



Programação de Dispositivos Móveis

Aula 7

Licenciatura em Engenharia Informática
Licenciatura em Informática Web

Sumário

Execução de tarefas e programas sem interface de utilizador em dispositivos móveis. Notificações, mensagens flutuantes não intrusivas e intentos pendentes. Comunicações entre processos através de vínculos entre *threads* e de mensagens para o sistema.

Programming of Mobile Devices

Lecture 7

Degree in Computer Science and Engineering
Degree in Web Informatics

Summary

Execution of tasks and programs with no user interface in mobile devices. System notifications, non intrusive floating messages and pending intents. Inter-processes communication via binding between threads and system messages.

1 A Componente Serviço em Android™

The Android™ Service Component

1.1 Introdução

Introduction

Anteriormente, foram discriminados os **4 componentes** que podem ser usados como **blocos de construção de uma aplicação Android™**, embora ainda só se tenha discutido com detalhe a **componente Atividade**. Esta componente é a que **lida com interfaces de utilizador**, sendo por isso **central à maior parte das aplicações Android™**. As restantes componentes servem diferentes propósitos, e esta secção foca-se, sobretudo, na componente *Serviço*.

Note que, atualmente, é comum ao utilizador de um computador **iniciar várias tarefas ao mesmo tempo**, ou **iniciar determinada tarefa e deixá-la a executar em segundo plano** enquanto este se foca noutra atividade. Por exemplo, é comum continuar a navegar na *web* enquanto se descarrega um ficheiro, ou então ler um documento enquanto se ouve uma música. Algumas vezes, inicia-se uma aplicação apenas para colocar determinada ação a correr em segundo plano, atirando essa aplicação para segundo plano logo de seguida. Se recordar a forma de operar das *atividades*, bem como o seu ciclo de vida, **irá concluir que este tipo de funcionalidade não pode ser conseguida com essa componente**.

As aplicações Android™ correm numa máquina virtual *java*. Assim, determinada instância de execução **corresponde apenas a um processo, onde primariamente corre a *main thread*** (por vezes também designada por *user interface thread* ou *UI thread*). É óbvio que o modelo de execução em que é **apenas usada uma única *thread* não pode responder a todos os cenários de utilização de uma aplicação móvel**. O uso de uma

só *thread* significa que qualquer conjunto de operações é executado de uma forma sequencial, pelo que a presença de uma operação lenta irá incorrer numa situação em que a aplicação deixa de responder, o que é inadmissível em termos de *user experience* (UX), também por construção, em sistemas atuais. Por exemplo, o Android™ define **o tempo máximo de reação das suas aplicações para os 5 segundos**. Depois disso, o sistema assume controlo, mostrando a **mensagem *Application not Responding* (ANR)** ao utilizador, oferecendo-lhe a hipótese de este terminar a aplicação.

A solução passa por colocar diferentes tarefas a executar de forma assíncrona, o que se concretiza em colocá-las a correr em diferentes processos ou diferentes *threads*. A plataforma Android™ suporta o processamento em segundo plano através de **4 formas distintas**:

1. A classe *Threads* está disponível em *java.lang.Thread* e pode ser usada para **processamento assíncrono**. Também é possível usar software do pacote *java.util.concurrent* para executar tarefas em segundo plano (e.g., usando classes como *ThreadPools* ou *Executor*). Contudo, conforme já sugerido em cima, **é preciso ter em conta que as *threads* criadas desta forma não podem atualizar a interface de utilizador diretamente**, visto estarem a correr isoladamente da *thread* principal. Para se conseguir esse efeito, **é preciso garantir que se comunica à *thread* principal quais as alterações a fazer**. As classes *Handler* e *AsyncTasks*, que não são discutidas com detalhe aqui, facilitam essa comunicação;

2. A classe *Handler* (*android.os.handler*)¹ permite definir um manipulador na *thread* principal, para o

¹Ver <http://developer.android.com/reference/android/os/Handler.html>.

qual se podem enviar mensagens ou código para ser executado (e.g., via método `post(Runnable)`). Neste caso, **declara-se a tarefa que se quer executar** de forma assíncrona **dentro de uma nova thread**, e todas as **operações de atualização da interface de utilizador são definidas** dentro de uma classe `Runnable`, **enviadas para serem realizadas pela thread principal** através do método `post(Runnable)`. Pode optar-se também por se definir uma mensagem (classe `Message`) a ser tratada na *thread* principal. Neste caso, é necessário reescrever o método `handleMessage(Message)` na *thread* principal, e colocar nele o código que atualiza a interface de utilizador;

3. A classe `AsyncTask` (`android.os.AsyncTask`)² pode ser **usada para criar threads** que facilmente comunicam com a *UI thread*, já que **define 4 métodos que podem ser reescritos**, sendo que **alguns correm na thread em segundo plano** (e.g., `doInBackground()`), **enquanto que outros** (e.g., `onProgressUpdate()` ou `onPostExecute()`) **correm na thread onde o objeto é criado**, que normalmente corresponde à *thread* principal.

4. Finalmente, a classe `Service` (`android.app.Service`)³, definida a seguir.

1.2 Definição de Serviço

Service Definition

Dado os seus propósitos, o serviço é uma das componentes mais importantes da plataforma.

Em Android™, um serviço (`Service`) não é mais do que **uma forma de anunciar o desejo de uma aplicação Android™ executar uma operação demorada sem interação com o utilizador**, ou então definir uma **forma de fornecer funcionalidades a outras aplicações**, que se podem vincular ao serviço para obter essas funcionalidade.

Convém endereçar algumas confusões que se geram quando se mencionam serviços, nomeadamente que:

- Um serviço **não é um processo separado** e que;
- Um serviço **não é, nem cria, uma thread** (separada).

Isto significa que, **a não ser que seja estritamente definido em contrário, um serviço corre no mesmo processo da aplicação**. Também significa que, **caso se queira fazer trabalho demorado**

dentro de um serviço, **uma nova thread** para lidar com esse trabalho **deve ser explicitamente declarada pelo programador**.

Uma das **características que melhor distingue um serviço** de uma atividade é o facto de **não ter uma interface de utilizador**. Os exemplos mais comuns dados no contexto dos serviços referem-se às **funcionalidades de download de ficheiros** ou de **ouvir música**. Para obter uma ideia prática da utilidade dos serviços, considere que um utilizador iniciava o *download* de um ficheiro a partir de uma atividade, e que era a própria atividade que lidava com esse *download* até terminar. Neste caso, e assim que o utilizador saísse da atividade (e.g., para ir para outra aplicação ou para o *home screen*), o *download* era interrompido, porque a atividade era suspensa (`onPause()` → `onStop()`). Por outro lado, o *download* do ficheiro **não seria interrompido caso as instruções relacionadas com o download fossem colocadas num serviço**, ainda que este execute na *thread* principal, porque o Android™ **deixa o processo da aplicação a correr em segundo plano**.

Enquanto que as *threads* criam módulos de execução separados dentro do mesmo processo, os serviços **declaram ao sistema que determinada tarefa deve continuar, ainda que outros processos/aplicações/atividades sejam trazidos para primeiro plano**. Os serviços permitem que outros componentes se vinculem a estes para obter funcionalidades e são por vezes usados para *InterProcess Communication* (IPC).

Existem basicamente **dois tipos de serviços**:

1. Serviços **sem vínculo** ou *started*, que são **colocados em execução por outro componente (como uma atividade) através do método `startService()`**. Uma vez iniciados, os serviços sem vínculo **podem correr indefinidamente em segundo plano, mesmo que o componente que os colocou em execução seja destruído**. Normalmente, os serviços deste tipo **fazem apenas uma operação e não devolvem resultados** para a componente que os invocou (e.g., fazem a atualização de uma base de dados). Adicionalmente, depois de cumprirem o seu destino, **o serviço deve auto terminar-se**;
2. Serviços **com vínculo** ou *bound*, que são **colocados em execução ou criado o vínculo através do método `bindService()`**. Este tipo de serviços providenciam **formas de definir uma interface que permite que outros componentes da mesma aplicação, ou de aplicações diferentes, interajam com o serviço** numa arquitetura cliente-servidor, **enviando-lhe pedidos e obtendo resultados**. É possível que **várias componentes se liguem simultaneamente a um serviço e este só sobrevive enquanto tiver componentes vinculadas** (i.e., após perder o último vínculo, o serviço é destruído).

²Ver <http://developer.android.com/reference/android/os/AsyncTask.html>.

³Ver <http://developer.android.com/reference/android/app/Service.html>.

Note que **um mesmo Serviço pode funcionar em modo *started* ou *bound*, dependendo de como é invocado** e dos métodos da sua superclasse que são implementados. Por exemplo, **um Serviço que funcione em modo *started* deve implementar o método `onStartCommand(...)`, enquanto que um Serviço que funcione em modo *bound* deve implementar o método `onBind()`** (ver em baixo), mas **não há nada que impeça que ambas sejam implementadas**, para que o serviço possa ser despoletado das duas formas.

1.3 Ciclo de Vida de um Serviço

Service Life Cycle

A figura ??, retirada diretamente da documentação oficial da plataforma Android™⁴, ilustra os **dois ciclos de vida que uma componente serviço pode percorrer, dependendo do seu tipo** (*started* ou *bound*). Como se pode concluir da observação da figura, **ambos os ciclos de vida invocam os métodos `onCreate()` e `onDestroy()`**, cujo conteúdo é executado quando o serviço é colocado em execução pela primeira vez ou imediatamente antes de ser destruído, respetivamente. Contudo, o método `onCreate()` não recebe, para os serviços, nenhum Bundle, como de resto acontece para Atividades.

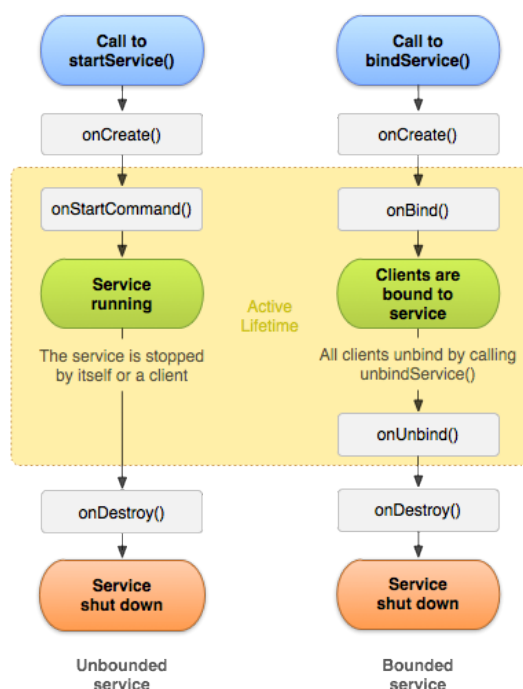


Figura 1: Os ciclos de vida de um serviço Android™. À esquerda é mostrado o ciclo de vida para serviços criados com `startService()`, enquanto que à direita está o ciclo de vida para o caso em que é criado com `bindService()`.

De modo a deixar claro quais **os parâmetros e retornos** de cada um dos métodos esquematizados na figura, incluem-se, em baixo, **os protótipos** de cada um deles.

⁴Ver <http://developer.android.com/guide/components/services.html>.

Enfatiza-se que **não é necessário reescrever todos os métodos** durante a implementação, e que alguns deles **não são**, inclusive, **necessários, dependendo do tipo de Serviço** (e.g., o método `onBind()` nunca é invocado para um serviço que funcione sempre sem vínculo).

```
public class ExampleService extends Service {
    @Override
    public void onCreate() {...}

    @Override
    public int onStartCommand(Intent intent, int flags, int startId) {...}

    @Override
    public IBinder onBind(Intent intent) {...}

    @Override
    public boolean onUnbind(Intent intent) {...}

    @Override
    public void onRebind(Intent intent) {...}

    @Override
    public void onDestroy() {...}
}
```

O **ciclo de vida** de um Serviço **começa pela invocação de `onCreate()`, onde devem ser feita a configuração inicial da instância do componente**, e pode terminar com a invocação de `onDestroy()`. Por exemplo, caso queira colocar música a correr em segundo plano, é no `onCreate()` que deve colocar o código que declara e configura a *thread* secundária. **O período de atividade** do ciclo de vida de um Serviço **começa com a invocação de um dos métodos `onStartCommand(...)` ou `onBind()`. A cada um destes métodos é automaticamente entregue o intento que invocou o Serviço** (como não pode deixar de ser, e sendo um Serviço um componente aplicacional Android™, este é invocado através de um intento). Contrariamente aos Serviços sem vínculo, **os com vínculo só terminam depois do método `onUnbind()` retornar**.

1.4 Criar um Serviço sem Vínculo

Creating an Unbinded Service

Para efeitos de exemplificação e demonstração da forma de criação de um **Serviço sem vínculo**, considere que queria criar uma aplicação cuja única finalidade era mostrar 10 avisos no ecrã em intervalos de 2 em 2 segundos. Os avisos devem ter o texto Warning Number 1, Warning Number 2, ..., Warning Number 10 e ser exibidos mesmo que a atividade principal da aplicação saia de foco, i.e., mesmo que o utilizador passe a utilizar outra aplicação. Para o exemplo seguinte, e para conseguir o objetivo de mostrar as mensagens, faz-se uso de *toasts* (ver adiante). Note que, conforme descrito, o cenário parece ideal para um Serviço. Assuma ainda que o Serviço só é despoletado depois de um botão associado a `onButtonClick()` ser clicado. Aparte alguns detalhes, o código que implementa a atividade principal terá uma implementação semelhante a:

```
package pt.di.ubi.pmd.exservice;
```

```
import android.app.Activity;
...

public class FloatingAlarms extends Activity {

    @Override
    protected void onCreate(Bundle state){
        super.onCreate(state);
        setContentView(R.layout.main);
    }
    public void onClick(View v){
        Intent oIntent = new Intent(this, ServiceAlarms.
            class);
        startService(oIntent);
    }
}
```

Enquanto que o respetivo serviço poderia ser implementado como se mostra a seguir:

```
package pt.di.ubi.pmd.exservice;

import android.app.Service;
...

public class ServiceAlarms extends Service {

    @Override
    public int onStartCommand
    (Intent intent, int flags, int startId) {
        for(int i=1; i<11; i++){
            Toast.makeText
            (this, "Warning #" + i, Toast.LENGTH_SHORT)
            .show();
            Thread.sleep(3000);
        }
        // Se o processo for morto, nao voltar a
        // tentar reinicia-lo (START_NOT_STICKY)
        return START_NOT_STICKY;
    }
}
```

Como se pode ver pelo exemplo, **criar um Serviço em Android™ envolve importar e estender a classe Service, contida no pacote android.app.Service** (envolve também declarar o Serviço no manifesto, conforme discutido em baixo). **Pode então ser necessário reescrever alguns dos métodos da superclasse**, nomeadamente o `onCreate()`, `onStartCommand()`⁵ e `onDestroy()`. Um dos **métodos mais importantes do ciclo de vida é o `onStartCommand()`, onde basicamente devem ser tomadas as providências necessárias para que as tarefas que definem o Serviço sejam desempenhadas**. No exemplo anterior, é neste método que se coloca um ciclo `for` que itera 10 vezes, de 3 em 3 segundos, mostrando uma mensagem em cada uma dessas iterações.

⁵Em APIs mais antigas (de nível inferior a 5), o método utilizado chamava-se `onStart()`, pelo que pode também implementar este método para efeitos de compatibilidade.

O serviço é começado por outra componente através de um intento. Neste caso, o **intento é explícito** porque na sua instanciação é indicado claramente qual é o seu componente destino, conforme se mostra a seguir:

```
Intent oIntent = new Intent(this, ServiceAlarms.
    class);
```

Neste caso, **o intento é passado ao método `startService(intent)`**. Após ser despoletado, o Serviço **passa automaticamente pelo método `onCreate()`** (não implementado no exemplo) **e depois para o método `onStartCommand(Intent,int,int)`**. O método mencionado em último **recebe o intento** enviado pela componente que o invocou.

Para que possa **ser visível para o sistema**, o Serviço deve **ser declarado no `AndroidManifest.xml` usando uma *tag* `<service>`** que pode ter vários atributos e sub-elementos, **inclusive o nome (`android:name`) e alguns filtros de intents** (usando a *tag* `<intent-filter>`). A seguir inclui-se o excerto que declararia o serviço mencionado antes no manifesto da respetiva aplicação:

```
<service android:name="ServiceAlarms">
    <intent-filter>
        <action android:name="pt.ubi.di.pmd.ServiceAlarms.
            SERVICE" ></action>
    </intent-filter>
</service>
```

Recorde que **os filtros de intents são usados pelo sistema para encontrar componentes que são capazes de lidar com determinada ação**. No fundo, estes filtros definem capacidades desses componentes e **podem ser definidos mais do que um filtro para cada componente**. Para o exemplo nesta secção, e dado o seu manifesto, pode-se dizer que **o sistema também permitia despoletar este Serviço via um intento implícito** que definisse a ação `pt.ubi.di.pmd.ServiceAlarms.SERVICE`, i.e., via

```
Intent service = new Intent("pt.ubi.di.pmd.
    ServiceAlarms.SERVICE");
startService(service);
```

Relembre também que os filtros de intents **permitem especificar, com mais granularidade, quais os os intents que são entregues a determinado componente**. Contudo, **estes filtros apenas só se aplicam a intents implícitos**, já que os intents explícitos são sempre entregues ao componente destino.

O método `onStartCommand(...)` **termina com um retorno** (no exemplo, é devolvido o valor `START_NOT_STICKY`). Há **3 retornos importantes** para este método:

1. O valor `START_NOT_STICKY` **define que caso o serviço seja morto pelo sistema enquanto está no estado ativo, este não deve voltar a tentar criá-lo se puder**. É útil para Serviços que são recomeçados por tarefas agendadas e que mesmo que sejam mortos por falta de memória, serão eventualmente recomeçados quando o agendamento se der;

2. O valor `START_STICKY` define que o sistema deve tentar a recomençar um serviço que matou logo que tenha recursos para isso. Neste caso, há que ter em atenção que o método `onStartCommand(...)` é invocado de novo, mas sem o intento que originalmente o chamou (o parâmetro do intento vai a `null`). Isto significa que é preciso lidar com este facto no próprio código, caso o intento seja preciso para o seu bom funcionamento;
3. O valor `START_REDELIVER_INTENT` é parecido ao anterior, mas indica ao sistema que este deve guardar o intento que criou determinado Serviço antes de o matar, para que possa mais tarde ser utilizado para o recriar com esse intento.

O exemplo contido no início desta secção não é, na realidade, um bom exemplo de como se deve criar ou implementar um Serviço. Há pelo menos dois detalhes que justificam esta afirmação:

1. Como o serviço corre na *thread* principal, colocar a *thread* a dormir corresponde a colocar a interface do utilizador num modo adormecido e que não responde. Caso o número de segundos seja aumentado de 2 para 5 ou superior, será exibida uma mensagem ANR;
2. Quando se cria um serviço sem vínculo deve sempre garantir-se que este termina quando o trabalho que tem para fazer também termina.

O excerto de código seguinte endereça ambos os problemas mencionados antes. Neste exemplo, é criada uma *thread* que irá executar a tarefa demorada definida como um objeto da classe *Runnable*. É também invocado o método `stopSelf()` depois do trabalho estar feito, efetivamente terminando o serviço.

```
package pt.di.ubi.pmd.exservice;

import android.app.Service;
import java.lang.Thread;
import java.lang.Runnable;
...

public class ServiceAlarms extends Service {

    @Override
    public int onStartCommand
        (Intent intent, int flags, int startId) {
        Runnable oTask = new Runnable() {
            @Override
            public void run() {
                try {

                    for(int i=1; i<11; i++){
                        Toast.makeText(this, "Warning #" + i,
                            Toast.LENGTH_SHORT).show();
                        Thread.sleep(3000);
                    }
                    stopSelf();

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

```
}

new Thread(oTask).start();

// Se o processo for morto, não voltar a
// tentar reiniciá-lo (START_NOT_STICKY)
return START_NOT_STICKY;
}
```

Recorde ou fique a saber que, ao ser chamado o método `start()` da classe *Thread* é automaticamente executado o método `run()`, definido para uma instância da classe *Runnable*. O método `run()` é executado numa nova *thread*, conforme sugerido por `new Thread(oTask).start()`; enquanto que o Serviço continua a correr na *thread* principal.

Note que o código anterior ainda contém um erro crasso (serve de lição). A *toast* está a ser invocada dentro de uma *thread* diferente da principal (que é a que trata da interface gráfica), pelo que não irá ser exibida.

1.5 A Classe *IntentService*

The IntentService Class

No final da secção anterior ficou claro que é muito frequente ser necessário definir uma nova *thread* para lidar com tarefas longas que correm em segundo plano. Devido a esse facto, a plataforma Android™ disponibiliza a classe chamada *IntentService* no pacote `android.app`, que cria, de forma transparente para o programador, uma *thread* separada da principal, que executa todos os intentos entregues na `onStartCommand()`. Na verdade, uma instância desta classe cria uma pilha de trabalho que permite lidar com vários intentos que sejam enviados para a componente, de forma sequencial. É também invocado, por defeito e automaticamente, o método `stopSelf()` assim que todas as tarefas tenham sido concluídas, evitando o problema de este poder ser esquecido. Adicionalmente, fornece uma implementação por defeito dos métodos `onBind()` (ver em baixo), para o qual devolve sempre `null`, e `onStartCommand()`, no qual é enviado o intento para a pilha de trabalho e depois para um método `onHandleIntent()`.

```
public class ServiceAlarms extends IntentService {
    // E necessario implementar um construtor simples,
    // invocando o super IntentService(String), de
    // forma a dar um nome a thread criada.
    public ServiceAlarms() {
        super("ServiceAlarms");
    }

    @Override
    protected void onHandleIntent(Intent intent) {
        for(int i=1; i<11; i++){
            Toast.makeText(this, "Warning #" + i, Toast.
                LENGTH_SHORT).show();
            Thread.sleep(3000);
        }
    }
}
```

Para utilizar esta classe, um programador apenas tem de reescrever o método `onHandleIntent()` e implementar o construtor que simplesmente invoca o seu análogo da super classe, fornecendo-lhe um nome para a *thread* que é criada. O excerto de código anterior exemplifica o que foi dito nesta secção, constituindo uma alternativa ao código que foi mencionado em último na secção anterior.

1.6 Criar um Serviço com Vínculo

Creating a Binded Service

Para muitos casos, é útil criar um vínculo entre a componente que invoca o Serviço e este último. Por exemplo, uma aplicação de música pode oferecer uma interface de utilizador para controlar se a música está a tocar ou pausada, e colocar um Serviço em segundo plano a tocar essa música, para que o utilizador possa continuar a usar o dispositivo móvel entretanto. Neste caso, a atividade que oferece a interface de utilizador precisa pedir ao Serviço que pare ou coloque determinada música a tocar, dependendo dos *inputs* do utilizador. A forma de conseguir esse objetivo é através da criação de um vínculo que funciona no modelo cliente-servidor, sendo que o Serviço toma o papel de servidor e a componente que o invoca toma o papel do cliente.

Há três formas básicas de definir um vínculo (*binder*) entre as componentes:

1. Estender a classe `Binder` (`android.os.Binder`), aplicável em situações em que o Serviço é apenas usado pela própria aplicação e corre no mesmo processo do cliente, o que acontece com frequência. Esta opção não é aplicável no caso em que o Serviço é implementado para ser usado por várias aplicações. Em baixo elabora-se um pouco mais nesta opção;
2. Combinar objetos das classes `Messenger` (`android.os.Messenger`) e `Handler` (`android.os.Handler`), em que um manipulador é criado num Serviço, e um conjunto de mensagens a tratar é definido programaticamente no seu código. As mensagens são enviadas para o serviço através do `Messenger`, sendo também possível ao cliente criar um destes objetos e enviá-lo via uma mensagem ao Serviço, para que este lhe possa enviar retorno. Esta é a forma mais simples de criar IPC, já que as mensagens enviadas pelos objetos da classe `Messenger` são todas colocadas numa única fila e tratadas numa só *thread*, evitando problemas de sincronização;
3. Finalmente, é possível usar **Android Interface Definition Language (AIDL)** para definir interfaces que determinado Serviço quer expor a clientes. Estas interfaces estão disponíveis para aplicações diferentes daquela em que a componente é definida. A definição é feita em ficheiros `.aidl` que

são colocados nas diretorias `src/` da aplicação que disponibiliza o Serviço e de todas as que o usam. Aquando da compilação, as ferramentas do SDK criam uma interface `IBinder` com base no conteúdo de do ficheiro `.aidl` e guardam o resultado em `gen/`. As interfaces podem depois ser implementadas no Serviço respetivo e invocadas nos clientes, com instruções semelhantes às seguintes:

```
IRemoteService oRS;  
private ServiceConnection mConnection = new  
    ServiceConnection() {  
        public void onServiceConnected(ComponentName  
            className, IBinder service) {  
            oRS = IRemoteService.Stub.asInterface(  
                service);  
        }  
    }
```

Se se optar por esta possibilidade, é preciso ter em consideração que não é garantido, por defeito, que as chamadas aos vários métodos da interface sejam feitas na *thread* principal, pelo que devem ser tomadas precauções relativas a **multi-threading**. O serviço deve ser implementado de forma a ser *thread-safe*⁶.

A explicação subsequente irá apenas elaborar na primeira das três formas de criar um vínculo entre um Serviço e outra componente aplicacional Android™. Recorde que, para esta opção, o vínculo é interno, i.e., entre componentes da mesma aplicação e que correm no mesmo processo. As condições gerais que permitem a criação do vínculo são os seguintes:

1. No Serviço, há que instanciar um objeto da classe `Binder` que:
 - Contém métodos públicos que o cliente poderá chamar; ou
 - Devolve uma instância de outra classe, cuja definição está contida na do Serviço e que contém métodos públicos que o cliente pode invocar; ou
 - Devolve a própria instância do Serviço atual, permitindo que o cliente invoque os seus métodos públicos;
2. Ainda no Serviço, há que devolver a instância da classe `Binder` no método *callback* `onBind()`;
3. No cliente, há que receber o `Binder` da função *callback* `onServiceConnected()`, bem como instanciar um objeto local da mesma classe do objeto instanciado no `Binder`, e tirar partido dos métodos públicos por ele fornecidos.

O exemplo seguinte mostra como é que esta forma de criar um vínculo pode funcionar na prática. Começa-se

⁶Ser *thread-safe* significa que a execução concorrente de várias operações que acedem aos mesmos recursos não incorrem em estados de inconsistência no estado da aplicação.

por mostrar a implementação de um Serviço simples chamado ToastasMistas, cujo único objetivo é lançar mensagens flutuantes com a expressão Toastas!, sempre que o método dizToastas() é invocado:

```
...
public class ToastasMistas extends Service {
    private final BinderLocal oBinder = new
        BinderLocal();

    // Redefinicao da classe Binder
    public class BinderLocal extends Binder {
        ToastasMistas getService() {
            // Devolve a classe do proprio servico
            return ToastasMistas.this;
        }
    }

    @Override
    public Binder onBind(Intent intent) {
        return oBinder;
    }

    // Metodo publico que pode
    // ser usado por um cliente.
    public int dizToastas() {
        Toast.makeText(this, "Toastas!",
            Toast.LENGTH_SHORT).show();
    }
}
```

Note que, tal como mencionado em cima, é estendida a classe Binder e tomada a opção de devolver a instância do Serviço em si através da instrução return ToastasMistas.this;. Isto significa que o método dizToastas() será o que estará disponível na componente que receber o Binder.

De seguida mostra-se a implementação da atividade principal, que cria o layout da aplicação e que despoleta o Serviço referido antes através do método bindService(intent, mConnection, int):

```
...
public class FloatingToastas extends Activity {
    ToastasMistas oTM;
    boolean bVinculo = false;

    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.main);
    }

    @Override
    protected void onStart() {
        super.onStart();
        // Fazer o vinculo ao Servico ToastasMistas.
        Intent oIntent = new Intent(this, ToastasMistas.
            class);
        bindService(oIntent, mConnection, Context.
            BIND_AUTO_CREATE);
    }

    @Override
    protected void onStop() {
        super.onStop();
        // Desfazer o vinculo do Servico.
        if (bVinculo) {
            unbindService(mConnection);
            bVinculo = false;
        }
    }
}
```

```
// Quando o botao e clicado, o Servico mostra
// uma mensagem no ecrã.
public void onClick(View v) {
    if (bVinculo) {
        // Pode-se chamar um metodo publico
        // diretamente do Servico.
        oTM.dizToastas();
    }
}

// A classe seguinte define os metodos
// callback que sao passados para bindService().
private ServiceConnection mConnection =
new ServiceConnection() {
    @Override
    public void onServiceConnected(
        ComponentName className, IBinder service) {
        // Fazer o cast do binder para
        // ToastasMistas.BinderLocal.
        BinderLocal binder =
            (ToastasMistas.BinderLocal) service;
        oTM = binder.getService();
        bVinculo = true;
    }

    @Override
    public void onServiceDisconnected
        (ComponentName arg0) {
        bVinculo = false;
    }
};
```

Há bastantes detalhes a notar no exemplo anterior. Por exemplo, que **o vínculo com o Serviço é criado apenas em onStart() e não em onCreate()**. Por outro lado, **o vínculo é destruído em onStop()**. Este forma de proceder **permite que o vínculo** (e, neste caso, também o Serviço) **só exista quando a atividade está em execução, libertando recursos quando esta está em segundo plano**. Também é notável que **o despoletar do Serviço** ou a criação do vínculo (se aquele já existir) se faça, como não podia deixar de ser, **através de um intento**. Ao ser invocado o método bindService(...), são chamados no Serviço os métodos onCreate() seguidos de onBind(), sendo que este último devolve um Binder. Este Binder é tratado no **método onServiceConnected(...)** **do cliente, que é automaticamente despoletado após a bindService(...)** **retornar**. Por fim, note que é o facto do Serviço e do cliente estarem na mesma aplicação e processo que permite que o cast no método onServiceConnected seja possível.

A criação do vínculo é assíncrona. Neste caso, o cliente que invoca bindService(.) tem de implementar uma instância da classe ServiceConnection e passá-la como parâmetro àquele método. **O método bindService(.) retorna imediatamente, e o IBinder só é devolvido, na realidade, quando o sistema Android™, eventualmente mais tarde, invocar o método callback onServiceConnected().**

No cliente, e de uma forma resumida, **a criação do vínculo ao Serviço é conseguida através dos dois passos seguintes:**

1. **Instancia-se um objeto da classe `ServiceConnection` e reescrevem-se os métodos `onServiceConnected()` e `onServiceDisconnected()`.** O primeiro método é invocado pelo Android™ para entrega do `IBinder` devolvido pelo método `onBind()` do Serviço. O segundo método é invocado automaticamente pelo Android™ quando a ligação ao serviço cai inesperadamente, nomeadamente quando o Serviço parou ou foi destruído. Note que, ao contrário do que se possa pensar, **este método não é invocado quando o cliente usa o método `unbindService()`**;
2. Invoca-se o método `bindService()`, passando-lhe o intento, a implementação específica de `ServiceConnection` e o contexto do cliente.

Note que apenas as **atividades, Serviços e provedores de conteúdos** (discutidos adiante) é que se **podem vincular a um serviço. Um recetor de conteúdos não o pode fazer.** Nos métodos *callback*⁷ `onServiceConnected()` deve ser colocada a lógica que, no mínimo, instancia a classe que contém os métodos públicos do Serviço, para que possam ser usados na componente atual.

O **terceiro parâmetro** do método `bindService()` é uma **flag** que especifica algumas opções para o vínculo e, portanto pode tomar vários valores, definidos também estaticamente na implementação da classe `Context`. No exemplo apresentado, é usada a **flag** `BIND_AUTO_CREATE`, que basicamente indica ao sistema que **o Serviço alvo deve ser despoletado caso não esteja a correr**. Outro exemplo inclui `BIND_IMPORTANT`, que indica ao sistema que **o serviço para o qual está a ser criado o vínculo é muito importante para o cliente e que, portanto, este deve ser promovido para o mesmo nível de execução em primeiro plano em que o cliente está**.

1.7 Terminar um Serviço

Ending a Service

Já aqui foi referida a importância de **terminar os Serviços quando estes não são necessários**. Cabe ao programador **ter essa tarefa em consideração**, mas com as seguintes **atenuantes**:

- Enquanto que os Serviços **sem vínculo devem ser explicitamente terminados**, quer **dentro do serviço**, após este terminar a tarefa para a qual tinha sido invocado (melhor solução), ou **por outra atividade** que a ele tenha acesso;
- Os serviços **com vínculo não precisam ser explicitamente terminados, desde que se garanta que**

⁷Recorde que os métodos *callback* são aqueles que são automaticamente executados pelo sistema convenientemente quando um método devolve algo.

é feita a **desvinculação** quando já não são necessários. A razão para esta forma de operar deve-se ao facto **do sistema matar automaticamente (e mais facilmente) os serviços que não têm qualquer vínculo**.

Já foi visto que **um Serviço *started* pode ser terminado num ponto da sua própria execução recorrendo ao método `stopSelf()`**. O mesmo tipo de Serviços pode ser **terminado usando o método `stopService(Intent)` a partir de uma componente** que a ele tenha acesso (e.g., a atividade que o despoletou). Como não podia deixar de ser, este método **aceita um intento** para identificar o Serviço a terminar, conforme se mostra no excerto de código seguinte:

```
Intent service = new Intent(this, ServiceAlarms.class);
stopService(service);
```

Para se **desvincular determinada componente de um Serviço** **recorre-se ao método `unbindService()`**, que **aceita o objeto da classe `ServiceConnection` como parâmetro**:

```
unbindService(mConnection);
```

Quanto um componente **cliente é destruído pelo sistema, também é automaticamente desvinculado** de qualquer Serviço ao qual tenha sido vinculado. De qualquer forma, é boa prática proceder à desvinculação no código, de forma a **poupar recursos quando o Serviço não estiver a ser usado pela aplicação**. Normalmente, a **vinculação e desvinculação é feita em momentos simétricos do ciclo de vida** da componente cliente, nomeadamente (para a componente atividade):

- Se a interação com o serviço só precisa acontecer enquanto **uma atividade está visível**, deve fazer-se a vinculação e desvinculação **nos métodos `onStart()` e `onStop()`, respetivamente**;
- Caso se queria que a **atividade receba respostas do Serviço mesmo enquanto está parada em segundo plano**, então o Serviço pode ser **vinculado no método `onCreate()` e desvinculado em `onDestroy()`**.

Note que **não é muito eficiente fazer a vinculação ou desvinculação a um Serviço nos métodos `onPause()` e `onResume()`**, principalmente porque isto causaria a **invocação dos métodos de *callback* a cada transição**, inclusive durante a apresentação de pequenos diálogos em que se desfoca a interface atual, causando algum **impacto na performance do sistema**. É preciso também ter em conta que, quando uma atividade se desvincula de um Serviço, este pode ser destruído pelo sistema, se não estiver ligado a mais nenhum componente. Caso a próxima atividade (e.g., da mesma aplicação) quisesse aceder a esse mesmo Serviço, este tem de ser recriado, enquanto que se não tivesse sido destruído, este passo não era necessário novamente.

2 Notificações e Mensagens Flutuantes

Notifications and Toast Messages

2.1 Introdução

Introduction

Dado um Serviço não possuir uma interface, é comum usar mensagens conhecidas por *toasts* ou notificações na barra de status para informar o utilizador de determinadas operações efetuadas por este componente. Enquanto que uma *toast* aparece na superfície da perspetiva atual, desaparecendo pouco tempo depois, uma notificação na barra de status é caracterizada por um ícone que fica residente até que a barra seja expandida e a mensagem que lhe está associada seja lida. Opcionalmente, o utilizador pode ainda atuar sobre essa mensagem através de uma ou mais ações que lhe são fornecidas na própria barra expandida (na gaveta). As notificações na barra de status são, portanto, ideais para aquelas tarefas que correm em segundo plano mas que necessitam que utilizador aja quando terminam (e.g., quando é feito o *download* do ficheiro).

2.2 Notificações via Mensagens Flutuantes

Notifications via Toast Messages

Na gíria Android™, uma *toast* é um objeto interativo (um *widget*) direcionado à exibição de mensagens curtas ao utilizador. Estas mensagens aparecem numa caixa de diálogo flutuante durante um período de tempo, desaparecendo depois disso de uma forma suave. Estas mensagens podem ser criadas de uma forma muito simples recorrendo à classe `Toast` (`android.widget.Toast`)⁸.

O trecho de código seguinte mostra como se pode despoletar uma mensagem do tipo discutido. No exemplo, o objeto `oToast` é primeiro instanciado (o construtor recebe o contexto da aplicação que a despoleta), depois configurado com a mensagem (através de `setText(CharSequence)`) e com o tempo de exibição (através de `setDuration(int)`), sendo por fim dada a ordem de exibição no ecrã por `oToast.show()`:

```
import android.widget.Toast;
...
Toast oToast = new Toast(this);
oToast.setDuration(Toast.LENGTH_LONG); // 3.5 seconds
oToast.setText("This is a toast message.");
oToast.show();
...
```

Note que o método `setDuration(int)` aceita um inteiro que, até à data, apenas pode tomar um de dois valores possíveis: `LENGTH_SHORT`, que equivale a uma exibição de 2 segundos, e `LENGTH_LONG`, que equivale a uma exibição com uma duração de 3.5 segundos. Estes

⁸<http://developer.android.com/reference/android/widget/Toast.html>

dois valores estão definidos estaticamente na classe `Toast`, para evitar enganos, mas correspondem apenas aos valores 0 e 1, respetivamente.

Da leitura do código anterior, fica claro que o ciclo de vida do objeto está confinado às operações de instanciação de um objeto, configuração e exibição da mensagem. Assim, sugere-se o uso de um dos dois métodos estáticos definidos na classe `makeText(...)`, que permitem combinar todas as operações numa única instrução, conforme se mostra a seguir:

```
import android.widget.Toast;
...
Toast.makeText(this, "This is a toast message.", Toast
    .LENGTH_LONG).show();
```

Note que ambos os métodos estáticos referidos têm o mesmo nome, sendo que apenas o tipo de parâmetro intermédio muda nas assinaturas e ambos devolvem um objeto da classe `Toast`. O método `makeText(context, CharSequence, int)` aceita o contexto da aplicação que a despoleta, bem com a mensagem a apresentar e o período, enquanto que `makeText(context, resID, int)` aceita um ID de um recurso, caso a mensagem a exibir esteja definida numa `strings.xml`, ao invés da própria *string*. Repare também no método `show()`, invocado no final da instrução. Sem esse método, a mensagem não é mostrada.

Se despoletada por um serviço, a *view* é mostrada sobre a interface de utilizador da aplicação atualmente em execução (seja a que lhe corresponde ou não). Contudo, a mensagem nunca chega a receber foco, pelo que o utilizador pode continuar a interagir com a aplicação. A ideia destas mensagens é precisamente serem o menos intrusivas possível, configurando o meio ideal para mostrar informação que pode ser útil (e.g., para avisar que terminou o *download* de um ficheiro).

2.3 Notificações na Barra de Estado

Status Bar Notifications

Uma notificação na barra de status é um objeto algo elaborado, principalmente porque permite interação e porque tem de necessariamente transmitir uma ideia ao utilizador da forma mais eficiente possível. Para tornar a criação de notificações mais simples, a plataforma fornece já outro objeto (um `NotificationCompat.Builder`) para construir estas notificações, sendo apenas necessário passar-lhe alguns elementos, antes de invocar o método `build()`, que debita já um objeto da classe pretendida. Os elementos necessários são os seguintes:

- O nome do recurso que aponta para um ícone (definido através de `setSmallIcon()`);
- Um título para a notificação (definido através de `setContentTitle()`);

- Uma pequena descrição (definido através de `setContentText()`).

O pequeno excerto de código seguinte mostra precisamente a criação de uma notificação simples recorrendo aos métodos e objetos referidos antes:

```
1 package pt.di.ubi.pmd.exservice;
2
3 import android.app.Service;
4 import android.app.NotificationManager;
5 ...
6
7 public class ServiceAlarms extends Service {
8     // A proxima variavel define um numero que
9     // identifica a notificacao e que pode ser
10    // usado para mais tarde a atualizar.
11    private int iID = 100;
12
13    @Override
14    public int onStartCommand
15    (Intent intent, int flags, int startId) {
16        ...
17        NotificationCompat.Builder oBuilder =
18        new NotificationCompat.Builder(this)
19        .setSmallIcon(R.drawable.icone_notificacao)
20        .setTitle("Floating Alarms")
21        .setContentText("O Servico dos alarmes terminou.
22        Carregue aqui para o reiniciar!");
23
24        NotificationManager oNM = (NotificationManager)
25        getSystemService(Context.NOTIFICATION_SERVICE);
26        oNM.notify(iID, oBuilder.build());
27    }
28 }
```

Note que o Java permite que a invocação dos 3 métodos referidos antes seja feita de forma **muito compacta**. No exemplo anterior, mostra-se que, ao mesmo tempo que se cria o objeto da classe Builder, são **imediatamente chamados os 3 métodos de forma contigua**, sem pontuação a indicar o final de cada método (à exceção do último), delineando apenas o seu início com um ponto. Na verdade, **passamos a ter uma só instrução terminada com o último ponto e vírgula**. As linhas 17, 18, 19, 20 e 21 são, por isso, equivalentes às seguintes:

```
NotificationCompat.Builder oBuilder =
new NotificationCompat.Builder(this);
oBuilder.setSmallIcon(R.drawable.icone_notificacao);
oBuilder.setTitle("Floating Alarms");
oBuilder.setContentText("O Servico dos alarmes terminou
. Carregue aqui para o reiniciar!");
```

A notificação é criada através do método `build()`, sendo imediatamente passada ao gestor de notificações do sistema na última linha de código útil do `onStartCommand()` através do método `notify(.,.)`⁹. A instância particular do gestor de notificações utilizada pelo sistema é obtida através da utilização do método do contexto `getService(String)`, cujo parâmetro determina o serviço Android™ que se quer obter. Muitos dos gestores disponíveis no sistema são obtidos desta forma.

Para além dos três elementos referidos antes, podem ser opcionalmente especificados outros. Na verdade, **é sempre uma boa ideia especificar pelo menos uma ação**¹⁰ para quando o utilizador clica na notificação na gaveta onde são apresentadas. Repare que o ideal será

⁹O primeiro parâmetro de `notify(.,.)` é um int que identifica a notificação e que pode ser usado para a cancelar remotamente.

¹⁰Podem especificar-se mais do que uma ação para determinada notificação.

inclusive redirecionar o utilizador para uma atividade onde passa interagir com a aplicação ou explorar melhor a mensagem (e.g., quando recebe uma notificação de *novo e-mail*, clicar na mesma leva-o para a aplicação de *e-mail*). **Para se conseguir esse efeito, tem de se construir um objeto conhecido por *Intento Pendente*** (i.e., um objeto da classe `PendingIntent`), conforme exemplifica o trecho de código seguinte:

```
1 package pt.di.ubi.pmd.exservice;
2
3 import android.app.Service;
4 import android.app.PendingIntent;
5 import android.app.NotificationManager;
6 ...
7
8 public class ServiceAlarms extends Service {
9     private int iID = 100;
10    private int iReqCode = 100;
11
12    @Override
13    public int onStartCommand
14    (Intent intent, int flags, int startId) {
15        ...
16        NotificationCompat.Builder oBuilder =
17        new NotificationCompat.Builder(this)
18        .setSmallIcon(R.drawable.icone_notificacao)
19        .setTitle("Floating Alarms")
20        .setContentText("O Servico dos alarmes terminou.
21        Carregue aqui para o reiniciar!");
22
23        // Intento explicito para uma determinada atividade.
24        Intent iMain = new Intent(this, FloatingAlarms.class);
25        PendingIntent iPMain = getActivity(this, iReqCode,
26        iMain, PendingIntent.FLAG_ONE_SHOT);
27
28        oBuilder.setContentIntent(iPMain);
29        NotificationManager oNM =
30        (NotificationManager) getSystemService(Context.
31        NOTIFICATION_SERVICE);
32
33        oNM.notify(iID, oBuilder.build());
34    }
35 }
```

No código incluído antes ilustra-se a criação de um intento que redireciona para a atividade principal da aplicação. Este intento é explícito, **encapsulado num intento pendente através de `PendingIntent iPMain = getActivity(.,.,.,.)`, e colocado na notificação que vai ser lançada (ver `oBuilder.setContentIntent(iPMain)`)**. Note que o intento toma a **qualificação de pendente porque simplesmente não é executado imediatamente, mas sim enviado para outra aplicação** (neste caso é enviado para o `NotificationManager`). Quando um utilizador carrega na mensagem da notificação, este intento é despoletado. Repare que, **como o intento pendente é criado com o contexto da aplicação, quando este é despoletado por outro componente, o sistema executa-o como se viesse da aplicação original, com as suas permissões**.

Um dos problemas que ainda aqui não foi referido, mas que deve merecer a consideração do programador é o facto da **pilha de retrocesso de atividades poder ficar inconsistente pela inserção de um intento pendente numa notificação**. Para obter uma ideia do problema, considere, por exemplo, que recebia uma notificação de nova mensagem de *e-mail* enquanto navegava no *browser*. Ao clicar na notificação, era redirecionado para a atividade de visualização do *e-mail*. Caso não sejam tomadas medidas em contrário, a pilha de atividades pas-

saria a conter a atividade do *e-mail* no topo, imediatamente precedida da do *browser*. O botão *back* do dispositivo móvel redirecionaria então o utilizador da atividade de visualização de *e-mail* para o *browser*, quando o ideal seria redirecioná-lo para a atividade principal da aplicação e depois para o ecrã *home* do sistema. **Para se resolver este problema, é necessário definir a hierarquia de atividades da aplicação no manifesto e dinamicamente especificar a pilha de retrocesso da atividade despoletada através de uma instância da classe `TaskStackBuilder`.** Este detalhe não é, contudo, discutido no âmbito das aulas.

2.4 Correr o Serviço em Primeiro Plano

Running the Service in Foreground

Devido ao facto dos **Serviços correrem normalmente em segundo plano**, estes **concretizam tipicamente bons candidatos à destruição quando o sistema precisa de recursos** (e.g., para mostrar uma página grande num *browser*). **Para se evitar** que determinado Serviço seja destruído tão facilmente em caso de necessidade, **pode-se definir um Serviço de primeiro plano**. Estes Serviços estão **normalmente associados a algo que o utilizador está ciente** (e.g., ouvir música) e que este não quer que seja facilmente terminado pelo sistema.

Para colocar um Serviço em primeiro plano **recorre-se ao método `startForeground(int, Notification)`, que aceita dois parâmetros:**

1. **Um inteiro**, que identifica univocamente a notificação (tem significado local); e
2. **Uma notificação**, que fica residente na barra de estado até que o serviço seja terminado. Esta notificação é colocada numa secção *Ongoing* da gaveta de notificações, o que significa que a notificação não pode ser limpa enquanto o serviço não terminar ou seja removido de primeiro plano.

O método `startForeground(.,.)` **deve ser colocado na implementação do Serviço respetivo, exatamente no ponto a partir do qual se quer que este execute em primeiro plano**. Antes de este poder ser invocado, a notificação e (opcionalmente) um intento pendente devem ser devidamente instanciados e configurados. O exemplo seguinte dá uma ideia de como usar estes recursos:

```
package pt.di.ubi.pmd.exservice;

import android.app.Service;
import android.app.PendingIntent;
import android.app.NotificationManager;
...

public class ServiceAlarms extends Service {
    static final int iID = 1;

    @Override
    public int onStartCommand
        (Intent intent, int flags, int startId) {
        ...
    }
}
```

```
NotificationCompat.Builder oBuilder =
    new NotificationCompat.Builder(this)
        .setSmallIcon(R.drawable.notification_icon)
        .setContentTitle("Super Alarm System")
        .setContentText("Super alarm system is running!")
        .setWhen(System.currentTimeMillis())
        .setOngoing(true);

Intent oIntent = new Intent(this,
    ExampleActivity.class);
PendingIntent oPI = PendingIntent.getActivity(
    this, 0, oIntent, 0);

oBuilder.setContentIntent(oPI);
startForeground(iID, oBuilder.build());
// O código seguinte corre em primeiro plano
...
stopForeground(true);
}
...
```

Para remover um Serviço do primeiro plano de execução, **deve recorrer-se a `stopForeground()`. Terminar o Serviço também tem o mesmo efeito**. Este método **aceita um booleano que define se a notificação introduzida por `startForeground()` deve ser retirada** (no caso de ser `true`) **ou não** (no caso de ser `false`).

Nota: o conteúdo exposto na aula e aqui contido não é (nem deve ser considerado) suficiente para total entendimento do conteúdo programático desta unidade curricular e deve ser complementado com algum empenho e investigação pessoal.