Android Wear
Paradigm

- Users are used to tap icons to launch apps
  - Android Wear is different
- A typical Wear app adds a card to the stream at a contextually relevant moment
  - It might also have a button that opens a full screen view for a fast micro interaction

- Wear works with Android 4.3 phones or newer ones
It is substantially different

- Screen size
- Specific assets required
- Peek card readability
- Information density
- Minimum notifications
- Use clear typography
- Be discreet if necessary
Different interactions

Real life

Get phone

Get lost in phone

More connected to the people around you, and those not with you right now
Complement, not substitute
New UI Model

• Android Wear UI consists of two main spaces centered around the core functions of **Suggest** and **Demand**

• Think of
  — Actions and not of applications
  — Context and not content
Creative vision

- **Launched automatically**: timely, relevant and very specific
- **Glanceable**: fast, sharp and immediate
- **All about suggest and demand**: it is like a great personal assistant
- **Zero or low interaction**: focused on simple interactions
Design Principles

• Focus on not stopping the user
• Design for big gestures
• Think about stream cards first
• Do one thing, really fast
• Design for the corner of the eye
• Don’t be a constant shoulder tapper
Context stream

- The context stream is a vertical list of cards, each showing a useful or timely piece of information.
- Users swipe vertically to navigate from card to card.
- Your application can create cards and inject them into the stream when they are most likely to be useful.
- Cards can be swiped horizontally to reveal additional pages:
  - Further horizontal swiping may reveal buttons, allowing the user to take action on the notification.
  - Cards can also be dismissed.
Cue Card

- If Android Wear does not suggest an answer proactively through the context stream, the cue card allows users to speak to Google
- Opened by saying, “OK Google” or by tapping on the background of the home screen
  - Swiping up on the cue card shows a list of suggested voice commands, which can also be tapped
  - Each suggested voice command activates a specific type of intent
- Applications can respond by adding or updating a stream card, or by launching a full screen application
Contextual cards in the stream

• Bridged notifications are pushed to the wearable from the connected handheld using standard Android notifications
  — We can add Wear-specific features like extra pages and voice replies by using the new notification APIs
• Contextual notifications are generated locally on the wearable and appear at contextually relevant moments
  — We can do more with this kind of cards than with a notification bridged from the handheld
  — Getting contextual triggering right is one of the most impactful things we can do to craft a great user experience
Full screen UI apps: 2D Picker

- This design pattern is available as GridViewPager
How to use a 2D Picker

- Minimize the number of cards
- Show the most popular card at the top
- Keep the cards extremely simple
- Optimize for speed over customization
Full screen UI apps

• Go full screen only when you can’t do what you want on a card
  — For example, you cannot swipe in many directions on a map or control a game with a joystick

• Quickly exit back to the stream the moment the user is done with the micro interaction

• A full screen design shouldn’t look too much like the card stream
Two options

Notifications

Applications
Wearable Data Layer

• To sync and send data between wearables and handhelds with the Wearable Data Layer APIs, you need the latest version of Google Play services
Notifications

• NotificationCompat.Builder allows us to build notifications that are displayed properly when they appear on a handheld or wearable.

• We can also add
  — Actions
  — Voice inputs
  — Predefined replies
Simple notification

```java
int notificationId = 001;
// Build intent for notification content
Intent viewIntent = new Intent(this, ViewEventActivity.class);
viewIntent.putExtra(EXTRA_EVENT_ID, eventId);
PendingIntent viewPendingIntent =
    PendingIntent.getActivity(this, 0, viewIntent, 0);

NotificationCompat.Builder notificationBuilder =
    new NotificationCompat.Builder(this)
    .setSmallIcon(R.drawable.ic_event)
    .setContentTitle(eventTitle)
    .setContentText(eventLocation)
    .setContentIntent(viewPendingIntent);

// Get an instance of the NotificationManager service
NotificationManagerCompat notificationManager =
    NotificationManagerCompat.from(this);

// Build the notification and issues it with notification manager.
nominationManager.notify(notificationId, notificationBuilder.build());
```
res/values/strings.xml

```xml
<?xml version="1.0" encoding="utf-8"?>
<resources>
    <string-array name="reply_choices">
        <item>Yes</item>
        <item>No</item>
        <item>Maybe</item>
    </string-array>
</resources>
```

```java
public static final String EXTRA_VOICE_REPLY = "extra_voice_reply";
...
String replyLabel = getResources().getString(R.string.reply_label);
String[] replyChoices = getResources().getStringArray(R.array.reply_choices);

RemoteInput remoteInput = new RemoteInput.Builder(EXTRA_VOICE_REPLY)
        .setLabel(replyLabel)
        .setChoices(replyChoices)
        .build();
```
Complete example (II)

```java
// Create an intent for the reply action
Intent replyIntent = new Intent(this, ReplyActivity.class);
PendingIntent replyPendingIntent =
    PendingIntent.getActivity(this, 0, replyIntent,
    PendingIntent.FLAG_UPDATE_CURRENT);

// Create the reply action and add the remote input
NotificationCompat.Action action =
    new NotificationCompat.Action.Builder(R.drawable.ic_reply_icon,
        getString(R.string.label), replyPendingIntent)
        .addRemoteInput(remoteInput)
        .build();

// Build the notification and add the action via WearableExtender
Notification notification =
    new NotificationCompat.Builder(mContext)
        .setSmallIcon(R.drawable.ic_message)
        .setContentTitle(getString(R.string.title))
        .setContentText(getString(R.string.content))
        .extend(new WearableExtender().addAction(action))
        .build();

// Issue the notification
NotificationManagerCompat notificationManager =
    NotificationManagerCompat.from(mContext);
noteificationManager.notify(notificationId, notification);
```
Wearable Apps

- Run directly on the device
- Fundamentally same as apps built for handheld but differ greatly in design and usability
- Basically activities with custom layouts. Access to sensors and GPU
  Small in size and functionality
Always-on apps

• Wearable apps that can transition into ambient mode are called always-on apps

• Interactive mode
  — Use full color with fluid animation in this mode
  — The app is also responsive to input

• Ambient mode
  — Render the screen with grayscale graphics and do not present any input cues in this mode
  — This display mode is only supported on devices running Android 5.1 or higher
Wearable UI Library

- BoxInsetLayout
- CardFragment
- CircledImageView
- CrossFadeDrawable
- DelayedConfirmationView
- DismissOverlayView
- DotsPageIndicator

- GridViewPager
- GridPagerAdapter
- FragmentGridPagerAdapter
- WatchViewStub
- WearableListView
Sending and Syncing Data

• The Wearable Data Layer API, part of Google Play services, provides a communication channel for your handheld and wearable apps
  – MessageApi
    • For one way communication “fire&forget”
  – DataApi
    • To sync data across all Wear devices
  – NodeApi
    • To learn about local or connected Nodes
Voice Capabilities

• System-provided voice actions are task-based and are built into the Wear platform
  – You filter for them in the activity that you want to start when the voice action is spoken
  – Examples include "Take a note" or "Set an alarm"

• App-provided voice actions are app-based, and you declare them just like a launcher icon
  – Users say "Start " to use these voice actions and an activity that you specify starts
<table>
<thead>
<tr>
<th>Call a car/taxi</th>
<th>&quot;OK Google, get me a taxi&quot;</th>
<th>Action</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>&quot;OK Google, call me a car&quot;</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>com.google.android.gms.actions.RESERVE_TAXI_RESERVATION</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Take a note</th>
<th>&quot;OK Google, take a note&quot;</th>
<th>Action</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>&quot;OK Google, note to self&quot;</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>android.intent.action.SEND</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Category</td>
</tr>
<tr>
<td></td>
<td></td>
<td>com.google.android.voicesearch.SELF_NOTE</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Extras</td>
</tr>
<tr>
<td></td>
<td></td>
<td>android.content.Intent.EXTRA_TEXT - a string with note body</td>
</tr>
</tbody>
</table>
Deployment

• The Wear app gets bundled into the phone / tablet APK and it will be automatically installed on the Wear device.

• Sometimes we will use a simple container APK that will encapsulate our Wear application and will get installed to the paired handheld (phone/tablet).
  
  — This will give our Android Wear app a Parent APK and upon installation the Wear module will be automatically installed on to a paired Wear device.
Testing
Different types

• Unit testing
• UI testing
• Functional testing
• Integration testing
• Security testing
• Compatibility testing
Three main elements

• JUnit
  — The Android JUnit extensions provide component-specific test case classes

• Monkeyrunner
  — An API for testing devices with Python programs

• UI/Application Exerciser Monkey
  — A command-line tool for stress-testing UIs by sending pseudo-random events to a device
Monkeyrunner

• Provides an API for writing programs that control an Android device or emulator from outside of Android code
• We can write a Python program that installs an Android application or test package, runs it, sends keystrokes to it, takes screenshots of its user interface, and stores screenshots on the workstation.
  — Functional testing
  — Regression testing
• One can develop an entire system of Python-based modules and programs for controlling Android devices
UI/Application Exerciser Monkey

- Is a program that runs on an emulator or device and generates pseudo-random streams of user events.
- It includes a number of options:
  - Basic configuration options, such as setting the number of events to attempt.
  - Operational constraints, such as restricting the test to a single package.
  - Event types and frequencies.
  - Debugging options.
Other testing tools

- Autoandroid
- Positron (not maintained anymore)
- Selendroid
- Robotium
- ...