UNIDADE 3 – CONCEITOS BÁSICOS DE UM APLICATIVO ANDROID
MÓDULO 1 – MODELOS TRADICIONAIS X MODELO ANDROID
01

1 - CARACTERÍSTICAS DO MODELO ANDROID

Partindo dos fundamentos apresentados nas unidades anteriores para criação de um código Java robusto e íntegro, esta unidade apresentará os principais conceitos de alto nível envolvidos na programação para a plataforma Android.

Android é uma plataforma de *software* que está revolucionando o mercado global de aplicativos para celulares. É a primeira plataforma de aplicativos para telefones celulares de código-fonte aberto baseado no kernel denominado de Linux e na máquina virtual (Dalvik ou ART).

O Android é lançado com duas licenças diferentes de código fonte aberto. O kernel linux é regido, em sua grande maioria, pela licença GNU GPLv2 enquanto que o restante da plataforma Android é regido pela licença ASL (*Apache Software License*). As duas licenças são orientadas a código-fonte aberto, sendo que a ASL é mais adequada ao ambiente comercial que a GNU GPL, principalmente em sua versão 3, que possui uma série de restrições que vão, literalmente, de encontro ao uso de patentes e direitos autorais (*copyright*). Já a licença ASL não oferece restrições ao uso conjunto entre *software*s/bibliotecas/módulos que sejam proprietários ou não.

Atualmente, o principal concorrente do Android é o IOS da Apple cujas bases são provenientes do 4.3BSD (*Berkley Software Distribution*) e da versão pública disponibilizada pela própria Apple do sistema operacional que ela mesma desenvolveu cujo nome é Darwin. Ambos os *software*s, BSD e Darwin também são *software*s livres, ressalvadas algumas diferenças importantes, cujo debate está além do escopo desta disciplina. Contudo, suas licenças são mais permissivas que a GNU GPL permitindo aos usuários que, por exemplo, alterarem o *software*, a não obrigatoriedade de repasse do conhecimento sob as mesmas licenças.

A variedade de linguagens de programação comerciais existentes atualmente no mercado é um diferencial percebido pelos desenvolvedores do sistema Android. Isso significa que pelo fato do sistema Android ser totalmente aberto ele permite que os desenvolvedores do mundo inteiro possam escolher em qual linguagem de programação desejam desenvolver seus aplicativos. Por óbvio, a escolha de qual linguagem utilizar pode ocasionar maior facilidade ou dificuldade neste desenvolvimento. Portanto, linguagens como Assembly, C, Java, PHP, Python, C++, Lua, Ruby, Perl, dentre outras podem ser usadas para desenvolver aplicativos Android. O ponto importante a ser observado quanto a isso é que o Android oferece suporte nativo a algumas delas e a outras não. Isso significa que pelo fato do Android usar o kernel Linux, praticamente, todas as linguagens de programação existentes atualmente podem ser executadas, seja de modo nativo (já suportado pelo ambiente) ou não (geralmente suportado após a instalação de componentes externos/extras). Por exemplo, o Android, até a sua versão 6.0 - Masrhmallow oferece suporte nativo as linguagens Assembly, C e Java. Já para linguagens como PHP e

Python é necessário a instalação de ferramentas extras ao sistema como o SL4A (https://github.com/damonkohler/sl4a).

IOS
O IOS é considerada uma plataforma de código proprietário e fechado, uma vez que a Apple não disponibiliza as alterações realizadas nos mesmos.

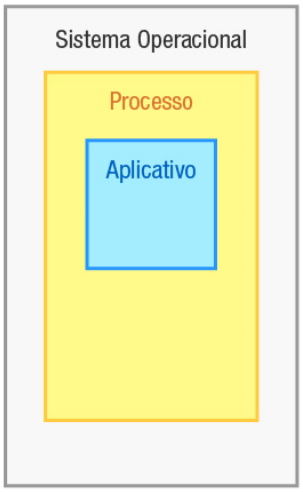
02

2 - Inicialização de Aplicativos

Quando inicializam os aplicativos, sistemas operacionais utilizam, geralmente, um único ponto de entrada, muitas vezes chamado de “main”, que pode processar alguns argumentos de linha de comando e então avançar, de modo iterativo ou não, realizando cálculos e demonstrando resultados.

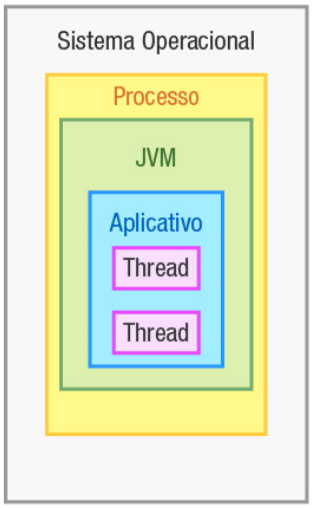
A inicialização desses aplicativos se dá por meio do trabalho do sistema operacional em criar um processo e, posteriormente, carregar o código binário do programa dentro desse processo, iniciando, posteriormente, sua execução, geralmente, por meio do ponto único denominado de “main”.

A figura abaixo é um exemplo de alto nível do conceito ora apresentado.



03

Com programas escritos em Java, as coisas são um pouco mais complexas, uma vez que a JVM (Java Virtual Machine ou Máquina Virtual de Java) introduz, no mínimo, uma camada a mais de abstração, conforme mostra a figura abaixo:



O Android, por sua vez, apresenta uma abordagem ainda mais rica e complexa, o que significa que mais camadas de abstração foram introduzidas neste tipo de ambiente. Programas Android oferecem suporte a múltiplos pontos de entrada, ao invés do tradicional ponto único (main). Portanto, programas Android devem prever que o sistema poderá inicializá-los em locais diferentes, dependendo do ponto em que se encontra o usuário e do que ele deseja fazer.

Desta forma, um programa Android pode ser considerado um grupo de componentes cooperantes que podem ser inicializados a partir de um ponto fora do fluxo normal de seu aplicativo.

Por exemplo, um componente para leitura de código de barras fornece uma função que muitos aplicativos podem integrar ao fluxo de suas interfaces. Em vez que depender do usuário para iniciar diretamente cada aplicativo, os próprios componentes invocam uns aos outros, realizando interações em nome do usuário por delegação.

04

3- Activity, Intent e Task

Uma ***Activity*** do Android é tanto uma unidade de interação do usuário – tipicamente preenchendo por completo a tela de um dispositivo móvel – quanto uma unidade de execução.

Quando você cria um programa Android interativo, inicia criando subclasses da classe Activity. Activities fornecem os componentes reutilizáveis e intercambiáveis do fluxo de componentes da interface gráfica com o usuário (GUI) em aplicativos Android.

Já classe ***Intent*** é utilizada para representar a intenção de um usuário, ou seja, aquilo que o usuário deseja fazer.

Intents são importantes por que eles não apenas facilitam a navegação de modo inovador, mas representam um dos aspectos mais importantes na codificação Android, que é o baixo acoplamento entre os componentes. Veja um exemplo.

Desta forma, a classe *Intent* representa o protocolo de comunicação abstrato entre, por exemplo, as *Activities,* formando a base de um mecanismo de baixo acoplamento que permitem as *Activities* se comunicarem umas com as outras, independentemente da ordem e do fluxo comum imposto por muitos aplicativos desenvolvidos em um contexto tradicional de programação. Quando um aplicativo qualquer dispara um *Intent*, é possível que, com o passar do tempo, *Activities* diferentes possam atender a operação, uma vez que tal atendimento depende de quais *Activities* estão atualmente registradas para tratar determinados *Intents*.

Exemplo
Por exemplo, a classe *Intent* representa a descrição abstrata da função que uma *Activity* requer que outra *Activity* desempenhe, como procurar um contanto na agenda, abrir um site web, tirar uma foto, tocar uma música, dentre outros.

05

Assim como a classe *Intent*, a classe Activity é uma das mais importantes no sistema Android, promovendo a modularidade dos aplicativos e permitindo o compartilhamento de funcionalidades. Um Activity interage com o tempo de execução do Android, para implementar aspectos essenciais do ciclo de vida do aplicativo. Contudo, apesar do *framework* do Android disponibilizar tal recurso, o seu uso, por muitas vezes se faz de modo diverso daquele para o qual foi projetado.

Deste modo, cada Activity em um aplicativo Android deve estar amplamente separada das outras, o que significa que o código que implementa uma Activity não deve chamar diretamente métodos no código que implementa outra Activity. Portanto, não é recomendado que o seu aplicativo mantenha referências a outras Activities.

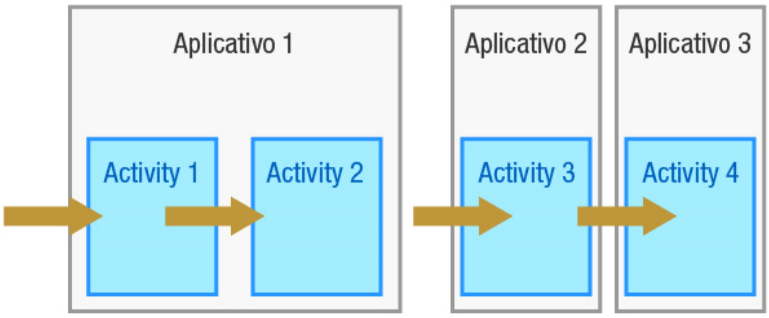
O ambiente de tempo de execução do Android cria e gerencia Activities e outros componentes do aplicativo, muitas vezes recuperando a memória que eles utilizam para manter as tarefas Java individuais em quantidades relativamente pequenas na memória. Portanto, não tente gerenciar esses componentes de forma personalizada.

Em aplicativos Android o fluxo de execução do processamento não deve ser visto como em um aplicativo convencional. Saiba+

Esse fluxo resultante de diversas Activities se comunicarem por meio de diversos *Intent*s é denominado de **Task**.

Task é uma cadeia de Activities que, muitas vezes, se estende a mais de um aplicativo e a mais de um processo.

A figura mostra um possível exemplo de como essa Task pode se materializar.



Nesta figura, a cadeia de Activities que compõem essa Task abrange três processos e heaps separados, e pode existir independente de outras Task que possam ter iniciado instâncias distintas das mesmas subclasses de Activity.

Saiba+
Em vez de um controle de fluxo da interface do usuário baseado em chamadas de métodos, os aplicativos Android descrevem *Intent*s que desejam executar e solicitam ao sistema operacional Android que encontre implementações que possam atender à solicitação.

06

4 - O AMBIENTE DE TEMPO DE EXECUÇÃO DE UM APLICATIVO ANDROID

A implementação diferenciada da arquitetura de componentes de um aplicativo Android é, em parte, resultado da forma como ele implementa um ambiente de multiprocessamento em Java por meio da VM (Virtual Machine).

Para tornar esse ambiente adequado a aplicativos múltiplos e fornecedores distintos, apresentando um requisito mínimo de confiança para cada fornecedor, o Android, até a versão 4.4 cujo apelido é Kitkat, utilizava-se da VM denominada de Dalvik.

A partir da versão 5.5 cujo apelido é Lollipop, a plataforma Android passou a utilizar uma nova VM denominada de ART (Android Runtime). Apesar de diferentes, as duas máquinas virtuais cumprem-se ao mesmo propósito: criar um ambiente de execução independente de plataforma para os aplicativos Java. Contudo, as diferenças entre ambas serão mostradas logo adiante.

Por padrão, o Android definiu executar cada aplicativo Android como sendo um processo separado e independente dos demais. Para isso, cada processo executa dentro de sua própria instância da VM, o que consequentemente, obriga a VM a dividir a memória de trabalho de modo eficiente para permitir

que muitos aplicativos possam ser executados simultaneamente em uma memória de trabalho muitas vezes limitada em termos de espaço e velocidade.

07

A abordagem do Android de multiprocessamento em Java, utilizando múltiplos processos e múltiplas instâncias de uma máquina virtual, requer que cada instância de máquina virtual seja eficiente em termos do espaço que ocupa.

Uma implicação direta da falta de memória em função, por exemplo, da quantidade de processos em execução simultânea é o término automático, por parte da política de escalonamento do sistema operacional Android, daquele processo considerado como sendo de menor prioridade. Ou seja, o processo deve ter uma forma de dar continuidade ao seu processamento, caso seja o escolhido para deixar a memória RAM.

Esse ponto é fundamental na arquitetura do Android, ou seja, para que o processo possa dar continuidade ao seu processamento após inúmeras interrupções, é fundamental que o programador compreenda o **ciclo de vida dos componentes** (Activity, Service, BroadcastReceiver, ContentProvider) da API do Android que possuem comportamentos predefinidos que serão estudados ao longo deste curso.

Observe que a utilização eficiente da memória depende da forma e do uso que se faz dos componentes envolvidos, ou seja, apesar de todos os recursos e facilidades que o *framework* Android criou para tornar o uso da memória eficiente, esses recursos por si só não são capazes de garantir o uso eficiente da memória de trabalho.

Os programadores que escrevem aplicativos para a plataforma Android devem compreender a forma correta de uso dos componentes para que toda a estrutura já existente da API Android possa funcionar de modo eficiente para o qual foi projetado. Em contrapartida, o desconhecimento do ciclo de vida dos componentes torna o uso dos mesmos equivocado, o que, fatalmente, levará a uma má utilização da memória de trabalho e, consequentemente, problemas (bugs) das mais variadas formas e aspectos ocorrendo em tempo de execução.

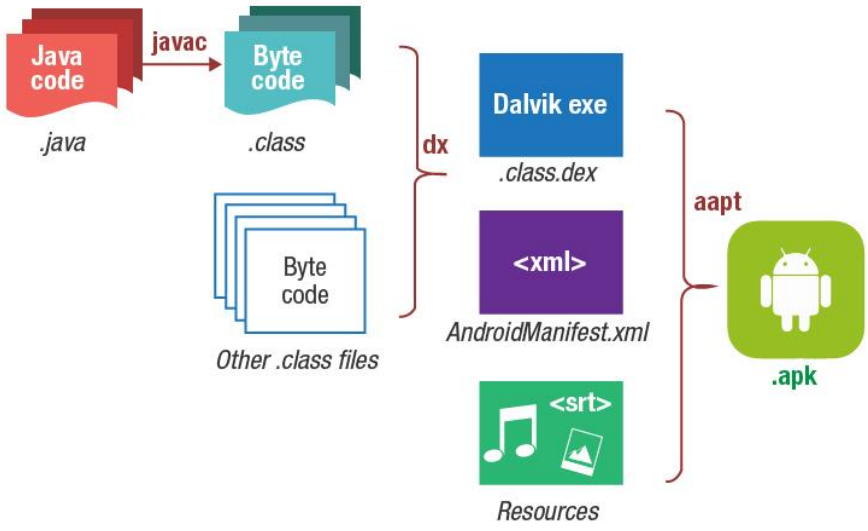
08

4.1- VM Dalvik

A máquina virtual Dalvik executa um sistema de bytecodes desenvolvido especificamente para ela, o **dex**. Bytecodes **dex** são, aproximadamente, duas vezes mais eficientes em termos de espaço do que os bytecodes Java nativo, inerentemente diminuindo pela metade o custo de memória das classes Java e cada processo.

A Dalvik, por meio do sistema operacional Android, também se utiliza da técnica denominada de “copy-on-write” para compartilhar memória entre múltiplas instâncias do mesmo executável Dalvik. Além disso, os arquivos que representam um aplicativo Android (apk) são compostos pelos recursos (imagens, vídeos, áudio, *streaming*, dentre outros) e pelo arquivo xml denominado de AndroidManifest.xml.

Abaixo, a figura abaixo representa como isso funciona:



09

4.2- Dalvik x ART

A máquina virtual ART é, em sua maioria, compatível com a máquina virtual Dalvik. Isso significa que para o usuário do sistema operacional Android essa modificação é totalmente transparente não necessitando qualquer intervenção por parte dos usuários. A ART é capaz, inclusive, de executar os aplicativos dex da máquina virtual Dalvik. A mudança de máquina virtual pelo sistema operacional Android se deu por motivos de otimização e melhoria na performance da plataforma em geral, procurando fazer um melhor uso de recursos considerados escassos como bateria, por exemplo.

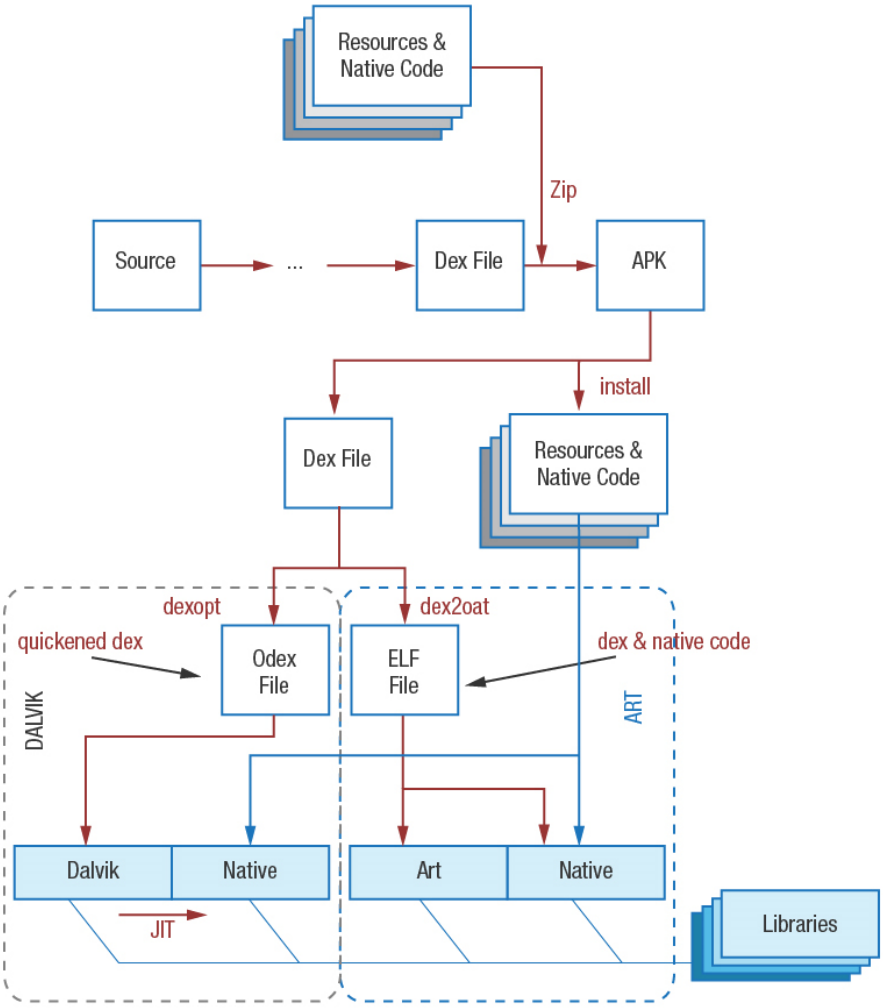
A diferença essencial entre a máquina virtual ART e a Dalvik se deve ao fato da ART compilar o arquivo dex do aplicativo em um código binário nativo para a plataforma Android cujo formato é denominado de ELF (Executable and Linkable Format). Esse procedimento é denominado de ***ahead-of-time*** (AOT) *compilation*.

Essa técnica é feita no momento da instalação do aplicativo e os benefícios desse procedimento são a eliminação da interpretação do bytecodes e dos traces JIT durante a execução do arquivo dex tornando-o de processamento mais rápido. Isso elimina uma quantidade significativa de overhead de processamento o que, consequentemente, economiza bateria aumentando a longevidade, a

durabilidade e a autonomia da mesma. Por outro lado, essa técnica necessita de um tempo maior para a instalação do aplicativo e também um maior espaço de armazenamento na memória secundária (flash memory).

10

A figura a seguir demonstra as diferenças entre a máquina virtual ART e a Dalvik.



11

Resumo

O objetivo deste módulo foi apresentar os pilares de sustentação que compõem o modelo de programação e a estrutura essencial na qual está implantada o sistema operacional Android. O modelo tradicional de programação sugere que os aplicativos tenham apenas um único ponto de entrada enquanto que o modelo Android de programação sugere que os aplicativos tenham vários pontos de entrada. Além disso, dois componentes essenciais foram apresentados: activty e intent. Uma Activity é,

basicamente, um componente gráfico da GUI (Grafical User Interface), ou seja, uma janela que se apresenta ao usuário. Já uma Intent, representa um mecanismo de IPC (Inter Process Comunication) poderoso que carrega consigo a intenção do usuário em fazer algo, ou seja, é a forma que o Android criou para permitir que componentes diversos do sistema possam se comunicar por meio de uma interface padronizada garantido que tais componentes trabalhem de modo fracamente acoplado.

No que diz respeito à estrutura essencial do Android, a mesma apresenta os conceitos relativos a VM Dalvik e a VM ART que são maquinas virtuais que executam bytecodes específicos denominados de dex. Bytecodes dex são, aproximadamente, duas vezes mais eficientes em termos de espaço do que os bytecodes Java nativo, o que em ambientes móveis é algo considerado crucial e necessário.

UNIDADE 3 – CONCEITOS BÁSICOS DE UM APLICATIVO ANDROID
MÓDULO 2 – OUTROS COMPONENTES ESSENCIAIS DO ANDROID
01

1 - SERVICE

No módulo anterior foram apresentados dois componentes importantes: *Activity* e *Intent*. Continuando a descrever os conceitos básicos sobre o sistema operacional Android, descreveremos agora outros três componentes importantes (*Service, Content Provider* e *Broadcast Receiver*) bem como a estrutura de diretórios de um típico projeto Android.

Antes de iniciar a explicação detalhada sobre cada componente, considere que cada um deles tem um propósito/objetivo e desta forma, pode-se resumir que:

A classe ***Service*** oferece suporte a funções não baseadas na interface gráfica do usuário (GUI).	A classe ***ContentProvider*** fornece novas APIs de interação de dados.	A classe ***BroadcastReceiver*** permite que múltiplos participantes escutem a transmissão de Intents.

É importante perceber que o aplicativo Android em si, se bem projetado, é uma unidade relativamente dissolvida no ambiente do Android, uma vez que esse aplicativo pode iniciar Activities em outros aplicativos para “pegar emprestado” suas funcionalidades e fornecer ou aprimorar suas próprias funcionalidades utilizando componentes de suporte do Android.

A classe *Service* do Android é utilizada em tarefas de segundo plano que podem estar ativas, mas não visíveis na tela. Exemplo

Um ponto importante é que a plataforma Android evita recuperar recursos de Service quando da escassez de memória de trabalho disponível, o que significa que uma vez iniciado um objeto do tipo *Service*, é provável que ele continue executando por um longo período de tempo. Portanto, os objetos do tipo *Service* somente serão retirados da memória em casos que o sistema operacional Android julgar de extrema necessidade.

Exemplo
Um aplicativo tocador de música provavelmente terá uma classe *Service* implementada para que possa funcionar enquanto o usuário, por exemplo, navega em um site web.

02

2 - CONTENT PROVIDER

Content Provider (Provedores de Conteúdo) no Android são praticamente análogos a um serviço web RESTful, uma vez que o acesso aos mesmos é feito por meio de URIs e as operações (CRUD) semelhantes àquelas orientadas a recursos.

Uma URI especial iniciada por ***content://***, reconhecido no dispositivo local, pode fornecer acesso aos dados do *Content Provider*. Para utilizar um *Content Provider* é necessário especificar uma URI e uma operação que indica o que deve ser feito nos dados referenciados.

REST significa “Representational State Transfer” e nada mais é que um conceito formalizado para uso do protocolo HTTP como meio de acesso facilitado aos dados de um sistema qualquer.

Ainda que implementações do REST possam apresentar diferenças, todas elas são suficientemente simples e semelhantes para que sejam prontamente estendidas. A classe *Content Provider* do Android é uma implementação de operações do tipo REST.

Por exemplo:

- **Inserção** (*insert*);
- **Consulta** (*query*);
- **Atualização** (*update*);
- **Exclusão** (*delete*);

Inserção
O método *insert* da classe *Content Provider* é análogo à operação REST POST e insere novos registros no banco de dados.

Consulta
O método *query* da classe *Content Provider* é análogo à operação REST GET e retorna um conjunto de registros em uma classe especializada de coleção chamada *Cursor*.

Atualização
O método *update* da classe *Content Provider* é análogo à operação REST UPDATE e substitui registros nos banco de dados por registros atualizados.

Exclusão
O método *delete* da classe *Content Provider* é análogo à operação REST DELETE e remove os registros indicados no banco de dados.

03

Um ponto importante a ser percebido é que o *Content Provider* é central quando se trata de conteúdo dos aplicativos no sistema operacional Android, pois é por meio dele que se torna possível aos aplicativos compartilharem dados e, por vezes, até gerenciar o modelo de dados.

Uma outra classe que trabalha conjuntamente com essa, é a classe ***Content Resolver***.

A classe ***Content Resolver*** permite que outros componentes em um sistema Android acessem provedores de conteúdo.

Por exemplo, os aplicativos centrais do Android utilizam *Content Providers* que podem fornecer funções rápidas e sofisticadas para novos aplicativos com as seguintes funções: Browser, Calendar, Contacts, Call log, Media e Settings. Os *Content Providers* são únicos, quando comparados aos mecanismos de IPC (Inter Process Comunication) encontrados em outras plataformas disponíveis no mercado tecnológico.

Como forma de materializar em algumas simples instruções de código, observe o código abaixo que poderia estar presente em algum método de uma Activity qualquer:

```
ContentProviderClient client;
client = getContentResolver().acquireContentProviderClient(“content://contacts/people”);
ContentProvider provider = client.getLocalContentProvider();
```

04

Observe a URI utilizada como parâmetro que segue a seguinte regra geral:

```
content://authoriry/path/id
```

“Authority” representa o pacote Java relativo ao namespace do provedor de conteúdo (muitas vezes o namespace Java da implementação do provedor de conteúdo). “Path” representa o caminho lógico estruturado e “id” o identificador do conteúdo. Abaixo alguns outros exemplos:

```
//faz referência a uma pessoa
content://contacts/people/25
```

```
//faz referências aos números de telefones de uma pessoa com id 25
content://contacts/people/25/phones
```

Quando um programador chama o método query em um provedor de conteúdo, a chamada retorna um objeto do tipo **android.database.Cursor**. Essa interface permite que você recupere um resultado (na forma de uma linha de um banco de dados), por vezes utilizando um índice automaticamente atualizado à medida que você recupera cada resultado. Programadores familiarizados com o JDBC podem fazer uma analogia dessa implementação *Cursor* com a implementação **java.sql.ResultSet**

05

3 - BROADCAST RECEIVER

A classe *Broadcast Receiver* implementa outra variante do mecanismo de alto nível de comunicação IPC (Inter Process Comunication) do Android, utilizando objetos do tipo *Intent*.

Se um aplicativo quiser receber e responder a evento global, como um telefone tocando ou uma mensagem de texto recebida, ele deve ser registrar como *Broadcast Receiver*. Esse registro pode ser feito de duas formas:

De forma ESTÁTICA	De forma DINÂMICA
Na forma estática, o aplicativo pode implementar um elemento XML denominado de <receiver> no arquivo AndroidManifest.xml que descreve o nome da classe que representa um Broadcast Receiver bem como uma enumeração de seus Intent Filters. Lembre-se que o *Intent Filter* descreve o Intent que um aplicativo quer processar. Se o receptor estiver registrado no arquivo AndroidManifest.xml, o aplicativo não precisa estar em execução para ser ativado. Quando o evento ocorre, o aplicativo é iniciado automaticamente na notificação do evento de ativação pelo próprio sistema operacional Android que é responsável por gerenciar esse procedimento.	Na segunda forma considerada **dinâmica**, o aplicativo pode, por meio do registro em tempo de execução, se credenciar através do método *Register Receiver* da classe *Context*.

Assim como os *Services*, os *Broadcast Receivers* não possuem um GUI (Grafical User Interface). Além disso, o código que será executado no método *on Receive* de um *Broadcast Receiver* não deve fazer suposições sobre a duração de operações longas. Caso seja necessário um tempo longo para execução da referida operação, é extremamente recomendado que o código do método *on Receive* inicie uma requisição para um *Service* completar a função requisitada, porque o componente de aplicativo *Service* é projetado para operações de longo prazo, enquanto que o *Broadcast Receiver* é projetado para operações de curtíssimo prazo.

06

4 - Árvore de Diretório (Recursos)

O código fonte de aplicativos Android quase sempre utiliza a seguinte hierarquia de diretórios abaixo:

```
AndroidManifest.xml
res/
        layout/
                        … contém arquivos de layout do aplicativo...
        drawable/
                        … contém imagens, xml desenhável...
        raw/
                        ...contém arquivos de dados que podem ser carregados como fluxo de
                                bytes...
        values/
```

...contém arquivos xml cujo conteúdo são strings e valores númericos
utilizados no desenvolvimento do código fonte...
src/
java/packages/directories

Vale lembrar que essa estrutura pode sofrer alterações à medida que o ambiente do Android vai se desenvolvendo. Portanto, essa visão hierárquica deve ser vista como uma estrutura modelo inicial. O diretório *res/* é chamado de diretório de **recursos**.

Os recursos podem ser qualquer coisa que o aplicativo necessite fazer uso enquanto está executando.

Os aplicativos acessam os recursos nesse diretório utilizando o método *get Resources* da classe *Context,* por meio da classe *R.*

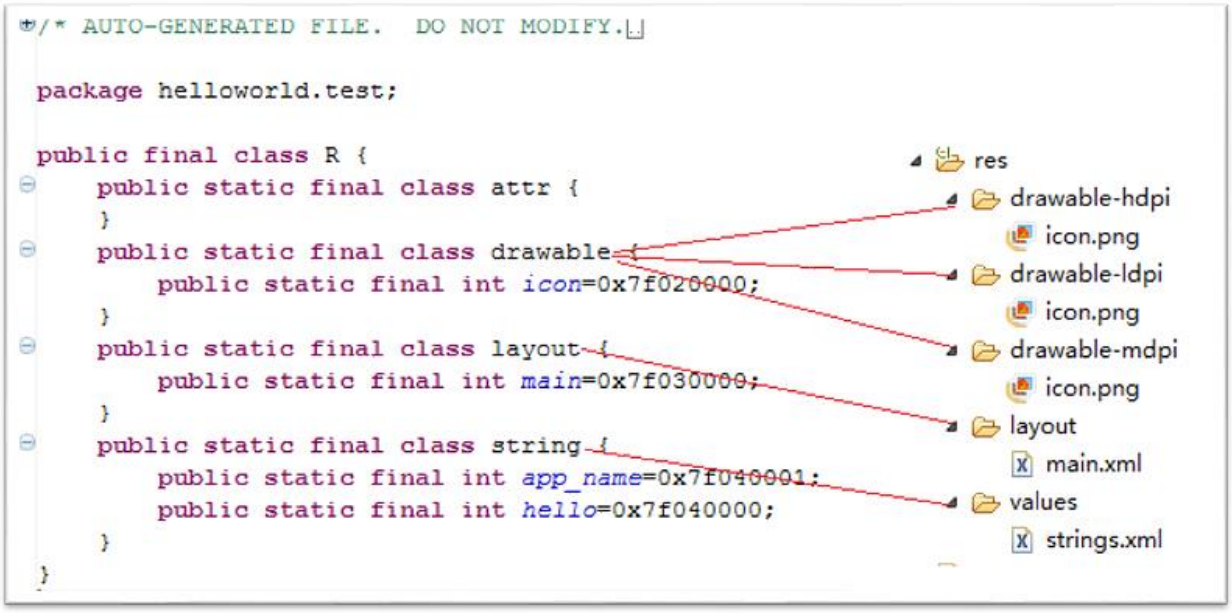
07

Todos os recursos presentes neste diretório são **automaticamente compilados** e disponibilizados para uso pelo aplicativo Android por meio de uma classe cujo nome é ***R***. A classe *R*, que é gerada automaticamente, contém campos que identificam unicamente todos os recursos na estrutura de pacotes do aplicativo.

A classe *R* é composta de classes estáticas internas (uma para cada tipo de recurso) que armazenam referências para todos os seus recursos na forma de um valor *int*. Esse valor é um ponteiro (endereço de memória) constante para um arquivo de objeto, por meio de uma tabela de recursos contida em um arquivo especial criado pela ferramenta *aapt* e utilizado pelo arquivo *R.java*.

É importante perceber que a classe *R* é onipresente dentro do Android, uma vez que a mesma permite fácil acesso a todos os recursos, inclusive a componentes presentes em arquivos de layout da GUI.

A imagem abaixo representa esse conceito:



08

5 - Resumo

O objetivo deste módulo foi apresentar os demais componentes específicos (*Service*, *Content Provider* e *Broadcast Receiver*) do framework de desenvolvimento de aplicativos para o sistema operacional Android. Um Service é utilizado em tarefas de segundo plano que podem estar ativas, mas não visíveis na tela, como, por exemplo, um download. Já um *Content Provider* é a forma que o sistema Android desenvolveu para permitir aos aplicativos o compartilhamento de seus dados. E por último, um *Broadcast Receiver*, que nada mais é que uma forma dos aplicativos responderem a eventos de interesse global, como por exemplo, registrar o áudio de todas as ligações recebidas de um determinado número de telefone.

Além disso, esse módulo apresentou também uma estrutura típica de organização dos recursos (imagens, vídeos, áudios, strings, dentre outros) de um aplicativo Android bem como a forma automatizada que a ferramenta aapt (pertencente ao SDK Android) cria a onipresente classe R.java. A referida classe é essencial no desenvolvimento de aplicativos Android, pois é por meio dela que os programadores passam a ter acesso aos recursos citados anteriormente.

Unidade 3 – Conceitos Básicos de um Aplicativo Android

Módulo 3 – *Callback* e AndroidManifest.xml

01

1 - Callback

Em programação de computadores, ***Callback*** é uma parte de um código executável que é passada como argumento para um outro código, o qual é esperado ser chamado de volta (executado) em algum momento conveniente do tempo.

O momento conveniente do tempo pode ser de imediato ou não. Isso significa que teremos, respectivamente, dois tipos de *Callback*s: síncronos e assíncronos. Ou seja, um *Callback* é um procedimento/função/método que é passado como argumento para outro procedimento/função/método que é invocado após algum evento ou condição.

Existem dois tipos de *Callback*s que diferem na forma como o controle do fluxo de execução é realizado:

- ***blocking Callback*** também conhecido como *Callback* síncrono e
- ***deferred Callback*** também conhecido como *Callback* assíncrono.

Enquanto *blocking Callback* é invocado antes de uma função retornar, deferred *Callback* pode ser invocado durante ou após uma função retornar. Deferred *Callback* são frequentemente utilizadas em contexto de operações de I/O, manipulação de eventos de GUI, interrupções e por diferentes threads em casos de ambientes de múltiplas threads. Devido a sua natureza, *blocking Callback*s não podem trabalhar sem interrupções ou com múltiplas threads, o que significa que *blocking Callback*s não são comumente utilizadas para sincronização ou delegação de trabalho para outra thread.

02

As formas oferecidas por cada linguagem de programação ao suporte de *Callback*s são variadas, ou seja, podem ser implementadas como sub-rotinas, expressões lambdas, blocos de código, ponteiros de função, classes/objetos, dentre outros.

No caso da linguagem Java em suas versões anteriores a versão 8, *Callback*s podem ser simuladas pela passagem de argumento/parâmetros como sendo uma instância (objeto) de classe ou interface. Neste caso, o receptor poderá receber um objeto que possua um ou mais métodos de *Callback*. Isso significa que um objeto pode ser visto como um conjunto de métodos de *Callback*s bem como seus respectivos dados que deverão ser manipulados.

Em termos de padrões de *software* orientado a objeto, podemos dizer que o tipo *deferred Callback* está diretamente associado ao padrão de projeto denominado de **Inversão de Controle** (*Inversion of Control*).

O padrão de desenvolvimento de programas de computadores ***Inversion of Control*** é aquele onde a sequência (controle) de chamadas dos métodos é invertida em relação à programação tradicional, ou seja, ela não é determinada diretamente pelo programador.

Este controle é delegado a uma infraestrutura de *software* muitas vezes chamada de *container* ou a qualquer outro componente que possa tomar controle sobre a execução. Esta é uma característica muito comum a alguns *frameworks* como, por exemplo, JEE e o Android. Inclusive, essa característica foi batizada como **princípio de hollywood** pela engenharia de *software* onde a célebre frase ficou eternizada: "*Don't call us, we'll call you"* ("Não nos chame, nós chamaremos você").

A seguir, veremos um exemplo em Java ilustrando o *Callback* síncrono e o assíncrono.

03

1.1- EventoInteresse.java

```
package br.aiec;

/**
 * Essa interface representa a interface de Callback
 *
 * @author Guilherme Veloso
 *
 */

public interface EventoInteresse {

	public void algoAconteceu(String mecanismo, int valor);
}
```

04

1.2- CallMe.java

```
package br.aiec;

/**
 * Essa classe implementa a interface de Callback
 *
 * @author Guilherme Veloso
 *
 */

public class CallMe implements EventoInteresse{

	@Override
	public void algoAconteceu(String mecanismo, int valor) {
		System.out.println(mecanismo + ": o método foi chamado pelo
notificador de evento apos: " + valor + " milisegundos");
	}
}
```

05

1.3- NotificadorEvento.java

```
package br.aiec;


/**
 * Essa classe representa o mecanimos chamador do Callback nas suas duas formas:
 * síncrona ou assíncrona. O modo de funcionamento dependerá do método a ser chamado pela classe Principalxxx.java,
 * onde xxx deve ser substituido por Sincrono ou Assincrono.
 *
 *  @author Guilherme Veloso
 *
 */

import java.util.concurrent.ExecutorService;
import java.util.concurrent.Executors;

public class NotificadorEvento implements Runnable {

	private String tipoCallback;

	private EventoInteresse evento;

	private boolean ehNumeroPar(int aleatorio) {
		return aleatorio % 2 == 0;
	}

	public NotificadorEvento(EventoInteresse evento) {
		this.evento = evento;
	}

	@Override
	public void run() {

		int contador = 0;

		int aleatorio = 0;

		try {
			do {
				aleatorio = (int) (Math.random() * 3000);

				if (ehNumeroPar(aleatorio)) {
```

```
                                //chamada ao método de Callback toda vez que
o número aleatório for um número PAR
                                evento.algoAconteceu(tipoCallback,
aleatorio);
                          }

                          Thread.sleep(aleatorio);

                    } while (contador++ < 10);

                    System.out.println("Thread secundária (Callback)
finalizou");

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

        /**
         * Método que executa o Callback de modo assíncrono
         */
        public void inicializarEventoAssincrono() {
              tipoCallback = "ASYNC";

              ExecutorService pool = Executors.newFixedThreadPool(1);

              pool.execute(this);

              pool.shutdown();
        }

        /**
         * Método que executa o Callback de modo síncrono
         */
        public void inicializarEventoSincrono() {
              tipoCallback = "SYNC";

              run();
        }

}
```

06

1.4- PrincipalAssincrono.java

```
package br.aiec;

public class PrincipalAssincrono {
	public static void main(String[] args) {

		CallMe call = new CallMe();
		NotificadorEvento notificador = new NotificadorEvento(call);

		notificador.inicializarEventoAssincrono();

		System.out.println("Thread principal finalizou");
	}
}
```

07

1.5- PrincipalSincrono.java

```
package br.aiec;

public class PrincipalSincrono {
	public static void main(String[] args) {

		CallMe call = new CallMe();
		NotificadorEvento notificador = new NotificadorEvento(call);

		notificador.inicializarEventoSincrono();

		System.out.println("Thread principal finalizou");
	}
}
```

Um ponto fundamental que deve ser observado é que após a execução da classe **PrincipalAssincrono.java**, a mensagem "Thread principal finalizou" pode aparecer a qualquer momento, ou seja, antes, durante ou depois das mensagens de *Callback*. No entanto, após a execução da classe PrincipalSincrono.java, a mesma mensagem, "Thread principal finalizou", sempre irá aparecer após a execução das mensagens de *Callback*.

É importante perceber que o *Callback* síncrono não altera o fluxo de controle da thread principal que é iniciada no método main. Já o *Callback* assíncrono, delega o fluxo de controle a uma thread secundária, independente da thread principal, cujo fluxo de controle é executado em um momento diferente do fluxo principal.

08

2 - AndroidManifest.xml

Nos módulos anteriores foram apresentados os elementos comuns de um aplicativo Android. Um fato fundamental do desenvolvimento do Android é que um aplicativo Android contém ao menos uma *Activity*, *Service*, *Broadcast Receiver* ou *Content Provider*. Alguns destes elementos anunciam os *Intents* que estão interessados em processar via o mecanismo *Intent Filter*. Todas essas informações precisam ser unidas para que um aplicativo Android seja executado. O mecanismo "cola" desta tarefa de definir tais relações entre os diversos componentes do Android é feito por meio do arquivo de configuração **AndroidManifest.xml**.

O arquivo de configuração de qualquer aplicativo Android utiliza uma linguagem de marcação chamada **XML** (Extensible Markup Language) que é um bom exemplo de SGML (Standard Generalized Markup Language).

Uma **linguagem de marcação** é um sistema para anotar um documento de uma forma que é sintaticamente distinguível do texto, ou seja, a ideia e a terminologia evoluíram a partir do "marcar" de manuscritos em papel.

Esse destaque, tradicionalmente escrito com um lápis azul em manuscrito dos autores, é frequentemente utilizado pelos editores de texto que fazem a revisão dos documentos dos autores antes de sua publicação e impressão.

SGML (Standard Generalized Markup Language), definida pela ISO 8879:1986, foi desenhada como uma linguagem de marcação genérica para documentos. SGML define dois postulados:

1) A marcação deve ser declarativa, ou seja, deve descrever a estrutura e outros atributos de um documento ao invés de especificar o processamento a ser executado.
2) A marcação deve ser rigorosa de modo que as técnicas disponíveis para o processamento, como programas e banco de dados, possam ser utilizados para a interpretação inequívoca do documento.

09

Como XML é uma linguagem genérica utilizada para diversos fins, existem basicamente duas formas de definir regras para validar um determinado documento que são:

- DTD (Document Type Definition) e
- XML Schema.

Ambas as regras de validação são utilizadas para determinar um conjunto personalizado de elementos, atributos e valores específicos e válidos para os documentos XML. Deste modo, os elementos, atributos e valores utilizados para se escrever o arquivo AndroidManifest.xml são predeterminados pelo

framework do sistema. Utilizar valores inválidos irá gerar um erro de sintaxe na hora de realizar o parser do documento.

Além disso, todo documento XML deve ser bem formado (well formed). Para ser um documento bem formado, devem ser estabelecidas regras sobre a declaração e tratamento dos elementos, atributos e valores. Tags são *case-sensitive* (de preferência em letras minúsculas), com seus respectivos atributos delimitados com aspas. Os elementos vazios têm regras estabelecidas. Elementos sobrepostos, ou seja, que foram abertos e não foram devidamente fechados, invalidam um documento. Idealmente, um documento bem-formado está em conformidade com as metas de design de XML.

Em resumo, podemos dizer que todo documento XML requer que:

- o conteúdo seja definido;
- o conteúdo deve ser delimitado entre as tags de abertura e fechamento;
- o conteúdo deve estar devidamente aninhado, ou seja, pais dentro de raízes e filhos dentro dos pais.

10

O arquivo AndroidManifest.xml existe na raiz do diretório do aplicativo e contém todas as relações projetadas entre um aplicativo específico e os *Intents*. Os arquivos AndroidManifest.xml de cada aplicativo agem como descritores de implementação de aplicativos.

Abaixo, um exemplo genérico de um arquivo AndroidManifest.xml.

```
<?xml version="1.0" encoding="utf-8"?>
<manifest xmlns:android="http://schemas.android.com/apk/res/android"
   package="br.aiec.acessoBD"
   android:versionCode="1"
   android:versionName="1.0" >

   <uses-sdk
      android:minSdkVersion="23"
      android:targetSdkVersion="23" />

   <application
      android:allowBackup="true"
      android:icon="@drawable/ic_launcher"
      android:label="@string/app_name"
      android:theme="@style/AppTheme" >
      <activity
         android:name=".ActivityPrincipal"
         android:label="@string/app_name" >
         <intent-filter>
```

```
            <action android:name="android.intent.action.MAIN" />
            <category android:name="android.intent.category.LAUNCHER" />
        </intent-filter>
    </activity>
  </application>
</manifest>
```

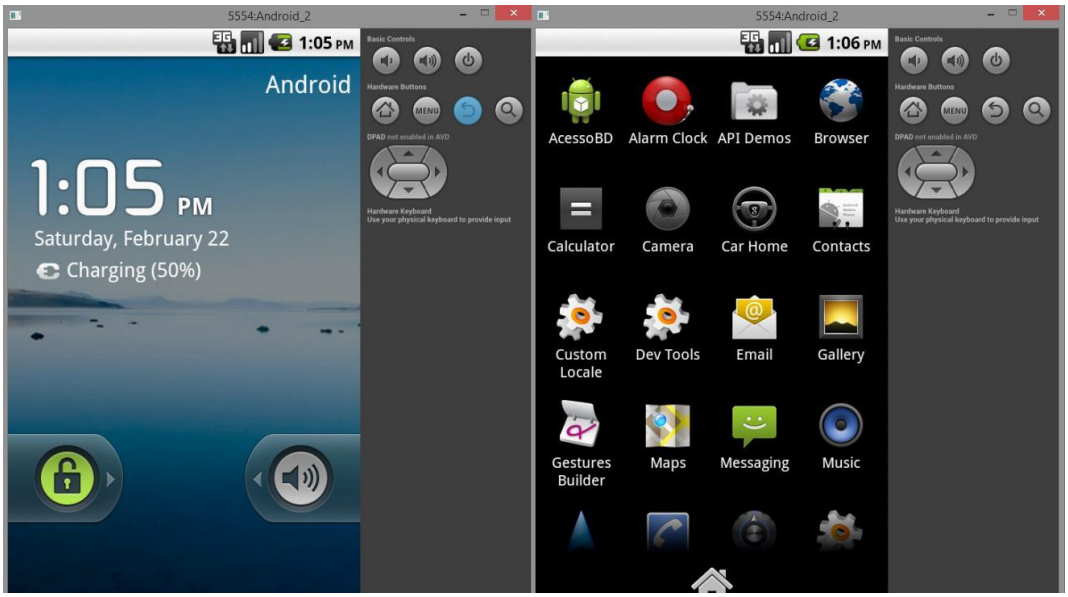
Examinando esse arquivo de configuração é possível perceber que o elemento ***<manifest>*** contém, dentre outros, o nome do pacote (*package*) desse aplicativo. O referido arquivo genérico declara ainda uma única *Activity* cujo nome é *Activity Principal*.

Observe também a sintaxe @string presente. Sempre que o simbolo de @ for utilizado em um arquivo AndroidManifest.xml, ele se refere a informações armazenadas em um dos arquivos de recursos (apresentados anteriormente). Nesse caso especifico, o valor para atributo label é obtido do recurso de string identificado como *app_name,* que neste exemplo possui o valor de *"AcessoBD"*.

O único componente deste aplicativo é a *Activity Principal*. Essa única *Activity* contém uma única definição *Intent Filter*. O *Intent Filter* utilizado aqui é o mais comum visto nos aplicativos Android. A ação *android.intent.action.MAIN* indica que esse é o ponto de entrada do aplicativo.

11

A categoria *android.intent.category.LAUNCHER* coloca essa *Activity* na janela de inicialização, juntamente com os demais aplicativos instalados, conforme a imagem do lado direito presente na figura abaixo:



Um ponto importante é que é possível ter múltiplos elementos *Activities,* visíveis ou não, em um único arquivo de configuração AndroidManifest.xml, ou seja, um único aplicativo pode ter zero ou mais *Activities* visíveis na janela de inicialização. O mais comum é que cada aplicativo possua apenas uma *Activity*, àquela considerada principal, na janela de inicialização.

12

Além dos elementos que foram mostrados no exemplo anterior, outras tags comuns são as seguintes:

- A tag **<service>** representa um *Service*. Os atributos da tag **<service>** incluem sua classe e seu nome. Um **<service>** pode também incluir a tag **<intent-filter>**
- A tag **<receiver>** representa um *Broadcast Receiver* que também pode ter um tag **<intent-filter>** explícita.
- A tag **<uses-permission>** diz ao Android que esse aplicativo requer certos privilégios de segurança. Por exemplo, se um aplicativo requer acesso aos contatos em um dispositivo, é necessário que a tag abaixo esteja presente em seu arquivo AndroidManifest.xml:

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

À medida que seu aplicativo Android for crescendo e se tornando mais complexo, o arquivo AndroidManifest.xml também se tornara maior e com inúmeros elementos presentes que o descreverão.

13

RESUMO

O objetivo deste módulo foi apresentar os conceitos relativos à *Callback*s síncronos (aqueles que são invocados antes de uma função retornar) e assíncronos (aqueles que podem ser invocados durante ou após uma função retornar) bem como suas principais áreas de aplicação. Além disso, foi apresentado o arquivo AndroidManifest.xml que serve como um descritor de implementação de aplicativos contemplando sua estrutura, em termos de seus quatro componentes essenciais (*Activity*, *Broadcast Receiver*, *Service* e *Intent*) e seus relacionamentos com o sistema operacional Android.

UNIDADE 3 – CONCEITOS BÁSICOS DE UM APLICATIVO ANDROID

MÓDULO 4 – SERIALIZAÇÃO E PARCELABLE

01

1 - SERIALIZAÇÃO

Nesta etapa do nosso estudo descreveremos duas funcionalidades utilizadas frequentemente em qualquer sistema computacional, principalmente no Android, que faz uso contínuo de tais conceitos. A compreensão destas funcionalidades é essencial para a programação de aplicativos voltados para o Android.

A **serialização** é a conversão de dados de uma representação rápida, eficiente e interna em algo que possa ser mantido em um armazenamento persistente ou transmitido por uma rede.

A conversão de dados para seu formato serializado é muitas vezes chamado de *marshalização*. Da mesma forma, converter os dados de volta a sua representação ativa, na memória, é chamado de *desserialização* ou *desmarshalização*.

A forma exata pela qual os dados são serializados depende do motivo da serialização. Dados serializados que serão transmitidos pela rede, por exemplo, talvez não tenham de ser legíveis durante a transmissão. Por outro lado, dados serializados para armazenamento em um banco de dados serão muito mais úteis se a representação permitir consultas SQL fáceis de serem construídas e que façam sentido direto para a capacidade cognitiva do ser humano. No primeiro caso, o formato de serialização pode ser **binário** e no segundo é mais provável que tenhamos um **texto rotulado**.

02

O ambiente Android aborda quatro **usos comuns da serialização**, como veremos a seguir.

a) Gerenciamento do ciclo de vida

Diferentemente de dispositivos computacionais maiores, como um laptop ou PC *desktop*, dispositivos Android não podem contar com a possibilidade de transferir um aplicativo para um armazenamento secundário quando o dispositivo estiver inativo. Em vez disso, o *framework* oferece um objeto chamado ***Bundle***.

Quando um aplicativo é suspenso, ele deve descrever seu estado no *Bundle*. Quando o aplicativo é recriado, o *framework* do Android promete fornecer uma cópia do mesmo Bundle durante a inicialização. Deste modo, um aplicativo deve ser capaz de serializar, por meio do *Bundle*, tudo aquilo que quiser manter durante sua suspensão pois esta é uma, dentre várias outras possibilidades metodológicas necessárias para o funcionamento adequado de um aplicativo Android.

b) Persistência

Além do estado imediato do aplicativo, mantido em um *Bundle*, a maioria dos aplicativos gerencia algum tipo de armazenamento persistente de dados, geralmente um banco de dados SQLite encapsulado em um *ContentProvider*.

O Android faz uso do SQLite, nativamente, para persistência dos dados dos diversos aplicativos presentes no ambiente. Neste contexto, aplicativos têm de converter entre a representação interna dos dados de objetos e as representações desses mesmos objetos no banco de dados seguidas vezes. Em sistemas maiores, esse procedimento, conhecido como ORM (Object-Relational Mapping), tem suporte de *framework*s como o *Hibernate*, por exemplo.

No caso do Android, ainda não existe um suporte nativo do *framework* para essas conversões necessárias entre as representações presentes na memória primária e na memória secundária. Deste modo, o conhecimento e uso do SQL de forma direta é requerido para que a persistência possa ser implementada em aplicativos Android. Contudo, projetos externos ao Android, como o ORMLite podem auxiliar também na persistência de dados.

ORMLite
http://ormlite.com/sqlite_java_android_orm.shtml

03

c) Comunicação entre Processos

O *framework* do Android promove uma arquitetura que quebra aplicativos extensos e monolíticos em componentes menores: Activity, Service, *Content Provider* ou *Broadcast Receiver*. Deste modo, quando dois ou mais processos, locais ou remotos, distintos precisam se comunicar, o Android oferece uma ferramenta altamente otimizada denominada de **AIDL** (*Android Interface Definition Language*).

Assim como a maioria de outras IDLs, a AIDL também utiliza o mecanismo de RPC (*Remote Procedure Call*), que é uma tecnologia popular para implementação do modelo cliente-servidor de computação distribuída. Na linguagem Java, o RMI (*Remote Method Invocation*) é uma API que realiza o equivalente para a orientação a objeto, semelhante as especificações RPC. Além disso, a implementação depende da JVM e somente funciona entre JVMs, ou seja, somente permite a comunicação entre objetos Java por meio da implementação/protocolo conhecido como **JRMP** (*Java Remote Method Protocol*). No caso de comunicações entre objetos Java e outros não Java deve-se utilizar o **CORBA** (*Common Object Request Broker Architecture*) ou também conhecido como RMI-IIOP (RMI over IIOP). Saiba+

d) Comunicação em Rede

Essa é a característica que torna dispositivos móveis tão empolgantes. A capacidade de se conectar à internet e de utilizar a imensa variedade de serviços que ela oferece é justamente o principal atrativo do Android.

Aplicativos devem ser capazes de manipular os protocolos impostos por qualquer dispositivo móvel, o que inclui a tradução de informações internas em consultas a esses serviços e a retradução das respostas.

AIDL
AIDL é similar a outras IDLs, ou seja, é utilizada para descrever a interface de componentes de *software* cujo objetivo é permitir a comunicação entre ambos.

RPC
RPC é uma tecnologia de comunicação entre processos que permite a um programa de computador chamar um procedimento em outro espaço de endereçamento de modo que tal procedimento possa ser reutilizado.

Saiba+
Como os detalhes envolvidos nestes protocolos estão além do escopo desta disciplina, caso queira saber mais, procure pelo padrão de projeto Proxy (GOF Estrutural de objeto), pois o mesmo define as características que serviram de base para a implementação do RMI na linguagem de programação Java.

04

1.1- Serialização Java

O Java define um *framework* de serialização por meio da interface *java.io.serializable* e de um par de tipos de serialização que são *java.io.ObjectOutputStream* e *java.io.ObjectInputStream.*

Como a serialização do Java geralmente funciona sem problemas, mesmo programadores experientes podem não conhecer sua complexidade. Saiba+

O Android oferece suporte a serialização Java por meio do tipo *Bundle*. O tipo *Bundle* possui dois métodos, *putSerializable* e *getSerializable*, que, respectivamente, adicionam e recuperam um *Serializable* Java em um *Bundle*.

O código abaixo é um exemplo genérico de uso deste recurso:

```
public class SampleSerialize extends Activity {

	public static final String APP_STATE = "br.aiec";

	private static class AppState implements Serializable {
		// AQUI DEVE SER REPRESENTADO O ESTADO DO APLICATIVO
	}

	private AppState applicationState;

	/**
	 * Esse método é chamado quando a Activity é criada
	 */
	@Override
	protected void onCreate(Bundle savedInstanceState) {
```

```
            super.onCreate(savedInstanceState);

            if(savedInstanceState == null){
                  applicationState = new AppState();

            }else{
                   applicationState = (AppState)


      savedInstanceState.getSerializable(APP_STATE);
            }

            //RESTANTE DO CODIGO.....
      }

      /**
       * Esse método é chamado pelo Android quando a Activity é pausada.
       */
      @Override
      protected void onSaveInstanceState(Bundle outState) {
            super.onSaveInstanceState(outState);
            outState.putSerializable(APP_STATE, applicationState);
      }
}
```

Nesse exemplo genérico, o aplicativo mantém algumas informações globais de estado como um objeto serializable. Quando a *Activity* do exemplo que está sendo representada pela classe SampleSerialize, é pausada, abrindo espaço na memória para outra *Activity*, o *framework* do Android invoca o método de Callback *onSaveInstanceState* de *SampleSerialize* para salvar o estado do objeto no *Bundle*. Quando *SampleSerialize* é executada novamente, o método *onCreate* recupera de volta o estado gravado no *Bundle*.

Esse comportamento se deve ao ciclo de vida de uma Activity que será explicado em todos os seus detalhes na unidade seguinte.

Saiba+
Como uma descrição completa sobre esse assunto está além do escopo desta disciplina, aqueles que quiserem aprofundar o estudo podem estudar pelo livro *Effective Java* do autor *Joshua Bloch.* Neste livro, o autor dedica algo em torno de 10% do seu conteúdo a forma correta de como utilizar a serialização em Java.

05

2 - PARCELABLE

Ainda que o *framework* do Android ofereça suporte à serialização Java, ela não costuma ser a melhor escolha para serialização do estado imediato de um programa. O próprio protocolo interno de serialização do Android (*Parcelable*) é leve, altamente otimizado e possui apenas alguns detalhes obrigatórios que para o tipo *Serializable* do Java são considerados opcionais, mesmo que muitos não consigam perceber a ilusão e o perigo destas "flexibilidades" do *Serializable*. Ou seja, muitos acreditam que implementar o *serializable* do Java é algo simples e prático uma vez que a linguagem permite o uso trivial de tal funcionalidade. Contudo, os custos envolvidos nesse procedimento são altos e caros se utilizado de modo impulsivo e precoce.

O uso adequado necessita de um projeto detalhado sobre as inúmeras situações em que tal aplicativo irá se apresentar. Portanto, o uso do *Parcelable* é a melhor escolha para a comunicação local entre processos utilizando-se o *framework* Android.

Um objeto deve atender a três **requisitos** para que possa ser serializado por meio do *Parcelable*:

1. Ele deve implementar a interface *Parcelable*;

2. Ele deve ter um serializador (uma implementação da operação da interface *writeToParcel*);

3. Ele deve ter um desserializador (uma variável final, estática e pública de nome CREATOR, que contenha uma referência a uma implementação de *Parcelable.Creator*).

06

A operação da interface *writeToParcel* é o serializador. Ele é chamado em um objeto quando é necessário serializar esse mesmo objeto em um *Parcel*. O trabalho do serializador é escrever tudo que é necessário para reconstruir o estado do objeto ao *Parcel* transmitido. Isso significa expressar o estado do objeto em termos de, basicamente, cinco tipos de dados primitivos do Java:

- byte,
- double,
- int,
- float,
- long.

Além desses, o tipo abstrato String também pode ser utilizado.

```
public class SimpleParcelable implements Parcelable {

	public enum State {
		BEGIN, MIDDLE, END
	};

	private static final Map<State, String> marshalState;

	static {
		Map<State, String> m = new HashMap<State, String>();
		m.put(State.BEGIN, "begin");
		m.put(State.MIDDLE, "middle");
		m.put(State.END, "end");

		marshalState = Collections.unmodifiableMap(m);
	}

	private State state;
	private Date date;

	private long dateToLong() {
		return (date == null) ? -1 : date.getTime();
	}

	private String stateToString() {
		return (state == null) ? "" : marshalState.get(state);
	}

	@Override
	public void writeToParcel(Parcel dest, int flags) {
		// traduz um date para um long
		dest.writeLong(dateToLong());

		// traduz o state para String
		dest.writeString(stateToString());
	}

	public State getState() {
		return state;
	}

	public void setState(State state) {
		this.state = state;
	}

	public Date getDate() {
		return date;
	}
```

```
	public void setDate(Date date) {
		this.date = date;
	}
}
```

Evidentemente, a implementação exata de *writeToParcel* dependerá do conteúdo do objeto sendo serializado. Nesse caso, o objeto *SimpleParcelable* tem duas informações de estado e escreve ambas no *Parcel* transmitido.

A escolha de uma representação para a maioria dos tipos simples de dados não exigirá nada além de um pouco de engenhosidade. O date, neste exemplo, é facilmente representado pelo tempo transcorrido. Não se esqueça, entretanto, de considerar alterações futuras nos dados, quando estiver escolhendo a representação serializada.

Certamente teria sido mais fácil, nesse exemplo, representar *state* como um *int* cujo valor poderia ser obtido chamando-se *state.ordinal()*. Ao fazê-lo, entretanto, tornaríamos muito mais difícil a compatibilidade futura do objeto. Suponha que se torne necessário em algum momento, adicionar um novo estado, *State.INIT*, antes de *State.BEGIN*. Essa alteração trivial faria com que novas versões do objeto se tornassem completamente incompatíveis com as versões anteriores.

07

Um argumento semelhante, mesmo que um pouco mais fraco, se aplica à utilização de *state.toString()* para criar uma representação serializada de do tipo *State*. O mapeamento entre um objeto e sua representação em um *Parcel* faz parte do procedimento específico de serialização. Não se trata de um atributo inerente ao objeto. É inteiramente possível que certo objeto tenha representações completamente diferentes quando serializado por serializadores distintos. Para ilustrar esse princípio – ainda que provavelmente estejamos exagerando, pois o tipo State é localmente definido-, o mapa utilizado para serializar state é um membro independente e explicitamente definido apenas para a serialização.

SimpleParcelable, como mostramos antes, compila sem nenhum erro. Ele poderia até mesmo ser serializado como um pacote. Até aqui, entretanto, não temos como pegá-lo de volta. Para isso necessitamos de um desserializador conforme demonstrado pelo restante do código abaixo:

```
public class SimpleParcelable implements Parcelable {

	//CODIGO OMITIDO....

	private static final Map<String, State> unmarshalState;

	static {
		Map<String, State> m = new HashMap<String, State>();
		m.put("begin", State.BEGIN);
```

```
            m.put("middle", State.MIDDLE);
            m.put("end", State.END);

            unmarshalState = Collections.unmodifiableMap(m);
      }

      private State state;
      private Date date;

      public static final Parcelable.Creator<SimpleParcelable> CREATOR =
new

      Parcelable.Creator<SimpleParcelable>() {
            public SimpleParcelable createFromParcel(Parcel in) {
                  return new SimpleParcelable(in.readLong(),
in.readString());
            }

            public SimpleParcelable[] newArray(int size) {
                  return new SimpleParcelable[size];
            }
      };

      public SimpleParcelable(long date, String state) {
            if (date >= 0) {
                  this.date = new Date(date);
            }

            if (state != null && state.length() > 0) {
                  this.state = unmarshalState.get(state);
            }
      }

      //CODIGO OMITIDO....
}
```

O trecho de código acima mostra apenas o desserializador recém adicionado, o campo final, estático e público chamado CREATOR. O campo é uma referência a uma implementação de Parcelable.Creator<T>, em que “T” é o tipo do objeto empacotável que será desserializado, nesse caso específico, um SimpleParcelable. É importante que isso seja feito de forma correta! Se CREATOR for protegido em vez de público, se não for estático, ou se estiver escrito “Creator”, o *framework* do Android não poderá desserializar o objeto em questão.

A implementação de Parcelable.Creator<T> é um objeto com um único método, *createFromParcel*, que desserializa uma única instância de *Parcel*. Isso deve ser feito de forma idiomática, ou seja, lendo cada seção de dados do Parcel exatamente na mesma ordem em que eles foram escritos em *writeToParcel*.

Essa ordem de leitura é essencial e fundamental para que o procedimento de serializar/desserializar seja correto. No caso do código genérico de exemplo representado pela classe SimpleParcelable, primeiro foi escrito (*writeToParcel*) um long que representa a data e segundo foi escrito uma string que representa o state. Desta forma, a desserialização (CREATOR) deve ler, primeiro, um long e, depois, uma string, nesta ordem.

Não respeitar a ordem é um erro grave de programação. Uma vez lido corretamente os dados, chama-se o construtor da classe passando como parâmetro os estado recuperado/desserializado. Como o construtor de desserialização é chamado a partir do escopo da própria classe, ele pode ser protegido em nível de pacote ou até mesmo privado.

08

3 - Classes que oferecem suporte à serialização

A API Parcel não está limitada aos seis tipos mencionados na seção anterior. A documentação do Android oferece uma lista completa de tipos empacotáveis, mas é interessante pensar neles como divididos em quatro grupos.

- O primeiro grupo, dos tipos simples, consiste em null, além dos tipos primitivos (int, float etc....) bem como seus tipos envelopadores – classes - (Integer, Float etc....)
- O segundo grupo consiste em objetos que implementam *Serializable* ou *Parcelable*. Eles não são objetos simples, mas sabem serializar a si mesmos, pois foram programados corretamente para isso.
- O terceiro refere-se aos tipos de coleções que são arrays, listas, mapas, bundles, dentre outros.
- E por fim, o quarto grupo em que temos alguns casos especiais: *CharSequence* e objetos ativos (IBinder).

Ainda que todos esses tipos possam ser serializados em um *Parcel*, há dois tipos que, se possível, devem ser evitados: ***Serializable*** e ***Map***. Vejamos a seguir as explicações.

09

Como mencionamos antes, o Android oferece suporte à serialização nativa do Java. Sua implementação não chega a ser tão eficiente quanto a do tipo *Parcelable*.

Implementar ***Serializable*** em um objeto não é uma forma eficiente, para o *framework* Android, de torná-lo empacotável. Em vez disso, objetos devem implementar *Parcelable* conforme demonstrado anteriormente.

Isso pode ser uma tarefa tediosa se a hierarquia de objetos for complexa, mas os ganhos em desempenho justificam a sua escolha.

O outro tipo que devemos evitar é o *Map*. *Parcel* não oferecer suporte a mapas em geral, apenas aqueles com chaves que sejam strings.

O tipo *Bundle*, específico do Android, fornece a mesma funcionalidade além de apresentar segurança de tipo. Objetos são adicionados a um *Bundle* com métodos como *putDouble*, *putString*, dentre outros, sendo que existe um método apropriado para cada tipo empacotável.

Métodos correspondentes como *getDouble*, *getString*, dentre outros, pegam os objetos de volta. Portanto, um *Bundle* é como um mapa, exceto pelo fato de que pode armazenar tipos diferentes de objetos para chaves distintas, implementando perfeitamente a segurança de tipos. Utilizando um *Bundle*, você elimina uma série de erros de difícil localização que podem ocorrer quando, por exemplo, um *float* serializado é interpretado erroneamente como um *int*.

10

4 - Serialização e o ciclo de vida dos aplicativos

Como mencionado antes, um aplicativo Android pode não contar com acesso à memória virtual. Em um pequeno dispositivo, não há armazenamento secundário para o qual possa ser enviado um aplicativo executado, mas que esteja em segundo plano, quando quisermos liberar espaço para um novo aplicativo visível. Ainda assim, uma boa experiência de usuário requer que, quando o usuário retorne a um aplicativo que foi, por exemplo, suspenso, este tenha a mesma aparência de quando foi visto pela última vez. A responsabilidade pela preservação do estado, mesmo quando ocorrem suspensões, é do próprio aplicativo. Felizmente, o *framework* do Android faz com que isso não seja tão complicado.

O exemplo em “Serialização Java” mostrou o mecanismo geral do *framework* que permite a um aplicativo preservar seu estado mesmo quando ocorrem suspensões.

Sempre que um aplicativo for removido da memória, seu método *onSaveInstanceState* será chamado com um *Bundle*, no qual ele poderá gravar informações de estado necessárias. Quando o aplicativo for reiniciado, o *framework* transmitirá o mesmo *Bundle* ao método *onCreate*, de modo que o aplicativo possa restaurar o seu estado. Armazenando dados de conteúdo de modo inteligente em um *ContentProvider*, e salvando estados leves (por exemplo, a página atualmente visível) com o *onSaveInstanceState*, um aplicativo pode ser reiniciado sem nenhuma interrupção.

O *framework* fornece ainda mais uma ferramenta para preservação do estado do aplicativo. A classe *View* – base para tudo que é visível na tela – tem um método de *callback onSaveInstanceState*, chamado como parte do procedimento de remoção de um aplicativo da memória. Na verdade, ele é chamado a partir de *Activity onSaveInstanceState*, o que explica por que a implementação desse método por seu aplicativo (sobrescrita) deve sempre chamar *super.onSaveInstanceState*. Esse método permite a preservação do estado no nível mais específico possível, como por exemplo, em um aplicativo de e-mail onde se deseja preservar a localização exata do cursor dentro de uma mensagem de texto de e-mail não ainda enviada.

11

5 - RESUMO

O objetivo deste módulo foi apresentar o modo como o Android consegue preservar (gravar e ler) o estado de um aplicativo por meio da serialização que, no caso do sistema Android, pode ser implementada por duas formas: Serializable ou Parcelable. A serialização é a conversão de dados de uma representação rápida, eficiente e interna em algo que possa ser mantido em um armazenamento persistente ou transmitido por uma rede. O ambiente Android aborda quatro usos comuns da serialização: Gerenciamento do ciclo de vida, Persistência, Comunicação entre Processos e Comunicação em Rede.

Um objeto deve atender a três requisitos para que possa ser serializado por meio do *Parcelable*: ele deve implementar a interface Parcelable; ele deve ter um serializador (uma implementação da operação da interface *writeToParcel*); ele deve ter um desserializador (uma variável final, estática e pública de nome CREATOR, que contenha uma referência a uma implementação de Parcelable.Creator).

Não se esqueça que o Parcelable, forma nativa do Android, é a forma preferida, basicamente, por motivos de possuir um melhor desempenho.