Dispositivos Móveis

Aula 13 - Services, Content Providers e Broadcast Receivers

Apresentação

Olá! Estamos chegando ao fim de nossa disciplina. Já vimos até agora vários aspectos bastante importantes do Android e, ainda assim, estamos com diversos assuntos tão importantes quanto para vermos. Nesta aula, continuaremos um estudo que começamos em nossas primeiras aulas. Veremos como funcionam os Services, os Content Providers e os Broadcast Receivers. Juntamente com as Activities, que já utilizamos de diversas maneiras durante o curso, esses componentes formam a base do Android. Vamos a eles!

Vídeo 01 - Apresentação

Objetivos

Ao final desta aula, você deverá ser capaz de:

- Implementar e entender o funcionamento de Services.
- Entender o funcionamento de Content Providers e Broadcast Receivers.

Services

Já vimos alguns conceitos sobre Services no começo do curso, porém, é sempre bom relembrar. Os Services são componentes que executam em background e por isso não possuem interface com o usuário. Essa é a grande diferença entre eles e as Activities. Outro aspecto importante em relação aos Services é que eles podem ser inicializados por uma aplicação e utilizados por outras. Estes também possuem a capacidade de permanecer executando mesmo depois do término da aplicação que o criou. Alguns exemplos da utilidade de Services são:

- Tocar músicas em background: na aula anterior, vimos como implementar uma Activity que fosse capaz de reproduzir uma mídia. Porém, sabemos que a Activity tem um ciclo de vida e que ela costuma ser paralisada quando o usuário entra em outra parte da aplicação. Para criarmos então uma reprodução de áudio que pudesse continuar mesmo após o encerramento da aplicação ou a troca de foco, poderíamos utilizar um Service para isso.

- Transmitir dados via rede: suponha que queiramos fazer um download de um arquivo. Esse processo é muito demorado para ser feito na thread principal e poderia facilmente bloqueá-la, causando um ANR. Para evitar isso, podemos lançar um Service em outra thread ou em outro processo, quando necessário, para que ele possa fazer o download dos dados que precisamos. Assim, podemos manter a thread principal livre para que a aplicação flua, mesmo enquanto fazemos a aquisição dos dados que foram requeridos. Uma vez completada a transmissão, o Service pode disponibilizar ao usuário o resultado obtido.

Através desses exemplos e da descrição, é fácil imaginar quando deveremos utilizar Services em nossa aplicação. O primeiro caso em que devemos criar um Service para nossas aplicações é quando há a necessidade de deixar alguma operação executando, mesmo após o usuário concluir a execução da aplicação. Um exemplo disso acontece na aplicação padrão de áudio do Android. Ao abrir a aplicação, você pode escolher qual música reproduzir e então fechar a aplicação. Mesmo após o encerramento da aplicação, a música fica tocando, indicada por uma notificação que pode levar o usuário de volta a aplicação para parar o serviço.

Outro caso que necessitaria da criação de um Service seria a execução de uma operação que precisasse realizar uma tarefa periodicamente, sem a intervenção do usuário. Imagine que você tem uma aplicação que depende de alguma atualização na web, como o Twitter, por exemplo. Nesse caso, seria necessário manter um serviço rodando em background que fosse capaz, de tempos em tempos, de buscar por atualizações na web. Esses são apenas alguns exemplos de casos em que um Service poderia ser utilizado. Existem, obviamente, diversos outros.

Criando um Service

Agora que já relembramos o que é um Service e para que utilizá-lo, vamos aprender a criar um. Essa criação é simples, como a de uma Activity, e não deve ser difícil de entender. Vamos ver os passos que devem ser seguidos para que tenhamos um Service funcional e que cumpra os objetivos dentro de nossa aplicação.

Assim como nas Activities, o primeiro passo na criação de um Service é estender a classe que vai implementar o serviço da classe Service, ou de alguma subclasse dela. Ao estender sua classe de Service, alguns callbacks devem ser implementados para tratar eventos do ciclo de vida do Service que você está implementando e também para prover mecanismos para que os outros componentes possam se ligar ao Service de maneira adequada.

Uma vez que a classe tenha sido criada, antes de qualquer outra coisa, deve-se adicionar ao Manifest a declaração do serviço que está sendo criado. Caso crie o Service diretamente pela IDE, isso é feito automaticamente. Caso contrário, para fazer isso, assim como nas Activities, basta declarar no Manifest uma propriedade indicando o nome do Service, da seguinte maneira:

```
1 <service android:name=".ExampleService" />
```

Uma vez declarado, o Service está pronto para receber Intents explícitos. Para que ele possa receber Intents implícitos, da mesma forma das Activities, a declaração precisa conter os campos de Intent Filter. É importante notar que, caso o Service deva ser usado apenas pela sua aplicação, ele não deve conter nenhum tipo de Intent Filter. Isso evita que aplicações desconhecidas iniciem o seu Service. Caso, ainda assim, você queira garantir que o Service só seja utilizado dentro de sua

aplicação, configure o atributo *android:exported* com o valor false. Com isso, o Android se encarregará de permitir apenas a classes de sua aplicação acessar o Service declarado.

Agora que já temos a classe criada, herdando de Service e devidamente declarada no Manifest, podemos passar a nos preocupar com os métodos que devem ser sobrescritos pelo novo Service. Veremos cada um dos métodos mais importantes e para que eles servem.

- ***onCreate():*** Esse método, da mesma forma das Activities, é o método chamado quando o Service é criado pela primeira vez. É nesse método que todas as inicializações necessárias devem ser feitas. Caso o serviço já esteja sendo executado quando for chamado, esse método não será chamado.

- ***onDestroy():*** Mais uma vez semelhante às Activities, esse método é o responsável por fazer todas as liberações e finalizações necessárias antes do encerramento de um Service. Ele é chamado antes do Service ser finalizado. É importante notar também que, similar ao que acontece com as Activities, existem casos extremos em que esse método não chega a ser executado antes da finalização do Service.

- ***onStartCommand():*** Esse método é um dos específicos de Services. É esse método que é responsável por iniciar um Service que tenha sido chamado pelo método *startService()*. Quando inicializado dessa maneira, um Service roda indefinidamente e é responsabilidade do programador chamar, em algum momento, o método *stopSelf()* dentro do próprio Service ou o método *stopService()* em algum outro componente. Se o programador não fizer isso, o Service rodará por tempo indeterminado, podendo comprometer a performance do aparelho e com isso a confiabilidade na sua aplicação. Caso prover um serviço desse tipo não seja de seu interesse, outro tipo é possível de ser implementado e isso é feito através do próximo método que veremos.

- ***onBind():*** Esse método é chamado pelo sistema quando um outro componente inicializa o Service que está sendo implementado pelo método *bindService()*. A implementação desse método deve fornecer ao usuário uma maneira de se comunicar com esse serviço. Isso normalmente é feito retornando a quem chamou uma interface

chamada IBinder. Esse método precisa sempre ser implementado e, caso não vá ser utilizado, deve retornar *null*.

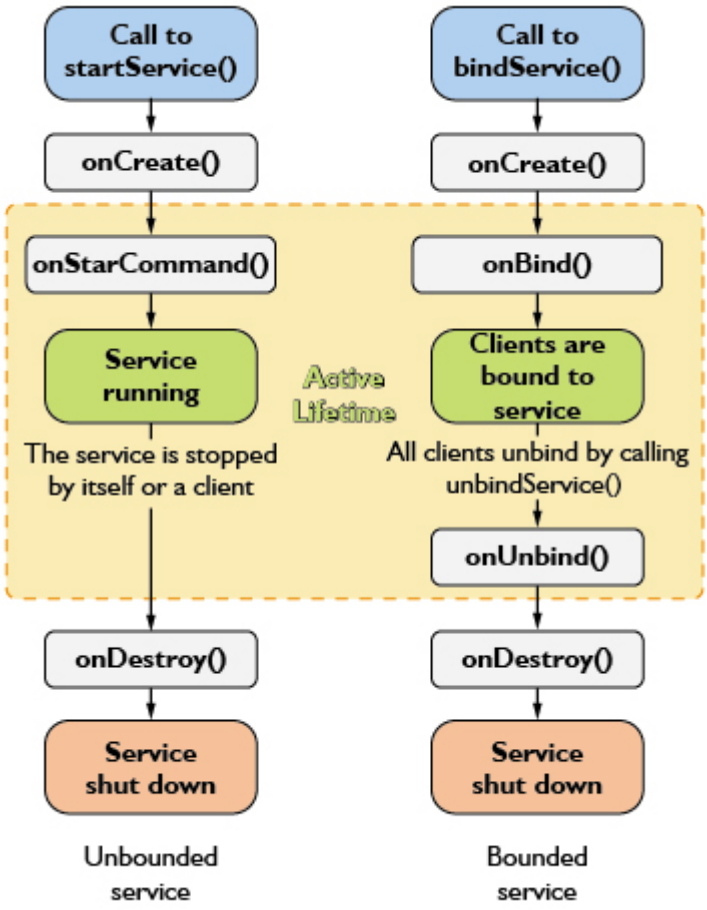
Esses quatro métodos são os métodos que devem ser sobrescritos quando se está criando um novo Service. É importante entender cada um deles e as peculiaridades envolvidas. Antes de implementar os dois últimos métodos, é de grande valia entender as diferenças entre um Service que pode ser inicializado, ou *started,* e um Service que pode ser atrelado, ou *bound.*

Formas de Serviços

Existem dois tipos de Services que proveem funcionalidades diferentes ao desenvolvedor. O do tipo Started é inicializado através do método *startService()* e, uma vez inicializado, pode rodar indefinidamente até que alguém chame *stopService()* ou o próprio serviço chame o método *stopSelf()*. Esse Service normalmente provê ao usuário serviços que normalmente são de longa duração como, por exemplo, executar alguma operação on-line. A Activity poderia inicializar um Service, passando a ele a operação a ser executada, e deixar que o Service fizesse o trabalho, provavelmente se encerrando após a conclusão do mesmo.

Já o Service do tipo Bound é inicializado através do método *bindService()*, chamado em algum outro componente. Após essa chamada, o componente se conecta ao Service em questão através de uma interface do tipo IBinder e passa a poder se comunicar com este Service. A grande diferença entre esse Service e o anterior é que agora é possível que haja uma comunicação numa espécie de cliente-servidor. Esse Service, diferentemente do outro, só é inicializado e começa a executar quando há alguém conectado a ele. A partir do momento em que o último cliente desconecta, o Service é parado e fica aguardando que outro componente se conecte a ele para que possa ser reinicializado. Vejamos, na **Figura 1**, como é o ciclo de vida de cada tipo de Service.

Figura 01 - Ciclo de vida de um Service (a esquerda) do tipo Started e de outro do tipo Bound (a direita).



Fonte: Android Developers.

Perceba que não existe um callback do tipo *onStop()* no ciclo de vida do Service do tipo Started. Isso indica que, quando o Service for parado, seja pelo *stopSelf()* ou pelo *stopService()*, ele não terá qualquer chance de reagir a isso. O sistema apenas checará se algum cliente ainda está conectado a ele e, caso ninguém esteja, o serviço será finalizado, passando para o seu método *onDestroy().*

Também é preciso notar que um Service pode, sem problema algum, implementar os dois tipos disponíveis. Imagine que você tem uma aplicação de música e que, em certo momento, o usuário põe uma música para tocar. Essa música pode rodar indefinidamente em um serviço do tipo Started. Agora suponha que o usuário quer interagir com a música, voltando a uma parte que ele precisa escutar novamente, por exemplo. Para fazer isso, ele pode se conectar ao serviço pela interface de bind dele e assim o serviço terá sido tanto iniciado quanto conectado.

Um último detalhe mais técnico sobre os Services é que, diferentemente das Activities, eles não precisam, em seus callbacks, incluir uma chamada à sua superclasse. Basta implementar o método normalmente e o Android saberá o que fazer com seu Service.

Agora que já sabemos as diferenças entre os serviços e os principais métodos envolvidos, vamos ver um exemplo básico de um Service do tipo Started.

Implementando um Service do tipo Started

Existem duas maneiras de se criar um Service desse tipo. A primeira delas pede que estendamos o nosso Service da classe Service, que é a classe base para todos os serviços. O problema da implementação utilizando esse método é que o Service será executado na mesma thread de quem o chamar, o que provavelmente vai gerar lentidão na aplicação e talvez até um ANR. Para evitar isso, uma nova thread deve ser lançada e todo o tratamento dela deve ser feito pelo desenvolvedor. Para facilitar esse processo, o Android fornece uma subclasse de Service chamada IntentService.

IntentService é uma subclasse que por default já roda em uma thread separada da principal e faz todo o tratamento necessário para você. Essa subclasse executa todas as requisições que chegam a ela em uma só thread, separada da thread principal e de maneira sequencial. É importante ter em mente esse comportamento antes de escolher a utilização dessa subclasse para sua aplicação. Se essa abordagem for adequada a sua aplicação, sem dúvidas é uma boa oportunidade para utilizar essa subclasse. Para que o IntentService funcione corretamente, é importante implementar o método *onHandleIntent()*, que será responsável por tratar as requisições que chegam. Além disso, é necessário criar um construtor que chama o super, passando o nome da thread como parâmetro. Também para essa subclasse, precisamos chamar o super em todos os outros métodos que venham a ser implementados além do *onHandleIntent().* Lembre-se de que a implementação dos outros métodos não é obrigatória nessa subclasse. A **Listagem 1** mostra um código simples que pode ser utilizado para criar um Service do tipo Started a partir do IntentSevice.

```
public class ServiceStarted extends IntentService {
   public ServiceStarted() {
      super("ServiceStarted");
   }
   @Override
   protected void onHandleIntent(Intent intent) {
      long endTime = System.currentTimeMillis() + 5*1000;
      while (System.currentTimeMillis() < endTime) {
         synchronized (this) {
            try {
              wait(endTime - System.currentTimeMillis());
            } catch (Exception e) {
            }
         }
      }
   }
}
```

Listagem 1 – Declaração de um Service Started que estende de IntentService.
Fonte: Android Developers.

Como foi dito anteriormente, apenas o construtor e o método *onHandleIntent()* precisam ser implementados para que o Service esteja pronto para ser utilizado. O construtor deve passar ao *super* o nome da thread que fará o trabalho.

Em seguida, temos a implementação do método *onHandleIntent().* Esse método é o responsável por executar as requisições que são feitas ao serviço. No caso do exemplo, a thread é apenas colocada para dormir por cinco segundos, para representar uma operação longa feita na thread. Perceba que o componente que chamar essa thread não ficará parado pelos cinco segundos. Perceba também que o Intent que foi utilizado para criar o Service é passado para o método para que possa ser utilizado adequadamente. Por fim, quando este método chegar ao seu fim, o IntentService se encarregará de encerrar a thread de maneira adequada. Quantas vantagens!

Vídeo 02 - Fazendo o Service Tocar a Música

Atividade 01

1. Crie uma Activity que possua um botão capaz de iniciar um Service como o mostrado no exemplo anterior. Quando o Service terminar, faça com que ele imprima no Log uma mensagem de sucesso.

Content Providers

Um Content Provider é responsável por apresentar informações armazenadas previamente a aplicações. Ele se assimila bastante a um banco de dados relacional por representar os dados como um conjunto de tabelas. Dentre os quatro componentes, o Content Provider é, sem dúvidas, o mais difícil de ser implementado e utilizado e, por esse motivo, cobriremos mais superficialmente esse tópico, já que não temos espaço o bastante para entrar tão a fundo nesse assunto.

Vários Content Providers já existem dentro da plataforma, contendo informações de diversos tipos. Informação de contato, áudio, vídeos, entre outros estão entre os Content Providers já conhecidos do sistema. Um exemplo de um Content Provider é o que é responsável por guardar novas palavras que o usuário adiciona aos dicionários do sistema. Dentro desse Content Provider é possível ao sistema saber quais palavras novas o usuário adiciona ao dicionário, a frequência com que elas são utilizadas e mais outras informações relevantes.

Também é importante ressaltar que é possível ao desenvolvedor criar seus próprios Content Providers, nos quais se armazenam dados de sua aplicação que podem ser compartilhados com outras aplicações e também reusados em outras execuções da aplicação.

Outro ponto importante é que todo Content Provider deve implementar uma interface comum, e é através dessa interface que é possível para os utilizadores dos Content Providers fazer inserções, consultas e alterações de dados nos provedores. Isso tudo acontece através da interface ContentResolver.

Para tornar possível o acesso a um Content Provider, é necessário que o sistema forneça um caminho até ele. Isso acontece através de uma URI especial. Todo Content Provider tem uma URI única capaz de identificá-lo. Todas essas URI começam com *content://*. Esse formato indica ao sistema que a URI leva até um Content Provider e também é ela que permite as interações com o ContentResolver. Vejamos agora, na **Listagem 2**, um exemplo de consulta feita a um Content Provider já existente. É importante lembrar que não entraremos em muitos detalhes de como criar ou mesmo editar Content Providers e utilizaremos esse exemplo para dar a ideia de como utilizar um provedor já existente e fundamentar estudos avançados.

```
1 // Consulta ao dicionário do usuário
2 mCursor = getContentResolver().query(
3    // O URI contendo o caminho ao ContentProvider
4    UserDictionary.Words.CONTENT_URI,
5    mProjection,            // As colunas a serem retornadas
6    mSelectionClause        // Filtro dos valores (similar ao WHERE do SQL)
7    mSelectionArgs,         // Valores dos parâmetros do filtro
8    mSortOrder); // Descrição da ordem dos valores (similar ao orderBy do SQL)
```

Listagem 2 – Exemplo de consulta a um provider.

A **Listagem 2** nos mostra a semelhança que as consultas a Content Providers têm em relação a consultas SQL. De fato, os Content Providers se assimilam bastante a um banco de dados. O importante é que, uma vez finalizada a consulta indicada na **Listagem 2**, a variável mCursor terá um Cursor preenchido com os dados obtidos na busca. Tendo o Cursor com os dados que queremos, podemos utilizar métodos como *mCursor.moveToNext()* para iterar pelos resultados e então utilizar *mCursor.getString(column)* para obter as informações que desejamos. Perceba também que uma instância do ContentResolver é buscada e utilizada para fazer a consulta que preenche o Cursor.

Como vemos, um Content Provider é uma maneira eficiente de compartilhar dados entre aplicações. Uma vez que uma aplicação preencha o Content Provider, consultas simples podem ser feitas por qualquer outra aplicação que tenha permissão para acessar o provedor. Esse é o principal propósito de se criar um Content Provider, como vimos em outras oportunidades. A seção **Leitura complementar** indica um excelente documento sobre o assunto para aprofundar os estudos, caso esse componente seja necessário em sua aplicação.

Vídeo 03 - Content Provider de Contatos

Atividade 02

1. Implemente um trecho de código que faz uma busca e recupera os nomes de todos os contatos da agenda telefônica do usuário.

Broadcast Receivers

Os Broadcast Receivers são responsáveis por receber eventos que são enviados pelo sistema ou até mesmo por outras aplicações. Estes eventos têm como propósito informar diversos tipos de ocorrências para outros componentes interessados nestas. Pouca bateria, desligamento da tela e chamada recebida são exemplos de eventos enviados pelo sistema que podem ser recebidos por Broadcast Receivers que estejam interessados. Esses eventos são enviados em forma de Intent, através do método *sendBroadcast()*.

Para criarmos um Broadcast Receiver, podemos chamar o método *Context.registerReceiver()*, para fazer isso dinamicamente, ou então declarar no Manifest o Broadcast Receiver através da tag <receiver>. Uma vez registrado, o Broadcast Receiver passa a receber mensagens enquanto aquele componente que o registrou estiver ativo. Por causa desse comportamento, é importante que o Broadcast Receiver esteja registrado apenas durante a vida do componente que o utiliza, ou seja, se tivermos uma Activity inicializando o Receiver, é importante registrar-se durante o método *onResume()* e se desligar durante o método *onPause()*. Isso garante que o tempo registrado do Broadcast Receiver é igual ao tempo de foco que a atividade tem.

Em relação aos tipos de Broadcast Receivers, existem duas principais classes. A primeira delas é a responsável pelos Broadcasts normais. Esses Receivers executam de maneira assíncrona e rodam em ordem indefinida, normalmente ao mesmo tempo.

Já o segundo tipo é o Ordered Broadcast. Os Receivers para eventos desse tipo recebem o evento em ordem previamente definida, baseada na propriedade <priority>, declarada no Intent Filter do Receiver, no Manifest. Neste caso, cada Receiver recebe o Intent em uma ordem predeterminada. Isso cria a possibilidade de que um Receiver que tenha a ordem mais alta anule o evento, consumindo-o e evitando que os próximos na ordem cheguem a vê-lo. Note que caso dois Receivers tenham a mesma <priority>, a ordem de entrega entre eles será definida de maneira arbitrária pelo próprio Android.

Mesmo no caso de um Broadcast normal ser usado, às vezes o sistema executa os Broadcasts um por um para evitar a sobrecarga do sistema. Isso é especialmente verdade quando mais de um Receiver cria processos. Para evitar que o sistema se sobrecarregue com a criação de diversos processos simultâneos e muita informação sendo processada, o Android cuida para que cada Receiver seja executado de uma vez. Mesmo assim, por não serem Ordered Broadcasts, não é possível para um Receiver consumir a requisição, ou mesmo retornar um valor para algum outro Receiver que venha depois.

Uma outra curiosidade bem importante é que, apesar de tanto Activities quando Broadcast Receivers utilizarem Intents como seus mecanismos de comunicação, os Intents utilizados são completamente diferentes de um para outro. Podemos afirmar com toda certeza que um Intent endereçado a Activities nunca iniciará um Receiver e vice-versa. Ou seja, apesar de utilizarem o mesmo mecanismo, os processos pelos quais eles passam é completamente diferente.

Quando se trata de segurança, quatro aspectos devem ser levados em conta. Eles são importantes porque o Broadcast Receiver é, por padrão, uma interface que funciona entre aplicações. Para evitar que problemas aconteçam por causa deles, é importante observar o seguinte:

- O espaço para os Intents é global, ou seja, se uma ação de um Intent tiver o mesmo nome de outra ação já existente, o Intent poderá ser

entregue para qualquer um, gerando um conflito indesejado para a sua aplicação.

- Quando você registra um Broadcast programaticamente, através do método *registerReceiver()*, qualquer aplicação pode mandar Intents que atingem o seu Broadcast Receiver. Tenha isso em mente quando for desenvolver um elemento desse tipo.

- Os Broadcast Receivers, assim como os Services, tem a possibilidade da inclusão do atributo *exported* no Manifest da aplicação para indicar que o componente não deve ser exportado, ou seja, não deve ser visível para outras aplicações.

- Quando utilizar o método *sendBroadcast()*, tenha em mente que qualquer aplicação pode receber o Broadcast enviado. Para evitar isso, é possível configurar uma permissão não nula tanto na requisição quando no recebedor. Isso pode ser feito nos métodos de envio e recepção de Broadcast ou ainda dentro da tag responsável, no Android Manifest, através do atributo <uses-permission>.

Um último ponto importante quando estamos lidando com o Broadcast Receiver é saber o ciclo de vida do componente e o que podemos ou não fazer dentro desse ciclo de vida. Quando estamos implementando um Broadcast Receiver, o método mais importante é o *onReceive()*, que é responsável por receber e processar um Intent que aquele Receiver esteja registrado para receber. E é justamente esse método que determina o tempo de vida de um componente desse tipo. Quando um Intent do tipo correto chega ao componente, ele é iniciado e o método *onReceive()* é chamado. Dentro desse método, todo o processamento que deve ser feito pelo Receiver é executado e então ele é finalizado, encerrando seu ciclo de vida ali mesmo.

Vendo esse curto ciclo de vida, podemos entender que nenhuma atividade demasiadamente demorada pode depender desse componente. Imagine, por exemplo, que lançamos uma thread para executar um processamento e retornar para o Receiver quando concluir. É muito provável que, ao buscar o componente para retornar, a thread já encontre o Receiver encerrado, o que geraria um erro de execução. Por esses motivos, é totalmente desencorajado o uso desse tipo de artifício dentro de um Broadcast Receiver.

Além disso, é proibido pelo Android criar Dialogs de dentro de um Broadcast Receiver. A única forma de interface que o componente tem com o usuário é a criação de notificações na área de notificação do sistema. Assim como em um Service, o Broadcast Receiver não tem a capacidade de criar uma tela para interagir com o usuário por si próprio. Também note que, graças ao seu ciclo de vida curto, um Broadcast Receiver não pode se conectar a um serviço de tipo Bound, podendo apenas iniciar um Service com o comando *startService().* Para concluir nossos estudos sobre Broadcast Receivers, vamos olhar a **Listagem 3**, que mostra a implementação de um Broadcast Receiver simples, o qual inicializa ou para um Service baseado na carga da bateria do aparelho. Caso o aparelho esteja com bateria fraca, o Service deve ser parado. Caso volte a estar com a bateria boa, o Service volta a funcionar.

```
1  public class myBroadcastReceiver extends BroadcastReceiver {
2      @Override
3      public void onReceive(Context context, Intent intent) {
4          String action = intent.getAction();
5          if (action.equals(Intent.ACTION_BATTERY_LOW)) {
6              Intent i = new Intent(context, ServiceBattery.class);
7              context.stopService(i);
8          } else if (action.equals(Intent.ACTION_BATTERY_OKAY)) {
9              Intent i = new Intent(context, ServiceBattery.class);
10             context.startService(i);
11         }
12     }
13 }
```

Listagem 3 – Broadcast Receiver que lida com eventos de bateria

A **Listagem 3** mostra bem os aspectos básicos necessários para a criação de um Broadcast Receiver. Primeiramente, estendemos da classe BroadcastReceiver. Em seguida, sobrescrevemos o método *onReceive()*, e então tratamos o Intent recebido de maneira adequada. Perceba que apenas comandos simples são executados dentro do método. Também é importante lembrar que para isso funcionar corretamente, o Manifest deve declarar os Intents que o Broadcast Receiver será capaz de receber.

Com esse exemplo, concluímos nossos estudos sobre Broadcast Receivers e também nossa aula. Estamos chegando perto do fim e aposto que todos já temos diversos conceitos novos em nossa mente para trabalhar. Nesse ponto, já temos capacidade de produzir aplicações bem úteis, dependendo das ideias de cada um.

Para completar nossos estudos, temos apenas mais um tema de programação a ser abordado, que é conexão web. Então, nos vemos na próxima aula, sobre internet e Android. Até lá!

Vídeo 04 - Broadcast Receiver de SMS

Atividade 03

1. Realizando pesquisas na internet, complemente o assunto que foi abordado, desenvolvendo um Broacast Receiver capaz de ser notificado sempre que chegar uma mensagem SMS.

Leitura Complementar

- DEVELOPERS. Services. Disponível em: http://developer.android.com/services.html. Acesso em: 23 ago. 2012.

- DEVELOPERS. Content Providers. Disponível em: http://developer.android.com/content-providers.html. Acesso em: 23 ago. 2012.

- DEVELOPERS. Broadcast Receivers. Disponível em: http://developer.android.com/BroadcastReceiver.html. Acesso em: 23 ago. 2012.

Resumo

Nesta aula, vimos os três componentes que faltavam dentre os existentes no Android. Iniciamos nossa aula vendo o mais importante deles, depois das Activities, é claro, que são os Services. Vimos como os Services funcionam, os principais cuidados que precisamos tomar ao implementar componentes desse tipo e aprendemos a lidar com o ciclo de vida deles. Ainda nessa primeira seção vimos um exemplo de código explicando como implementar um dos dois tipos de Service que estão disponíveis na plataforma Android e que também foram discutidos na primeira parte desta aula. Na sequência, vimos o componente mais complicado de se utilizar, que é o Content Provider. Evitamos entrar em detalhes sobre esse componente, pois suas peculiaridades são muitas e sem dúvidas levaria mais de uma aula inteira para explicar seu funcionamento em detalhes. Cobrimos alguns pontos principais desse componente e indicamos a referência ao site oficial do Android para maiores informações. Por fim, vimos o componente que faltava, o Broadcast Receiver. Estudamos em detalhes os pontos importantes a serem levados em consideração quando implementamos um componente desse e também discutimos os aspectos de segurança que devem ser levados em conta. Para finalizar o tópico e a aula, vimos um código mostrando a implementação de um Broadcast Receiver simples.

Autoavaliação

1. O que são Services? Cite duas situações em que poderíamos criar Services em uma aplicação.
2. Quais os dois tipos de Service e quais as principais diferenças entre eles?
3. É possível acessar os Content Providers do Android? Que tipos de informações podemos encontrar nos Content Providers padrões?
4. Implemente um Broadcast Receiver que seja capaz de receber Intents que indicam se o aparelho está conectado à energia ou não e então notifique no Log mudanças nesse estado.

Referências

ANDROID DEVELOPERS. 2012. Disponível em: <http://developer.android.com>. Acesso em: 25 abr. 2012.

DIMARZIO, J. **Android:** A Programmer's Guide. New York:McGraw-Hill, 2008. Disponível em: <http://books.google.com.br/books?id=hoFI5pxjGesC>. Acesso em: 23 ago. 2012.

LECHETA, Ricardo R. **Google android:** aprenda a criar aplicaçoes. 2. ed. São Paulo: Novatec, 2010.

_____. **Google android para tablets**. São Paulo: Novatec, 2012.

MEIER, R. **Professional Android 2 Application Development**. Hoboken, NJ: John Wiley, 2010. Disponível em: <http://books.google.com.br/books?id=ZthJlG4o-2wC>. Acesso em: 23 ago. 2012.