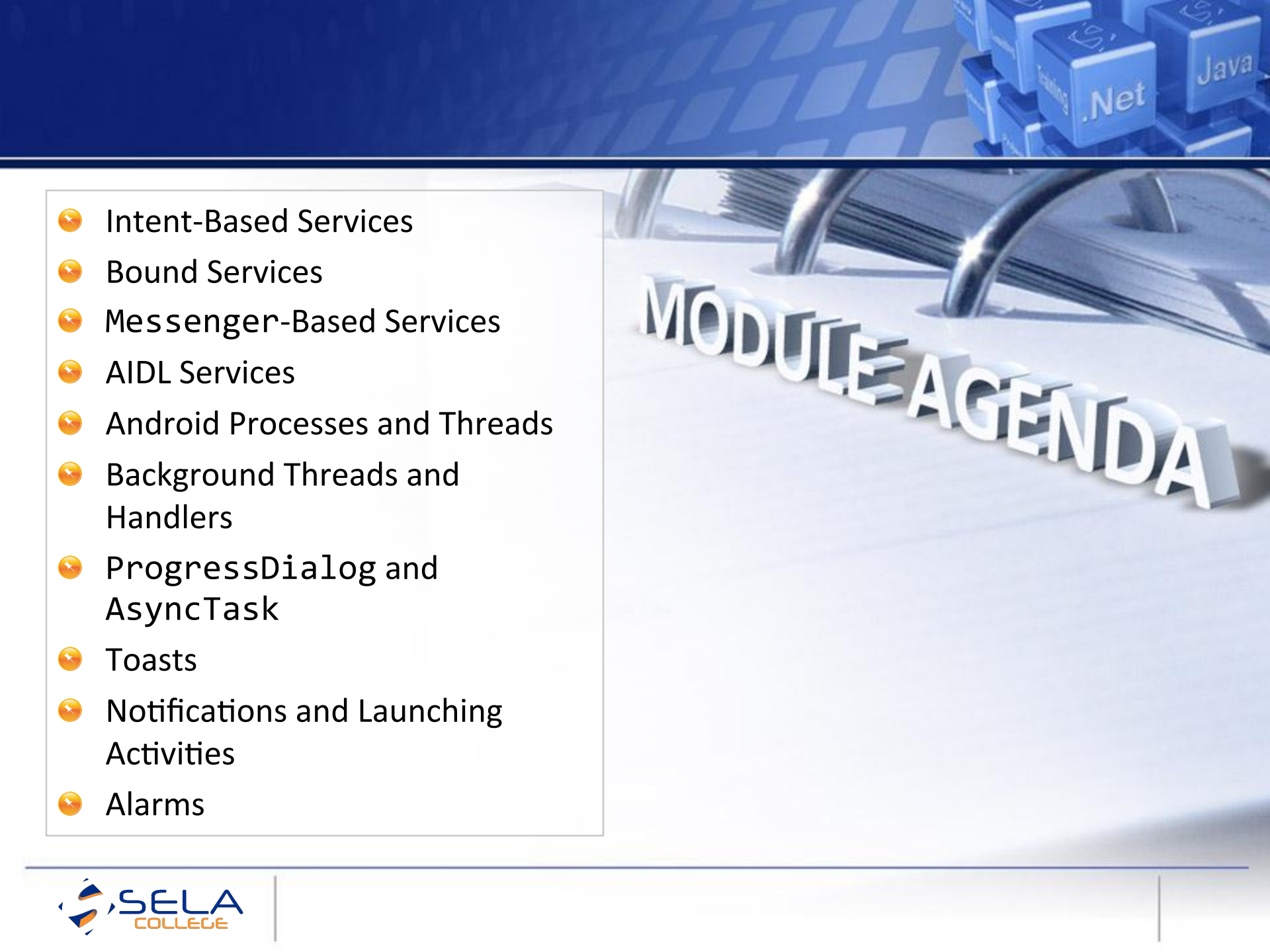
Advanced Android Application Development

Module 2 – Working in the Background



- Intent-Based Services
- Bound Services
- `Messenger`-Based Services
- AIDL Services
- Android Processes and Threads
- Background Threads and Handlers
- `ProgressDialog` and `AsyncTask`
- Toasts
- Notifications and Launching Activities
- Alarms

Services

- Long-running processes may need to run even if the user doesn't interact with the activity
  - Even if the activity doesn't exist anymore
  - Examples: playing music, chat session, RSS updates
- Android **services** can run until stopped or until the system is short on RAM
  - Decoupled from any activity
  - Limit background work to a minimum to conserve power
- Short story: derive from `Service`
  - Long story follows

Declare the Service in the Application's Manifest

- Declare any intent filters the service might need
- Permissions needed by the service are also declared in the manifest


```
<application ...>
  <activity ...>
  </activity>
  <service android:name=".TwitterSearchService"></service>
  <service android:name=".WeatherService"></service>
</application>
```

Intent-Based Services

- Activities can start and stop services based on `Intent` instances
  - `startService()` to start a service
  - `stopService()` to stop a service
  - The service doesn't go away until an activity stops it, or until it stops itself (`stopSelf()`)
  - The `IntentService` base class simplifies such services
- Information exchange is limited:
  - Parameter passing is limited to the intent extras
  - Results flow through broadcasts or other intents

Intent-Based One-Shot Service

- Override `onStart()`, handle the intent, stop self
- Results are delivered through a broadcast intent


```
public class WeatherService extends Service {
  public void onStart(Intent intent, int startId) {
    super.onStart(intent, startId);
    String location = intent.getStringExtra("Location");
    Intent results = getWeatherUpdate(location);
    sendBroadcast(results);
    stopSelf();
  }
  ...
}
```

Starting the Service from an Activity

- Create an implicit or explicit intent targeting the service
  - Like activities, services can use intent filters
- Pass it to `startService()`


```
Intent start = new Intent(this,
  WeatherService.class);
start.putExtra(
  "Location", input.getText().toString());
startService(start);
```

Bound Services

- Services can be bound to activities, creating a long-term connection, which may be bidirectional
  - When there are no bound clients, the service is destroyed
  - Bound services can support `startService()` and `stopService()`; will not be destroyed if clients are bound
- To provide binding, override `onBind()` and return an `IBinder` implementation
  - For local process services, derive from `Binder`
  - For simple cross-process messaging, use `Messenger`
  - For advanced cross-process messaging, use AIDL

Local Services with Binder

- Return a Binder-derived class from onBind()
- Callers will down-cast and get your service instance
  - Your class can return an interface, or be a façade


```
public class TwitterSearchService extends Service {
  public class _Binder extends Binder {
    public TwitterSearchService getService() {
      return TwitterSearchService.this;
    }
  }
  private Binder binder = new _Binder();
  public IBinder onBind(Intent intent) { return binder; }
  ...
}
```

Callers will get the instance and work as usual

Binding to a Service from an Activity

- Implement the `ServiceConnection` interface
- Retrieve the service by down-casting the `IBinder`


```
private ServiceConnection connection = new ServiceConnection() {
  @Override
  public void onServiceConnected(ComponentName name, IBinder service) {
    twitterService = ((TwitterSearchService._Binder)service).getService();
  }
  @Override
  public void onServiceDisconnected(ComponentName name) {
    twitterService = null;
  }
};
```

Binding to a Service from an Activity

- Call `bindService()` to create the connection, and `unbindService()` to destroy it
- You can use the connection only after `onServiceConnected()` has been called


```
protected void onStart() {
  super.onStart();
  Intent intent = new Intent(this, TwitterSearchService.class);
  bindService(intent, connection, Context.BIND_AUTO_CREATE);
}
protected void onStop() {
  super.onStop();
  unbindService(connection);
}
```

Foreground Services

- Services that perform critical work that the user is aware of should register as foreground services
  - For example, a music player service
  - This requires an active notification (discussed later)
- Call `startForeground()` with the notification, and `stopForeground()` when done

Background Services

Messenger-Based Services

- Instead of calling methods, the client can send messages to the service
- Using the `Handler` and `Messenger` classes the service doesn't need to take care of binding and queuing client calls to a single thread
- This works across processes
  - However, only `Parcelable` data can be transferred between processes
- Two-way communication requires another `Messenger` on the client's side

Parcelable

- Only objects that implement `Parcelable` can be passed across process boundaries
  - `writeToParcel()` stores the object's state in a `Parcel`
  - Static CREATOR field is used when reading from the `Parcel` and creating an instance of your class
- Many Android types are `Parcelable`, too
- `List<T>` and `Map<K,V>` of `Parcelable` objects are also supported

Example of a Parcelable Class


```
public class WeatherResult implements Parcelable {
    private String location;
    private float temperature;
    public WeatherResult(String location, float temperature) { ... }
    public String getLocation() { return location; }
    public float getTemperature() { return temperature; }
    public int describeContents() { return 0; }
    public void writeToParcel(Parcel dest, int flags) {
      dest.writeString(location);
      dest.writeFloat(temperature);
    }
    public static final Parcelable.Creator<WeatherResult> CREATOR = new Creator<WeatherResult>()
    {
      public WeatherResult[] newArray(int size) { return new WeatherResult[size]; }
      public WeatherResult createFromParcel(Parcel source) { return new WeatherResult(source); }
    };
    private WeatherResult(Parcel source) {
      location = source.readString();
      temperature = source.readFloat();
    }
}
```

Implementing a Messaging Service

- Create a `MessageHandler` for incoming messages
- Create a `Messenger` on top of it
  - Return `Messenger.getBinder()` from `onBind()`


```
public class MessagingWeatherService extends Service {
  private class MessageHandler extends Handler {
    public void handleMessage(Message msg) {
      switch (msg.what) {
        case MESSAGE_GET_WEATHER:
          getWeather(msg.getData().getString("Location"));
        default: super.handleMessage(msg);
  } } }
  private final Messenger messenger = new Messenger(new MessageHandler());

  public IBinder onBind(Intent intent) { return messenger.getBinder(); }
}
```

Binding to a Messaging Service

- The `ServiceConnection` implementation creates a `Messenger` instance from the `IBinder`


```
private ServiceConnection weatherConnection = new ServiceConnection() {
  public void onServiceConnected(ComponentName name, IBinder service) {
     weatherMmessenger = new Messenger(service);
  }
  public void onServiceDisconnected(ComponentName name) {
    weatherMessenger = null;
  }
};
```

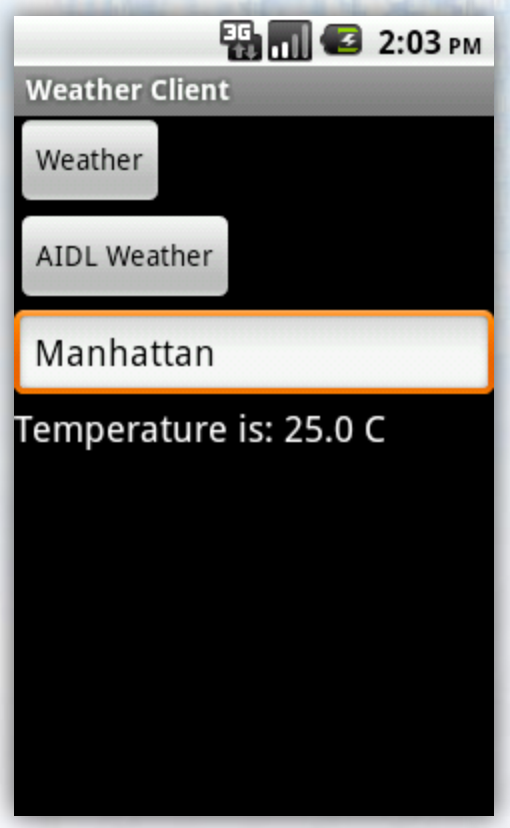
Sending Messages

- Create a message and populate it with data
  - `arg1`, `arg2`, and `obj` are there for speed
- Send messages using `Messenger.send()`


```
Message message = Message.obtain(
  null, MessagingWeatherService.MESSAGE_GET_WEATHER);
Bundle data = new Bundle();
data.putString("Location", input.getText().toString());
message.setData(data);
try {
  weatherMessenger.send(message);
} catch (RemoteException e) {
  ...
}
```

Set replyTo field to a Messenger instance for two-way communication

Messenger-Based Service



AIDL

- Android Interface Definition Language is used to define cross-process service interfaces
  - Allows multiple clients to call your service using method calls through a proxy-stub design
  - Multiple client calls may be serviced concurrently (on multiple threads)
- Clients and services share **.aidl** files that describe custom `Parcelable` types and service interfaces
  - Only `Parcelable` types can appear in the interface
- Good form to configure AIDL services with an intent filter for the interfaces they implement

AIDL interface Definition

- Only methods are allowed – no static fields
- Non-primitive parameters must have a direction
  - in, inout, out
- Must import all non-primitive types


```
package com.selagroup.services;

import com.selagroup.services.WeatherResult;

interface IWeatherService {
   WeatherResult getWeather(in String location);
}
```

AIDL parcelable Definition

- Along with the `Parcelable` class definition, must provide an **.aidl** file with a declaration


```
package com.selagroup.services;

parcelable WeatherResult;
```

Implementing The Service

- To implement the service, use the automatically-generated stub, which serves as a binder
- Return it from onBind()


```
private final IWeatherService.Stub binder = new Stub() {
  @Override
  public WeatherResult getWeather(String location) throws RemoteException
   {
    String escapedLocation = location.replace(' ', '+');
    float temperature = WeatherUtil.getTemperature(escapedLocation);
    return new WeatherResult(location, temperature);
  }
};
public IBinder onBind(Intent intent) { return binder; }
```

Implementing The Client

- The client must have access to the **.aidl** files
- Use `bindService()` as always
- ServiceConnection implementation uses `Stub.asInterface()` to convert binder


```
private ServiceConnection c = new ServiceConnection() {
  public void onServiceConnected(
          ComponentName name, IBinder service) {
    theService = IWeatherService.Stub.asInterface(service);
  }
  public void onServiceDisconnected(ComponentName name) { ... }
};
```

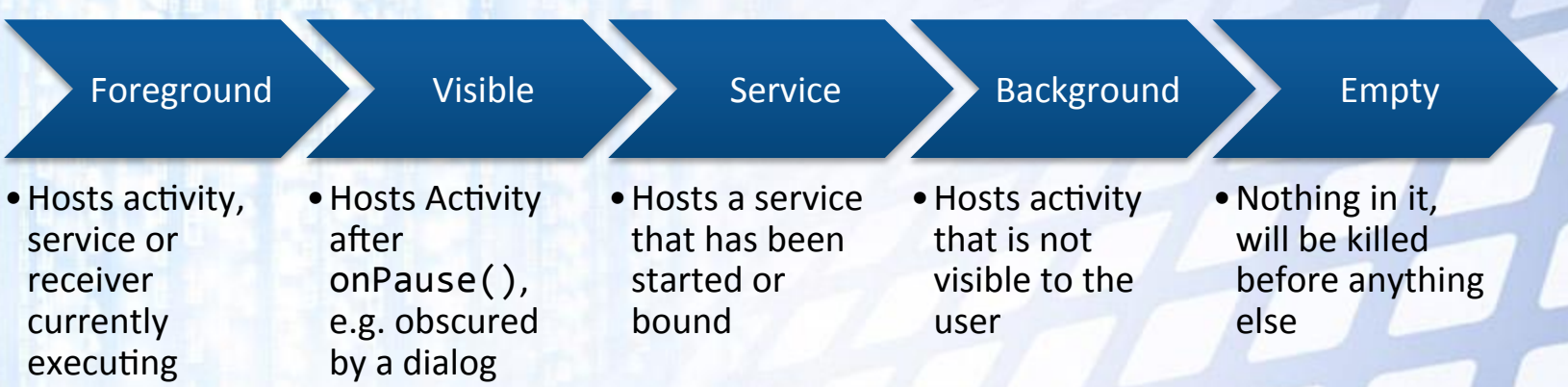
Threading Model

- Local (in-process) calls are serviced by the calling thread
- Remote calls are serviced by a process-specific thread pool, with multiple calls at the same time
  - The caller's thread blocks while waiting for your service
- The oneway keyword causes the caller to return immediately
  - Does not change the behavior of in-process calls
- Callbacks are supported – the client can implement an AIDL interface and pass to the service

Cross-Process Service Communication

Android Processes

- Android kills processes in order of importance



  - Priority is inherited – a service or content provider serving a component have at least the component's priority
  - Service processes are ranked higher than background activities, so it's a good idea to create a service rather than a background activity thread

Background Threads

- Why bother with threads?

Responsiveness

- Android is fairly aggressive towards non-responsive activities, broadcast receivers, etc.
- Users are fairly aggressive towards such apps, too

Concurrency

- Do I/O while doing other things
- Allow additional user interaction

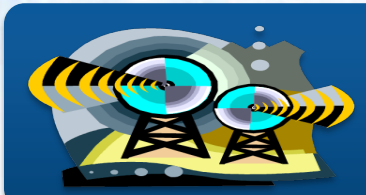
Parallelism

- New devices have two, or even four processing cores

Responsiveness Requirements

- Android ensures overall system responsiveness, and displays “Application Not Responding” messages when appropriate

Activities must respond to an input event within 5 seconds



Broadcast receivers must complete their `onReceive()` method within 10 seconds

- Failure to comply with these guidelines may result in termination

Two Responsiveness Guidelines

- Do Not Block The UI Thread
- Do Not Access UI Widgets From A Non-UI Thread

Threads and Handlers

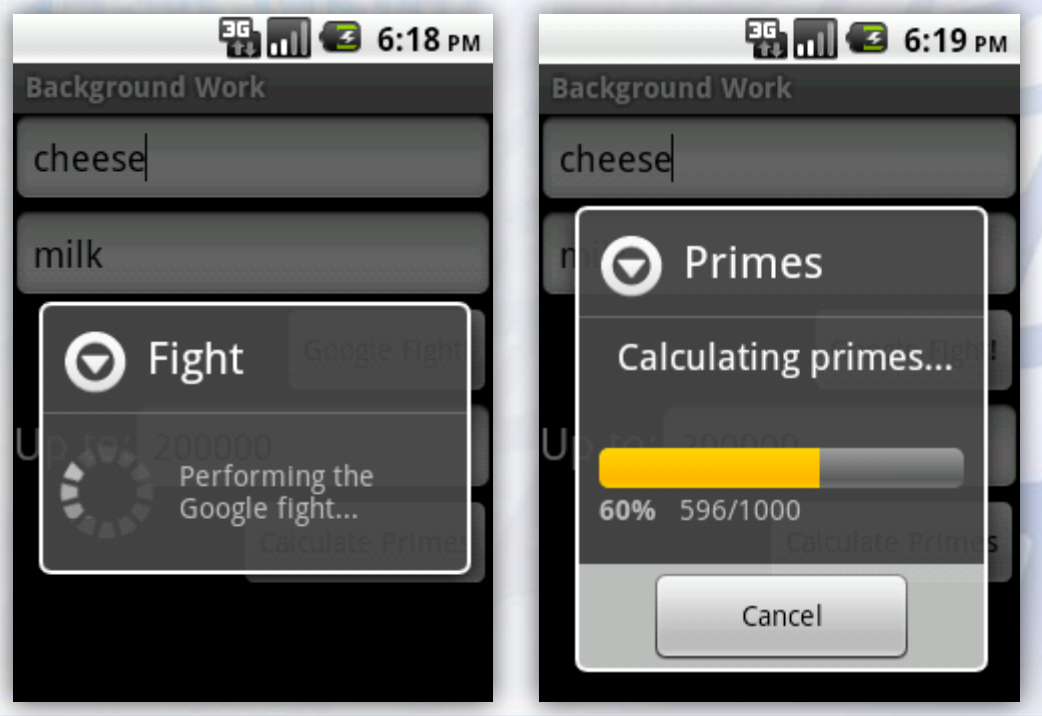
- Create a thread using `new Thread().start()`, specifying a `Runnable`
  - Can give the thread a name, priority
- To communicate with the UI, use `Handlers`
  - Create a `Handler` instance on the UI thread – it captures the context and you can `post(Runnable)` to it later
- For updating specific views, use `View.post()`
- Consider `Activity.runOnUiThread()` – will not marshal your call if you're on the right thread

Running Multiple Threads And Updating the UI


```
AtomicInteger count = new AtomicInteger(0);
Runnable runnable = new Runnable() {
  public void run() {
    //...do some lengthy operation
    if (count.incrementAndGet() == 2) {
      runOnUiThread(new Runnable() {
        public void run() {
          //...report completion of all tasks
        }
      });
    }
  }
};
new Thread(runnable).start();
new Thread(runnable).start();
```

ProgressDialog

- Easily display progress of background operations in a standard way with `ProgressDialog`
- Two styles: spinner and horizontal



AsyncTask

- Derive from `AsyncTask<In,Out,Progress>` for easy background execution with progress reports
- Override `doInBackground()` which does the work
  - Optionally override `onProgressUpdate()`, `onPostExecute()` to report progress (automatically marshaled to UI thread)
- Goes hand-in-hand with `ProgressDialog` for reporting progress to the user

Using AsyncTask


```
public class PrimeFinder extends AsyncTask<Integer, Void, Integer> {
    //...set up progress dialog, etc.
    protected void onProgressUpdate(Integer... values) {
          progressDialog.incrementProgressBy(1);
    }
    protected void onPostExecute(Integer result) {
          progressDialog.dismiss();
          Toast.makeText(context, "Finished work.", Toast.LENGTH_SHORT).show();
    }
    protected Integer doInBackground(Integer... params) {
          for (int i = params[0]; i < params[1]; ++i) {
                //...do some heavy lifting with the number
                if (i % CHUNK_SIZE == 0) {
                      publishProgress(start, end, i);
                } //...could also check for cancellation with isCancelled()
          }
    }
}
```

Background Threads

Additional Threads = Problems

- By creating additional threads, you expose yourself to the traditional multithreading problems

Tips

- Consider `java.util.concurrent` and other packages
- Take threads seriously

Notification Types

- A service might need to alert the user to various interesting events that demand his attention

Toast

- A simple transient message

Notification

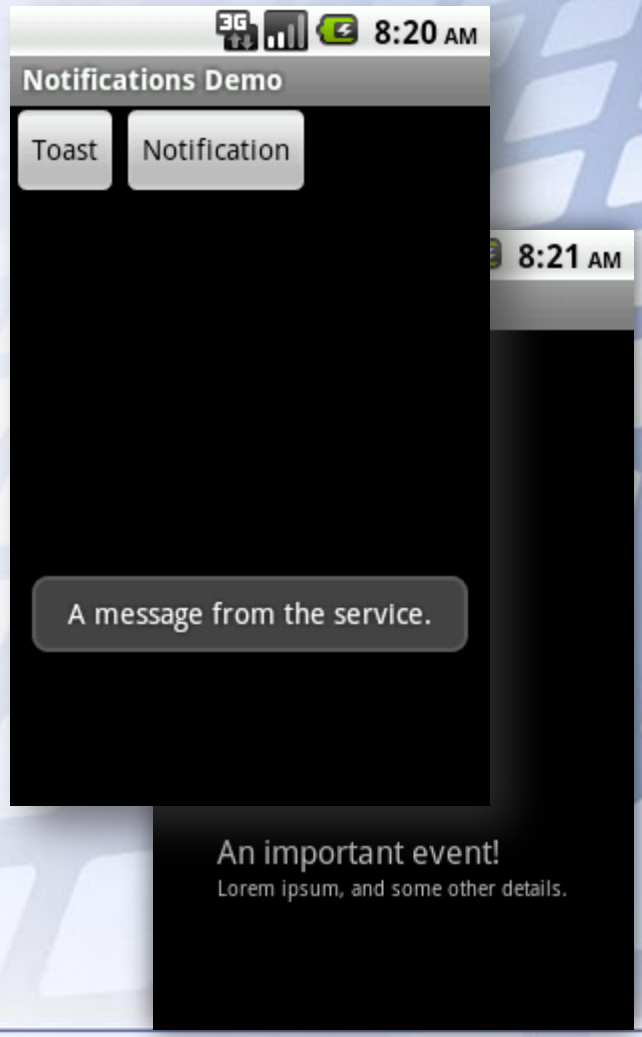
- Recommended; doesn't interrupt the user
- Appears in the title bar, can use LED, vibration, etc.

Dialog

- Not recommended; interrupts the user
- If at all, start an activity with a dialog-theme

Toasts

- A toast is a simple transient message that doesn't steal focus
  - "Battery is fully charged"
  - "File saved successfully"
- Use `Toast.makeText()` for the simplest toasts
  - Customize a toast with `setView()` for an arbitrary view
  - Set the toast's duration
- Use toasts only from GUI thread



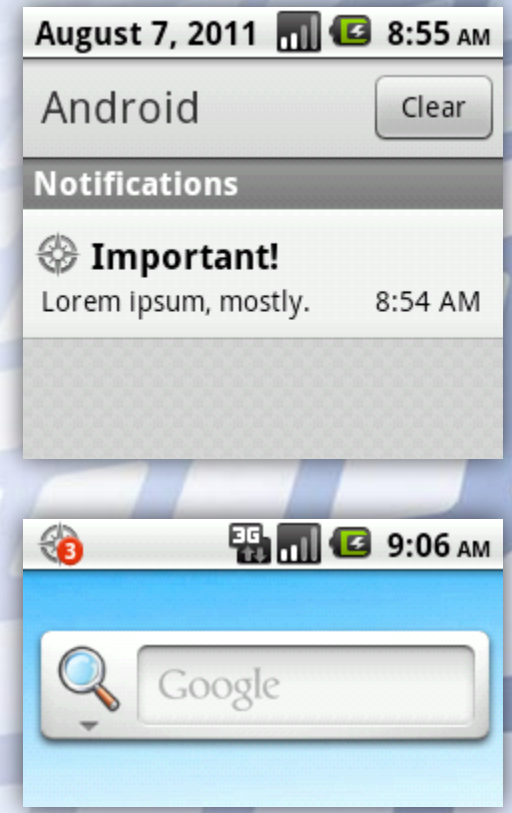
Simple and Customized Toasts


```
Toast.makeText(getApplicationContext(),
  "A message from the service.",
  Toast.LENGTH_SHORT).show();

Toast toast = new Toast(getApplicationContext());
LinearLayout root = new LinearLayout(getApplicationContext());
root.setOrientation(LinearLayout.VERTICAL);
TextView firstTextView = new TextView(getApplicationContext());
firstTextView.setText("An important event!");
root.addView(firstTextView);
...
root.addView(secondTextView);
toast.setView(root);
toast.setDuration(Toast.LENGTH_LONG);
toast.show();
```

Notifications

- For user interaction, use the `NotificationManager` service
- `notify()` puts an icon in the status bar and an expanded notification message
  - Fires an intent when the user selects the message
  - Usually, this intent will start an activity
- Can specify a notification "count" – how many events of this type have occurred



Starting an Activity from a Notification

- Create a `PendingIntent` based on an activity-launching intent
  - Must set `FLAG_ACTIVITY_NEW_TASK`


```
NotificationManager manager =
  (NotificationManager)getSystemService(NOTIFICATION_SERVICE);
Notification note = new Notification(R.drawable.icon, title, when);
note.flags |= Notification.FLAG_AUTO_CANCEL;
Intent launch = new Intent(context, MyActivity.class);
launch.setFlags(Intent.FLAG_ACTIVITY_NEW_TASK);
PendingIntent intent = PendingIntent.getActivity(
  context, -1, activityLaunch, 0);
note.setLatestEventInfo(context, title, details, intent);
manager.notify(NOTIFICATION_ID, notification);
```

Auto-cancel when clicked

Additional Notification Customizations

- Configure device vibration (need VIBRATE permission)
  - There is also an explicit API – the `Vibrator` class

CODE

```
note.vibrate = new long[] { 500, 500, 1000, 1000 };
                            On, off   On, off
```

- Configure LED flashing pattern
  - How long to stay on, how long to stay off, flashing color

CODE

```
note.ledOnMS = note.ledOffMS = 1000;
notification.ledARGB = Color.GREEN;
note.flags |= Notification.FLAG_SHOW_LIGHTS;
```

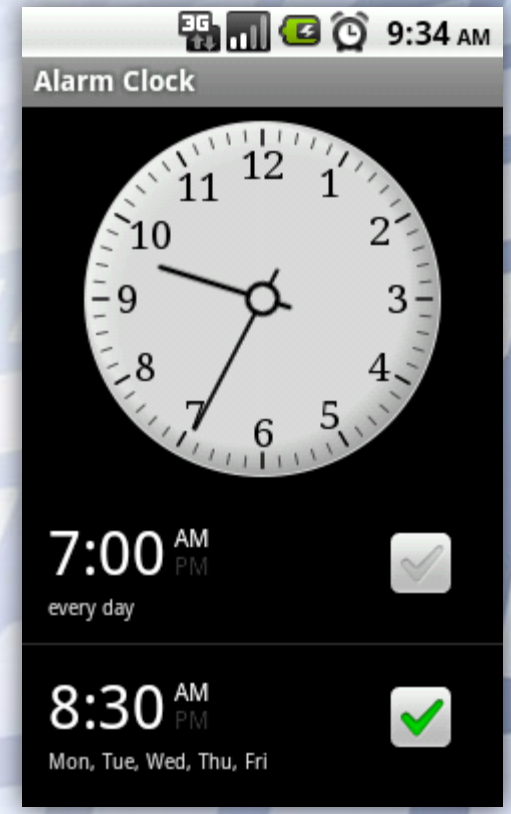
Reusing and Canceling Notifications

- Notification ID (and optional tag) passed to `notify()` controls reuse of notifications
  - Reuse the same `Notification` instance by using `setLatestEventInfo()` and `notify()` multiple times
- Cancel with `NotificationManager.cancel()`
  - There's also the opposite: `FLAG_NO_CLEAR` so that the "Clear All" button doesn't affect you
  - There's also `cancelAll()`
- `FLAG_INSISTENT` vs. `FLAG_ONGOING`
  - Insistent – repeats continuously; ongoing – represents an event currently in progress (e.g. VOIP call)

Notifications and Toasts

Alarms

- Alarms allow you to fire a broadcast intent at a predefined time
  - Designed to be used when your application or service is not running – otherwise, can use `Timer` or `Handler` services
- Use the `AlarmManager` service to set
  - Alarms can fire if the device enters sleep
  - Alarms are cleared when the device reboots
  - Can configure recurrent (interval) alarms



Setting an Alarm

- Get an instance of `AlarmManager` and set alarm
  - `RTC_WAKEUP`, `RTC` – launch at specified time
  - `ELAPSED_REALTIME_WAKEUP`, `ELAPSED_REALTIME` – launch after a specified interval has elapsed
  - `..._WAKEUP` controls whether to wake the device from sleep


```
AlarmManager alarmManager =
  (AlarmManager)getSystemService(ALARM_SERVICE);
Intent launch = new Intent(AlarmReceiver.ALARM_ACTION);
PendingIntent intent = PendingIntent.getBroadcast(
  this, ALARM_REQUEST_CODE, launch, 0);
alarmManager.set(AlarmManager.ELAPSED_REALTIME_WAKEUP,
  SystemClock.elapsedRealtime() + timeInMs, intent);
```

Canceling Alarms

- To cancel the alarm, you need the same `PendingIntent`
  - Doesn't have to be the same instance, just the same wrapped intent and request code


```
Intent launch = new Intent(
  AlarmReceiver.ALARM_ACTION);
PendingIntent intent = PendingIntent.getBroadcast(
  this, ALARM_REQUEST_CODE, launch, 0);
alarmManager.cancel(intent);
```

Setting and Canceling Alarms

Backgrounding the Time Logger

- Intent-Based Services
- Bound Services
- `Messenger`-Based Services
- AIDL Services
- Android Processes and Threads
- Background Threads and Handlers
- `ProgressDialog` and `AsyncTask`
- Toasts
- Notifications and Launching Activities
- Alarms