Tópicos Avançados de Programação

Aula 5

Prof. Marcelo Rodrigues do Nascimento

Conversa Inicial

Se você tem ou conhece alguém que possua um aparelho celular rodando sistema Android, já deve ter notado que, todas as vezes que um novo aplicativo é instalado no dispositivo, somos apresentados a uma lista de permissões requeridas para sua execução. Sabe por que isso ocorre?

Dispositivos Android possuem rígidas regras de segurança, que definem o comportamento padrão a todos os aplicativos nele instalados. Quando qualquer aplicativo necessita de acesso a um componente diferenciado, não pode simplesmente fazer uso deste recurso, como aconteceria normalmente em softwares desenvolvidos para Desktop. No caso de aplicativos Android, devemos solicitar permissões ao sistema operacional, que informará nossas intenções ao usuário final no ato de instalação de nossos aplicativos.

Outra característica importante da plataforma Android é a de oferecer múltiplos pontos de entrada a seus aplicativos, permitindo assim a reutilização de componentes por terceiros, através de sua correta declaração no arquivo de Manifesto. Seja utilizando o componente de Activity ou de serviços, este é com certeza um dos recursos mais interessantes do ponto de vista do desenvolvedor, uma vez que podemos aproveitar componentes de terceiros, como o ato de se tirar uma fotografia ou gravar um vídeo. Isso torna o desenvolvimento de aplicativos Android mais ágil e seguro.

Contextualizando

Quando desenvolvemos aplicativos para dispositivos Android, somos apresentados a uma série de componentes que nos permitem interagir com os mais variados tipos de recursos disponíveis em dispositivos Android, como câmeras, sistema de localização, armazenamento de dados e até mesmo funções de comunicação, como envio de mensagens ou ligações.

No entanto, diferentemente de outros sistemas operacionais, aplicativos Android trabalham no princípio do privilégio mínimo, ou seja: **toda ação que requeira acesso a componentes fora dos privilégios padrão de aplicativos deve ter sua permissão requisitada no ato de instalação do aplicativo**.

Por isso, o desenvolvedor Android deve declarar corretamente as permissões que seu sistema possa precisar.

Também já discutimos anteriormente a respeito do reaproveitamento de componentes no sistema operacional, uma vez que nesta plataforma possuímos vários pontos de entrada para nossos aplicativos. Utilizaremos este conceito para acionar uma Activity por uma ação implícita, ou seja: aproveitaremos um componente desenvolvido por terceiros para adquirir informações a respeito do dispositivo. Aprenderemos também a disponibilizar nossos próprios componentes para ações padrão.

Também trabalharemos o conceito de serviços e sua disponibilização ao sistema operacional para tarefas que não necessitem de uma interface com o usuário e que possam ter um longo período de duração, ou ainda, alta necessidade de processamento.

Tema 1 – Trabalhando com Permissões

Cada aplicativo Android é executado partindo-se do princípio do privilégio mínimo, ou seja, cada aplicativo tem acesso somente aos componentes necessários a seu trabalho e nada mais. Se um aplicativo necessita de algum recurso ou informação extra, é necessário que seja declarada a necessidade desta permissão no Manifesto do Aplicativo.

Todo aplicativo Android possui um arquivo chamado AndroidManifest.xml, responsável por passar ao sistema Android informações sobre o aplicativo. Entre suas funções, o manifesto:

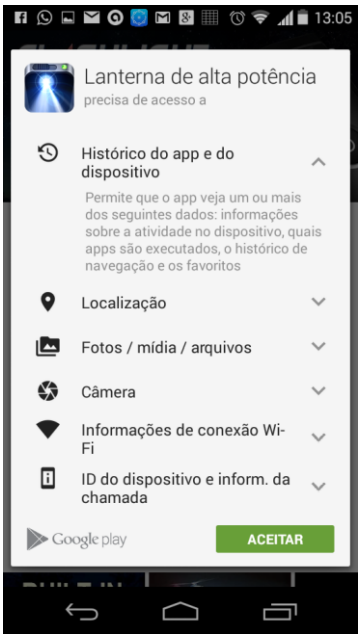
- Nomeia o pacote Java para o aplicativo;
- Descreve os componentes do aplicativo (Activities, Services, Broadcast Receivers e Content Providers);
- Lista as bibliotecas vinculadas ao aplicativo;
- Determina o nível mínimo da API do Android que o aplicativo exige;
- Declara permissões que outros aplicativos devem ter para interagir com os componentes do aplicativo descrito no manifesto;
- Declarar as permissões que o aplicativo deve ter para acessar partes protegidas da API e interagir com outros aplicativos.

À medida que desenvolvemos aplicativos é comum estendermos suas funcionalidades, de maneira que possamos utilizar recursos do dispositivo no qual nosso aplicativo está instalado, como a câmera fotográfica, o cartão de memória ou até mesmo o acesso à internet.

Por uma questão de segurança (ou seja, o princípio do privilégio mínimo), todo componente que não se encontra dentro do básico

para execução de um aplicativo requer uma permissão especial devidamente declarada no arquivo de manifesto.

Quando seu usuário tenta instalar seu aplicativo, as informações de privilégios especiais requisitados são mostradas a ele, informando-o quais componentes serão utilizados.



Na figura, vemos um aplicativo chamado “Lanterna de alta potência” que requer permissões de acesso ao histórico de aplicativos e dispositivo, localização via GPS / WIFI, acesso aos arquivos, fotos e mídias, acesso à câmera fotográfica, informações a respeito da conexão de WIFI e identificação do dispositivos e informações referentes à chamada telefônica (se o dispositivo está agora em uma ligação telefônica, por exemplo).

Em um ambiente Desktop, este tipo de utilização de componentes poderia passar desapercebido, mas graças ao princípio do privilégio mínimo, este tipo de informação é tornada explícita ao usuário no momento de instalação.

Quando estamos escrevendo o Manifesto, devemos tomar o cuidado de definir claramente qual tipo de permissão necessitaremos e seu real uso. Solicitar permissões que não

condizem ao propósito do aplicativo podem levar a dúvidas por parte do usuário e a não instalação de nosso aplicativo.

Um ponto importante a se ressaltar, no entanto, é que nossos aplicativos necessitam de permissões somente para ações que executem diretamente. Em casos onde nosso aplicativo requisita a outro aplicativo que execute uma ação ou forneça uma informação, estas permissões não são solicitadas.

Por exemplo, caso seu aplicativo deseje acesso à lista de contatos do dispositivo dentro de sua própria Activity, deverá solicitar a permissão READ_CONTACTS. No entanto, caso sua Activity utilize-se de uma intente para solicitar informação (retorno) de uma Activity do aplicativo de contato do dispositivo, o seu aplicativo não mais necessitará da permissão, já que é considerada necessária somente ao aplicativo de contatos do dispositivo.

Vamos agora criar um aplicativo que fará a leitura de todos os contatos disponíveis no dispositivo através de sua própria Activity, ou seja, necessitando permissões especiais para tanto.

Em seu Android Studio, crie um novo projeto chamado ContatosActivities, utilizando como template de sua main activity uma Activity Vazia. Para este exemplo, utilizaremos uma interface para nossa main_ activity.xml composta de apenas uma TextView. Não se esqueça de identificar seu componente TextView com um id único e que permita desambiguação.

Para sua comodidade, confira o xml completo de nosso arquivo activity_main.xml a seguir:

```
<?xml version="1.0" encoding="utf-8"?>

<RelativeLayout
xmlns:android="http://schemas.android.com/apk/res/android
"
    xmlns:tools="http://schemas.android.com/tools"
    android:layout_width="match_parent"
    android:layout_height="match_parent"

android:paddingBottom="@dimen/activity_vertical_margin"

android:paddingLeft="@dimen/activity_horizontal_margin"

android:paddingRight="@dimen/activity_horizontal_margin"
    android:paddingTop="@dimen/activity_vertical_margin"

tools:context="br.com.grupouninter.aula.contatosactivities.
MainActivity">
    <TextView
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:id="@+id/txtContatos" />
</RelativeLayout>
```

Em seu arquivo MainActivity.java, devemos agora vincular nosso componente de layout TextView a um objeto, portanto declare:

```
TextView txtContato;
```

Uma vez declarado, em nosso método onCreate() iremos inicializar o objeto, vinculando-o ao layout xml:

```
txtContato = (TextView) findViewById(R.id.txtContatos);
```

Agora criaremos o método que fará acesso aos dados de nossos contatos. Chamaremos este método de buscaContato().

```
public void buscaContato(){

}
```

Nos utilizaremos dos objetos que nos dão acesso às informações de contatos, ou seja, o objeto ContactsContract; este objeto define um banco de dados de informações de contatos, armazenadas da seguinte maneira:

- Uma linha na tabela ContactContract.Data pode armazenar qualquer tipo de dados pessoais, como telefone ou endereços de e-mails;
- Uma linha na tabela ContactsContratcs.RawContacts representa um conjunto de dados que descreve a pessoa e a associa a uma única conta, como a conta de e-mail do usuário;
- Uma linha na tabela ContactContract.Contacts representa a agregação de um ou mais RawContacts, descrevendo a mesma pessoa.

Entendido isto, utilizaremos então o objeto ContactsContract.Contacts para armazenar informações a respeito do ID, Nome e se o contato possui telefone.

Utilizaremos o objeto ContactcsContract.CommonDataKinds para buscarmos informações a respeito do NUMERO de telefone e, finalmente ContactsContract.CommonDataKinds para obtermos informações a respeito do e-mail de nosso Contato.

Estas informações são transportadas em objetos Uri, ou seja, objetos *Uniform Resource Identified* (Identificador Uniforme de Recurso). Este tipo de dado é basicamente uma cadeia de caracteres utilizada para identificar ou denominar um recurso.

Primeiramente, declaramos uma variável que conterá todos os nossos contatos:

```
String contatos = ""
```

Criamos então o objeto Uri, que armazenará informações referentes ao objeto ContactsContract.Contacs:

```
Uri CONTENT_URI = ContactsContract.Contacts.CONTENT_URI;

String _ID = ContactsContract.Contacts._ID;

String DISPLAY_NAME = ContactsContract.Contacts.DISPLAY_NAME;

String HAS_PHONE_NUMBER = ContactsContract.Contacts.HAS_PHONE_NUMBER;
```

Definimos o objeto URI, que armazenará informações referentes aos números de telefone.

```
Uri PhoneCONTENT_URI = ContactsContract.CommonDataKinds.Phone.CONTENT_URI;

String Phone_CONTACT_ID = ContactsContract.CommonDataKinds.Phone.CONTACT_ID;
```

```
String NUMBER = ContactsContract.CommonDataKinds.Phone.NUMBER;
```

E finalmente o objeto URI, que conterá informações referentes aos e-mails:

```
Uri EmailCONTENT_URI = ContactsContract.CommonDataKinds.Email.CONTENT_URI;

String EmailCONTACT_ID = ContactsContract.CommonDataKinds.Email.CONTACT_ID;

String DATA = ContactsContract.CommonDataKinds.Email.DATA;
```

Feito isso, buscamos o ContentResolver, que criará uma interface de acesso entre a URI, e um Cursor, que nos permitirá navegar entre os registros.

```
ContentResolver contentResolver = getContentResolver();
```

Agora que temos nosso contentResolver definido, vamos à primeira busca: Desejamos receber todos os contatos armazenados no dispositivo:

```
Cursor cursor = contentResolver.query(CONTENT_URI, null,null, null, null);
```

Caso o número de contatos seja maior que 0 (possuímos contatos cadastrados) e enquanto nosso cursor tiver registros a visitar, executaremos um laço de repetição:

```
if (cursor.getCount() > 0){

while (cursor.moveToNext()){

        }

}
```

Agora damos início às buscas mais específicas dentro de nossas URI. Antes de mais nada, vamos armazenar o id e o nome da pessoa dentro do registro atual:

```
String id = cursor.getString(cursor.getColumnIndex(_ID));
String nome = cursor.getString(cursor.getColumnIndex(
DISPLAY_NAME));
```

Uma vez armazenado, devemos descobrir se a pessoa no registro atual possui um número de telefone cadastrado. Para tanto, converteremos o valor em String que nos foi retornado da coluna HAS_PHONE_NUMBER. Este valor nos informará quantos números telefônicos temos registrados para esta pessoa em específico:

```
int possuitelefone =
Integer.parseInt(cursor.getString(cursor.getColumnIndex(HAS_P
HONE_NUMBER)));
```

Caso esta pessoa possua algum número cadastrado...

```
if (possuitelefone > 0){

armazenaremos seu nome em nossa variável de contato

contatos += "\nNome: " + nome;
```

E criaremos um novo cursor para iterar em nossa URI de PhoneCONTENT_URI, buscando somente registros referentes ao id atual no looping:

```
Cursor cursorTelefone =
contentResolver.query(PhoneCONTENT_URI, null,
Phone_CONTACT_ID + " = ?", new String[]{id}, null);
```

Iteramos no cursor, enquanto este possuir registros a respeito do ID atual, armazenando sua informação também em nossa variável de contato:

```
while (cursorTelefone.moveToNext()){
   contatos += "\nTelefone: " + cursorTelefone.getString(cursorTelefone.getColumnIndex(NUMBER));
}
```

E, finalmente, fechamos o cursor de telefone.

```
cursorTelefone.close();
```

Abrimos agora um novo cursor, buscando informações de EmailCONTENT_URI a respeito de e-mails armazenados para o id de contato atual, iteramos nele e armazenamos eventuais registros localizados, fechando o cursor ao final.

```
Cursor emailCursor = contentResolver.query(EmailCONTENT_URI, null, EmailCONTACT_ID + " = ?", new String[] {id}, null);
while (emailCursor.moveToNext()){

contatos += "\nEmail: " + emailCursor.getString(emailCursor.getColumnIndex(DATA));
}
emailCursor.close();
```

Finalmente, transferimos o conteúdo de nossa variável contatos ao objeto txtContato, em sua propriedade text:

```
txtContato.setText(contatos);
```

Não esqueça de chamar o método buscaContato() na finalização de seu método onCreate():

```
buscaContato();
```

Para sua comodidade, acesse o código completo de nosso arquivo MainActivity.java. no material *on-line*.

Execute agora seu aplicativo. Como não solicitamos permissão de acesso a leitura de contatos, nosso aplicativo sofre uma exceção, encerra e em nossa janela do Android Monitor recebemos a seguinte informação:

```
java.lang.RuntimeException: Unable to start activity

ComponentInfo{br.com.grupouninter.aula.contatosactivities/br.com.grupouninter.aula.contatosactivities.MainActivity}:
java.lang.SecurityException: Permission Denial: opening provider com.android.providers.contacts.ContactsProvider2 from ProcessRecord {71b8ccb 29001 : br.com.grupouninter.aula.contatosactivities / u0a162} (pid=29001, uid=10162) requires android.permission.READ_CONTACTS or android.permission.WRITE_CONTACTS
```

Ou seja, nosso aplicativo necessita de uma permissão especial que não foi definida corretamente por nós em seu manifesto. Abra o arquivo AndroidManifest.xml e requisite a permissão de leitura de contatos, digitando a chave:

```
<uses-permission android:name="android.permission.READ_CONTACTS"></uses-permission>
```

Execute novamente seu aplicativo. Como você pode perceber, temos agora acesso à toda lista de contatos disponível no

dispositivo. Iteramos por eles e apresentamos a nosso usuário estas informações, em um formato desenvolvido por nós.

Tema 2 – Interagindo com outros aplicativos

Um aplicativo Android é composto geralmente de várias Activities, onde cada Activity representa uma interface com o usuário. Conforme já conversamos anteriormente, para que uma Activity dispare outra, uma Intent deve ser criada, informando qual a classe de Activity a ser instanciada. Essa Intent é então passada ao sistema operacional, que se responsabiliza por iniciar a Activity desejada.

Eventualmente, no entanto, desejamos executar uma ação para a qual o sistema Android já possua um aplicativo que execute isso a contento. Por exemplo, ao invés de desenvolvermos uma interface para se tirar fotos completamente do início, podemos nos aproveitar de aplicativos que executem a mesma tarefa. Isso é feito através de **intenções implícitas**.

As intenções implícitas não declaram o nome da classe do componente que desejam iniciar, mas sim a ação que desejam executar. Estas ações especificam o que deve ser feito, como editar, visualizar ou enviar algo.

Quando criamos uma intenção implícita, geralmente desejamos passar dados relacionados a esta ação, como por exemplo um número de telefone a ser discado pela Activity responsável pela discagem. Estes dados podem ser do tipo Uri, ou outros tipos definidos pela Activity registrada para determinada ação, além é claro, de não necessitar de dado algum.

Por exemplo, podemos criar uma Intent para iniciar uma chamada telefônica passando dados Uri que especifiquem o número de telefone:

```
Uri número = Uri.parse(“tel:5555555”)
```

Devemos então criar uma Intent implícita, passando nossa URI como argumento. Isto é possível pois existe um construtor para Intents que utiliza Uri como formato de dado a ser passado como argumento para a nova Activity.

```
Intent intent = new Intent(Intent.ACTION_DIAL, número);
```

Quando executarmos a chamada à Intent através do método startActivity, a Activity de telefone iniciará uma chamada para o número especificado. Entre as Intenções Implícitas comuns em dispositivos Android, podemos destacar:

ACTION_SET_ALARM – Cria um alarme

ACTION_SHOW_ALARMS – Mostra uma listagem de todos os alarmes definidos

ACTION_IMAGE_CAPTURE – Abre o aplicativo de câmera e recebe a imagem capturada

ACTION_VIDEO_CAPTURE – Abre o aplicativo de câmera e recebe o vídeo capturado

Uma listagem e exemplos de utilização de Intenções Implícitas comuns está disponível a seguir:

https://developer.android.com/guide/components/intents-common.html

Como vimos anteriormente, a utilização de determinados componentes no sistema Android (como câmeras, lista de contatos, etc) nos obriga a solicitar permissão específica em nosso arquivo de manifesto. No entanto, quando executamos esta ação utilizando uma Activity de outro aplicativo que já possui este tipo de permissão, não somos obrigados a solicitar

diretamente a permissão no arquivo de manifesto de nosso aplicativo, já que a permissão foi atribuída pelo usuário no ato de instalação do aplicativo do qual utilizaremos a Activity.

No entanto, ficamos vinculados ao tipo de ação e resposta que a Activity chamada fornece. Ou seja, caso você necessite que seu aplicativo abra a câmera de dispositivo e grave imagens sem a intervenção do usuário, deve criar este evento em sua própria Activity e solicitar a permissão do usuário durante a instalação de seu aplicativo. Caso opte por utilizar uma intenção implícita, ficará vinculado ao tipo de ação e resposta definido pela Activity registrada para aquela intenção. Por exemplo, a ação de tirar uma foto no aplicativo de câmera padrão exige que o usuário clique na tela.

Vamos colocar este conceito à prova agora.

Em nosso novo aplicativo, buscaremos os contatos disponíveis no dispositivo. Entretanto, desta vez, delegaremos este trabalho à Activity responsável por gerenciamento de contatos no dispositivo. Quando esta Activity for chamada e nosso usuário clicar em um dos contatos disponíveis, ao invés de seguir o fluxo normal do aplicativo de Contatos (ou seja, listar maiores informações a respeito do contato diretamente no próprio aplicativo), o valor selecionado será retornado a nossa Activity.

Como estaremos utilizando uma Activity de outro aplicativo para executar esta tarefa, podemos nos aproveitar das permissões que este aplicativo solicitou ao sistema operacional. No entanto, lembre-se:

Nosso acesso aos componentes de contatos agora fica limitado à informação que a Activity do aplicativo de Contatos nos retornar, diferente de nosso aplicativo anterior, que listava todos os contatos do dispositivo e nos permitia interagir com eles.

Crie um novo aplicativo no Android Studio. Chamaremos este aplicativo de ContatosporResultado.

No layout de nossa MainActivity, apresentaremos a nosso usuário dois componentes. Um componente de TextView que se encarregará de mostrar o número de telefone selecionado pelo usuário e um Button que servirá para a criação de nossa intenção implícita.

No componente TextView, defina as seguintes propriedades:

```
id:txtNumero

text: Número Selecionado
```

No componente Button, defina as seguintes propriedades:

```
id: btnContato

text: Buscar Contato
```

Tente criar esta interface por conta própria. Para sua referência, segue o código completo de nosso arquivo activity_main.xml, no material *on-line.*

Uma vez definido nosso layout, devemos agora criar os objetos que representam estes componentes.

Em sua classe Main, defina:

```
Button btnContato;

TextView txtNumero;
```

E então instancie os objetos corretamente:

```
btnContato = (Button) findViewById(R.id.btnContato);

txtNumero = (TextView) findViewById(R.id.txtNumero);
```

O próximo passo é redefinir o comportamento do listener onClickListener, para que este inicie a intenção implícita. Porém, antes de iniciarmos esta intenção, vamos nos ater às particularidades deste projeto. Quando iniciarmos a intenção por uma ação, sabemos que uma nova Activity que responde àquela ação será iniciada.

Também sabemos que, após um valor da nova Activity ser selecionado, ela deverá ser encerrada, retornando o valor selecionado para a Activity de nosso aplicativo. Ou seja, desta vez desejamos solicitar uma Activity aguardando um resultado. Portanto, devemos informar o request code para esta Activity, para que possamos mais tarde recupera-lo em nosso listener **onActivityResult**.

Mais uma vez então criaremos uma variável final que identifica este código, tornando assim nosso código fonte mais legível:

```
static final int REQUEST_SELECT_PHONE_NUMBER = 1;
```

Agora, tudo que nos resta é executar a chamada. Para que possamos iniciar a partir de uma intenção implícita, devemos informar a ação desejada:

```
ACTION_PICK
```

Para este tipo de ação, no caso do aplicativo de contatos, devemos também passar um MIME Type do tipo:

```
Contacts.CONTENT_TIPE

Intent intent = new Intent(Intent.ACTION_PICK);
```

```
intent.setType(ContactsContract.CommonDataKinds.Phone.CONTENT_TYPE);
```

Após definirmos a intenção implícita e seu tipo de MIME, devemos consultar o sistema e verificar se existe alguma Activity vinculada a este tipo de ação.

```
if (intent.resolveActivity(getPackageManager()) != null){
```

Caso haja uma activity vinculada, chamamos o método startActivityForResult, passando o request code desejado.

```
startActivityForResult(intent,
REQUEST_SELECT_PHONE_NUMBER);
```

Devemos agora redefinir o método onActivityResult, para que, ao retorno para esta Activity, possamos validar se o requestCode retornou o valor desejado e se o resultCode foi de sucesso:

```
RESULT_OK

if (requestCode == REQUEST_SELECT_PHONE_NUMBER &&
resultCode == RESULT_OK){
```

Caso seja positivo, podemos então recolher os dados que foram transmitidos através da Intent. Verificando o manual para o ACTION_PICK, descobrimos que a informação retornada vem no formato Uri.

```
Uri contactUri = data.getData();
```

De posse dessa informação, podemos agora utilizar a Uri retornada aliada a um cursor, para obtermos acesso à informação desejada.

```
String[] projection = new
String[]{ContactsContract.CommonDataKinds.Phone.NUMBER};
```

```
Cursor cursor = getContentResolver().query(contactUri, projection, null, null, null);
```

Caso o cursor não seja um objeto nulo (ou seja, não inicializado), e caso existam registros nele, então buscamos o número do telefone:

```
if (cursor != null && cursor.moveToFirst()){
   int numberIndex = cursor.getColumnIndex(ContactsContract.CommonDataKinds.Phone.NUMBER);
   txtNumero.setText(cursor.getString(numberIndex));
}
```

Como não utilizamos neste exemplo acesso direto à lista de contatos do dispositivo, não necessitamos solicitar permissão dentro do arquivo de manifesto.

Para sua comodidade, segue o código fonte completo de nosso arquivo MainActivity.java (no material *on-line).*

Tema 3 - Filtros de intenção

Intenções implícitas são intents criadas que buscam aplicativos aptos a realizar determinada ação. Conforme vimos, existem várias ações padrão definidas no ambiente Android, além de ações registradas especificamente por aplicativos.

Agora, aprenderemos como registrar uma ação padrão a ser respondida por uma Activity em nosso aplicativo. Desta maneira, podemos sugerir ao sistema operacional que uma das Activities de nosso aplicativo está apta a responder a uma determinada necessidade.

Para que possamos anunciar quais intenções implícitas nosso aplicativo está apto a responder, devemos declarar um ou mais

filtros de intenção nos componentes de nosso aplicativo com um elemento <intent-filter> dentro do nosso arquivo de manifesto.

Cada filtro de intenção especifica o tipo de intenção aceito com base na ação dos dados e na categoria da intenção. Sempre que houver uma solicitação de uma intenção implícita, o sistema Android buscará quais componentes estão aptos a responde-la e então passará a eles somente se ela estiver registrada para isso.

Outro ponto importante é o fato de que os aplicativos devem declarar filtros separados para cada trabalho exclusivo que podem fazer. Por exemplo, sua atividade pode exibir dados de um contato ou permitir a sua alteração.

Para nosso teste, criaremos um aplicativo que registrará uma ação que retorna a data atual.

Crie um novo projeto no Android Studio chamado RespondeData. No layout activity_main.xml adicionaremos um Button com as seguintes propriedades:

```
id: btnData

text: Que dia é hoje?
```

E um TextView com as seguintes propriedades:

```
id: txtData;

text: Hoje é
```

Para sua comodidade, segue o xml completo da MainActivity (no material *on-line).*

Agora devemos instanciar os objetos que representam este TextView e o Button:

```
txtData = (TextView) findViewById(R.id.txtData);
btnData = (Button) findViewById(R.id.btnData);
```

Redefinimos então o comportamento do listener OnClickListener de nosso button:

```
btnData.setOnClickListener(new View.OnClickListener() {
    @Override
    public void onClick(View view) {

    }
});
```

Estamos prontos agora para chamar a Intent que retornará a data atual. Vamos criar agora uma segunda Activity, chamada DataIntent.

Lembre-se de utilizar o wizard do Android Studio para criação de uma nova Activity, para que este já crie também automaticamente o arquivo xml referente ao layout e também efetue o registro necessário em nosso arquivo de Manifesto. Nesta nova Activity teremos um TextView que apresenta a data atual e um button que retorna este resultado para quem chamou a Activity:

```
textView:

id: txtDataAtual

text: Data Atual

Button

id: btnRetornaData
```

```
text: Retorna a Data
```

Para sua conveniência, segue o arquivo XML de nosso arquivo activity_data.xml (no material *on-line).*

Devemos agora criar os objetos em nossa classe DataActivity.java para que possamos vincula-los ao layout:

```
TextView txtDataAtual;
Button btnRetornaData;
```

E instanciá-los corretamente:

```
txtDataAtual = (TextView) findViewById(R.id.txtDataAtual);
btnRetornaData = (Button) findViewById(R.id.btnRetornaData);
```

Antes de mais nada, vamos capturar a data atual e atribui-la ao nosso comonente txtDataAtual.

```
SimpleDateFormat sdf = new SimpleDateFormat("dd/MM/yyyy");
String data = sdf.format(new Date(System.currentTimeMillis()));
txtDataAtual.setText(data);
```

Agora, devemos redefinir o listener de btnRetornaData.setOnclickListener, para que este retorne a data atual:

```
btnRetornaData.setOnClickListener(new View.OnClickListener()
{
    @Override
    public void onClick(View view) {
        Uri retorno = Uri.parse(txtDataAtual.getText().toString());
        Intent intent = new Intent();
        intent.setData(retorno);
        setResult(RESULT_OK, intent);
```

```
        finish();
    }
});
```

Para sua referência, segue o código fonte completo de nossa Activity DataActivity.java:

Voltando a nossa MainActivity, devemos criar uma variável para controlar o nosso request code, já que mais uma vez iniciaremos uma activity aguardando um resultado:

```
final static int BUSCA_DATA = 1;
```

Redefinimos agora o listener onClickListener de nosso btnData, para que este inicie uma Activity aguardando um resultado, e utilizando o requestCode BUSCA_DATA:

```
btnData.setOnClickListener(new View.OnClickListener() {
    @Override
    public void onClick(View view) {
        Intent intent = new Intent(view.getContext(),
DataActivity.class);
        startActivityForResult(intent, BUSCA_DATA);
    }
});
```

Finalmente, redefinimos o método onActivityResult para que processe o resultado retornado na intente, quando a Activity DataActivity for finalizada:

```
@Override
protected void onActivityResult(int requestCode, int
resultCode, Intent data) {

    if (requestCode == BUSCA_DATA && resultCode ==
RESULT_OK){
```

```
        txtData.setText(Uri.decode(data.getDataString()));
    }

}
```

Aqui está o código completo para sua referência. Execute este código (no material *on-line).*

Você deve ter percebido que o método que utilizamos até o momento foi exatamente igual ao dos exercícios de startActivityForResult, ou seja, não utilizamos ainda intenções implícitas. Para que possamos utiliza-las, lembre-se que devemos antes de mais nada registrar o filtro de intenção em nosso arquivo AndroidManifest.

```
<activity android:name=".DataActivity">
    <intent-filter>
        <action
android:name="br.com.grupouninter.aula.respondedata.RET
ORNADATA" />
        <category
android:name="android.intent.category.DEFAULT" />
    </intent-filter>
</activity>
```

Para sua referência, aqui está o código completo de nosso AndroidManifest.xml (no material *on-line).*

Agora que registramos esta ação em nossa Actvity, qualquer aplicativo pode executa-la através de uma intenção implícita que a chame:

```
br.com.grupouninter.aula.respondedata.RETORNADATA
```

Volte então a nosso arquivo MainActivity.java e altere o método onClick de nosso btnData, para que agora não mais inicie uma activity a partir do nome de sua classe, mas sim a partir de uma intenção de ação:

```
btnData.setOnClickListener(new View.OnClickListener() {
    @Override
    public void onClick(View view) {
        Intent intent = new
Intent("br.com.grupouninter.aula.respondedata.RETORNADA
TA");
        if (intent.resolveActivity(getPackageManager()) != null){
            startActivityForResult(intent, BUSCA_DATA);
        } else {
            Toast.makeText(view.getContext(), "nao achei",
Toast.LENGTH_LONG);
        }

    }
});
```

Segue o código completo para sua referência (no material *on-line).*

Tema 4 - Serviços

Um serviço é um componente que representa o desejo da aplicação de executar uma tarefa de longa duração sem interação com o usuário final, ou suprir funcionalidades para uso de outras aplicações.

No entanto, é importante ressaltarmos que um serviço não é um processo separado, tampouco é uma Thread. Portanto, os serviços são executados no processo principal que os solicitam. Isto significa que, caso você deseje executar algo que faça uso

intenso da CPU, tal qual tocar uma música MP3, ou ainda que execute qualquer função de bloqueio, como operações de rede, estes devem ser executados em sua própria linha de processamento.

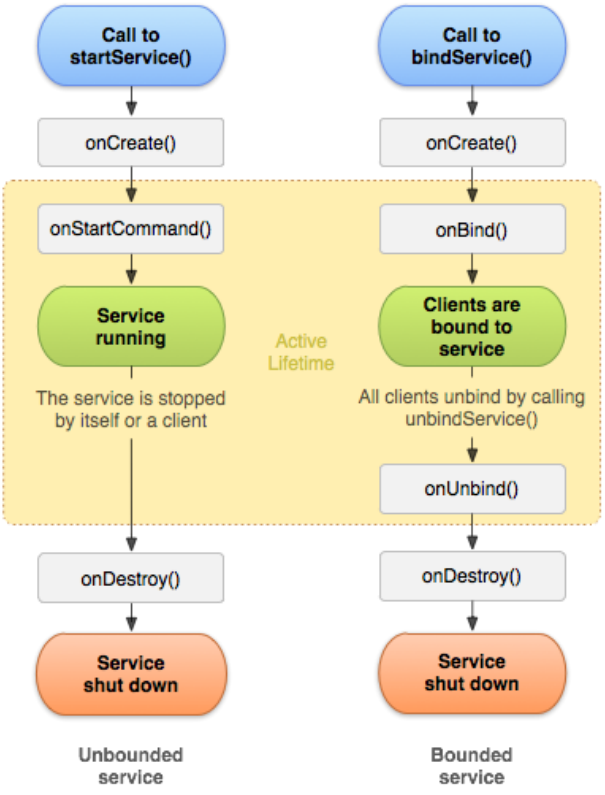
Um serviço, na verdade, busca facilitar à aplicação para que diga ao sistema algo que deseje executar em segundo plano, mesmo quando o usuário não esteja interagindo com a aplicação, e também facilita a exposição de algumas funcionalidades a outras aplicações.

O ciclo de vida de um serviço é muito mais simples do que o de uma Activity e pode seguir por dois caminhos:

Serviço iniciado: criado quando outro componente chama startService() e permanece em execução indefinidamente, devendo ser interrompido pela chamada stopSelf(). Outro componente pode também interrompê-lo chamando o método stopService(). Quando um serviço é interrompido, o sistema o elimina.

Serviço vinculado: é criado quando outro componente executa o método bindService. O componente que o chamou comunica-se então com o serviço através da interface IBinder, e vários componentes podem ser vinculados a um mesmo serviço. Quando todos os vínculos terminam o sistema destrói o serviço, ou seja, o serviço não precisa ser interrompido por uma chamada ao final.

Na imagem a seguir, podemos visualizar o ciclo de vida de um serviço:



A principal diferença entre um serviço iniciado e um serviço vinculado (bind) é que um serviço iniciado será executado em segundo plano até que alguém o finalize ou este finalize sozinho. Ele é usado para executar operações que possam levar muito tempo. Um serviço vinculado (bind) será executado enquanto houver uma ligação com algum componente e este interaja com ele. Além disso, serviços vinculados podem retornar valores aos componentes com os quais estão vinculados.

Serviços iniciados ficarão na memória mesmo depois que o componente que os iniciou já não esteja mais em memória, ou seja, encerrado. Um bom exemplo de uso de serviços iniciados são envio de mensagens SMS, download de arquivos, etc.

Serviços vinculados são executados quando existe a necessidade de retorno para o componente que os iniciou. Como o fluxo de dados acontece do serviço para o componente, um canal de comunicação é criado. Tal qual Activities, os serviços

também requerem sua definição no arquivo de Manifesto de sua aplicação, para que possam ser executados.

Vamos colocar este conceito à prova criando um aplicativo com seu próprio serviço. Inicie seu Android Studio e então crie um novo projeto chamado PrimeiroServico.

Para este aplicativo, teremos apenas um Button para o início do serviço em nosso layout principal, com as seguintes propriedades:

```
id: btnIniciaServico

text: Iniciar o Serviço
```

Para sua referência, segue o arquivo activity_main.xml completo (no material *on-line).*

Mais uma vez, devemos criar um objeto que faça referência ao componente de layout:

```
Button btnIniciaServico;
```

Iniciamos este objeto:

```
btnIniciaServico = (Button)
findViewById(R.id.btnIniciaServico);
```

Redefinimos seu listener onClickListener. Agora devemos criar a classe que representará este serviço.

Crie uma nova classe java chamada MeuServico.

```
package br.com.grupouninter.aula.primeiroservico;
/**
 * Created by Marcelo on 20/08/2016.
```

```
 */
public class MeuServico {
}
```

A primeira tarefa que temos é passar a nossa classe as propriedades e comportamentos da classe de serviço.

```
public class MeuServico extends Service {
```

Uma vez que definimos que a nossa classe MeuServico é uma classe filha de Service, devemos implementar os métodos obrigatórios a serviços:

```
@Override
public IBinder onBind(Intent intent) {
    return null;
}
@Override
public int onStartCommand(Intent intent, int flags, int startId) {
}
@Override
public void onDestroy(){
    super.onDestroy();
}
```

Para serviços, existem dois modos de operação principais que dependem de qual valor de retorno foi definido no método onStartCommand:

START_STICKY: se o sistema eliminar o serviço após o retorno de onStartCommand(), reinicie o serviço e chame onStartCommand(), mas não entregue novamente à ultima intenção. Ao invés disso, o sistema chama o onStartCommand() com a intenção nula, a não ser que tenha intenções pendentes para iniciar o serviço.

START_NOT_STICKY: se o sistema eliminar o serviço após o retorno de onStartCommand(), não recrie o serviço a não ser que tenha intenções pendentes para entregar. Esta opção é mais segura para evitar executar o serviço quando desnecessário.

START_REDELIVER_INTENT: se o sistema eliminar o serviço após o retorno de onStartCommand(), recrie o serviço e chame onStartCommand() com a última intente que foi entregue ao serviço. É utilizado em serviços que estejam realizando um trabalho que deva ser retomado imediatamente, como o download de um arquivo.

Para nosso exemplo, tudo que faremos será monitorar o ciclo de vida de um Serviço. Para isso, antes de mais nada declararemos nossa variável estática, que indica qual classe estamos monitorando.

```
private static String TAG = "MeuServico";
```

E então redefinimos os métodos onStart, onBind e onDestroy para que anunciem sua execução no debugger do Android Studio:

```
@Override
public IBinder onBind(Intent intent) {
    Log.d(TAG, "onBind()");
    return null;
}
@Override
public int onStartCommand(Intent intent, int flags, int startId) {
    Log.d(TAG, "onStartCommand()");
    return START_NOT_STICKY;
}
@Override
public void onDestroy(){
```

```
    Log.d(TAG, "onDestroy()");
    super.onDestroy();
}
```

Finalmente, forçaremos a finalização de execução deste serviço em nosso método onStart, uma vez que nosso serviço não é um serviço vinculado:

```
@Override
public int onStartCommand(Intent intent, int flags, int startId) {
    Log.d(TAG, "onStartCommand()");
    this.stopSelf();
    return START_NOT_STICKY;
}
```

Segue o código completo da classe MeuServico.java, para sua referência (no material *on-line).*

Devemos agora definir nosso serviço no Manifesto de nosso aplicativo, para que ele possa ser executado:

```
<service android:name=".MeuServico" ></service>
```

Segue o código completo do arquivo AndroidManifest.xml, para sua referência (no material *on-line).*

Finalmente, em nossa Activity, no método onClick do nosso componente Button, inicializaremos nosso novo serviço:

```
btnIniciaServico.setOnClickListener(new View.OnClickListener()
{
    @Override
    public void onClick(View view) {
        Intent intent = new Intent(view.getContext(),
MeuServico.class);
        startService(intent);
```

```
    }
});
```

Para sua referência, segue o código fonte completo de nossa MainActivity.java (no material *on-line).*

Por fim, execute o aplicativo e monitore o ciclo de vida do serviço no debugger do Android Studio.

Na prática

Vamos aprofundar nossos estudos? Para isso, lançamos os seguintes desafios práticos relacionados aos conteúdos estudados até aqui:

- Quais outras informações são possíveis de se obter utilizando a classe ContactsContract? Expanda seu aplicativo, fornecendo maiores informações a respeito do usuário.
- Utilize o conceito de adaptadores e ListView para criar um aplicativo de Contatos com maior usabilidade.
- Você consegue estender nosso aplicativo de ContatosPorResultado para que apresente dados como nome, endereço, e-mail(s), telefone(s), etc, em uma activity que se assemelhe à Activity de detalhes de contato do aplicativo padrão de contatos?
- Crie um aplicativo que se utilize da intenção implícita:

br.com.grupouninter.aula.respondedata.RETORNADATA

- Você consegue ampliar a DataActivity para que retorne não somente a Data, mas também a Hora e dia da Semana? Lembre-se de enviar estas informações em uma estrutura, não diretamente em modo texto.

- Crie um serviço vinculado para que seu ciclo de vida seja ligado à Activity que o criou. Lembre-se que em serviços vinculados você não tem a necessidade de finalizá-los utilizando o método stopSefl().

Síntese

Nessa aula trabalhamos o conceito de permissões do Android, utilizando o Content Provider de contatos para buscar todos os contatos disponíveis no dispositivo. Trabalhamos também a utilização de intenções implícitas, buscando componentes que respondessem ao tipo de ação declarada.

Como prova de conceito, criamos um aplicativo que se utiliza da Activity de contatos disponível em dispositivos Android para retornar informações de um único contato, sem a necessidade de se declarar as permissões requeridas para tanto.

Disponibilizamos também uma Activity que se registrou ao sistema Android como disponível para a execução de uma determinada ação e, finalmente, trabalhamos o conceito de serviços dentro do Android, criando uma classe que monitora o ciclo de vida de um serviço e declarando-a em nosso Manifesto.

Referências

ADAPTER. Disponível em: <https://developer.android.com/reference/android/widget/Adapter.html>. Acesso em: 25 ago. 2016.

DEITEL, Paul; DEITEL: Harvey; WALD, Alexander. **Android 6 para Programadores:** Uma Abordagem Baseada em aplicativos. Porto Alegre: Bookman, 2016.