Java SE or ME implementation, but a subset of the retired Apache Harmony implementation, based on Java 5. It provides the fundamental Java threading mechanisms as the java.lang.Thread-class and java.util.concurrent-package.

Application framework

The Android classes that handle the window system, UI toolkit, resources, etc.—basically everything that is required to write an Android application in Java. The framework defines and manages the lifecycles of the Android components and their inter-communication. Furthermore, it defines a set of Android specific asynchronous mechanisms that applications can utilize to simplify the thread management: HandlerThread, AsyncTask, IntentService, AsyncQueryHandler and Loaders. All these mechanisms will be described in this book.

Native libraries

C/C++ libraries that handle graphics, media, database, fonts, OpenGL, etc. Java applications do not interact directly with the native libraries, because the Application framework provides Java wrappers for the native code.

Runtime

The Dalvik Virtual Machine, which provides a sandbox for each Java application and executes compiled Android application code, with an internal bytecode representation stored in Dalvik Executable (*.dex*) files. Every application runs in its own Dalvik runtime.

Linux kernel

: Underlying operating system that allows applications to use the hardware functions of the device, e.g., sound, network, camera, etc. It also manages processes and threads. A process is started for every application, and every process holds a Dalvik runtime with a running application. Within the process, multiple threads can execute the application code. The kernel splits the available CPU execution time for processes and their threads through *scheduling*.

Application Architecture

The cornerstones of an application are the Application object and the Android components: Activity, Service, BroadcastReceiver, and ContentProvider.

Application

The representation of an executing application in Java is the android.app.Application object, which is instantiated upon application start and destroyed when the application stops–i.e., an instance of the Application class lasts for the lifetime of the Linux process of the application. When the process is terminated and restarted, a new Application instance is created.

Components

The fundamental pieces of an Android application are the components managed by the Dalvik runtime: Activity, Service, BroadcastReceiver, and ContentProvider. The configuration and interaction of these components define the application's behavior. These entities have different responsibilities and lifecycles, but they all represent application entry points, where the application can be started. Once a component is started, it can trigger another component, and so on, throughout the application's lifecycle.

A component triggers the start of another component through an Intent, either within the application or between applications. The Intent specifies actions for the receiver to act upon—for instance, sending an email message or taking a photograph—and can also provide data from the sender to the receiver. An Intent can be explicit or implicit:

Explicit Intent

: Defines the fully classified name of the component, which is known within the application at compile time.

Implicit Intent

: A run-time binding to a component that has defined a set of characteristics in an IntentFilter. If the Intent matches the characteristics of a component's IntentFilter, the component can be started.

Components and their lifecycles are Android-specific terminologies, and they are not directly matched by the underlying Java objects. A Java object can outlive its component, and the runtime can contain multiple Java objects related to the same live component. This is a source of confusion, and as we will see in Chapter 6, it poses a risk for memory leaks.

An application implements a component by subclassing it, and all components in an application must be registered in the *AndroidManifest.xml* file.

Activity

An Activity is a screen—almost always taking up the device's full screen—shown to the user. It displays information, handles user input, etc. It contains all the UI components–buttons, texts, images, and so forth–shown on the screen and holds an object reference to the view hierarchy with all the View instances. Hence, the memory footprint of an Activity can grow large.

When the user navigates between screens, Activity instances form a stack. Navigation to a new screen pushes an Activity to the stack, whereas backward navigation causes a corresponding pop.

In Figure 1-2, the user has started an initial Activity A and navigated to B while A was finished, then on to C and D. A, B and C are full-screen, but D covers only a part of the display. Thus, A is destroyed, B is totally obscured, C is partly shown, and D is fully shown at the top of the stack. Hence, *D* has focus and receives user input. The position in the stack determines the state of each Activity:

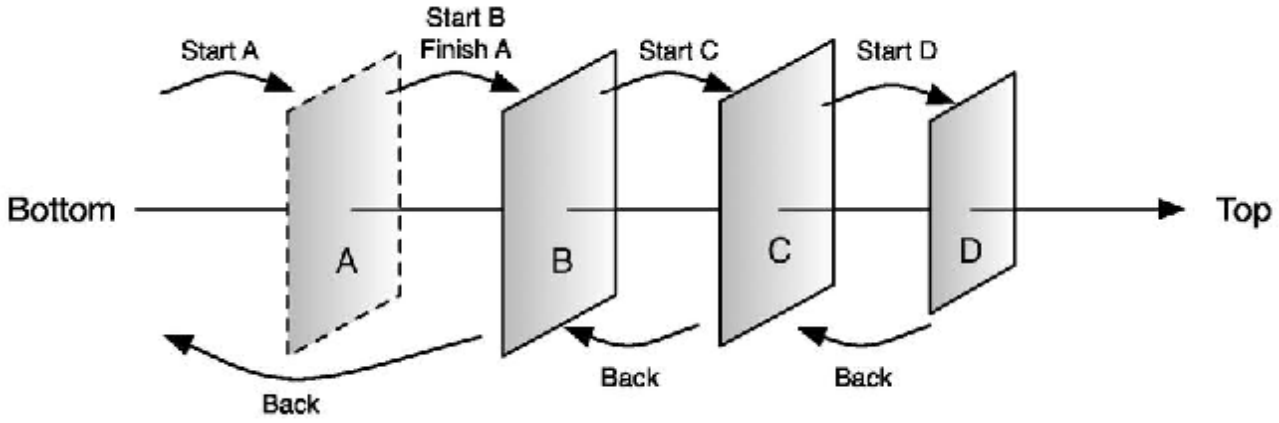


Figure 1-2. Activity stack

- Active in the foreground: D
- Paused and partly visible: C
- Stopped and invisible: B
- Inactive and destroyed: A

The state of an application's topmost Activity has an impact on the application's system priority—a.k.a. *process rank*—which in turn affects both the chances of terminating an application (Application Termination) and the scheduled execution time of the application threads (Chapter 3).

An Activity lifecycle ends either when the user navigates back—i.e. presses the back button—or when the Activity explicitly calls finish().

Service

A Service can execute invisibly in the background without direct user interaction. It is typically used to offload execution from other components, when the operations can outlive their lifetime. A Service can be executed in either a *started* or a *bound* mode:

Started Service
: The Service is started with a call to Context.startService(Intent) with an explicit or implicit Intent. It terminates when Context.stopService(Intent) is called.

Bound Service
: Multiple components can bind to a Service through Context.bindService(Intent, ServiceConnection, int) with explicit or implicit Intent parameters. After the binding, a component can interact with the Service through the ServiceConnection interface, and it unbinds from the Service through Context.unbindService(ServiceConnection). When the last component unbinds from the Service, it is destroyed.

ContentProvider

An application that wants to share substantial amounts of data within or between applications can utilize a ContentProvider. It can provide access to any data source, but it is most commonly used in collaboration with SQLite databases, which are always private to an application. With the help of a ContentProvider, an application can publish that data to applications that execute in remote processes.

BroadcastReceiver

This component has a very restricted function: it listens for Intents sent from within the application, remote applications, or the platform. It filters incoming Intents to determine which ones are sent to the BroadcastReceiver. A BroadcastReceiver should be registered dynamically when you want to start listening for Intents, and unregistered when it stops listening. If it is statically registered in the AndroidManifest, it listens for Intents while the application is installed. Thus, the BroadcastReceiver can start its associated application if an Intent matches the filter.

Application Execution

Android is a multi-user, multi-tasking system that can run multiple applications at the same time and let the user switch between applications without noticing a significant delay. The multi-tasking is handled by the Linux kernel, and application execution is based on Linux processes.

Linux Process

Linux assigns every user a unique user ID, basically a number tracked by the OS to keep the users apart. Every user has access to private resources protected by permissions, and no user (except *root*, the superuser, which does not concern us here) can access another user's private resources. Thus, sandboxes are created to isolate users. In Android, every application package has a unique user ID; i.e., an application in Android corresponds to a unique user in Linux and cannot access other applications' resources.

What Android adds to each process is a runtime execution environment, the Dalvik virtual machine, for each instance of an application. Figure 1-3 shows the relationship between the Linux process model, the VM, and the application.

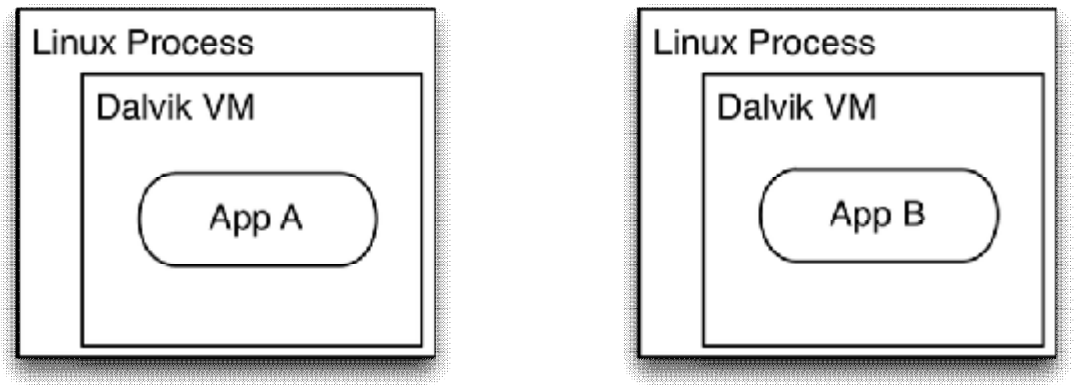


Figure 1-3. Applications execute in different processes and VMs

By default, applications and processes have a one-to-one relationship, but if required, it is possible for an application to run in several processes, or for several applications to run in the same process.

Lifecycle

The application lifecycle is encapsulated within its Linux process, which—in Java—maps to the android.app.Application class. The Application object for each app starts when the runtime calls its onCreate() method. Ideally, the app terminates with a call by the runtime to

its onTerminate(), but an application can not rely upon this. The underlying Linux process may have been killed before the runtime had a chance to call onTerminate(). The Application object is the first component to be instantiated in a process and the last to be destroyed.

Application Start

An application is started when one of its components is initiated for execution. Any component can be the entry point for the application and once the first component is triggered to start a Linux process is started—unless it is already running—which leads to the following start up sequence:

1. Start Linux process.
2. Create Dalvik Virtual Machine.
3. Create Application instance.
4. Create the entry point component for the application.

Setting up a new Linux process and the runtime is not an instantaneous operation. It can degrade performance and have a noticeable impact on the user experience. Thus, the system tries to shorten the startup time for Android applications by starting a special process called Zygote on system boot. Zygote has the entire set of core libraries preloaded. New application processes are forked from the Zygote process without copying the core libraries, which are shared across all applications.

Application Termination

A process is created at the start of the application and finishes when the system wants to free up resources. Because a user may request an application at any later time, the runtime avoids destroying all its resources until the number of live applications leads to an actual shortage of resources across the system. Hence, an application isn't automatically terminated even when all of its components have been destroyed.

When the system is low on resources, it's up to the runtime to decide which process should be killed. To make this decision, the system imposes a ranking on each process depending on the application's visibility and the components that are currently executing. In the following ranking, the bottom-ranked processes are forced to quit before the higher-ranked. The process ranks are (highest first):

Foreground
: Application has a visible component in front, Service is bound to an Activity in front in a remote process or BroadcastReceiver is running.

Visible
: Application has a visible component but is partly obscured.

Service

Service is executing in the background and is not tied to a visible component.

Background

A non-visible Activity. This is the process level that contains most applications.

Empty

A process without active components. Empty processes are kept around to improve startup times, but they are the first to be terminated when the system reclaims resources.

In practice, the ranking system ensures that no visible applications will be terminated by the platform when it runs out of resources.

Lifecycles of two interacting applications.

This example illustrates the lifecycles of two processes P1 and P2 that interacts in a typical way (Figure 1-4). P1 is a client application that invokes a Service in P2, a server application. The client process, P1, starts when it is triggered by a broadcasted Intent. At startup, the process starts both a BroadcastReceiver and the Application instance. After a while, an Activity is started, and during all of this time P1 has the highest possible process rank: Foreground.

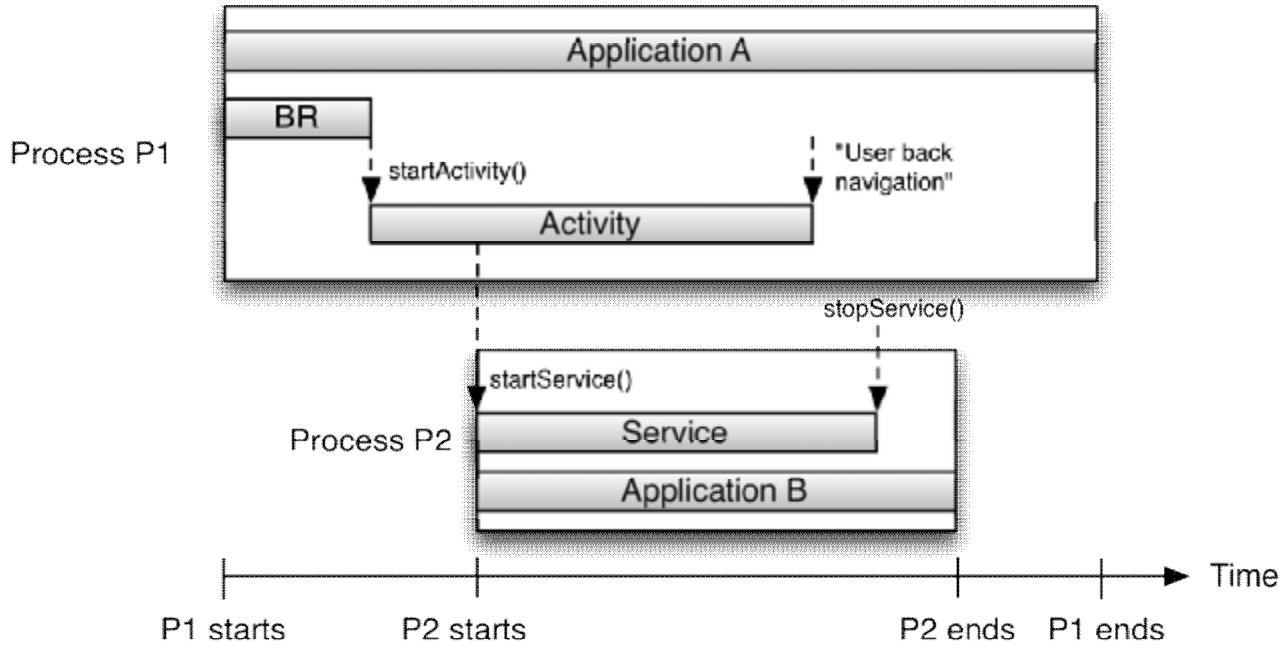


Figure 1-4. Client application starts Service in other process.

The Activity offloads work to a Service that runs in process P2, which starts the Service and the associated Application instance. Hence, the application has split the work into two different processes. The P1 Activity can terminate while the P2 Service keeps running.

Once all components have finished—the user has navigated back from the Activity in P1, and the Service in P2 is asked by some other process or the runtime to stop—both processes are ranked as Empty, making them plausible candidates for termination by the system when it requires resources.

A detailed list of the process ranks during the execution appears in Table 1-1.

Table 1-1. Process rank transitions.

Application state	P1 Process Rank	P2 Process Rank
P1 starts with BroadcastReceiver entry point	Foreground	N.A.
P1 starts Activity	Foreground	N.A.
P1 starts Service entry point in P2	Foreground	Foreground
P1 Activity is destroyed	Empty	Service
P2 Service is stopped	Empty	Empty

It should be noted that there is a difference between the actual application lifecycle—defined by the Linux process–and the perceived application lifecycle. The system can have multiple application processes running even while the user perceives them as terminated. The empty processes are lingering–if system resources permit it—to shorten the startup time on restarts.

Structuring Applications for Performance

Android devices are multiprocessor systems that can run multiple operations simultaneously, but it is up to each application to ensure that operations can be partitioned and executed concurrently to optimize application performance. If the application does not enable partitioned operations, but prefers to run everything as one long operation, it can exploit only one CPU, leading to suboptimal performance. Unpartitioned operations must run *synchronously*, whereas partitioned operations can run *asynchronously*. With asynchronous operations, the system can share the execution among multiple CPUs and therefore increase throughput.

An application with multiple independent tasks should be structured to utilize asynchronous execution. One approach is to split application execution into several processes, because those can run concurrently. However, every process allocates memory for its own substantial resources, so the execution of an application in multiple processes will use more memory than an application in one process. Furthermore, starting and communicating between processes is slow, and not an efficient way of achieving asynchronous execution. Multiple processes may still be a valid design, but that decision should be independent of performance. To achieve higher throughput and better performance, an application should utilize multiple threads within each process.

Creating Responsive Applications Through Threads

An application can utilize asynchronous execution on multiple CPU's with high throughput, but that doesn't guarantee a responsive application. Responsiveness is the way the user perceives the application during interaction: that the UI responds quickly to button clicks, smooth animations, etc. Basically, performance from the perspective of the user experienced is determined by how fast the application can update the UI components. The responsibility for updating the UI components lies with the *UI thread*,[1] which is the only thread the system allows to handle UI updates.

To make the application responsive, it should ensure that no long-running tasks are executed on the UI thread. If they do, all the other execution on that thread will be delayed. Typically, the first symptom of executing long running tasks on the UI thread is that the UI becomes unresponsive because it is not allowed to update the screen or accept user button presses properly. If the application delays the UI thread too long, typically 5-10 seconds, the runtime displays an "Application Not Responding" (ANR) dialog to the user, giving her an option to close the application. Clearly, you want to avoid this. In fact, the run-time prohibits certain time-consuming operations, such as network downloads, from running on the UI thread.

So long operations should be handled on a background thread. Long-running tasks typically include:

- Network communication
- Reading or writing to a file
- Creating, deleting, and updating elements in databases
- Reading or writing to SharedPreferences
- Image processing
- Text parsing

What is a Long Task?

There is no fixed definition of a long task or a clear indication when a task should execute on a background thread, but as soon as a user perceives a lagging UI—e.g., slow button feedback and stuttering animations—it is a signal that the task is too long to run on the UI thread. Typically, animations are a lot more sensitive to competing tasks on the UI thread than button clicks, because the human brain is a bit vague about when a screen touch actually happened. Hence, let us do some coarse reasoning with animations as the most demanding use case.

Animations are updated in an event loop where every event updates the animation with one frame, i.e., one drawing cycle. The more drawing cycles that can be executed per time frame, the better the animation is perceived. If the goal is to do 60 drawing cycles per second—a.k.a.

frames per second (fps)—every frame has to render within 16ms. If another task is running on the UI thread simultaneously, both the drawing cycle and the secondary task have to finish within 16ms to avoid a stuttering animation. Consequently, a task may require less than 16ms execution time and still be considered as long.

The example and calculations are coarse and meant as an indication of how an applications responsiveness can be affected not only by network connections that last for several seconds, but also tasks that at a first glance look harmless. Bottlenecks in your application can hide anywhere.

Threads in Android applications are as fundamental as any of the component building blocks. All Android components and system callbacks—unless denoted otherwise—runs on the UI thread, and should use background threads when executing longer tasks.

Summary

An Android application runs on top of a Linux OS in a Dalvik runtime, which is contained in a Linux process. Android applies a process ranking system that priorities the importance of each running application to ensure that it is only the least prioritized applications that are terminated. To increase performance, an application should split operations among several threads so that the code is executed concurrently. Every Linux process contains a specific thread that is responsible for updating the UI. All long operations should be kept off the UI thread and executed on other threads.

---

[1] Also known as the *main thread*, but throughout this book we stick to the convention of calling it the “UI thread.”

Part II. Fundamentals

This part of the book covers the building blocks for asynchronous processing, provided by Linux, Java, and Android. You should understand how these work, the trade-offs involved in using the various techniques, and what risks they introduce. That understanding will give you the basis for using the techniques in the next part of the book.

Chapter 2. Multithreading in Java

Every Android application should adhere to the multithreaded programming model built-in to the Java language. With multithreading comes improvements to performance and responsiveness that is required for a great user experience, but it is accompanied with increased complexities:

- Handling the concurrent programming model in Java.
- Keeping data consistency in a multithreaded environment.
- Setting up task execution strategies.

Thread Basics

Software programming is all about instructing the hardware to perform an action—e.g. show images on a monitor, store data on the file system, etc. The instructions are defined by the application code that the CPU processes in an ordered sequence, which is the high level definition of a *thread*. From an application perspective a thread is execution along a code path of Java statements that is executed sequentially. A code path that is sequentially executed on a thread is referred to as a *task* — a unit of work that coherently executes on one thread. A thread can either execute one or multiple tasks in sequence.

Execution

A thread in an Android application is represented by java.lang.Thread. It is the most basic execution environment in Android, that executes tasks when it starts and terminates when the task is finished or there are no more tasks to execute; the alive time of the thread is determined by the length of the task. Thread supports execution of tasks that are implementions of the java.lang.Runnable interface. An implementation defines the task in the run method:

```
private class MyTask implements Runnable {
    public void run() {
```

```
            int i = 0; // Stored on the thread local stack.
        }
}
```

All the local variables in the method calls from within a run() method–direct or indirect—will be stored on the local memory stack of the thread. The task's execution is started by instantiating and starting a Thread:

```
Thread myThread = new Thread(new MyTask());
myThread.start();
```

On the operating system level the thread has both an instruction- and a stack pointer. The instruction pointer references the next instruction to be processed and the stack pointer references a private memory area—not available to other threads—where thread local data is stored. Thread local data is typically variable literals that are defined in the Java methods of the application.

A CPU can process instructions from one thread at the time, but a system normally has multiple threads that require processing at the same time, e.g. a system with multiple simultaneously running applications. For the user to perceive that applications can run in parallel the CPU has to share its processing time between the application threads. The sharing of a CPU's processing time is handled by a *scheduler*. It determines what thread the CPU should process and for how long. There scheduling strategy can be implemented in various ways but it is mainly based on the thread *priority*: a high priority-thread gets the CPU allocation before a low-priority thread, which gives more execution time to high-priority threads. Thread priority in Java can be set between 1 (lowest) and 10 (highest), but—unless explicitly set—the normal priority is 5:

```
myThread.setPriority(8);
```

If, however, the scheduling is only priority-based the low-priority threads may not get enough processing time carry out the job it was intended for—known as *starvation*. Hence, schedulers also take the processing time of the threads into account when changing to a new thread. A thread change is known as *context switch*. A context switch starts by storing the state of the executing thread—so that the execution can be resumed at a later point—whereafter that thread has to wait. The scheduler then restores another waiting thread for processing.

Two concurrently running threads—executed by a single processor—are split into execution intervals as Figure 2-1 shows.

```
Thread T1 = new Thread(new MyTask());
```

```
T1.start();
Thread T2 = new Thread(new MyTask());
T2.start();
```

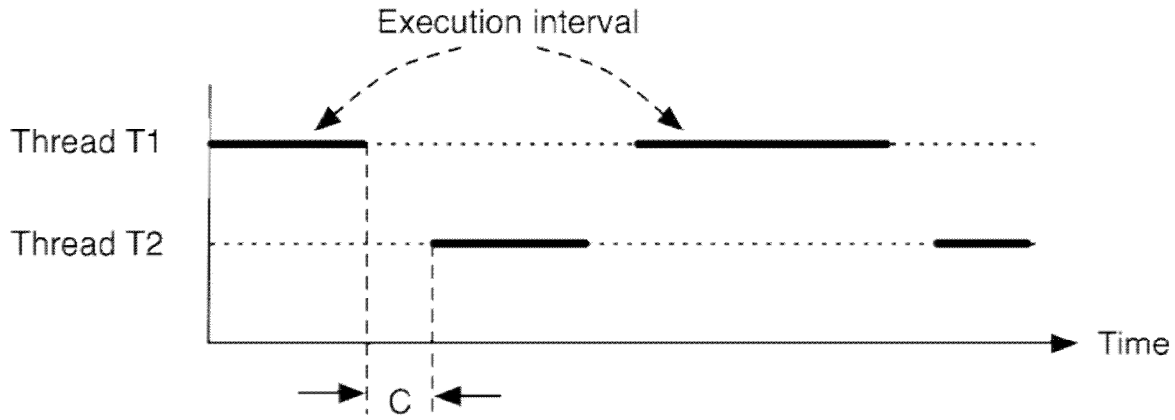


Figure 2-1. Two threads executing on one CPU

Every scheduling point includes a context switch, where the operating system has to use the CPU to carry out the switch. One such context switch is noted as C in the figure.

Single-Threaded Application

Each application has at least one thread that defines the code path of execution. If no more threads are created, all of the code will be processed along the same code path, and an instruction has to wait for all preceding intructions to finish before it can be processed.

The single-threaded execution is a simple programming model with deterministic execution order, but most often it is not a sufficient approach as instructions may be postponed for a long time by preceding instructions, although the latter instruction is not depending on the preceeding instructions. For example, a user who presses a button on the device should get immediate visual feedback that the button is pressed, but in a single-threaded environment, the UI event can be delayed until preceding instructions have finished execution, which degrades both performance and responsiveness. To solve this, an application needs to split the execution into multiple code paths, i.e., threads.

Multi-Threaded Application

With multiple threads, the application code can be split into several code paths, so that operations are perceived to be executing concurrently. If the number of executing threads exceeds the number of processors, true concurrency can not be achieved, but the scheduler switches rapidly between threads to be processed, so that every code path is split into execution intervals that are processed in a sequence.

Multi-threading is a must-have, but the improved performance comes at a cost—increased complexity, increased memory consumption, non-deterministic order of execution—that the application has to manage.

Increased Resource Consumption

Threads come with an overhead in terms of memory and processor usage. Each thread allocates a private memory area that is mainly used to store method local variables and parameters during the execution of the method. The private memory area is allocated when the thread is created and deallocated once the thread terminates, i.e. as long as the thread is active it holds on to system resources—even if it is idle or blocked.

The processor entails overhead for the setup and teardown of threads, and to store and restore threads in context switches. The more threads that executes, the more context switches may occur and deteriorate performance.

Increased Complexity

Analyzing the execution of a single-threaded application is relatively simply, because the order of execution is known. In multi-threaded applications, it is a lot more difficult to analyze how the program is executed and in which order the code is processed. The execution order is indeterministic between threads as it is not known beforehand how the scheduler will allocate execution time to the threads. Hence, multiple threads introduce uncertainty into execution. Not only does this indeterminacy make it much harder to debug errors in the code, but the necessity of coordinating thread poses a risk for introducing new errors.

Data Inconsistency

A new set of problems arise in multi-threaded programs when the order of resource access is nondeterministic. If two or more threads uses a shared resource it is not known in which order the threads will reach and process the resource. For example, if two threads t1 and t2 tries to modify the member variable sharedResource, the access order is indeterminate—it may either be incremented or decremented first.

```
public class RaceCondition {

    int sharedResource = 0;

    public void startTwoThreads() {
        Thread t1 = new Thread(new Runnable() {
            @Override
            public void run() {
```

```
                sharedResource++;
            }
        });
        t1.start();

        Thread t2 = new Thread(new Runnable() {
            @Override
            public void run() {
                sharedResource--;
            }
        });
        t2.start();
    }
}
```

The sharedResource is exposed to a *race condition*, that can occur because the ordering of the code execution can differ from every execution; it can not be guaranteed that thread t1 always comes before thread t2. In this case it is not only the ordering that is troublesome, but both the incrementer and decrementer operations are multiple byte code instructions—read, modify and write. Context switches can occur between the byte code instructions, leaving the end result of sharedResource dependent on the order of execution; it can be either 0, -1 or 1. The first result occurs if the first thread manages to write the value before the second thread reads it, whereas the two latter results occur if both threads first read the initial value 0 making the last written value determine the end result.

Because context switches can occur while one thread is executing a part of the code that should not be interrupted, it is necessary to create *atomic regions* of code instructions that always are executed in sequence without interleaving of other threads. If a thread executes in an atomic region other threads will be *blocked* until no other thread executes in the atomic region. Hence, an atomic region in Java is said to be *mutually exclusive* as it only allows access to one thread. An atomic region can be created in various ways, see Intrinsic Lock and Java Monitor, but the most fundamental synchronization mechanism is the *synchronized* keyword.

```
synchronized (this) {
    sharedResource++;
}
```

If every access to the shared resource is synchronized, the data can not be inconsistent in spite of multithreaded access. Many of the threading mechanisms discussed in this book were designed to reduce the risk of such errors.

Thread Safety

Giving multiple threads access to the same object is a great way for threads to communicate quickly—one thread writes another thread reads—but it threatens correctness. Multiple threads can execute the same instance of an object simultaneously, causing concurrent access to the state in shared memory. That imposes a risk of threads either seeing the value of the state before it has been updated, or corrupting the value.

Thread safety is achieved when an object always maintains the correct state when accessed by multiple threads. This is achieved by synchronizing the object's state, so that the access to the state is controlled. Synchronization should be applied to code that reads or writes any variable that otherwise could be accessed by one thread while being changed by another thread. Such areas of code are called *critical sections* and must be executed atomically, i.e., by only by one thread at the time. Synchronization is achieved by using a locking mechanism that checks whether there currently is a thread executing in a critical section. If so, all the other threads trying to enter the critical section will block until the thread is finished with executing the critical section.

Note

If a shared resource is accessible from multiple threads and the state is mutable—i.e. the value can be changed during the lifetime of the resource—every access to the resource needs to be guarded by the same lock.

In short, locks guarantee atomic execution of the regions they lock. Locking mechanisms in Android include:

- Object intrinsic lock
    - The synchronized keyword
- Explicit locks
    - java.util.concurrent.locks.ReentrantLock
    - java.util.concurrent.locks.ReentrantReadWriteLock

Intrinsic Lock and Java Monitor

The synchronized keyword operates on the intrinsic lock that is implicitly available in every Java object. The intrinsic lock is mutually exclusive, meaning that thread execution in the critical section is exclusive to one thread. Other threads that try to access a critical region—while being occupied—are blocked and can not continue executing until the lock has been released. The intrinsic lock acts as a *monitor*, see Figure 2-2. The Java monitor can be modeled with three states:

Blocked

Threads that are suspended, while they wait for the monitor to released by another thread.

Executing

The one and only thread that owns the monitor, and is currently running the code in the critical section.

Waiting

Threads that voluntarely has given up ownership of the monitor before it has reached the end of the critical section. The threads are waiting to be signalled before they can take ownership again.

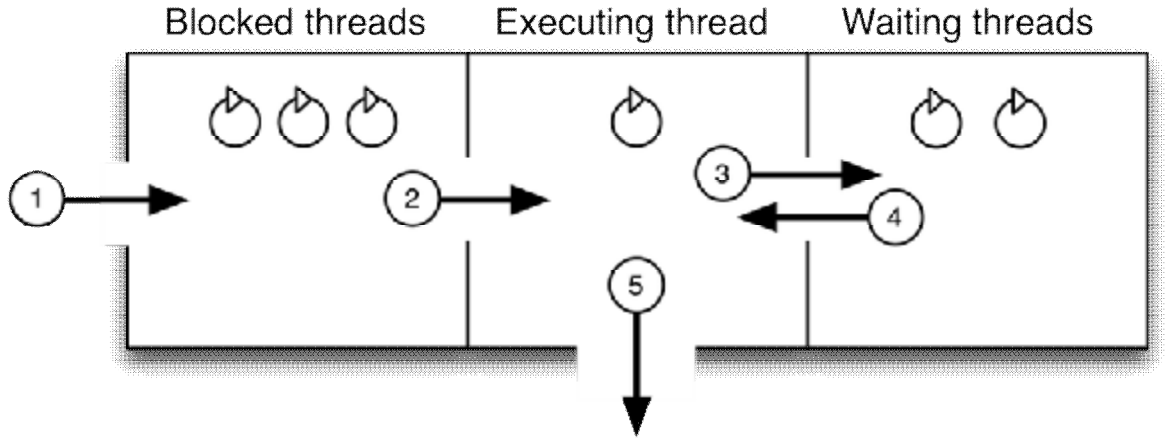


Figure 2-2. Java monitor

Threads transition between the monitor states when it reaches and executes a code block protected by the intrinsic lock:

1. *Enter the monitor*. A thread tries to access a section that is guarded by an intrinsic lock. It enters the monitor, but if the lock is already acquired by another thread it is suspended until the
2. *Acquire the lock*. If there is no other thread that owns the monitor a blocked thread can take ownership and execute in the critical section. If there are more than one blocked thread the scheduler selects thread to execute. There is no FIFO ordering among the blocked threads, i.e. the first thread to enter the monitor is not necessarily the first one to be selected for execution.
3. *Release the lock and wait*. The thread suspends itself through Object.wait() as it wants to wait for a condition to be fulfilled before it continues to execute.
4. *Acquire the lock after signal*. Waiting threads are signalled from another thread through Object.notify() or Object.notifyAll() and can take ownership of the monitor again, if selected by the scheduler. However, the waiting threads have no precedence over potentially blocked threads that also want to own the monitor.
5. *Release the lock and exit the monitor*. At the end of a critical section the thread exits the monitor and leaves room for another thread to take ownership.

The transitions map to a synchronized code block accordingly:

synchronized (this) { ❶// Execute code ❷wait();❸// Execute code ❹}❺

Synchronize Access to Shared Resources

A shared mutable state that can be accessed and altered by multiple threads requires a synchronization strategy to keep the data consistent during the concurrent execution. The strategy involves choosing the right kind of lock for the situation and the setting the scope for the critical section.

Using the Intrinsic Lock

An intrinsic lock can guard a shared mutable state in different ways. The synchronization strategy is about the intrinsic lock itself can differ depending on how the keyword synchronized is used:

Method-level that operates on the intrinsic lock of the enclosing object instance. +

```
synchronized void changeState() {
    sharedResource++;
}
```

Block-level that operates on the intrinsic lock of the enclosing object instance. +

```
void changeState() {
    synchronized(this) {
        sharedResource++;
    }
}
```

Block-level with other objects intrinsic lock. +

```
private final Object mLock = new Object();

void changeState() {
    synchronized(mLock) {
            sharedResource++;
    }
}
```

Method-level that operates on the intrinsic lock of the enclosing class instance. +

```
synchronized static void changeState() {
    staticSharedResource++;
}
```

Block-level that operates on the intrinsic lock of the enclosing class instance. +

```
static void changeState() {
    synchronized(this.getClass()) {
        staticSharedResource++;
    }
}
```

A reference to the this object in block-level synchronization uses the same intrinsic lock as method-level synchronization. But by using this syntax, you can control the precise block of code covered by the critical section, and therefore reduce it to cover only the code that actually concerns the state you want to protect. It's bad practice to create larger atomic areas than necessary, because that may block other threads when not necessary, leading to slower execution across the application.

Synchronizing on other objects' intrinsic lock enables the use of multiple locks within a class. An application should strive to protect each independent state with a lock of its own. Hence, if a class has more than one independent state, performance is improved by using several locks.

Warning

The synchronized keyword can operate in different intrinsic locks. Keep in mind that synchronization on static methods operates on the intrinsic lock of the class object and not the instance object.

Using Explicit Locking Mechanisms

If a more advanced locking strategy is needed, ReentrantLock or ReentrantReadWriteLock classes can be used instead of the synchronized keyword. Critical sections are protected by explicitly locking and unlocking regions in the code:

```
int sharedResource;
private ReentrantLock mLock = new ReentrantLock();

public void changeState() {
        mLock.lock();
        try {
```

```
                sharedResource++;
        }
        finally {
                mLock.unlock();
        }
}
```

The synchronized keyword and ReentrantLock have the same semantics: they both block all threads trying to execute a critical section if another thread already has entered that region. This is a defensive strategy that assumes that all concurrent accesses are problematic, but it is not harmful for multiple threads to read a shared variable simultaneously. Hence, synchronized and ReentrantLock can be overprotective.

The ReentrantReadWriteLock lets reading threads execute concurrently but still blocks readers versus writers and writers versus other writers.

```
int sharedResource;
private ReentrantReadWriteLock mLock = new ReentrantReadWriteLock();

public void changeState() {
        mLock.writeLock().lock();
        try {
                sharedResource++;
        }
        finally {
                mLock.writeLock().unlock();
        }
}

public int readState() {
        mLock.readLock().lock();
        try {
                return sharedResource;
        }
        finally {
                mLock.readLock().unlock();
        }
}
```

The ReentrantReadWriteLock is relatively complex, which leads to a performance penalty as the evaluation to determine whether a thread should be allowed to execute or be blocked

requires is longer than with synchronized and ReentrantLock. Hence, there is a trade off between the performance gain from letting multiple threads read shared resources simultaneously and the performance loss from evaluation complexity. The typically good use case for ReentrantReadWriteLock is when there are many reading threads and few writing threads.

Example: Consumer and Producer

A common use case with collaborating threads is the consumer-producer pattern—i.e. one thread that produces data and one thread that consumes the data. The threads collaborate through a list that is shared between the threads. When the list is not full the producer thread adds items to the list, whereas the consumer thread removes items while the list is not empty. If the list is full the producing thread should block, and if the list is empty the consuming thread is blocked.

The ConsumerProducer class contains a shared resource LinkedList and two methods: produce() to add items and consume to remove items.

```
public class ConsumerProducer {

    private LinkedList<Integer> list = new LinkedList<Integer>();
    private final int LIMIT = 10;
    private Object lock = new Object();

    public void produce() throws InterruptedException {

        int value = 0;

        while (true) {
            synchronized (lock) {
                while(list.size() == LIMIT) {
                        lock.wait();
                }
                list.add(value++);
                lock.notify();
            }
        }
    }

    public void consume() throws InterruptedException {
```

```
            while (true) {
                synchronized (lock) {
                    while(list.size() == 0) {
                        lock.wait();
                    }
                    int value = list.removeFirst();
                    lock.notify();
                }
            }
        }
}
```

Both produce and consume uses the same intrinsic lock for guarding the shared list. Threads that tries to access the list are blocked as long another thread owns the monitor, but producing threads give up execution—i.e. wait() if the list is full and consuming threads if the list is empty.

When items are either added or removed from the list, the monitor is signalled—i.e. notify() is called—so that waiting threads can execute again. The consumer threads signal producer threads and vice versa.

Two threads that executes the producing and consuming operations.

```
final ConsumerProducer cp = new ConsumerProducer();

Thread t1 = new Thread(new Runnable() {
    @Override
    public void run() {
        try {
            cp.produce();
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
    }
}).start();

Thread t2 = new Thread(new Runnable() {
    @Override
    public void run() {
        try {
            cp.consume();
```

```
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
        }
}).start();
```

Task Execution Strategies

To make sure that multiple threads are used properly to create responsive applications, applications should be designed with thread creation and task execution in mind. Two suboptimal designs and extremes are:

One thread for all tasks

: All tasks are executed on the same thread. The result is often an unresponsive application that fails to use available processors.

One thread per task

: Tasks are always executed on a new thread that is started and terminated for every task. If the tasks are frequently created, with short lifetimes, the overhead of thread creation and teardown can degrade performance.

Although these extremes should be avoided, they both represent variants of sequential and concurrent execution at the extreme.

Sequential execution

: Tasks are executed in a sequence that requires one task to finish before the next is processed. Thus, the execution interval of the tasks does not overlap. Advantages of this design are:

  - It is inherently thread-safe.
  - Can be executed on one thread, which consumes less memory than multiple threads.

Disadvantages include:

- Low throughput.
- The start of each task's execution depends on previously executed tasks. The start may either be delayed or possibly not executed at all.

  Concurrent execution

Task are executed in parallel and interleaved. The advantage is better CPU-utilization, whereas the disadvantage is that the design is not inherently thread-safe, so synchronization may be required.

An effective multi-threaded design utilizes execution environments with both sequential and concurrent execution; the choice depends on the tasks. Isolated and independent tasks can execute concurrently to increase throughput, but tasks that require an ordering or share a common resource without synchronization should be executed sequentially.

Concurrent Execution Design

Concurrent execution can be implemented in many ways, so design has to consider how to manage the number of executing threads and their relationships. Basic principles include:

- Favor reuse of threads, instead of always creating new threads, so that the number of times the creation and teardown of resources can be reduced.
- Do not use more threads than required. The more threads that are used, the more memory and processor time is consumed.

Summary

Android applications should be multithreaded to improve performance on both single- and multi-processor platforms. Threads can share execution on a single processor or utilize true concurrency when multiple processors are available. The increased performance comes at a cost of increased complexity and a responsibility to guard resources shared among threads, to preserve data consistency.

Chapter 3. Threads on Android

Every Android application is started with numerous threads that are bundled with the Linux process and the Dalvik VM to manage the internal execution of the application. But the application is exposed to system threads like the UI and Binder threads, and creates background threads of its own. It is now time to get in under the hood of threading on the Android platform:

- Differences and similarities between UI, Binder and background threads.
- The Linux thread coupling.
- How thread scheduling is affected by the application process rank.
- Observe running Linux threads.

Android Application Threads

All application threads are based on the native pthreads in Linux with a Thread representation in Java, but the platform still assigns special properties to threads that make them differ. From an application perspective the thread types are UI, binder, and background threads.

UI Thread

The UI thread is started when the application is started and stays alive during the lifetime of the Linux process. The UI thread is the main thread of the application, used for executing Android components and updating the UI elements on the screen. If the platform detects that UI updates are attempted from any other thread, it will promptly notify the application by throwing a CalledFromWrongThreadException. This harsh platform behavior is required because the Android UI Toolkit is not thread-safe, so the runtime allows access to the UI elements from one thread only.

Note

UI elements in Android are often defined as instance fields of activities, so they constitute a part of the object's state. However, access to those elements doesn't require synchronization because UI elements can be accessed only from the UI thread. In other words, the run-time enforces a single-threaded environment for UI elements, so they are not susceptible to concurrency problems.

The UI thread is a sequential event handler thread that can execute events sent from any other thread in the platform. The events are handled serially and are queued if the UI thread

is occupied with processing a previous event. Any event can be posted to the UI thread, but if events are sent to the thread that does not explicitly require the UI thread for execution, the UI-critical events may have to wait in the queue before being processed and responsiveness is decreased.

Android Message Passing describes event handling in detail.

Binder threads

Binder threads are used for communicating between threads in different processes. Each process maintains a set of threads, called a *thread pool*, that are never terminated or recreated, but can run tasks at the request of another thread in the process. These threads handle incoming requests from other processes: e.g., system services, intents, content providers, and services. When needed, a new binder thread will be created to handle the incoming request. In most cases, an application does not have to be concerned about binder threads because the platform normally transforms the requests to use the UI thread first. The exception is when the application offers a Service that can be bound from other processes via an AIDL interface. Binder threads are discussed more thoroughly in Chapter 5.

Background threads

All the threads that an application explicitly creates itself are background threads. This means that they have no predefined purpose, but are empty execution environments waiting to execute any task. The background threads are descendants of the UI thread, so they inherit the UI thread properties, such as its priority. By default, a newly created process doesn't contain any background threads. It is always up to the application itself to create them when needed.

Note

The second part of this book, Part III, is all about creating background threads.

A background thread created here in the application would look like this in the *ps -t* output. The last field is the name. The thread name, by default, ends with the number assigned by the run-time to the thread as its ID.

u0_a72 4283 4257 320304 34540 ffffffff 00000000 S Thread-12412

In the application, the use cases for the UI thread and worker threads are quite different, but in Linux they are both plain native threads and are handled equal. The constraints on the UI thread—that it should handle all UI updates—are enforced by the Window Manager in the Application Framework and not by Linux.

The Linux Process and Threads

The execution of long operations on background threads on Android can be handled in many ways, but no matter how the application implements the execution mechanism, the threads in the end are always the same on the operating system level. The Android platform is a Linux-based OS and every application is executed as a Linux application in the OS. Both the Android application and its threads adhere to the Linux execution environment. As we will see, knowledge of the Linux environment helps us not only to grasp and investigate the application execution, but also improve our applications' performance.

Each running application has an underlying Linux process, forked from the pre-started Zygote-process. A process has the following properties:

User id (UID)
: A process has a unique user identifier that represents a user on a Linux system. Linux is a multi-user system and on Android each application represents a user in this system. When the application is installed it is assigned a user id.

Process identifier (PID)
: A unique identifier for the process.

Parent process identifier (PPID)
: After system start-up, each process is created from another process. The running system forms a tree hierarchy of the running processes. Hence, each application process has a parent process. For Android, the parent of all processes is the Zygote.

Stack
: Local function pointers and variables.

Heap
: The address space allocated to a process. The address space is kept private to a process and can't be accessed by other processes.

Finding application process information

The process information of a running application is retrieved by the *ps* (process status) command, which you can call from the ADB shell. The Android *ps* command retrieves process information just as it would on any Linux distribution. However, the set of options is differentthan the traditional Linux version of *ps*:

-t
: Shows thread information in the processes.

-x
: Shows time spent in user code (utime) and system code (stime) in "jiffies", which typically is units of 10ms.

-p

Shows priorities.

-P

Shows scheduling policy, normally indicating whether the application is executing in the foreground or background.

-c

Shows which CPU is executing the process.

name|pid

Filter on the application's name or process ID. Only the last defined value is used.

You can also filter through the *grep* command. For instance, executing the *ps* command for a com.wifill.eat application [2] process would look like this:

```
$ adb shell ps | grep com.wifill.eat
USER      PID   PPID  VSIZE  RSS     WCHAN    PS  NAME
u0_a72    4257  144   320304 34540 ffffffff 00000000 S com.wifill.eat
```

From this output, we can extract the following interesting properties of the com.wifill.eat application:

- UID: u0_a72
- PID: 4257
- PPID: 144 (Process number of the parent, which in the case of an Android application always is the Zygote)

Another way of retrieving process and thread information is with DDMS[3] in the Android tools.

All the threads that an application creates and starts are native Linux threads, a.k.a. *pthreads* because they were defined in a POSIX standard. The threads belong to the process where they where created, and the parent of each thread is the process. Threads and processes are very much alike. The difference lies in the sharing of resources. The process is an isolated execution of a program in a sandboxed environment compared to other processes, whereas the threads share the resources within a process. An important distinction between process and threads is that processes don't share address space with each other, but threads share the address space within a process. This memory sharing makes it a lot faster to communicate between threads than between processes, which requires remote procedure calls that take up more overhead. Thread communication is covered in Chapter 4 and process communication in Chapter 5.

When a process starts, a single thread is automatically created for that process. A process always contains at least one thread to handle the execution. In Android, the thread created automatically in a process is the one we've already seen as the UI thread.

Let's take a look at the threads created in a process for an Android application with the package name com.wifill.eat:

```
$ adb shell ps -t | grep u0_a72
USER      PID   PPID  VSIZE  RSS     WCHAN    PS  NAME
u0_a72    4257  144   320304 34540 ffffffff 00000000 S com.wifill.eat
u0_a72    4259  4257  320304 34540 ffffffff 00000000 S GC
u0_a72    4262  4257  320304 34540 ffffffff 00000000 S Signal Catcher
u0_a72    4263  4257  320304 34540 ffffffff 00000000 S JDWP
u0_a72    4264  4257  320304 34540 ffffffff 00000000 S Compiler
u0_a72    4265  4257  320304 34540 ffffffff 00000000 S ReferenceQueueDemon
u0_a72    4266  4257  320304 34540 ffffffff 00000000 S FinalizerDaemon
u0_a72    4267  4257  320304 34540 ffffffff 00000000 S FinalizerWatchdogDaemon
u0_a72    4268  4257  320304 34540 ffffffff 00000000 S Binder_1
u0_a72    4269  4257  320304 34540 ffffffff 00000000 S Binder_2
```

On application start, no less than ten threads are started in our process. The first thread, named com.wifill.eat, is started by default when the application launches. Hence, that is the UI thread of the application. All the other threads are spawned from the UI thread, which is seen on the parent process id (PPID) of the other threads. Their PPID corresponds to the process id (PID) of the UI thread.

Most of the threads are Dalvik internal threads and we don't have to worry about them from an application perspective. They handle garbage collection, debug connections, finalizers, etc. Let's focus on the threads we need to care about:

```
u0_a72    4257  144   320304 34540 ffffffff 00000000 S com.wifill.eat
u0_a72    4268  4257  320304 34540 ffffffff 00000000 S Binder_1
u0_a72    4269  4257  320304 34540 ffffffff 00000000 S Binder_2
```

Scheduling

Linux treats threads and not processes as the fundamental unit for execution. Hence, scheduling on Android concerns threads and not processes. Scheduling allocates execution time for threads on a processor. Each thread that is executing in an application is competing with all the other threads in the application for execution time. The scheduler decides which thread should execute and for how long it should be allowed to execute before the scheduler picks a new thread to execute, i.e., a context switch occurs. A scheduler picks the next thread to execute depending on some thread properties, which are different for each scheduler type, although the thread priority is the most important one. In Android the application threads are scheduled by the standard scheduler in the Linux kernel and not by the Dalvik Virtual Machine. In practice, this means that the threads in our application are not only competing

directly with each other for execution time but also against all threads in all the other applications.

The Linux kernel scheduler is known as a “completely fair scheduler” (CFS). It is “fair” in the sense that it tries to balance the execution of tasks not only based on the priority of the thread but also by tracking the amount of execution time[4] that has been given to a thread. If a thread has previously had low access to the processor it will be allowed to execute before higher prioritized threads. If a thread doesn’t use the allocated time to execute, the CFS will ensure that the priority is lowered so that it will get less execution time in the future.

The platform has mainly two ways of affecting the thread scheduling:

- Priority: change the Linux thread priority.
- Control group: change the Android specific control group.

Priority

All threads in an application are associated with a priority that indicates to the scheduler which thread it should allocate execution time to on every context switch. On Linux the thread priority is called *niceness* or *nice value*, which basically is an indication of how nice a certain thread should behave towards other threads. Hence, a low niceness corresponds to a high priority. In Android, a Linux thread has niceness values in the range of -20 (most prioritized) to 19 (least prioritized), with a default niceness of 0. A thread inherits its priority from the thread where it is started and keeps it unless it’s explicitly changed by the application.

An application can change priority of threads from two classes:

java.lang.Thread

setPriority(int priority);

Sets the new priority based on the Java priority values from 0 (least prioritized) to 10 (most prioritized).

android.os.Process

Process.setThreadPriority(int priority); // Calling thread.
Process.setThreadPriority(int threadId, int priority); // Thread with specific id.

Sets the new priority using Linux niceness, i.e. -20 to 19.

Java priority vs Linux niceness

Thread.setPriority() is platform-independent. It represents an abstraction of the underlying platform-specific thread priorities. The abstract priority values correspond to Linux niceness values according to the following table.

Thread.setPriority(int)	Linux niceness
1 (Thread.MIN_PRIORITY)	19
2	16
3	13
4	10
5 (Thread.NORM_PRIORITY)	0
6	-2
7	-4
8	-5
9	-6
10 (Thread.MAX_PRIORITY)	-8

The mapping of Java priorities is an implementation detail and may vary depending on platform version. The niceness mapping values in the table are from Jelly Bean.

Control groups

Android not only relies the regular Linux CFS for thread scheduling but also imposes thread control groups[5] on all threads. The thread control groups are Linux containers that are used to manage the allocation of processor time all threads in one container. All threads created in an application belong to one of the thread control groups.

Android defines multiple control groups, but the most important ones for applications are the Foreground Group and Background Group. The Android platform defines execution constraints so that the threads in the different control groups are allocated different amount of execution time on the processor. Threads in the Foreground Group are allocated a lot more execution time than threads in the Background Group[6] and Android utilizes this to ensure that visible applications on the screen get more processor allocation than applications that are not visible on the screen. The visibility on the screen relates to the process levels (see The Linux Process and Threads), as illustrated in Figure 3-1.

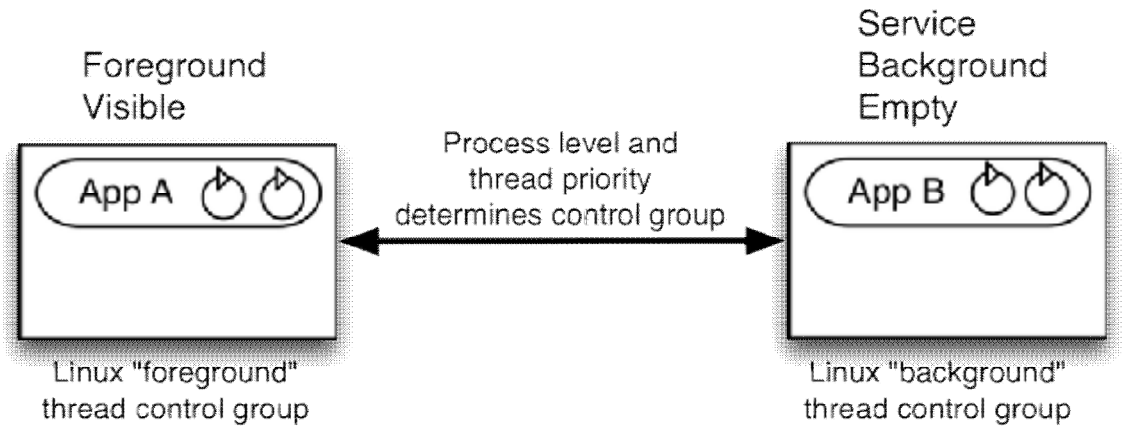


Figure 3-1. Thread control groups

If an application runs at the Foreground or Visible process level, the threads created by that application will belong to the Foreground Group and receive most of total processing time, while the remaining time will be divided among the threads in the other applications. A *ps* command issued on a foreground thread shows something like this. Note that the fg group appears.

```
$ adb shell ps -P | grep u0_a72
u0_a72       4257   144     320304 34504 fg    ffffffff 00000000 S com.wifill.eat
```

If the user moves an application to the background, such as by pressing the Home button, all the threads in that application will switch control group to the Background Group and will get less processor allocation. *ps* shows something like, with the application in the bg group:

```
$ adb shell ps -P | grep u0_a72
u0_a72       4257   144     318700 32164 bg    ffffffff 00000000 S com.wifill.eat
```

When the application is seen on the screen again, the threads moves back to the Foreground Group. This moving of threads between control groups is done as soon as application become visible or invisible. The use of control groups increases the performance of the applications seen on the screen and reduces the risk of background applications disturbing the applications actually seen by the user, hence improving the user experience.

Although the control groups ensure that background applications interfere as little as possible with the performance of visible applications, an application can still create a lot of threads that compete with the UI thread. The threads created by the application by default, have the same priority and control group membership as the UI thread so they compete on equal terms for processor allocation. Hence, an application that creates a lot of background threads may reduce the performance of the UI thread even though the intention is the

opposite. To solve this, it's possible to decouple background threads from the control group where the application threads execute by default. This decoupling is ensured by setting the priority of the background threads low enough so that they always belong to the Background Group even though the application is visible.

Tip

Lowering the priority of a thread with Process.setThreadPriority(Process.THREAD_PRIORITY_BACKGROUND) will not only reduce the priority but also ensure that this thread is decoupled from the process level of the application and always put in the Background Group.

Summary

All thread types in Android—UI, binder, and background thread—are Linux Posix threads. An application has a UI thread and binder threads when the process is started, but the application has to create background threads itself. All Android components execute on the UI thread by default, but long-running tasks should be executed on background threads to avoid slow UI rendering and the risk for ANRs. The UI thread is the most important thread, but it gets no special scheduling advantage compared to the other threads—the scheduler is unaware of which thread is the UI thread. Instead, it is up to the application to not let the background threads interfere more than necessary with the UI thread–typically by lowering the priority and letting the less important background threads execute in the background control group.

---

[2] I have used the string EAT to create a namespace for applications in this book. The string is the abbreviation for the book's title.

[3] Dalvik Debug Monitor Service

[4] The CFS calls this the *virtual runtime* of a thread.

[5] *cgroups* in Linux.

[6] The threads in the Background Group can't get more than ~5-10% execution time all together.

Chapter 4. Thread Communication

In multithreaded appplications, tasks can run in parallel and collaborate to produce a result. Hence, threads have to be able to communicate to enable true asynchronous processing. In Android, the importance of thread communication is emphasized in the platform-specific Handler/Looper mechanism that is the focus in this chapter, together with the traditional Java techniques. The chapter covers:

- Passing data through a one-way data pipe
- Shared memory communication
- Implementing a consumer-producer pattern with BlockingQueue
- Operations on message queues
- Sending tasks back to the UI Thread

Pipes

Pipes are a part of the java.io package. That is, they are general Java functionality and not Android specific. A pipe provides a way for two threads, within the same process, to connect and establish a one-way data channel. A producer thread writes data to the pipe, whereas a consumer thread reads data from the pipe.

Note

The Java pipe is comparable to the Unix and Linux pipe operator (the | shell character) that is used to redirect the output from one command to the input for another command. The pipe operator works across processes in Linux, but Java pipes work across threads in the Virtual Machine, i.e., within a process.

The pipe itself is a circular buffer allocated in memory, available only to the two connected threads. No other threads can access the data. Hence, thread safety—discussed in Thread Safety—is ensured. The pipe is also one-directional, permitting just one thread to write and the other to read (Figure 4-1).

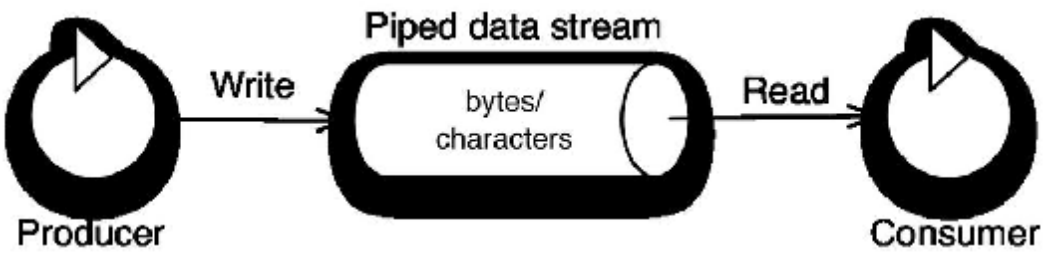


Figure 4-1. Thread communication with pipes.

Pipes are typically used when you have two long running tasks and one has to offload data to another continuously. Pipes make it easy to decouple tasks to several threads, instead of having only one thread handle many tasks. When one task has produced a result on a thread, it pipes the result on to the next thread that processes the data further. The gain comes from clean code separation and concurrent execution. Pipes can be used not only between worker threads, but also to offload work from the UI thread, which you want to keep light to preserve a responsive user experience.

A pipe can transfer either binary or character data. Binary data transfer is represented by PipedOutputStream (in the producer) and PipedInputStream (in the consumer), whereas character data transfer is represented by PipedWriter (in the producer) and PipedReader (in the consumer). Apart from the data transfer type, the two pipes have similar functionality. The lifetime of the pipe starts when either the writer or the reader thread establishes a connection, and it ends when the connection is closed.

Basic Pipe Use

The fundamental pipe life cycle can be summarized in three steps: setup, data transfer (which can be repeated as long as the two threads want to exchange data), and disconnection. The following examples are created with PipedWriter/PipedReader, but the same steps work with PipedOutputStream/PipedInputStream.

1. Set up the connection
2. PipedReader r = new PipedReader();
3. PipedWriter w = new PipedWriter();
   w.connect(r);

   Here, the connection is established by the writer connecting to the reader. The connection could just as well be established from the reader. Several constructors also implicitly set up a pipe. The default buffer size is 1024 but is configurable from the consumer side of the pipe, as shown later.

   ```
   int BUFFER_SIZE_IN_CHARS = 1024 * 4;
   PipedReader r = new PipedReader(BUFFER_SIZE_IN_CHARS);
   PipedWriter w = new PipedWriter(r);
   ```

4. Pass the reader to a processing thread
5. Thread t = new MyReaderThread(r);
   t.start();

   After the reader thread starts, it is ready to receive data from the writer.

6. Transfer data

```
// Producer thread: Write single character or array of characters
w.write('A');

// Consumer thread: Read the data
int result = r.read();
```

Communication adheres to the consumer-producer pattern with a blocking mechanism. If the pipe is full, the write() method will block until enough data has been read, and consequently removed from the pipe, to leave room for the data the writer is trying to add. The read() method blocks whenever there is no data to read from the pipe. It's worth noticing that the read() method returns the character as an integer value to ensure that enough space is available to handle various encoding with different sizes. You can cast the integer value back to a character.

In practice, a better approach would look like this:

```
// Producer thread: Flush the pipe after a write.
w.write('A');
w.flush();

// Consumer thread: Read the data in a loop.
int i;
while((i = reader.read()) != -1){
      char c = (char) i;
      // Handle received data
}
```

Calling flush() after a write to the pipe notifies the consumer thread that new data is available. This is useful from a performance perspective, because when the buffer is empty, the PipedReader uses a blocking call to wait() with one second timeout. Hence, if the flush() call is omitted, the consumer thread may delay the reading of data up to one second. By calling flush(), the producer cuts short the wait in the consumer thread and allows data processing to continue immediately.

11. Close the connection

When the communication phase is finished, the pipe should be disconnected.

```
// Producer thread: Close the writer.
w.close();
```

```
// Consumer thread: Close the reader.
r.close();
```

If the writer and reader are connected, it's enough to close only one of them. If the writer is closed, the pipe is disconnected but the data in the buffer can still be read. If the reader is closed, the buffer is cleared.

Text processing on a worker thread

This next example illustrates how pipes can process text that a user enters in an EditText. To keep the UI thread responsive, each character entered by the user is passed to a worker thread, which presumably handles some time-consuming processing. For the sake of brevity, exception handling and proper thread cancellation have been omitted from the example.

```
public class PipeExampleActivity extends Activity {

    private EditText editText;

    PipedReader r;
    PipedWriter w;

    private Thread workerThread;

    public void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);

        r = new PipedReader();
        w = new PipedWriter();
        w.connect(r);

        setContentView(R.layout.activity_pipes);
        editText = (EditText) findViewById(R.id.edit_text);
        editText.addTextChangedListener(new TextWatcher() {
            @Override
            public void onTextChanged(CharSequence charSequence, int start, int before,
int count) {
                // Only handle addition of characters
                if(count > before) {
                        // Write the last entered character to the pipe
                    w.write(charSequence.subSequence(before, count).toString());
                }
```

```
            }
        });

        workerThread = new Thread(new TextHandlerTask(r));
        workerThread.start();
    }

    @Override
    protected void onDestroy() {
        super.onDestroy();
        r.close();
        w.close();
    }

    private static class TextHandlerTask implements Runnable {

        private final PipedReader reader;

        public TextHandlerTask(PipedReader reader){
            this.reader = reader;
        }

        @Override
        public void run() {
            while(true){
                // Block background thread and wait for data to process.
                while((i = reader.read()) != -1){
                    char c = (char) i;
                    // Add text processing logic here
                }
            }
        }
    }
}
```

When the PipeExampleActivity is created, it will show an EditText box, which has a listener (TextWatcher) for changes in the content. Whenever a new character is added in the EditText, the character will be written to the pipe and read in the TextHandlerTask. The consumer task is an infinite loop that reads a character from the pipe as soon as there is anything to read. The inner while-loop will block when calling read() if the pipe is empty.

Warning

Be careful when involving the UI thread with pipes, due to the possible blocking of calls if the pipe is either full (producer blocks on its write() call) or empty (consumer blocks on its read() call).

Shared memory

Shared memory (using the memory area known in programming as the *heap*) is a common way to pass information between threads. All threads in an application can access the same address space within the process. Hence, if one thread writes a value on a variable in the shared memory, it can be read by all the other threads, as shown in Figure 4-2.

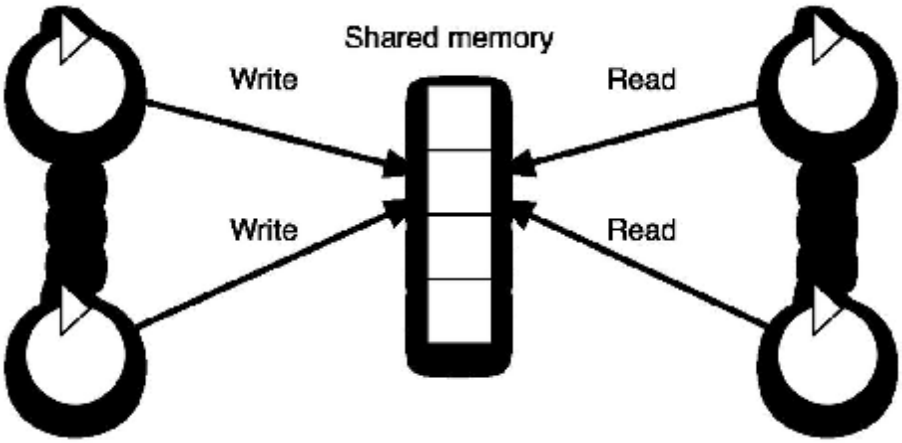


Figure 4-2. Thread communication with shared memory.

If a thread stores data as a local variable, no other thread can see it. By storing it in shared memory, it can use the variables for communication and share work with other threads. Objects are stored in the shared memory if they are scoped as one of the following:

- Instance member variables
- Class member variables
- Objects declared in methods

The reference of an object is stored locally on the thread's stack, but the object itself is stored in shared memory. The object is accessible from multiple threads only if the method publishes the reference outside the method scope, e.g. by passing the reference to another object's method. Threads communicate through shared memory by defining instance and class fields that are accessible from multiple threads.

Signalling

While threads are communicating through the state variables on the shared memory, they could poll the state value to fetch changes to the state. But a more efficient mechanism is the Java library's built-in signalling mechanism that lets a thread notify other threads of changes in the state. The signalling mechanism varies depending on the synchronization type (see Table 4-1).

Table 4-1. Thread signalling.

	synchronized	ReentrantLock	ReentrantReadWriteLock
Blocking call, waiting for a state.	Object.wait() Object.wait(timeout)	Condition.await() Condition.await(timeout)	Condition.await() Condition.await(timeout)
Signal blocked threads	Object.notify() Object.notifyAll()	Condition.signal() Condition.signalAll()	Condition.signal() Condition.signalAll()

When a thread cannot continue execution until another thread reaches a specific state, it calls wait()/wait(timeout) or the equivalents await()/await(timeout), depending on the synchronization used. The timeout parameters indicate how long the calling thread should wait before continuing the execution.

When another thread has changed the state it, signals the change with notify()/notifyAll() or the equivalents signal()/signalAll(). Upon a signal, the waiting thread continues execution. The calls thus support two different design patterns that use conditions: the notify() or signal() version wake one thread, chosen at random, whereas the notifyAll() or signalAll() version wakes all threads waiting on the signal.

Because multiple threads could receive the signal and one could enter the critical section before the others wake, receiving the signal does not guarantee that the correct state is achieved. A waiting thread should apply a design pattern where it checks that the wanted condition is fulfilled before executing further. For example, if the shared state is protected with synchronization on the intrinsic lock, check the condition before calling wait():

```
synchronized(this) {
        while(isConditionFulfilled == false) {
                wait();
        }
        // When the execution reaches this point,
        // the state is correct.
}
```

This pattern checks whether the condition predicate is fulfilled. If not, the thread blocks by calling wait(). When another thread notifies on the monitor and the waiting thread wakes up, it checks again whether the condition has been fulfilled and, if not, it blocks again, waiting for a new signal.

Warning

A very common Android use case is to create a worker thread from the UI thread and let the worker thread produce a result to be used by some UI element, so the UI thread should wait for the result. Thread signalling should not be used to the UI thread. The UI thread should never block to wait for another thread to create a state.

BlockingQueue

Thread signalling is a low-level, highly configurable mechanism that can be adapted to fit many use cases, but it may also be considered as the most error prone technique. Hence, the Java platform builds high-level abstractions upon the thread signaling mechanism to solve one-directional hand-off of arbitrary objects between threads. The abstraction is often called "solving the producer-consumer synchronization problem." The problem consists of use cases where there can be threads producing content (producer threads) and threads consuming content (consumer threads). The producers hand off messages for the consumers to process. The intermediator between the threads is a queue with blocking behavior, i.e., java.util.concurrent.BlockingQueue (see Figure 4-3).

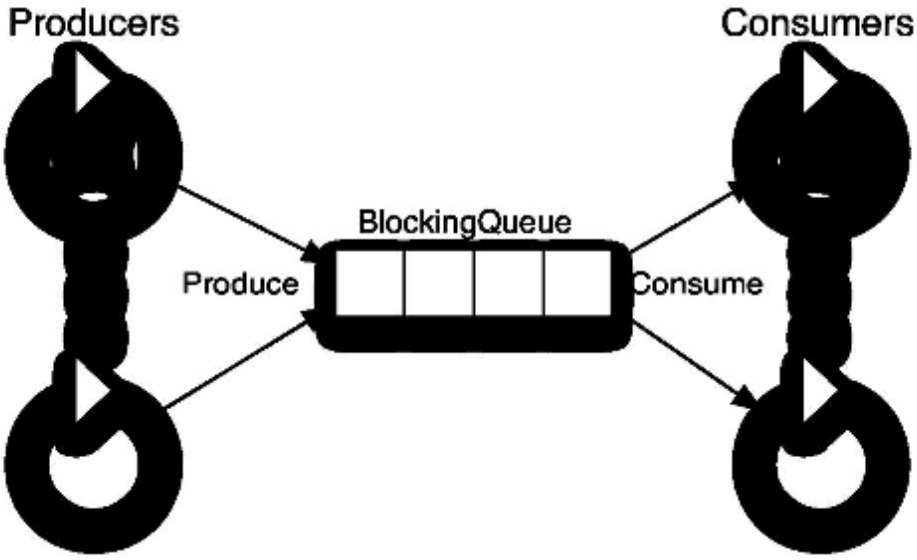


Figure 4-3. Thread communication with BlockingQueue.

The BlockingQueue acts as the coordinator between the producer and consumer threads, wrapping a list implementation together with thread signalling. The list contains a configurable number of elements that the producing threads fill with arbitrary data messages. On the other side, the consumer threads extract the messages in the order that they were enqueued and process them. Coordination between the producers and consumers is necessary if they get out of sync, for example if the producers hands off more messages than the consumers can handle. So BlockingQueue uses thread conditions to ensure that producers cannot enqueue new messages if the BlockingQueue list is full, and that consumers know when there are messages to fetch. Synchronization between the threads can be achieved with thread signalling as Example: Consumer and Producer shows. But the BlockingQueue both blocks threads and signals the important state changes—i.e. the list is not full and list is not empty.

The consumer-producer pattern implemented with the LinkedBlockingQueue-implementation is easily implemented by adding messages to the queue with put(), and removing them with take(), where put() blocks the caller if the queue is full, and take() blocks the caller if the queue is empty.

```
public class ConsumerProducer {

    private final int LIMIT = 10;
    private BlockingQueue<Integer> blockingQueue = new
LinkedBlockingQueue<Integer>(LIMIT);

    public void produce() throws InterruptedException {
        int value = 0;

        while (true) {
            blockingQueue.put(value++);
        }
    }

    public void consume() throws InterruptedException {

        while (true) {
            int value = blockingQueue.take();
        }
    }
}
```

Android Message Passing

So far, the thread communication options discussed have been regular Java, available in any Java application. The mechanisms—pipes, shared memory and blocking queues—apply to Android applications, but impose problems for the UI thread because of their tendency to block. The UI thread responsiveness is at risk when using mechanisms with blocking behavior, because that may occasionally hang the thread.

The most common thread communication use case in Android is between the UI thread and worker threads. Hence, the Android platform defines its own message passing mechanism for communication between threads. The UI thread can offload long tasks by sending data messages be processed on background threads. The message passing mechanism is a non-blocking consumer-producer pattern, where neither the producer thread nor the consumer thread will block during the message hand-off.

The message handling mechanism is fundamental in the Android platform and the API is located in the android.os package, with a set of classes shown in Figure 4-4 that implement the functionality.

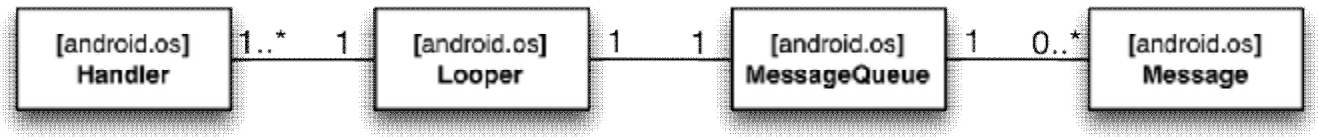


Figure 4-4. API overview.

android.os.Looper
: A message dispatcher associated with the one and only consumer thread.

android.os.Handler
: Consumer thread message processor, and the interface for a producer thread to insert messages into the queue. A Looper can have many associated Handlers, but they all insert messages into the same queue.

android.os.MessageQueue
: Unbounded linked list of messages to be processed on the consumer thread. Every Looper—and Thread—has at most one MessageQueue.

android.os.Message
: Message to be executed on the consumer thread.

Messages are inserted by producer threads and processed by the consumer thread, as illustrated in Figure 4-5.

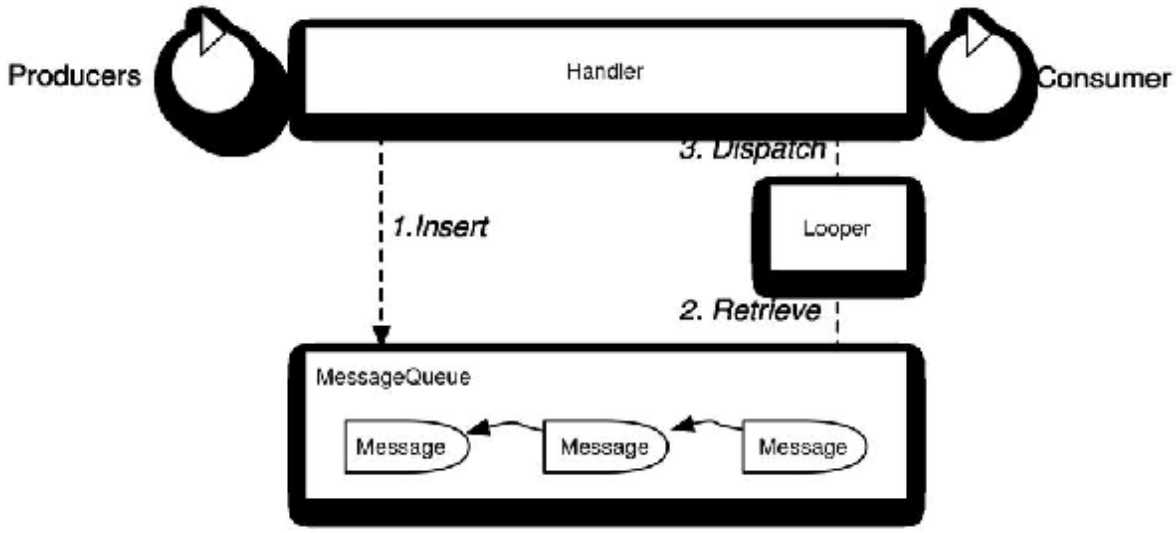


Figure 4-5. Overview of the message passing mechanism between multiple producer threads and one consumer thread. Every message refers to to the next message in the queue.

1. Insert

   The producer thread inserts messages in the queue by using the Handler connected to the consumer thread, as shown in Handler.

2. Retrieve

   The Looper, discussed in Looper, runs in the consumer thread and retrieves messages from the queue in a sequential order.

3. Dispatch

   The Handler is responsible for processing the messages on the consumer thread. A thread may have multiple Handler instances for processing messages; the Looper ensures that messages are dispatched to the correct Handler.

Basic Message Passing Example

Before we dissect the components in detail, let's look at a fundamental message passing example to get us acquainted with the code setup.

The following code implements what is probably one of the most common use cases. The user presses a button on the screen that could trigger a long operation, e.g., a network operation. To avoid stalling the rendering of the UI, the long operation, represented here by a dummy doLongRunningOperation() method, has to be executed on a worker thread. Hence, the setup is merely one producer thread (the UI thread) and one consumer thread (LooperThread).

Our code sets up a message queue. IT handles the button click as usual in the onClick() callback, which executes on the UI thread. In our implementation, the callback inserts a dummy message into the message queue. For sake of brevity, layouts and UI components have been left out of the example code.

```
public class LooperActivity extends Activity {

        LooperThread mLooperThread;

        private static class LooperThread extends Thread { ❶public Handler mHandler;

                public void run() {
                        Looper.prepare();❷mHandler = new Handler() { ❸public void
handleMessage(Message msg) { ❹if(msg.what == 0) {
                                                        doLongRunningOperation();
                                                }
                                        }
                        };
                        Looper.loop(); ❺}
        }

    public void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        mLooperThread = new LooperThread(); ❻mLooperThread.start();
    }

    public void onClick(View v) {
        if (mLooperThread.mHandler != null) {❼Message msg =
mLooperThread.mHandler.obtainMessage(0); ❽
mLooperThread.mHandler.sendMessage(msg); ❾}
    }

        private void doLongRunningOperation() {
                // Add long running operation here.
        }

    protected void onDestroy() {
        mLooperThread.mHandler.getLooper().quit(); ❿}
}
❶
```

Definition of the worker thread, acting as a consumer of the message queue.

❷

Associate a Looper — and implicitly a MessageQueue — with the thread.

❸

Set up a Handler to be used by the producer for inserting messages in the queue. Here we use the default constructor so it will bind to the Looper of the current thread. Hence, this Handler can created only after Looper.prepare(), or it will have nothing to bind to.

❹

Callback that runs when the message has been dispatched to the worker thread. It checks the what parameter, and then executes the long operation.

❺

Start dispatching messages from the message queue to the consumer thread. This is a blocking call, so the worker thread will not finish.

❻

Start the worker thread, so that it is ready to process messages.

❼

There is race condition between the setup of mHandler on a background thread and this usage on the UI thread. Hence, validate that mHandler is available.

❽

Initialize a Message-object with the what argument arbitrarily set to 0.

❾

Insert the message in the queue.

❿

Terminate the background thread. The call to Looper.quit() stops the dispatching of messages and releases Looper.loop() from blocking, so the run method can finish, leading to the termination of the thread.

Classes Used in Message Passing

Let's take a more detailed look now at the specific components of message passing and their use.

MessageQueue

The message queue is represented by the android.os.MessageQueue class. It is built with linked messages, constituting an unbound one-directional linked list. Producer threads insert messages that will later be dispatched to the consumer. The messages are sorted based on timestamps. The pending message with the lowest timestamp value is first in line for dispatch to the consumer. However, a message is dispatched only if the timestamp value is less than the current time. If not, the dispatch will wait until the current time has passed the timestamp value.

Figure 4-6 illustrates a message queue with three pending messages, sorted with timestamps where t1 < t2 < t3. Only one message has passed the dispatch barrier, i.e., the current time. Messages eligible for dispatch have a timestamp value less than the current time, i.e. "Now" in the figure.

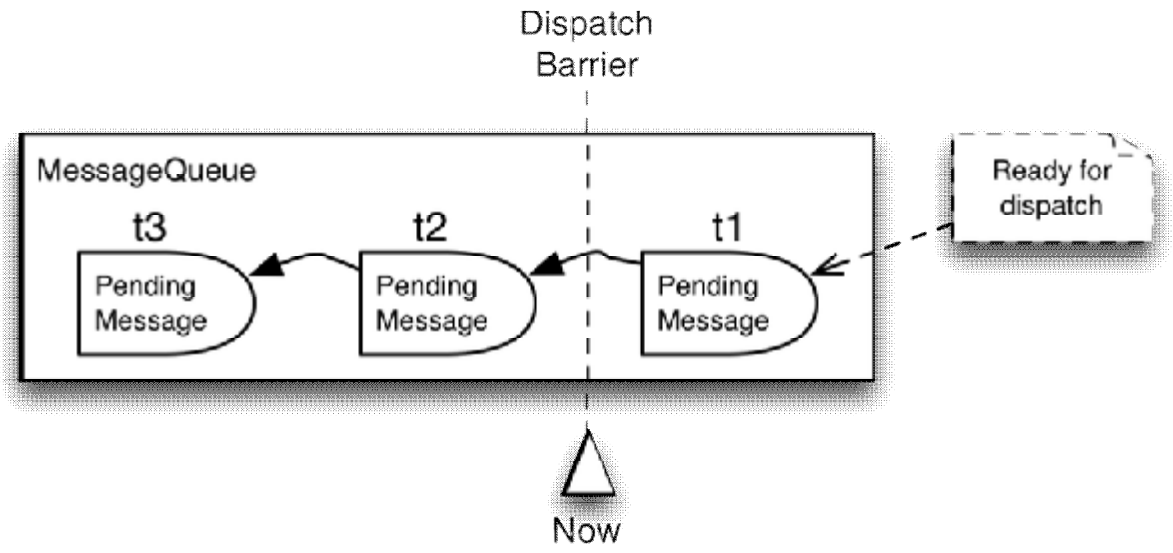


Figure 4-6. Pending messages in the queue. The rightmost message is first in queue to be processed.

If no message has passed the dispatch barrier when the Looper is ready to retrieve the next message, the consumer thread blocks. Execution is resumed as soon as a message passes the dispatch barrier.

The producers can insert new messages in the queue at any time and on any position in the queue. The insert position in the queue is based on the timestamp value. If a new message has the lowest timestamp value compared to the pending messages in the queue, it will occupy the first position in the queue, which is next to be dispatched. Insertions always comply to the timestamp sorting order. Message insertion is discussed further in Handler.

MessageQueue.IdleHandler

If there is no message to process, a consumer thread has some idle time. For instance, Figure 4-7 illustrates a time slot where the consumer thread is idle. By default, the consumer thread simply waits for new messages during idle time, but instead of waiting, the thread can be utilized to execute other tasks during these idle slots. This feature can be utilized to let non-critical tasks postpone their execution until no other messages are competing for execution time.

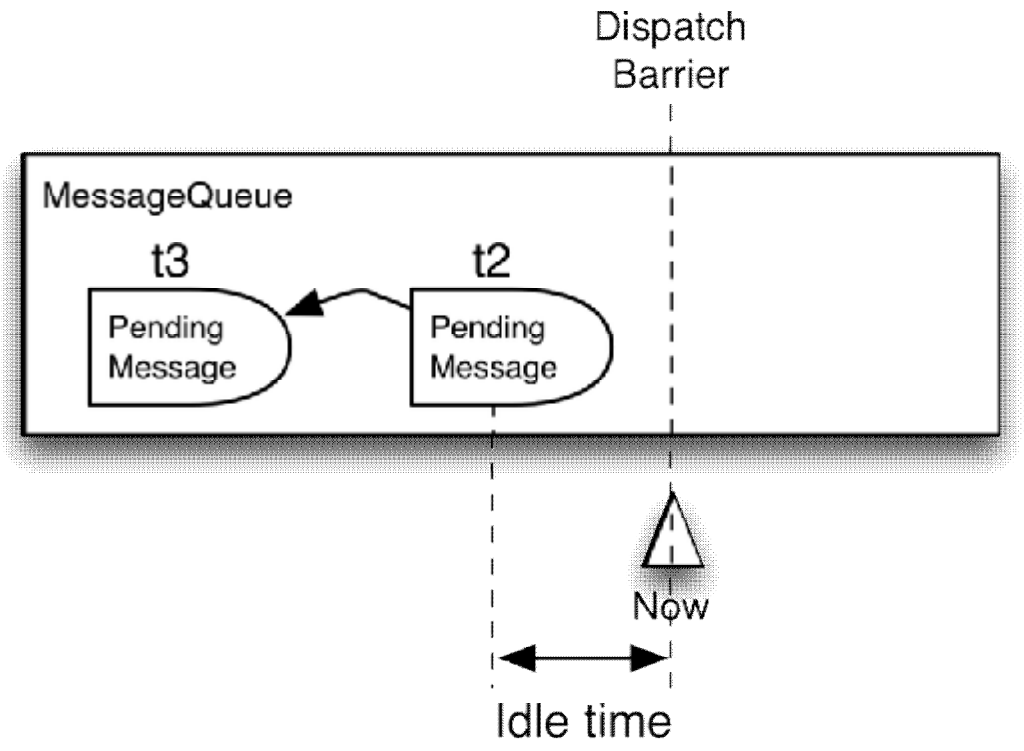


Figure 4-7. If no message has passed the dispatch barrier, there is a time slot that can be utilized for execution before the next pending message needs to be executed

When a pending message has been dispatched, and no other message has passed the dispatch barrier, a time slot occurs where the consumer thread can be utilized for execution of other

tasks. An application gets hold of this time slot with the android.os.MessageQueue.IdleHandler-interface; it is a listener that generates callbacks when the consumer thread is idle. The listener is attached to the MessageQueue and detached from it through the following calls:

```
// Get the message queue of the current thread.
MessageQueue mq = Looper.myQueue();
// Create and register an idle listener.
MessageQueue.IdleHandler idleHandler = new MessageQueue.IdleHandler();
mq.addIdleHandler(idleHandler)
// Unregister an idle listener.
mq.removeIdleHandler(idleHandler)
```

The idle handler interface consists of one callback method only:

```
interface IdleHandler {
    boolean queueIdle();
}
```

When the message queue detects idle time for the consumer thread, it invokes queueIdle() on all registered IdleHandler-instances. It is up to the application to implement the callback responsibly. You should usually avoid long-running tasks, because they will delay pending messages during the time they run.

The implementation of queueIdle() must return a Boolean value with the following meanings:

true
: The idle handler is kept active; it will continue to receive callbacks for successive idle time slots.

false
: The idle handler is inactive; it will not receive anymore callbacks for successive idle time slots. This is the same thing as removing the listener through MessageQueue.removeIdleHandler().

Example: Using IdleHandler to terminate an unused thread

All registered IdleHandlers to a MessageQueue are invoked when a thread has idle slots, where it waits for new messages to process. The idle slots can occur before the first message, between messages and after the last message. If multiple content producers should process data sequentially on a consumer thread, the IdleHandler can be used to terminate the consumer thread when all messages are processed, so that the unused thread does not linger

in memory. With the IdleHandler, it is not necessary to keep track of the last inserted message to know when the thread can be terminated.

Warning

This use case applies only when the producing threads insert messages in the MessageQueue without delay, so that the consumer thread is never idle until the last message is inserted.

The ConsumeAndQuitThread method shows the structure of a consuming thread with Looper and MessageQueue that terminates the thread when there are no more messages to process.

```
public class ConsumeAndQuitThread extends Thread implements
MessageQueue.IdleHandler {

    private static final String THREAD_NAME = "ConsumeAndQuitThread";

    public Handler mConsumerHandler;
    private boolean mIsFirstIdle = true;

    public ConsumeAndQuitThread() {
        super(THREAD_NAME);
    }

    @Override
    public void run() {
        Looper.prepare();

        mConsumerHandler = new Handler() {
            @Override
            public void handleMessage(Message msg) {
                    // Consume data
            }
        };
        Looper.myQueue().addIdleHandler(this);❶Looper.loop();
    }


    @Override
    public boolean queueIdle() {
        if (mIsFirstIdle) { ❷mIsFirstIdle = false;
            return true; ❸}
```

```
            mConsumerHandler.getLooper().quit(); ❹return false;
        }

        public void enqueueData(int i) {
            mConsumerHandler.sendEmptyMessage(i);
        }
    }
```

❶

Register the IdleHandler on the background thread when it is started and the Looper is prepared, so that the MessageQueue is set up.

❷

Let the first queueIdle invocation pass, because it occurs before the first message is received.

❸

Return true on the first invocation so that the IdleHandler still is registered.

❹

Terminate the thread.

The message insertion is done from multiple threads concurrently, with a simulated randomness of the insertion time.

```
final ConsumeAndQuitThread consumeAndQuitThread = new ConsumeAndQuitThread();
consumeAndQuitThread.start();

for (int i = 0; i < 10; i++) {
    new Thread(new Runnable() {
        @Override
        public void run() {
            for (int i = 0; i < 10; i++) {
                SystemClock.sleep(new Random().nextInt(10));
                consumeAndQuitThread.enqueueData(i);
            }
        }
    }).start();
```

Message

Each item on the MessageQueue is of the android.os.Message class. This is a container object carrying either a data item or a task, never both. Data is processed by the consumer thread, whereas a task is simply executed when it is dequeued and you have no other processing to do.

Note

The message knows its recipient processor—i.e. Handler —and can enqueue itself through Message.sendToTarget():

```
Message m = Message.obtain(handler, runnable);
m.sendToTarget();
```

As we will see in Handler, the handler is most commonly used for message enqueuing, as it offers more flexibility with regard to message insertion.

Data message

The data set has multiple parameters that can be handed off to the consumer thread, as shown in Table 4-2.

Table 4-2. Message parameters

Parameter name	Type	Usage
what	int	Message identifier. Communicates intention of the message.
arg1, arg2	int	Simple data values to handle the common use case of handing over integers. If a maximum of two integer values are to be passed to the consumer, these parameters are more efficient than allocating a Bundle, as explained under the data parameter.
obj	Object	Arbitrary object. If the object is handed off to a thread in another process, it has to implement Parcelable.
data	Bundle	Container of arbitrary data values.
replyTo	Messenger	Reference to Handler in some other process. Enables inter-process message communication, as described in Two-way communication.
callback	Runnable	Task to execute on a thread. This is an internal instance field that holds the Runnable object from the Handler.post methods in Handler.

Task message

The task is represented by a java.lang.Runnable object to be executed on the consumer thread. Task messages cannot contain any data beyond the task itself.

A MessageQueue can contain any combination of data and task messages. The consumer thread processes them in a sequential manner, independent of the type. If a message is a data message, the consumer processes the data. Task messages are handled by letting the Runnable execute on the consumer thread, but the consumer thread does not receive a message to be processed in Handler.handleMessage(Message), as it does with data messages.

The lifecycle of a message is simple: the producer creates the message, and eventually it is processed by the consumer. This description suffices for most use cases, but when a problem arises, a deeper understanding of message handling is invaluable. Let us take a look into what actually happens with the message during its lifecycle, which can be split up into four principal states shown in Figure 4-8. The runtime stores message objects in an application-wide pool to enable the reuse of previous messages; this avoids the overhead of creating new instances for every hand-off. The message object execution time is normally very short and many messages are processed per time unit.

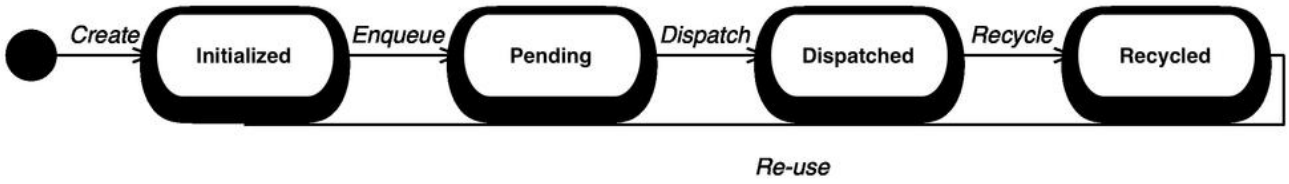


Figure 4-8. Message lifecycle states.

The state transfers are partly controlled by the application and partly by the platform. Note that the states are not observable, and an application cannot follow the changes from one state to another (although there are ways to follow the movement of messages, explained later in Observing the Message Queue). Therefore, an application should not make any assumptions about the current state when handling a message.

Initialized

In the initialized state, a message object with mutable state has been created and, if it is a data message, populated with data. The application is responsible for creating the message object using one of the following calls. They take an object from the object pool.

- Explicit object construction

  ```
  Message m    = new Message();
  ```

- Factory methods
  - Empty message

    ```
    Message m = Message.obtain();
    ```

- Data message
- Message m = Message.obtain(Handler h);
- Message m = Message.obtain(Handler h, int what);
- Message m = Message.obtain(Handler h, int what, Object o);
- Message m = Message.obtain(Handler h, int what, int arg1, int arg2);

  Message m = Message.obtain(Handler h, int what, int arg1, int arg2, Object o);

- Task message

  Message m = Message.obtain(Handler h, Runnable task);

- Copy constructor

  Message m = Message.obtain(Message originalMsg);

Pending

The message has been inserted into the queue by the producer thread, and it is waiting to be dispatched to the consumer thread.

Dispatched

In this state, the Looper has retrieved and removed the message from the queue. The message has been dispatched to the consumer thread and is currently being processed. There is no application API for this operation because the dispatch is controlled by the Looper, without the influence of the application. When the looper dispatches a message, it checks the delivery information of the message, and delivers the message to the correct recipient. Once dispatched, the message is executed on the consumer thread.

Recycled

At this point in the lifecycle, the message state is cleared and the instance is returned to the message pool. The Looper handles the recycling of the message when it has finished executing on the consumer thread. Recycling of messages is handler by the runtime and should not be done explicitly by the application.

Note

Once a message is inserted in the queue, the content should not be altered. In theory, it is valid to change the content before the message is dispatched. However, because the state is not observable, the message may be processed by the consumer thread while the producer tries to change the data, raising thread safety concerns. It would be even worse if the message

has been recycled, because it then has been returned to the message pool and possibly used by another producer to pass data in another queue.

Looper

The android.os.Looper class handles the dispatch of messages in the queue to the associated handler. All messages that have passed the dispatch barrier, as illustrated in Figure 4-6, are eligible for dispatch by the Looper. As long as the queue has messages eligible for dispatch, the Looper will ensure that the consumer thread receives the messages. When no messages have passed the dispatch barrier, the consumer thread will block until a message has passed the dispatch barrier.

The consumer thread does not interact with the message queue directly to retrieve the messages. Instead, a message queue is added to the thread when the looper has been attached. The looper manages the message queue and facilitates the dispatch of messages to the consumer thread.

By default, only the UI thread has a Looper; threads created in the application need to get a Looper associated explicitly. When the Looper is created for a thread, it is connected to a message queue. The Looper acts as the intermediator between the queue and the thread. The Looper setup is done in the run method of the thread:

```
class ConsumerThread extends Thread {
    @Override
    public void run() {
        Looper.prepare(); ❶// Handler creation omitted.

        Looper.loop(); ❷}
}
```

❶

The first step is to create the Looper, which is done with the static prepare() method; it will create a message queue and associate it with the current thread. At this point, the message queue is ready for insertion of messages, but they are not dispatched to the consumer thread.

❷

Start handling messages in the message queue. This is a blocking method that ensures the run() method is not finished; while run() blocks, the Looper dispatches messages to the consumer thread for processing.

A thread can have only one associated Looper; a runtime error will occur if the application tries to set up a second one. Consequently, a thread can have only one message queue, meaning that messages sent by multiple producer threads are processed sequentially on the consumer thread. Hence, the currently executing message will postpone subsequent messages until it has been processed. Messages with long execution times shall not be used if they can delay other important tasks in the queue.

Looper termination

The Looper is requested to stop processing messages with either quit or quitSafely: quit() stops the looper from dispatching any more messages from the queue; all pending messages in the queue-including those that have passed the dispatch barrier will be discarded. quitSafely, on the other hand, only discards the messages that has not passed the dispatch barrier. Pending messages that are eligable for dispatch will be processed before the Looper is terminated.

Note

quitSafely was added in API level 18 (Jelly Bean 4.3). Previous API levels only support quit.

Terminating a Looper does not terminate the thread; it merely exits Looper.loop() and lets the thread resume running in the method that invoked the loop call. But you cannot start the old looper or a new one, so the thread can no longer enqueue or handle messages. If you call Looper.prepare(), it will throw RuntimeException because the thread already has an attached Looper. If you call Looper.loop(), it will block, but no messages will be dispatched from the queue.

The UI thread Looper

The UI thread is the only thread with an associated Looper by default. It is a regular thread, like any other thread created by the application itself, bút the Looper is associated with the thread[7] before the application components are initialized.

There are a few practical differences between the UI thread Looper and other application thread Loopers:

- It is accessible from everywhere, through the Looper.getMainLooper() method.
- It cannot be terminated. Looper.quit() throws RuntimeException.
- The runtime associates a Looper to the UI thread by Looper.prepareMainLooper(). This can be done only once per application. Thus, trying to attach the main looper to another thread will throw an exception.

Handler

So far, the focus has been on the internals of Android thread communication, but an application mostly interacts with the android.os.Handler class. It is a two-sided API that both handles the insertion of messages into the queue and the message processing. As Figure 4-5, it is invoked from both the producer and consumer thread typically used for:

- Creating messages
- Inserting messages into the queue
- Processing messages on the consumer thread
- Managing messages in the queue

Setup

While carrying out its responsibilities, the Handler interacts with the Looper, message queue, and message. As Figure 4-4 illustrates, the only direct instance relation is to the Looper, which is used to connect to the MessageQueue. Without a Looper, a Handler can not function; it cannot couple with a queue to insert messages and consequently it will not receive any messages to process. Hence, a Handler instance is bound to a Looper instance already at construction time:

- Constructors without an explicit Looper bind to the Looper of the current thread.
- new Handler();
  new Handler(Handler.Callback)

- Constructors with an explicit Looper bind to that Looper.
- new Handler(Looper);
  new Handler(Looper, Handler.Callback);

If the constructors without an explicit Looper are called on a thread without a Looper (i.e., it has not called Looper.prepare()), there is nothing the Handler can bind to, leading to a RuntimeException. Once a handler is bound to a Looper, the binding is final.

A thread can have multiple Handlers; messages from them coexist in the queue but are dispatched to the correct Handler instance, as shown in Figure 4-9.

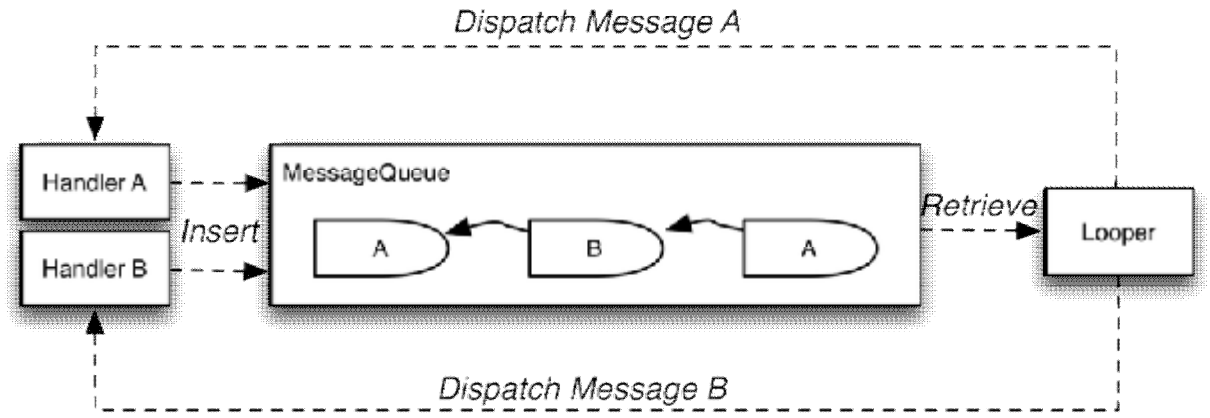


Figure 4-9. Multiple handlers using one Looper. The handler inserting a message is the same handler processing the message.

Note

Multiple handlers will not enable concurrent execution. The messages are still in the same queue and are processed sequentially.

Message creation

For simplicity, the Handler class offers wrapper functions for the factory methods shown in Initialized to create objects of the *Message* class.

Message obtainMessage(int what, int arg1, int arg2)
Message obtainMessage()
Message obtainMessage(int what, int arg1, int arg2, Object obj)
Message obtainMessage(int what)
Message obtainMessage(int what, Object obj)

The message obtained from a Handler is retrieved from the message pool and implicitly connected to the Handler instance that requested it. This connection enables the looper to dispatch each message to the correct handler.

Message insertion

The handler inserts messages in the message queue in various ways depending on the message type. Task messages are inserted through methods whose names begin with post, whereas data messages are inserted through methods whose names begin with send:

- Add a task to the message queue.
- boolean post(Runnable r)f
- boolean postAtFrontOfQueue(Runnable r)

- boolean postAtTime(Runnable r, Object token, long uptimeMillis)
- boolean postAtTime(Runnable r, long uptimeMillis)

  boolean postDelayed(Runnable r, long delayMillis)

- Add a data object to the message queue.
- boolean sendMessage(Message msg)
- boolean sendMessageAtFrontOfQueue(Message msg)
- boolean sendMessageAtTime(Message msg, long uptimeMillis)

  boolean sendMessageDelayed(Message msg, long delayMillis)

- Add simple data object to the message queue.
- boolean sendEmptyMessage(int what)
- boolean sendEmptyMessageAtTime(int what, long uptimeMillis)

  boolean sendEmptyMessageDelayed(int what, long delayMillis)

All insertion methods put a new Message object in the queue, even though the application does not create the Message object explicitly. The objects, such as Runnable in a task post and what in a send, are wrapped into Message objects, because those are the only data types allowed in the queue.

Every message inserted in the queue comes with a time parameter indicating the time when the message is eligible for dispatch to the consumer thread. The sorting is based on the time parameter, and it is the only way an application can affect the dispatch order.

default
: Immediately eligible for dispatch.

at_front
: This message is eligible for dispatch at time 0. Hence, it will be the next dispatched message, unless another is inserted at the front before this one is processed.

delay
: The amount of time after which this message is eligible for dispatch.

uptime
: The absolute time at which this message is eligible for dispatch.

Even though explicit delays or uptimescan be specified, the time required to process each message is still indeterminate. It depends both on whatever existing messages need to be processed first and the operating system scheduling.

Inserting a message in the queue is not failsafe. Some common errors that can occur are listed in Table 4-3.

Table 4-3. Message insertion errors

Failure	Error response	Typical application problem
Message has no Handler.	RuntimeException	Message was created from a Message.obtain() method without a specified Handler.
Message has already been dispatched and is being processed.	RuntimeException	The same message instance was inserted twice.
Looper has exited.	Return false	Message is inserted after Looper.quit() has been called.

Warning

The dispatchMessage method of the Handler class is used by the Looper to dispatch messages to the consumer thread. If used by the application directly, the message will be processed immediately on the calling thread and not the consumer thread.

Example: Two-way message passing

The HandlerExampleActivity simulates a long running operation that is started when the user clicks a button. The long running task is executed on a background thread; meanwhile, the UI displays a progress bar that is removed when the background thread reports the result back to the UI thread.

First, the setup of the Activity:

```
public class HandlerExampleActivity extends Activity {

    private final static int SHOW_PROGRESS_BAR = 1;
    private final static int HIDE_PROGRESS_BAR = 0;
    private BackgroundThread mBackgroundThread;

    private TextView mText;
    private Button mButton;
    private ProgressBar mProgressBar;

    @Override
    public void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_handler_example);

        mBackgroundThread = new BackgroundThread();
```

```
        mBackgroundThread.start();❶mText = (TextView) findViewById(R.id.text);
        mProgressBar = (ProgressBar) findViewById(R.id.progress);
        mButton = (Button) findViewById(R.id.button);
        mButton.setOnClickListener(new OnClickListener() {
            @Override
            public void onClick(View v) {
                mBackgroundThread.doWork(); ❷}
        });
    }

    @Override
    protected void onDestroy() {
        super.onDestroy();
        mBackgroundThread.exit();❸}

    // ... The rest of the Activity is defined further down

}
```

❶

A background thread with a message queue is started when the HandlerExampleActivity is created. It handles tasks from the UI thread.

❷

When the user clicks a button, a new task is sent to the background thread. As the tasks will be executed sequentially on the background thread, multiple button clicks may lead to queueing of tasks before they are processed.

❸

The background thread is stopped when the HandlerExampleActivity is destroyed.

BackgroundThread is used to offload tasks from the UI thread. It runs—and can receive messages—during the lifetime of the HandlerExampleActivity. It does not expose its internal Handler; instead it wraps all accesses to the Handler in public methods doWork and exit.

```
private class BackgroundThread extends Thread {

    private Handler mBackgroundHandler;

    public void run() { ❶Looper.prepare();
```

```
        mBackgroundHandler = new Handler(); ❷Looper.loop();
    }

    public void doWork() {
        mBackgroundHandler.post(new Runnable() { ❸@Override
            public void run() {
                Message uiMsg = mUiHandler.obtainMessage(
                    SHOW_PROGRESS_BAR, 0, 0, null); ❹
mUiHandler.sendMessage(uiMsg); ❺Random r = new Random();
                int randomInt = r.nextInt(5000);
                SystemClock.sleep(randomInt); ❻uiMsg = mUiHandler.obtainMessage(
                    HIDE_PROGRESS_BAR, randomInt, 0, null); ❼
mUiHandler.sendMessage(uiMsg); ❽}
        });
    }

    public void exit() { ❾mBackgroundHandler.getLooper().quit();
    }
}
```

❶

Associate a Looper with the thread.

❷

The Handler processes only Runnables. Hence, it is not required to implement Handler.handleMessage.

❸

Post a long task to be executed in the background.

❹

Create a Message object that contains only a what argument with a command—SHOW_PROGRESS_BAR—to the UI thread so that it can show the progress bar.

❺

Send the start message to the UI thread.

❻

Simulate a long task of random length, that produces some data *randomInt*.

❼

Create a Message object with the result randomInt, that is passed in the *arg1* parameter. The what parameter contains a command—HIDE_PROGRESS_BAR—to remove the progress bar.

❽

The message with the end result that both informs the UI thread that the task is finished and delivers a result.

❾

Quit the Looper so that the thread can finish.

The UI thread defines its own Handler that can receive commands to control the progress bar and update the UI with results from the background thread.

```
private final Handler mUiHandler = new Handler() {
    public void handleMessage(Message msg) {

        switch(msg.what) {
            case SHOW_PROGRESS_BAR: ❶mProgressBar.setVisibility(View.VISIBLE);
                break;
            case HIDE_PROGRESS_BAR: ❷mText.setText(String.valueOf(msg.arg1));
                mProgressBar.setVisibility(View.INVISIBLE);
                break;
        }
    }
};
```

❶

Show the progress bar.

❷

Hide the progress bar and update the TextView with the produced result.

Message processing

Messages dispatched by the Looper are processed by the Handler on the consumer thread. The message type determines the processing:

Task messages

Task messages contain only a Runnable and no data. Hence, the processing to be executed is defined in the run method of the Runnable, which is executed automatically on the consumer thread, without invoking Handler.handleMessage().

Data messages

When the message contains data, the handler is the receiver of the data, and is responsible for its processing. The consumer thread processes the data by overriding the Handler.handleMessage(Message msg) method. There are two ways to do this, described in the text that follows.

One way to define handleMessage is to do it as part of creating a Handler. The method should be defined as soon as the message queue is available (after Looper.prepare() is called) but before the message retrieval starts (before Looper.loop() is called).

A template follows for setting up the handling of data messages:

```
class ConsumerThread extends Thread {
    Handler mHandler;
    @Override
    public void run() {
        Looper.prepare();
        mHandler = new Handler() {
            public void handleMessage(Message msg) {
                // Process data message here
            }
        };)
        Looper.loop();
    }
}
```

In this code, the Handler is defined as an anonymous inner class, but it could as well have been defined as a regular or inner class.

A convenient alternative to extending the Handler class is to use the Handler.Callback interface, which defines a handleMessage method with an additional return parameter not in Handler.handleMessage().

```
public interface Callback {
    public boolean handleMessage(Message msg);
}
```

With the Callback interface, it is not necessary to extend the Handler class. Instead, the Callback implementation can be passed to the Handler constructor, and it will then receive the dispatched messages for processing.

```
public class HandlerCallbackActivity extends Activity implements Handler.Callback {
    Handler mUiHandler;

    @Override
    public void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        mUiHandler = new Handler(this); ❶}

    @Override
    public boolean handleMessage(Message message) { ❷// Process messages
        return true;
    }
}
```

Callback.handleMessage should return true if the message is handled, which guarantees that no further processing of the message is done. If, however, false is returned, the message is passed on to the Handler.handleMessage method for further processing. Note that the Callback does not override Handler.handleMessage. Instead, it adds a message preprocessor that is invoked before the Handlers own method. The Callback preprocessor can intercept and change messages before the Handler receives them. The following code shows the principle for intercepting messages with the Callback:

```
public class HandlerCallbackActivity extends Activity implements Handler.Callback {❶
@Override
    public boolean handleMessage(Message msg) { ❷Logg.d(TAG, "Primary Handler- msg =
" + msg.what);
        switch (msg.what) {
            case 1:
                msg.what = 11;
                return true;
            default:
                msg.what = 22;
                return false;
```

```
        }
    }

    // Invoked on button click
    public void onHandlerCallback(View v) {
        Handler handler = new Handler(this) {
            @Override
            public void handleMessage(Message msg) {
                Log.d(TAG, "Secondary Handler - msg = " + msg.what); ❸}
        };
        handler.sendEmptyMessage(1); ❹handler.sendEmptyMessage(2); ❺}
}
```

❶❶

The HandlerCallbackActivity implements the Callback interface to intercept messages.

❷❷

The Callback implementation intercepts messages. If msg.what is 1, it returns true—the message is handled. Otherwise, it changes the value of msg.what to 22 and returns false—the message is not handled, so it is passed on to the Handler implementation of handleMessage.

❸

Log the messages sent to the secondary handleMessage

❹

Insert a message with msg.what == 1. The message is intercepted by the Callback as it returns true.

❺

Insert a message with msg.what == 2. The message is changed by the Callback and passed on to the Handler that prints Secondary Handler - msg = 22.

Removing Messages from the Queue

After enqueuing a message, the producer can invoke a method of the Handler class to remove the message, so long as it has not been dequeued by the Looper. Sometimes an application may want to clean the message queue by removing all messages, which is possible, but most often a more fine-grained approach is desired: an application wants to target only a subset of

the messages. For that, it needs to be able to identify the correct messages. Hence, messages can be identified from certain properties:

Table 4-4. Message identifiers

Identifier type	Description	Messages to which it applies
Handler	Message receiver	Both task and data messages
Object	Message tag	Both task and data messages
Integer	what parameter of Message	Data messages
Runnable	Task to be executed	Task messages

The handler identifier is mandatory for every message, because a message always knows what handler it will be dispatched to. This requirement implicitly restricts each Handler to removing only messages belonging to that Handler. It is not possible for a Handler to remove messages in the queue that were inserted by another Handler.

The methods available in the Handler class for managing the message queue are:

- Remove a task from the message queue.
- removeCallbacks(Runnable r)
  removeCallbacks(Runnable r, Object token)

- Remove a data message from the message queue.
- removeMessages(int what)
  removeMessages(int what, Object object)

- Remove tasks and data messages from the message queue.

  removeCallbacksAndMessages(Object token)

The Object identifier used in both the data and task message. Hence, it can be assigned to messages as a kind of tag, allowing you later to remove related messages that you have tagged with the same Object.

For instance, the following excerpt inserts two messages in the queue, to make it possible to remove them later based on the tag.

```
Object tag = new Object(); ❶Handler handler = new Handler()
    public void handleMessage(Message msg) {
        // Process message
        Log.d("Example", "Processing message");
    }
};
```

```
Message message = handler.obtainMessage(0, tag); ❷handler.sendMessage(message);

handler.postAtTime(new Runnable() { ❸public void run() {
            // Left empty for brevity
      }
}, tag, SystemClock.uptimeMillis());

handler.removeCallbacksAndMessages(tag); ❹
```

❶

The message tag identifier, common to both the task and data message.

❷

The object in a Message instance is used both as data container and implicitly defined message tag.

❸

Post a task message with an explicitly defined message tag.

❹

Remove all messages with the tag.

As indicated before, you have no way to find out whether a message was dispatched and handled before you issues a call to remove it. Once the message is dispatched, the producer thread that enqueued it cannot stop its task from executing or its data from being processed.

Observing the Message Queue

It is possible to observe pending messages and the dispatching of messages from a Looper to the associated Handlers. The Android platform offers two observing mechanisms. Let us take a look at them by example.

The first example shows how it is possible to log the current snapshot of pending messages in the queue.

Taking a snapshot of the current message queue

This example creates a worker thread when the Activity is created. When the user presses a button, causing onClick to be called, six messages are added to the queue in different ways. Afterwards we observe the state of the message queue.

```
public class MQDebugActivity extends Activity {

    private static final String TAG = "EAT";
    Handler mWorkerHandler;

    public void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_mqdebug);

        Thread t = new Thread() {
            @Override
            public void run() {
                Looper.prepare();
                mWorkerHandler = new Handler() {
                    @Override
                    public void handleMessage(Message msg) {
                        Log.d(TAG, "handleMessage - what = " + msg.what);
                    }
                };
                Looper.loop();
            }
        };
        t.start();
    }

    // Called on button click, i.e. from the UI thread.
    public void onClick(View v) {
        mWorkerHandler.sendEmptyMessageDelayed(1, 2000);
        mWorkerHandler.sendEmptyMessage(2);
        mWorkerHandler.obtainMessage(3, 0, 0, new Object()).sendToTarget();
        mWorkerHandler.sendEmptyMessageDelayed(4, 300);
        mWorkerHandler.postDelayed(new Runnable() {
            @Override
            public void run() {
                Log.d(TAG, "Execute");
            }
        }, 400);
```

```
            mWorkerHandler.sendEmptyMessage(5);

            mWorkerHandler.dump(new LogPrinter(Log.DEBUG, TAG), "");
        }
}
```

Six messages, with the parameters shown in Figure 4-10, are added to the queue.

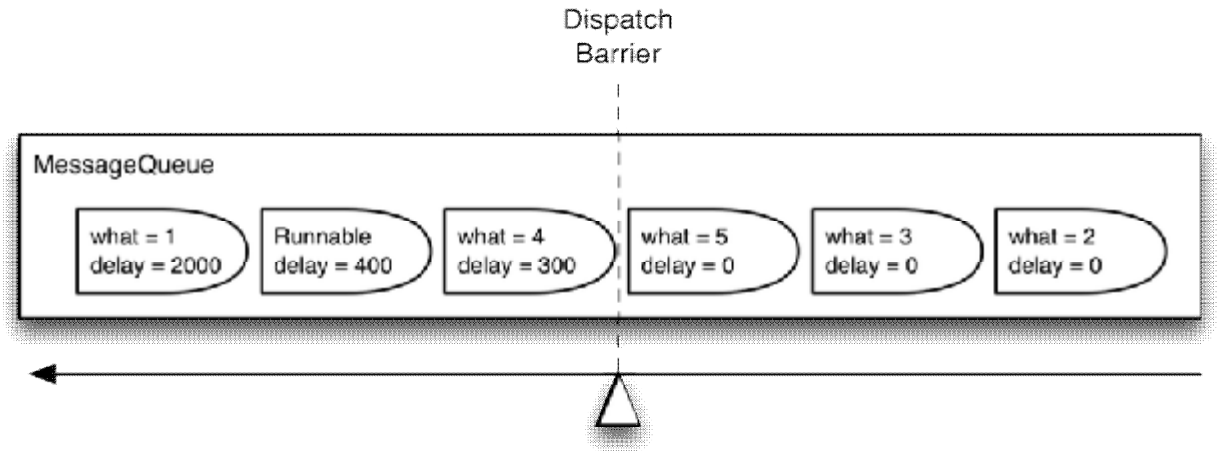


Figure 4-10. Added messages in the queue.

Right after the messages are added to the queue, a snapshot is printed to the log. Only pending messages are observed. Hence, the number of messages actually observed depends on how many messages have already been dispatched to the handler. Three of the messages are added without a delay, which makes them eligible for dispatch at the time of the snapshot.

A typical run of the preceding code produces the following log:

```
49.397: handleMessage - what = 2
49.397: handleMessage - what = 3
49.397: handleMessage - what = 5
49.397: Handler (com.wifill.eat.ui.MQDebugActivity$1$1) {412cb3d8} @ 5994288
49.407: Looper{412cb070}
49.407:         mRun=true
49.407:         mThread=Thread[Thread-111,5,main]
49.407:         mQueue=android.os.MessageQueue@412cb090
49.407:             Message 0: { what=4 when=+293ms }
49.407:             Message 1: { what=0 when=+394ms }
49.407:             Message 2: { what=1 when=+1s990ms }
49.407:             (Total messages: 3)
```

```
49.707: handleMessage - what = 4
49.808: Execute
51.407: handleMessage - what = 1
```

The snapshot of the message queue shows that the messages with what parameters (0, 1, and 4) are pending in the queue. These are the messages added to the queue with a dispatch delay, whereas the others without a dispatch delay apparently have been dispatched already. This is a reasonable result because the handler processing is very short—just a print to the log.

The snapshot also shows how much time is left before each message in the queue will pass the dispatch barrier. For instance, the next message to pass the barrier is Message 0 (what= 4) in 293 ms. Messages still pending in the queue but eligible for dispatch will have a negative time indication in the log, e.g., when is less than zero.

Tracing the message queue processing

The message processing information can be printed to the log. Message queue logging is enabled from the Looper class. The following call enables logging on the message queue of the calling thread.:

```
Looper.myLooper().setMessageLogging(new LogPrinter(Log.DEBUG, TAG));
```

Let’s look at an example of tracing a message that is posted to the UI thread:

```
mHandler.post(new Runnable() {
    @Override
    public void run() {
        Log.d(TAG, "Executing Runnable");
    }
});

mHandler.sendEmptyMessage(42);
```

The example posts two events to the message queue: first a Runnable followed by an empty message. As expected, with the sequential execution in mind, the Runnable is processed first and consequently the first to be logged:

```
>>>>> Dispatching to Handler (android.os.Handler) {4111ef40}
com.wifill.eat.ui.MessageTracingActivity$1@41130820: 0
Executing Runnable
<<<<< Finished to Handler (android.os.Handler) {4111ef40}
com.wifill.eat.ui.MessageTracingActivity$1@41130820
```

The trace prints the start and end of the event identified by three properties:

Handler instance
    android.os.Handler 4111ef40
Task instance
    com.wifill.eat.ui.MessageTracingActivity$1@41130820
The what parameter
    0 (Runnable tasks do not carry a what parameter)

Similarly the trace of an message with the what parameter set to 42 prints the message argument but not any Runnable instance:

>>>>> Dispatching to Handler (android.os.Handler) {4111ef40} null: 42
<<<<< Finished to Handler (android.os.Handler) {4111ef40} null

Combining the two techniques of message queue snapshots and dispatch tracing allows the application to observe message passing in detail.

Communicating with the UI Thread

The UI thread is the only thread in an application that has an associated Looper by default, which is associated on the thread before the first Android component is started. The UI thread can be a consumer, to which other threads can pass messages. It's important to send only short-lived tasks to the UI thread. The UI thread is application global and processes both android component and system messages sequentially. Hence, long-lived tasks will have a global impact across the application.

Messages are passed to the UI thread through its Looper that is accessible globally in the application from all threads with Looper.getMainLooper():

```
Runnable task = new Runnable() {...};
new Handler(Looper.getMainLooper()).post(task);
```

Independent of the posting thread, the message is inserted in the queue of the UI thread. If it is the UI thread that posts the message to itself, the message can be processed at the earliest after the current message is done:

```
// Method called on UI thread.
private void postFromUiThreadToUiThread() {
    new Handler().post(new Runnable() { ... });

    // The code at this point is part of a message being processed
```

```
    // and is executed before the posted message.

}
```

However, a task message that is posted from the UI thread to itself can bypass the message passing and execute immediately within the currently processed message on the UI thread with the convenience method Activity.runOnUiThread(Runnable):

```
// Method called on UI thread.
private void postFromUiThreadToUiThread() {
    runOnUiThread(new Runnable() { ... });

    // The code at this point is executed after the message.

}
```

If it is called outside the UI thread, the message is inserted in the queue. The runOnUiThread method can only be executed from an Activity instance, but the same behavior can be implemented by tracking the ID of the UI thread, for example with a convenience method customRunOnUiThread in an Application subclass. The customRunOnUiThread inserts a message in the queue like the following example:

```
public class EatApplication extends Application {
    private long mUiThreadId;
    private Handler mUiHandler;

    @Override
    public void onCreate() {
        super.onCreate();
        mUiThreadId = Thread.currentThread().getId();
        mUiHandler = new Handler();
    }

    public void customRunOnUiThread(Runnable action) {
        if (Thread.currentThread().getId() != mUiThreadId) {
            mUiHandler.post(action);
        } else {
            action.run();
        }
    }
}
```

Summary

Android applications have access to the regular Java thread communication techniques, which suit worker thread communication well. However, they rarely fit the use case when one of the threads is the UI thread, which is the most common case. Android message passing is used extensively throughout applications, either explicitly or implicitly through various wrapping techniques, discussed in the second part of this book.

---

[7] The UI thread is managed by the platform internal class android.app.ActivityThread.

Chapter 5. Interprocess Communication

Android applications threads most often communicate within a process, sharing the process's memory as discussed in Chapter 4. However, communication across process boundaries—i.e. *Interprocess Communication* (IPC)—is supported by the Android platform through the *Binder Framework*, that manages the data transactions when there is no shared memory area between the threads.

The most common IPC use cases are handled by high-level components in Android, such as intents, system services, content providers, etc. They can be used by an application without it having to know whether it communicates within the process or between processes. Sometimes, however, it is necessary for an application to define a more explicit communication model, and be more involved in the actual communication. This chapter covers how threads communicate across process boundaries, which includes:

- Synchronous and asynchronous Remote Procedure Calls (RPC).
- Message communication through Messenger.
- Returning data with ResultReceiver

Android RPC

IPC is managed by the Linux OS, which supports several IPC techniques: signals, pipes, message queues, semaphores, and shared memory. In Android's modified Linux kernel the Linux IPC techniques have been replaced[8] by the *Binder Framework*, which enables a *Remote Procedure Call* (RPC) mechanism between processes; a client process can call remote methods in a server process as if the methods were executed locally. Hence, data can be passed to the server process, executed on a thread, and return a result value to the calling thread. The RPC method call itself is trivial, but the underlying RPC mechanism is complex:

- Method and data decomposition, also known as *marshalling*
- Transferring the marshalled information to the remote process
- Recomposing the information in the remote process, also known as *unmarshalling*
- Transferring return values back to the originating process

The Android application framework and core libraries abstract out the process communication with the Binder Framework and the Android Interface Definition Language (AIDL).

Binder

The Binder enables applications to transfer both functions and data—method calls—between threads running in different processes. The server process defines a remote interface supported by the android.os.Binder class, and threads in a client process can access the remote interface through this remote object.

A remote procedure call that transfers both a function and data is called a *transaction*; the client process calls the transact method and the server process receives the call in the onTransact method (Figure 5-1).

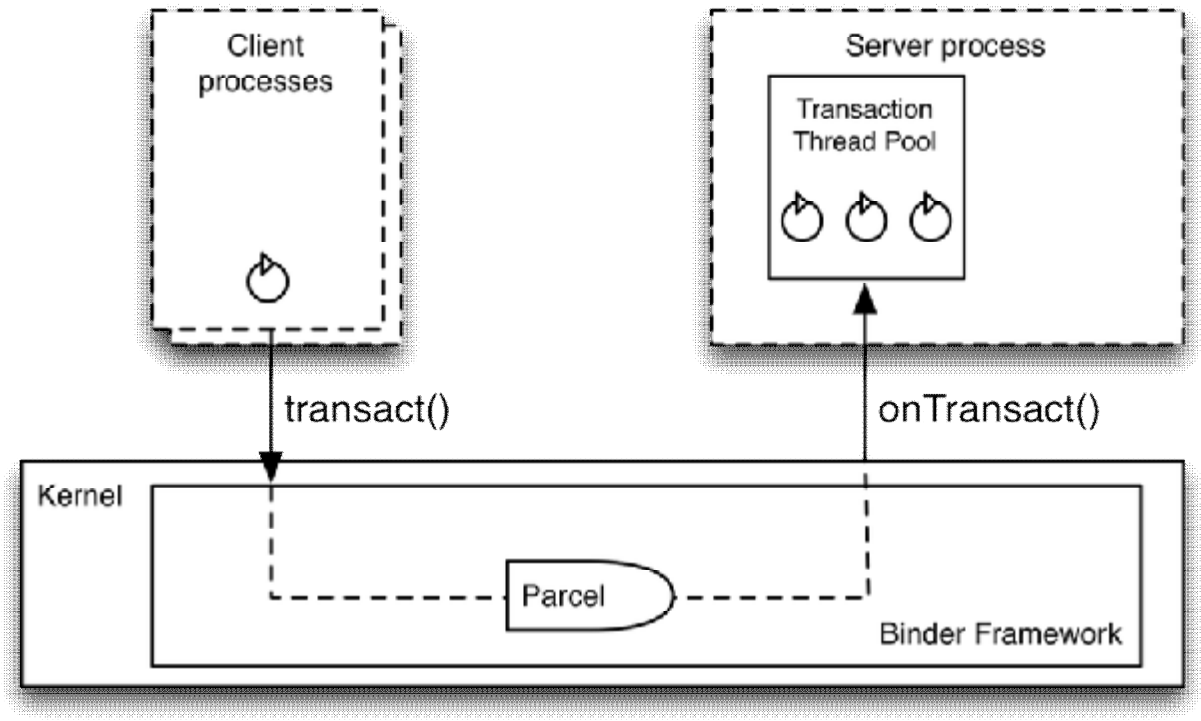


Figure 5-1. IPC through Binder

The client thread calling transact is blocked by default until onTransact has finished executing on the remote thread. Transaction data consists of android.os.Parcel objects, which are optimized to be sent across processes via the Binder. Arguments and return values are transferred as Parcel objects. These can contain literal arguments or custom objects that implement android.os.Parcelable—an interface that defines marshalling and unmarshalling of data in a more efficient way than a Serializable.

The onTransact method is executed on a thread from a pool of binder threads, discussed in Binder threads. This pool exists only to handle incoming requests from other processes. It has a maximum of 16 threads,[9] so 16 remote calls can be handled concurrently in every process. This requires the implementations of the calls to ensure thread safety.

IPC can be bidirectional, i.e., the server process can issue a transaction to the client process and reverse the flow: the former server process becomes the client and executes a transaction on another binder implemented in the former client process, whose own binder threads handle the processing. Hence, a two-way communication mechanism between two processes can be established. As we will see, this mechanism is important to enable asynchronous RPC.

Note

If the server process starts a transaction, calling transact to send a request to the client process while executing onTransact, the client process will not receive the incoming request on a binder thread, but on the thread waiting for the first transaction to finish.

The Binder also supports asynchronous transactions, which you can specify by setting IBinder.FLAG_ONEWAY. With that flag set, the client thread will call transact and return immediately. The Binder will continue calling onTransact on the Binder thread in the server process, but cannot return any data synchronously to the client thread.

AIDL

When a process wants to expose functionality for other processes to access, it has to define the communication contract. Basically, the server defines an interface of methods that clients can call. The simplest and most common way to describe the interface in the *Android Interface Definition Language* (AIDL)—defined in an *.aidl*.file. Compilation [10] of the aidl-file generates Java code that supports IPC. Android applications interact with the generated Java code but the applications only need to be aware of the interface. The procedure for defining the communication contract is shown in Figure 5-2.

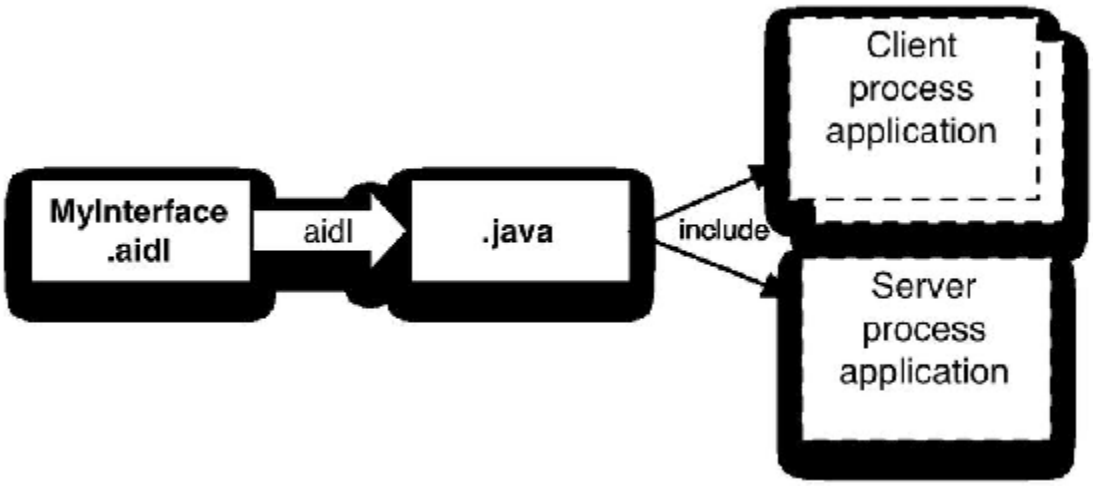


Figure 5-2. Construction of the remote communication interface

The generated Java interface is included in all the client applications and in the server application. The interface file defines two inner classes, Proxy and Stub, that handle all the marshalling and unmarshalling of the data and the transaction itself. Hence, the creation of AIDL automatically generates Java code that wraps the Binder Framework and sets up the communication contract.

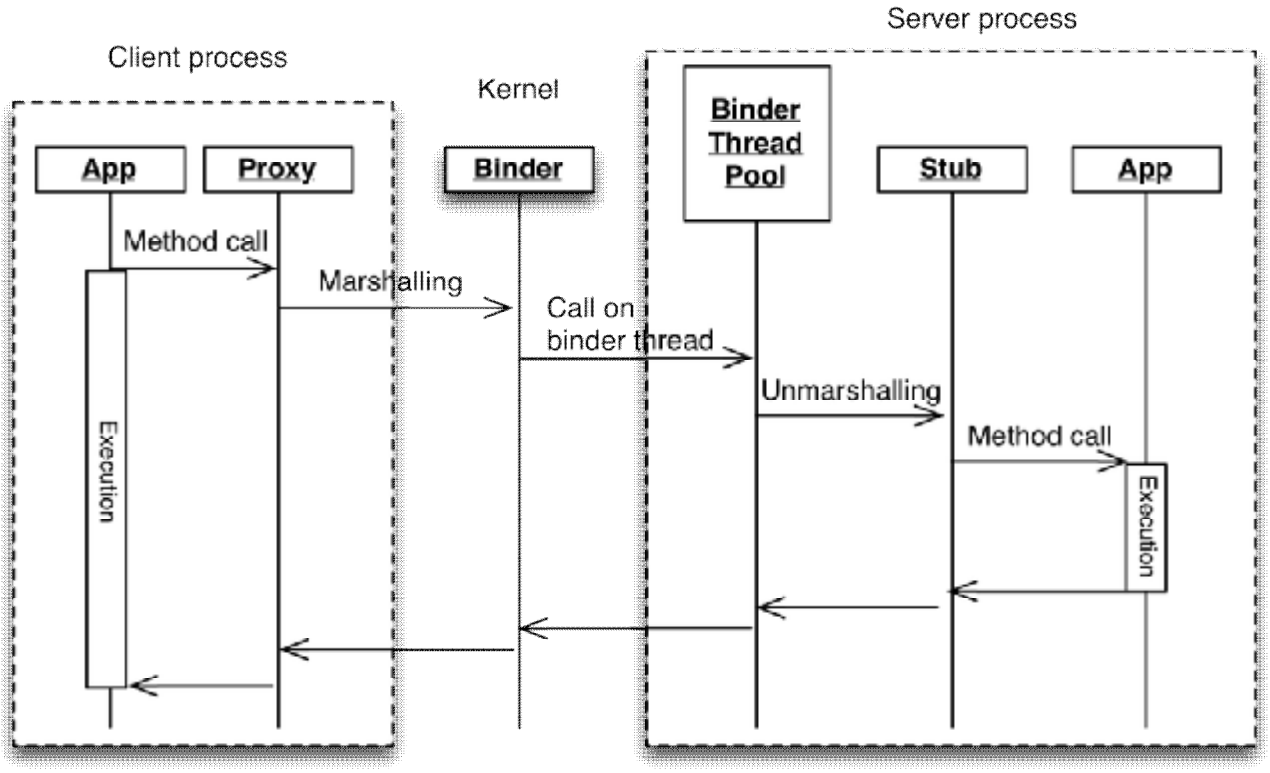


Figure 5-3. Remote procedure calls over AIDL and the generated Proxy and Stub classes

As Figure 5-3 shows, the client proxy and the server stub manages the RPC on the behalf of the two applications, allowing clients to invoke methods locally, although executed in a server process—more precisely, on binder threads belonging to a thread pool in the server. The server has to support simultaneous method invocations from multiple client processes and threads—i.e. ensure that the execution is thread safe.

Synchronous RPC

Although remote method calls are executed concurrently in the server process, the calling thread in the client process will experience it as a synchronous, i.e., blocking, call. The calling thread will resume execution when the remote execution on the binder thread has finished, and possibly has returned a value to the client.

Let us illustrate synchronous RPC, and its implications, with a conceptual code example that only returns the thread name in the remote process:

The first step is to define the interface, i.e., communication contract, in an *.aidl* file. The interface description contains method definitions that a client process can invoke in a server process:

```
interface ISynchronous {
    String getThreadNameFast();
    String getThreadNameSlow(long sleep);
    String getThreadNameBlocking();
    String getThreadNameUnblock();
}
```

The Java interface with Proxy and Stub inner classes is generated from the *aidl* tool, and the server process overrides the Stub class to implement the functionality to be supported:

```
private final ISynchronous.Stub mBinder = new ISynchronous.Stub() {

    CountDownLatch mLatch = new CountDownLatch(1);

    @Override
    public String getThreadNameFast() throws RemoteException {
        return Thread.currentThread().getName();
    }

    @Override
    public String getThreadNameSlow(long sleep) throws RemoteException {
        // Simulate a slow call
        SystemClock.sleep(sleep);
        return Thread.currentThread().getName();
    }

    @Override
    public String getThreadNameBlocking() throws RemoteException {
        mLatch.await();
        return Thread.currentThread().getName();
    }

    @Override
    public String getThreadNameUnblock() throws RemoteException {
        mLatch.countDown();
        return Thread.currentThread().getName();
    }
```

};

The method implementations all return the name of the execution thread in the server process, but after various delays. getThreadNameFast returns immediately, whereas getThreadNameSlow sleeps for a duration defined by the client, and getThreadNameBlocking blocks by waiting for a CountDownLatch to be decremented. The decrement has to be done from another thread by calling getThreadNameUnblock.

A client process with access to the Binder object of the server process can retrieve the Proxy implementation and invoke the methods that will be executed remotely:

```
ISynchronous mISynchronous = ISynchronous.Stub.asInterface(binder);
String remoteThreadName = mISynchronous.getThreadNameFast();
Log.d(TAG, "result = " + remoteThreadName);
```

The result of the remote call is printed as *result = Binder_1*, i.e. the execution occured on a binder thread in the remote process—as expected.

The Proxy, implementing the ISynchronous interface, is used to call the remote methods on binder threads. So let us look at a few ways to use RPC:

Invoking short-lived operations remotely

> Calls to mISynchronous.getThreadNameFast() return as fast as the run-time can handle the communication, and the calling client threads will block only briefly. Concurrent calls from one or multiple clients will utilize multiple binder threads if necessary, but since the implementation returns quickly, binder threads can be reused efficiently.

Invoking long-lived operations remotely

> Calls to mISynchronous.getThreadNameSlow(long sleep) execute for a configurable duration before returning a value to the client. The calling client thread will block for equally long until a return value can be retrieved.
>
> Every client call occupies one binder thread for a long time. Consequently, multiple calls may strain the binder thread pool so that it runs out of threads. In that case, the next thread that invokes one of these remote methods will get that transaction put on a binder queue and has to wait for the execution to start until there is an available binder thread.

Invoking blocking methods

> Blocking binder threads—as mISynchronous.getThreadNameBlocking illustrates—also block the client threads until the remote method can finish. If multiple client threads

invoke blocking methods in the server process concurrently, the binder thread pool may run out of threads, preventing other client threads from getting results from remote calls. If the server process has no more available binder threads due to blocking, there are no more binder threads available to wake up the blocked threads. The server then has to rely on its own internal threads to handle the wake up, otherwise the server will never handle any more incoming calls and all the client threads that are waiting for the return of the server method will block forever.

Blocked Java threads are usually interruptible, meaning that another thread can interrupt a currently blocked thread in an attempt to let it finish the execution. However, threads in a client process have no direct access to the threads in the server process, so they cannot interrupt the remote threads. Furthermore client threads that are waiting for a synchronous RPC to return cannot catch and handle an interruption.

Invoking methods with shared state

AIDL enables client processes to execute methods in the server process concurrently. The normal rules for concurrent execution apply: thread safety is the responsibility of the interface implementation. In the example code shown for our server above, mISynchronous.getThreadNameBlocking and mISynchronous.getThreadNameUnblock share a CountDownLatch, but there is nothing protecting it from being accessed from multiple threads simultaneously. Hence, a client cannot rely on getThreadNameBlocking to remain blocked until it has called getThreadNameUnblock itself. Another client may already have done this call.

Note

A client should not assume that a certain synchronous remote method invocation is short-lived, and thus safe to call from the UI thread. A server process implementation may change over time and have a negative impact on the UI thread's responsiveness. Consequently, use client worker threads to make remote method calls, unless the remote method execution is known and under your control.

Asynchronous RPC

The strength of synchronous RPC lies in its simplicity: it is easy both to understand and to implement. The simplicity, however, comes at a cost, because the calling threads are blocked. This, of course, also applies to process-local calls, but often the execution of remote calls is done in code unknown to the client developer. The amount of time that calling threads spend blocked may differ as the remote implementation changes. Hence, synchronous remote calls may have unpredictable impacts on your applications' responsiveness. Impacts on the UI thread are commonly avoided by executing all remote calls in worker threads, running asynchronously with the UI thread. However, if the server thread is blocked, the client thread

will also block indeterminately, keeping the thread and all its object references alive. This risks memory leaks, as we will see in Chapter 6.

Enter asynchronous RPC! Instead of letting the client implement its own asynchronous policy, every remote method call can be defined to execute asynchronously. The client thread initiates a transaction with asynchronous RPC and returns immediately. The Binder gives the transaction to the server process and closes the connection from the client to the server.

Asynchronous methods must return void, and must not have arguments declared out or inout. To retrieve results, the implementation will use a callback.

Asynchronous RPC is defined in AIDL with the oneway keyword. It can be applied at either the interface level or on an individual method:

Asynchronous interface

All methods are executed asynchronously.

```
oneway interface IAsynchronousInterface {
    void method1();
    void method2();
}
```

Asynchronous method

The method is executed asynchronously.

```
interface IAsynchronousInterface {
    oneway void method1();
    void method2();
}
```

The simplest form of asynchronous RPC defines a callback interface in the method call. The callback is a reverse RPC, i.e., a call from the server to the client. Thus, the callback interface is also defined in AIDL.

The following AIDL shows a simple example of asynchronous RPC, where the remote interface is defined by one method that contains a callback interface.

```
interface IAsynchronous1 {
    oneway void getThreadNameSlow(IAsynchronousCallback callback);
}
```

The implementation of the remote interface in the server process follows. At the end of the method, the result is returned in the callback method:

```
IAsynchronous1.Stub mIAsynchronous1 = new IAsynchronous1.Stub() {
    @Override
    public void getThreadNameSlow(IAsynchronousCallback callback) throws
RemoteException {
        // Simulate a slow call
        String threadName = Thread.currentThread().getName();
        SystemClock.sleep(10000);
        callback.handleResult(threadName);
    }
};
```

The callback interface is declared in AIDL as follows:

```
interface IAsynchronousCallback {
    void handleResult(String name);
}
```

And the implementation of the callback interface in the client process handles the result:

```
private IAsynchronousCallback.Stub mCallback = new IAsynchronousCallback.Stub() {

    @Override
    public void handleResult(String remoteThreadName) throws RemoteException {
        // Handle the callback
        Log.d(TAG, "remoteThreadName = " + name);
        Log.d(TAG, "currentThreadName = " + Thread.currentThread().getName());
    }
}
```

Note that both thread names—remote and current—are printed as “Binder_1”, but they belong to different binder threads, i.e., from the client and server process respectively. The asynchronous callback will be received on a binder thread. Hence, the callback implementation should ensure thread safety if it shares data with other threads in the client process.

Message-Passing Using the Binder

As we have seen in Android Message Passing, the Android platform provides a flexible kind of interthread communication through message passing. However, it requires that the threads execute in the same process, because the Message objects are located in the memory shared by the threads. If the threads execute in different processes, they do not have any common memory for sharing messages; instead, the messages have to be passed across process boundaries, using the Binder framework. For this purpose, you can use the android.os.Messenger class to send messages to a dedicated Handler in a remote process. The Messenger utilizes the Binder framework both to pass a Messenger reference to a client process and to send Message objects. The Handler is not sent across processes; instead, the Messenger acts as the intermediary.

Figure 5-4 shows the elements of message passing between processes. A Message can be sent to a thread in another process with the Messenger, but the sending process (the client) has to retrieve a Messenger reference from the receiving process (server). Hence, there are two steps:

1. Pass a Messenger reference to client processes.
2. Send a message to server process. Once the client has the Messenger reference, this step can be repeated as often as desired.

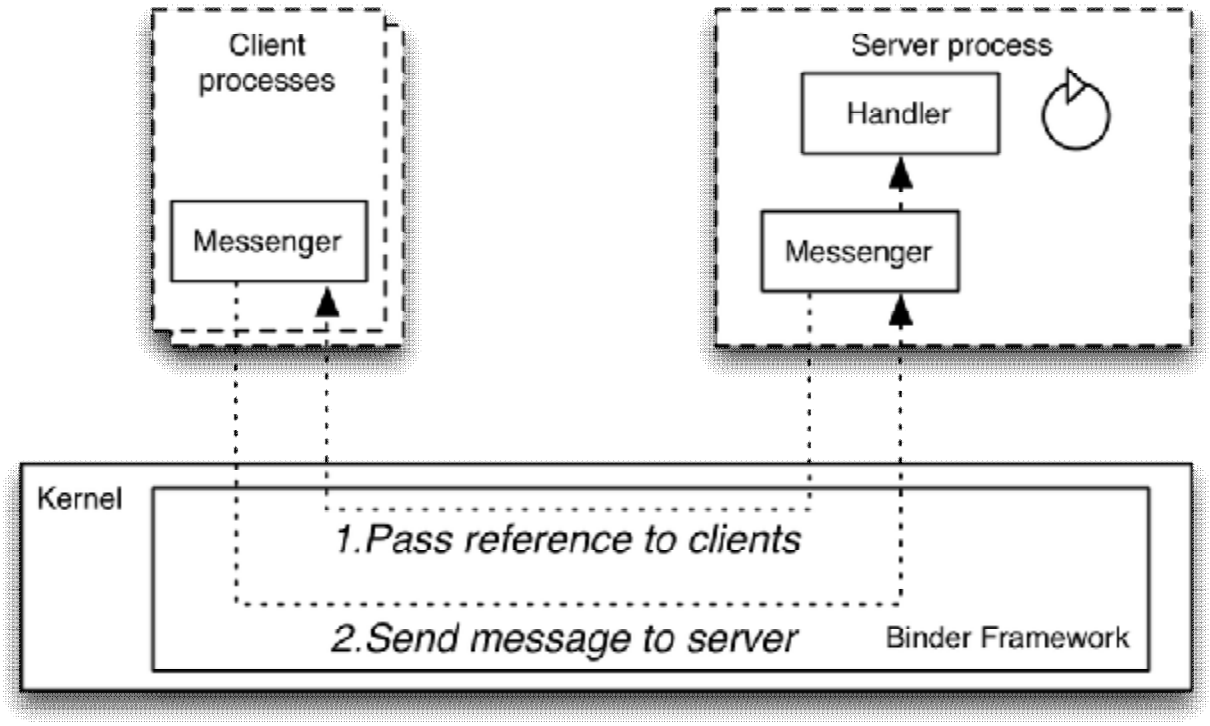


Figure 5-4. Inter-process communication with Messenger.

One-way communication

In the following example, a Service executes in the server process and communicates with an Activity in the client process. Hence, the Service implements a Messenger and passes it to the Activity, which in return can pass Message objects to the Service. First let's look into the Service:

```
public class WorkerThreadService extends Service {

    WorkerThread mWorkerThread;
    Messenger mWorkerMessenger;

    @Override
    public void onCreate() {
        super.onCreate();
        mWorkerThread.start(); ❶}

    /**
    * Worker thread has prepared a looper and handler.
    **/
    private void onWorkerPrepared() {
        mWorkerMessenger = new Messenger(mWorkerThread.mWorkerHandler); ❷}

    public IBinder onBind(Intent intent) { ❸return mWorkerMessenger.getBinder();
    }

    @Override
    public void onDestroy() {
        super.onDestroy();
        mWorkerThread.quit();
    }

    private class WorkerThread extends Thread {

        Handler mWorkerHandler;

        @Override
        public void run() {
            Looper.prepare();
            mWorkerHandler = new Handler() {
                @Override
```

```
                public void handleMessage(Message msg) { ❹// Implement message
processing
                }
            };
            onWorkerPrepared();
            Looper.loop();
        }

        public void quit() {
            mWorkerHandler.getLooper().quit();
        }
    }
}
```

❶

The messages are handled on a worker thread, which is started upon Service creation. All binding clients will use the same worker thread.

❷

A Handler to the worker thread is connected to the Messenger upon construction. This Handler will process incoming messages from client processes.

❸

The Messenger refers to a binder, which is returned to binding clients.

❹

Process incoming messages.

On the client side, an Activity binds to the Service in the server process and sends messages.

```
public class MessengerOnewayActivity extends Activity {

    private boolean mBound = false;
    private Messenger mRemoteService = null;

    private ServiceConnection mRemoteConnection = new ServiceConnection() {

        public void onServiceConnected(ComponentName className, IBinder service) {
            mRemoteService = new Messenger(service); ❶mBound = true;
```

```
        }

        public void onServiceDisconnected(ComponentName className) {
            mRemoteService = null;
            mBound = false;
        }
    };

    public void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        Intent intent = new Intent("com.wifill.eatservice.ACTION_BIND");
        bindService(intent, mRemoteConnection, Context.BIND_AUTO_CREATE); ❷}

    public void onSendClick(View v) {
        if (mBound) {
            mRemoteService.send(Message.obtain(null, 2, 0, 0)); ❸}
    }
}
```

❶

Create a Messenger instance from the binder that was passed from the server.

❷

Bind to the remote service.

❸

Send a Message when the button is clicked.

Two-way communication

A Message passed across processes keeps a reference to a Messenger in the originating process in the Message.replyTo argument, which is one of the data types that a Message can carry (refer back to Table 4-2). This reference can be used to create a two-way communication mechanism between two threads in different processes.

The following code example illustrates two-way communication between an Activity and a Service executing in different processes. The Activity sends a Message, with a replyTo argument, to a remote Service.

```
public void onSendClick(View v) {
```

```
    if (mBound) {
        try {
            Message msg = Message.obtain(null, 1, 0, 0);
            msg.replyTo = new Messenger(new Handler() { ❶@Override
                public void handleMessage(Message msg) {
                    Log.d(TAG, "Message sent back - msg.what = " + msg.what);
                }
            });
            mRemoteService.send(msg);
        } catch (RemoteException e) {
            Log.e(TAG, e.getMessage());
        }
    }
}
```

❶

Create a Messenger that is passed to the remote service. The Messenger holds a Handler reference to the current thread that executes messages from other processes.

The Service receives the Message and sends a new Message back to the Activity:

```
public void run() {
    Looper.prepare();
    mWorkerHandler = new Handler() {
        @Override
        public void handleMessage(Message msg) {
            switch (msg.what) {
                case 1:
                    try {
                        msg.replyTo.send(Message.obtain(null, msg.what, 0, 0));
                    } catch (RemoteException e) {
                        Log.e(TAG, e.getMessage());
                    }
                    break;
                }
            }
        };
        onWorkerPrepared();
        Looper.loop();
    }
}
```

Note

The Messenger is coupled with a Handler that processes messages on the thread it belongs to. Hence, task execution is sequential by design, compared to AIDL that can execute tasks concurrently on binder threads.

Summary

Most of the interprocess communication in an application is handled behind the scenes by high-level components. But where needed, you can drop down to the low-level mechanisms of the Binder: RPC and Messenger. RPC is preferred if you want to improve performance by handling incoming requests concurrently. If not, Messenger is an easier approach to implement, but its execution is single-threaded.

Chapter 11 will go into detail about the Service component, which is the most common from of interprocess communication on Android.

---

[8] See the NDK documentation in the *ndk/docs/system/libc/SYSV-IPC.html* file in your Android SDK directory.

[9] The number of threads is a platform implementation detail and may change.

[10] See the *platform-tools/aidl* file in the Android SDK directory for information about the compiler.

Chapter 6. Memory Management

Memory leaks can be extremely detrimental to your app strategy, because they not only lead to app crashes but affect the performance of the whole device and will deteriorate performance. A good deal of this chapter is devoted to background, because to prevent leaks, you need to understand how Android can have them in general, and specifically how threads and thread communication can cause them. The chapter does, however, offer strategies for managing the risk of memory leaks involved with threading, through correct design and lifecycle handling.

Memory leaks can occur on different occasions for each of the asynchronous techniques used on Android and described in Part III. So this chapter takes a general approach, and specific practical measures will be discussed when each of the asynchronous techniques is explained in detail.

Garbage collection

The Dalvik VM is a memory-managed system that frequently reclaims allocated memory with the garbage collector (GC) from the shared memory, known as the *heap*, when it grows too large. Each process—and consequently each application—has its own VM and its own garbage collector. In spite of this, an application can fill up the heap with allocated objects that cannot be reclaimed in time, which causes memory leaks.

Note

A memory leak is sometimes strictly defined as memory allocated by the application that is not used anymore, but never identified by the GC as memory that can be reclaimed. Hence, the memory is occupied until the application process is terminated. A wider definition would also include memory allocated for too long a time, essentially hogging memory. Throughout this book we use the wider definition of memory leak, because allocating memory longer than required may lead to memory exhaustion.

An application continuously creates new objects during its lifetime, and the objects are created on the heap irrespective of the scope—i.e., whether instance fields or local variables are allocated. When an object is not used anymore, the GC removes the object from the heap, freeing up the memory for new allocations. The GC can reclaim the memory when an object, or its parents, has no more strong references[11] to it. As long as an object is referenced, it is considered to be *reachable*, and not eligible for garbage collection. But once it becomes *unreachable*, the GC can finalize the object and reclaim the memory. If objects are reachable without being used anymore, the GC cannot reclaim the allocated memory. Ultimately, this

leakage can lead to the exhaustion of the application heap, causing termination of the process and a notification to the application through a *java.lang.OutOfMemoryError*.

Up until Gingerbread, garbage collection on Android executed sequentially with the application, i.e., application execution halted while memory was being reclaimed. This could lead to sudden hiccups in the UI rendering, reducing the user experience. As of Honeycomb, the GC executes concurrently on its own thread, not halting the application threads.

The Dalvik GC uses a very common two-step mechanism called *mark and sweep*. The mark step traverses the object trees and marks all objects that are not referenced by any other objects as *unused*. Unused objects become eligible for garbage collection, and the sweep step deallocates all the marked objects. An object is said to be unused if it is unreachable from any of the application's *garbage collection roots*, which are Java objects acting as starting points for the traversal of the object trees. GC roots themselves are not considered unused or eligible for garbage collection, even no other application object references them.

A small example of an object tree appears in Figure 6-1. It leads to the following dependency chain:

1. GC root A → A1 → A2
2. GC root B → B1 → B2 → B3 → B4
3. GC root B → B1 → B2 → B4

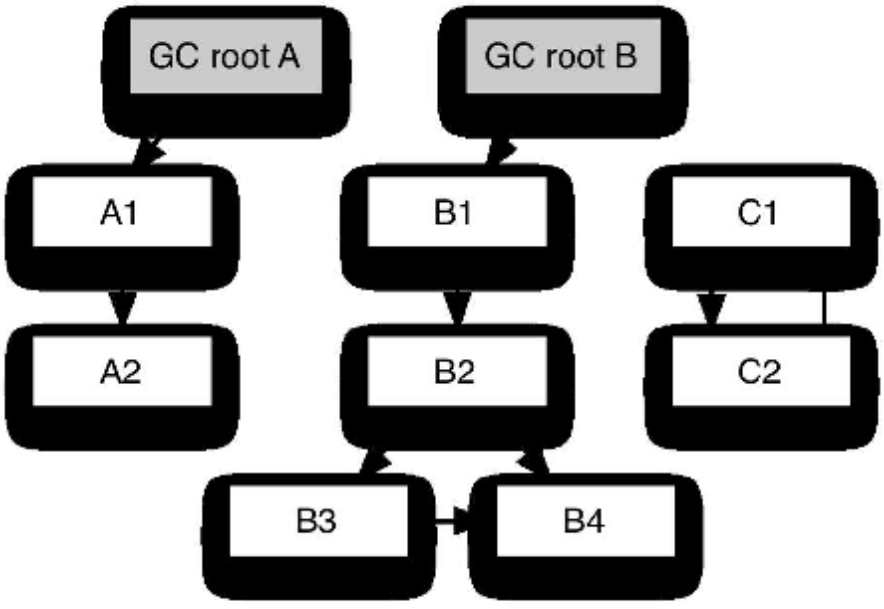


Figure 6-1. Objects traversed during the mark-step of the garbage collector

All A and B objects are linked to a GC root and will not be considered unused, because all of them are referenced by other objects and connected to a GC root. Each of the C objects has a

reference to it (from the other C object), but since neither of them connects to a GC root, they are considered to be an island of objects that can be removed. An object is unreachable once the last reference to it is removed, or (as in the case of the C objects in Figure 6-1) if none of the remaining references have any connection to a GC root.

Any object that is accessible from outside the heap is considered to be a GC root. This includes static objects, local objects on the stack, and threads. Thus, objects directly or indirectly referenced from a thread will be reachable during the execution of the thread.

Thread related memory leaks

Application threads potentially pose a risk for memory leaks, because they can hinder the garbage collector from reclaiming memory while they execute; it is only when a thread has finished the execution that its objects can be deallocated from the heap.

The application threads are the UI thread, binder threads, and worker threads, as described in Android Application Threads. The latter two thread types can be started and terminated during the application's execution. Consequently, they release their object references upon termination, making the objects unreachable if they are not reachable from another GC root. The time during which these threads are alive determines the risk of a memory leak.

There are two important characteristics of memory leaks in regard to threads:

- Potential risk

  The risk for a memory leak increases with the time a thread is alive and keeps object references. Short-lived thread instances are seldom a cause for memory leaks, but threads that are long-lived due to running long tasks, processing messages, blocking, etc., can keep references to objects that may not be required anymore, as illustrated in The lifecycle mismatch.

- Leak size

  An application that occasionally leaks a small amount of memory will probably work fine for most of the time and the leakage will pass by unnoticed. But if the leak size is large—e.g., bitmaps, view hierarchies, etc.—a few leaks may be enough to exhaust the application's heap.

In conclusion, an application shall be implemented with a low risk for memory leaks, and to keep potential leaks as small as possible. With these two characteristics in mind, the next sections will get more into detail about threads and memory leaks on Android.

Thread execution

An application that executes a task on a worker thread utilizes Thread and Runnable instances. As illustrated in Figure 6-2, both are involved in the creation of objects that obstruct garbage collection.

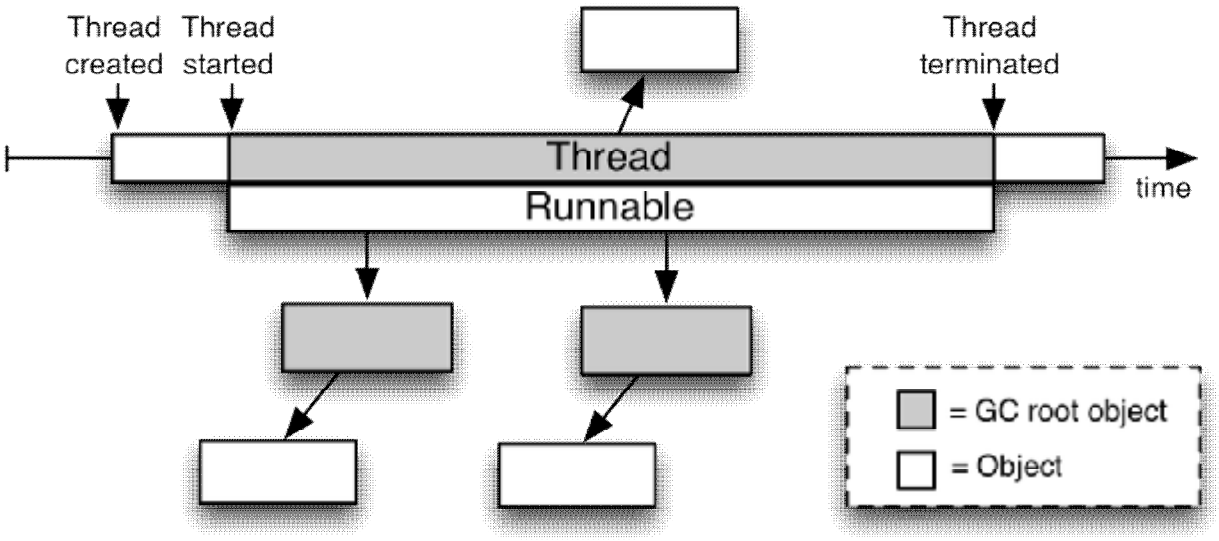


Figure 6-2. Thread lifecycle and its GC roots

The lifecycle of worker thread execution basically constitutes three steps:

1. Create a thread object.

   A Thread object is instantiated.

2. Start thread execution.

   The thread is started and the Runnable is executed.

3. Thread terminated.

   The Runnable has finished execution.

When the thread is executing, i.e. during step 2 above, the Thread object itself becomes a GC root, and all objects it references are reachable. Similarly, all objects directly referenced from an executing Runnable are GC roots. Hence, while a thread executes, both the Thread and the Runnable instance can hold references to other objects that cannot be reclaimed until the thread terminates.

Objects created in a method are eligible for garbage collection when the method returns, unless the method returns the object to its caller so that it can be referenced from other methods.

Note

Threads that execute for a long time or are blocked pose a risk for memory leaks, because they hold on to objects that will be linked to GC roots. Consequently, those objects cannot be garbage collected.

Let us illustrate the object tree for a couple of simple thread definition examples.

Inner classes

Inner classes are members of the enclosing object, i.e., the outer class, and have access to all the other members of the outer class. Hence, the inner class implicitly has a reference to the outer class (see Figure 6-3). Consequently, threads defined as inner classes keep references to the outer class, which will never be marked for garbage collection as long as thread is executing. In the following example, any objects in the Outer class must be left in memory, along with objects in the inner SampleThread class, as long as that thread is running.

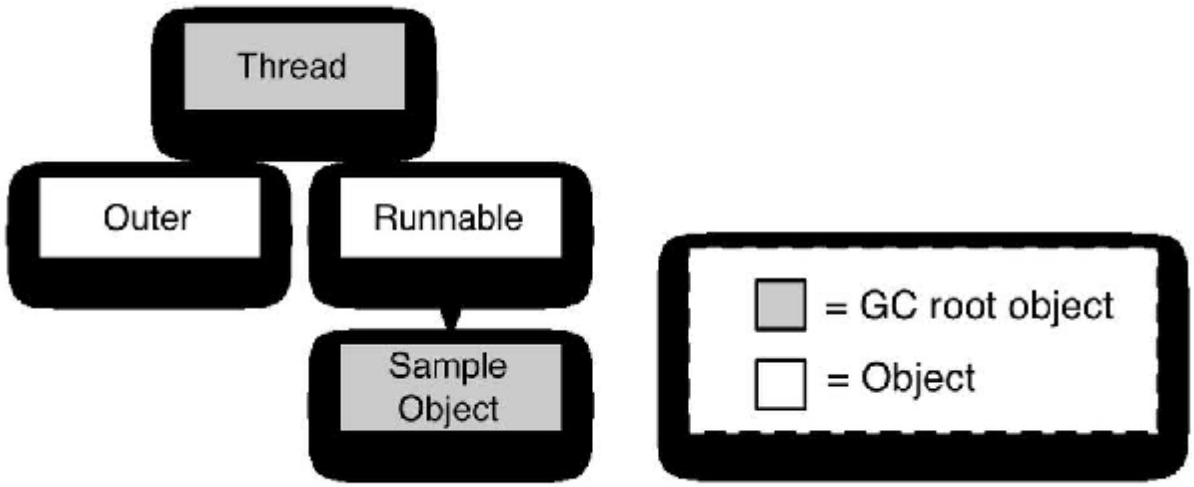


Figure 6-3. Object dependency tree with inner class thread

```
public class Outer {

    public void sampleMethod() {
        SampleThread sampleThread = new SampleThread();
        sampleThread.start();
    }
```

```
    private class SampleThread extends Thread {
        public void run() {
            Object sampleObject = new Object();

            // Do execution

        }
    }
```

Threads defined as local classes and anonymous inner classes have the same relations to the outer class as inner classes, keeping the outer class reachable from a GC root during execution.

Static inner classes

Static inner classes are members of the class instance of the enclosing object. Threads defined in a static inner class therefore keep references to the class of the outer object, but *not* to the outer object itself (Figure 6-4). Therefore, the outer object can be garbage collected once other references to it disappear. This rule, applies, for instance, in the following code.

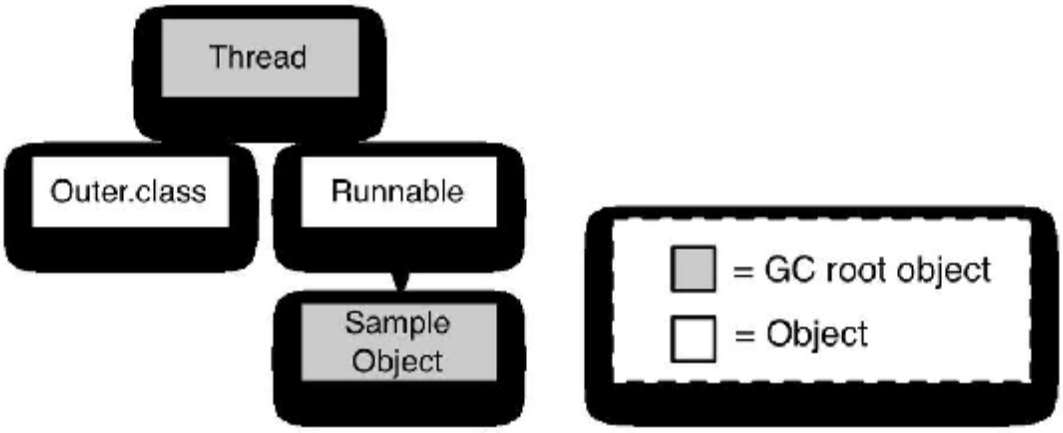


Figure 6-4. Object dependency tree with static inner class thread

```
public class Outer {

    public void sampleMethod() {
        SampleThread sampleThread = new SampleThread();
        sampleThread.start();
    }

    private static class SampleThread extends Thread {
        public void run() {
```

```
            Object sampleObject = new Object();

            // Do execution

        }
    }
}
```

However, on most occasions the programmer wants to separate the execution environment (Thread) from the task (Runnable). If you create a new Runnable as an inner class, it will hold a reference to the outer class during the execution, even if it is run by a static inner class. Code such as the following produces the situation in Figure 6-5.

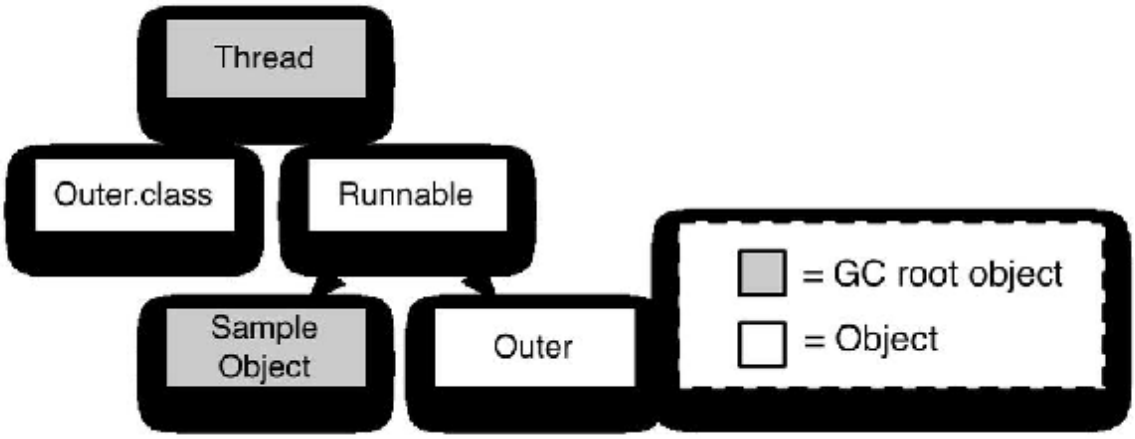


Figure 6-5. Object dependency tree with static inner class thread executing an external Runnable

```
public class Outer {

    public void sampleMethod() {

        SampleThread sampleThread = new SampleThread(new Runnable() {
            @Override
            public void run() {
                Object sampleObject = new Object();

                // Do execution

            }
        });
        sampleThread.start();
```

```
    }

    private static class SampleThread extends Thread {
        public SampleThread(Runnable runnable) {
            super(runnable);
        }
    }
}
```

The lifecycle mismatch

A fundamental reason for leakage on Android is the lifecycle mismatch between components, objects, and threads. Objects are allocated on the heap, can be eligible for garbage collection, and are kept in memory when they are referenced by threads. In Android, however, it is not only the lifecycle of the object that the application has to handle, but also the lifecycle of its components. All components—Activity, Service, BroadcastReceiver and ContentProvider—have their own lifecycles that do not comply with their objects' lifecycles.

Leaking Activity objects is the most serious—and probably the most common—component leak. An Activity holds references, for instance, to the view hierarchy that may contain a lot of heap allocations. Figure 6-6 illustrates the component and object lifecycles of an Activity.

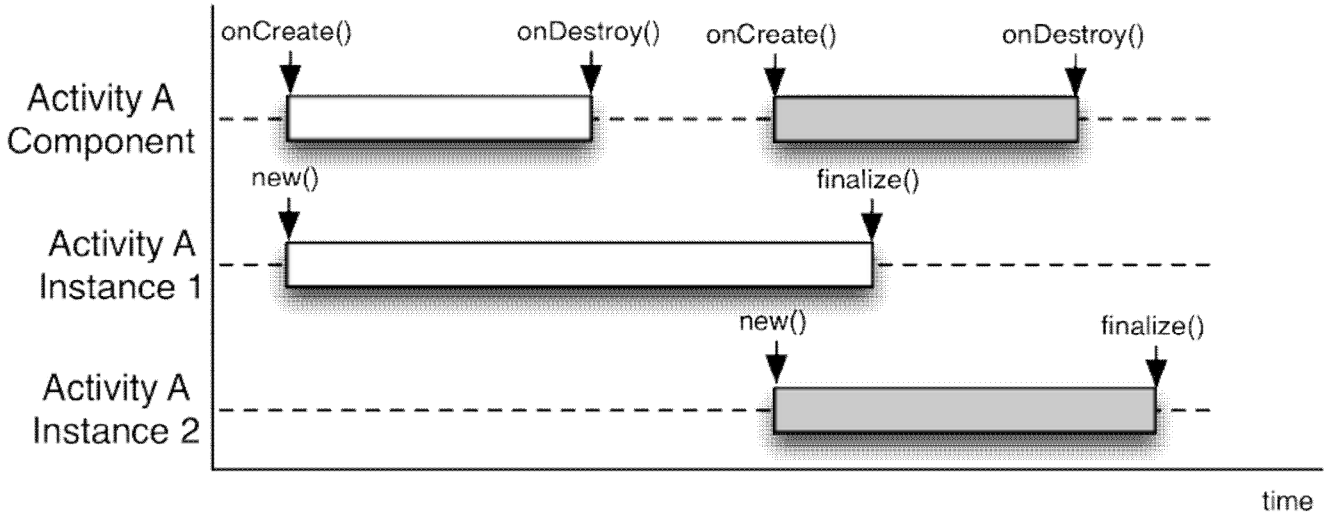


Figure 6-6. Lifecycles of Activity component and instance

The creation of an Activity component—i.e. onCreate() is called— leads to the construction of an Activity Java object. During the entire lifetime of the Activity component—from onCreate() to onDestroy()—the underlying Java object is of course in use. When the Activity object is finished or destroyed by the system, the component lifecycle ends with a call to onDestroy, which can happen for multiple reasons:

- The user presses the back-button. This implicitly finishes the activity.
- The Activity explicitly calls finish().
- The system has determined that the application process, where the Activity executes, can be killed to spare system resources.
- A configuration change occurs, e.g., the device is rotated, which destroys the Activity component and creates a new one.

When the Activity component has finished, the Activity object may still remain on the heap. It is the garbage collector that determines when the Activity object can be removed. If any references to the Activity object linger after the component is destroyed, the object remains on the heap and is not eligible for garbage collection. As Figure 6-6 illustrates, multiple Activity objects for the same Activity component can co-exist on the heap. In the figure, the component runs its course and is destroyed, but the first Activity object (Activity A) remains in memory for some time longer. In the meantime, Activity B starts, causing the component's object to be recreated. Activity B also remains in memory for some time after the component is destroyed. Activity A and B are of no use without the component, and can be considered a memory leak if they remain in memory for a long time.

Worker threads may impose a memory leak in the application, because threads can continue to execute in the background even after the component is destroyed. Thus, Figure 6-7 illustrates how an Activity object lingers on the heap long after the component has finished its lifetime. The reason is that Activity A started a worker thread that is still executing in the background. Having been created by the Activity, the thread references the Activity object.

During back and forth navigation to an Activity, as well as configuration changes, new component lifecycles are initiated, and thus a new Activity object for every lifecycle. If threads are automatically started upon creation, every new lifecycle can create a thread with a reference to the Activity object—and its view hierarchy too. This poses a risk for problematic memory leaks if the Activity re-creation cycle is shorter than the lifetime of the threads.

Note

Automatically started threads pose a higher memory leakage risk than user started threads, as configuration changes and user navigation can yield many concurrent threads with Activity object references.

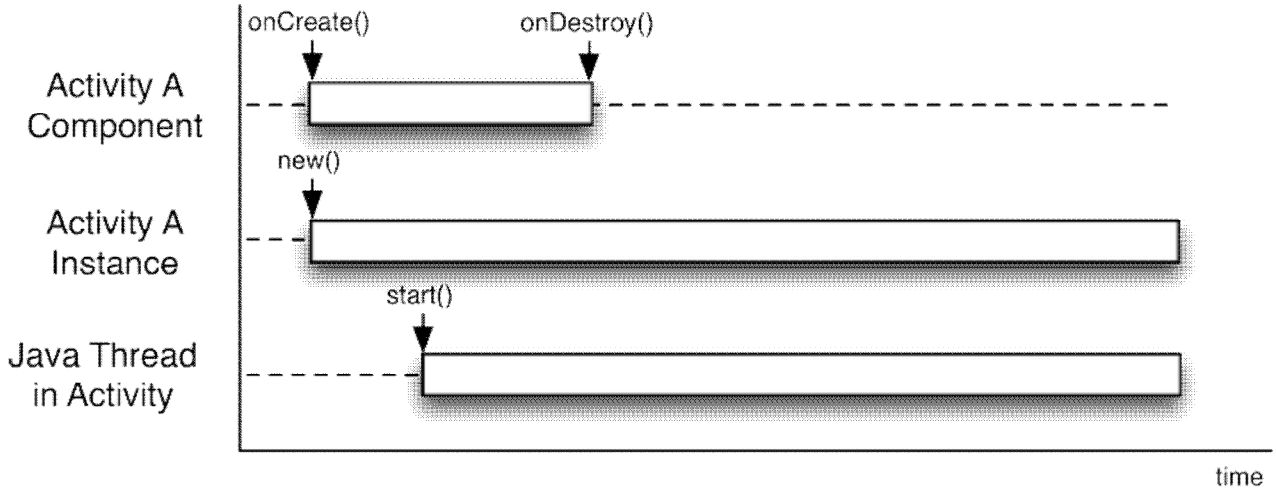


Figure 6-7. Activity lifecycle with executing thread

Thread communication

Thread execution is a potential source for memory leaks, and so is the message passing mechanism between threads. These leaks can happen whether the executor is the UI thread or another thread created by the application. For a thread to receive a Message object from another thread, it needs to have a MessageQueue to hold pending messages, a Looper to dispatch the messages, and a Handler to execute the messages, as explained in Android Message Passing. Most of these objects are referenced only from the producer thread and therefore can be garbage collected when it exits, but Handler is a candidate for memory leaks, because it is referenced from the consumer thread through a chain of objects, as shown in Figure 6-8. The Handler instance and the objects it references can not be deallocated until the thread terminates.

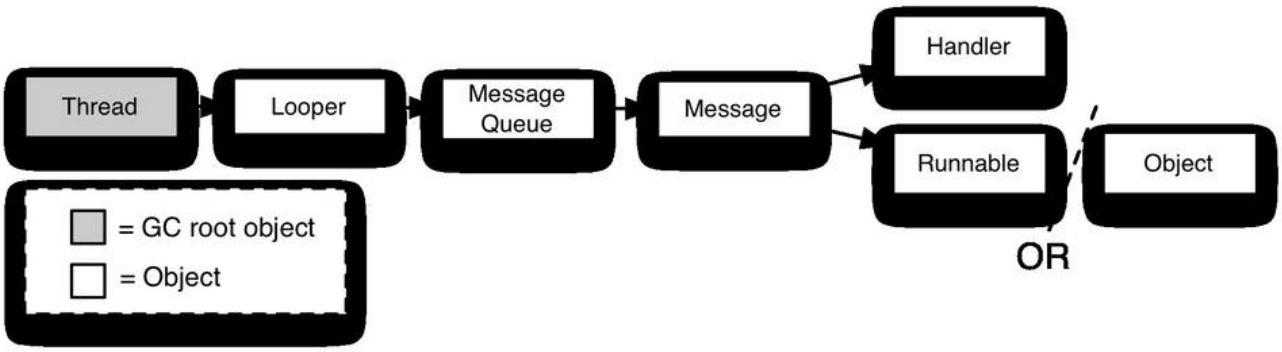


Figure 6-8. A thread that can receive messages, and the referenced objects

While the consumer thread executes, all the direct or indirect referenced objects are still reachable from a GC root and are ineligible for garbage collection. The Message instance, passed between the threads, holds references to a Handler and either to data (Object) or to a task (Runnable) (refer to Table 4-2). From the creation through the recycling of a Message, it

holds a Handler reference to the consumer thread. While the message is pending in the message queue or executed on the thread, it is ineligible for garbage collection. Furthermore, the Handler and the Object or Runnable, together with all their implicit and explicit references, are still reachable from the GC root.

The definition of Thread and Runnable as seen in Thread execution can increase the memory leak, and that also applies to Handler and Runnable during thread communication. We will look at two code examples to illustrate problems involving these two objects: when sending a data messages and when posting a runnable.

Sending a data message

Data messages can be passed in various ways; the chosen implementation determines both the risk for, and the size of, a memory leak. The following code example illustrates the implementation pitfalls. The example contains an Outer class with a Handler to process messages. The Handler is connected to the thread that creates the Outer class.

```
public class Outer {

    Handler mHandler = new Handler() {
        public void handleMessage(Message msg) {
            // Handle message
        }
    };

    public void doSend() {
        Message message = mHandler.obtainMessage();
        message.obj = new SampleObject();
        mHandler.sendMessageDelayed(message, 60 * 1000);
    }
}
```

Figure 6-9 illustrates the object reference tree in the executing thread, from the time the Message has been sent to the message queue till the time it is recycled, i.e., after the Handler has processed it. The reference chain has been shortened for brevity to cover just the key objects we want to trace.

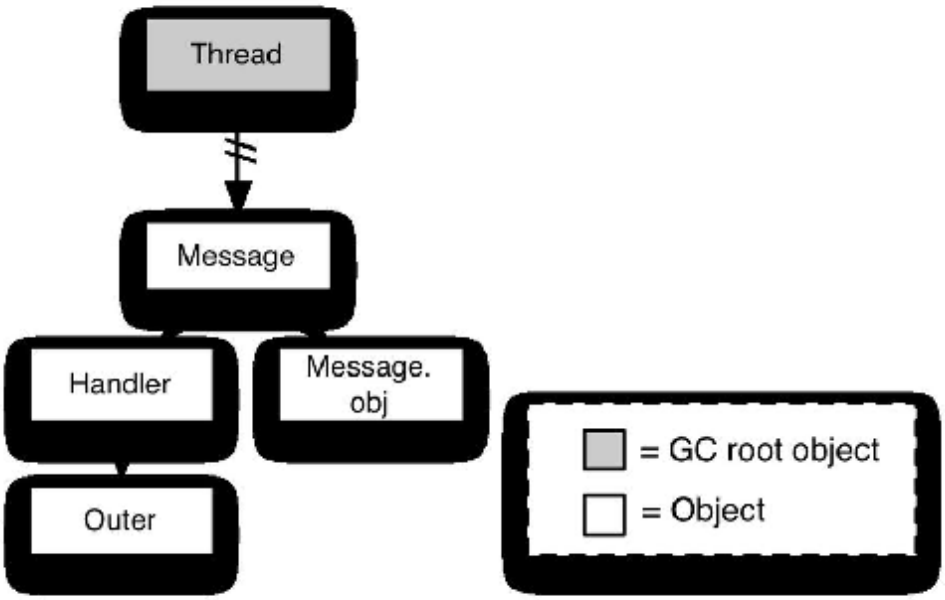


Figure 6-9. Object reference tree when sending data message.

The code example violates both memory leak characteristics: it lets a thread hold references to more objects than necessary, and it keeps the references reachable for a long time. We will look shortly at how to avoid these problems.

Posting a task message

Posting a Runnable, to be executed on a consumer Thread with a Looper, raises the same concerns as sending a Message, but with an additional extra outer class reference to watch out for:

```
public class Outer {

    Handler mHandler = new Handler() {
        public void handleMessage(Message msg) {
            // Handle message
        }
    };

    public void doPost() {
        mHandler.post(new Runnable() {
            public void run() {
                // Long running task
            }
        });
    }
```

}

This simple code example posts a Runnable to a thread that calls doPost. Both the Handler and Runnable instances refer to the outer class and increase the size of a potential memory leak, as shown in Figure 6-10.

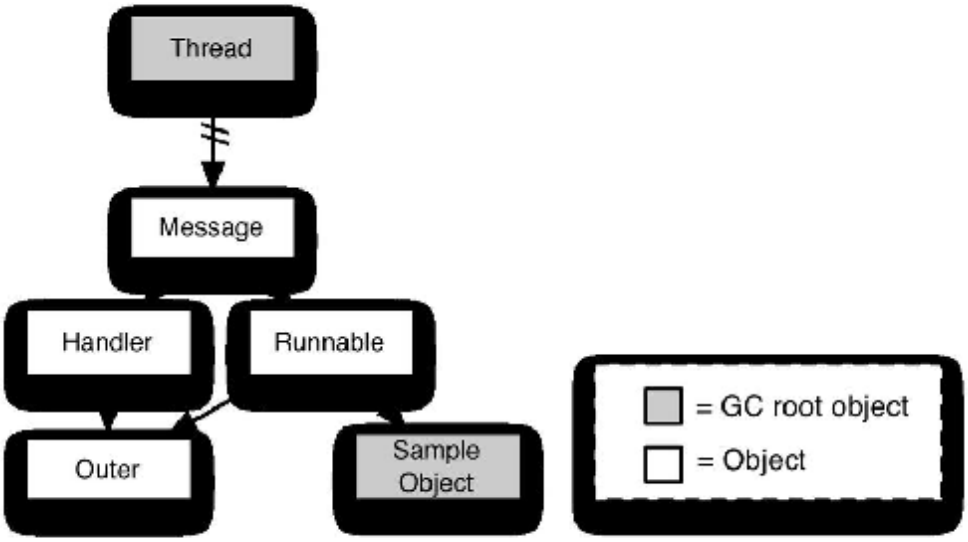


Figure 6-10. Object reference tree when posting task message

The risk for a memory leak increases with the length of the task. Short-lived tasks are better in avoiding the risk.

Note

Once a Message object is added to the message queue, the Message is indirectly referenced from the consumer thread. The longer the Message is pending, is in the queue, or does a lengthy execution on the receiving thread, the higher is the risk for a memory leak.

Avoiding memory leaks

As noted earlier, most memory leaks involving threads are caused by objects lingering in memory longer than required. Threads and handlers can—unintentionally—keep objects reachable from a thread GC root even when they are not used anymore. Let us look at how to avoid—or mitigate—these memory leaks.

Use Static Inner Classes

Local classes, inner classes, and anonymous inner classes all hold implicit references to the outer class they are declared in. Hence, they can leak not only their own objects, but also

those referenced from the outer class. Typically, an Activity and its view hierarchy can cause a major leak, through the outer class reference.

Instead of using nested classes with outer class references, it is preferred to use static inner classes, because they reference only the global class object and not the instance object. This just mitigates the leak, because all explicit references to other instance objects from the static inner class are still live while the thread executes.

Use Weak References

As we have seen, static inner classes do not have access to instance fields of the outer class. This can be a limitation if an application would like to execute a task on a worker thread and access or update an instance field of the outer instance. For this need, java.lang.ref.WeakReference comes to the rescue:

```
public class Outer {
    private int mField;
    private static class SampleThread extends Thread {

        private final WeakReference<Outer> mOuter;

        SampleThread(Outer outer) {
            mOuter = new WeakReference<Outer>(outer);
        }

        @Override
        public void run() {
            // Do execution and update outer class instance fields.
            // Check for null as the outer instance may have been GC'd.
            if (mOuter.get() != null) {
                mOuter.get().mField = 1;
            }
        }
    }
}
```

In the code example, the outer class is referenced through a *weak reference*, meaning that the static inner class holds a reference to the outer class, and can access the outer instance fields. The weak reference is not a part of the garbage collector's reference counting, as all strong references, i.e. normal references, are. Hence, if the only remaining reference to the

outer object is the weak reference from the inner class, the garbage collector sees this object as eligible for garbage collection and may de-allocate the outer instance from the heap.

Stop Worker Thread Execution

Implementing Thread, Runnable and Handler as static inner classes, nullifying explicit strong references, or using weak references will mitigate a memory leak, but not totally prevent it. The executing thread may still hold some references that cannot be garbage collected. So to prevent the thread from delaying object de-allocation, it should be terminated as soon as it is not required anymore.

Retain Worker Threads

Figure 6-7 shows how the lifecycle mismatch between component, activity, and thread can keep objects live longer than necessary. Typically, the long times are caused by changing the configurations of Activities, where the old object is kept in memory for as long as the thread executes. By retaining the thread from the old to the new Activity, and removing the thread reference from the old Activity, you can allow the Activity's object to be garbage collected. The retention technique in practice differs according to the threading mechanisms available in the platform, and is explained for each mechanism in the chapters of Part III.

Clean Up the Message Queue

A message sent to a thread may be pending in the message queue either if it is sent with a long execution delay, or if messages with a lower timestamp have not finished execution. If a message is pending when it is no longer needed, you should remove it from the message queue so that all its referenced objects can be de-allocated.

Messages sent to a worker thread can be garbage collected once the worker thread finishes, but the UI thread cannot finish until the application process terminates. Hence, cleaning up messages sent to the UI thread is a valuable way to avoid memory leaks. Both Message and Runnable can be removed from the queue:

```
removeCallbacks(Runnable r)
removeCallbacks(Runnable r, Object token)
removeCallbacksAndMessages(Object token)
removeMessages(int what)
removeMessages(int what, Object object)
```

The Runnables shall be removed with a reference to the instance, whereas the Messages can be removed with the identifiers *what* and *token*, see Message.

Summary

This chapter has summarized the main reasons for memory leaks on Android, and shown both code design and execution lifecycle considerations you should take to avoid—or at least reduce—the memory leaks.

In Part III, more practical aspects of avoiding memory leaks for the asynchronous mechanisms in the platform is discussed.

---

[11] The default object reference is a strong reference, but references can also have weaker semantics, as mentioned in Avoiding memory leaks. Refer to the java.lang.ref documentation for an overview of reference types.

Part III. Asynchronous Techniques

The simplest form of asynchronous execution on Android is the regular java.lang.Thread class, which is the basis for asynchronous behavior on Android. Other mechanisms build upon the Thread class to enable better resource control, message passing, or relief for the UI thread.

Chapter 7. Managing the Lifecycle of a Basic Thread

This chapter contains some of the basics of using threads, discusses threads in collaboration with Android components, and finishes with thread management. It covers the cancellation of tasks, how to retain threads across Activities and Fragments, and other essential techniques.

Basics

The Thread class in Android is no different from the Thread class in regular Java programming. It is the closest representation of the underlying Linux native thread an application gets. The Thread class creates the execution environment for tasks, represented by Runnable. The Thread implements Runnable, so the task to be executed is either defined by the thread itself or injected during thread creation.

Lifecycle

This section explains the observable states a thread can enter during its existence. These states are defined in the Thread.State class and are illustrated in Figure 7-1.

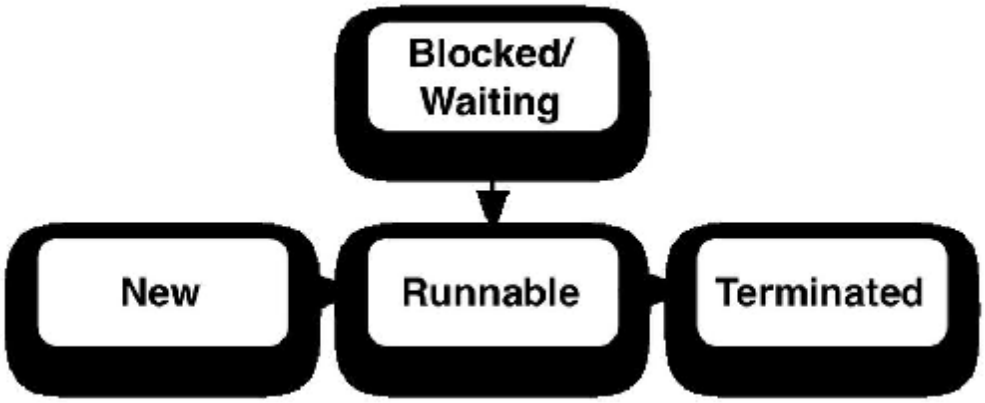


Figure 7-1. Lifecycle of a thread.

New

Before the execution of a thread, a Thread object is created. The instantiation does not set up the execution environment, so it is no heavier than any other object instantiation. The default construction assigns the newly created thread to the same thread group as the thread that is doing the creation, with the same priority. Specifically, worker threads created from the UI thread belong to the same thread group, with the same priority, as the UI thread.

Runnable

When Thread.start() is called, the execution environment is set up and the thread is ready to be executed. It is now in the Runnable state. When the scheduler selects the thread for execution, the run-method is called and the task is executed.

Blocked/Waiting

Execution can halt when the thread has to wait for a resource that is not directly accessible — e.g. an I/O operation, synchronized resources used by other threads, blocking API calls, etc. But execution can also be given up explicitly:

1. Thread.sleep() Let the thread sleep for certain amount of time and then make it available to be scheduled for execution again.
2. Thread.yield() Give up execution, and let the scheduler make a new decision which thread to execute. The scheduler selects freely what thread to execute, and there is no guarantee that the scheduler will choose a different thread.

Terminated

When the run method has finished execution, the thread is terminated and its resources can be freed. This is the final state of the thread; no re-use of the Thread instance or its execution environment is possible. Setting up and tearing down the execution environment is a heavy operation; doing it over and over again is a sign that another solution, e.g. thread pools (see Chapter 9), is preferred.

Interruptions

Occasionally, an application wants to terminate the thread's execution before it has finished its task. For instance, if a thread is taking a long time to download a video and the user presses a button to cancel the download, the UI thread captures the button press and would like to terminate the downloading thread. There is, however, no way a thread can be directly terminated. Instead, threads can be *interrupted*, which is an request to the thread that is should terminate, but it is the thread itself that determines whether to oblige or not. Interruptions are invoked on the thread reference:

```
Thread t = new SimpleThread();
t.start(); // Start the thread
```

```
t.interrupt(); // Request the the thread to terminate
```

Thread interruption is implemented collaboratively: the thread makes itself available to be interrupted, and other threads issue the call to interrupt it. Issuing an interruption has no direct impact on the execution of the thread; it merely sets an internal flag on the thread that marks it as interrupted. The interrupted thread has to check the flag itself to detect the interruption and terminate gracefully. A thread must implement *cancellation* points in order to allow other threads to interrupt it and get it to terminate.

```
public class SimpleThread extends Thread {

    @Override
    public void run() {
        while (isInterrupted() == false) {
            // Thread is alive
        }
        // Task finished and thread terminates
    }
}
```

Cancellation points are implemented by checking the interrupt flag with the isInterrupted() instance method, which returns a Boolean value of true if the thread has been interrupted, false otherwise. If it returns true, the thread is informed that it is requested to terminate. Typically, cancellation points are implemented in loops, or before long running tasks are executed, to enable the thread to skip the next part in the task execution.

The interrupt flag is also supported by most blocking methods and libraries; a thread that is currently blocked will throw an InterruptedException upon being interrupted. Hence, the thread can clean up the state of shared objects in the catch clause before the thread terminates. When an InterruptedException is thrown, the interrupted flag is reset, i.e., isInterrupted will return false even though the thread has been interrupted. This may lead to problems further up in the thread callstack, because no one will know that the thread has been interrupted. Hence, if the thread doesn't have to perform any clean-up upon interruption, the thread should pass the InterruptedException further up in the callstack. If clean-up is required, it should be done in the catch-clause, after which the thread should interrupt itself again so that callers of the executed method are aware of the interruption, as shown in the following example:

```
void myMethod() {
        try {
            // Some blocking call
```

```
        } catch (InterruptedException e) {
            // 1. Clean up
            // 2. Interrupt again
            Thread.currentThread().interrupt();
        }
    }
```

Note

Interruption state can also be checked with the Thread.interrupted() static method, which returns a Boolean value in the same way as isInterrupted(). However, Thread.interrupted() comes with a side-effect: it clears the interruption flag.

Warning

Do not use Thread.stop() to terminate an executing thread, because it can leave the shared objects in an inconsistent and unpredictable state. It has been deprecated since API Level 1, but is implemented up to API Level 10 (Gingerbread); in newer platform versions the implementation has been replaced with throwing UnsupportedOperationException.

Uncaught Exceptions

A running Java thread terminates normally when the code path reaches the end and there is no more code to execute—i.e. at the end of the Runnable.run()-method. If, somewhere along the code path, an unexpected error occurs the result may be that an unchecked exception is thrown. Unchecked exceptions are decendants of RuntimeException and they do not require mandatory handling in a try/catch clause, so they can propagate along the callstack of the thread and and when the starting point of the thread is reached: the thread terminates. To avoid that unexpected errors go unnoticed, a thread can be attached with an UncaughtExceptionHandler that is called before the thread is terminated. This handler offers a chance for the application to finish the thread gracefully, or at least make a note of the error to a network or file resource.

The UncaughtExceptionHandler interface is used by implementing the method uncaughtException and attaching it to a thread. If thread is terminated due to an unexcpected error the implementation is invoked on the terminating thread, before it terminates. The UncaughtExceptionHandler is attached to either all threads or a specific thread in the Thread class:

- Thread global handler

static void setDefaultUncaughtExceptionHandler(Thread.UncaughtExceptionHandler handler);

- Thread local handler

void setUncaughtExceptionHandler(Thread.UncaughtExceptionHandler handler);

If a thread has both a global and a local handler set, the local handler has precedence over the global handler, which will not be called.

A local UncaughtExceptionHandler can be attached to the thread instance, i.e. either with Thread.currentThread() in the executing task or—as the code listing shows–by using the thread reference itself:

```
Thread t = new Thread(new Runnable() {
    @Override
    public void run() {
        throw new RuntimeException("Unexpected error occurred");
    }
});

t.setUncaughtExceptionHandler(new Thread.UncaughtExceptionHandler() {
    @Override
    public void uncaughtException(Thread thread, Throwable throwable) {
        // A basic logging of the message.
        // Could be replaced by log to file or a network post.
        Log.d(TAG, throwable.toString());
    }
});

t.start();
```

Unhandled Exceptions on the UI Thread

The Android runtime attaches a process global UncaughtExceptionHandler when an application is started[12]. Thus, the exception handler is attached to all threads in the application, and treats unhandled exceptions equally for all threads: the process is killed.

The default behavior can be overriden either globally for all threads—including the UI thread—or locally for specific threads. Typically, the overriden behavior should only add functionality to the default runtime behavior, which is achieved by redirecting the exception to the runtime handler:

```
// Set new global handler
Thread.setDefaultUncaughtExceptionHandler(new ErrorReportExceptionHandler());

// Error handler that redirects exception to the system default handler.
public class ErrorReportExceptionHandler
    implements Thread.UncaughtExceptionHandler {

    private final Thread.UncaughtExceptionHandler defaultHandler;

    public ErrorReportExceptionHandler() {
        this.defaultHandler = Thread.getDefaultUncaughtExceptionHandler();
    }

    @Override
    public void uncaughtException(Thread thread, Throwable throwable) {
        reportErrorToFile(throwable);
        defaultHandler.uncaughtException(thread, throwable);
    }
}
```

Thread management

Each application is responsible for the threads it uses and how they are managed. An application should decide on the number of threads to use, how to reuse them, when to interrupt them and if they should be retained during a rotation change.

You can implement these qualities throughout the lifecycle of the thread, particularly at three phases: definition and start, retention, and cancellation.

Definition and start

The lifecycles of threads, components, and their respective object do not match up (see The lifecycle mismatch). The thread can outlive several component lifecycles, and keep old component objects in memory even if they are never reused. The way threads are defined and started has an impact on the both the risk and the size of a memory leak. We will now look into the most common ways of defining and starting worker threads on Android, and observe the implications of each one.

The examples are based on generalized and simplified code, with an outer class (AnyObject) and threads that are started from a method (anyMethod) called in the UI thread.

Anonymous inner class

First, we will look at the properties of an inner class. The code example utilizes an anonymous inner class, because that is the syntactically shortest form, but the same principles apply to all nested and local classes as well.

```
public class AnyObject {
    @UiThread
    public void anyMethod() {
        new Thread() {
            public void run() {
                doLongRunningTask();
            }
        }.start();
    }
}
```

Pros:

- Simple

Cons:

- Maintains a reference to the outer class.
- No thread reference available.

Public thread

A thread can be defined as a standalone class, not directly defined by the class that runs it.

```
class MyThread extends Thread {
    public void run() {
        doLongRunningTask();
    }
}

public class AnyObject {
    private MyThread mMyThread;

    @UiThread
    public void anyMethod() {
        mMyThread = new MyThread();
```

```
            mMyThread.start();
        }
}
```

Pros:

- No outer class reference.
- Thread reference available.

Cons:

- The application code base gets cluttered with thread classes.

Static inner class thread definition

Instead of defining the thread as an inner class, it can be defined as a static inner class, i.e., defined in the class-object instead of the instance.

```
public class AnyObject {
    static class MyThread extends Thread() {
        public void run() {
            doLongRunningTask();
        }
    };

    private MyThread mMyThread;

    @UiThread
    public void anyMethod() {
        mMyThread = new MyThread();
        mMyThread.start();
    }
}
```

Pros:

- No outer class reference.
- Thread reference available.

Cons:

- The application code base gets cluttered with thread classes.

Summary of options for thread definition

Inner classes have outer references that may leak larger memory chunks, a weakness avoided in both public classes and static inner classes. The anonymous inner class does not store any reference to the thread instance, which leaves the thread out of control. If there is no thread reference stored, the thread cannot be influenced by the application.

All the code examples have a problem with uncontrolled thread creation. If anyMethod can be called often, for example following a button click, the number of threads cannot be controlled. New threads will be created over and over again, using up more memory for every creation. Also, the thread reference stored in the mMyThread member variable, for the public class and the static inner class, is overwritten on every execution and not usable anymore. The application can apply logic to store thread references in lists, or make sure to start new threads only if the previously started thread is not alive anymore, but Thread may require some additional logic to constrain the number of concurrent tasks.

Thread pools (Chapter 9) or HandlerThread (Chapter 8) offers constraints on the number of executing threads.

Retention

A thread does not follow the lifecycle of an Android component that have started it, or its underlying objects (see The lifecycle mismatch). Once a thread is started, it will execute until either its run method finishes or the whole application process terminates. Therefore, the thread lifetime can outlive the component lifetime.

When the thread finishes, it may have produced a result that was meant to be used by the component, but there is no receiver available. Typically, this situation occurs on configuration changes in Activity components. The default behavior[13] is to restart the component when its configuration has changed, meaning that the original Activity object is replaced by a new one without any knowledge of the executing background thread. Only the Activity object that started the thread knows that the thread was started, so the new Activity cannot utilize the thread's result; it has to restart the thread over again to collect the data.

This can lead to unnecessary work. For example, if a worker thread is set to download a large chunk of data, and a configuration change occurs during the download, it is a waste to throw the partial result away. Instead, a better approach is to retain the thread during the configuration change and let the new Activity-object handle the thread started by the old Activity-object.

Retaining threads is done differently depending on the platform version. Before API level 13 (Honeycomb), the retention is handled in the Activity, but was simplified with the

introduction of fragments. With the Support Library, Fragment is backported to older platform versions and the previous Activity retention methods are deprecated. We will now look into both methods, starting with the older Activity variant.

Retaining a Thread in an Activity

The Activity class contains two methods for handling thread retention:

public Object onRetainNonConfigurationInstance()
: Called by the platform before a configuration change occurs, which causes the current Activity-object to be destroyed and replaced by another instance. The implementation should return any object that you want to be retained across a configuration change (e.g., a thread) and passed to the new Activity object.

public Object getLastNonConfigurationInstance()
: Called in the new Activity object to retrieve the retained object returned in onRetainNonConfigurationInstance() after a configuration change has been made. It can be called in onCreate or onStart, and returns null if the Activity is started for another reason than a configuration change.

As the ThreadRetainActivity listing shows, an alive thread can be passed across Activity objects during a configuration change. The example is simplified for brevity, e.g., it doesn't show the preservation of UI state, network operation, etc.

```
public class ThreadRetainActivity extends Activity {

    private static class MyThread extends Thread { ❶ private ThreadRetainActivity mActivity;

        public MyThread(ThreadRetainActivity activity) {
            mActivity = activity;
        }

        private void attach(ThreadRetainActivity activity) {
            mActivity = activity;
        }

        @Override
        public void run() {
            final String text = getTextFromNetwork();
            mActivity.setText(text);
```

```
        }

        // Long operation
        private String getTextFromNetwork() {
            // Do network call
        }
    }

    private static MyThread t;
    private TextView textView;

    @Override
    public void onCreate(Bundle savedInstanceState) {
        setContentView(R.layout.activity_retain_thread);
        textView = (TextView) findViewById(R.id.text_retain);

        Object retainedObject = getLastNonConfigurationInstance(); ❷if (retainedObject !=
null) {
            t = (MyThread) retainedObject;
            t.attach(this);
        }
    }

    @Override
    public Object onRetainNonConfigurationInstance() { ❸if (t != null && t.isAlive()) {
            return t;
        }
        return null;
    }

    public void onStartThread(View v) {
        t = new MyThread(this);
        t.start();
    }

    private void setText(final String text) {
        runOnUiThread(new Runnable() {
            @Override
            public void run() {
                textView.setText(text);
```

```
                }
            });
        }
}
```

❶

Worker thread declared as a static inner class to avoid outer class references. The thread contains a reference to a Activity instance. The attach method is used to set the Activity reference to the currently executing object.

❷

If there is a retained thread object, it is restored. The new Activity instance is registered to the thread.

❸

Retains an executing thread before any configuration change occurs.

Warning

Retained objects—e.g., threads—bring their references over to the next Activity. Threads declared with references to the outer class—i.e. the Activity—will stop the garbage collector from reclaiming the old Activity and its view tree, although it will never be used anymore.

Retaining a Thread in a Fragment

A Fragment normally implements part of the user interface in an Activity, but since instance retention is easier with a Fragment, the responsibility to retain Thread instances can be moved from an Activity to a Fragment. The Fragment can be added to an Activity just to handle thread retention, without containing any UI elements. In a Fragment, all that is required to retain a thread, or any other state, is to call setRetainInstance(true) in Fragment.onCreate(). The whole Fragment is then retained during a configuration change. The actual Fragment lifecycle is changed so that it does not get destroyed during configuration changes. Worker threads remain in the same Fragment instance while the platform handles the retention between the Activity and Fragment.

Let's take a look how a Fragment changes thread retainment compared to the ThreadRetainActivity listing. The Activity now refers to a Fragment instead of the worker thread:

```
public class ThreadRetainWithFragmentActivity extends Activity {
    private ThreadFragment mThreadFragment;
```

```
    private TextView mTextView;

    public void onCreate(Bundle savedInstanceState) {
        setContentView(R.layout.activity_retain_thread);
        mTextView = (TextView) findViewById(R.id.text_retain);

        FragmentManager manager = getFragmentManager(); ❶ mThreadFragment =
(ThreadFragment) manager.findFragmentByTag("threadfragment");
        if (mThreadFragment == null) {
            FragmentTransaction transaction = manager.beginTransaction();
            mThreadFragment = new ThreadFragment();
            transaction.add(mThreadFragment, "threadfragment");
            transaction.commit();
        }
    }

    // Method called to start a worker thread
    public void onStartThread(View v) {
        mThreadFragment.execute(); ❷}

    public void setText(final String text) { ❸runOnUiThread(new Runnable() {
            @Override
            public void run() {
                mTextView.setText(text);
            }
        });
    }
}
```

❶

Create the Fragment if this is the first time the Activity is started.

❷

Execution of the worker thread is delegated to the Fragment.

❸

Published method for the Fragment to set the produced worker thread result.

The Fragment defines the worker thread and starts it.

```
public class ThreadFragment extends Fragment {
    private ThreadRetainWithFragmentActivity mActivity; ❶private MyThread t;

    private class MyThread extends Thread {
        @Override
        public void run() {
            final String text = getTextFromNetwork();
            mActivity.setText(text);
        }

        // Long operation
        private String getTextFromNetwork() {
            // Simulate network operation
            SystemClock.sleep(5000);
            return "Text from network";
        }
    }

    @Override
    public void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setRetainInstance(true); ❷}

    @Override
    public void onAttach(Activity activity) {
        super.onAttach(activity);
        mActivity = (ThreadRetainWithFragmentActivity) activity;
    }

    @Override
    public void onDetach() {
        super.onDetach();
        mActivity = null;
    }

    public void execute() { ❸t = new MyThread();
        t.start();
    }
}
❶
```

Reference to the parent Activity.

❷

Retain the platform handle on configuration changes.

❸

Interface for the Activity to start the worker thread.

Tip

Worker threads that execute across the lifecycle of multiple Activities are better handled with services, (see Chapter 11 and Chapter 12).

Summary

A java.lang.Thread object represents the most fundamental abstraction in Android's underlying thread execution environment. Every started worker thread corresponds to a native Linux thread. A thread is started explicitly and terminates when the task has finished the execution. Interrupts are the only explicit way for an outsider to terminate a thread, but it is up to the thread itself to implement an interruption policy. An application should manage threads to reduce the risk and size of memory leaks, and control the starting and termination of threads. The Android platform contains many other asynchronous mechanisms built upon Thread, but application developers may still consider using simple worker threads for tasks that simply spawn worker threads to offload tasks such as network communication.

---

[12] https://android.googlesource.com/platform/frameworks/base/+/master/core/java/com/android/internal/os/RuntimeInit.java

[13] The default behavior can be overridden by android:configChanges in the *AndroidManifest.xml* file.

Chapter 8. HandlerThread: a High-Level Queueing Mechanism

Android Message Passing described background execution on a thread using a message queue and dispatch mechanism. The application explicitly coupled the message queue and the dispatch mechanism to a thread. Instead, you can use a HandlerThread, a convenient wrapper that automatically sets up the internal message passing mechanisms.

This chapter covers:

- How to use the HandlerThread
- The advantages of HandlerThread compared to manually setting up the message passing mechanism
- Use cases for the HandlerThread

Fundamentals

HandlerThread is a thread with a message queue that incorporates a Thread, a Looper, and a MessageQueue. It is constructed and started in the same way as a Thread. Once it is started, HandlerThread sets up queuing through a Looper and MessageQueue, and then waits for incoming messages to process.

```
HandlerThread handlerThread = new HandlerThread("HandlerThread");
handlerThread.start();

mHandler = new Handler(handlerThread.getLooper()) {
    @Override
    public void handleMessage(Message msg) {
        super.handleMessage(msg);
        // Process messages here
    }
};
```

There is only one queue to store messages, so execution is guaranteed to be sequential—and therefore thread-safe—but with potentially low throughput, because tasks can be delayed in the queue.

The HandlerThread sets up the Looper internally and prepares the thread for receiving messages. The internal setup gurantees that there is no race condition between creating the Looper and sending messages, which can occur in the manual setup (see Basic Message

Passing Example). The platform solves the race condition problem by making handlerThread.getLooper() a blocking call until the HandlerThread is ready to receive messages.

If additional setup is required on the HandlerThread before it starts to process messages, the application should override HandlerThread.onLooperPrepared(), which is invoked on the background thread when the Looper is prepared. The application can define any initialization code in onLooperPrepared, e.g. creating a Handler that will be associated with the HandlerThread.

Limit access to HandlerThread

A Handler can be used to pass any data message or task to the HandlerThread, but the access to the Handler can be limited by keeping it private in a subclass implementation—MyHandlerThread, in the following example—and ensuring that the Looper is not accessible. The subclass defines public methods for clients to use so that the thread itself defines the communication contract for how it should be accessed.

```
public class MyHandlerThread extends HandlerThread {

    private Handler mHandler;

    public MyHandlerThread() {
        super("MyHandlerThread", Process.THREAD_PRIORITY_BACKGROUND);
    }

    @Override
    protected void onLooperPrepared() {
        super.onLooperPrepared();
        mHandler = new Handler(getLooper()) {
            @Override
            public void handleMessage(Message msg) {
                switch(msg.what) {
                    case 1:
                        // Handle message
                        break;
                    case 2:
                        // Handle message
                        break;
                }
            }
```

```
            };
        }

        @Override
        public Looper getLooper() {
            throw new RuntimeException("Looper not public");
        }

        public void publishedMethod1() {
            mHandler.sendEmptyMessage(1);
        }
        public void publishedMethod2() {
            mHandler.sendEmptyMessage(2);
        }
}
```

Lifecycle

A running HandlerThread instance processes messages that it receives until it is terminated. A terminated HandlerThread can not be reused. To process more messages after termination, create a new instance of HandlerThread. The lifecycle can be described in a set of states:

1. Creation

   The constructor for HandlerThread takes a mandatory name argument and an optional priority for the thread:

   HandlerThread(String name)
   HandlerThread(String name, int priority)

   The name argument simplifies debugging, because the thread can be found more easily in both thread analysis and logging. The priority argument is optional and should be set with the same Linux thread priority values used in Process.setThreadPriority (see Priority). The default priority is Process.THREAD_PRIORITY_DEFAULT —the same priority as the UI thread—and can be lowered to Process.THREAD_PRIORITY_BACKGROUND to execute non-critical tasks.

2. Execution

The HandlerThread is active while it can process messages; i.e., as long as the Looper can dispatch messages to the thread. The dispatch mechanism is set up when the thread is started through HandlerThread.start, and is ready when either HandlerThread.getLooper returns or on the onLooperPrepared callback. A HandlerThread is always ready to receive messages when the Handler can be created, as getLooper blocks until the Looper is prepared.

3. Reset

The message queue can be reset, so that no more of the queued messages will be processed, but the thread remains alive and can process new messages. The reset will remove all pending messages in the queue, but not affect a message that has been dispatched and executing on the thread.

```
public void resetHandlerThread() {
    mHandler.removeCallbacksAndMessages(null);
}
```

The argument to removeCallbacksAndMessages removes the message with that specific identifier. null, shown here, removes all the messages in the queue. Further details on message removal are described in Removing Messages from the Queue.

4. Termination

A HandlerThread is terminated either with quit or quitSafely, which corresponds to the termination of the Looper (Looper termination). With quit no further messages will be dispatched to the HandlerThread, whereas quitSafely ensures that messages that have passed the dispatch barrier are processed before the thread is terminated. You can also send an interrupt to the HandlerThread to cancel the currently executing message, as explained in Interruptions.

```
public void stopHandlerThread(HandlerThread handlerThread) {
    handlerThread.quit();
    handlerThread.interrupt();
}
```

A terminated HandlerThread instance has reached its final state and can not be restarted.

A HandlerThread can also be terminated by sending a finalization task to the Handler that quits the Looper, and consequently the HandlerThread:

```
handler.post(new Runnable() {
```

```
        @Override
        public void run() {
            Looper.myLooper().quit();
        }
    });
```

The finalization tasks ensures that this will be the last executed task on this thread, once it has been dispatched by the Looper. There is, however, no guarantee that other tasks will not move ahead of the finalization task by being posted to the front of the queue through Handler.postAtFrontOfQueue (see Message insertion).

Use Cases

A HandlerThread is applicable to many background execution use cases, where sequential execution and control of the message queue is desired. This section shows a range of use cases where HandlerThread comes in handy.

Repeated Task Execution

Many Android components relieve the UI thread by executing tasks on background threads. If it is not necessary to have concurrent execution in several threads—e.g., multiple independent network requests—the HandlerThread provides a simple and efficient way to define tasks to be executed sequentially in the background. Hence, the execution setup for this situation is the UI thread—available by default—and a HandlerThread with a lifecycle that follows that of the component. Thus, HandlerThread.start is called upon the start of a component and HandlerThread.quit upon the termination of the component. In between, there is a background thread available for off-loading the UI thread.

The tasks to execute can be either pre-defined Runnable or Message instances. Both types can be configured with input data as follows:

Runnable
: Provide input data through shared member variables. (Requires synchronization to ensure correct data)

Message
: Pass data using the types shown Table 4-2.

Warning

Don’t mix long or blocking tasks with shorter tasks, because the shorter ones may be postponed unnecessarily. Instead, split execution among several HandlerThread or use an Executor (Chapter 9).

Related Tasks

Interdependent tasks—e.g., those that access shared resources such as the file system—can be executed concurrently, but they normally require synchronization to render them thread-safe and ensure uncorrupted data. The sequential execution of HandlerThread guarantees thread safety, task ordering, and lower resource consumption than the creation of multiple threads. Hence, it is useful for executing non-independent tasks.

Example: Data persistence with SharedPreferences

SharedPreferences is persistent storage for user preferences on the file system. Consequently, it should only be accessed from background threads. But file system access is not thread-safe, so a HandlerThread—with sequential execution—makes the access thread-safe without adding synchronization, whoch normally is a simpler approach. The following example shows how a HandlerThread can carry out the job.

```
public class SharedPreferencesActivity extends Activity {

    private Handler mUiHandler = new Handler() { ❶@Override
        public void handleMessage(Message msg) {
            super.handleMessage(msg);
            if (msg.what == 0) {
                Integer i = (Integer)msg.obj;
                // Use the stored value on the UI thread
            }
        }
    };

    private class SharedPreferenceHandler extends Handler { ❷private static final String KEY = "key";
        private SharedPreferences mPrefs;
        static final int READ = 1;
        static final int WRITE = 2;

        private SharedPreferenceHandler(Looper looper, SharedPreferences prefs) {
            super(looper);
            this.mPrefs = prefs;
        }

        @Override
        public void handleMessage(Message msg) {
```

```
            super.handleMessage(msg);
            switch(msg.what) {
                case READ:
                    mUiHandler.sendMessage(
                        mUiHandler.obtainMessage(0, mPrefs.getInt(KEY, 0)));
                    break;
                case WRITE:
                    SharedPreferences.Editor editor = mPrefs.edit();
                    editor.putInt(KEY, msg.arg1);
                    editor.commit();
                    break;
            }
        }
    }

    private int i;
    private HandlerThread mHandlerThread;
    private SharedPreferenceHandler mSharedPreferencesHandler;


    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        mHandlerThread = new HandlerThread("BgThread",
Process.THREAD_PRIORITY_BACKGROUND);
        mSharedPreferencesHandler = new
SharedPreferenceHandler(mHandlerThread.getLooper(),
            getSharedPreferences("LocalPrefs", MODE_PRIVATE));
        mHandlerThread.start(); ❸}

    public void onButtonClickWrite(View v) { ❹mSharedPreferencesHandler.sendMessage(
            mSharedPreferencesHandler.obtainMessage(SharedPreferenceHandler.WRITE,
i++));
    }

    public void onButtonClickRead(View v) { ❺mSharedPreferencesHandler.sendMessage(

mSharedPreferencesHandler.obtainMessage(SharedPreferenceHandler.READ));
    }
```

```
    @Override
    protected void onDestroy() {
        super.onDestroy();
        mHandlerThread.quit(); ❻}
}
```

❶

Handler to the UI thread, used by the background thread to communicate with the UI thread.

❷

Background thread Handler that reads and writes values to SharedPreferences.

❸

Start background thread when the Activity starts.

❹

Write a value to SharedPreference on button click.

❺

Read a value from SharedPreferences on button click.

❻

Terminate the background thread when the Activity is destroyed.

Task Chaining

A well-designed task executes a single contextual operation independently of other tasks. Contextual operations that should be executed on background threads in Android include retrieval of a network resource, data persistence, image processing, etc. Quite often, these types of operations are used in combination: download and persist, network data mashup, etc. HandlerThread provides an infrastructure for task chaining with some favorable properties:

- Easy setup
- Independent, reusable tasks that may be chained
- Sequential execution
- Natural decision points, where you determine whether to continue with the next task in the chain or not

- Reporting the current state
- Easy passing of data from one task to another in the task chain

The task chaining pattern is implemented in the Handler by defining tasks that are decoupled and reusable in handleMessage. The execution is controlled by the Message.what parameter; any of the tasks can be reached individually, for isolated execution, or executed consecutively through internal message passing within the Handler. Once a background task has finished, it can report the status to the UI thread, stop the task chain, or initiate a new task in the chain. Basically, every task has a natural decision point where the chain can stop or continue the execution.

Example: chained network calls

Network intensive applications commonly utilize the result from one network resource as input to a second network resource. When the first network operation finishes successfully, the call to the second network resource is made. Upon failure, the application stops the chain and terminates the background thread. During the execution, the user sees a progress dialog that can be controlled from every step in the chain: dismissed, updated, or just continuously shown until the chain has completed. This example has a HandlerThread with two chained tasks. It communicates with the UI thread through a Handler that controls the dialog seen by the user.

```
public class ChainedNetworkActivity extends Activity {

    // Dialog id
    private static final int DIALOG_LOADING = 0;

    // Message types handled on UI thread
    private static final int SHOW_LOADING = 1;
    private static final int DISMISS_LOADING = 2;

    Handler dialogHandler = new Handler() { ❶@Override
        public void handleMessage(Message msg) {
            super.handleMessage(msg);
            switch (msg.what) {
                case SHOW_LOADING:
                    showDialog(DIALOG_LOADING);
                    break;
                case DISMISS_LOADING:
                    dismissDialog(DIALOG_LOADING);
            }
```

```
        }
    };

    private class BackgroundHandler extends Handler { ❷public static final int STATE_A =
1;
        public static final int STATE_B = 2;

        public BackgroundHandler(Looper looper) {
            super(looper);
        }

        @Override
        public void handleMessage(Message msg) {
            super.handleMessage(msg);
            switch (msg.what) {
                case STATE_A: ❸dialogHandler.sendEmptyMessage(SHOW_LOADING);
                    String result = networkOperation1();
                    if (result != null) {
                        sendMessage(obtainMessage(STATE_B, result));
                    } else {
                        dialogHandler.sendEmptyMessage(DISMISS_LOADING);
                    }
                    break;
                case STATE_B: ❹networkOperation2((String) msg.obj);
                    dialogHandler.sendEmptyMessage(DISMISS_LOADING);
                    break;
            }
        }
    }

    private BackgroundHandler stateHandler;
    private HandlerThread handlerThread;

    public void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);

        // Create the HandlerThread
        handlerThread = new HandlerThread("state-handlerthread",
            Process.THREAD_PRIORITY_BACKGROUND);
        handlerThread.start();
```

```
            stateHandler = new BackgroundHandler(handlerThread.getLooper());
            // Send the first network request
            stateHandler.sendEmptyMessage(BackgroundHandler.STATE_A);

        }

        public void onDestroy() {
            handlerThread.quit();
        }
}
```

❶

DialogHandler that processes messages on the UI thread. It is used to control the dialogs shown to the user.

❷

BackgroundHandler that defines two chained network tasks, executed on a HandlerThread.

❸

The first network call, which is initiated in the onCreate method. It passes a message to the UI thread that initiates a progress dialog. When the network operation is done, the successful result is either passed on to the second task— STATE_B —or the progress dialog is dismissed.

❹

Execution of the second network operation.

Conditional task insertion

HandlerThread offers great control over the Message instances in the queue and opportunites to observe their state. These features can be used to optionally insert new tasks in the queue, depending on the pending tasks in the queue when the message is sent. Message insertion control can be fine-grained, based on identifying messages in the queue by the what parameter and an optional tag.

handler.hasMessages(int what)
handler.hasMessages(int what, Object tag)

Conditional task insertion can be used in various ways depending on the problem. A common use case is to ensure that your program does not send a message of a type that is already in the queue, because the queue should never contain more than one message of the same type at any time.

```
if (handler.hasMessages(MESSAGE_WHAT) == false) {
    handler.sendEmptyMessage(MESSAGE_WHAT);
}
```

Summary

HandlerThread provides a single-threaded, sequential task executor with fine-grained message control. It is the most fundamental form of message passing to a background thread, and it can be kept alive during a component lifecycle to provide low-resource background execution. The flexibility of message passing makes the HandlerThread a strong candidate for customizable sequential executors.

Message passing provides a powerful asynchronous mechanism, but not always the most straightforward way to provide data to tasks. As we will see in the next chapters, the platform contains higher-level components that abstract message passing to make life easier for application developers and solve common asynchronous problems. However, when it comes to having full control of the background execution, HandlerThread and message passing is there to assist you!

Chapter 9. Control Over Thread Execution Through the Executor Framework

Java's Executor Framework opens up new dimensions of control over threads and the resources they use on the system. Sometimes you want to launch as many threads as the system can handle to resolve tasks quickly; other times you want to let the system manage the number of threads; and sometimes you want to cancel threads because they are no longer needed. The Executor Framework, along with related classes, allow such things as:

- Set up pools of worker threads, and queues to control the number of tasks that can wait to be executed on these threads.
- Check the errors that caused threads to terminate abnormally.
- Wait for threads to finish and retrieve results from them.
- Execute batches of threads and retrieve their results in a fixed order.
- Launch background threads at convenient times so that results are available to the user faster.

Executor

The fundamental component of the Executor Framework is the simple Executor interface. Its main goal is to separate the creation of a task (such as a Runnable) from its execution, thus enabling the sorts of application behaviors listed at the beginning of the chapter. The interface includes just one method:

```
public interface Executor {
    void execute(Runnable command);
}
```

Despite its simplicity, the Executor is the foundation of a powerful execution environment, and is more often used than the basic Thread interface because it provides a better separation between submitting a task and its actual execution. The Executor does not execute any tasks by itself—it is merely an interface—so your implementations provide the actual execution and define how tasks will be executed. Normally, you only implement an Executor if there are special requirements. Instead—as we will se soon—there are provided Executor implementations in the platform, but first a custom implementation to grasp the concepts.

An Executor implementation in its simplest form creates a thread for every task:

Example 9-1. One thread per task executor

```
public class SimpleExecutor implements Executor {
```

```
    @Override
    public void execute(Runnable runnable) {
        new Thread(runnable).start();
    }
}
```

The SimpleExecutor provides no more functionality than creating threads as anonymous inner classes directly, so it may look superfluous, but it provides advantages nevertheless: decoupling, scalability, and reduced memory references. You can alter the implementation in the Executor without affecting the code that submits the task through execute(Runnable), and scale the number of threads that handle the tasks. Furthermore, the SimpleExecutor holds no reference to the outer class, as an anonymous inner class does, and hence reduces the memory referenced by the thread.

In short, wherever you consider using the Thread class directly, consider replacing it by an Executor to leave the application open for future changes.

Other execution behaviors that can be controlled are:

- Task queueing
- Task execution order
- Task execution type (serial or concurrent)

An example of a more elaborate Executor is shown in Example 9-2. It implements a serial task executor, which is then used in the AsyncTask. (Chapter 10 explains the implications of this executor). The SerialExecutor implements a producer-consumer pattern, where producer threads create Runnable tasks and place them in a queue. Meanwhile, consumer threads remove and process the tasks off the queue.

Example 9-2. Serial executor

```
private static class SerialExecutor implements Executor {
        final ArrayDeque<Runnable> mTasks = new ArrayDeque<Runnable>();
        Runnable mActive;

        public synchronized void execute(final Runnable r) {
            mTasks.offer(new Runnable() {
                public void run() {
                    try {
                        r.run();
                    } finally {
                        scheduleNext();
                    }
```

```
                }
            });
            if (mActive == null) {
                scheduleNext();
            }
        }

        protected synchronized void scheduleNext() {
            if ((mActive = mTasks.poll()) != null) {
                THREAD_POOL_EXECUTOR.execute(mActive);
            }
        }
    }
```

The executor applies the following execution behavior:

Task queueing
: An ArrayDeque—i.e. a double ended queue—holds all submitted tasks until they are processed by a thread.

Task execution order
: All tasks are put at the end of the double ended queue through mTasks.offer(), so the result is a FIFO ordering of the submitted tasks.

Task execution type
: Tasks are executed serially, but not necessarily on the same thread. Whenever a task has finished executing—i.e., r.run() has finished—scheduleNext() is invoked. It takes the next task from the queue and submits it to another Executor in the thread pool, where any thread can execute the task.

In short, SerialExecutor constitutes an execution environment that guarantees serial execution with the the ability to process tasks on different threads.

Warning

Changing task execution type from serial to concurrent may give the application increased performance, but it raises thread safety concerns for the tasks that must be thread-safe relative to each other.

As seen, the Executor is useful for asynchronous execution, but we seldom want to implement execution behavior from scratch. The most useful executor implementation is the thread pool, which we will look at next.

Thread Pools

A thread pool is the combination of a task queue and a set of worker threads that forms a producer-consumer setup (Pipes). The worker threads consume tasks from an ordered queue. The consuming threads can be kept alive to execute more than one task, reducing the performance overhead of setting up and tearing down threads for every task. Hence, the worker thread pool contains both active threads executing tasks, and idle threads waiting for tasks to execute.

At the end of this chapter, ExecutorCompletionService contains a complete example showing a thread pool along with some of the other features discussed in this chapter.

Predefined Thread Pools

The Executor Framework contains several types of predefined thread pools, created from the factory class Executors:

Fixed size

> The fixed size thread pool maintains a user defined number of worker threads in the pool. Terminated threads are replaced by new threads to keep the number of worker threads constant. This type of pool is created with Executors.newFixedThreadPool(n).
>
> This type of thread pool uses an unbounded LinkedBlockingQueue as task queue, meaning that the queue is allowed to grow freely as new tasks are added; producer threads do not block due to a full queue. If the number of tasks in the queue outnumbers the number of worker threads, the tasks wait for execution until there is an idle worker thread available.

Dynamic size

> The dynamic size—a.k.a. cached—thread pool creates a new thread if necessary when there is a task to process. Idle threads wait for 60 seconds for new tasks to execute and are then terminated if the task queue remains empty. Hence, the thread pool grows and shrinks with the number of tasks to execute. This type of pool is created with Executors.newCachedThreadPool().
>
> This type of pool has a SynchronousQueue as task queue, which never holds any tasks—i.e., its size is always zero. When a task is added to the queue, it is directly handed over to an idle worker thread in the pool; if there are no idle threads, a new thread will be started. No matter how many tasks are added to the executor, they will never have to wait for threads to finish executing a previously added task.

Single thread executor

This has only one worker thread to process the tasks from the queue. The tasks are executed serially and thread safety cannot be violated. This type of pool is created with Executors.newSingleThreadExecutor().

The tasks are added to an unbounded LinkedBlockingQueue. Long running tasks may pose a risk because they will delay the execution of the following tasks.

Note

Executors.newSingleExecutor() and Executors.newFixedThreadPool(1) both have one worker thread to process tasks. The difference is that a single executor always has only one worker thread, whereas a fixed thread pool actually can reconfigure the number of worker threads after creation, e.g. from one to four:

```
ExecutorService executor = Executors.newFixedThreadPool(1);
((ThreadPoolExecutor)executor).setCorePoolSize(4);
```

The reconfiguration API is accessible through the ThreadPoolExecutor class, which can be used for customizing thread pools.

Custom Thread Pools

The predefined thread pool types from Executors cover the most common scenarios, but applications can create customized thread pools. The predefined Executors thread pools are based on the ThreadPoolExecutor class, which can be used directly to create thread pool behavior in detail. This section will go into more into the details of thread pools and their customization, i.e., configuration and extension.

ThreadPoolExecutor Configuration

A thread pool's behavior is based on a set of properties concerning the threads and the task queue, which you can set to control the pool. The properties are used by the ThreadPoolExecutor to define thread creation and termination as well as the queuing of tasks.

Core pool size

The lower limit of threads that are contained in the thread pool. Actually, the thread pool starts with zero threads, but once the core pool size is reached, the number of threads does not fall below this lower limit. If a task is added to the queue when the number of worker threads in the pool is lower than the core pool size, a new thread will be created even if there are idle threads waiting for tasks. Once the number of

worker threads are equal or higher than the core pool size, new worker threads are only created if the queue is full—i.e. queuing has precedence over thread creation.

Maximum pool size

The maximum number of threads that can be executed concurrently. Tasks that are added to the queue when the maximum pool size is reached will wait in the queue until there is an idle thread available to process the task.

Maximum idle time (keep-alive time)

Idle threads are kept alive in the thread pool to be prepared for incoming tasks to process, but the alive time in idle mode can be set to avoid keeping too many unused threads in the pool. The alive time is configured in TimeUnits, the unit the time is measured in. By default, the alive time applies only to threads in excess of the core pool size. To set alive time for the core pool thread, invoke allowCoreThreadTimeOut(true).

Task queue type

An implementation of BlockingQueue (BlockingQueue) that hold tasks added by the consumer until they can be processed by a worker thread. Depending on the requirements the queuing policy can vary.

Bounded

Limited capacity. If the queue is full when a task is added, it is rejected, as described in Rejecting Tasks.

Unbounded

Unlimited capacity.

Handoff

Empty queue defined by SynchronousQueue. This queue never contains any elements. Each task that is added gets handed off to a thread immediately, and a new thread is created if other threads are busy.

The constructor configures the thread pool:

ThreadPoolExecutor executor = new ThreadPoolExecutor(int corePoolSize, int maximumPoolSize, long keepAliveTime, TimeUnit unit, BlockingQueue<Runnable> workQueue);

Once constructed, the thread pool configuration—except the work queue—can be altered at run time:

setCorePoolSize(int corePoolSize);
setKeepAliveTime(long time, TimeUnit unit);
setMaximumPoolSize(int maximumPoolSize)
Prestarting core threads

A thread pool starts with zero worker threads and creates them when needed. Thread creation is triggered when tasks are submitted to the thread pool, but if there is no submission, no worker threads are created. This can be a problem when the task queue is preloaded with tasks and no submission is done. Under those conditions, no threads are started and no tasks are executed.

Preloaded task queues can be handled by prestarting the core threads by calling either prestartAllCoreThreads() or prestartCoreThread() on the ThreadPoolExecutor instance. Then the preloaded tasks in the queue are processed.

```
BlockingQueue<Runnable> preloadedQueue = new LinkedBlockingQueue<Runnable>();
final String[] alphabet = {"Alpha","Beta", "Gamma","Delta","Epsilon","Zeta"};
for(int i = 0; i < alphabet.length; i++){
    final int j = i;
    preloadedQueue.add(new Runnable() {
        @Override
        public void run() {
            Logg.d(TAG, alphabet[j]);
        }
    });
}
ThreadPoolExecutor executor = new ThreadPoolExecutor(
    5, 10, 1, TimeUnit.SECONDS, preloadedQueue);
executor.prestartAllCoreThreads();
```

Thread configuration

The ThreadPoolExecutor defines not only the number of worker threads—and the pool's creation and termination—but also the properties of every thread. One common application behavior is to lower thread priorities so they don't compete with the UI thread.

Worker threads are configured with implementations of the ThreadFactory interface. Thread pools can define properties on the worker threads: e.g., priority, name, exception handler, etc. As example appears in Example 9-3.

Example 9-3. Fixed thread pool with customized thread properties

```
class LowPriorityThreadFactory implements ThreadFactory {
    private static int count = 1;

    public Thread newThread(Runnable r) {
        Thread t = new Thread(r);
        t.setName("LowPrio " + count++);
```

```
            t.setPriority(4);
            t.setUncaughtExceptionHandler(new Thread.UncaughtExceptionHandler()
            {
                @Override
                public void uncaughtException(Thread t, Throwable e)
                {
                    Log.d(TAG, "Thread = " + t.getName() + ", error = " + e.getMessage());
                }
            });
            return t;
        }
}
```

```
Executors.newFixedThreadPool(10, new LowPriorityThreadFactory());
```

Because thread pools often have many threads and they compete with the UI thread for execution time, it is normally a good idea to assign the worker threads a lower priority than the UI thread. (Priorities are described in Priority.) If the priority is not lowered by a custom ThreadFactory, the worker threads by default get the same priority as the UI thread.

Extending ThreadPoolExecutor

The ThreadPoolExecutor is commonly used standalone, but it can be extended to let the program track the executor or its tasks. An application can define the following methods to add actions taken each time a thread is executed:

void beforeExecute(Thread t, Runnable r)
: Executed by the run-time library just before executing a thread.

void afterExecute(Runnable r, Throwable t)
: Executed by the run-time library after a thread terminates, whether normally or through an exception.

void terminated()
: Executed by the run-time library after the thread pool is shut down and there are no more tasks executing or waiting to be executed.

The Thread and Runnable objects are passed to the first two methods; note that the order is reversed in the two methods. Track the number of ongoing tasks in the thread pool illustrates a basic custom thread pool that tracks how many tasks are currently executing in the thread pool.

Track the number of ongoing tasks in the thread pool.

```
public class TaskTrackingThreadPool extends ThreadPoolExecutor{

    private AtomicInteger mTaskCount = new AtomicInteger(0);

    public TaskTrackingThreadPool() {
        super(3, 3, 0L, TimeUnit.SECONDS, new LinkedBlockingQueue<Runnable>());
    }

    @Override
    protected void beforeExecute(Thread t, Runnable r) {
        super.beforeExecute(t, r);
        mTaskCount.getAndIncrement();
    }

    @Override
    protected void afterExecute(Runnable r, Throwable t) {
        super.afterExecute(r, t);
        mTaskCount.getAndDecrement();
    }

    public int getNbrOfTasks() {
        return mTaskCount.get();
    }
}
```

beforeExecute increments the mTaskCounter before task execution, and afterExecute decrements the counter after execution. At any point in time, an external observer can request the number of tasks currently executing through getNbrOfTasks. The worker threads and external observer threads can access the shared member variable concurrently. Hence, it is defined as an AtomicInteger to ensure thread safety.

Lifecycle

The lifecycle of a thread pool ranges from its creation to the termination of all its worker threads. The lifecycle is managed and observed through the ExecutorService interface that extends Executor, and is implemented by ThreadPoolExecutor. The internal thread pool states are shown in Figure 9-1.

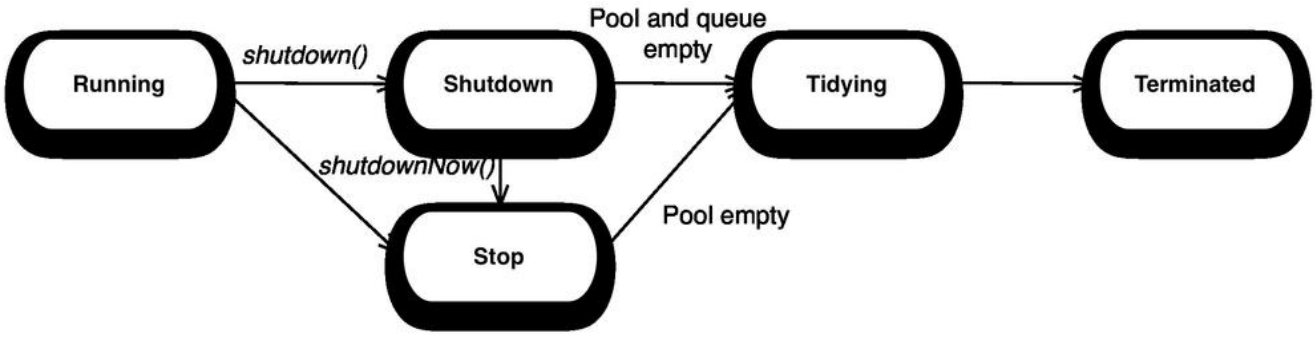


Figure 9-1. Thread pool lifecycle

Running
: The initial state of the thread pool, when it is created. It accepts incoming tasks and executes them on worker threads.

Shutdown
: The state after ExecutorService.shutdown is called. The thread pool continues to process the currently executing tasks and the tasks in the queue, but new tasks are rejected.

Stop
: The state after ExecutorService.shutdownNow is called. The worker threads are interrupted and tasks in the queue are removed.

Tidying
: Internal cleaning.

Terminated
: Final state. There are no remaining tasks or worker threads. ExecutorService.awaitTermination stops blocking, and ThreadPoolExecutor.terminated is called. After the threads finish, all data structures related to the pool are freed.

The lifecycle states are irreversible; once a thread pool has left the Running state, it has initiated the path towards termination and it can not be reused again. The only controllable transitions at that point are to the Shutdown and Stop states. The subsequent transitions depend on the processing of the tasks and occur internally in the thread pool. Consequently, the actual termination of the threads and reclaiming of memory can not be controlled without a cancellation policy—i.e. interrupt-handling—in the tasks.

Shutting Down the Thread Pool

Executors should not process tasks for longer than necessary; doing so can potentially leave a lot of active threads executing in the background for no good reason, holding on to memory that is not eligible for garbage collection. Typically, a fixed size thread pool can keep a lot of threads alive in the background. Explicit termination is required to make the executor finish. Two methods—with somewhat different impacts—are available:

void shutdown()
List<Runnable> shutdownNow()

Table 9-1 explains the different impacts of the two calls on tasks in various states. Refer to Figure 9-2 for the numbers in the table.

Table 9-1. How tasks are affected by shutdown.

Figure reference	shutdown()	shutdownNow()
1. Newly added tasks	New tasks are rejected.	New tasks are rejected.
2. Tasks pending in the queue	Pending tasks will be executed.	Pending tasks are not executed, but returned instead to a List<Runnable>, so that they potentially can be executed on other threads.
3. Tasks being processed	Processing continues.	Threads are interrupted.

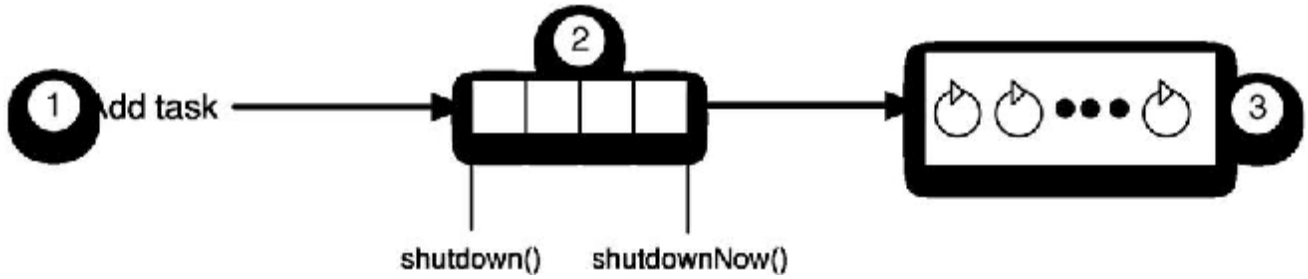


Figure 9-2. Executor shutdown

Consequently, shutdown() is considered to be a graceful termination of the executor, where both the executing and queued tasks are allowed to finish. shutdownNow() returns the queued tasks to the caller and tries to terminate currently executing tasks through interrupts. Hence, tasks should implement a cancellation policy to make them manageable. Without a cancellation policy, the tasks in the executor will terminate no earlier with shutdownNow() than with shutdown().

If the thread pool is not manually shutdown, it will automatically do so when it has no remaining threads and is not referenced by the application any more. However, threads will remain in an idle state unless the keep-alive time is set. Consequently, the automatic shutdown applies only to thread pools where all threads have a keep-alive time set, so that they terminate after a certain time. Automatic shutdown can not occur earlier than the defined keep-alive time, as threads can still linger in the pool until the timeout occurs.

Note

Once the thread pool has initiated a shutdown, it can not be reused for tasks. The application will have to create a new thread pool for subsequent tasks, or to execute tasks returned by shutdownNow().

Task management

The execution environment running tasks can also be managed. In this section we will look into the individual task and how we manage them as well.

Task representation

A task is a unit of work that needs to be executed somewhere, at some point in time. It can be executed once or repeatedly, but the task itself should be independent of how it is executed; i.e., the task and the execution environment are separate entities. We have seen tasks represented by the Runnable interface; this section introduces the more powerful Callable interface, which can manage tasks and retrieve their results with the help of the Future interface.

The Runnable interface has been around since the first version of Java. It provides a simple interface that contains one method named run, which is called when the task is processed in an execution environment.

```
public interface Runnable {
    public void run();
}
```

Callable offers a much larger set of functionalities that you need to carry out the behaviors shown in this chapter. Among these functionalities are a way to cancel a thread (if it implements an cancellation policy) and to retrieve results from a thread, including the error from a thread that terminates abnormally.

Callable defines a call method that can return a value—defined as a generic type—and throw an exception.

```
public interface Callable<V> {
    public <V> call() throws Exception;
}
```

A Callable task cannot be directly executed by Thread instances, because it was introduced first in Java 5. Instead, the execution environment should be based on ExecutorService implementations—e.g., thread pools—to process the tasks. Once a Callable task is processed

by the ExecutorService it can be observed and controlled through the Future interface, which is available after task submission (Submitting Tasks).

Methods provided by Future are:

boolean cancel(boolean mayInterruptIfRunning)
V get()
V get(long timeout, TimeUnit unit)
boolean isCancelled()
boolean isDone()

The result from an asynchronous computation is retrieved with the blocking get methods. It retrieves the result as a generic type, V, as declared in the task (e.g., Callable<V>). If the task does not end successfully by returning a result, it may have thrown a checked exception, ExecutionException, which can be caught by the caller of get.

get with no arguments blocks until the task exits, with no time limit. The two-argument version waits a limited amount of time, using the same time-out and unit shown in Extending ThreadPoolExecutor. If the task does not finish within the specified time, the calling thread stops waiting and continues to execute. The returned result is *null*.

A submitted task can be cancelled, in which case the executor tries to avoid executing it. If the task is still in the queue, it will be removed and is never executed. If it is currently executing, cancel(false) will not affect it, but cancel(true) interrupts the thread executing the task, and the task can terminate prematurely if it has implemented a cancellation policy.

isCancelled checks to see whether the task has been cancelled, but a return value true does not mean that the task will not execute—it just means someone wants this task to be cancelled. isDone returns true if the task is actually cancelled, has finished successfully, or has thrown an exception.

FutureTask

The Future has one concrete implementation class, FutureTask, to observe and manage the states of a single task. It can be executed in any execution environment—because it also implements Runnable—and it also holds the result of the task, which is retrievable from the blocking get method. In Example 9-4, a FutureTask is used to start preloading data asynchronously upon the creation of an Activity. When the user clicks a button, instead of the click launching the Activity, the data may already be available and the user gets a more responsive experience. This implementation is simplified, omitting the handling of the case where the task has not finished. To be more graceful, the application would have to show a progress bar until the task is done and then show the result.

Example 9-4. Preloading data with FutureTask

```
public class FutureTaskPreloadActivity extends Activity {

    private static FutureTask<String> future = new FutureTask(new Callable<String>() { ❶
@Override
        public String call() throws Exception {
            return doLongRunningOperation();
        }
    });

    public void onCreate(Bundle savedInstanceState) {
        new SimpleExecutor().execute(future); ❷}

    public void onClickGetData(View v) {
            String result = future.get(); ❸// Use the result.
    }
}
```

❶

Definition and creation of the FutureTask that has a Callable task.

❷

The executor that will process the task, see Example 9-1.

❸

When the user presses a button, the result is retrievable from the task.

Submitting Tasks

Before a task is submitted to a thread pool, it is by default an empty queue without threads. The state of the thread pool and queue of waiting threads determines how a thread pool can respond to a new task:

- If the core pool size has not been reached yet, a new thread can be created so the task can start immediately.
- If the core pool size has been reached but the queue has open slots, the task can be added to the queue.
- If the core pool size has been reached and the queue is full, the task must be rejected.

There are numerous ways of submitting tasks to a thread pool, both individually and batched. When there are multiple tasks to execute concurrently, they can be submitted one-by-one with the execute or submit methods. But the platform provides convenience methods that handle common use cases for batched submissions: invokeAll and invokeAny.

Individual submission

The most fundamental way to add an task to the thread pool is to build on its implementation of Executor and call the execute method.

```
ExecutorService executor = Executors.newSingleThreadExecutor();
executor.execute(new Runnable() {
    public void run() {
        doLongRunningOperation();
    }
});
```

The Executor interface can handle only Runnable tasks, but the ExecutorService extension contains more general methods allowing tasks to be submitted as instances of either Runnable or Callable. Every submitted task is represented by a Future to manage and observe the task, but only the Callable can be used for retrieving a result:

```
Callable
        ExecutorService executor = Executors.newSingleThreadExecutor();
        Future<Object> future = executor.submit(new Callable<Object>() {
            public Object call() throws Exception {
                Object object = doLongRunningOperation();
                return object;
            }
        });

        // Blocking call - Returns 'object' from the Callable
        Object result = future.get();
Runnable without result
        ExecutorService executor = Executors.newSingleThreadExecutor();
        Future<?> future = executor.submit(new Runnable() {
            public void run() {
                doLongRunningOperation();
            }
        });
```

```
// Blocking call - Always return null
Object result = future.get();
```

invokeAll

ExecutorService.InvokeAll executes mutiple independent tasks concurrently and lets the application wait for all tasks to finish by blocking the calling thread until all asynchronous computations are done or a timeout has expired.

```
List<Future<T>> invokeAll(Collection<? extends Callable<T>> tasks)
List<Future<T>> invokeAll(Collection<? extends Callable<T>> tasks,
                          long timeout, TimeUnit unit)
```

The call adds its collection of tasks to the thread pool in an ordered fashion. Depending on the thread pool definition, each added task may result in thread creation, task queuing or task rejection, similar to submit and execute. A timeout can be defined to expire when invokeAll should stop waiting for the background tasks to finish. If the timeout expires, the unfinished tasks are cancelled.

Once all tasks have finished, the results are stored in a list of futures, in the same order as the input tasks; i.e., the result is in the same position in the returned list as the task was in the input collection. After a task finishes, a call to Future.isDone returns true, and Future.get returns the task's result without blocking.

The following example utilizes invokeAll to execute two independent tasks concurrently on worker threads and combine the results when both have finished. It is typically used for retrieving network data from two different locations, where the results are mashed together before used.

The data retrieval is initiated from a button click in an Activity, i.e., on the UI thread, but because invokeAll is a blocking call it it is executed from a SimpleExecutor (Example 9-1).

Mashing data together from two network resources.

```
public class InvokeActivity extends Activity {

    @UiThread
    public void onButtonClick(View v) {

        SimpleExecutor simpleExecutor = new SimpleExecutor();❶
simpleExecutor.execute(new Runnable() {
            @Override
```

```
            public void run() {
                List<Callable<String>> tasks = new ArrayList<Callable<String>>();❷
tasks.add(new Callable<String>() {
                    public String call() throws Exception {
                        return getMockedFirstPartialDataFromNetwork();
                    }
                });
                tasks.add(new Callable<String>() {
                    public String call() throws Exception {
                        return getMockedSecondPartialDataFromNetwork();
                    }
                });

                ExecutorService executor = Executors.newFixedThreadPool(2);❸try {
                    List<Future<String>> futures = executor.invokeAll(tasks);❹String
mashedData = mashupResult(futures);❺} catch (InterruptedException e) {
                    e.printStackTrace();
                } catch (ExecutionException e) {
                    e.printStackTrace();
                }

                executor.shutdown(); ❻}
        });
    }

    private String getFirstDataFromNetwork() { /* Network call omitted */ }

    private String getSecondDataFromNetwork() { /* Network call omitted */ }

    private String mashupResult(List<Future<String>> futures) throws ExecutionException,
InterruptedException {
        for (Future<String> future : futures) {
            /* Get asynchronous computation result from future.get() */ ❼/* Mash data */
        }
        return /* Mashed data */
    }
```

❶

Executor that offloads invokeAll from the UI thread.

❷

Collection that holds s two long-running tasks, here simulated network calls.

❸

Execute the tasks on a fixed thread pool with two threads, because the example has only two tasks.

❹

Add the tasks in a batch to the thread pool. The call will block the simpleExecutor thread, but not the UI thread.

❺

The result from the two asynchronous tasks are retrieved from the respective Future instances.

❻

Shut down the thread pool to terminate the worker threads.

❼

Presumably one would issue get calls with no arguments, so each would block until the thread it is tracking ends.

invokeAny

ExecutorService.invokeAny adds a collection of tasks to an executor, returns the result from the first finished task, and disregards the rest of the tasks. This can be useful in situations where you are doing a search through many different data sets and want to stop as soon as the item is found, or any similar situation where you need results from just of the tasks you are running in parallel.

+<T> invokeAny(Collection<? extends Callable<T>> tasks)+
+<T> invokeAny(Collection<? extends Callable<T>> tasks, long timeout, TimeUnit unit)+

invokeAny blocks until one of the tasks have returned a result, and then cancels the remaining tasks in the executor by issuing future.cancel(true) on each. If, however, the remaining tasks do not respond to interruption, the tasks will continue to execute on worker threads in the background without reporting a result. Hence, lingering tasks may have side effects—e.g., changing a shared variable—even after invokeAll has returned. If the execution times out, invokeAll stops blocking but no result is returned.

Note

Tasks added with invokeAll should normally utilize a thread pool with a number of threads that can allow for concurrent execution of all tasks, without queuing. Delaying any of the tasks defeats the value of running them all to get the fastest possible result. So the minimum core pool size should not be lower than the number of tasks.

Rejecting Tasks

Task addition can fail for two reasons: because both the number of worker threads and the queue are saturated, or because the executor has initiated a shutdown.The application can customize rejection handling by providing an implementation of RejectedExecutionHandler to the thread pool. RejectedExecutionHandler is an interface with a single method that is called upon task rejection.

void rejectedExecution(Runnable r, ThreadPoolExecutor executor)

The platform provides four predefined handlers for rejected tasks, implemented as inner classes of the ThreadPoolExecutor:

AbortPolicy
: Rejects the task by throwing a RejectedExecutionException. Unless another rejection policy is defined for the thread pool, this is the default behavior.

CallerRunsPolicy
: Executes the task in the caller's thread, i.e., synchronously. This is not an alternative when long tasks are added from the UI thread.

DiscardOldestPolicy
: Removes the oldest task in the queue and inserts the rejected task again. Hence, the task first in the queue is removed and the added task is placed last in the queue.

DiscardPolicy
: Silently ignores the rejection of the task.

ExecutorCompletionService

A thread pool manages a task queue and the worker threads, but does not manage the results of the finished task. That is done by the ExecutorCompletionService. It holds a completion queue (based on a BlockingQueue) of finished tasks, as shown in Figure 9-3. When a task finishes, a Future object is placed in the queue, available to consumer threads so they can process the results in the order that the tasks have finished.

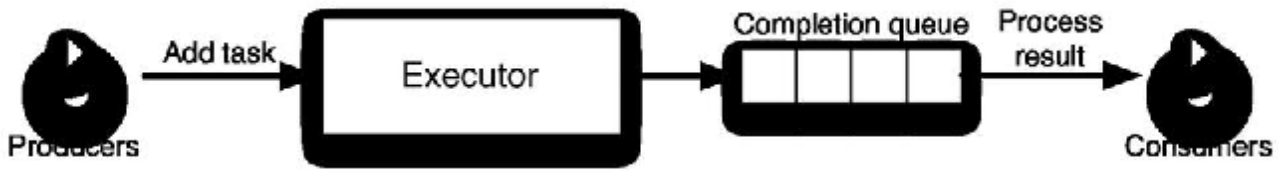


Figure 9-3. ExecutorCompletionService

Displaying multiple downloaded images in an Activity is a common use case. The UI is populated asynchronously, and a downloaded image should be displayed as soon as it is available, independently of the other image downloads. This is a job that fits well with an ExecutorCompletionService, because downladed images can be put in the completion queue, and processed as soon as they are available.

The following example illustrates the use of an ExecutorCompletionService to display downloaded images in an Activity. The image download is done with a dynamic sized thread pool of worker threads, when the Activity is created. The downloaded images are held in the completion queue and processed by a consumer thread.

```
public class ECSImageDownloaderActivity extends Activity {
    private LinearLayout layoutImages;

    private class ImageDownloadTask implements Callable<Bitmap> { ❶@Override
        public Bitmap call() throws Exception {
            return downloadRemoteImage();
        }

        private Bitmap downloadRemoteImage() {
            /* Do image download */
        }
    }

    private class DownloadCompletionService extends ExecutorCompletionService { ❷
private ExecutorService mExecutor;

        public DownloadCompletionService(ExecutorService executor) {
            super(executor);
            mExecutor = executor;
        }

        public void shutdown() {
            mExecutor.shutdown();
```

```
        }

        public boolean isTerminated() {
            return mExecutor.isTerminated();
        }
    }

    private class ConsumerThread extends Thread { ❸private DownloadCompletionService mEcs;

        private ConsumerThread(DownloadCompletionService ecs) {
            this.mEcs = ecs;
        }

        @Override
        public void run() {
            super.run();
            try {
                while(!mEcs.isTerminated()) { ❹Future<Bitmap> future = mEcs.poll(1, TimeUnit.SECONDS); ❺if (future != null) {
                        addImage(future.get());
                    }
                }
            } catch (InterruptedException e) {
                e.printStackTrace();
            } catch (ExecutionException e) {
                e.printStackTrace();
            }
        }
    }

    public void onCreate(Bundle savedInstanceState) { ❻DownloadCompletionService ecs = new DownloadCompletionService(Executors.newCachedThreadPool());
        new ConsumerThread(ecs).start();

        for (int i = 0; i < 5; i++) {
            ecs.submit(new ImageDownloadTask());
        }

        ecs.shutdown();
```

```
    }


    private void addImage(final Bitmap image) { ❼runOnUiThread(new Runnable() {
            @Override
            public void run() {
                ImageView iv = new ImageView(ECSImageDownloaderActivity.this);
                iv.setImageBitmap(image);
                layoutImages.addView(iv);
            }
        });
    }
}
```

❶

A Callable instance that represents a task producing a result. It returns a bitmap image when downloaded over a network connection.

❷

A ExecutorCompletionService that holds the Executor and exposes lifecycle methods—shutdown and isTerminated—to control the executor.

❸

A consumer thread that polls the completion queue for results from finished tasks.

❹

If the executor is terminated, all tasks have finished, and it is safe to stop polling the completion queue for more tasks. The consumer thread will finish once the executor is terminated.

❺

Polling mechanism: the consumer thread waits for one second, in every iteration, for finished tasks. After that it continues execution to check again if the executor has terminated, as described in the previous item.

❻

Create the Activity that initiates DownloadCompletionService with a cached thread pool and a ConsumerThread. Five download tasks are submitted.

❼

Shut down the executor gently; let the submitted tasks finish before the worker threads terminate.

Summary

The executor framework provides a cornerstone in the Android specific asynchronous techniques. It is often used with message passing (Android Message Passing), to create application specific asynchronous execution behaviors. If the Android-specific techniques seem limiting, the executor framework provide the developer with the full control of the asynchronous execution. It separates tasks from their execution environment, providing greater flexibility and simplifying future change to execution. Tasks designed with independence in mind can be executed in any execution environment, and if the Executor interface abstracts the actual execution, it can be easily altered without concerning the tasks. The platform's concrete implementations of Executor—thread pools—provide applications with better thread management and ways to manage concurrency in sophisticated ways. Queues and rejection policies help maintain a balance between CPU resources and tasks needing to be executed.

Chapter 10. Tying a Background task to the UI Thread with AsyncTask

The most important role for asynchronous tasks on Android, as we've discussed, is to relieve the UI thread from long-running operations. This calls for defining an execution environment, creating the task to do the long operation, and finally determining how the UI thread and the background threads communicate. All of these properties are encapsulated in an AsyncTask to make asynchronous execution as easy as it gets.

This chapter gets into the details of AsyncTask class and shows how smoothly it can handle background task execution, but also raises concerns about the pitfalls you need to watch for.

Fundamentals

As the name indicates, an AsyncTask is an asynchronous task that is executed on a background thread. The only method you need to override in the class is doInBackground(). Hence, a minimal implementation of an AsyncTask looks like this:

```
public class MinimalTask extends AsyncTask {
    @Override
    protected Object doInBackground(Object... objects) {
        // Implement task to execute on background thread.
    }
}
```

The task is executed by calling the execute method, which triggers a callback to doInBackground on a background thread.

```
new MinimalTask().execute(Object... objects);
```

When an AsyncTask finishes executing, it cannot be executed again—i.e., execute is a one-shot operation and can be called only once per AsyncTask instance, the same behavior as a Thread.

In addition to background execution, AsyncTask offers a data passing mechanism from execute to doInBackground. Objects of any type can be passed from the initiating thread to the background thread. This is like HandlerThread (Chapter 8), but with AsyncTask you do not have to be concerned about sending and processing Message instances with a Handler.

Warning

The data passed in execute to doInBackground is shared by both threads, and needs to be accessed in a mutually exclusive way. In other words, synchronization is required to protect the data from corruption.

In the common case we discussed at the beginning of the chapter, where you want to execute a task in the background and deliver a result back to the UI thread, AsyncTask shines; it is all about handling the flow of preparing the UI before executing a long task, executing the task, reporting progress of the task, and finally returning the result. All of this is available as optional callbacks to subclasses of the AsyncTask, which look like this:

```
public class FullTask extends AsyncTask<Params, Progress, Result> {
    @Override
    protected void onPreExecute() { ... }

    @Override
    protected Result doInBackground(Params... params) { ... }

    @Override
    protected void onProgressUpdate(Progress... progress) { ... }

    @Override
    protected void onPostExecute(Result result) { ... }

    @Override
    protected void onCancelled(Result result) { ... }
}
```

This implementation extends the AsyncTask and defines the arguments of the objects that are passed between threads:

Params
: Input data to the task executed in the background.

Progress
: Progress data reported from the background thread—i.e., from doInBackground—to the UI thread in onProgressUpdate.

Result
: The result produced from the background thread and sent to the UI thread.

All callback methods are executed sequentially, except onProgressUpdate, which is initiated by and runs concurrently with doInBackground. Figure 10-1 shows the lifecycle of an AsyncTask and its callback sequence.

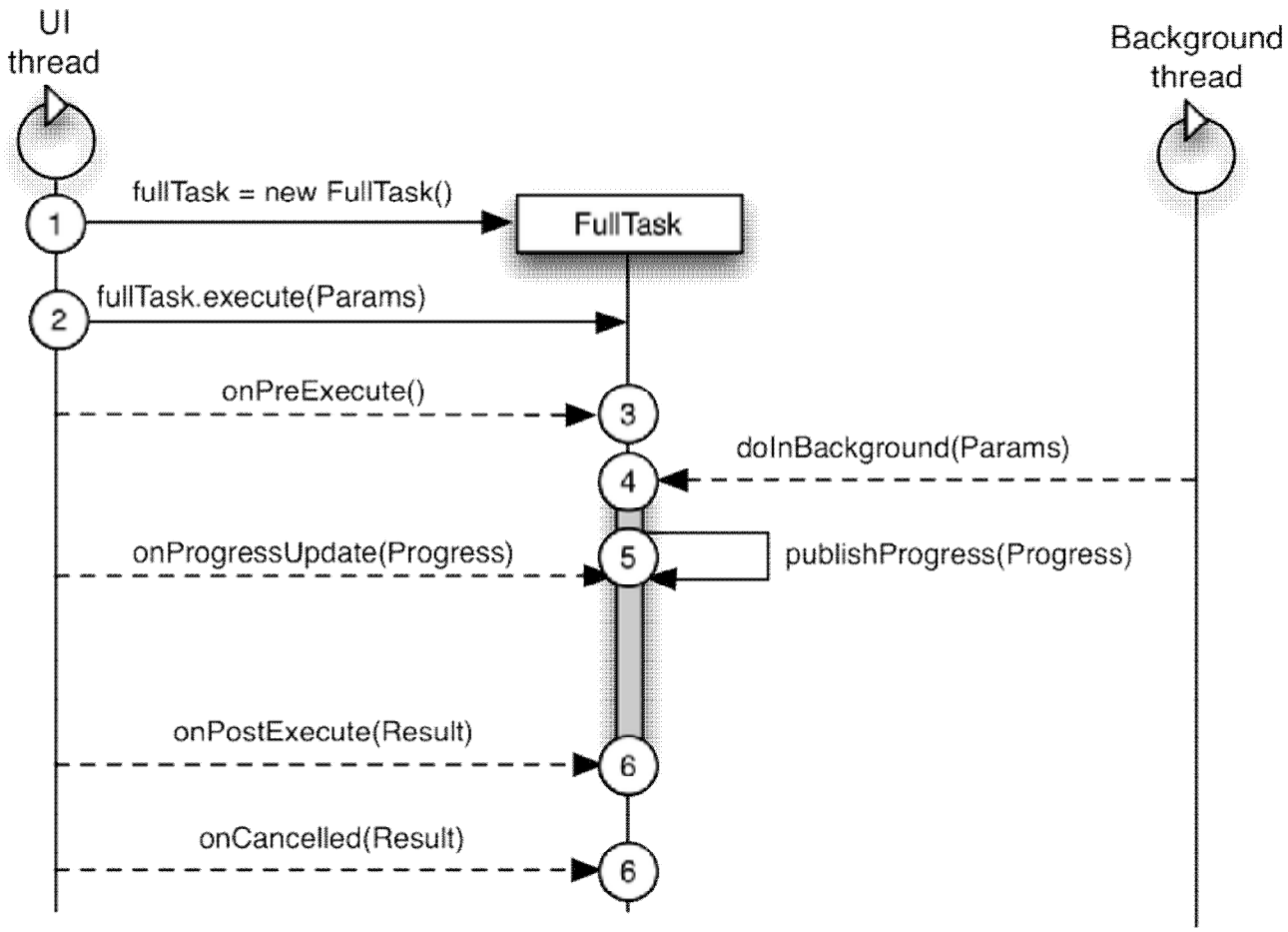


Figure 10-1. AsyncTask overview

The steps in the figure are:

1. Create the AsyncTask instance.
2. Start execution of the task.
3. First callback on the UI thread: onPreExecute. This usually prepares the UI for the long operation, e.g., by displaying a progress indicator on the screen.
4. Callback on a background thread: doInBackground. This executes the long running task.
5. Report progress updates from the publishProgress method on the background thread. These trigger the onProgressUpdate callback on the UI thread, which typically handles the update by changing a progress indicator on the screen. The progress is defined by the Progress parameter.
6. The background execution is done, and is followed by running a callback on the UI thread to report the result. There are two possible callbacks: onPostExecute is called

by default, but if the AsyncTask has been cancelled, the callback onCancelled gets the result instead. It is guaranteed that only one of the callbacks can occur.

The progress update mechanism solves two use cases:

- Displaying to the user how the long-running operation is progressing, by continuously reporting how many of the total tasks are executed.
- Delivering the result in portions, instead of delivering everything at the end in onPostExecute. For example, if the task downloads multiple images over the network, the AsyncTask does not have to wait and deliver all images to the UI thread when they are all downloaded; it can utilize publishProgress to send one image at the time to the UI thread. In that way the user gets a continuous update of the UI.

Creation and Start

AsyncTask implementations are created with the default constructor, which should be called from the UI thread. If another thread creates the AsyncTask—on platforms prior to the Jelly Bean release—the callbacks may not occur on the correct thread. The thread where the AsyncTask is created decides what thread runs the callbacks to onProgressUpdate, onPostExecute and onCancelled.

Actually, the first created AsyncTask in the application process controls the callback threads for all consecutive AsyncTask implementations in the application. The callback thread is set only once per application lifetime. Starting with Jelly Bean, an AsyncTask is class loaded at application start on the UI thread, so that the callbacks are guaranteed to occur on the UI thread.

No configuration parameters are offered by the constructor. Instead, arguments are passed to the start of the task:

execute(Object... objects);

The input consists of a variable sized object list that can receive any number of objects of arbitrary type. The input arguments are retrieved in the doInBackground callback. In this way, data is shared between the UI thread and the background thread, and is always available to both.

The execute method should be called from the UI thread; otherwise, the onPreExecute callback will not occur on the UI thread. Execution is a one-shot task; calling it more than once generates an IllegalStateException error.

Cancellation

If the UI thread decides not to use the results of an AsyncTask (perhaps because the user indicated that she changed her mind, or put the app in the background), it can send a termination request through a call to cancel(boolean).

```
// Start the task
AsyncTask task = new MyAsyncTask().execute(/* Omitted */);
// Cancel the task
task.cancel(true);
```

If the argument to the call is false, the call merely sets a flag that the background thread can check through isCancelled(). If the argument is true, an interrupt is also sent (Interruptions). Sending the interrupt is a stronger message, because blocking methods calls are released and the the background thread can catch the InterruptedException or check the Thread.isInterrupted() flag.

As usual with background tasks, it is best to terminate as early as possible if the result of the background execution cannot be used anymore. Termination releases allocated resources and reduces the risk of memory leaks. Just like a Thread, an AsyncTask cannot be forced to terminate, but requires a cancellation strategy (Interruptions) to end the execution gracefully.

When it receives a cancellation request, the task skips the call to onPostExecute and calls one of the cancel callbacks—onCancelled() or onCancelled(Result)—instead. You can use the cancel callbacks to update the UI with a different result or a different message to the user from the ones that take place when the asynchronous task finishes successfully. A cancelled task does not necessarily finish earlier than it would without cancellation, because cancellation only ensures that the onCancelled callback is called after the background task execution.

A cancellation policy involves two parts: Finish doInBackground when a blocking method throws InterruptedException, and use checkpoints in the code to see whether the task has been cancelled ¨before starting any long operation. Checkpoints can be inserted anywhere in the code, but it becomes impractical to add them everywhere in the code, so they are best used as a condition in a loop or between two long operations. The checkpoint condition can be determined either by checking AsyncTask.isCancelled or catching an interrupt. But the two checkpoint conditions responds differently to the cancel method, as shown in Table 10-1.

Table 10-1. Difference between cancellation and interruption check.

cancel(boolean)	isCancelled() returns	Thread.currentThread().isInterrupted() returns
false	true	false

cancel(boolean)	isCancelled() returns	Thread.currentThread().isInterrupted() returns
true	true	true

The strongest checkpoint condition is isCancelled, because it observes the actual call to cancel and not the interruption. Hence, the strongest cancellation policy is to combine checkpoints and interrupt-handling as follows:

```
public class InterruptionTask extends AsyncTask<String, Void, Void> {
    @Override
    protected Void doInBackground(String... s) {
        try {
            while (!isCancelled()) {
                doLongInterruptibleOperation(s[0]);
            }
        } catch (InterruptedException iex) {
            // Do nothing. Let's just finish.
        }
        return null;
    }
}
```

States

An AsyncTask has the following possible states: PENDING, RUNNING and FINISHED, in that order.

Pending
: The AsyncTask instance is created, but execute has not been called on it.

Running
: Execution has started; i.e., execute is called. The task remains in this state when it finishes, so long as its final method (such as onPostExecute) is still running.

Finished
: Both the background execution and the optional final operation—onPostExecute or onCancelled—is done.

Backward transitions are not possible, and once the task is in RUNNING state, it is not possible to start any new executions. FINISH is a terminal state; a new AsyncTask instance must be created for every execution.

The state of the AsyncTask can be observed with AsyncTask.getStatus() and its useful for determining whether a task is currently executing, as shown in the following example.

Example: Limiting an AsyncTask execution to one at the time

If an AsyncTask should not be allowed to execute while another task is executing, you can store a reference to the task and check the task's status before a new execution is allowed. In the AsyncTaskStatusActivity, the AsyncTask is started from the onExecute method, which could be called from anywhere on the UI thread, e.g., from the onClick() method triggered by a button click.

```
public class AsyncTaskStatusActivity extends Activity {

    private AsyncTask mMyAsyncTask;

    // Activity lifecycle code omitted.

    public void onExecute(View v) {
        if (mMyAsyncTask != null && mMyAsyncTask.getStatus() !=
AsyncTask.Status.RUNNING) {
            mMyAsyncTask = new MyAsyncTask().execute();
        }
    }

    private static class MyAsyncTask extends AsyncTask<String, Void, Void> {
        @Override
        protected Void doInBackground(String... s) {
            // Details omitted.
            return null;
        }
    }
}
```

Implementing the AsyncTask

Implementing an AsyncTask is straightforward: create a subclass, override doInBackground for background execution, and add helper methods for any desired UI updates before, during, and after the background execution. Despite its simplicity, there are a few more things to consider:

Avoid memory leaks

As long as the worker thread is alive, all of its referenced objects are held in memory, as explained in Thread related memory leaks. The AsyncTask should therefore be

declared as a standalone or static inner class so that the worker thread avoids implicit references to outer classes.

Coupling with a Context and its lifecycle

An AsyncTask usually issues updates to the UI thread that may reference the Context, typically an Activity with a view hierarchy. But you should avoid referencing the view hierarchy, so that the views won't be kept in memory when they are not needed. Hence, the AsyncTask should be declared as a static inner class that holds a reference to the associated Context. The reference is removed by setting it to null when it is not needed anymore.

Cancellation policy

Allow tasks to be interrupted and cancelled, as explained in Cancellation and Interruptions.

Example: Downloading Images

This example shows the implementation of an Activity that displays four images that are downloaded from the network with an AsyncTask. The UI consists of a determinate progress bar—an mProgressBar—that shows the number of downloaded images) and a layout—using mLayoutImages—whose children constitute the downloaded images. The download starts upon Activity creation and is cancelled on destruction..

```
public class FileDownloadActivity extends Activity {

    private static final String[] DOWNLOAD_URLS = { ❶
"http://developer.android.com/design/media/devices_displays_density@2x.png",

"http://developer.android.com/design/media/iconography_launcher_example2.png",

"http://developer.android.com/design/media/iconography_actionbar_focal.png",

"http://developer.android.com/design/media/iconography_actionbar_colors.png"
            };

    DownloadTask mFileDownloaderTask;

    // Views from layout file
    ProgressBar mProgressBar;
    LinearLayout mLayoutImages;

    @Override
    public void onCreate(Bundle savedInstanceState) {
```

```
        super.onCreate(savedInstanceState);

        setContentView(R.layout.activity_file_download);
        mProgressBar = (ProgressBar) findViewById(R.id.progress_bar);
        mProgressBar.setMax(DOWNLOAD_URLS.length);
        mLayoutImages = (LinearLayout) findViewById(R.id.layout_images);

        mFileDownloaderTask = new DownloadTask(this);
        mFileDownloaderTask.execute(DOWNLOAD_URLS); ❷}

    @Override
    protected void onDestroy() { ❸super.onDestroy();
        mFileDownloaderTask.setActivity(null);
        mFileDownloaderTask.cancel(true);
    }

    private static class DownloadTask extends AsyncTask<String, Bitmap, Void> { ❹private
FileDownloadActivity mActivity; ❺private int mCount = 0;

        public DownloadTask(FileDownloadActivity activity) {
            mActivity = activity;
        }

        public void setActivity(FileDownloadActivity activity) { ❻mActivity = activity;
        }

        @Override
        protected void onPreExecute() {
            super.onPreExecute();
            mActivity.mProgressBar.setVisibility(View.VISIBLE);                    ❼
mActivity.mProgressBar.setProgress(0);
        }

        @Override
        protected Void doInBackground(String... urls) {
            for (String url : urls) {
                if (!isCancelled()) { ❽ Bitmap bitmap = downloadFile(url); ❾
publishProgress(bitmap); ❿}
            }
            return null;
```

```
    }

    @Override
    protected void onProgressUpdate(Bitmap... bitmaps) { ⓫
super.onProgressUpdate(bitmaps);
        if (mActivity != null)  {
            mActivity.mProgressBar.setProgress(++mCount);
            ImageView iv = new ImageView(mActivity);
            iv.setImageBitmap(bitmaps[0]);
            mActivity.mLayoutImages.addView(iv);
        }
    }

    @Override
    protected void onPostExecute(Void aVoid) {
        super.onPostExecute(aVoid);
        mActivity.mProgressBar.setVisibility(View.GONE); ⓬}

    @Override
    protected void onCancelled() {
        super.onCancelled();
        if (mActivity != null) {
            mActivity.mProgressBar.setVisibility(View.GONE); ⓭}
    }


    private Bitmap downloadFile(String url) {
        Bitmap bitmap = null;
        try {
            bitmap = BitmapFactory
                    .decodeStream((InputStream) new URL(url)
                            .getContent());
        } catch (MalformedURLException e) {
            e.printStackTrace();
        } catch (IOException e) {
            e.printStackTrace();
        }
        return bitmap;
    }
```

```
    }
}
```

❶

URLs of the images.

❷

Pass the URLs to doInBackground.

❸

When the Activity is destroyed, the AsyncTask is cancelled and its reference to the destroyed Activity is nullified so that it can be garbage collected, although the worker thread is alive.

❹

Definition of AsyncTask: An array of String objects is passed to doInBackground, and Bitmap objects are returned during background execution. The background thread does not pass a result to the UI thread, so the last parameter is declared Void.

❺

The Activity is referenced for updates on the UI thread.

❻

Setter for changing or nullifying the Activity reference. The Activity is changed upon task retention during a configuration change.

❼

Show the progress bar before the background task executes.

❽

Check whether the task is cancelled before starting the download of the next image, so that the task can terminate as early as possible.

❾

Download the Image over the network and assign it to the Bitmap argument passed to the AsyncTask.

⑩

Send the image to the UI thread.

⑪

Respond to the progress update sent by the background thread by updating the progress bar and displaying the new image.

⑫⑬

Remove the progress bar.

Background Task Execution

Because AsyncTask executes its tasks asynchronously, multiple tasks can be executed either sequentially or concurrently. The execution environment can be defined explicitly in the application; otherwise it is set implicitly by the platform. The method that starts the execution determines how the task is executed. Table 10-2 shows the possibilities.

Table 10-2. Overview of task execution.

Execution method	Lowest API level	Return value	Executor	Background task	Execution type
execute(Params...)	3	The started AsyncTask	Defined in AsyncTask	doInBackground	Sequential or concurrent
(static) execute(Runnable)	11	None	Defined in AsyncTask	Runnable	Sequential only
executeOnExecutor(Executor, Params...)	11	The started AsyncTask	Customizable	doInBackground	Customizable

Before API level 11, only the first option in the table is available. From API level 11 onward, the AsyncTask offers three methods for task execution, with different properties:

execute(Params...)
: The version described above and the only method available on all platform versions. It utilizes the AsyncTask internal execution environment, but this has changed during platform evolution. See Execution Across Platform Versions.

execute(Runnable)

Added in API level 11 for executing Runnable tasks instead of overriding doInBackground. The Runnable is processed in the AsyncTask internal execution environment, but does not use message passing to communicate between threads. onPreExecute, onPostExecute, and onCancelled are not called and progress can not be published. This use case should probably be replaced with a different solution (see Using execute(Runnable)).

executeOnExecutor(Executor, Params...)

Added in API level 11 for configuring the actual execution environment on which the task is processed. It can utilize internal execution environments or use a custom Executor.

The Executor argument executeOnExecutor can be one of the following execution environments:

AsyncTask.THREAD_POOL_EXECUTOR

Tasks are processed concurrently in a pool of threads, with a core pool size of five threads and a maximum limit of 128 threads. The task queue is limited to 10 elements. Hence, the AsyncTask can handle 138 tasks before it starts rejecting tasks.

AsyncTask.SERIAL_EXECUTOR

A sequential task scheduler that ensures thread safe task execution. It contains no threads of its own, relying instead on THREAD_POOL_EXECUTOR for execution. It stores the tasks in an unbounded queue and passes each one to the THREAD_POOL_EXECUTOR to be executed in sequence. The tasks can be executed in different threads of the thread pool, but the SERIAL_EXECUTOR guarantees that consecutive tasks are not added to the thread pool until the previous task has finished, so thread safety is preserved.

Figure 10-2 summarizes how these two execution environments operate and use the thread pool.

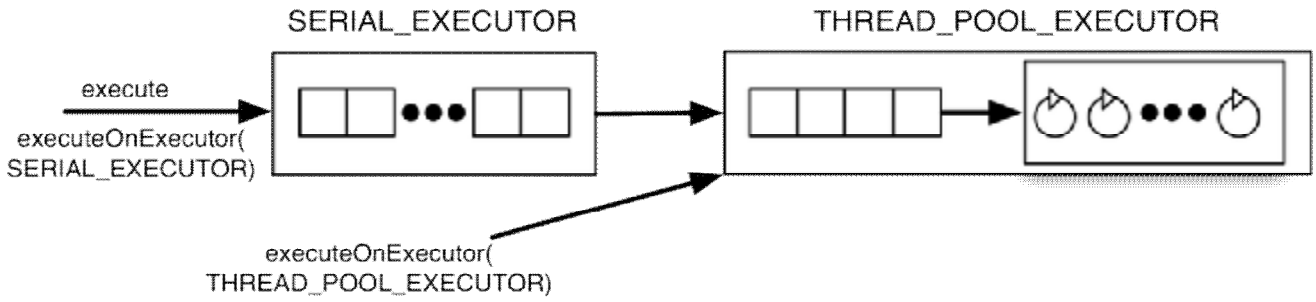


Figure 10-2. AsyncTask Execution

Both execution environments use the AsyncTask worker threads for the doInBackground callbacks. The threads have no Looper attached, so the AsyncTask cannot receive messages

from other threads. Furthermore, the threads' priority is lowered to Process.THREAD_PRIORITY_BACKGROUND, so that it will interfere less with the UI thread (priorities are described in Priority).

Application Global Execution

AsyncTask implementations can be defined and executed from any component in the application, and several instances in the RUNNING state can co-exist. However, all AsyncTask instances shared an application-wide, global execution property (Figure 10-3). That means that even if two different threads launch two different tasks (As in the following example) at the same time, they will be executed sequentially. Whichever happens to be executed first by the run-time environment will keep the other from executing till it terminates.

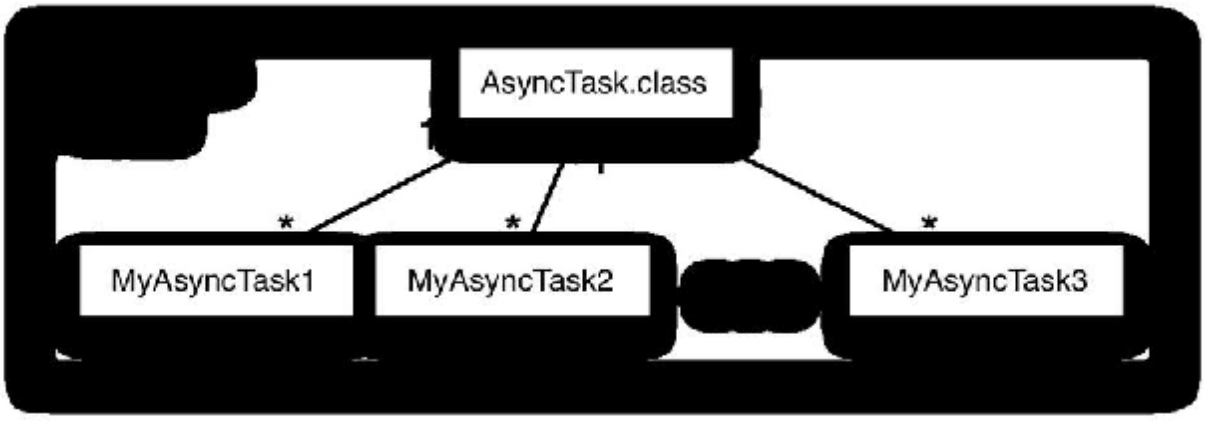


Figure 10-3. AsyncTask application global behavior

```
new FirstLongTask().execute();
...
new SecondLongTask().execute();
```

It does not matter whether the AsyncTask implementations are executed from an Activity, a Service, or any other part of the application—they still use the same application global execution environment and run sequentially

Important

The application global execution of AsyncTask instances poses a risk that the execution environment gets saturated and that background tasks get delayed, or worse still, not executed at all.

Because all AsyncTask instances share this global execution, they all can have an impact on each other, which depends on the execution environment:

Sequential execution (SERIAL_EXECUTOR)

Tasks that are executed sequentially will not be processed on a worker thread until all the preceding tasks in the application have been processed. This applies to any tasks launched through executeOnExecutor(AsyncTask.SERIAL_EXECUTOR) or through execute() on API level 13 or higher.

Concurrent execution (THREAD_POOL_EXECUTOR)

The concurrent executor can handle five AsyncTask concurrently. When a sixth task is started, it will be placed in the waiting queue until one of the first five tasks has finished and left a worker thread idle. This may seem odd, because I previously stated that the THREAD_POOL_EXECUTOR can hold 138 tasks, but the reason lies in the implementation of the ThreadPoolExecutor with five core pool threads. When the core pool threads are all occupied, the implementation chooses queuing over the creation of new threads (see ThreadPoolExecutor Configuration).

Execution Across Platform Versions

It may be important to know whether tasks are being executed sequentially or concurrently, if tasks depend on the guaranteeed ordering or thread safety of sequential execution. Tasks launched through execute run sequentially, whereas executeOnExecutor can be run with a concurrent Executor. However, executeOnExecutor was added first in API level 11 (Honeycomb). This section explains what you need to know to handle the AsyncTask across platform versions. The differences in execution are summarized in Table 10-3.

Table 10-3. Execution differences depending on platform version.

API level	execute	executeOnExecutor
1 - 3	Sequential	Not available
4 - 10	Concurrent	Not available
11 - 12	Concurrent	Sequential/Concurrent (Customizable)
13+	Sequential	Sequential/Concurrent (Customizable)

At first, execution was always sequential, but to gain performance, the execute method was changed in API level 4 to process tasks concurrently. Unfortunately, tasks that depended on ordered or thread-safe execution could fail when they were exposed to a non-ordered and non-thread safe enviroment. So in API level 11 the API was extended with the executeOnExecutor method, and in API level 13 the execute method was reverted to sequential execution to restore the previous safe behavior. The executeOnExecutor method supports custom Executor implementations, which can be used for concurrent execution.

Also in API level 13, the platform added a check to the targetSdkVersion in the applications' *AndroidManifest.xml* file to avoid unexpected behavior for existing applications:

targetSdkVersion<13
: execute keeps concurrent execution, even on platforms with API level 13 or higher.

targetSdkVersion>=13
: execute causes sequential execution on platforms with API level 13 or higher.

Important

The execution behavior of execute is dependent both on the API level of the platform and the targetSdkVersion in the application's Manifest.

An application that needs consistent execution behavior on all platform versions has to handle this itself by setting targetSdkVersion.

Until API level 13, it is not possible to achieve sequential execution across all platform versions with an AsyncTask.[14] Concurrent execution across platform versions is achieved either by setting targetSdkVersion < 13+ or by changing the actual executor, depending on the platform version:

Consistent sequential execution
: Sequential execution of AsyncTask instances cannot be guaranteed on API levels 4-10, because execute is concurrent and executeOnExecutor is not available until API level 11. Instead, background tasks that require consistent sequential execution should utilize either Executors.newSingleThreadExecutor (Chapter 9), or HandlerThread (Chapter 8).

Consistent concurrent execution
: The targetSdkVersion setting determines how concurrent execution can be consistently achieved across platform versions. For API levels lower than 13 the execute method suffices, but for higher API levels the application has to vary the execution call depending on the build:

```
if (Build.VERSION.SDK_INT <= Build.VERSION_CODES.HONEYCOMB_MR1) {
    new MyAsyncTask().execute();
} else {
        new
MyAsyncTask().executeOnExecutor(AsyncTask.THREAD_POOL_EXECUTOR);
}
```

Because checking the API for every execution is tedious, you can define a wrapper class can be implemented to handle the platform check:

```
public class ConcurrentAsyncTask {
    public static void execute(AsyncTask as) {
        if (Build.VERSION.SDK_INT < Build.VERSION_CODES.HONEYCOMB_MR2) {
```

```
                as.execute(...);
            } else {
                as.executeOnExecutor(AsyncTask.THREAD_POOL_EXECUTOR, ...);
            }
        }
}
```

The caller passes AsyncTask it wants executed to the wrapper class:

```
ConcurrentAsyncTask.execute(new MyAsyncTask());
```

Custom Execution

The predefined executors—SERIAL_EXECUTOR and THREAD_POOL_EXECUTOR—in the AsyncTask are application global, which risks a performance penalty when the application executes a lot of tasks. To circumvent global execution, tasks should be processed in a custom Executor:

```
new AsyncTask().executeOnExecutor(Params, MyCustomExecutor);
```

The custom executor replaces the execution environment in the AsyncTask, but preserves the communication between threads for progress updating: the overridden methods are called in the same way as they would have been with a predefined executor.

Sequential execution can be handled better by using a customized—non-global—executor in combination with the AsyncTask; e.g., the single threaded executor described in Chapter 9.

Example: non-global sequential execution

Sequential execution that is shared globally in an application may cause unexpected execution delays if a task from one component has to wait for a task from another component to finish. Hence, to utilize sequential execution—but avoid the application global behavior—a custom executor should be shared between the tasks.

The following, very bare-bones example shows the use of a sequential executor, Executors.newSingleThreadExecutor, in both an Activity and a Service. Because the tasks are executed in different components–but still require the same executor—the executor instance is held in the Application instance.

```
public class EatApplication extends Application {
    private Executor customSequentialExecutor;

    public Executor getCustomSequentialExecutor() {
```

```
            if (customSequentialExecutor == null) {
                customSequentialExecutor = Executors.newSingleThreadExecutor();
            }
            return customSequentialExecutor;
        }
}
public class MyActivity extends Activity {
        private void executeTaskSequentially() {
            new
MyActivityAsyncTask().executeOnExecutor(((EatApplication)getApplication).getCustomSeq
uentialExecutor());
}
public class MyService extends Service {
        private void executeTaskSequentially() {
            new
MyServiceAsyncTask().executeOnExecutor(((EatApplication)getApplication).getCustomSequ
entialExecutor());
}
```

AsyncTask Alternatives

Due to its simplicity, the AsyncTask is a popular asynchronous technique. It allows background task execution in combination with thread communication, offering a generic and adaptable asynchronous technique that can be applied on many use cases—the AsyncTask itself does not impose any constraints. However, as we have seen in this chapter, it has a couple of concerns you need to consider:

- Because AsyncTask has a global execution environment, the more tasks you execute with an AsyncTask, the higher is the risk that tasks will not be processed as expected, because there is other tasks in the application that holds the execution environment.
- Inconsistency in execution environments over different platform versions makes it more difficult to either optimize execution for performance (concurrent execution) or thread safety (sequential execution).

The AsyncTask is often overused in applications, due to its simplicity. It is not a silver bullet solution for asynchronous execution on Android. For many use cases, you should look into alternative techniques, for reasons of architecture, program design, or just because they are less error-prone.

When an AsyncTask is trivially implemented

Two trivial use cases where the AsyncTask can cause more complexity than the alternatives are:

- Running the task without parameters (AsyncTask<Void, Void, Void>):: An AsyncTask that does not define any parameters cannot pass data between the UI thread and the background thread. Data can not be entered into the background thread, no progress can be reported, and no result is not passed from background thread to the UI thread.
- Implementing only the doInBackground method:: Without the callbacks that give progress updates or report results, the AsyncTask is merely a background task

In either of these cases, use a Thread (Chapter 7) or HandlerThread (Chapter 8) instead.

Background Tasks that Need a Looper

The worker thread that executes the background task under AsyncTask has no associated Looper or MessageQueue, so message passing is unfeasible. In theory, it is possible to associate a Looper with the worker thread in either doInBackground or an executed Runnable, but this will block the used worker thread until the Looper finishes. When sequential execution is in effect, it will block all other AsyncTask executions in the application.

Even if the Looper is just prepared–but does not loop through the message queue–it will not be removed from the worker thread so that the thread can be used by other task executions. If a second task tries to prepare another Looper on that thread, a RuntimeException will be thrown.

If your application wants a Looper, use a HandlerThread (Chapter 8) instead of an AsyncTask.

Local Service

A local Service executes in parallel with other components in an application, typically to handle execution of long operations. The Service executes in the UI thread of the hosting application, and requires additional background threads to execute the long operations. The AsyncTask is a candidate, but the application global execution of tasks allows other components to utilize the execution environment simultaneously, and interfere with each other.

Services, therefore, should use one of these alternative solutions:

- Thread (Chapter 7).
- Executor Framework (Chapter 9).
- HandlerThread (Chapter 8).

- An AsyncTask with a custom executor.

Using execute(Runnable)

Executing tasks as Runnable instances eliminates the major advantages of the AsyncTask; it merely puts the Runnable in the working queue and runs it when an idle thread is available in the thread pool. Because message passing is not enabled, the UI thread will receive no callbacks. Hence, this use case is like execution with a regular Thread, but with two main differences:

- Advantage: The task is executed in the AsyncTask internal thread pool that may already exist, which makes it resource-efficient.
- Disadvantage: The task always executes in the application global execution environment, and can interfere with other tasks.

Alternative solutions include Thread (Chapter 7) and Executor Framework (Chapter 9).

Summary

This chapter looked into the—probably—most popular asynchronous technique in Android. It is easily understood, because it abstracts away many of the underlying complexities of background execution and thread communication. For clean use cases—e.g., executing a background task in an Activity, where the UI should be updated before, during, and after the execution—it is a great option. It is less desirable if execution is done in a Service or if it is necessary to tweak the AsyncTask to something it is not, e.g., a background thread with a Looper.

---

[14] API levels 1-3 are considered to be obsolete, and are therefore omitted.

Chapter 11. Services

Android provides the Service component to run operations that are invisible to the user or that should be exposed to other applications. This chapter focuses on asynchronous execution with Service, although it is not an asynchronous execution environment by itself. The Service runs in the UI thread, so it can degrade responsiveness and cause ANRs, even though it does not interact directly with the UI. Still, the Service in combination with an asynchronous executor is a powerful tool for background task execution.

Why Use a Service For Asynchronous Execution?

Two risks are inherent in using regular threads instead of services for background operation.

Decouple lifecycles of components and threads
: The thread lifecycle is independent of the Android components and their underlying Java object lifecycles. A thread continues to run until the task either finishes or the process is killed, even after the component that started the thread finishes itself. Threads may keep references to Java objects so that they can not be garbage collected until the thread terminates, as described in Chapter 6.

Lifecycles of the hosting processes
: If the runtime terminates the process, all of its threads are terminated. Thus, background tasks are terminated and not restarted by default when the process is restored. The runtime terminates processes depending on their process rank—as described in Application Termination—and a process with no active components has a low ranking and is likely to be eligible for termination. This may cause unexpected termination of background tasks that should be allowed to finish. For example, an Activity that stores user data to a database in a background thread while the user navigates back leaves an empty process if there are no other components running. This increases the risk of process termination, aborting the background thread before it can persist the data.

A Service can mitigate both the risk for memory leaks and the risk of having tasks terminated prematurely. The Service has a lifecycle that can be controlled from background threads: it couples the component lifecycle with the thread's lifetime. Hence, the Service component can be active while the background thread runs and be destroyed when it finishes, which enables better lifecycle control. As Figure 11-1 illustrates, the BroadcastReceiver and Activity lifecycles are decoupled from the background thread's execution, whereas the Service lifecycle can end when the background task is done. Consequently, the process contains a Service component throughout the background thread's execution. The details of starting and stopping Services are explained later in this chapter.

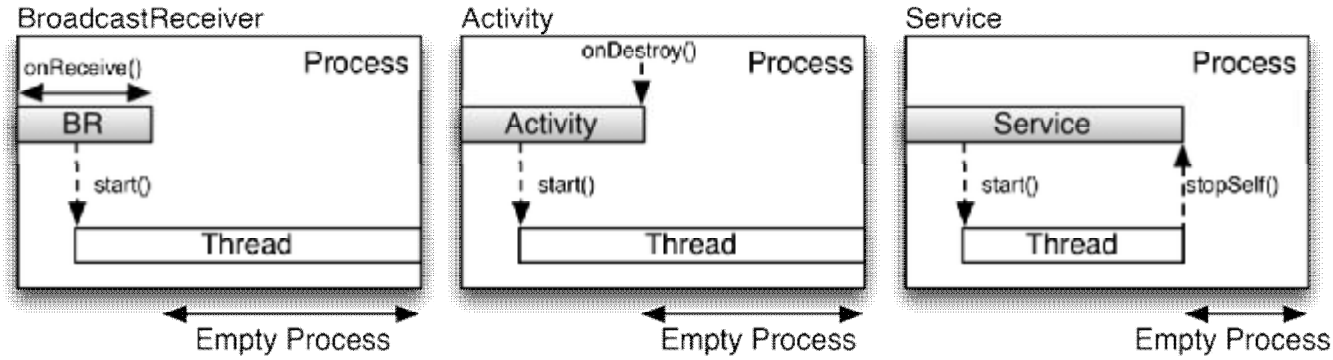


Figure 11-1. Background thread execution in BroadcastReceiver and Activity compared to Service, that both can execute in the same process as other components or in separate process.

To offload background execution, a BroadcastReceiver or Activity should start a Service that then starts a thread, as shown in Figure 11-2.

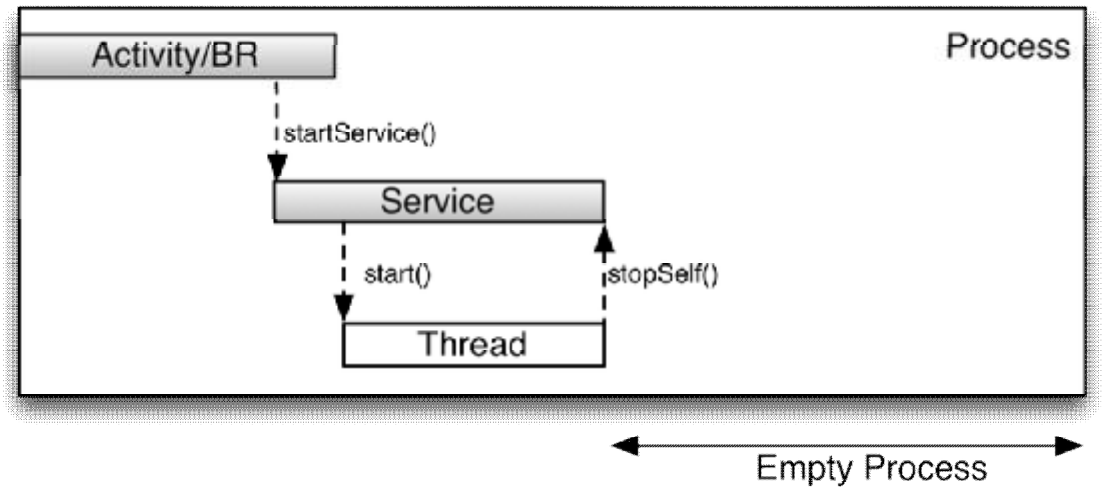


Figure 11-2. Offloading background tasks to Service

Local, Remote, and Global Services

A Service is started through Intents that are sent from other components, referred to as *client components*. The invocation can occur either local to the process or across process boundaries, depending on where the Service runs with respect to the client component. If the Service is used within the same process as the client, it is local, whereas a Service used from external processes is remote Figure 11-3 shows the possibilities.

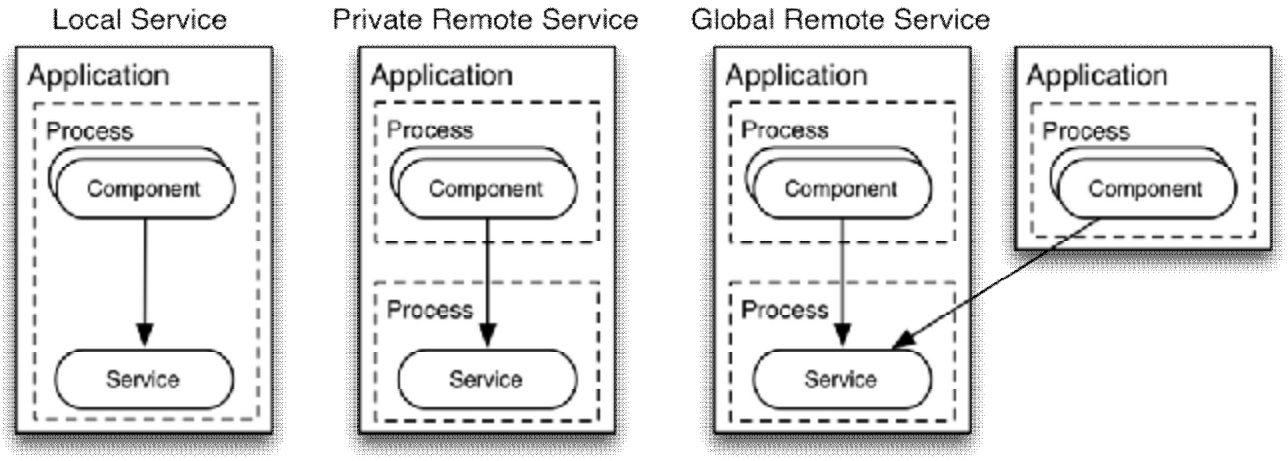


Figure 11-3. Service execution

Local service

The Service runs in the same process as the invoking component, i.e. the components run on the same UI thread and share the same heap memory area. Hence, the Service can share Java objects with clients, so that the shared objects run on the calling thread in the client.

Private remote service

The Service runs in a remote process but is accessible only to client components that belong to the application. The remote process has its own UI thread. Thus, the Service UI thread does not delay the execution of the UI threads of the client components. The remote Service cannot expose objects directly to the clients, as they do not share the the same heap memory area. Instead, clients call the Service through the Binder mechanism in Android–see Chapter 5–which contains a pool of binder threads used to invoke the remote methods. Thus, method invocations can be invoked concurrently on different threads from the binder thread pool, although the clients call the methods sequentially. The Service has to ensure thread safety during remote access from binder threads.

Global remote service

The Service is exposed to other applications. It has the same properties as the private remote service with its own UI thread, heap memory, and execution on binder threads, but it can not be referred to by the Service class name because that is not known to external applications. Instead, external access is provided through IntentFilters.

By far the most common use case is to run the Service in the local process, where it executes background tasks deployed by client components, e.g. a music player service. The advantage of the local service are simplicity and saving memory. The possibility of sharing Java objects within the process avoids the complexities of IPC and AIDL. Furthermore, the Service is part of an already running process—i.e., you don't need to run another process that consumes memory. Each process consumes several megabytes of RAM memory, even if it only hosts a

Service component. Applications should be good Android citizens in the ecosystem and not start processes unless needed.

In short, the advantages with running in the local process are:

- Easier and faster communication through shared Java objects instead of IPC
- Control over Service task execution from client threads
- Less memory consumption

Although local Services are normally preferred and sufficient, there may be requirements that call for remote execution. Typically, that's when multiple applications require the same functionality that is independent enough to be shared across the applications—e.g., a GPS or music service. Another advantage of remote execution is that errors that stops the Service are contained in the remote process and do not affect the processes run by the client components.

Creation and Execution

Services are defined as extensions of the Service class and must be defined in the Android Manifest. The <name> attribute contains a fully qualified classname, which points to an implementation class that extends Service in the application:footnote[See the documentation for the <service> element for the full list of attributes and elements.]

```
<service
    android:name="com.wifill.eat.EatService"/>
```

By default, the Service component runs locally in the application process where it is defined and shares the same UI thread with all other components in that process. But just as with all Android components, the Service can be assigned to run in a remote process, in which case it is not executing on the same UI thread as the other application components. The assigned process is defined by the android:process attribute, which declares the name of thh process and the access rights. A private remote process has an attribute value that starts with a colon (":").

```
<service
    android:name="com.wifill.eat.EatService"
    android:process=":com.wifill.eat.PrivateProcess"/>
```

Execution in a global remote process–accessible from other applications with the right permissions–is defined by leading off the process name with a capital letter:

```
<service
```

```
    android:name="com.wifill.eat.EatService"
    android:process="Com.wifill.eat.PrivateProcess">
    <intent-filter>
        <action android:name="..." />
        <category android:name="..." />
    </intent-filter>
</service>
```

As the Service class name is not visible to other applications, it defines an IntentFilter that external applications have to match against.

Lifecycle

A Service component is active between the callbacks to onCreate and +onDestroy+–both are called once per lifecycle–where the implementation can initialize and clean up data, respectively.

```
public class EatService extends Service {

    @Override
    public void onCreate() { /* Initialize component */ }

    @Override
    public void onDestroy() { /* Clean up used resources */ }

    @Override
    public IBinder onBind(Intent intent) { /* Return communication interface */ }
}
```

The only mandatory method is onBind, which returns a communication interface to clients that bind to the Service. Through this interface, clients can invoke methods defined in the Service either in the local process or remotely.

There are two types of Services:

Started Service
: Created by the first start request and destroyed by the first stop request. In between, start requests only pass data to the Service.

Bound Service
: Created when the first component binds to the service and destroyed when all components have unbound from it. In other words, a bounded Service lifecycle is

based on the number of binding components. As long at least one component is bound to the Service, it stays active.

The Service component is created by client components that either start it through Context.startService or bind to it through Context.bindService. These are two fundamentally different approaches with different access methods and communication mechanisms, summarized in Table 11-1.

Table 11-1. Started vs. Bound Service.

	Started	Bound
Create	Context.startService(Intent)	Context.bindService
Destroy	Context.stopService(Intent) from a client component or Service.stopSelf() from itself.	Occurs when all bound components have unbound from the Service with Context.unbindService().
Communication	Data is passed in the Intent of the start request.	Binding component receives a communication interface.

If some process starts a Service component, the component lasts until it stops itself or is stopped externally. If processes bind to a Service component, it starts when the first remote process binds to it, and lasts until all processes that have bound to it unbind from it. If one process starts the component and others bind to it, it lasts as long as both conditions hold: it will not be terminated until it is explicitly stopped and all processes have unbound from it. The various cases are shown in Figure 11-4.

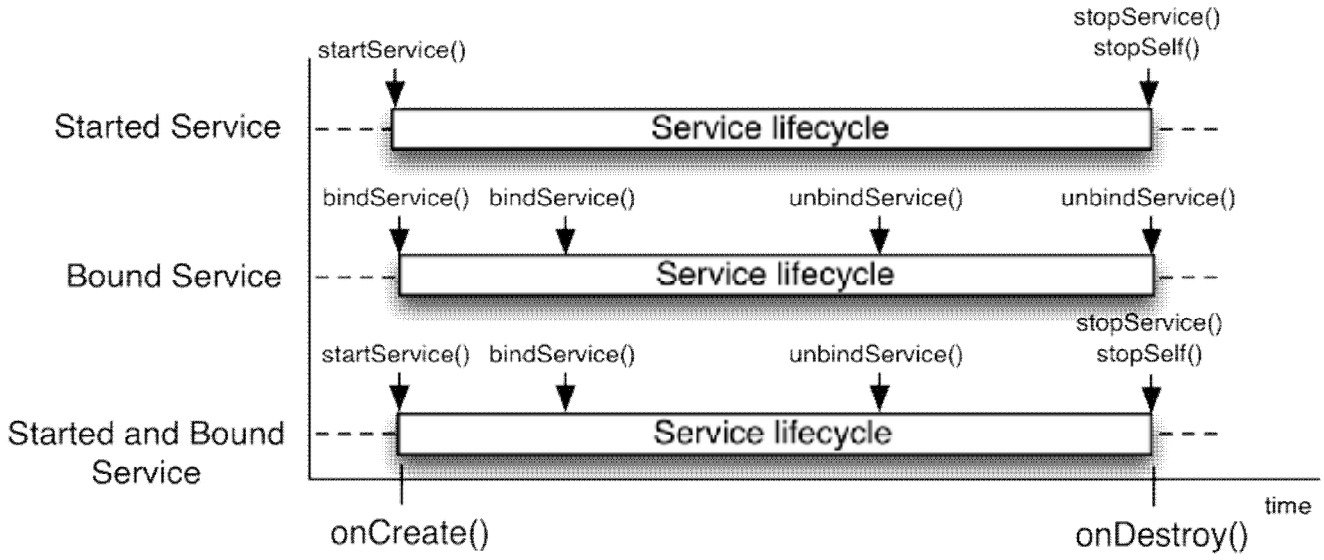


Figure 11-4. Service lifecycle.

Started Service

Components invoke Context.startService(Intent) to send start requests to a Service, which can be invoked by multiple components and multiple times from every component during a lifecycle. The first start request creates and starts the Service, whereas consecutive start requests just pass on the Intent to the started Service so that the data conveyed in the Intent can be processed.

Services always have to implement onBind, but started Services—which do not support binding—should provide a trivial implementation that just return null.

```
public class StartedEatService extends Service {

    @Override
    public int onStartCommand(Intent intent, int flags, int startId) { ... }

    @Override
    public IBinder onBind(Intent intent) { return null; }

}
```

Started Services must implement an onStartCommand method that handles start requests. The method is invoked each time a start request (Context.startService) from a client component is ready to be processed. Start requests are delivered sequentially to onStartCommand and remain pending in the runtime until preceding start requests are processed or offloaded from the UI thread. In spite of the sequential processing of start requests, calls to startService do not block even if they have to wait to be processed in the Service. A start request supplies an Intent that conveys data that can be processed in the Service.

onStartCommand is executed on the UI thread, so you should spawn background threads within the method to execute long running operations, not only to preserve responsiveness but also to enable concurrent execution of multiple start requests.

The sequential processing of onStartCommand on the UI thread guarantees thread safety. No synchronization is required unless the tasks are processed concurrently on background threads spawned from the UI thread.

Implementing onStartCommand

onStartCommand is the key method for implementing started Services and initiating asynchronous task execution. Tasks should be offloaded to an asynchronous executor that can be either sequential or concurrent. Because the Service process along with all its background threads can be terminated by the runtime, onStartCommand includes a flag that defines how start requests—both pending and processed—should be handled after process termination.

The arguments are:

Intent
: Data to be used for the asynchronous execution, e.g., a URL to a network resource that shall be retrieved.

Delivery method
: A flag reflecting the history of the start request. This argument may contain other flags in future versions of Android. Possible values are currently 0, START_FLAG_REDELIVERY (value of 1), or START_FLAG_RETRY (value of 2). The following section will explain this argument.

Start ID
: A unique identifier provided by the runtime for this start request. If the process is terminated and restarted, onStartCommand is called with the same start ID.

The return value tells the runtime whether to restart the Service and resubmit the +Intent= argument in case the runtime has to terminate the process for lack of resources and then restart it. This will be explained in the following section.

Options for Restarting

Like any Android application, your Service may be terminated by the runtime if there are too many processes running on the device. In fact, as a background process, your Service has a greater chance of being killed than many other processes (Application Termination). A client can also terminate your Service through a Context.stopService call. Finally, a Service can terminate itself through Service.stopSelf. This section covers termination by the runtime or a client, not a clean termination through Service.stopSelf.

The return value of onStartCommand and the second argument (the delivery method flag) let you control what happens after your Service is terminated. You may want to restart the same request, which you can do by asking the runtime to resubmit the +Intent that you received in this onStartCommand call. Or you may abandon that Intent. If you abandon the current Intent, you have another choice:

- You can ask the runtime to restart your Service only if other requests be pending, and if there are none currently, to restart the Service when a new request comes from a client.
- You can ask the runtime to restart your Service as soon as it can, even if there are no requests.

Note

If a Service is terminated while the onStartCommand method is running, the Intent that was submitted never gets a chance to start. Therefore, it is still considered a pending request, not a started request

All these options are enabled by the return value from the call, which is called the *operational mode*.

START_STICKY

The Service will be restarted in a new process whether or not there are any requests pending. The Intent of the request that was terminated will not be resubmitted. However, the Service will receive any pending start requests that remained undelivered when the previous Service process was terminated. The pending start requests are delivered with the START_FLAG_RETRY flag set in the call's second argument. If there are no pending start requests, onStartCommand is invoked with a null value for the Intent argument.

START_NOT_STICKY

Like START_STICKY, except that the Service will be restarted only if there were pending start requests when the process was terminated. An Intent will always be passed.

START_REDELIVER_INTENT

The Service will be restarted and receives both pending requests and requests that were previously started and had no chance to finish. The pending requests are delivered with the START_FLAG_RETRY flag set in the second argument, whereas the previously started requests are redelivered with the START_FLAG_REDELIVERY set.

Value 0

Default value, used when a new request is submitted. A new process is started to run the Service.

START_FLAG_REDELIVERY (Value 1)

Indicates that the Intent is one that was previously submitted to the Service.

START_FLAG_RETRY (Value 2)

Indicates that the Intent was not previously submitted or never had a chance to start, but was pending when the Service was terminated and restarted.

Here are sample use cases for the return value;

Restart a single background task
: In this scenario, you want the Service to execute a background task that should be controlled from the client components. The Service should not finish until one component has invoked stopService. Return START_STICKY so that the Service is always restarted and the background task can be restarted.

Ignore background tasks
: In this scenario, the Service executes a task that should not be resumed after process termination. An example might be a periodic task that can wait until its next scheduled execution. Return START_NOT_STICKY so that the Service will not be automatically restarted.

Restart unfinished background tasks
: In this scenario, the Service executes background tasks that you want to be resumed. Return START_REDELIVER_INTENT so that all the Intents are redelivered, and the tasks can be restarted on new threads. Furthermore, the tasks can be configured with the same data from the Intent as the original task.

User-Controlled Service

A user-controlled Service may be used for operations that should be running until instructed by other Android components that they should terminate. The termination is typically triggered by a user action. Applications that should continue to execute operations and handle events without a visible UI are candidates for extracting functionality to a Service: for instance, playing music, location management, tracking updates from a network resource, or setting up a Bluetooth connection as Example: Bluetooth Connection illustrates.

Long-running or blocking operations need to be executed asynchronously on background threads created by the Service. But the Service lifecycle does not control the lifetime of background threads: they continue to run after the component has been destroyed. In other words, the threads are asynchronous tasks that run after a client has called stopService. As Chapter 6 showed, the lingering threads pose a risk for leaking the memory of the objects they reference. A Service that is started and stopped repeatedly can leave more threads lingering for every lifecycle, each one referencing a different Service instance and forcing it to be retained in memory.[15] To reduce the risk for memory leaks and performance issues caused by many lingering threads, the Service should not be restarted repeatedly if possible. It is better to let the Service stay alive as long as needed, to limit the number of lingering threads and the memory leakage.

Example: Bluetooth Connection

A Bluetooth connection is generally initialized by the user when pairing with another device. The pairing can take a long time, and while waiting, the user may navigate away from the application. Therefore, to ensure that the pairing process can continue independently of the user navigation, the pairing should be done in a Service. The bluetooth pairing API is synchronous and cannot be called from the UI thread methods, so the Service requires additional asynchronous pairing in a background thread.

The following example contains a user-controlled BluetoothService to set up and cancel the pairing, and a basic BluetoothActivity with two buttons that control the Service lifecycle. When the BluetoothService is started–i.e., onStartCommand is called–it initiates a Thread to handle the pairing and keeps a state variable–mListening+–to ensure that only one pairing operation is active. Consequently, only one thread at the time will be alive. The pairing task defined in a +Runnable executes until a connection is set up or cancelled, after which the background thread terminates.

The BluetoothService shares an entry point–COMMAND_START_LISTENING+–to start the pairing process, but no method for stopping it. Instead, the pairing process is terminated when the +BluetoothService stops itself, and its onDestroy method cancels the pairing so that the background thread can terminate. Hence, there are no lingering background threads after the Service is destroyed.

```
public class BluetoothService extends Service {

    public static final String COMMAND_KEY = "command_key";
    public static final String COMMAND_START_LISTENING =
"command_start_discovery";

    private static final UUID MY_UUID = ...;
    private static final String SDP_NAME = ...;

    private BluetoothAdapter mAdapter;
    private BluetoothServerSocket mServerSocket;
    private boolean mListening = false;
    private Thread listeningThread;

    public IBinder onBind(Intent intent) {
        return null;
    }
```

```
    @Override
    public void onCreate() {
        super.onCreate();
        mAdapter = BluetoothAdapter.getDefaultAdapter();
    }

    @Override
    public int onStartCommand(Intent intent, int flags, int startId) {
        if (mAdapter != null) {
            if
(intent.getStringExtra(COMMAND_KEY).equals(COMMAND_START_LISTENING)
                    && mListening == false) { ❶startListening();
            }
        }
        return START_REDELIVER_INTENT; ❷}

    private void startListening() {
        mListening = true;
        listeningThread = new Thread(new Runnable() {

            @Override
            public void run() {
                BluetoothSocket socket = null;
                try {
                    mServerSocket                                                    =
mAdapter.listenUsingInsecureRfcommWithServiceRecord(
                            SDP_NAME, MY_UUID);
                    socket = mServerSocket.accept(); ❸if (socket != null) {
                        // Handle BT connection
                    }

                } catch (IOException e) {
                    Log.d(TAG, "Server socket closed");
                }
            }
        });
        listeningThread.start();
    }

    private void stopListening() {
```

```
            mListening = false;
            try {
                if (mServerSocket != null) {
                    mServerSocket.close(); ❹}
            } catch (IOException e) {
                // Handle error
            }
        }

        @Override
        public void onDestroy() {
            super.onDestroy();
            stopListening(); ❺}
}
```

❶

Control the number of threads so that only one pairing at the time can be done.

❷

If the process is shut down, the Intent will be redelivered so that we can resume the pairing.

❸

Blocking call.

❹

Release the blocking call.

❺

The Service is destroyed and stops the pairing so that the background thread can finish.

The BluetoothActivity controls the Service lifecycle with a start command that initiates the pairing and stops the pairing by destroying the BluetoothService that manages the background thread.

```
public class BluetoothActivity extends Activity {

    public void onCreate(Bundle savedInstanceState) {
```

```
        ...
    }

    public void onClickStartListening(View v) {
        Intent intent = new Intent(this, BluetoothService.class);
        intent.putExtra(BluetoothService.COMMAND_KEY,
BluetoothService.COMMAND_START_LISTENING);
        startService(intent);
    }


    public void onClickStopListening(View v) {
        Intent intent = new Intent(this, BluetoothService.class);
        stopService(intent);
    }
}
```

Task-Controlled Service

A task-controlled Service is typically used to ensure that background threads can be allowed to finish execution with reduced risk of being stopped due to process termination. When the Service stops itself with stopSelf, control over the Service component's lifecycle lies with the processed task. In other words, the task determines when the component will be destroyed.

The lifetime of the background thread determines the lifetime of the Service, as shown earlier in Figure 11-2. Therefore, the component is always active while the task is running on the background thread, which raises its chances of being kept alive. (As Application Termination explained, an empty process is among the first to be terminated by the runtime.) So a task-controlled Service allows applications to utilize a Service for long-running operations on background threads.

Example: Concurrent Download

This example illustrates the use of a concurrent file download executor running in a Service. The time it takes to download a file over the network is non-deterministic—and often very long—because it depends on both the file size and the network connection quality. Hence, we want to avoid downloading the files from an Activity because the user can navigate away from the application during download, leaving an empty process that can be terminated by the runtime before the download has finished. Instead, we let the download be handled by a task-controlled Service that is independent of user navigation and reduces the probability that the runtime terminates the process during the download.

The implementation fulfills some requirements:

- Concurrent downloads improve performance.
- The Service can resume downloading of unfinished files if the process is terminated by the runtime.
- The Service stops when there are no more download requests to handle, so that its process can be terminated and free up resources when it is not needed anymore. In this example, the Service runs in the application process, but it can easily be configured to run in a separate process (Creation and Execution).

The DownloadService is registered in the Android Manifest with an action and a scheme:

```
<service android:name=".DownloadService">
    <intent-filter>
        <action android:name="com.wifill.eat.ACTION_DOWNLOAD"/>
        <data android:scheme="http" />
    </intent-filter>
</service>
```

The file download can be triggered from any application component by issuing an Intent with an ACTION_DOWNLOAD action, as shown in the following code from the DownloadActivity. Typically, the intent would be started by a button click that triggers the onStartDownload callback:

```
public class DownloadActivity extends Activity {

    String mUrl = ...; // url details omitted

    public void onStartDownload(View v) {
        Intent intent = new Intent("com.wifill.eat.ACTION_DOWNLOAD");
        intent.setData(Uri.parse(mUrl));
        startService(intent);
    }
}
```

The DownloadService is defined to be started and not bound to, and it stops itself when all the start requests have been processed. Concurrent execution is done in a thread pool with a fixed size of four worker threads. The long-operating requests are submitted to the thread pool to offload them from the UI thread and to make the Service available to receive new start requests.

```
public class DownloadService extends Service {
```

```
private ExecutorService mDownloadExecutor;
private int mCommandCount; ❶public IBinder onBind(Intent intent) { ❷return null;
}

@Override
public void onCreate() {
    super.onCreate();
    mDownloadExecutor = Executors.newFixedThreadPool(4);
}

@Override
public void onDestroy() {
    super.onDestroy();
    mDownloadExecutor.shutdownNow();
}

@Override
public int onStartCommand(Intent intent, int flags, int startId) {
    synchronized (this) {
        mCommandCount++;
    }
    if (intent != null) {
            downloadFile(intent.getData(), startId);
    }
    return START_REDELIVER_INTENT; ❸}

private void downloadFile(final Uri uri) {
    mDownloadExecutor.submit(new Runnable() {
        @Override
        public void run() {

            // The file download code is omitted

            synchronized (this) { ❹if (--mCommandCount <= 0) {
                    stopSelf();
                }
            }
        }
    });
```

❶

Track the number of ongoing file downloads. The variable is incremented when a start request is received and decremented when the background task is finished.

❷

The interface requires an onBind method to be implemented, but because this is a started–and not a bound–Service, we do not have to return an IBinder implementation; instead return null.

❸

If the process is terminated by the runtime, we want the Service to be restarted with the Intents from the start requests that hold the URLs to download, so that the downloads can be resumed. Hence, we return START_REDELIVER_INTENT.

❹

When a download has finished, the task checks whether it is time to stop the Service. If it is the last start request, indicated when mCommandCount <= 0, we stop the Service.

Bound Service

A bound Service defines a communication interface that the binding components–referred to as _client components_–can utilize to invoke methods in the Service. The communication interface is defined as a set of methods that the Service implements and executes in the Service's process. The client components can bind to a Service through Context.bindService. Multiple client components can bind to a +Service+–and invoke methods in it–simultaneously.

The Service is created when the first binding is set up, and is destroyed when there are no more client components keeping the binding alive. Internally, the Service keeps a reference counter on the number of bound components, and when the reference counter is decremented to zero, the Service is destroyed. A client component can terminate a binding explicitly with Context.unbindService, but the binding is also terminated by the runtime if the client component lifecycle ends.

The outline for a bound Service is:

public class EatService extends Service {

```
    @Override
    public IBinder onBind(Intent intent) { /* Return communication interface */ }

    @Override
    public boolean onUnbind(Intent intent) { /* Last component has unbound */ }
}
```

onBind

Called when a client binds to a Service the first time through Context.bindService. The invocation supplies an Intent and the method returns an IBinder implementation that the client can use to interact with the Service.

onUnbind

Called when all bindings are unbound.

Components bind to Services and retrieve a communication interface for sending requests and receiving responses, either within the process or across processes, as in IPC through a Binder. The communication interface consists of methods to be invoked by the client component, but the execution is implemented in the Service component.

A bound Service returns an IBinder implementation from onBind that the client can use as a communication channel. The IBinder is returned to the client through the ServiceConnection interface that the binding client supplies when invoking bindService:

boolean bindService (Intent service, ServiceConnection conn, int flags)

The Intent identifies the Service to bind to, but it cannot pass on any Extra parameters to the Service. The binding is observed by the ServiceConnection, and the binding client component provides an implementation to get notified when the binding is established:

```
private class EatServiceConnection implements ServiceConnection {

    @Override
    public void onServiceConnected(ComponentName componentName, IBinder iBinder) {
        /* Connection established. Retrieve communication interface from the IBinder */
    }

    @Override
    public void onServiceDisconnected(ComponentName componentName) {
        /*    Service not available. Remote service process is probably terminated. */
    }
```

The flags argument can be either 0 or a collection of allowed arguments. Options for this argument can adjust the rank of the Service's process or determine the strategy for restarting

the Service. The most commonly used flag is BIND_AUTO_CREATE, which recreates the Service if it is destroyed, as long as a client component is bound to it.

From a client perspective, the fundamental binding to the Service is the same regardless of which process the Service runs in. However, if the Service runs in the local process, communication is easier than with remote processes. Also the behavior and management of asynchronous execution differs between the three communication types:

- Local binding, covered in the following section
- Remote binding with Messenger, covered in Message-Passing Using the Binder
- Remote binding with AIDL, covered in AIDL

Local Binding

Local binding to a Service is the most common type. A client component that binds to a Service in the same application and process can benefit because both components run in the same VM and share the same heap memory. There is no need to be concerned with the complexities of IPC. Instead, the Service can define the communication interface as a Java class and send it to the client wrapped in a Binder object. The following example provides itself as communication interface (through return BoundLocalService.this), but it could just as well define the interface as an internal class.

```
public class BoundLocalService extends Service {
    private final ServiceBinder mBinder = new ServiceBinder();

    public IBinder onBind(Intent intent) {
        return mBinder;
    }

    public class ServiceBinder extends Binder {
        public BoundLocalService getService() {
            return BoundLocalService.this;
        }
    }

    // Methods published to clients.
    public void publishedMethod1() { ... }
    public void publishedMethod2() { ... }

}
```

The BoundLocalService creates a Binder subclass called ServiceBinder, which wraps the Java object of the Service, and returns it to the binding client components. The Service implements methods that can be invoked by bound clients.

The clients retrieve the Binder implementation in the ServiceConnection.onServiceConnected() callback, which contains the communication interface.

The client BoundLocalActivity retrieves the communication interface–i.e., the BoundLocalService object itself–from the ServiceBinder and stores it as a member variable for later reference.

```
public class BoundLocalActivity extends Activity {
    private LocalServiceConnection mLocalServiceConnection = new
LocalServiceConnection();
    private BoundLocalService mBoundLocalService;
    private boolean mIsBound;

    public void onCreate(Bundle savedInstanceState) {
        bindService(new Intent(BoundLocalActivity.this, BoundLocalService.class),
            mLocalServiceConnection, Service.BIND_AUTO_CREATE);
            mIsBound = true;
    }

    @Override
    protected void onDestroy() {
        super.onDestroy();
        if (mIsBound) {
            try {
                unbindService(mLocalServiceConnection);
                mIsBound = false;
            } catch (IllegalArgumentException e) {
                // No bound service
            }
        }
    }

    private class LocalServiceConnection implements ServiceConnection {

        @Override
```

```
        public void onServiceConnected(ComponentName componentName, IBinder
iBinder) {
            mBoundLocalService =
((BoundLocalService.ServiceBinder)iBinder).getService();

            // At this point clients can invoke methods in the Service,
            // i.e. publishedMethod1 and publishedMethod2.

        }

        @Override
        public void onServiceDisconnected(ComponentName componentName) {
            mBoundLocalService = null;
        }
    }
}
```

When the client has bound to the local Service, it can invoke the methods that were published in its communication interface. The invocations are done directly to the Java object created in the Service, and they are executed on the thread of the calling client. Thus, long operations that require execution off the UI thread can utilize asynchronous behavior implemented either in the client or in the Service.

Our BoundLocalService executes tasks on a new background thread for every invocation of the published method in the communication interface:

```
public class BoundLocalService extends Service {

    public interface OperationListener {❶public void onOperationDone(int i);
    }

    public void doLongAsyncOperation(final int i, final OperationListener listener) { ❷new
Thread(new Runnable() {
            @Override
            public void run() {
                int result = longOperation(i);
                listener.onOperationDone(result); ❸}
        }).start();
    }

    private int longOperation(int i) {
```

```
        // Return a result from the long operation.
    }
}
```

❶

Callback listener to report the result to the client.

❷

Communication interface published to the client.

❸

The thread holds a reference to the listener that is defined in the binding client component. This poses a risk for a memory leak of the object tree that the listener references in the client.

A BoundLocalActivity, which invokes BoundLocalService and its executor, defines an implementation of the OperationListener to retrieve the result from the background execution and use it on the UI thread, which is a common use case. In the BoundLocalService, the background thread references the listener; anything referenced in the listener cannot be garbage collected while the thread is running (see Chapter 6). Hence, our BoundLocalActivity defines a static inner class ServiceListener with weak references to the Activity.

```
public class BoundLocalActivity extends Activity {

    // Activity creation and Service binding omitted. See previous BoundLocalService listing.

    private static class ServiceListener implements BoundLocalService.OperationListener {

        private WeakReference<BoundLocalActivity> mWeakActivity;

        public ServiceListener(BoundLocalActivity activity) {
            this.mWeakActivity = new WeakReference<BoundLocalActivity1>(activity);
        }

        @Override
        public void onOperationDone(final int someResult) {
            final BoundLocalActivity localReferenceActivity = mWeakActivity.get();
            if (localReferenceActivity != null) {
                localReferenceActivity.runOnUiThread(new Runnable(){
                    @Override
```

```
                    public void run() {

                        // Update on the UI thread

                    }
                });
            }
        }
    }

    public void onClickExecuteOnClientUIThread(View v) {
        if (mBoundLocalService != null) {
            mBoundLocalService.doLongAsyncOperation(new ServiceListener(this));
        }
    }
}
```

Choosing an Asynchronous Technique

Asynchronous task execution in Services can be sequential or concurrent, and can utilize any of the techniques described in this book. However, for two use cases alternatives may be considered:

Sequential execution for a task-controlled service

: A task controlled Service can execute tasks concurrently or sequentially, depending on used asynchrounous technique. When the tasks are executed sequentially—e.g. with a HandlerThread, Executors.newSingleThreadExecutor, or any customized Executor—it is better to use an IntentService (Chapter 12), because it has built-in support for sequential task execution that terminates the Service when there are no more tasks to execute.

AsyncTask with global executor in local process

: The Service component runs independently of—and often simultaneously with—other components in the same process. Hence, background tasks are likely to execute at the same time as background tasks of other components in the process. The AsyncTask risks delaying tasks because it has a process global executor, as described in Chapter 10, and the same executor is shared for all tasks. Hence, AsyncTasks that execute in the same process as other components should use a custom executor—i.e. a non-global executor—or an alternative technique, such as a thread pool.

Summary

A Service runs in the background on the UI thread and its of great use to offload tasks to background threads. The lifecycle of the Service component can be better controlled to fit background threads without letting user interaction interfere with the destruction of the component–as with an Activity. The Service can be active until all background tasks have finished.

---

[15] Like Activities, a new Service object is created for every lifecycle of the Service component.

Chapter 12. IntentService

In Chapter 11 we discussed how useful the Service lifecycle can be to handle asynchronous execution while increasing the process rank and avoiding termination of the background threads by the runtime. The Service, however, is not an asynchronous technique by itself, because it executes on the UI thread. This shortcoming is addressed in the IntentService, which extends the Service class. The IntentService has the properties of the Service lifecycle but also adds built-in task processing on a background thread.

Fundamentals

The IntentService executes tasks on a single background thread—i.e. all tasks are executed sequentially. Users of the IntentService triggers the asynchronous execution by passing an Intent with Context.startService. If the IntentService is running, the Intent is queued until the background thread is ready to process it. If the IntentService is not running, a new component lifecycle is initiated and it finishes when there are no more Intents to process. Hence, the IntentService runs only while there are tasks to execute.

Like a task-controlled Service (Task-Controlled Service), an IntentService always has an active component, reducing the risk of terminating the task prematurely.

Note

The background task executor in the IntentService is a HandlerThread. Unlike the default executor in AsyncTask, the IntentService executor is per instance and not per application. So an application can have multiple IntentService instances, where every instance executes tasks sequentially, but independent of other IntentService instances.

To use the IntentService, override it with an application specific implementation, declaring it in the AndroidManifest.xml as a Service component:

```
<service android:name=".SimpleIntentService"/>
```

The implementation that subclasses IntentService only has to define a constructor and override the onHandleIntent-method, as the following SimpleIntentService shows:

```
public class SimpleIntentService extends IntentService {
    public SimpleIntentService() {
        super(SimpleIntentService.class.getName());
        setIntentRedelivery(true);
    }
```

```
    @Override
    protected void onHandleIntent(Intent intent) {
        // Called on a background thread
    }
}
```

The constructor has to call the superclass with a string that names the background thread—for debugging purposes. You also specify here whether the IntentService shall be restored if the process is killed. By default, the IntentService is restored only if there are pending start requests, but an invocation of setIntentRedelivery(true) will redeliver the last delivered intent.

Note

The IntentService internally handles the two start types START_NOT_STICKY and START_REDELIVER_INTENT described in Options for Restarting. The first is default, so the latter needs to be set with setIntentRedelivery(true).

Clients that want to use the IntentService create a start request with Context.startService and pass an Intent with data that the service should handle.

```
public class SimpleActivity extends Activity {

    public void onButtonClick(View v) {
        Intent intent = new Intent(this, SimpleIntentService.class);
        intent.putExtra("data", data);
        startService(intent);
    }
}
```

Note

There is no need to stop the IntentService with stopSelf, because that is done internally.

Good Ways To Use An IntentService

The IntentService is suitable for when you want to offload tasks easily from the UI thread to a background thread with sequential task processing, giving the task a component that is always active in order to raise the process rank.

Sequential Ordered Tasks

Tasks that should be executed sequentially and independently of the originating component can use an IntentService to ensure that all submitted tasks are queued in the active IntentService component.

Example: Web Service Communication

Communication with network resources, such as web services, are often done in a sequential manner, i.e., one resource is retrieved that contains future instructions on how to interact with other resources.[16] The HTTP protocol interacts with the network resources using the GET, POST, PUT and DELETE request types. The request can originate in a user interaction or a scheduled system operation, but can be processed by an IntentService.

In this example, the request originates from an Activity, typically initialized by a user action. For simplicity's sake, only the most common types of requests are handled: GET for retrieving data and POST for sending it. Both are offloaded to an IntentService. The responses from the requests are returned in a ResultReceiver:

```
public class WebServiceActivity extends Activity {

    private final static String getUrl = "...";
    private final static String postUrl = "...";

    private ResultReceiver mReceiver;

    public WebServiceActivity() {
        mReceiver = new ResultReceiver(new Handler()) { ❶@Override
            protected void onReceiveResult(int resultCode, Bundle resultData) {
                int httpStatus = resultCode;
                String jsonResult =
resultData.getString(WebService.BUNDLE_KEY_REQUEST_RESULT);

                // Handle response
            }

        };
    }

    private void doPost() { ❷Intent intent = new Intent(this, WebService.class);
        intent.setData(Uri.parse(postUrl));
```

```
        intent.putExtra(WebService.INTENT_KEY_JSON, "{\"foo\":\"bar\"}");
        intent.putExtra(WebService.INTENT_KEY_RECEIVER, mReceiver);
        startService(intent);
    }

    private void doGet() { ❸Intent intent = new Intent(this, WebService.class);
        intent.setData(Uri.parse(getUrl));
        intent.putExtra(WebService.INTENT_KEY_RECEIVER, mReceiver);
        startService(intent);
    }

}
```

❶

Create the ResultReceiver that is passed to the IntentService so that the result of the operation can be returned.

❷

Issue a POST request with JSON-formatted content.

❸

Issue a GET request.

The IntentService receives the requests in onHandleIntent and processes them sequentially. The data from the WebServiceActivity determines the request type, url, ResultReceiver and possibly the data to be sent:

```
public class WebService extends IntentService {
    private static final String TAG = WebService.class.getName();
    public static final int GET = 1;
    public static final int POST = 2;

    public static final String INTENT_KEY_REQUEST_TYPE =
"com.wifill.eat.INTENT_KEY_REQUEST_TYPE";
    public static final String INTENT_KEY_JSON = "com.wifill.eat.INTENT_KEY_JSON";
    public static final String INTENT_KEY_RECEIVER =
"com.wifill.eat.INTENT_KEY_RECEIVER";
    public static final String BUNDLE_KEY_REQUEST_RESULT =
"com.wifill.eat.BUNDLE_KEY_REQUEST_RESULT";
```

```
    public WebService() {
        super(TAG);
    }

    @Override
    protected void onHandleIntent(Intent intent) {

        Uri uri = intent.getData(); ❶ int requestType =
intent.getIntExtra(INTENT_KEY_REQUEST_TYPE, 0);
        String json = (String)intent.getSerializableExtra(INTENT_KEY_JSON);
        ResultReceiver receiver = intent.getParcelableExtra(INTENT_KEY_RECEIVER);

        try {
            HttpRequestBase request = null;
            switch (requestType) { ❷case GET: {
                    request = new HttpGet();
                    // Request setup omitted
                    break;
                }
                case POST: {
                    request = new HttpPost();
                    if (json != null) {
                        ((HttpPost)request).setEntity(new StringEntity(json));
                    }
                    // Request setup omitted
                    break;
                }
            }

            if (request != null) {
                request.setURI(new URI(uri.toString()));
                HttpResponse response = doRequest(request); ❸HttpEntity httpEntity =
response.getEntity();
                StatusLine responseStatus = response.getStatusLine();
                int statusCode = responseStatus != null ? responseStatus.getStatusCode() :
0;

                if (httpEntity != null) {
                    Bundle resultBundle = new Bundle();
```

```
                    resultBundle.putString(BUNDLE_KEY_REQUEST_RESULT,
EntityUtils.toString(httpEntity));
                    receiver.send(statusCode, resultBundle); ❹}
                else {
                    receiver.send(statusCode, null);
                }
            }
            else {
                receiver.send(0, null);

            }
        }
        catch (IOException e) {
            receiver.send(0, null);
        } catch (URISyntaxException e) {
            e.printStackTrace();
        }
    }

    private HttpResponse doRequest(HttpRequestBase request) throws IOException {
        HttpClient client = new DefaultHttpClient();

        // HttpClient configuration omitted

        return client.execute(request);
    }
}
```

❶

Retrieve the necessary data from the Intent.

❷

Create request type depending on Intent data.

❸

Do the network request.

❹

Return the successful result to the WebServiceActivity.

Asynchronous Execution in BroadcastReceiver

A BroadcastReceiver is an application entry point; i.e., it can be the first Android component to be started in the process. The start can be triggered from other applications or system services. Either way, the BroadcastReceiver receives an Intent in the onReceive callback—which is invoked on the UI thread. Hence, asynchronous execution is required if any long running operations shall be executed.

However, the BroadcastReceiver component is active only during the execution of onReceive. Thus, an asynchronous task may be left executing after the component is destroyed–leaving the process empty if the BroadcastReceiver was the entry point–which potentially makes the runtime kill the process before the task is finished. The result of the task is then lost.

To circumvent the problem of an empty process, the IntentService is an ideal candidate for asynchronous execution from a BroadcastReceiver. Once a start request is sent from the BroadcastReceiver, it is not a problem that onReceive finishes because a new component is active during the background execution.

1. Prolonged lifetime with goAsync

As of API level 11, the BroadcastReceiver.goAsync() method is available to simplify asynchronous execution. It keeps the state of the asynchronous result in a BroadcastReceiver.PendingResult and extends the lifetime of the broadcast until the BroadcastReceiver.PendingResult is explicitly terminated with finish, which can be called after the asynchronous execution is done.

A minimalistic asynchronous receiver is shown in AsyncReceiver, where the BroadcastReceiver is kept alive until the PendingResult is finished:

```
public class AsyncReceiver extends BroadcastReceiver {
    public void onReceive(Context context, Intent intent) {
        final PendingResult result = goAsync();

        new Thread() {
            public void run() {

                // Do background work

                result.finish();
            }
        }.start();
    }
```

```
}
```

Example: Periodical Long Operations

Applications that should trigger periodical tasks even when the applications themselves are not executing can utilize the AlarmManager system service in the platform. It can be configured with a periodic interval when it will send an Intent to an BroadcastReceiver in the application. Thus, if the application is not running, the BroadcastReceiver is the entry point of the application and long-running operations should be executed in another component, typically an IntentService.

This example checks a network resource to see whether any updates have been madesince the last time the IntentService ran.[17] If so, a notification is added to the status bar.

The BroadcastReceiver and AlarmManager are set up in an Activity:

```
public class AlarmBroadcastActivity extends Activity {
    private static final long ONE_HOUR = 60 * 60 * 1000;

    AlarmManager am;
    AlarmReceiver alarmReceiver;

    public void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);

        alarmReceiver = new AlarmReceiver();
        registerReceiver(alarmReceiver, new IntentFilter("com.wilfill.eat.alarmreceiver"));
❶PendingIntent pendingIntent = PendingIntent.getBroadcast(this, 0,
            new Intent("com.wilfill.eat.alarmreceiver"),
PendingIntent.FLAG_UPDATE_CURRENT);

        am = (AlarmManager)(this.getSystemService( Context.ALARM_SERVICE ));
        am.setRepeating(AlarmManager.ELAPSED_REALTIME,
            SystemClock.elapsedRealtime() + ONE_HOUR, ONE_HOUR, pendingIntent);
❷}
}
```

❶

Register the BroadcastReceiver that will receive Intent from the AlarmManager.

❷

Configure the AlarmManager so that it starts the application every hour.

The AlarmReceiver is started every hour and redirects the invocation to an IntentService that can handle the network operation:

```
public class AlarmReceiver extends BroadcastReceiver {
    public void onReceive(Context context, Intent intent) {
        context.startService(new Intent(context, NetworkCheckerIntentService.class));
    }
}
```

The NetworkCheckerIntentService receives the start request in onHandleIntent, makes a network call, and possibly updates the status bar:

```
public class NetworkCheckerIntentService extends IntentService {

    public NetworkCheckerIntentService() {
        super("NetworkCheckerThread");
    }

    @Override
    protected void onHandleIntent(Intent intent) {
        if (isNewNetworkDataAvailable()) { ❶addStatusBarNotification();
        }
    }

    private boolean isNewNetworkDataAvailable() {
        // Network request code omitted. Return dummy result.
        return true;
    }

    private void addStatusBarNotification() {
        Notification.Builder mBuilder =
                new Notification.Builder(this)
                        .setSmallIcon(R.drawable.new_data_available)
                        .setContentTitle("New network data")
                        .setContentText("New data can be downloaded.");

        NotificationManager mNotificationManager =
                (NotificationManager)
getSystemService(Context.NOTIFICATION_SERVICE);
```

```
            mNotificationManager.notify(1, mBuilder.build());
        }
}
```

❶

Contains the network call.

IntentService Versus Service

The IntentService inherits its character from the Service: same declaration, same impact on process rank, and same start request procedure for clients. It implements the start request handling semantics of the Service so that an application that uses IntentService just has to implement onHandleIntent. Thus, the use of IntentService matches the commonly used task-controlled Service (Task-Controlled Service), but with built-in support for asynchronous execution and component lifecycle management.

The IntentService is appealingly simple to use and is often the perfect solution for the right use case, such as the one just described. However, the simplicity comes with limitations, and a Service may be preferred:

Control by clients
: When you want the lifecycle of the component to be controlled by other components, choose a user-controlled +Service (User-Controlled Service). This goes for both started and bound services.

Concurrent task execution
: To execute tasks concurrently, starting multiple threads in Service.

Sequential and rearrangeable tasks
: Tasks can be prioritized so that the task queue can be bypassed. For example a music service that is controlled by buttons—play, pause, rewind, fast-forward, stop, etc.—would typically prioritize a stop request so that it is executed prior to any other tasks in the queue. This requires a Service.

Summary

The IntentService is an easy-to-use, sequential task processor that is very useful for offloading operations not only from the UI thread, but also from other originating components. Other sequential task processors discussed in this book, such as HandlerThread, Executors.newSingleThreadExecutor, and to some extent AsyncTask, can be compared to the IntentService, but IntentService has the advantage of running as an independent component, which the others do not.

[16] This is the basis of a REST interface.

[17] Use Google Cloud Messaging if applicable, before implementing your own mechanism.