# Introdução a atividades

bookmark_border

A classe Activity é um componente crucial de um app para Android, e a maneira como as atividades são lançadas e reunidas é uma parte fundamental do modelo de aplicativo da plataforma. Diferentemente dos paradigmas de programação em que os apps são lançados com um método `main()`, o sistema Android inicia o código em uma instância Activity invocando métodos de callback que correspondem a estágios específicos do ciclo de vida.

Este documento apresenta o conceito de atividades e fornece uma orientação simplificada sobre como trabalhar com elas. Para mais informações sobre as práticas recomendadas para a arquitetura do seu app, consulte o Guia para a arquitetura do app.

## O conceito de atividades

A experiência em apps para dispositivos móveis é diferente da versão para computador, porque a interação do usuário com o app nem sempre começa no mesmo lugar. Em vez disso, a jornada do usuário geralmente começa de forma não determinista. Por exemplo, se você abrir um app de e-mails na tela inicial, poderá ver uma lista de e-mails. Por outro lado, se você estiver usando um app de mídia social que inicialize seu app de e-mails, poderá ir diretamente para a tela do app de e-mails para escrever uma mensagem.

A classe Activity foi desenvolvida para facilitar esse paradigma. Quando um app invoca outro, o app de chamada invoca uma atividade no outro, em vez do app como um todo. Dessa forma, a atividade serve como ponto de entrada para a interação de um app com o usuário. Você implementa uma atividade como uma subclasse da classe Activity.

Uma atividade fornece a janela na qual o app desenha a própria IU. Essa janela normalmente preenche a tela, mas pode ser menor do que a tela e flutuar sobre outras janelas. Geralmente, uma atividade implementa uma tela em um app. Por exemplo, uma das atividades de um app pode implementar uma tela *Preferências*, enquanto outra atividade implementa uma tela *Selecionar foto*.

A maioria dos apps contém várias telas, o que significa que elas abrangem várias atividades. Normalmente, uma atividade em um app é especificada como a *atividade principal*, que é a primeira tela a ser exibida quando o usuário inicia o app. Cada atividade pode iniciar outra para realizar ações diferentes. Por exemplo, a atividade principal em um app de e-mails simples pode fornecer a tela que mostra uma caixa de entrada de e-mail. A partir daí, a atividade principal pode iniciar outras atividades que fornecem telas para tarefas como escrever e-mails e abrir e-mails individuais.

Embora as atividades funcionem juntas para formar uma experiência do usuário coesa em um app, cada uma é vagamente vinculada às outras. Geralmente, há dependências mínimas entre as atividades em um app. Na verdade, com frequência elas iniciam atividades pertencentes a outros apps. Por exemplo, um app de navegação pode iniciar a atividade "Compartilhar" de um app de mídia social.

Para usar as atividades no seu app, é necessário registrar informações sobre elas no manifesto e você precisa gerenciar os ciclos de vida da atividade adequadamente. O restante deste documento introduz esses assuntos.

# Como configurar o manifesto

Para que seu app possa usar as atividades, você precisa declará-las junto a alguns dos atributos delas no manifesto.

## Declarar atividades

Para declarar sua atividade, abra o arquivo de manifesto e adicione um elemento <activity> como filho do elemento <application>. Exemplo:

```
<manifest ... >
  <application ... >
    <activity android:name=".ExampleActivity" />
    ...
  </application ... >
  ...
</manifest >
```

O único atributo obrigatório para esse elemento é android:name, que especifica o nome da classe da atividade. Você também pode adicionar atributos que definem características de atividade, como etiqueta, ícone ou tema da IU. Para mais informações sobre esses e outros atributos, consulte a documentação de referência do elemento <activity>.

**Observação:** depois de publicar o app, não mude os nomes das atividades. Se fizer isso, você poderá corromper alguma funcionalidade, como atalhos de apps. Para mais informações sobre as mudanças a serem evitadas após a publicação, consulte Itens que não podem ser mudados (em inglês).

## Declarar filtros de intent

Filtros de intent são um recurso muito poderoso da plataforma Android. Eles oferecem a capacidade de iniciar uma atividade com base não apenas em uma solicitação *explícita*, mas também *implícita*. Por exemplo, uma solicitação explícita pode dizer ao sistema para "Iniciar a atividade de envio de e-mail no app Gmail". Por outro lado, uma solicitação implícita diz ao sistema para

"Iniciar uma tela de envio de e-mail em qualquer atividade que possa fazer o trabalho". Quando a IU do sistema pergunta a um usuário qual app usar na execução de uma tarefa, um filtro de intent está em ação.

Você pode aproveitar esse recurso declarando um atributo <intent-filter> no elemento <activity>. A definição desse elemento inclui um elemento <action> e, opcionalmente, um <category> e/ou um <data>. Esses elementos são combinados para especificar o tipo de intent ao qual sua atividade pode responder. Por exemplo, o seguinte snippet de código mostra como configurar uma atividade que envia dados de texto e recebe solicitações de outras atividades para fazer isso:

```
<activity android:name=".ExampleActivity" android:icon="@drawable/app_icon">
    <intent-filter>
        <action android:name="android.intent.action.SEND" />
        <category android:name="android.intent.category.DEFAULT" />
        <data android:mimeType="text/plain" />
    </intent-filter>
</activity>
```

Nesse exemplo, o elemento <action> especifica que essa atividade envia dados. Declarar o elemento <category> como DEFAULT permite que a atividade receba solicitações de inicialização. O elemento <data> especifica o tipo de dado que essa atividade pode enviar. O snippet de código a seguir mostra como chamar a atividade descrita acima:

Kotlin

```
val sendIntent = Intent().apply {
    action = Intent.ACTION_SEND
    type = "text/plain"
    putExtra(Intent.EXTRA_TEXT, textMessage)
}
startActivity(sendIntent)
```

Se você pretende que seu app seja autossuficiente e não permita que outros apps ativem as atividades dele, você não precisa de outros filtros de intent. As atividades que você não quiser disponibilizar para outros apps não precisam ter filtros de intent, e você pode iniciá-las usando intents explícitos. Para mais informações sobre como suas atividades podem responder aos intents, consulte Intents e filtros de intents.

## Declarar permissões

Você pode usar a tag <activity> do manifesto para controlar quais apps podem iniciar uma atividade específica. Uma atividade mãe não pode iniciar uma atividade filha a menos que as duas tenham as mesmas permissões no manifesto. Se você declarar um elemento <uses-permission> para uma atividade

mãe, cada atividade filha precisará ter um elemento <uses-permission> correspondente.

Por exemplo, se seu app quiser usar um app hipotético chamado SocialApp para compartilhar uma postagem em mídias sociais, o próprio SocialApp precisa definir a permissão que o app autor da chamada precisa ter:

```
<manifest>
<activity android:name="...."
  android:permission=”com.google.socialapp.permission.SHARE_POST”

/>
```

Em seguida, para poder chamar o SocialApp, seu app precisa corresponder à permissão definida no manifesto do SocialApp:

```
<manifest>
  <uses-permission android:name="com.google.socialapp.permission.SHARE_POST" />
</manifest>
```

Para mais informações sobre permissões e segurança em geral, consulte Segurança e permissões.

# Como gerenciar o ciclo de vida da atividade

Ao longo da vida útil de uma atividade, ela passa por vários estados. Uma série de callbacks são usados para lidar com transições entre estados. As seções a seguir apresentam esses callbacks.

## onCreate()

Você precisa implementar esse callback, que é acionado quando o sistema cria sua atividade. A implementação inicializará os componentes essenciais da atividade. Por exemplo, aqui, o app criará visualizações e vinculará dados a listas. Mais importante, é nesse local que você precisa chamar setContentView() para definir o layout da interface do usuário da atividade.

Quando onCreate() termina, o próximo callback sempre é onStart().

## onStart()

Quando onCreate() sai, a atividade entra no estado "Iniciado" e se torna visível para o usuário. Esse callback contém o que equivale aos preparativos finais da atividade para ir para o primeiro plano e se tornar interativa.

## onResume()

O sistema invoca esse callback imediatamente antes de a atividade começar a interagir com o usuário. Neste ponto, a atividade fica na parte superior da pilha de atividades e captura toda a entrada do usuário. A maior parte da funcionalidade principal de um app é implementada no método onResume().

O callback onPause() sempre segue onResume().

## onPause()

O sistema chama onPause() quando a atividade perde o foco e entra em um estado "Pausado". Esse estado ocorre quando, por exemplo, o usuário toca no botão "Voltar" ou "Recentes". Quando o sistema chama onPause() para sua atividade, isso significa, tecnicamente, que ela ainda está parcialmente visível. Porém, na maioria das vezes, é uma indicação de que o usuário está deixando a atividade e que logo ela entrará no estado "Interrompido" ou "Retomado".

Uma atividade no estado "Pausado" pode continuar atualizando a IU se o usuário estiver esperando por isso. Exemplos de uma dessas atividades incluem a exibição da tela de um mapa de navegação ou de um player de mídia sendo reproduzido. Mesmo que essas atividades percam o foco, o usuário espera que a IU continue sendo atualizada.

**Não** use onPause() para salvar dados do app ou do usuário, fazer chamadas de rede ou executar transações de banco de dados. Para mais informações sobre como salvar dados, consulte Como salvar e restaurar o estado da atividade.

Quando a execução de onPause() termina, o próximo callback é onStop() ou onResume(), dependendo do que acontece depois que a atividade entra no estado "Pausado".

## onStop()

O sistema chama onStop() quando a atividade não está mais visível para o usuário. Isso pode acontecer porque a atividade está sendo destruída, uma nova atividade está sendo iniciada ou uma atividade existente está entrando em um estado "Retomado" e está cobrindo a atividade interrompida. Em todos esses casos, a atividade interrompida não fica mais visível.

O próximo callback que o sistema chamar será onRestart(), se a atividade voltar a interagir com o usuário, ou onDestroy(), se essa atividade for completamente encerrada.

## onRestart()

O sistema invoca esse callback quando uma atividade no estado "Interrompido" está prestes a ser reiniciada. onRestart() restaura o estado da atividade a partir do momento em que ela foi interrompida.

Esse callback é sempre seguido por onStart().

### onDestroy()

O sistema invoca esse callback antes de uma atividade ser destruída.

Esse é o último callback que a atividade recebe. onDestroy() normalmente é implementado para garantir que todos os recursos de uma atividade sejam liberados quando ela (ou o processo que a contém) for destruída.

Esta seção é apenas uma introdução a esse tema. Para ver um tratamento mais detalhado do ciclo de vida da atividade e dos callbacks dele, consulte Ciclo de vida da atividade.

# Entenda o ciclo de vida da atividade

À medida que o usuário navega no aplicativo, sai dele e retorna a ele, as instâncias Activity no aplicativo transitam entre diferentes estados no ciclo de vida. A classe Activity fornece uma quantidade de callbacks que permite que a atividade saiba sobre a mudança do estado: informa a respeito da criação, interrupção ou retomada de uma atividade ou da destruição do processo em que ela reside por parte do sistema.

Dentro dos métodos de callback do ciclo de vida, você pode declarar como a atividade deve se comportar quando o usuário sai e retorna dela. Por exemplo, se estiver construindo um reprodutor de vídeos de transmissão em sequência, você pode pausar o vídeo e encerrar a conexão da rede quando o usuário alternar para outro aplicativo. Quando o usuário retornar, será possível reconectar a rede e permitir que ele reinicie o vídeo de onde parou. Ou seja, cada callback permite que você realize o trabalho específico adequado a determinada mudança de estado. Fazer o trabalho certo no momento apropriado e gerenciar as transições da maneira correta faz com que seu aplicativo seja mais robusto e tenha melhor desempenho. Por exemplo, uma boa implementação dos callbacks de ciclo de vida pode ajudar a garantir que seu aplicativo evite os problemas a seguir:

- Falhas se o usuário receber uma chamada telefônica ou mudar para outro aplicativo enquanto estiver usando seu aplicativo.
- Consumo de recursos importantes do sistema quando o usuário não estiver usando ativamente o aplicativo.
- Perda do progresso do usuário se ele sair do aplicativo e retornar mais tarde.
- Falhas ou perda do progresso do usuário quando a orientação da tela mudar entre paisagem e retrato.

Este documento explica detalhadamente o ciclo de vida da atividade. O documento começa descrevendo o paradigma do ciclo de vida. Em seguida, cada um dos callbacks é explicado: o que acontece internamente quando eles operam e o que você deve implementar durante a execução deles. Depois é apresentada a relação entre o estado da atividade e a vulnerabilidade de um processo que está sendo eliminado pelo sistema. Por fim, vários tópicos relacionados às transições entre os estados de atividade são discutidos.

Para informações sobre como gerenciar ciclos de vida, inclusive orientação sobre práticas recomendadas, veja Como gerenciar ciclos de vida com componentes que os reconhecem e Como salvar estados da IU. Para saber como arquitetar um aplicativo robusto e de qualidade de produção usando atividades em combinação com os componentes da arquitetura, consulte Guia para a arquitetura do app.

## Conceitos do ciclo de vida da atividade

Para navegar entre as fases do ciclo de vida da atividade, a classe "Activity" fornece um conjunto principal de seis callbacks: onCreate(), onStart(), onResume(), onPause(), onStop() e onDestroy(). Conforme a atividade entra em um novo estado, o sistema invoca cada um desses callbacks.

A imagem 1 demonstra a representação visual desse paradigma.

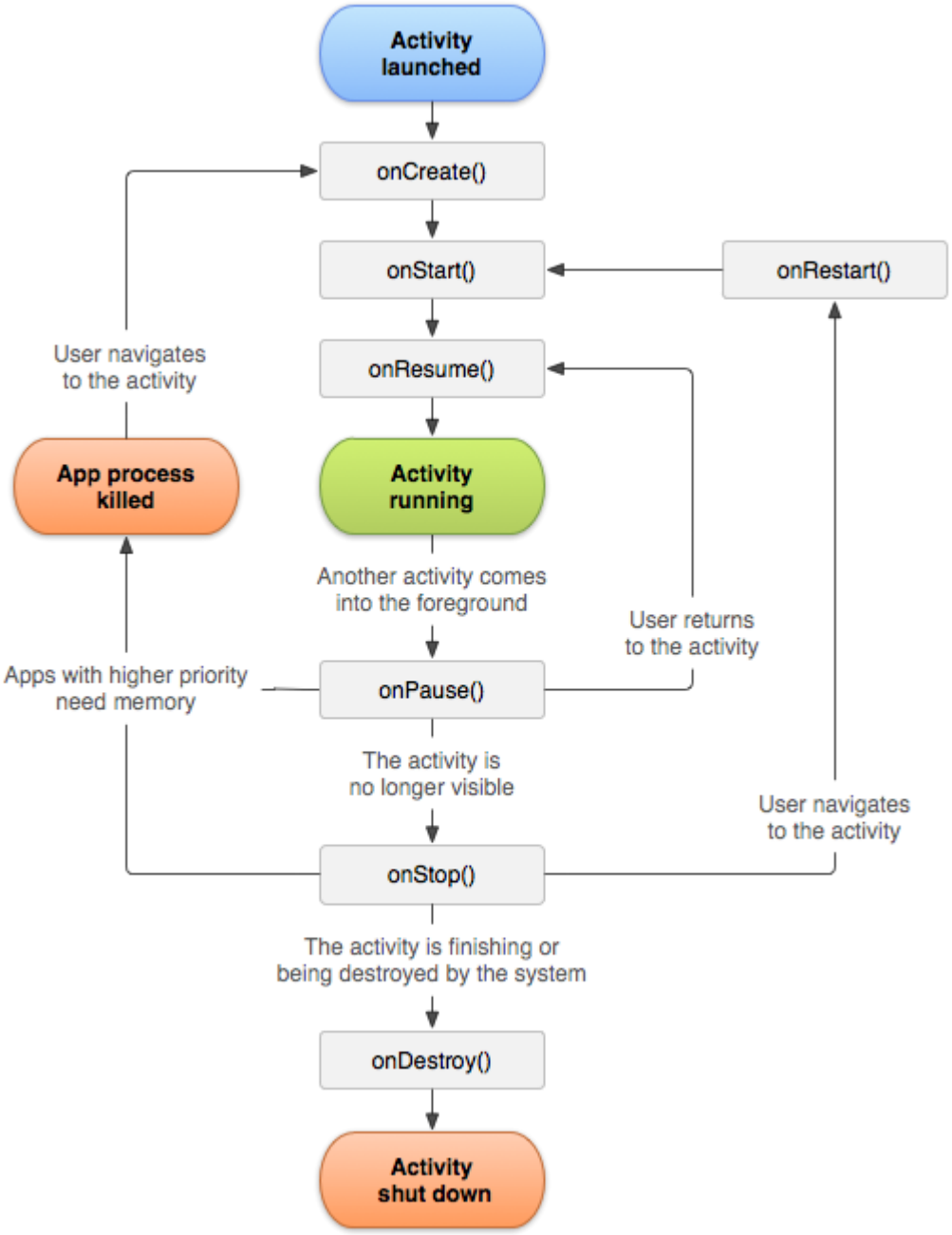


**Figura 1.** Ilustração simplificada do ciclo de vida da atividade.

À medida que o usuário começa a sair da atividade, o sistema chama métodos para eliminá-la. Em alguns casos, essa eliminação é somente parcial. A atividade ainda reside na memória, como quando o usuário alterna para outro aplicativo, e ainda pode voltar ao primeiro plano. Se o usuário retornar a essa atividade, a atividade será retomada de onde o usuário parou. Com algumas exceções, os aplicativos são impedidos de iniciar atividades quando executados em segundo plano.

A probabilidade do sistema eliminar um determinado processo, com as atividades nele, depende do estado da atividade no momento. Em Estado da atividade e ejeção da memória, há mais informações sobre o relacionamento entre o estado e a vulnerabilidade para ejeção.

Dependendo da complexidade de sua atividade, não é necessário implementar todos os métodos do ciclo de vida. No entanto, é importante compreender cada um deles e implementar somente os que garantem que o aplicativo tenha o desempenho esperado pelo usuário.

A próxima seção deste documento fornece detalhes sobre os callbacks usados para processar as transições entre os estados.

# Callbacks do ciclo de vida

Esta seção fornece informações conceituais e de implementação sobre os métodos de callback usados durante o ciclo de vida da atividade.

Algumas ações, como chamar setContentView(), fazem parte dos métodos do ciclo de vida da atividade em si. No entanto, o código que implementa as ações de um componente dependente deve ser colocado no próprio componente. Para fazer isso, você precisa tornar o componente dependente ciente do ciclo de vida. Veja Como gerenciar ciclos de vida com componentes que os reconhecem para saber como tornar seus componentes dependentes cientes do ciclo de vida.

### onCreate()

Esse callback precisa ser implementado. Ele é acionado assim que o sistema cria a atividade. Quando a atividade é criada, ela insere o estado *Criado*. No método onCreate(), você executa a lógica básica de inicialização do aplicativo. Isso deve acontecer somente uma vez durante todo o período que a atividade durar. Por exemplo, sua implementação de onCreate() pode vincular dados a listas, associar a atividade a um ViewModel e instanciar algumas variáveis com escopo de classe. Esse método recebe o parâmetro savedInstanceState, um objeto Bundle que contém o estado anteriormente salvo da atividade. Se a atividade nunca existiu, o valor do objeto Bundle será nulo.

Caso você tenha um componente ciente do ciclo de vida conectado ao ciclo de vida da sua atividade, ele receberá o evento ON_CREATE. O método anotado com @OnLifecycleEvent será chamado para que seu componente ciente do ciclo de vida possa executar qualquer código de configuração necessário para o estado criado.

O seguinte exemplo do método onCreate() mostra uma configuração fundamental para a atividade, como declarar a interface do usuário (definida em um arquivo de layout XML), definir variáveis de associação e configurar parte da IU. Neste exemplo, o arquivo de layout XML é especificado passando o código de recurso R.layout.main_activity do arquivo para setContentView().

Kotlin

```
lateinit var textView: TextView
```

```
// some transient state for the activity instance
var gameState: String? = null

override fun onCreate(savedInstanceState: Bundle?) {
   // call the super class onCreate to complete the creation of activity like
   // the view hierarchy
   super.onCreate(savedInstanceState)

   // recovering the instance state
   gameState = savedInstanceState?.getString(GAME_STATE_KEY)

   // set the user interface layout for this activity
   // the layout file is defined in the project res/layout/main_activity.xml file
   setContentView(R.layout.main_activity)

   // initialize member TextView so we can manipulate it later
   textView = findViewById(R.id.text_view)
}

// This callback is called only when there is a saved instance that is previously saved by using
// onSaveInstanceState(). We restore some state in onCreate(), while we can optionally
restore
// other state here, possibly usable after onStart() has completed.
// The savedInstanceState Bundle is same as the one used in onCreate().
override fun onRestoreInstanceState(savedInstanceState: Bundle?) {
   textView.text = savedInstanceState?.getString(TEXT_VIEW_KEY)
}

// invoked when the activity may be temporarily destroyed, save the instance state here
override fun onSaveInstanceState(outState: Bundle?) {
   outState?.run {
      putString(GAME_STATE_KEY, gameState)
      putString(TEXT_VIEW_KEY, textView.text.toString())
   }
   // call superclass to save any view hierarchy
   super.onSaveInstanceState(outState)
}
```

Como alternativa para definir o arquivo XML e passá-lo para setContentView(), você pode criar novos objetos View no seu código de atividade e criar uma hierarquia de visualização por meio da inserção de novas View em um ViewGroup. Em seguida, você usará o layout passando a raiz ViewGroup para setContentView(). Para ter mais informações sobre a criação de uma interface do usuário, consulte a documentação Interface do usuário.

Sua atividade não reside no estado "Criado". Após o método onCreate() concluir a execução, a atividade insere o estado *Iniciado* e o sistema chama os métodos onStart() e onResume() em rápida sucessão. A próxima seção explica o callback onStart().

## onStart()

Quando a atividade insere o estado "Iniciado", o sistema invoca esse callback. A chamada onStart() torna a atividade visível ao usuário, à medida que o aplicativo prepara a atividade para inserir o primeiro plano e se tornar interativa. Por exemplo, é nesse método que o aplicativo inicializa o código que mantém a IU.

Quando a atividade é movida para o estado "Iniciado", qualquer componente ciente do ciclo de vida que esteja ligado ao ciclo de vida da atividade receberá o evento ON_START.

O método onStart() faz a conclusão muito rapidamente e, como no caso do estado "Criado", a atividade não reside no estado "Iniciado". Quando a finalização é feita pelo callback, a atividade insere o estado *Retomado* e o sistema invoca o método onResume().

## onResume()

Quando a atividade insere o estado "Retomado", ela vem para o primeiro plano e o sistema invoca o callback onResume(). É nesse estado que o aplicativo interage com o usuário. O app permanece nesse estado até que algo afete o foco do app. Esse evento pode ser, por exemplo, receber uma chamada telefônica, navegar pelo usuário para outra atividade ou desativar a tela do dispositivo.

Quando a atividade é movida para o estado "Retomado", qualquer componente ciente do ciclo de vida ligado ao ciclo de vida da atividade receberá o evento ON_RESUME. É nesse momento que os componentes do ciclo de vida podem ativar qualquer funcionalidade que precise operar enquanto o componente estiver visível e em primeiro plano, como o início da visualização da câmera.

Quando ocorre um evento de interrupção, a atividade insere o estado *Pausado* e o sistema invoca o callback onPause().

Caso a atividade retorne do estado "Pausado" para o estado "Retomado", o sistema chamará novamente o método onResume(). Dessa forma, implemente o onResume() para inicializar os componentes liberados durante onPause() e execute outras inicializações que devem ocorrer sempre que a atividade entrar no estado "Retomado".

Este é um exemplo de componente ciente do ciclo de vida que acessa a câmera quando o componente recebe o evento ON_RESUME:

Kotlin

```
class CameraComponent : LifecycleObserver {

    ...

    @OnLifecycleEvent(Lifecycle.Event.ON_RESUME)
    fun initializeCamera() {
        if (camera == null) {
            getCamera()
        }
    }

    ...
}
```

O código acima inicializa a câmera quando o LifecycleObserver recebe o evento ON_RESUME. No entanto, no modo de várias janelas, a atividade poderá ficar completamente visível mesmo quando estiver no estado "Pausado". Por exemplo, quando o usuário está no modo de várias janelas e toca na outra janela que não contém a atividade, essa atividade será movida para o estado "Pausado". Se você quiser que a câmera seja ativada somente quando o aplicativo estiver Retomado (visível e ativo em primeiro plano), inicialize a câmera após o evento ON_RESUME demonstrado acima. Caso queira mantê-la ativada quando a atividade estiver pausada embora visível (por exemplo, no modo de várias janelas), inicialize a câmera após o evento ON_START. Perceba, no entanto, que deixar a câmera ativa quando a atividade estiver pausada pode negar o acesso de outro aplicativo retomado no modo de várias janelas à câmera. Às vezes, pode ser necessário manter a câmera ativa enquanto sua atividade é pausada, mas, na verdade, ela pode prejudicar a experiência do usuário. Pense cuidadosamente sobre onde, no ciclo de vida, é mais apropriado assumir o controle de recursos compartilhados do sistema no contexto de várias janelas. Para saber mais sobre o suporte ao modo de várias janelas, consulte Suporte a várias janelas.

Independentemente de qual evento de construção você escolher para executar uma operação de inicialização, certifique-se de usar o evento de ciclo de vida correspondente para liberar o recurso. Se você inicializar algo após o evento ON_START, libere ou finalize esse item após o evento ON_STOP. Caso você inicialize após o evento ON_RESUME, faça a liberação após o evento ON_PAUSE.

O snippet de código acima posiciona o código de inicialização da câmera em um componente ciente do ciclo de vida. Em vez disso, você pode colocar esse código diretamente nos callbacks do ciclo de vida da atividade, como onStart() e onStop(), mas isso não é recomendado. A adição dessa lógica a um componente independente e ciente do ciclo de vida permite reutilizar o componente em várias atividades sem ter que duplicar o código.

Consulte Como gerenciar ciclos de vida com componentes que os reconhecem para saber como criar um componente ciente do ciclo de vida.

## onPause()

O sistema chama esse método como a primeira indicação de que o usuário está deixando sua atividade, embora nem sempre signifique que a atividade esteja sendo destruída. Isso indica que a atividade não está mais em primeiro plano, embora ainda possa estar visível se o usuário estiver no modo de várias janelas. Use o método onPause() para pausar ou ajustar operações que não devem continuar (ou que precisem continuar com moderação) enquanto a Activity estiver no modo "Pausado" e aquelas que você espera retomar em breve. Há vários motivos pelos quais uma atividade pode entrar nesse estado. Por exemplo:

- Algum evento interromper a execução do aplicativo, conforme descrito na seção onResume(). Esse é o caso mais comum.
- No Android 7.0 (API de nível 24) ou mais recentes, diversos aplicativos operam no modo de várias janelas. Como só um dos aplicativos (janelas) tem foco a qualquer momento, o sistema pausa todos os outros aplicativos.
- Uma nova atividade semitransparente (como uma caixa de diálogo) é aberta. Enquanto a atividade estiver parcialmente visível, mas não for a atividade em foco, ela permanecerá pausada.

Quando a atividade é movida para o estado pausado, qualquer componente ciente do ciclo de vida ligado ao ciclo de vida da atividade receberá o evento ON_PAUSE. É nesse momento que os componentes do ciclo de vida podem interromper qualquer funcionalidade que não precise operar enquanto o componente não estiver em primeiro plano, como na pausa de uma visualização da câmera.

Também é possível usar o método onPause() para liberar recursos do sistema, tratamento de sensores (como GPS) ou quaisquer recursos que possam afetar a duração da bateria enquanto a atividade estiver pausada e o usuário não precisar deles. No entanto, como mencionado acima, na seção onResume(), uma atividade pausada ainda poderá ser completamente visível no modo de várias janelas. Assim, considere usar onStop() em vez de onPause() para liberar ou ajustar completamente operações e recursos relacionados à IU para melhorar o suporte do modo de várias janelas.

O exemplo a seguir da reação de um LifecycleObserver ao evento ON_PAUSE é a contraparte do evento ON_RESUME acima, que libera a câmera inicializada após o recebimento do evento ON_RESUME:

Kotlin

```
class CameraComponent : LifecycleObserver {
```

```
    ...

    @OnLifecycleEvent(Lifecycle.Event.ON_PAUSE)
    fun releaseCamera() {
        camera?.release()
        camera = null
    }

    ...
}
```

O snippet de código acima posiciona o código de liberação da câmera após o recebimento do evento ON_PAUSE pelo LifecycleObserver. Como mencionado antes, consulte Como gerenciar ciclos de vida com componentes que os reconhecem para saber como criar um componente ciente do ciclo de vida.

A execução onPause() é muito breve e não oferece necessariamente tempo suficiente para realizar operações de salvamento. Por isso, **não** use onPause() para salvar dados do aplicativo ou do usuário, fazer chamadas de rede ou executar transações do banco de dados. Esse tipo de trabalho pode não ser concluído antes da finalização do método. Em vez disso, realize operações de desligamento pesadas durante onStop(). Para mais informações sobre operações adequadas a serem executadas durante onStop(), consulte onStop(). Para mais informações sobre como salvar dados, consulte Como salvar e restaurar o estado da atividade.

A conclusão do método onPause() não significa que a atividade saia do estado "Pausado". Na verdade, a atividade permanece nesse estado até que ela seja retomada ou fique completamente invisível para o usuário. Se a atividade for retomada, o sistema invocará mais uma vez o callback onResume(). Caso a atividade retorne do estado "Pausado" para o estado "Retomado", o sistema manterá a instância Activity residente na memória, chamando novamente a instância quando o sistema invocar onResume(). Nesse cenário, não é necessário reiniciar componentes criados durante qualquer método de callback que leve ao estado "Retomado". Se a atividade ficar completamente invisível, o sistema chamará onStop(). A próxima seção discutirá o callback onStop().

## onStop()

Quando a atividade não estiver mais visível ao usuário, ela inserirá o estado *Interrompido* e o sistema invocará o callback onStop(). Isso pode ocorrer, por exemplo, quando uma atividade recém-iniciada preenche toda a tela. O sistema também poderá chamar onStop() quando a atividade parar de operar e estiver prestes a ser concluída.

Quando a atividade é movida para o estado interrompido, qualquer componente ciente do ciclo de vida ligado ao ciclo de vida da atividade receberá o evento ON_STOP. É nesse momento que os componentes do ciclo de vida podem interromper qualquer funcionalidade que não precise operar enquanto o componente não estiver visível na tela.

No método onStop(), o aplicativo liberará ou ajustará recursos desnecessários enquanto o aplicativo não estiver visível ao usuário. Por exemplo, o aplicativo poderá pausar animações ou alternar de atualizações de local mais específicas para as menos detalhadas. O uso de onStop() em vez de onPause() garante que o trabalho relacionado à IU continue, mesmo quando o usuário estiver visualizando a atividade no modo de várias janelas.

Use onStop() também para realizar operações de desligamento de uso intensivo da CPU. Por exemplo, se você não encontrar um momento mais oportuno para salvar informações em um banco de dados, poderá fazer isso durante onStop(). O exemplo abaixo mostra uma implementação de onStop() que salva o conteúdo de uma nota de rascunho no armazenamento persistente:

Kotlin

```
override fun onStop() {
   // call the superclass method first
   super.onStop()

   // save the note's current draft, because the activity is stopping
   // and we want to be sure the current note progress isn't lost.
   val values = ContentValues().apply {
      put(NotePad.Notes.COLUMN_NAME_NOTE, getCurrentNoteText())
      put(NotePad.Notes.COLUMN_NAME_TITLE, getCurrentNoteTitle())
   }

   // do this update in background on an AsyncQueryHandler or equivalent
   asyncQueryHandler.startUpdate(
         token,     // int token to correlate calls
         null,      // cookie, not used here
         uri,       // The URI for the note to update.
         values,    // The map of column names and new values to apply to them.
         null,      // No SELECT criteria are used.
         null       // No WHERE columns are used.
   )
}
```

A amostra de código acima usa o SQLite diretamente. Em vez disso, use o Room, uma biblioteca de persistência que oferece uma camada de abstração acima do SQLite. Para saber mais sobre os benefícios de usar o Room e como

implementá-lo no seu aplicativo, consulte o guia Biblioteca de persistência Room.

Quando sua atividade entra no estado "Parado", o objeto Activity é mantido residente na memória: ele mantém todas as informações de estado e membro, mas não é anexado ao gerenciador de janelas. Quando a atividade é retomada, ela chama novamente essas informações. Não é necessário reiniciar componentes criados durante qualquer método de callback que leve ao estado "Retomado". O sistema também acompanha o estado atual de cada objeto View no layout. Portanto, se o usuário inserir um texto em um widget EditText, o conteúdo será retido e você não precisará salvar e restaurar.

**Observação:** quando a atividade for interrompida, o sistema poderá destruir o processo que contém a atividade se precisar recuperar a memória. Mesmo que o sistema destrua o processo quando a atividade estiver interrompida, ele ainda manterá o estado dos objetos `View` (como o texto em um widget `EditText`) em um `Bundle` (um blob de pares de chave-valor) e os restaurará caso o usuário navegue de volta à atividade. Para mais informações sobre a restauração de uma atividade a que um usuário retorne, consulte Como salvar e restaurar o estado da atividade.

A partir do estado "Interrompido", a atividade volta a interagir com o usuário ou para de operar e é encerrada. Se a atividade voltar, o sistema invocará onRestart(). Caso a Activity deixe de operar, o sistema chamará onDestroy(). A próxima seção explica o callback onDestroy().

## onDestroy()

onDestroy() é chamado antes de a atividade ser destruída. O sistema invoca esse callback porque:

1. a atividade está sendo finalizada (pelo fato do usuário descartá-la completamente ou devido a finish() ser chamado na atividade); ou
2. o sistema está destruindo temporariamente a atividade devido a uma mudança na configuração (como a rotação do dispositivo ou o modo de várias janelas)

Quando a atividade é movida para o estado destruído, qualquer componente ciente do ciclo de vida ligado ao ciclo de vida da atividade receberá o evento ON_DESTROY. É nesse momento que os componentes do ciclo de vida podem limpar qualquer item que eles precisarem antes da destruição da atividade.

Em vez de inserir lógica à sua atividade para determinar por que ela está sendo destruída, use um objeto ViewModel para manter os dados de visualização relevante a ela. Se a atividade for recriada devido a uma mudança na configuração, o ViewModel não precisará realizar nenhuma ação. Ela será preservada e fornecida à próxima instância da atividade. Se a atividade não for recriada, o ViewModel chamará o método onCleared(), em que ele pode limpar os dados necessários antes da destruição.

É possível distinguir entre essas duas situações com o método isFinishing().

Caso a atividade esteja sendo encerrada, onDestroy() será o callback do ciclo de vida final recebido pela atividade. Se onDestroy() for chamado como o resultado da mudança na configuração, o sistema criará imediatamente uma nova instância de atividade e chamará onCreate() nessa instância na nova configuração.

Os callbacks onDestroy() liberarão todos os recursos ainda não liberados pelos callbacks anteriores, como onStop().