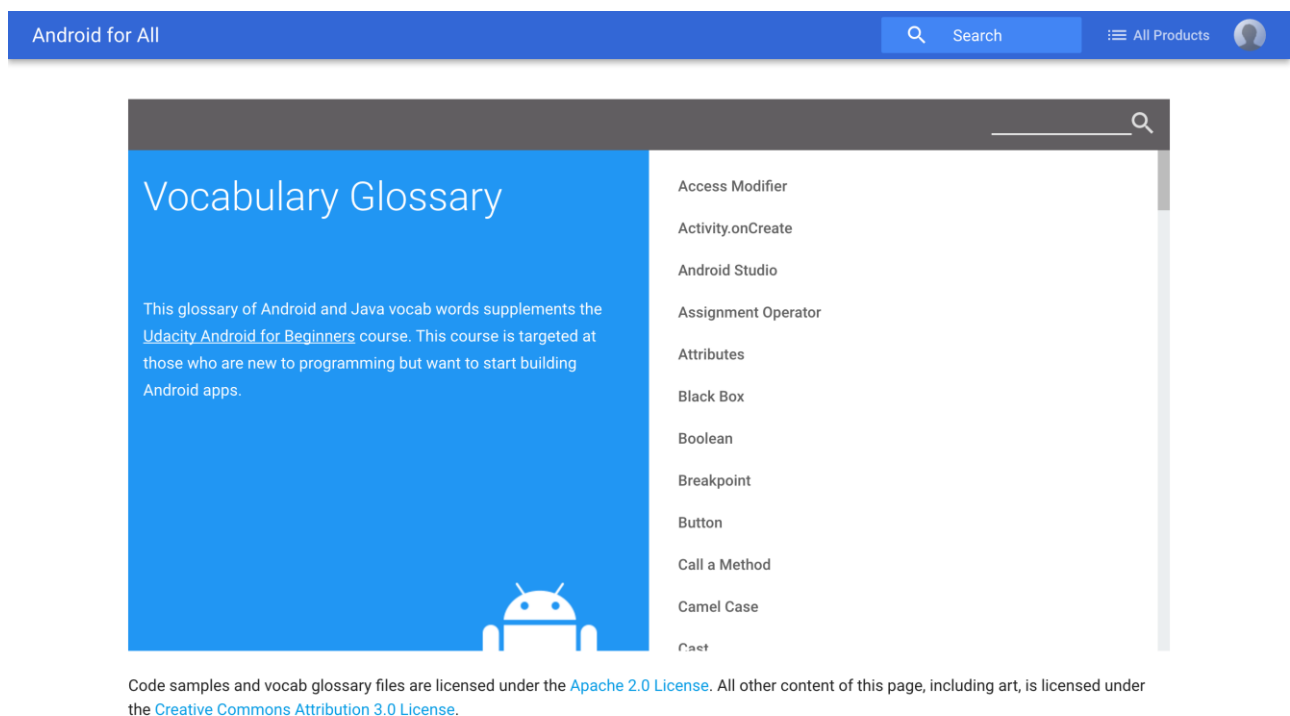


Android for All 术语表

下面的 **Android** 和 **Java** 术语表是对优达学城的 Google: Android 基础 纳米学位项目（适用于想要着手构建 **Android** 应用的编程初学者）的补充资料，由优达学城翻译提供。

如果你对其中某些术语的翻译有更好的建议，欢迎你在[论坛](#)中提出。



点击[这里](#)，访问英文版术语表。

Code samples and vocab glossary files are licensed under the [Apache 2.0 License](#). All other content of this page, including art, is licensed under the [Creative Commons Attribution 3.0 License](#).

目录

[Access Modifier（访问修饰符）](#)
[Activity.onCreate](#)
[Android Studio](#)
[Assignment Operator（赋值操作符）](#)
[Attributes（属性）](#)
[Black Box（黑盒）](#)
[Boolean（布尔）](#)
[Breakpoint（断点）](#)
[Button](#)
[Call a Method（调用方法）](#)
[Camel Case（驼峰式大小写）](#)
[Cast](#)
[Checkbox](#)
[Class（类）](#)
[Class Name（类名）](#)
[Code（代码）](#)
[Comment（注释）](#)
[Compile-time Error（编译时错误）](#)
[Constructor（构造函数）](#)
[Control Flow（控制流）](#)
[Crash（崩溃）](#)
[Data Type（数据类型）](#)
[Debug（调试）](#)
[Declare（声明）](#)
[Define a Method（定义方法）](#)
[Documentation（文档）](#)
[dp \(Density-Independent Pixel\)（与密度无关的像素）](#)
[Encapsulation（封装）](#)
[Event-Driven Programming（事件驱动编程）](#)
[Execute（执行）](#)
[Expression（表达式）](#)
[Field（域）](#)
[findViewById](#)
[Getter Method（Getter 方法）](#)
[Gist](#)
[Global Variable（全局变量）](#)
[Gradle](#)
[Hardcode（硬编码）](#)
[Hexadecimal Color \(Hex Color\)（十六进制颜色）](#)
[if/else Statement（if/else 语句）](#)
[ImageView](#)
[Import Statement（Import 语句）](#)
[Inflate](#)
[Initialize（初始化）](#)

[Input Parameter（输入参数）](#)
[Instance（实例）](#)
[Integer（整数）](#)
[Intent](#)
[Java Programming Language（Java 编程语言）](#)
[Javadoc](#)
[Layout（布局）](#)
[layout_margin](#)
[layout_weight](#)
[LinearLayout](#)
[Literal（文本）](#)
[Local Variable（局部变量）](#)
[match_parent](#)
[Method Signature（方法签名）](#)
[Method（方法）](#)
[Nested ViewGroups（嵌套式 ViewGroup）](#)
[Object（对象）](#)
[OnClickListener](#)
[Operator（操作符）](#)
[Override（覆盖）](#)
[Package Name（包名）](#)
[Padding（内边距）](#)
[Parent View（父视图）](#)
[Parse（解析）](#)
[Prototype（原型）](#)
[Pseudocode（伪代码）](#)
[Redlines（红线）](#)
[RelativeLayout](#)
[Return Value（返回值）](#)
[Robust（可靠）](#)
[Root View（根视图）](#)
[Runtime Error（运行时错误）](#)
[Setter Method（Setter 方法）](#)
[sp \(Scale-Independent Pixel\)（与比例无关的像素）](#)
[Stack Trace（堆栈跟踪）](#)
[State（状态）](#)
[String（字符串）](#)
[Style（样式）](#)
[Subclass（子类）](#)
[Superclass or Base Class（超类或基类）](#)
[System Log（系统日志）](#)
[Text Localization（文本本地化）](#)
[TextView](#)
[Theme（主题）](#)
[User Interface（用户界面）](#)
[Variable（变量）](#)
[Variable Declaration（变量声明）](#)

[Variable Name \(变量名\)](#)

[Variable Scope \(变量范围\)](#)

[View \(视图\)](#)

[ViewGroup](#)

[View Hierarchy \(视图层次结构\)](#)

[void](#)

[wrap_content](#)

[XML](#)

[XML Tag \(XML 标记\)](#)

Code samples and vocab glossary files are licensed under the [Apache 2.0 License](#). All other content of this page, including art, is licensed under the [Creative Commons Attribution 3.0 License](#).

Access Modifier（访问修饰符）

定义

计算机是按照一系列称为**程序**的指令运行的机器。**Android 设备**便是计算机，**应用**是程序。设备内部是称为**变量**的容器，用于存储数字或文本片段等**值**。

对象是变量，但在以下两个方面特殊。第一，对象中可包含更小的变量，即对象的**域**。第二，我们可向对象附加称为**方法**的一系列指令，实际上是小程序。

对象分多种**类**（类型）。每个给定类的对象都有一组相同的域和方法。针对每个类，我们必须编写**定义**：即属于此类的所有对象的域和方法列表。

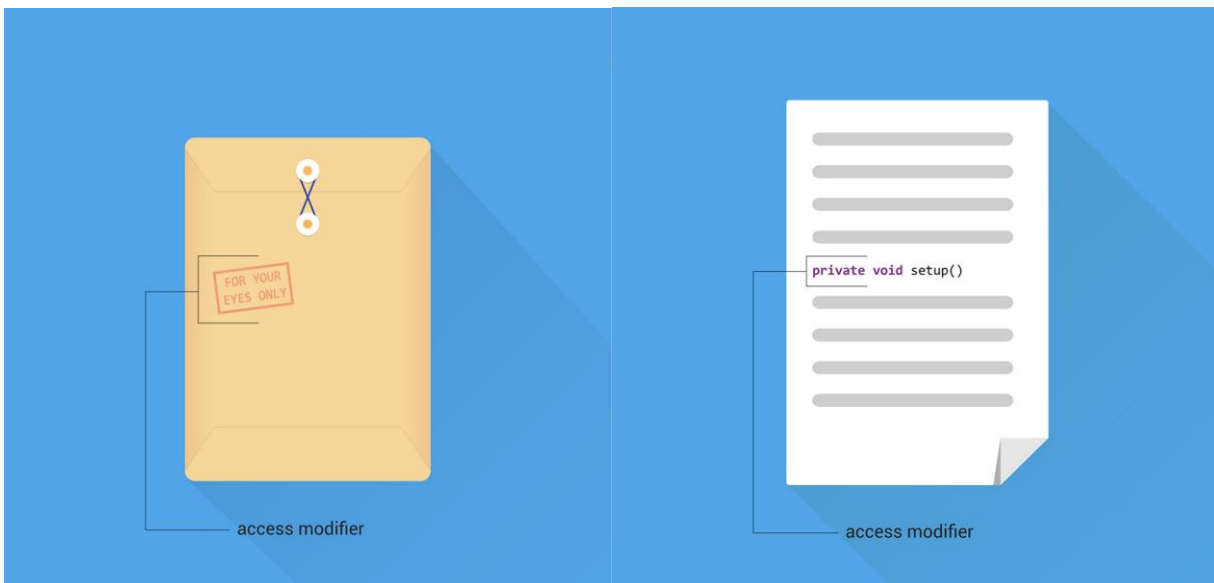
代码样例是 **Car** 类的定义。每个 **Car** 对象都包含一个名为 **mModel** 的域，用于存储该特定 **Car** 对象的型号。每个 **Car** 对象也有两个**构造函数**方法，创建 **Car** 时，必须**调用**（执行）其中的一个方法。每个构造函数为新建 **Car** 对象的 **mModel** 域赋值，同时调用 **setup** 方法以完成对象设置。

类定义为类的每个域和方法指定一个**访问修饰符**。例如，类 **Car** 的构造函数为 **public**：可在应用的其他类对象的方法中调用这些构造函数。这样，其他类的对象便可创建 **Car** 类的对象。另一方面，**mModel** 域为 **private**：只能在该域所属的类的方法内使用该域。当某个域的访问修饰符为 **private** 时，便称该域**已被封装**。**setup** 方法也为 **private**，因为该方法仅由此类的其他方法使用。

代码示例

```
public class Car {  
    private int mModel;  
  
    public Car() {  
        mModel = 0;  
        setup();  
    }  
  
    public Car(int model) {  
        mModel = model;  
    }  
}
```

```
    setup();  
}  
  
private void setup() {  
    String message = "Created a car of model number " + mModel + ".";  
    Toast toast = Toast.makeText(this, message, Toast.LENGTH_SHORT);  
    toast.show();  
}  
}
```



Activity.onCreate

定义

计算机是按照一系列称为**程序**的指令运行的机器。**Android 设备**便是计算机，**应用**是程序。设备内部是称为**变量**的容器，用于存储数字或文本片段等**值**。

能够包含小变量的大变量称为**对象**。还可向对象附加称为**方法**的一系列指令，实际上是小程序。。执行方法指令时，便是在**调用**该方法。

对象有许多个**类**，其中一个类是 **Activity**。启动应用时，将自动创建一个属于此类的对象，然后调用其中一个名为 **onCreate** 的对象方法。此方法的指令指示设备为应用创建并显示**用户界面**。此界面由屏幕上的信息显示区域和触摸敏感区（如按钮）组成。

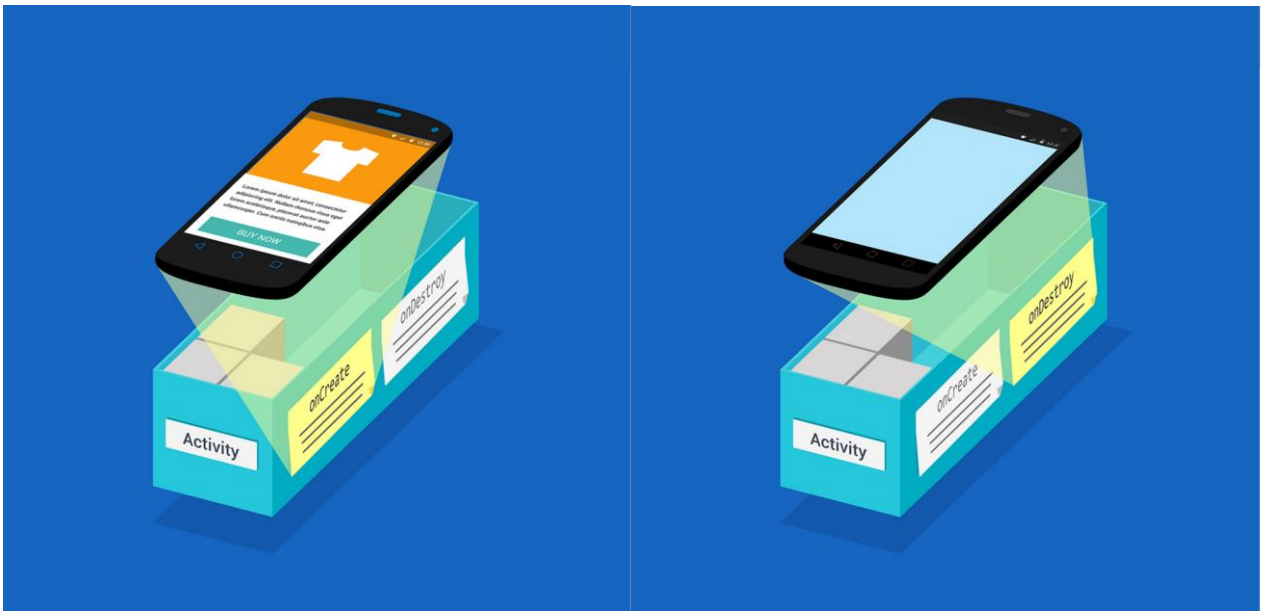
当此应用的用户界面被其他应用覆盖时（例如，电话响起时），将自动调用 **Activity** 对象的其他方法。当不再需要应用的用户界面，可销毁该界面时，将调用最后一个方法 **onDestroy**。**onCreate** 是 **Activity** 对象众多**生命周期方法**中的第一个方法。

代码示例

```
public class MainActivity extends AppCompatActivity {

    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        // The following statement creates and displays the user interface.
        setContentView(R.layout.activity_main);
    }

}
```



Android Studio

定义

即使是最简单的 Android 应用，也包含几十个文件夹和文件。Android Studio 是桌面应用程序，用于撰写和编辑这些文件。文件完成后，Android Studio 将其从 Java 语言转换为 Android 设备（通常为手机或平板电脑）能够理解的内部语言，并将完成的应用加载到设备中。Android Studio 可在 Macintosh、PC 和 Linux 上运行。



Assignment Operator（赋值操作符）

定义

计算机是按照一系列称为**程序**的指令运行的机器。**Android 设备**便是计算机，**应用**是程序。

变量是 Android 设备内部用于存储数字或文本片段等值的容器。每个变量都有一个名称，如 "x"、"y" 或 "greeting"。

在应用的指令中，我们通过编写诸如以下**表达式**指示设备将值复制到变量中。

`x = 10`

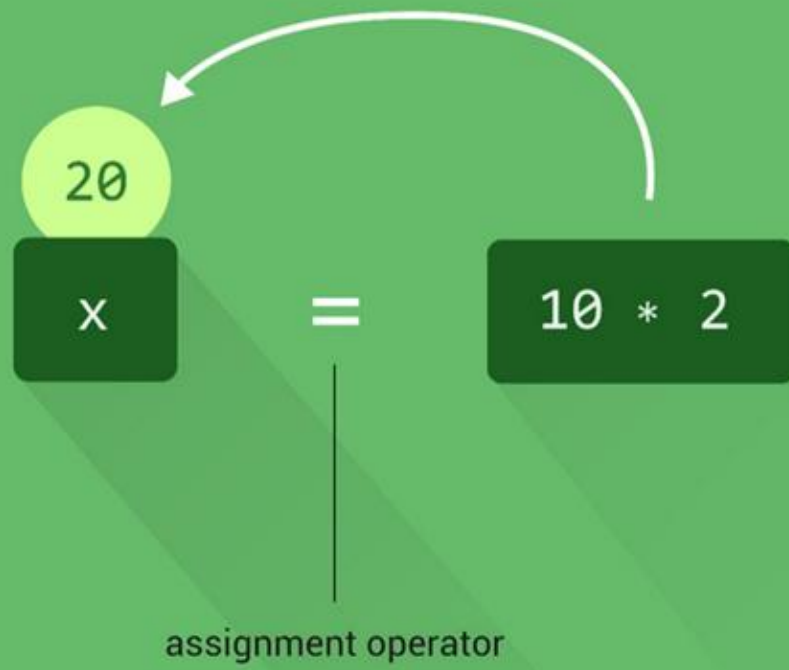
`x = y`

`greeting = "Hello"`

等号称为**赋值操作符**。等号右侧的表达式是赋值操作符的**右操作数**，等号左侧的变量是**左操作数**。赋值操作符指示设备将右操作数的值复制到左操作数中。

赋值操作符连同其两个操作数称为**赋值表达式**。Java 语言中的每个语句以分号结束，因此编写包含赋值表达式的语句时，可以是

`x = 10;`



Attributes（属性）

定义

可扩展标记语言 (XML) 是一种表示法，编写的文件包含称为**元素**的信息片段。例如，描述屏幕布局的文件可能包含表示按钮和图像的元素。每个元素的开头用两侧加有 < 和 > 的**标记**进行标记。小元素可以只包含这个标记。

元素初始**标记**内部可写入称为**属性**的小块信息。每个属性都由**名称**和**值**组成。例如，**TextView** 元素可能具有一个名称为 "text"、值为 "Hello" 的属性。我们将属性名写在左侧，值写在右侧，中间用等号连接。请务必用双引号将值括上。

attribute ☐
attribute ☐
attribute ☐

```
<TextView  
    android:layout_width="match_parent"  
    android:layout_height="wrap_content"  
    android:text="Hello"/>
```

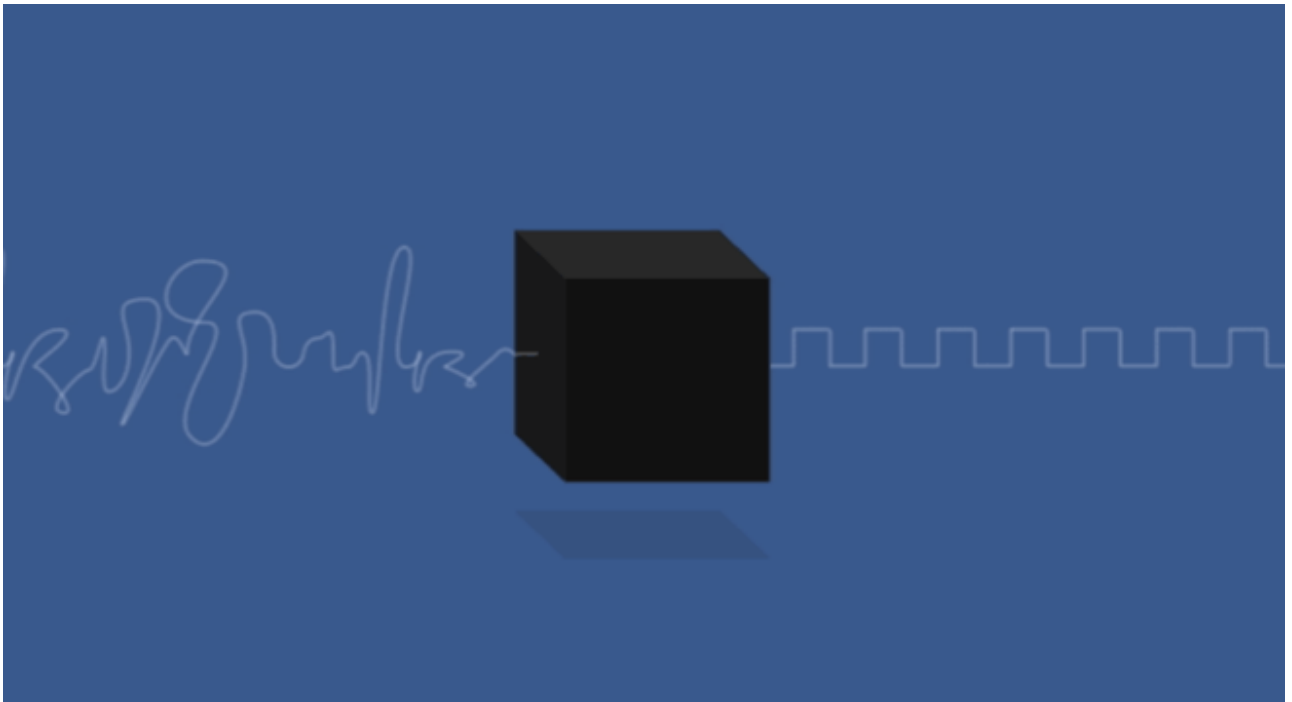
tag

Black Box（黑盒）

定义

计算机是按照一系列称为**程序**的指令运行的机器。**Android 设备**便是计算机，**应用**是程序。应用的指令划分为多个部分，称为**方法**。当指示设备执行方法时，便是在**调用**该方法。

如果知道如何调用某个方法、如何反馈信息以及如何接收结果，可将该方法视为**黑盒**：虽然不了解具体的运行方式，但能够安全地使用。



Boolean（布尔）

定义

计算机是按照一系列称为**程序**的指令运行的机器。**Android 设备**便是计算机，**应用**是程序。

应用能够指示设备操纵数字或文本片段等**值**。值分多种**类型**。例如，包含分数的数值为 **float** 类型，整数为 **int** 类型。int 类型的值可能有几十亿个，其中包括熟悉的数字 1、2 和 3，int 值可能由多种运算（包括加法和减法）计算得出。

布尔类型并不常见，因为这种类型只有两个值：**true** 和 **false**。布尔值可能由两个数字进行**比较**得出。例如，如果我们检查 x 是否等于 y，结果将是布尔类型，结果的值将是 **true** 或 **false**。该类型以 19 世纪逻辑学家 **乔治·布尔** 的名字进行命名。

变量是设备内部可包含值的容器。**布尔变量**是包含布尔值的变量。此类型的变量可以记录比较结果，供以后在应用中使用。通常，布尔变量的名称为 "is" 后接形容词。

代码示例

```
// Remember whether or not the user won the game.
boolean isWinner;
if (score >= 100) {
    isWinner = true;
} else {
    isWinner = false;
}
// Start a new game.
score = 0;
String message;
if (isWinner) {
    message = "You're a person who has won before.";
} else {
    message = "I'm glad you're going to try it again.";
```

```
}
```



Head



Tail

A **boolean** variable can contain one of two possible values, **true** or **false**.

Breakpoint（断点）

定义

计算机是按照一系列称为**程序**的指令运行的机器。**Android 设备**便是计算机，**应用**是程序。

完成应用中的各条指令后，默认情况下，**Android 设备**将进入下一指令。但某些指令会指挥设备跳转到列表中的不同指令。指令还可通知设备更改**变量**内容，变量是存储诸如数字或文本块之类**值**的小型容器。

所以现在我们知道应用有两种方式可以指示设备出错。应用可指示设备偏离正常执行，因为设备是通过指令列表的方式来执行的，或者应用会指示设备为变量赋予不正确的值。这些错误在应用中称为**程序错误**，识别并移除这些错误称为**调试**。

调试应用的一种方式时刻提高警惕，因为设备每次执行一条应用指令。我们可能还想查看变量的内容，并观看设备在每步中更改这些内容。但是有一个问题：设备通常每秒执行上百万条指令。

要将应用速度减慢到百万分之一，甚至完全暂停，则可通过 **Android Studio** 内置的工具（称为**调试器**）进行控制。调试器还可以为我们显示变量内容，甚至允许我们更改那些内容以执行小的测试与实验来找出错误所在。

但观看应用执行指令是一个极为漫长的过程，可能需要重复数百万次指令，直至我们获得可能隐藏有程序错误的指令。为了快速找到错误指令，我们可以在希望设备暂停的指令位置插入一个断点，类似停车标志或路障。然后我们可以让调试器从第一条指令开始全速运行应用，运行至断点时，设备一定会暂停供我们进行检查。



Button

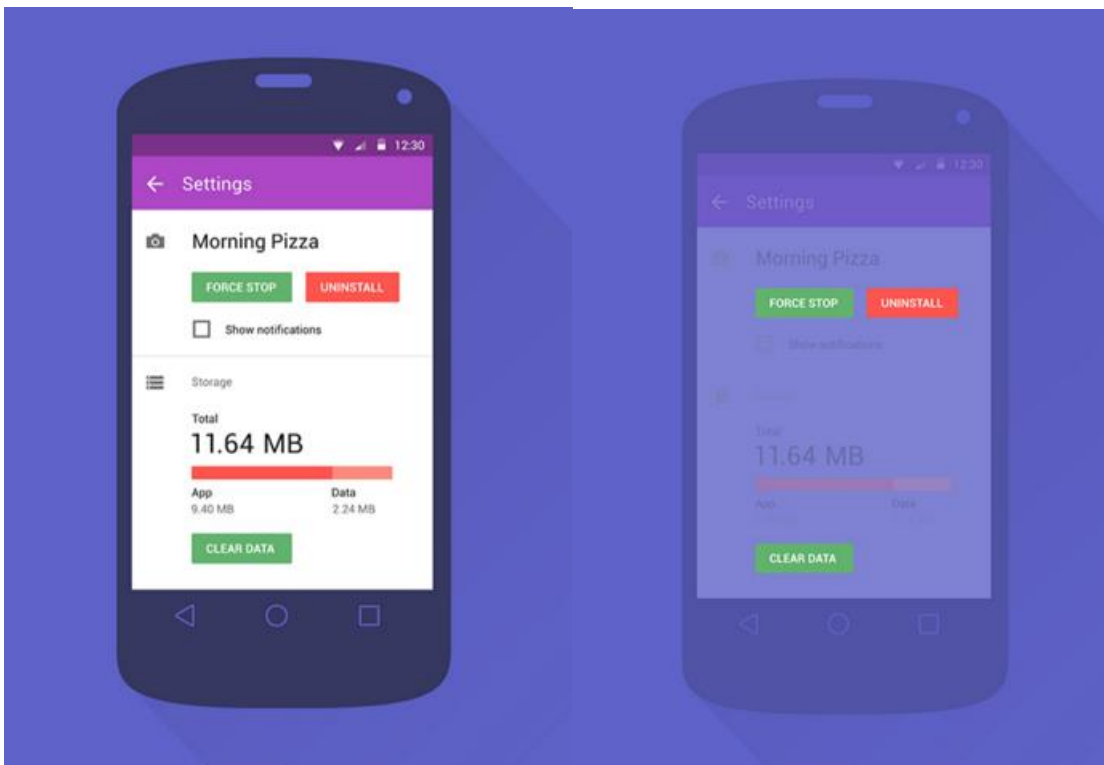
定义

视图是屏幕上的矩形区域。有一种视图是 **Button**，显示一段文本。触摸时，正确配置的 **Button** 会指示 Android 设备执行**方法**，即一系列指令，如小程序。

屏幕上的 **Button** 由 Android 设备内部的 **Java** 对象绘制。事实上，**Java** 对象是真正的 **Button**。但是在谈到用户所看到的内容时，将屏幕上的矩形区域视为“按钮”将更为方便。

代码示例

```
<Button
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:padding="8dp"
    android:background="#00FF00"
    android:text="Force stop"
    android:onClick="forceStop"/>
```



Call a Method（调用方法）

定义

计算机是按照一系列称为**程序**的指令运行的机器。**Android 设备**便是计算机，**应用**是程序。设备内部是称为**变量**的容器，用于存储数字或文本片段等**值**。

对象是变量，但在以下两个方面特殊。第一，对象中可包含更小的变量，即对象的**域**。例如，在 **MediaPlayer** 对象中可能包含多个域，用于存储正在播放的声音文件的名称、音量等级、文件回放的当前位置以及指示是否采用无限循环方式播放文件。

第二，我们可向对象附加称为**方法**的一系列指令，实际上是小程序。例如，我们的 **MediaPlayer** 对象可能具有 **play**、**pause** 和 **stop** 方法。对象的方法可使用对象内部这些方法所附加到的域。例如，**play** 方法需要使用全部四个域。

当指示计算机执行对象的方法时，便是在**调用该方法**。例如，我们可以调用 **MediaPlayer** 的 **play** 方法，让其执行播放声音文件的指令。

Call the `pause()` method that belongs to the `MediaPlayer` object.

```
MediaPlayer.pause();
```

object name

dot

method name

Zero or more parameters go here.

Camel Case（驼峰式大小写）

定义

计算机是按照一系列称为**程序**的指令运行的机器。**Android 设备**便是计算机，**应用**是程序。

通常，我们用两个单词的短语（如 "linear layout" 或 "main activity"）对设备中的内容命名。但 Java 语言不允许名称的两个单词之间有空格。如果消除空格，并将第二个词的首字母大写，这样便可知晓第一个单词的结尾位置和第二个单词的开头位置。这种排版惯例称为**驼峰式大小写**，可生成多个单词，为 Android 编程提供别样特色：LinearLayout、MainActivity、onClick。



Cast

定义

计算机是按照一系列称为**程序**的指令运行的机器。**Android 设备**便是计算机，**应用**是程序。

我们编写**表达式**，如 `10 + 20` 或 `"cup" + "cake"`

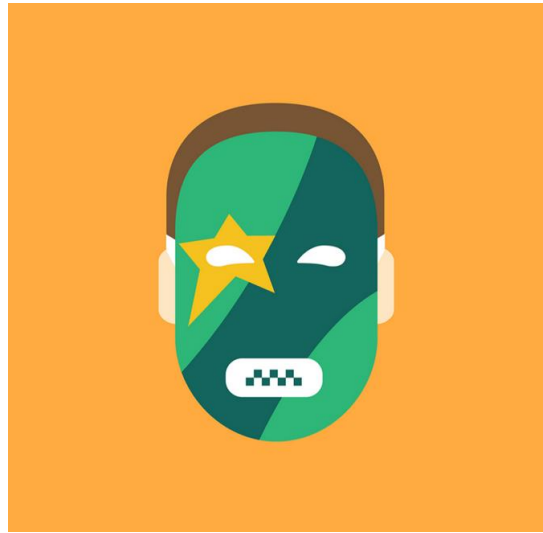
在应用的指令中，指示设备计算**值**，如 `30` 或 `"cupcake"`。值有多种**类型**：`30` 是 **integer**（整数），`"cupcake"` 是 **string**（一段文本）。

每个表达式都有特定类型的值。例如，`10 + 20` 的值是 **integer** 类型。在应用的指令中，有时一种类型的表达式中需要编写其他类型的表达式。出现这种情况时，我们需要在表达式前面加上 **cast**。**cast** 就像化妆舞会上戴的面具。在程序周围的对象看来，表达式像是另外一种类型。

cast 不会对它所施加到的表达式造成任何影响。面具下，表达式的值仍保持原始类型。

代码示例

```
// The value of the expression  
// findViewById(R.id.textView)  
// is of type View. The parenthesized (TextView) is a cast.  
// The value of the larger expression  
// (TextView) findViewById(R.id.textView)  
// is of type TextView.  
TextView textView = (TextView) findViewById(R.id.textView);
```

Checkbox

定义

计算机是按照一系列称为**程序**的指令运行的机器。**Android 设备**便是计算机，**应用**是程序。在 Android 设备内部，**变量**是用于保存数字或文本片段等**值**的容器。**对象**是大变量，其中可包含小变量。我们可向对象附加称为**方法**的一系列指令，实际上是小程序。对象分多种**类**（类型）。

Android 应用的屏幕由称为**视图**的矩形区域组成。例如，**checkbox** 是外观和作用都像复选框的触摸敏感视图：我们可以轻击选中或取消选中。请参阅材料设计规范中的 [Checkbox（复选框）](#) 和 [Checkboxes Guide（复选框指南）](#)。

屏幕上的每个视图都由对应的 Java 对象绘制。例如，复选框由 **CheckBox** 类的对象绘制。事实上，Java 对象是真正的复选框。但是在谈到用户所看到的内容时，将屏幕上的矩形区域视为“复选框”将更为方便。

CheckBox 对象有一个名为 **isChecked** 的方法，返回 **true** 或 **false** 以指示复选框当前是否已选中。

代码示例

```
<!-- Two elements in the file activity_main.xml. -->
<CheckBox
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:text="@string/sprinkles"
    android:onClick="clickSprinkles"/>

<CheckBox
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:text="@string/cherry"
    android:onClick="clickCherry"/>
```

// Two methods of class MainActivity in the file MainActivity.java.

```
public void clickSprinkles(View view) {
    CheckBox checkBox = (CheckBox) view;
    String message;
    if (checkBox.isChecked()) {
        message = "Thanks for selecting sprinkles.";
    } else {
        message = "Thanks for not selecting sprinkles.";
    }
    Toast.makeText(this, message, Toast.LENGTH_SHORT).show();
}

public void clickCherry(View view) {
    CheckBox checkBox = (CheckBox) view;
    int resourceId;
    if (checkBox.isChecked()) {
        resourceId = R.string.cherry;
    } else {
        resourceId = R.string.no_cherry;
    }
    checkBox.setText(getString(resourceId));
}
```



With sprinkles



With cherry

A checkbox can be **checked** or **unchecked**.

Class（类）

定义

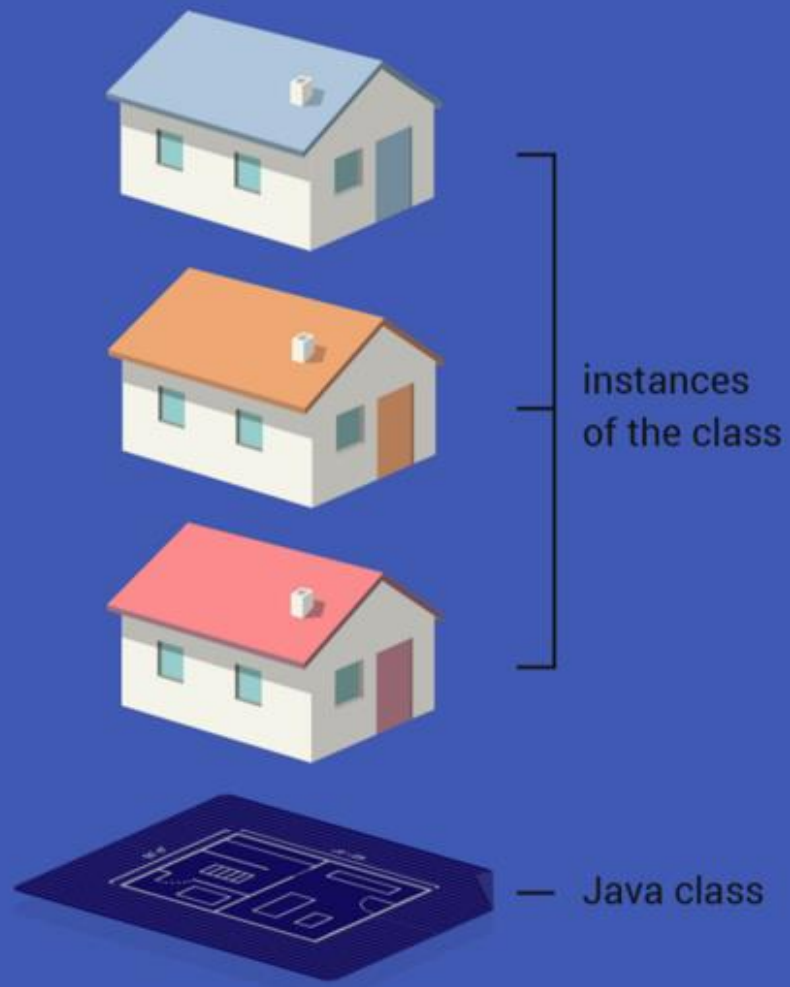
计算机是按照一系列称为**程序**的指令运行的机器。**Android 设备**便是计算机，**应用**是使用 **Java** 语言编写的程序。设备内部是称为**变量**的容器，用于存储数字或文本片段等**值**。

对象是变量，但在以下两个方面特殊。第一，对象中可包含更小的变量，即对象的**域**。例如，表示房屋的对象可能包含一个 **color** 域。第二，我们可向对象附加称为**方法**的一系列指令，实际上是小程序。

house 对象可能具有一个 **setColor** 方法，用于将房屋设置为不同的颜色。

对象分多种**类**（类型）。针对每个类，我们必须编写定义：即属于各个类对象的域和方法列表。每个给定类的对象都有一组相同的域和方法。例如，每个 **house** 对象必须具有称为 **color** 的域和称为 **setColor** 的方法。但是每个 **house** 对象都可在其 **color** 域中包含不同的值：一个房屋可以是红色，另一个房屋为蓝色。

在 **Java** 文件中编写类的定义。由于该定义包含关于该类的所有重要数据，因此图例使用文件表示类。属于类的对象称为类的**实例**。每个实例都具有在类定义中列出的所有域和方法。



Class Name（类名）

定义

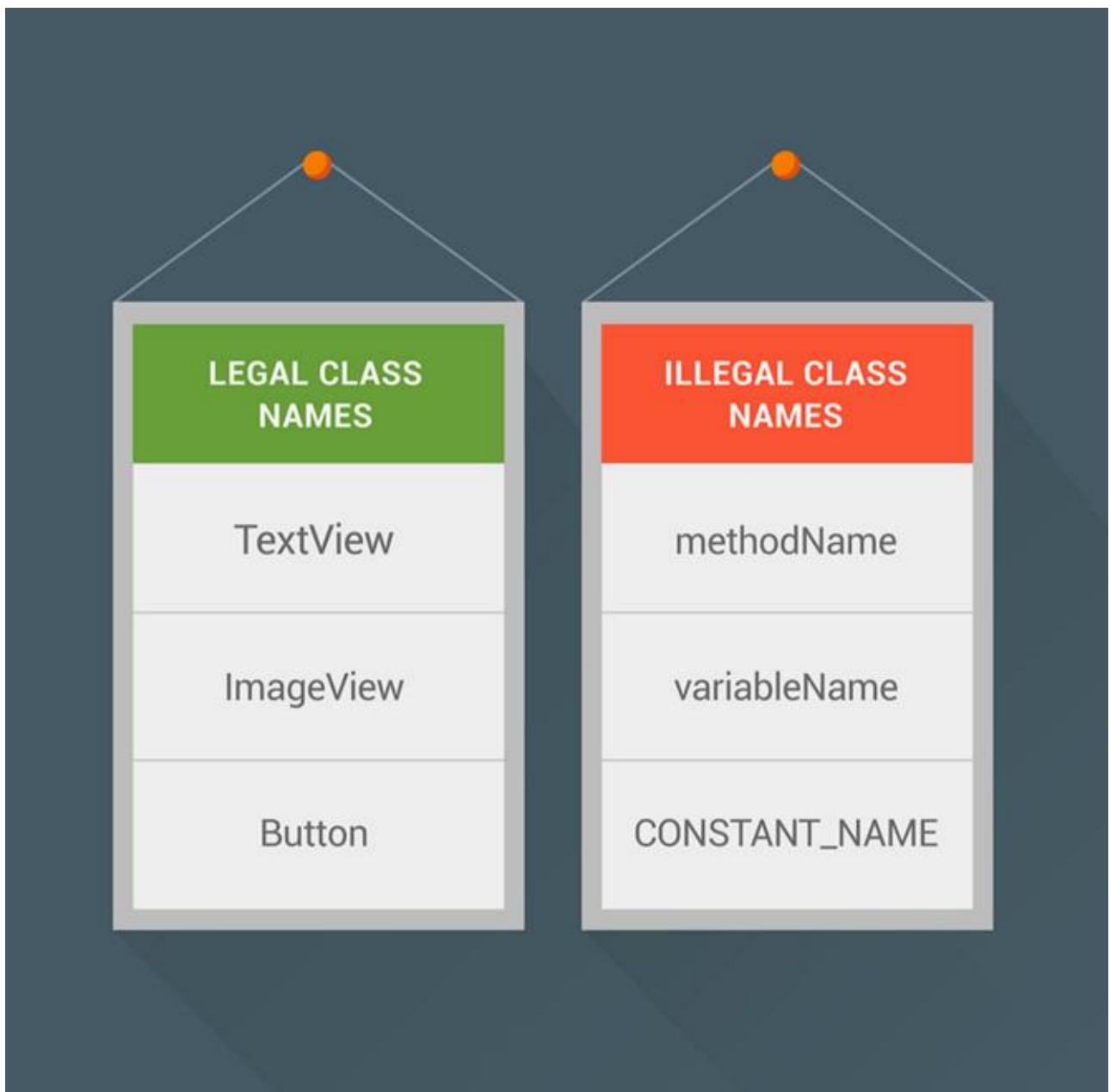
计算机是按照一系列称为**程序**的指令运行的机器。**Android 设备**便是计算机，**应用**是使用 **Java** 语言编写的程序。

计算机内部是称为**变量**的容器，用于存储数字或文本片段等**值**。大变量称为**对象**，其中可包含若干个小变量，小变量是该对象的**域**。还可向对象附加称为**方法**的一系列指令，实际上是小程序。。

对象有多个**类**，给定类的对象都有一组相同的域和方法。例如，**TextView** 类的每个对象都具有用于在屏幕上显示文本的域和方法。与此同时，**ImageView** 类的对象具有用于显示图像的另外一组域和方法。

请注意，同一类的两个对象的域中可能包含不同的值。例如，**TextView** 类的每个对象的域中都存储一段文本。在一个 **TextView** 对象中，此域可能包含 "Hello"；在另一个 **TextView** 中，此域可能包含 "Goodbye"。

在 **Java** 中，**类名**的开头是大写字母，并且为驼峰式大小写，因此很容易识别。



Code（代码）

定义

计算机是按照一系列称为**程序**的指令运行的机器。**Android 设备**便是计算机，**应用**是程序。

由于 Android 设备尚未可靠地理解人类语言，因此必须以较简单的语言（例如 **Java**）编写应用。使用 **Java** 编写的指令称为**代码**，并且能为设备所理解。

但是，采用以 **Java** 描述的细小步骤编写应用可能会相当冗长，因此我们首先用人类语言勾绘出应用的轮廓。这种复述称为**伪代码**，只能为人类所理解。

| PSEUDOCODE | JAVA CODE |
|------------------------------|-------------------------|
| Create a variable named "i". | <code>int i;</code> |
| Put 10 into it. | <code>i = 10;</code> |
| Then subtract 1 from it. | <code>i = i - 1;</code> |

Comment（注释）

定义

计算机是按照一系列称为**程序**的指令运行的机器。**Android 设备**便是计算机，**应用**是程序。

应用必须以编程语言（如 **Java**）编写，因为设备还不能可靠地理解人类语言（如英语）。事实上，如果我们在应用指令中使用英语编写句子，设备将变得极度混乱。

但通常我们又恰恰想这样做，为的是为写给人类的应用提供解释。这个解释称为**注释**，前后必须加**注释分隔符**：即指示设备切勿读取所括文本的标点符号。

在 **Java** 中，以双斜线开始的注释称为**行内注释**，只有阅读 **Java** 文件的人员才会看到该注释。行内注释长度只能占一行。如果希望编写其他注释行，必须另外写入双斜线。

由 **/**** 和 ***/** 括起来的注释称为 **Javadoc** 注释，长度可以占多行。可将应用中的 **Javadoc** 注释收集到文档的单独文件中，以此提供应用的总结。

代码示例

```
public class MainActivity extends AppCompatActivity {  
    /**  
     * Show a message in a TextView.  
     *  
     * @param newMessage The message.  
     * @param resourcelId The resource id of the TextView.  
     * @return The TextView's previous message.  
     */  
    private String showMessage(String newMessage, int resourcelId) {  
        // Create a local variable named textView  
        // that refers to a TextView object created in the file activity_main.xml.  
        TextView textView = (TextView) findViewById(R.id.textView);  
        // Make a copy of the TextView's message,  
        // since the new message will wipe it out.  
        // Must convert the CharSequence returned by getText to a String.
```

```
String oldMessage = textView.getText().toString();  
// Display the new message in the TextView, wiping out the old one.  
textView.setText(newMessage);  
// Return the previous message, in case anyone is still interested in it.  
return oldMessage;  
}  
}
```

Comment



Java file

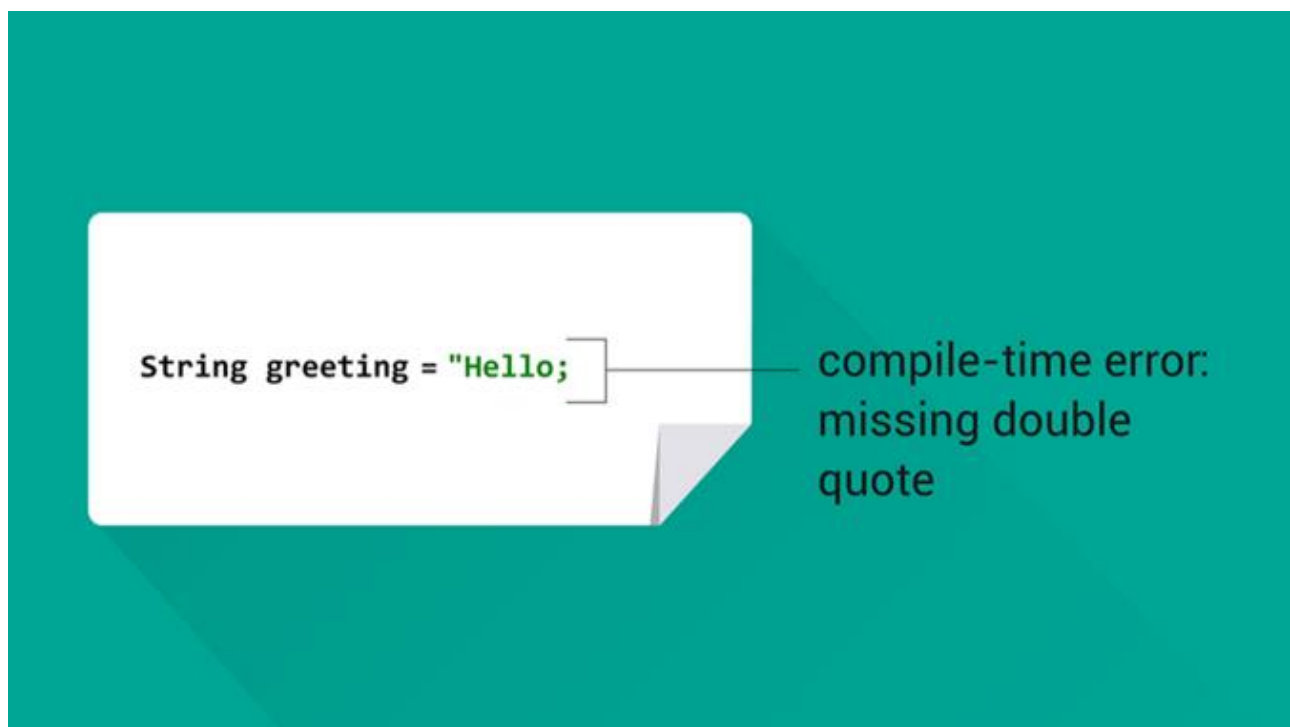
Compile-time Error（编译时错误）

定义

计算机是按照一系列称为**程序**的指令运行的机器。**Android 设备**便是计算机，**应用**是程序。应用采用**Java** 语言编写。

我们喜欢假定 Android 设备可以理解用 **Java** 编写的应用，但事实并非如此。设备只能理解自己的**本机语言**，即 1 和 0，只有在将应用指令翻译成这种语言（或翻译成其他代码）后，设备才能执行这些指令。

这项翻译工作由称为**编译器**的工具执行，该工具由 **Android Studio** 运行。如果编译器尝试将指令**编译**（翻译）成本机语言时在应用中发现任何错误，则编译器将显示**编译时错误消息**，而不生成译文。**编译时**（例如编译过程期间）可检测应用的语法错误 - 语法和标点。但其他错误只能在稍后执行应用时进行检测。



Constructor（构造函数）

定义

计算机是按照一系列称为**程序**的指令运行的机器。**Android 设备**便是计算机，**应用**是使用 **Java** 语言编写的程序。设备内部是称为**变量**的容器，用于存储数字或文本片段等**值**。

对象是变量，但在以下两个方面特殊。第一，对象中可包含更小的变量，即对象的**域**。例如，**TextView** 对象在屏幕上显示文本，并且可能包含称为 **mText** 的域。第二，我们可向对象附加称为**方法**的一系列指令，实际上是小程序。**TextView** 对象可能具有称为 **setText** 的方法，可将文本片段放入对象的 **mText** 域中。

对象分多种**类**（类型）。针对每个类，我们必须编写定义：即属于各个类对象的域和方法列表。每个给定类的对象都有一组相同的域和方法。例如，每个 **TextView** 对象必须具有称为 **mText** 的域和称为 **setText** 的方法。但是每个 **TextView** 对象都可在其 **mText** 域中包含不同的值：一个 **TextView** 可能会说“您好”而另一个说“再见”。

对象的每个类具有称为**构造函数**的方法，可在创建该类的对象时自动执行（且不可避免！）。为使对象可供应用的其余部分使用，构造函数负责执行全部所需操作。例如，构造函数通常将初始化（放入第一个值）正在构建对象的各个字段。

构造函数的名称与其所属类的名称相同。一个类可具有多个构造函数，前提是每个构造函数具有不同的参数列表。第一个代码示例**定义**（创建）不含任何参数的构造函数。

创建对象的唯一方式是**调用**（执行）对象的构造函数。在 **Java** 中，使用第二个代码示例中的特殊命令 **new** 来完成。

代码示例

```
// Simplified definition of class TextView in the file TextView.java,  
// showing two fields and a constructor that initializes them.
```

```
public class TextView extends View {
```

```
// the text to be displayed on the screen
private String mText;
// the color of the text, as an rgb number (red/green/blue)
private int mTextColor;

// This method is a constructor for class TextView.
public TextView() {
    mText = "";
    mTextColor = Color.BLACK;
}
}

// In another .java file, call the constructor to create an object of class TextView.
// Store a reference to the newborn TextView object in the variable textView.
TextView textView = new TextView();
```



Control Flow（控制流）

定义

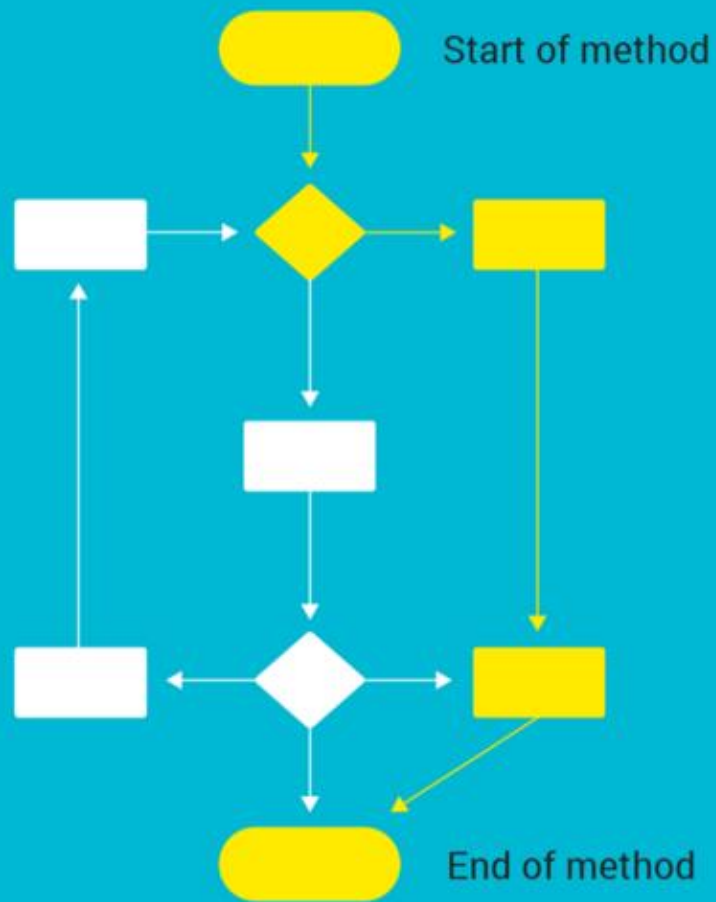
计算机是按照一系列称为**程序**的指令运行的机器。**Android 设备**便是计算机，**应用**是程序。应用划分为多个部分，称为**方法**。每个方法都是相对较短的指令列表。

默认情况下，设备按照指令的编写顺序，自上而下直线执行方法的指令。但某些指令可选择指挥设备跳过其他指令，或者重复其他指令。导致设备偏离直线路径的这些指令，称为方法的**控制结构**。由此产生的路径或可能的路径集，称为方法的**控制流**。

若每次执行应用时均会跳过应用包括的一组指令，则会造成存储浪费。相反，不必要地重复一组指令也会造成时间浪费。变量是包含诸如数字或文本片段之类**值**的容器，控制结构通过检查应用**变量**的当前内容，可做出正确选择。例如，对于存储用户所需购买项目数量的变量，若变量中存储的值为零，则控制结构会指示设备跳过执行方法的“采购”部分。

代码示例

```
// Skip the purchase method if the numberOfItems is 0.  
// Execute the purchase method only if the number of items is not 0.  
// The operator != means “is not equal to”.  
// If the numberOfItems is not equal to 0, we will execute the code in the {curly braces}.  
if (numberOfItems != 0) {  
    purchase();  
}
```



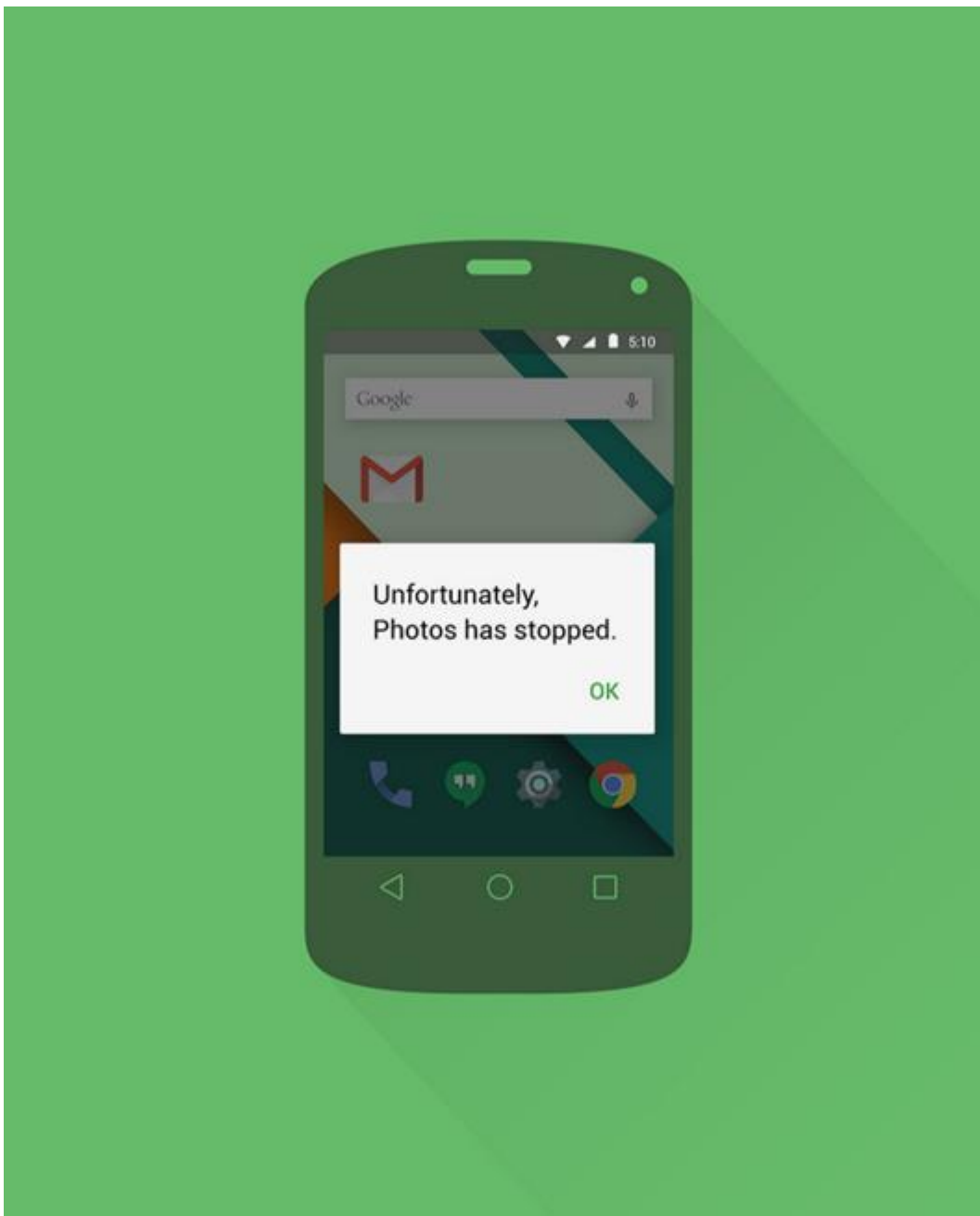
There can be many possibilities for the **control flow** through a method.

Crash（崩溃）

定义

计算机是按照一系列称为**程序**的指令运行的机器。**Android 设备**便是计算机，**应用**是程序。

有时，应用指令不会告知设备应用的发起人意图做什么。这些指令甚至可以令设备进入到一种无法继续执行下一条指令的状态。出现这种情况时，我们称此应用已**崩溃**。



Data Type（数据类型）

定义

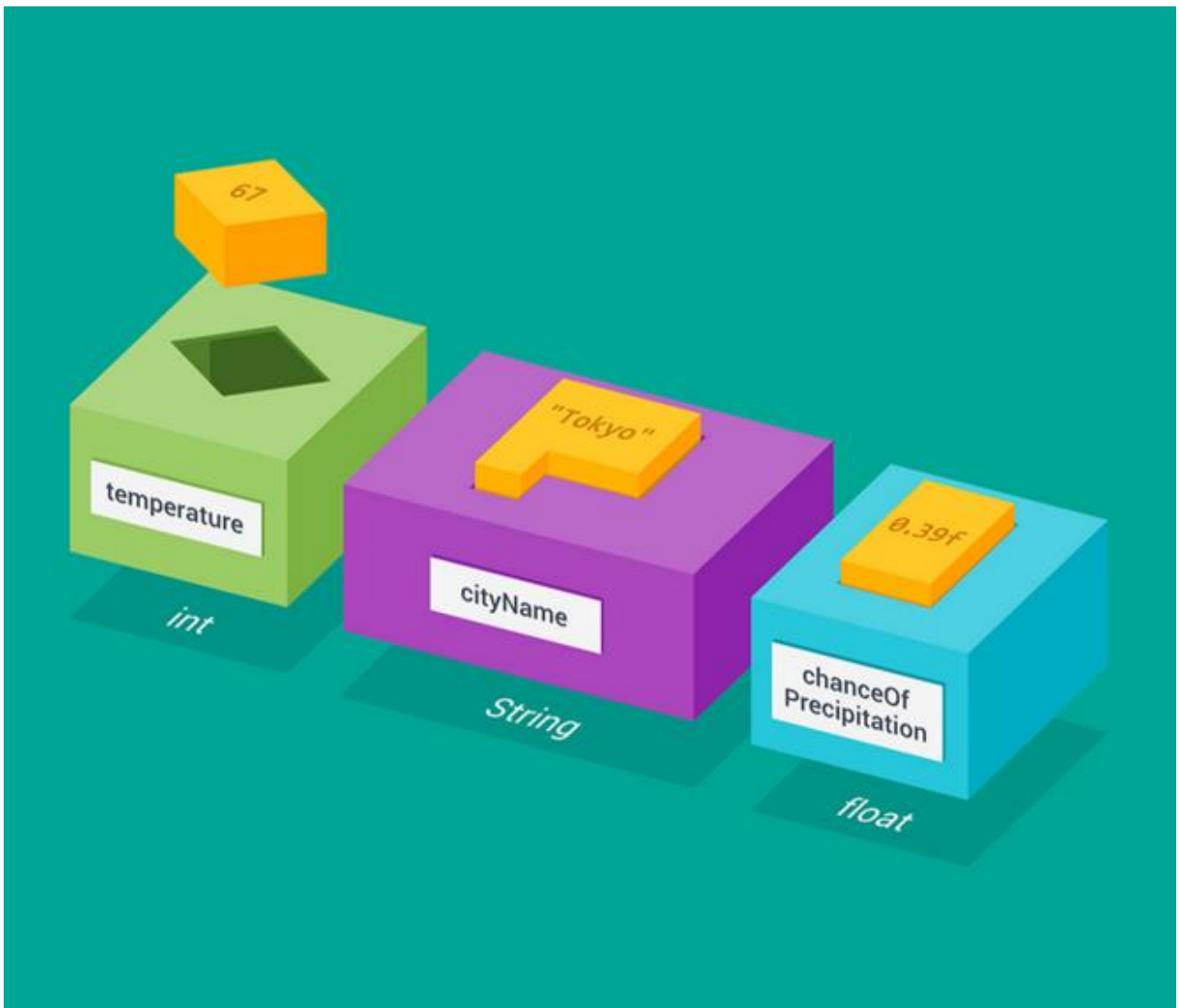
计算机是按照一系列称为**程序**的指令运行的机器。**Android 设备**便是计算机，**应用**是程序。

一个应用可以操纵多种信息类型。其中包括整数、分数和文本片段。在 **Java** 语言中，这些**数据类型**称为**整型**、**浮点型**和**字符串**。其他数据类型还包括**字符型**（单一字符的信息片段）和**布尔型**（值为真或假）。

不同数据类型的值以完全不同的方式存储在计算机中。例如，**字符串**几乎总比整数占用更多的**内存**（存储空间）。为此，我们必须使用不同类型的**变量**（容器）来存储各种类型值。**整型变量**无法包含字符串。

代码示例

```
// Create variable named temperature capable of holding any  
// value of type int. Put 67 into it.  
int temperature = 67;  
  
// Create variable named cityName capable of holding any  
// value of type String. Put "Tokyo" into it.  
String cityName = "Tokyo";  
  
// Create variable named chanceOfPrecipitation capable of  
// holding any value of type float. Put 0.39f into it.  
float chanceOfPrecipitation = 0.39f;
```



Debug（调试）

定义

计算机是按照一系列称为**程序**的指令运行的机器。**Android 设备**便是计算机，**应用**是程序。

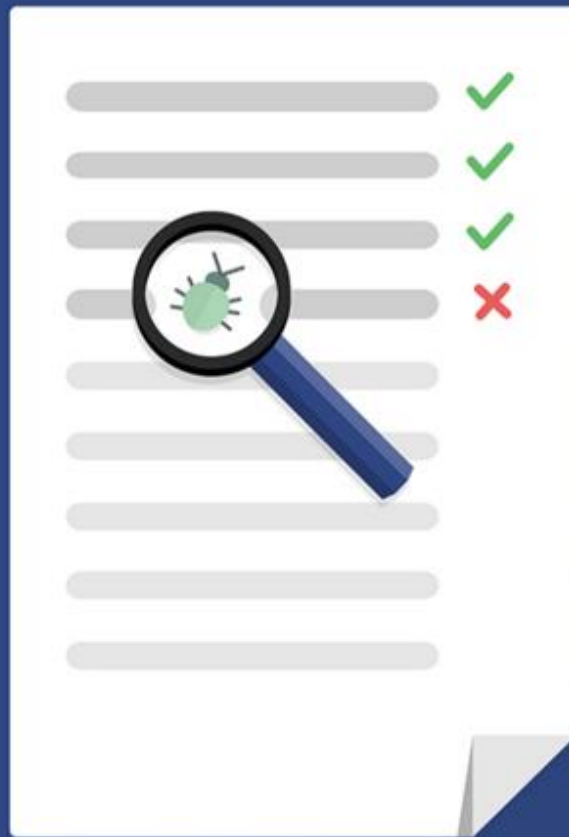
完成应用中的各条指令后，默认情况下，**Android 设备**将进入下一指令。但某些指令会指挥设备跳转到列表中的不同指令。指令还可通知设备更改**变量**内容，变量是存储诸如数字或文本块之类**值**的小型容器。

所以现在我们知道应用有两种方式可以指示设备出错。应用可指示设备偏离正常执行，因为设备是通过指令列表的方式来执行的，或者应用会指示设备为变量赋予不正确的值。这些错误在应用中称为**程序错误**，识别并移除这些错误称为**调试**。

调试应用的一种方式是在应用中插入**调试语句**。这些指令会指示设备显示当前正在执行的指令，并显示变量的内容。在 **Android** 应用中，我们可使用 **Log** 指令作为调试语句。

调试应用的另一种方式是时刻提高警惕，因为设备每次执行一条应用指令。我们可能还想查看变量的内容，并观看设备在每步中更改这些内容。但是有一个问题：设备通常每秒执行上百万条指令。

要将应用速度减慢到百万分之一，甚至完全暂停，则可通过 **Android Studio** 内置的工具（称为**调试器**）进行控制。调试器还可以为我们显示变量内容，甚至允许我们更改那些内容以执行小的测试与实验来找出错误所在。与此同时，出故障应用不会察觉到其正在接受高超的详细检查。



Declare（声明）

定义

计算机是按照一系列称为**程序**的指令运行的机器。**Android 设备**便是计算机，**应用**是程序。

应用中的指令可指示设备创建多种事物类型。最常见的为**变量**，设备中的一种容器。存在许多**类型**的变量，如存储数字或存储文本片段的那些变量。每个变量必须给定一个名称。

声明是应用指令中的语句，用于宣布变量的名称和类型。当我们在其他应用指令中提及变量名称时，为了使设备理解我们正在谈论的内容，所以我们需要编写声明。

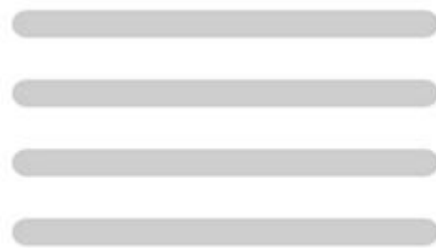
除变量外，我们也可以声明许多其他事物。**方法**是供设备执行的短指令列表 - 即小程序。方法声明是用于宣布方法名称和类型的语句。而**类**是一种特殊类型的变量。类的声明同样是语句，用于宣布类的名称、变量以及附加到此类型各个变量的方法。

代码示例

```
// This declaration announces the existence of a variable named firstName, of type TextView.  
TextView firstName;  
  
// Now that we have declared the variable textView, we can go ahead and use it.  
// For example, we can assign a value to it (store a value into it).  
firstName = (TextView)findViewById(R.id.firstName);  
  
// But the following statement will give us an error message because we have never declared any variable named lastName.  
lastName = (TextView)findViewById(R.id.lastName);
```



```
TextView firstName;
```



firstName **can't** be
mentioned in these
instructions

Declaration

firstName **can** be
mentioned in these
instructions

Define a Method（定义方法）

定义

计算机是按照一系列称为**程序**的指令运行的机器。**Android 设备**便是计算机，**应用**是程序。**Android** 应用采用 **Java** 语言编写。设备内部是称为**变量**的容器，用于存储数字或文本片段等**值**。

对象是变量，但在以下两个方面特殊。第一，对象中可包含更小的变量，即对象的**域**。例如，在 **MediaPlayer** 对象中可能包含多个域，用于存储正在播放的声音文件的名称、音量等级、文件回放的当前位置以及指示是否采用无限循环方式播放文件。

第二，我们可向对象附加称为**方法**的一系列指令，实际上是小程序。例如，我们的 **MediaPlayer** 对象可能具有 **play**、**pause** 和 **stop** 方法。对象的方法可使用对象内部这些方法所附加到的域。例如，**play** 方法需要使用全部四个域。当我们指示设备执行方法时，我们称我们正在调用方法。

对象有多个**类**，给定类的对象都有一组相同的域和方法。例如，**MediaPlayer** 类的每个对象具有上面讨论的域和方法。与此同时，**TextView** 类的对象具有用于在屏幕上显示文本的不同域和方法集。

对象的各个类均具有**定义**：即充当类对象蓝图的描述。类的定义在应用中编写并列出于属于各个类对象的域，指定各个域的名称和数据类型。还列出属于各个对象的方法，指定各方法的名称及构成方法的指令列表。此**方法定义**还指定方法在**参数**、**返回值**的名称及数据类型：用于存储调用方法时在方法间所传递信息的变量。我们编写方法定义时，我们称我们在**定义方法**。


```
// This the definition of class MediaPlayer,  
// with some parts omitted.  
// Although the class has many methods,  
// we show the definition of only one of them.
```

```
public class MediaPlayer {
```

```
    // This is the definition of one of the methods of class  
    // MediaPlayer.  
    public void pause() {  
        // The instructions that constitute the method go here.  
    }
```

```
}
```

Documentation（文档）

定义

计算机是按照一系列称为**程序**的指令运行的机器。**Android 设备**便是计算机，**应用**是程序。应用采用**编程语言**编写（例如 **Java**），并由设备读取和执行。

文档是设备和应用的相关信息。文档以人类语言编写（例如，英语），供人类阅读。

应用的各部分相关文档可使用应用指令编写，前后加**注释分隔符**：即指示设备忽略所括文本的标点符号。应用的整体相关文档可在单独文档中编写，存储在网站上或打印为手册。

应用的相关文档可能描述应用目的、用户指令、设备的版本以前可运行此应用的操作系统，也可能为许可证。有关如何编写应用的文档位于 [Android 网站](#)。

dp (Density-Independent Pixel)（与密度无关的像素）

定义

Android 设备的屏幕由称为**像素**的发光点行和列构成。设备可根据**屏幕密度**移动，即屏幕上的每英寸像素数（或点/英寸）。例如，mdpi（或中等密度设备）具有 160 点/英寸，而 xxhdpi（超高密度设备）具有 480 点/英寸。

如果我们以像素值指定视图大小，则视图在较高密度设备上将显得很小时，也就是将许多像素装填到较小区域内。如果按钮过小，则用户触摸将比较困难。

为在不同屏幕密度的设备间实现一致物理大小的视图，我们使用称为**与密度无关的像素**的度量单位（**dp** 或 **dip**，发音为 "dee pee" 或 "dip"）。在 mdpi 设备上，1 dp 等于 1 像素。在 xxhdpi 设备上，1 dp 等于 3 像素，**其他设备**以此类推。按照材料设计指南，屏幕上的任何触摸目标均应**至少为 48dp 宽乘以 48dp 高**。这样，一台设备上的应用按钮将与使用不同屏幕密度在设备上运行的相同应用中的按钮大致具有相同物理大小。

Android 设备将自动处理从 dp 到像素值的转换，因此开发人员在其布局中指定尺寸时使用 dp 值即可。例如，dp 值可用于指定在插图中显示视图的宽度和高度。

代码示例

```
<TextView
    android:layout_width="160dp"
    android:layout_height="80dp"
    android:background="#00FF00"
    android:text="Hello"/>
```



Encapsulation（封装）

定义

计算机是按照一系列称为**程序**的指令运行的机器。**Android 设备**便是计算机，**应用**是程序。设备内部是称为**变量**的容器，用于存储数字或文本片段等**值**。

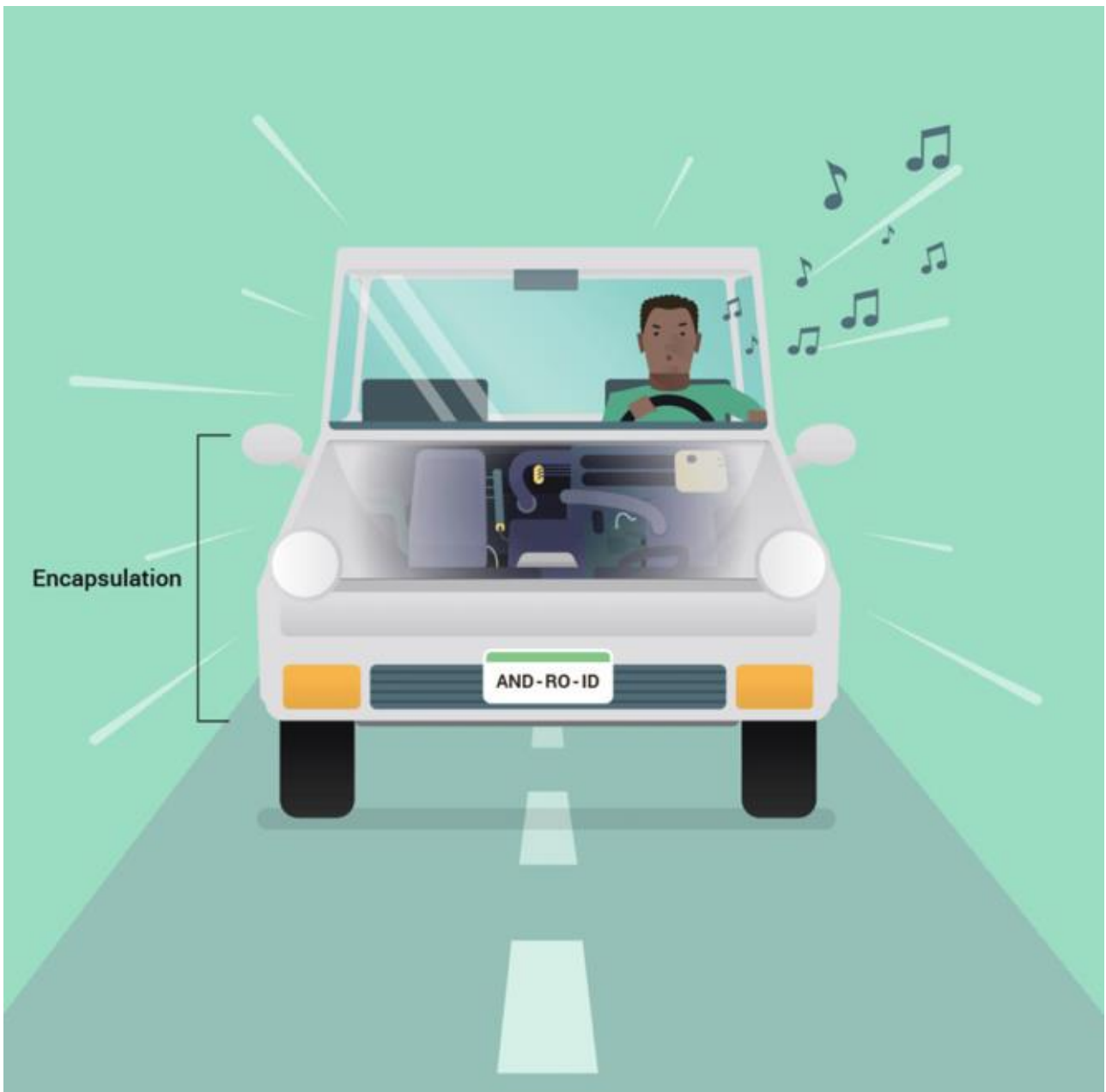
对象是变量，但在以下两个方面特殊。第一，对象中可包含更小的变量，即对象的**域**。例如，表示汽车的对象可能包含称为 **color** 的域。第二，我们可向对象附加称为**方法**的一系列指令，实际上是小程序。**car** 对象可能具有称为 **setColor** 的方法，用于将汽车颜色设置为给定颜色，还可以具有称为 **getColor** 的方法，用于**返回**（给我们）汽车的当前颜色。用于设置和获取域值的一对方法称为 **getter** 和 **setter**。

对象分多种**类**（类型）。针对每个类，我们必须编写**定义**：即属于各个类对象的域和方法列表。每个给定类的对象都有一组相同的域和方法。例如，每个 **Car** 类对象必须具有称为 **color** 的域和称为 **setColor** 的方法。但是每个 **Car** 对象可在其 **color** 域中包含不同的值：一台汽车为红色，而另一台为蓝色。

可以指定对象类各个域和方法的**访问修饰符**。例如，可以将 **color** 域设为 **private**，即除 **Car** 类的方法外，任何其他类的方法都不能访问。可以将字段的 **getter** 和 **setter**（**getColor** 和 **setColor** 方法）设为 **public**，即应用中任意类的方法都可以访问。这样一来，如果 **Car** 类中再无其他可以获取和设置域的公共方法，则域已**封装**，即只能通过 **getter** 和 **setter** 进行访问。

封装域可使对象更易于使用。仅通过调用 **getColor** 和 **setColor** 方法我们便可以获取并设置 **Car** 对象的颜色，而无需了解颜色实际上是如何存储在对象内部的。（颜色可能存储为单一数字，或存储为三个数字的组合 - 红色、绿色和蓝色，或以其他格式存储。）汽车可以在无需了解汽车内部安排详细信息的情况下进行驱动，已封装域的对象同样可以在无需了解这些域详细信息的情况下使用。

封装域还会使应用更易于调试。如果已将域设置为错误的值，则只能是该域的 **setter** 所实现的。这一点为我们确定了搜索的目标范围。我们可检查应用中指示设备执行此 **setter** 的每一条指令，以此带我们找出程序错误。



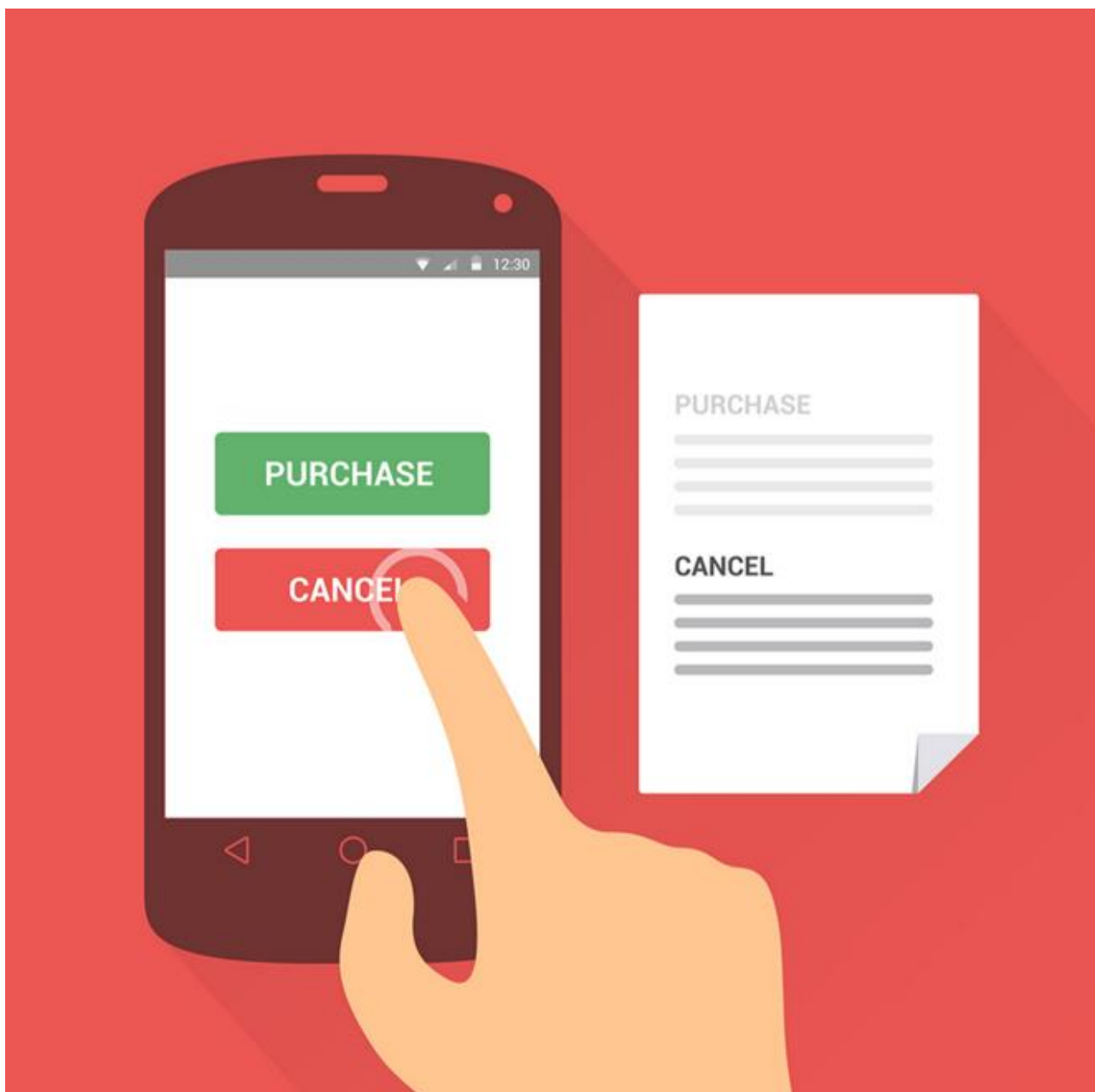
Event-Driven Programming（事件驱动编程）

定义

计算机是按照一系列称为**程序**的指令运行的机器。**Android 设备**便是计算机，**应用**是程序。

如果此应用为**顺序**程序，则设备在应用的第一条指令处启动，并按照编写指令的顺序或按照指令本身指示的顺序来执行指令。由于已编写指令，所以我们可提前预测设备的操作。

如果应用为**事件驱动**程序，则可将其划分为多个部分，并且在收到来自外部世界的刺激时，设备会从一个部分跳转到另一个部分。例如，当手指触摸按钮时，我们可跳转到执行按钮建议操作的应用部分。在事件驱动编程中，无法提前预测设备的操作，因为操作取决于用户的想法。



Execute（执行）

定义

计算机是按照一系列称为**程序**的指令运行的机器。**Android 设备**便是计算机，**应用**是程序。

默认情况下，设备将按照应用中指令的编写顺序**执行**指令。但某些指令会指示设备向前跳转或返回到早期指令。此外，诸如用户在屏幕上单击按钮等事件可导致设备跳转到完全不同的应用指令部分。



Expression（表达式）

定义

计算机是按照一系列称为**程序**的指令运行的机器。**Android 设备**便是计算机，**应用**是程序。

在应用的指令中，可以写入**文字**（如 10 或 "John"）来表示数字或文本片段的**值**。还可以写入**变量**的名称，变量是在设备内部用于存储值的容器。通常变量名称可以是 "x"、"y" 或 "greeting"。

通过写入如下所示包含文本和变量的**表达式**，可指示设备操纵数字或文本。

$$x + y$$
$$x - 10$$
$$\text{greeting} + ", \text{John}!"$$

其中，加号和减号称为**操作符**，所操作的值称为**操作数**。通过操作符计算出的结果称为**表达式的值**。

通过操作符可将简单表达式连接到一起，用多个简单表达式构建复杂表达式。例如，使用表达式

$$a * b$$

和

$$c * d$$

均包含乘法操作符 $*$ ，可另外使用 $+$ 操作符将二者连接起来，构成较为复杂的表达式。

$$a * b + c * d$$



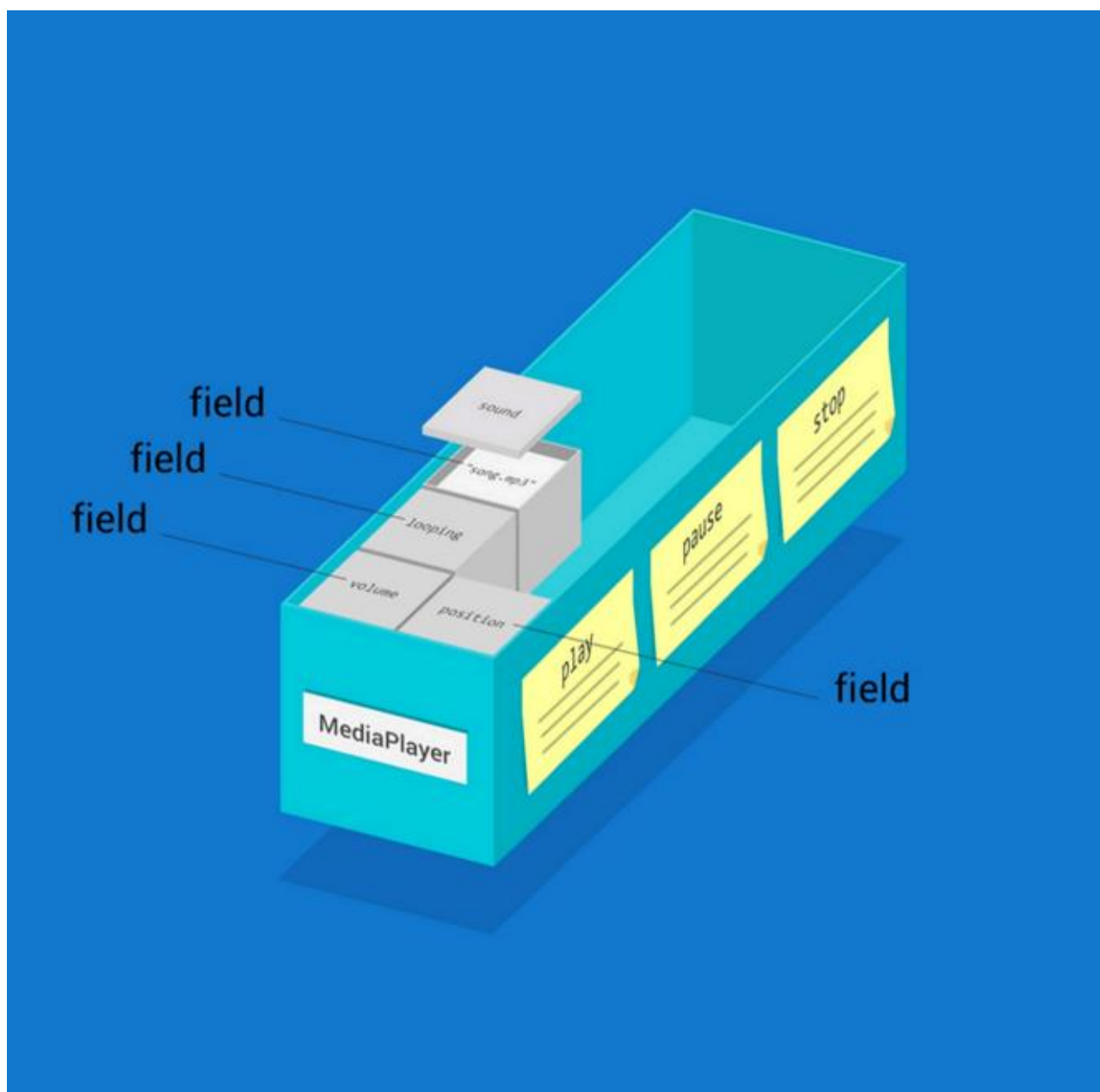
Field（域）

定义

计算机是按照一系列称为**程序**的指令运行的机器。**Android 设备**便是计算机，**应用**是程序。设备内部是称为**变量**的容器，用于存储数字或文本片段等**值**。

对象是变量，但在以下两个方面特殊。第一，对象中可包含更小的变量，即对象的**域**。例如，在 **MediaPlayer** 对象中可能包含多个域，用于存储正在播放的声音文件的名称、音量等级、文件回放的当前位置以及指示是否采用无限循环方式播放文件。第二，我们可向对象附加称为**方法**的一系列指令，实际上是小程序。我们的 **MediaPlayer** 对象可能具有 **play**、**pause** 和 **stop** 方法。

对象域在创建对象时创建，在销毁对象时销毁。在对象的生命期中，对象方法中的指令可使用域。



findViewById

定义

计算机是按照一系列称为**程序**的指令运行的机器。**Android 设备**便是计算机，应用是使用 **Java** 语言编写的程序。设备内部是称为**变量**的容器，用于存储数字或文本片段等**值**。**对象**是一种较大变量，其中可包含较小变量。我们可向对象附加称为**方法**的一系列指令，实际上是小程序。

对象分多种**类**（类型）。例如，**ImageView** 类的对象可在屏幕上显示图像，**TextView** 类对象可显示文本片段。通常使用 **XML** 语言编写**布局文件**来创建这些**视图对象**，用于描述对象以及对象在屏幕中的位置。

给定类的每个对象都附有相同的方法集。例如，**活动**类的每个对象都具有创建**用户界面**对象的方法：即 **ImageView**、**TextView**、**Button** 等用户能够在屏幕上看到的内容。

创建视图对象后，还需要使用活动对象分别对这些对象进行配置。这就是每个视图对象都有一个 **ID 编号** 的原因。这些编号使得应用中使用 **Java** 编写的部分与使用 **XML** 编写的部分之间能够相互通讯。具体地说，利用这些编号，使用 **Java** 创建的对象方法可以调用使用 **XML** 创建的对象方法。

各视图的 **ID 编号** 存储在可传递至活动方法的 **Java** 变量中。该变量通过在布局文件的视图对象描述中写入变量名称进行创建。例如，在代码示例中创建名为 **R.id.today** 的变量，用于存储 **TextView** 的 **ID 编号**。通过在布局文件的 **TextView** 描述中写入属性 **android:id="@id+/today"**，可完成创建。（**R** 代表“资源”。）

每个活动都有一个名为 **findViewById** 的方法，用于查找已给出其 **ID 编号** 的视图对象。代码示例会将变量 **R.id.today** 传递给此方法，用于查找 **TextView**。如果运行正确，**返回值**（方法生成的结果）将引用 **TextView**，即允许活动调用 **TextView** 方法的一条信息片段。为方便使用此引用，必须将其存储于只能存储“引用 **TextView**”这一种类型值的特殊用途变量中。

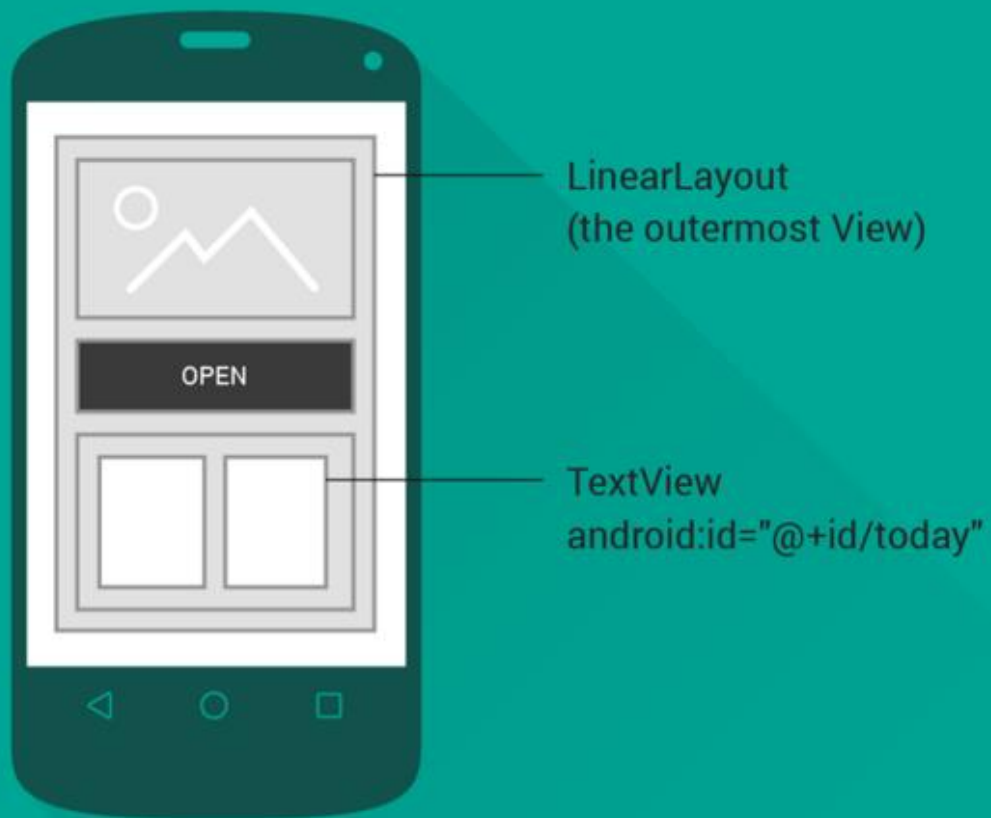
由于调用 **findViewById** 可查找许多不同类（**TextView**、**ImageView** 等）的视图对象，因此其返回值为常规用途引用，可能引用这些类中任何一个的对象。必须将返回值从常规**转换**（转变）至特殊后，才能将该返回值存储到特殊用途变量中。此方向的转换称为**向下转换**，由括号中的命令 **TextView** 执行。

另一项要求是，**findViewById** 必须在执行 **setContentView** 后执行，该方法用于创建 **TextView** 及布局文件中描述的其他视图对象。**JavaScript** 的程序员会发现 **Android findViewById** 与 **JavaScript getElementById** 相类似。

代码示例

```
<!-- Excerpt from the layout file activity_main.xml. -->
<TextView
    android:id="@+id/today"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"/>

// Excerpt from the file MainActivity.java.
// When the app is launched, put the current date (but not the time) into the TextView.
@Override
protected void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);
    setContentView(R.layout.activity_main);
    TextView today = (TextView) findViewById(R.id.today);
    if (today != null) {
        // The Date object also contains the current time.
        Date todaysDate = new Date();
        DateFormat justTheDate = DateFormat.getDateInstance();
        String s = justTheDate.format(todaysDate);
        today.setText(s);
    }
}
```



`findViewById` begins its search at the outermost View and works its way inwards.

Getter Method（Getter 方法）

定义

计算机是按照一系列称为**程序**的指令运行的机器。**Android 设备**便是计算机，**应用**是程序。设备内部是称为**变量**的容器，用于存储数字或文本片段等**值**。

对象是变量，但在以下两个方面特殊。第一，对象中可包含更小的变量，即对象的**域**。例如，表示房屋的对象可能包含一个 **color** 域。第二，我们可向对象附加称为**方法**的一系列指令，实际上是小程序。

House 对象中可具有 **setColor** 方法，用于将房屋设置为给定颜色，还可以具有 **getColor** 方法，用于返回（至开发人员）房屋的当前颜色。用于设置和获取域值的一对方法称为 **getter** 和 **setter**。

对象分多种**类**（类型）。针对每个类，我们必须编写**定义**：即属于各个类对象的域和方法列表。每个给定类的对象都有一组相同的域和方法。例如，每个 **house** 对象必须具有称为 **color** 的域和称为 **setColor** 的方法。但是每个 **house** 对象都可在其 **color** 域中包含不同的值：可以一个房屋为红色，而另一个为蓝色。

可以指定对象类各个域和方法的**访问修饰符**。例如，可以将颜色字段设为 **private**，即除 **House** 类的方法外，任何其他类的方法都不能引用。可以将域的 **getter** 和 **setter**（**getColor** 和 **setColor** 方法）设为 **public**，即应用中任意类的方法都可以引用。这样一来，如果 **House** 类中再无其他可以获取和设置字段的 **public** 方法，则字段已**封装**，即只能通过 **getter** 和 **setter** 进行访问。

Gist

定义

计算机是按照一系列称为**程序**的指令运行的机器。**Android 设备**便是计算机，**应用**是程序。

代码段是应用引用的自包含小片段，演示指示设备执行操作的方法。通过将代码段粘贴到自己的一个应用中，然后视需要进行细微的更改，便可**重用**代码段。

共享代码段的一种方式是将其发布到 GitHub 网站 <https://gist.github.com/>。共享的代码段称为 **gist**。之后便可以将该链接分享给朋友。

代码示例

```
// This gist shows how to display a message  
// in a TextView created in an XML layout file.  
TextView textView = (TextView) findViewById(R.id.textView);  
TextView.setText("Hello");
```




Global Variable（全局变量）

定义

计算机是执行一系列称为**程序**的指令的机器。**Android 设备**便是计算机，**应用**是程序。

变量是用于存储数字或文本片段等**值**的容器。大变量称为**对象**，其中可包含若干个小变量，小变量是该对象的**域**。我们可向对象附加称为**方法**的一系列指令，实际上是小程序。

对象分多种**类**。针对每个类，我们必须编写**定义**：即属于类对象的域和方法列表。

局部变量是由对象方法中的指令创建的变量。变量的名称只能在该方法的指令中提及，且只在设备执行该方法时变量才存在。尽管局部变量如上所述依赖于对象的方法之一，但实际上却并不存储在对象或方法中。而是存储在称为**堆栈**的单独内存区域中。

全局变量是对象的域，因此存储在对象中。全局变量在创建对象时创建，在销毁对象时销毁。对象的任何方法都可以提及该变量的名称，也就是说所有这些方法都可以访问变量中包含的值。

只要对象存在，就必须牢记使用全局变量存储的信息。**Android** 惯例是以小写字母 **m** 为开头命名全局变量，其中 **m** 代表**成员**，也是“域”的另一种说法。

代码示例

```
// This is the definition for a class of objects named MainActivity. Each object of this class
// contains a field named mTag and has two methods named onCreate and onDestroy.
public class MainActivity extends AppCompatActivity {
    // The following variable mTag is global because it was not created within the
    // curly braces of a method.
    String mTag = "MainActivity";
    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_main);
        // The following instruction declares (creates) a variable named textView. The variable
```

*// is local because it was declared within the curly braces of a method. The variable's
// scope (the set of instructions where its name can be mentioned) extends from its
// declaration to the closing curly brace that ends the method.*

```
TextView textView = (TextView) findViewById(R.id.textView);  
textView.setText("Hello");  
Log.d(mTag, "onCreate");  
}  
@Override  
protected void onDestroy() {  
    super.onDestroy();  
    Log.d(mTag, "onDestroy");  
}  
}
```

Scope
of global
variable

```
MyClass {
```

```
    _____  
    _____  
    _____  
    _____
```

```
    myMethod {
```

```
        _____  
        _____  
        _____  
        _____
```

```
    }
```

```
    _____  
    _____  
    _____  
    _____
```

```
}
```

Scope
of local
variable

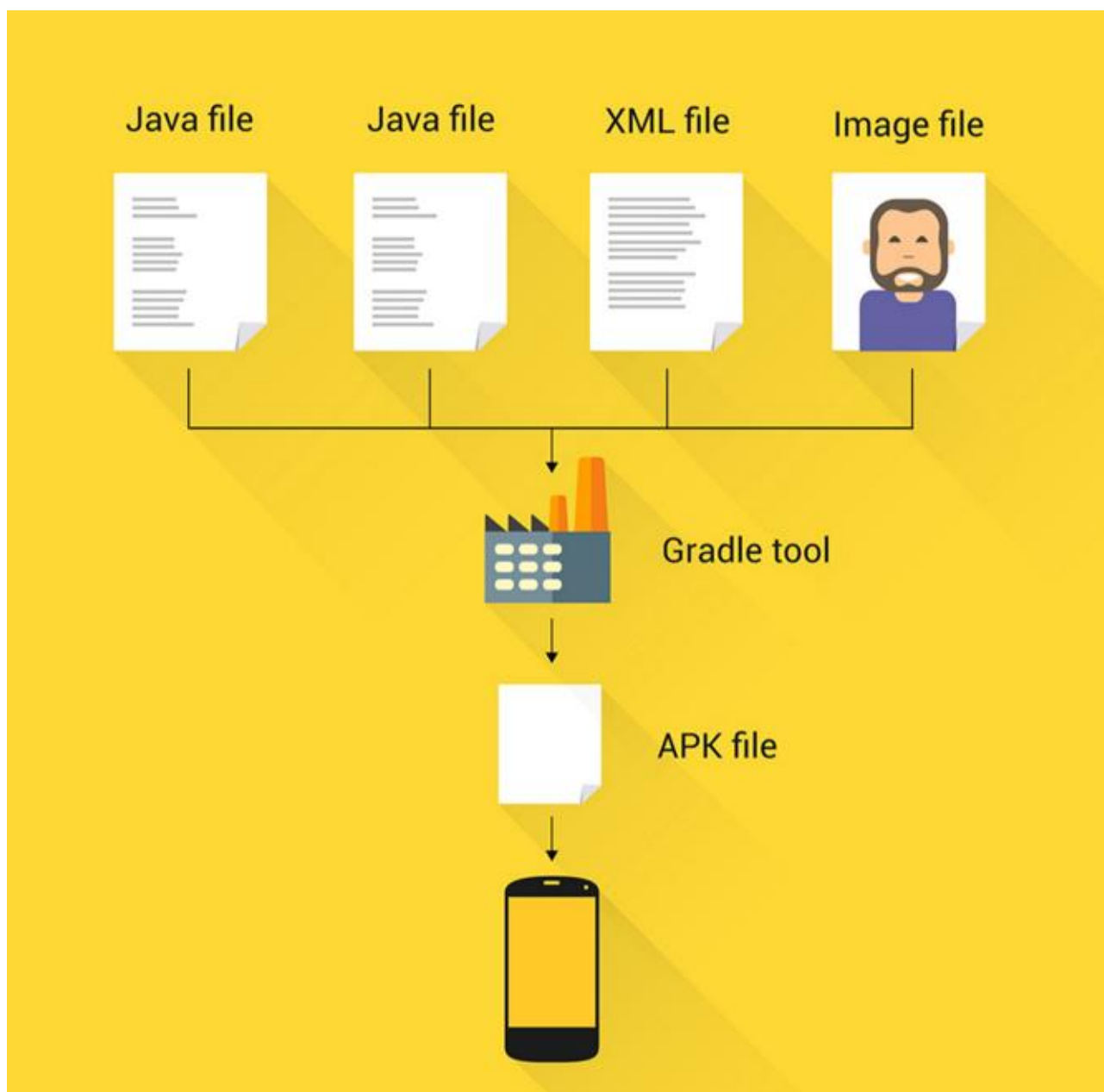
Gradle

定义

计算机是按照一系列称为**程序**的指令运行的机器。**Android 设备**便是计算机，**应用**是程序。

即使是最简单的应用也包括许多指令文件，以及配套**资源**，如图像文件。我们使用称为 **Android Studio** 桌面应用程序创建这些文件和资源。

创建完成后，需要将指令和资源以 **Android 设备**能够理解的形式打包。**Android Studio** 会委派称为 **Gradle** 的工具完成此任务。**Gradle** 将指令翻译成易于 **Android 设备**理解的语言，并将翻译的指令和资源压缩到称为 **APK** 的文件中或 **Android 包**中。之后便可以将 **APK** 复制到 **Android 设备**并在其中运行。



Hardcode（硬编码）

定义

计算机是按照一系列称为**程序**的指令运行的机器。**Android 设备**便是计算机，**应用**是程序。

向应用提供信息的方式之一是将信息写入应用指令中：add 10 + 20，或使 `TextView` 的宽为 100dp，这称为向应用中**硬编码**信息。

上述指令中的 **TextView** 是**视图**示例，是屏幕上可显示信息的矩形区域。视图可以包含在较大视图中，这个大视图称为其**父视图**。假设包含 `TextView` 的父视图宽度为 100dp，若希望 `TextView` 与父视图等宽，则可以写入上述指令。将 100dp 值用硬编码写入指令的缺点之一是，如果想要更改 `TextView` 父视图的宽度则必须要重新写入该值。

这是我们避免写入**硬编码值**的原因之一。如果能够写入一条指令，指示 `TextView` 自动从其父视图获取宽度，就可以减少我们的记忆负担。减少维护也就意味着减少程序错误。

代码示例

```
<!-- This TextView has its width hardcoded into it. -->
<LinearLayout
    android:layout_width="100dp"
    android:layout_height="wrap_content"
    android:orientation="vertical">
    <TextView
        android:layout_width="100dp"
        android:layout_height="wrap_content"
        android:text="Hello"/>
</LinearLayout>

<!-- This TextView gets its width from its parent. -->
<LinearLayout
    android:layout_width="100dp"
    android:layout_height="wrap_content"
    android:orientation="vertical">
```

```
<TextView  
    android:layout_width="match_parent"  
    android:layout_height="wrap_content"  
    android:text="Hello World"/>  
</LinearLayout>
```

`android:layout_width="100dp"`

hardcoded width
value

`android:layout_width="match_parent"`

a width value that
comes automatically
from this View's parent

Hexadecimal Color (Hex Color)（十六进制颜色）

定义

通过按红、绿、蓝顺序将颜色混合到一起可创建各种颜色。写入井号 (#)，然后使用一对“十六进制数字”指定各成分的量。00 是最小量，FF 是最大量，80 是中间量。

使用以上三个数值创建颜色后，请访问[材料设计网站](#)体验更多微妙的调色。

代码示例

```
<LinearLayout
    android:layout_width="match_parent"
    android:layout_height="wrap_content"
    android:orientation="vertical">
    <TextView
        android:layout_width="match_parent"
        android:layout_height="wrap_content"
        android:background="#FF0000"
        android:text="FF0000 Red"/>
    <TextView
        android:layout_width="match_parent"
        android:layout_height="wrap_content"
        android:background="#FF8000"
        android:text="FF8000 Orange"/>
    <TextView
        android:layout_width="match_parent"
        android:layout_height="wrap_content"
        android:background="#FFFF00"
        android:text="FFFF00 Yellow"/>
    <TextView
        android:layout_width="match_parent"
        android:layout_height="wrap_content"
```



```

        android:background="#00FF00"
        android:text="00FF00 Green"/>
<TextView
    android:layout_width="match_parent"
    android:layout_height="wrap_content"
    android:background="#0000FF"
    android:textColor="#FFFFFF"
    android:text="0000FF Blue"/>
<TextView
    android:layout_width="match_parent"
    android:layout_height="wrap_content"
    android:background="#FF00FF"
    android:text="FF00FF Purple"/>
<TextView
    android:layout_width="match_parent"
    android:layout_height="wrap_content"
    android:background="#FFFFFF"
    android:text="FFFFFF White"/>
<TextView
    android:layout_width="match_parent"
    android:layout_height="wrap_content"
    android:background="#808080"
    android:text="808080 Gray"/>
<TextView
    android:layout_width="match_parent"
    android:layout_height="wrap_content"
    android:background="#000000"
    android:textColor="#FFFFFF"
    android:text="000000 Black"/>
</LinearLayout>

```



if/else Statement（if/else 语句）

定义

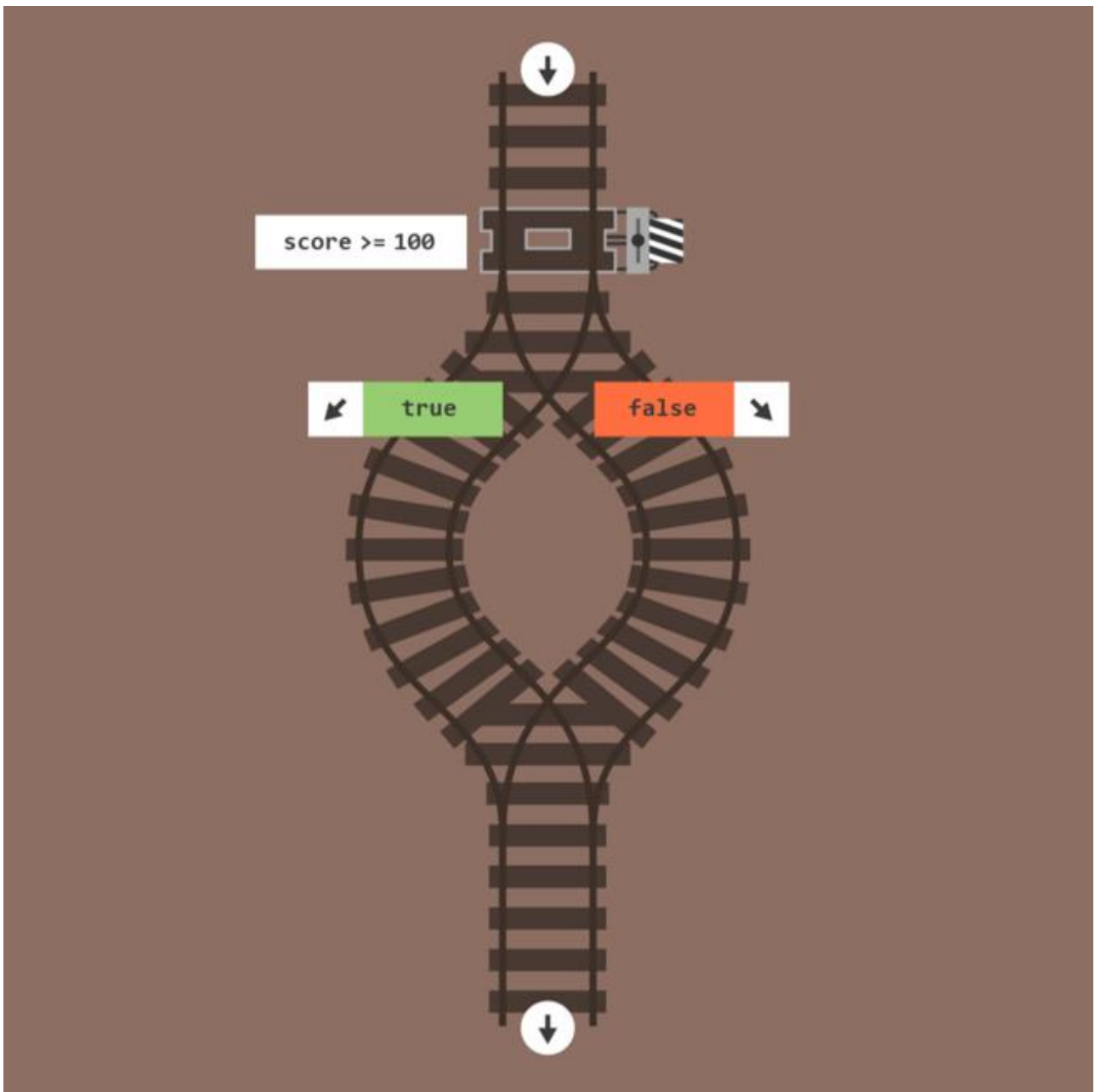
计算机是按照一系列称为**程序**的指令运行的机器。**Android 设备**便是计算机，**应用**是程序。应用划分为多个部分，称为方法。每个方法都是相对较短的指令列表。

默认情况下，设备按照指令的编写顺序，自上而下执行方法的指令。但通过称为 **if 语句** 的特殊指令类型，可选择指示设备跳过一组指令。**if/else 语句** 与其相似，可以指挥设备在两组备选指令中选择其中一组执行。

变量是包含诸如数字或文本片段之类**值**的容器，**if** 和 **if/else** 语句通过检查应用**变量**的内容可做出选择。如代码示例所示，**if/else** 语句根据变量 **score** 中存储的值，控制设备在两种可能方向中选择一种继续。

代码示例

```
if (score >= 100) {  
    showYouWon();  
} else {  
    showTryAgain();  
}
```



ImageView

定义

视图是屏幕上的矩形区域。有一种视图是 **ImageView**，用于显示图标或照片等图像。

屏幕上的 **ImageView** 由 Android 设备内部的 Java 对象绘制。事实上，Java 对象是真正的 **ImageView**。但是在谈到用户所看到的内容时，将屏幕上的矩形区域视为 "ImageView" 将更为方便。

代码示例

```
<ImageView  
    android:layout_width="match_parent"  
    android:layout_height="match_parent"  
    android:src="@drawable/cake"  
    android:scaleType="centerCrop"/>
```



Import Statement（Import 语句）

定义

计算机是按照一系列称为**程序**的指令运行的机器。Android 设备便是计算机，应用是使用 **Java** 语言编写的程序。Android 设备内部是称为**变量**的容器，用于存储数字或文本片段等值。

能够包含小变量的大变量称为**对象**。还可向对象附加称为**方法**的一系列指令，实际上是小程序。。对象分多种不同的**类**，每一类都有各自的名称。

要在应用文件中引用某类对象的名称，必须先在该文件顶部的 **import 语句**中声明该名称。声明中必须包含类的**完全限定名称**（即全名）。由于 **import** 语句非常复杂且一般数量较多，因此我们使用桌面应用程序 Android Studio 来自动包括 **import** 语句。

代码示例

```
// The following import statement allows this file to mention a class of objects  
// named TextView. The full name of the class is android.widget.TextView  
// because the class belongs to a package (family) of classes named "widget".  
// A widget is something that we can see on the screen and that is often  
// sensitive to touch. The widget package belongs to a larger packaged named  
// "android".  
import android.widget.TextView;  
  
// Later in the file, we can mention the word TextView.  
// For example,  
// Create a variable named firstName of class TextView.  
TextView firstName;
```

```
import android.widget.TextView;
```

Import statement

Thanks to the
import statement,
we can mention
class **TextView** here.

Inflate

定义

XML 代表“可扩展标记语言”。它是一种表示法，用于编写以层次结构或家族树形式组织的信息。

可在 XML 文件中列出我们想要创建并显示在屏幕上的 **Java** 对象：**TextView**、**ImageView** 和 **Button**。

但该文件实际上并不 *包含* 这些对象。只是用于 *描述* 对象及其在屏幕中相对位置的文本文件。

创建 XML 文件中所描述对象的过程称为 **inflation**。要对文件使用 **inflate** 方法，请将**资源 ID**（标识号）传递到 **inflator** 对象，读取文件并创建文件中描述的对象。

通常，我们没有将标识号直接传递到 **inflator**，而是将其传递到名为 **setContentView** 的 **Activity** 方法。此方法将完成两项工作：代替我们将编号传递到 **inflator**，将产生的结果显示在屏幕上。



Initialize（初始化）

定义

计算机是按照一系列称为**程序**的指令运行的机器。**Android 设备**便是计算机，**应用**是程序。

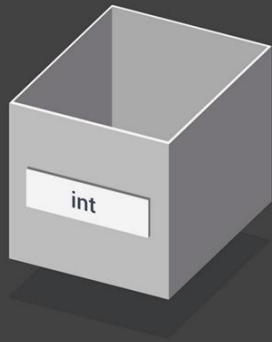
变量是设备内部的一种容器。每个变量可保存一个值，如数字或文本片段。变量的值并非始终不变，可替换为其他值，因此称为“变量”。大变量称为**对象**，其中可包含若干个小变量，小变量是该对象的**域**。

为变量赋值时，第一个值称为变量的初始值，为变量赋值的过程称为初始化变量。有些变量的初始化过程根本不需要我们编写任何指令。例如，当**整型变量**（该变量用于存储整数）作为某对象的域时，将具有默认值 0。

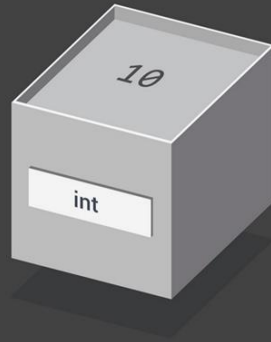
对于其他变量，则必须编写指令才能执行初始化。否则设备将拒绝执行试图使用该变量值的应用。例如，某个整型变量不是某个对象的域，没有默认值。

代码示例

```
// Create a variable named currentScore capable of holding an integer.  
// This variable is not a field of an object.  
int currentScore;  
  
// At this point the variable has not been initialized yet.  
// An attempt to use its contents here would prevent the app from executing.  
// Initialize the variable to contain the value 10.  
currentScore = 10;  
  
// Safe to use the value of the variable here.  
int oldScore = currentScore;
```



Before initialization



After initialization

Input Parameter（输入参数）

定义

计算机是按照一系列称为**程序**的指令运行的机器。**Android 设备**便是计算机，**应用**是程序。设备内部是称为**变量**的容器，用于存储数字或文本片段等**值**。

对象是变量，但在以下两个方面特殊。第一，对象中可包含更小的变量，即对象的**域**。例如，在 **MediaPlayer** 对象中可能包含多个域，用于存储正在播放的声音文件的名称、音量等级、文件回放的当前位置以及指示是否采用无限循环方式播放文件。

第二，我们可向对象附加称为**方法**的一系列指令，实际上是小程序。例如，我们的 **MediaPlayer** 对象可能具有 **play**、**pause** 和 **stop** 方法。对象的方法可使用对象内部这些方法所附加到的域。例如，**play** 方法需要使用全部四个域。

当应用中的指令指示设备执行某对象的方法时，便是在**调用该方法**。例如，我们可以调用 **MediaPlayer** 的 **play** 方法，让其执行播放声音文件的指令。

在调用对象的方法时，可为其赋予要使用的信息片段，称为**输入参数**。例如，**MediaPlayer** 对象所具有的 **setDataSource(String)** 方法需要一个 **String**（一系列字符）类型的输入参数，用于确定 **MediaPlayer** 将从哪一个媒体文件中获取输入。某给定方法始终需要相同数量的参数，且必须始终为可存储于指定类型变量中的表达式。

Pass one input parameter (a filename) to the method

```
mediaPlayer.setDataSource("/mnt/sdcard/song.mp3");
```

object name

method name

input parameter

dot

Instance（实例）

定义

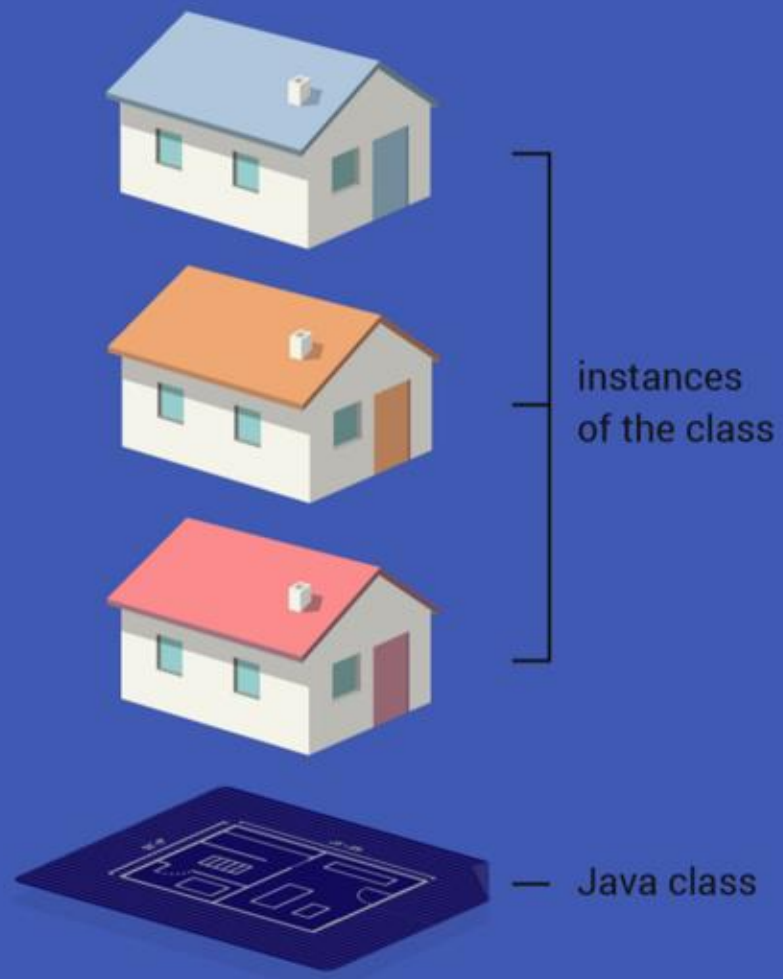
计算机是按照一系列称为**程序**的指令运行的机器。**Android 设备**便是计算机，**应用**是使用 **Java** 语言编写的程序。设备内部是称为**变量**的容器，用于存储数字或文本片段等**值**。

对象是变量，但在以下两个方面特殊。第一，对象中可包含更小的变量，即对象的**域**。例如，表示房屋的对象可能包含一个 **color** 域。第二，我们可向对象附加称为**方法**的一系列指令，实际上是小程序。

house 对象可能具有一个 **setColor** 方法，用于将房屋设置为不同的颜色。

对象分多种**类**（类型）。针对每个类，我们必须编写**定义**：即属于各个类对象的域和方法列表。每个给定类的对象都有一组相同的域和方法。例如，每个 **house** 对象必须具有称为 **color** 的域和称为 **setColor** 的方法。但是每个 **house** 对象都可在其 **color** 域中包含不同的值：一个房屋可以是红色，另一个房屋为蓝色。

在 **Java** 文件中编写类的定义。由于该定义包含关于该类的所有重要数据，因此图例使用文件表示类。属于类的对象称为类的**实例**。每个实例都具有在类定义中列出的所有域和方法。



Integer（整数）

定义

整数包括正数、负数和零。在 Android 设备内部，能够存储 `int` 的容器称为 **int 类型的变量**，它能够存储 $-2,147,483,648 = -2^{31}$ 到 $2,147,483,647 = 2^{31} - 1$ （含）之间的任意整数。（使用 Java 语言时，不要编写逗号。）

要存储不在此范围内的整数，必须使用 **long** 类型的变量。但 **long** 类型的成本更高：它占用的**内存**（存储空间）相当于 `int` 类型的两倍。



Intent

定义

计算机是按照一系列称为**程序**的指令运行的机器。**Android 设备**便是计算机，**应用**是程序。**变量**是设备中能够包含数字或文本片段等**值**的容器。能够包含小变量的大变量称为**对象**。我们可向对象附加称为**方法**的一系列指令，实际上是小程序。

对象分多种**类**。例如，每个应用至少包含一个 **Activity** 类的对象。此类的对象包含能够指示设备在屏幕上显示**用户界面**（如图像、按钮和文本片段）的方法。除了显示界面，每个 **Activity** 对象类都可以完成一项有用的工作。可以拨打电话、发送电子邮件或播放视频。

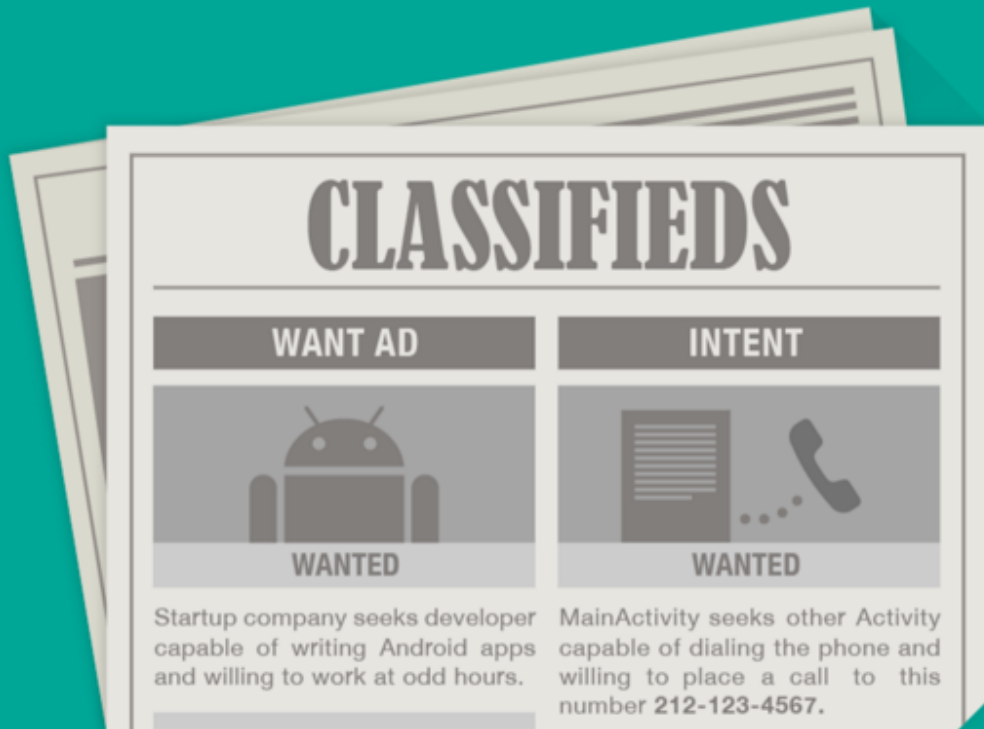
一个应用中的 **Activity** 对象可以向同一应用甚至其他应用中的其他 **Activity** 对象寻求帮助。第一个 **Activity** 可以通过加上 "I am looking for an Activity that is capable of displaying a Google map" 或 "I am looking for an Activity that can scan a barcode" 来描述其需求。该对象将此描述加载到 **Intent** 类型的对象中，然后发送意图以搜索 **Android** 设备。

设备上的每个应用都包含一个**清单文件**，其中列出了该应用中各 **Activity** 类的名称。此清单还包含**过滤器**，用于描述每个 **Activity** 的功能。**Intent** 对象将会检查这些清单。如果找到了满足需求的 **Activity** 类，将创建一个该类的对象并执行其 **onCreate** 方法。如果未找到，便向原始 **Activity** 返回这一坏消息。

代码示例

```
// One Activity object can execute the following code  
// to create another Activity object that can make a phone call.  
// The other Activity will then make the call.  
Uri uri = Uri.parse("tel:2121234567");  
// Uniform Resource Identifier contains phone number  
Intent intent = new Intent(Intent.ACTION_CALL, uri);  
try {  
    startActivity(intent);  
} catch (ActivityNotFoundException exception) {  
    textView.setText("could not find an Activity that meets the requirements: " + exception);  
}
```

}



An intent is like a want ad for a job
we don't want to do ourselves.

Java Programming Language（Java 编程语言）

定义

计算机是按照一系列称为**程序**的指令运行的机器。**Android 设备**便是计算机，**应用**是程序。

未来将会使用英语编写应用，但这一目标尚未实现。目前，我们必须使用更简单的语言（例如 **Java**）编写应用。

| PSEUDOCODE | JAVA CODE |
|------------------------------|-------------------------|
| Create a variable named "i". | <code>int i;</code> |
| Put 10 into it. | <code>i = 10;</code> |
| Then subtract 1 from it. | <code>i = i - 1;</code> |

Javadoc

定义

计算机是按照一系列称为**程序**的指令运行的机器。**Android 设备**便是计算机，**应用**是程序。

应用必须以编程语言（如 **Java**）编写，因为设备还不能可靠地理解人类语言（如英语）。事实上，如果我们在应用指令中使用英语编写句子，设备将变得极度混乱。

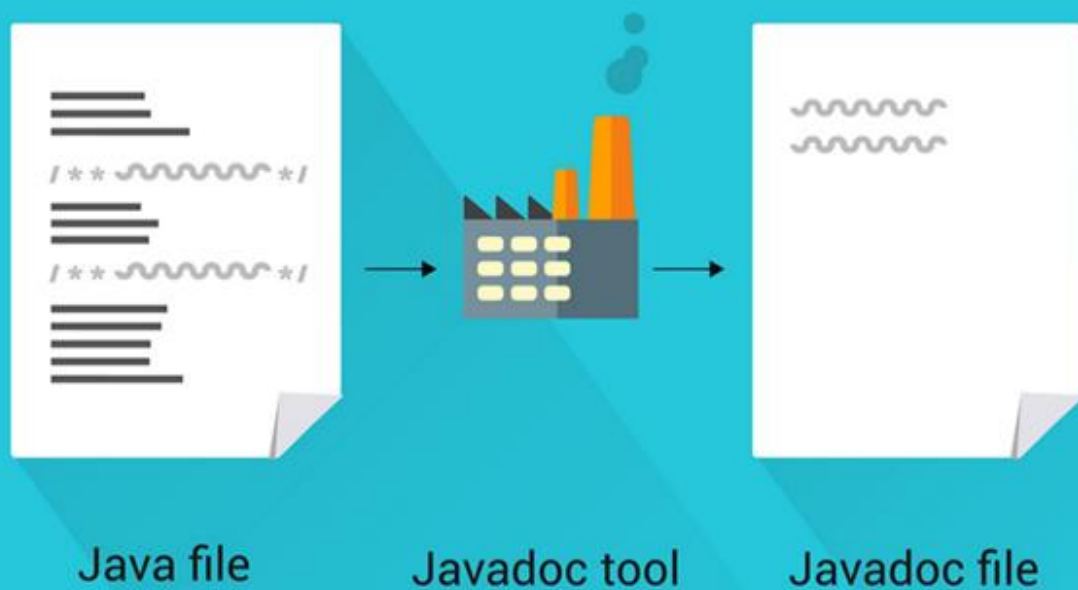
但通常我们又恰恰想这样做，为的是为写给人类的应用提供解释。这个解释称为**注释**，前后必须加**注释分隔符**：即指示设备切勿读取所括文本的标点符号。

在 **Java** 文件中，可通过 **Javadoc** 工具将前后加有特殊分隔符 **/**** 和 ***/** 的注释自动复制到单独的文件中。因此，这些注释称为 **Javadoc 注释**，产生的文档文件为 **Javadoc 文件**。如果在我们于 **Java** 文件中创建的每个类和方法前面都写一条 **Javadoc** 注释，**Javadoc** 文件便会为读者提供一份 **Java** 文件摘要。为了鼓励我们大家写这些注释，**Android Studio** 可以编写每条注释的一部分。

代码示例

```
public class MainActivity extends AppCompatActivity {  
    /**  
     * Show a message in a TextView.  
     *  
     * @param newMessage The message.  
     * @param resourceId The resource id of the TextView.  
     * @return The TextView's previous message.  
     */  
    private String showMessage(String newMessage, int resourceId) {  
        // Create a local variable named textView  
        // that refers to a TextView object created in the file activity_main.xml.  
        TextView textView = (TextView) findViewById(resourceId);  
        // Make a copy of the TextView's message,  
        // since the new message will wipe it out.
```

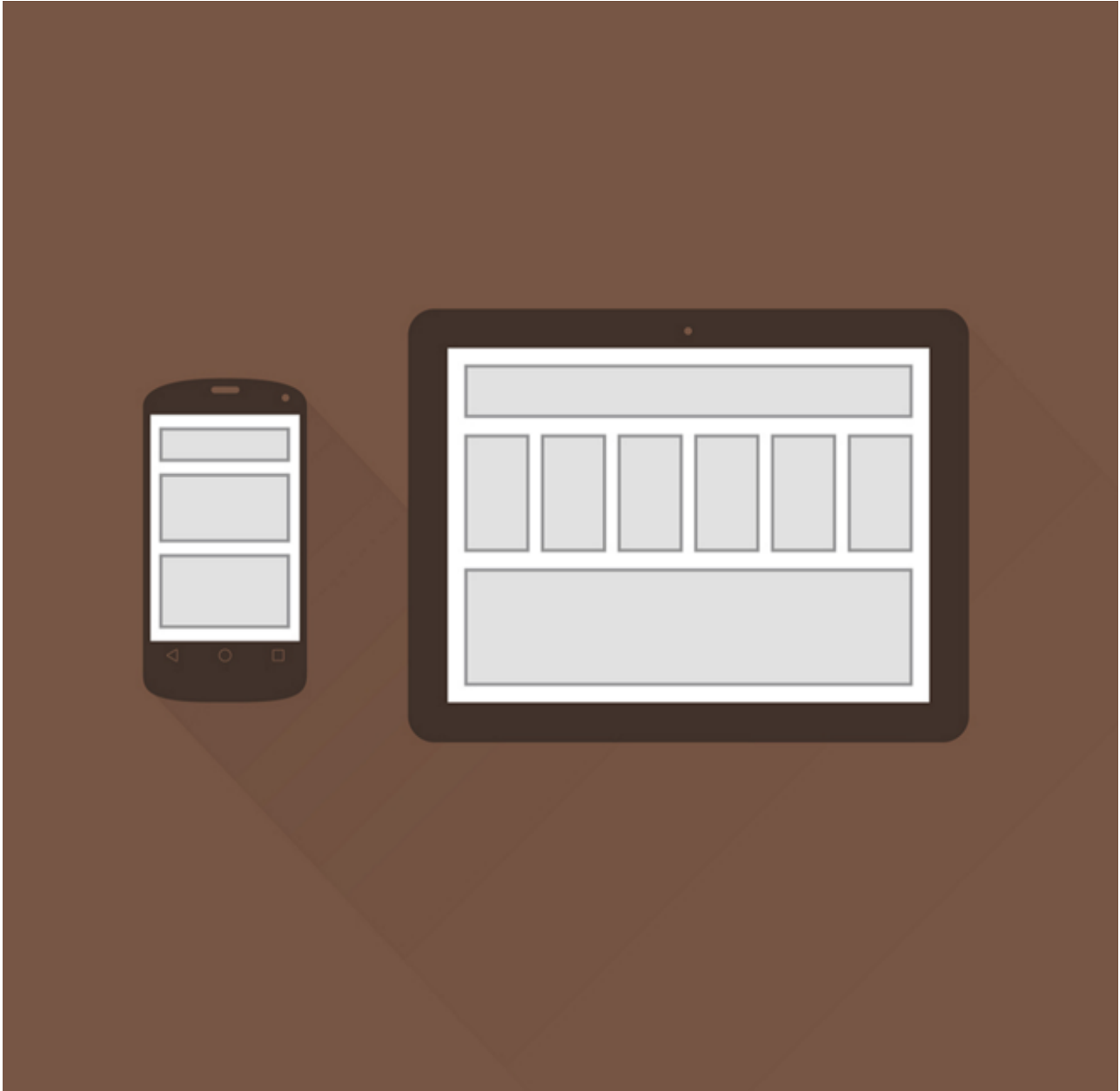
```
// Must convert the CharSequence returned by getText to a String.  
String oldMessage = textView.getText().toString();  
  
// Display the new message in the TextView, wiping out the old one.  
textView.setText(newMessage);  
  
// Return the previous message, in case anyone is still interested in it.  
return oldMessage;  
}  
}
```



Layout（布局）

定义

应用的**布局**指屏幕上显示的设计或布置。该**用户界面**由**视图**矩形区域构成。大视图可以包含小视图，并且始终有一个包含所有其他视图的最大视图。



layout_margin

定义

视图是屏幕上的矩形区域。例如，**TextView** 显示文本，**ImageView** 显示图像。

默认情况下，两个视图彼此相邻放置。如果不想让两个视图相互接触，可以沿着一个视图的一条边留出部分**空白**。事实上，可要求沿着视图的全部四条边各留出部分空白。

代码示例

```
<LinearLayout
    android:layout_width="match_parent"
    android:layout_height="wrap_content"
    android:orientation="horizontal">
    <TextView
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:background="#FFC400"
        android:text="The"/>
    <TextView
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:layout_marginLeft="8dp"
        android:background="#2196F3"
        android:text="End"/>
</LinearLayout>
```



`android:layout_marginLeft="8dp"`

layout_weight

定义

视图是屏幕上的矩形区域。**LinearLayout** 是大视图，其中可包含小视图，即**子视图**。**horizontal** **LinearLayout** 将其子视图排成一行，而 **vertical** **LinearLayout** 将其子视图排成一行。

horizontal **LinearLayout** 中的每个子视图都可以为自己申请一个最小的宽度。如果布局足够宽，则在满足这些请求之后将会剩下一些宽度。

之后，系统会在要求平分宽度的子视图中平分剩下的宽度。每个子视图要求平分的数量称为该子视图的**布局权重**。

在代码样例和图例中，**horizontal** **LinearLayout** 包含三个子视图，要求的 **layout_width** 总计为 48dp。这远远小于布局的宽度。这样，**EditText** 子视图要求平分剩余的宽度，而另外两个子视图不要求平分宽度。因此，剩余宽度共分为一份，**EditText** 会进行扩展以填充其要求平分的宽度。

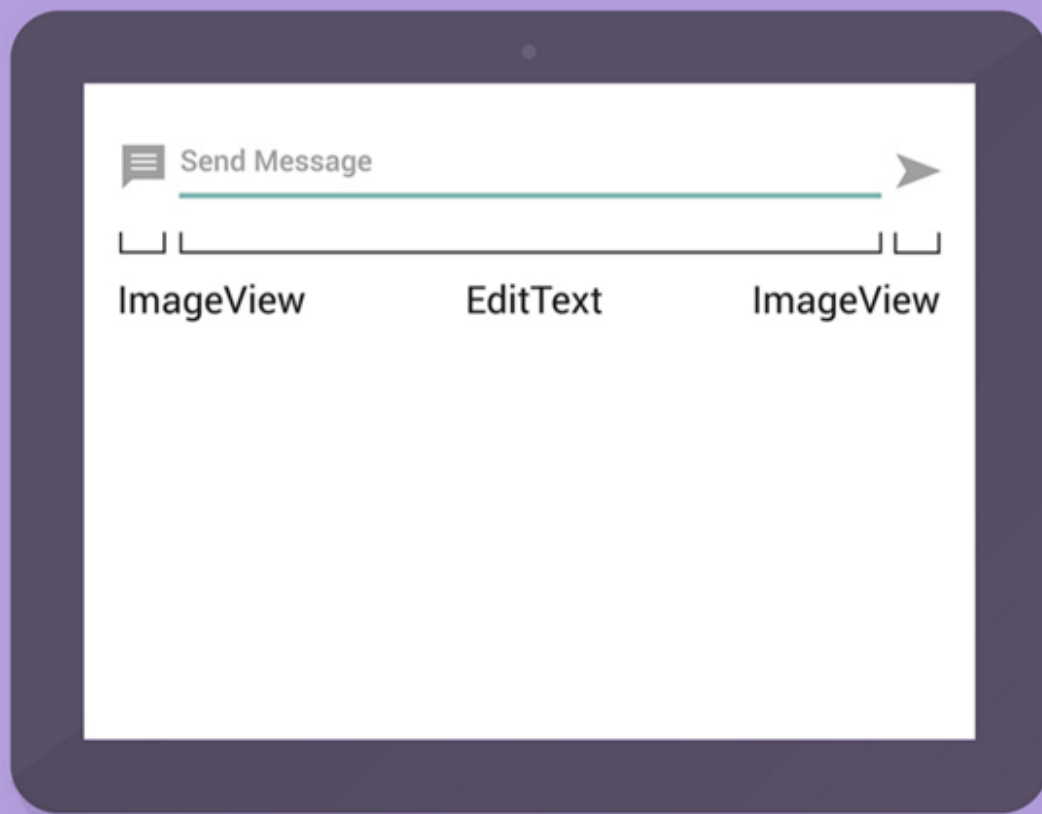
vertical **LinearLayout** 的子视图也适用类似的规则。如果 **LinearLayout** 足够高，将会在要求平分高度的子视图之间分配剩余垂直空间。

代码示例

```
<LinearLayout
    android:layout_width="match_parent"
    android:layout_height="wrap_content"
    android:orientation="horizontal"
    android:padding="16dp">
    <ImageView
        android:layout_width="24dp"
        android:layout_height="24dp"
        android:layout_gravity="center_vertical"
        android:src="@drawable/ic_chat"/>
    <EditText
        android:layout_width="0dp"
```

```
        android:layout_height="wrap_content"
        android:layout_marginLeft="8dp"
        android:layout_marginRight="8dp"
        android:layout_weight="1"
        android:textAppearance="?android:textAppearanceMedium"
        android:hint="Send Message"/>
    <ImageView
        android:layout_width="24dp"
        android:layout_height="24dp"
        android:layout_gravity="center_vertical"
        android:src="@drawable/ic_send"/>
</LinearLayout>
```

horizontal LinearLayout



LinearLayout

定义

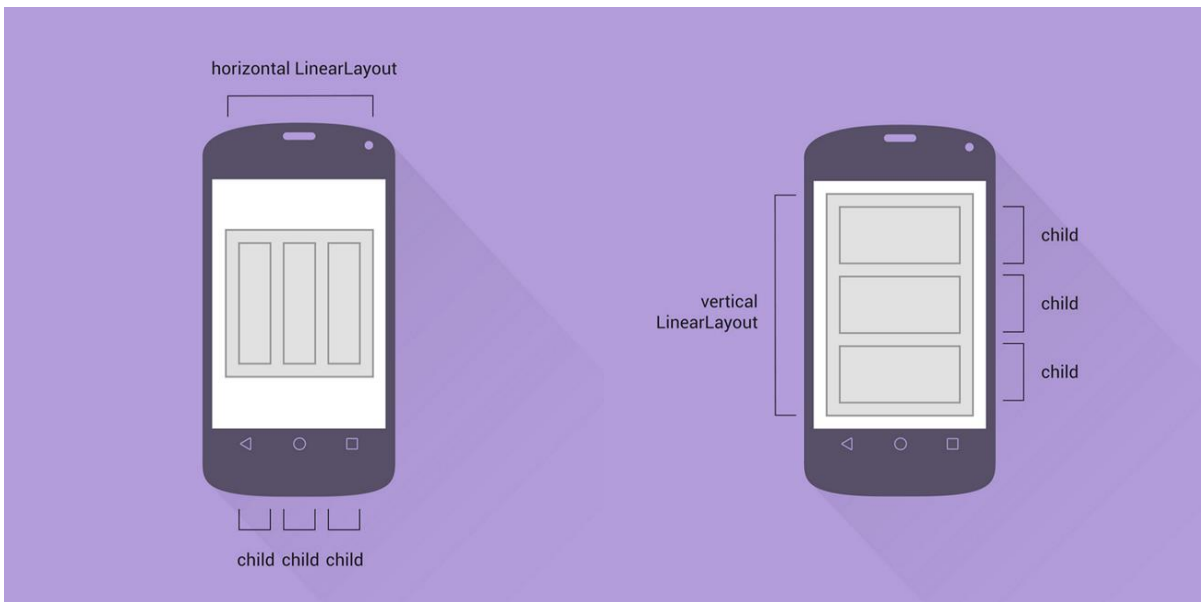
视图是屏幕上的矩形区域。例如，**TextView** 显示文本，**ImageView** 显示图像。

ViewGroup 是大视图，其中可包含小视图。小视图称为 ViewGroup 的**子视图**，可以是 TextView 或 ImageView。ViewGroup 称为其子视图的**父视图**。

LinearLayout 是一种常见的 ViewGroup。它将其子视图排成垂直列或水平行。

代码示例

```
<LinearLayout
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    android:orientation="vertical"/>
    <TextView
        android:layout_width="match_parent"
        android:layout_height="wrap_content"
        android:text="some content"/>
    <TextView
        android:layout_width="match_parent"
        android:layout_height="wrap_content"
        android:text="some content"/>
    <TextView
        android:layout_width="match_parent"
        android:layout_height="wrap_content"
        android:text="some content"/>
</LinearLayout>
```



Literal（文本）

定义

计算机是按照一系列称为**程序**的指令运行的机器。**Android 设备**便是计算机，**应用**是程序。我们使用**Java 语言**编写应用。

在应用中，可以写入**文本**（如 10 或 "John"）来表示**固定值**，如数字或文本片段。要表示可以在执行应用时发生更改的值，必须编写另外一种表达式，即**变量**。

Java 语言规定了多条文本编写规则。**数字文本**可以包含数字，但不得包含逗号、空格或标点符号（如美元符号）。**字符串文本**（表示文本片段）必须括在双引号中。**布尔文本**必须为小写的 true 或 false。



Local Variable（局部变量）

定义

计算机是执行一系列称为**程序**的指令的机器。**Android 设备**便是计算机，**应用**是程序。

变量是用于存储数字或文本片段等**值**的容器。大变量称为**对象**，其中可包含若干个小变量，小变量是该对象的**域**。我们可向对象附加称为**方法**的一系列指令，实际上是小程序。

对象分多种**类**。针对每个类，我们必须编写**定义**：即属于类对象的域和方法列表。

局部变量是由对象方法中的指令创建的变量。变量的名称只能在该方法的指令中提及，且只在设备执行该方法时变量才存在。

全局变量是对象的域。此变量由对象本身而非方法创建。对象的任何方法都可以提及该变量的名称，因此所有这些方法都可以访问变量中包含的值。只要包含全局变量的对象存在，全局变量就存在。

代码示例

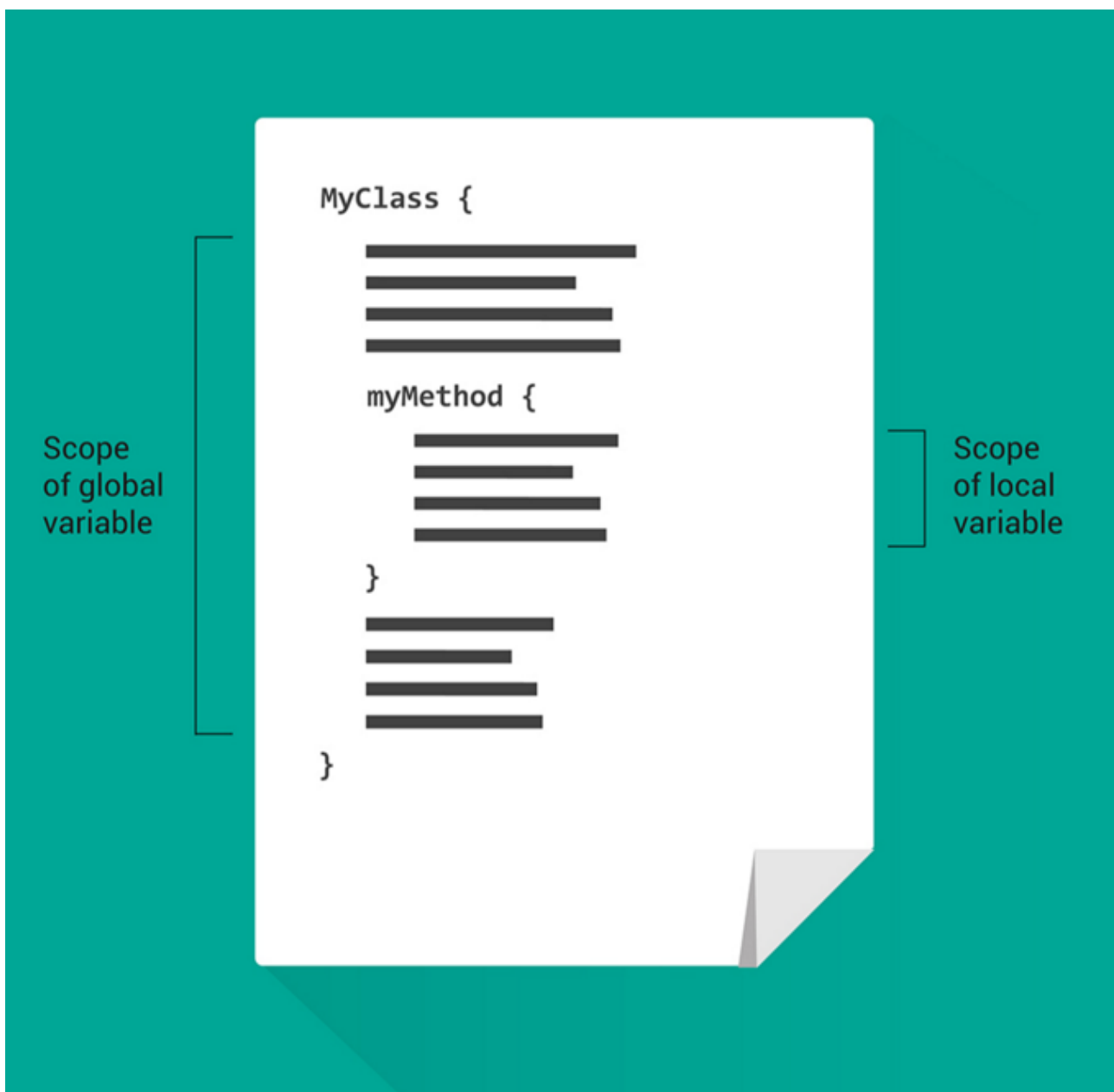
```
// This is the definition for a class of objects named MainActivity. Each object of this class
// contains a field named mTag and has two methods named onCreate and onDestroy.
public class MainActivity extends AppCompatActivity {
    // The following variable mTag is global because it was not created within the
    // curly braces of a method.
    String mTag = "MainActivity";
    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_main);
        // The following instruction declares (creates) a variable named textView. The variable
        // is local because it was declared within the curly braces of a method. The variable's
        // scope (the set of instructions where its name can be mentioned) extends from its
        // declaration to the closing curly brace that ends the method.
        TextView textView = (TextView) findViewById(R.id.textView);
    }
}
```



```

    textView.setText("Hello");
    Log.d(mTag, "onCreate");
}
@Override
protected void onDestroy() {
    super.onDestroy();
    Log.d(mTag, "onDestroy");
}
}

```



match_parent

定义

视图是屏幕上的矩形区域。例如，**TextView** 显示文本，**ImageView** 显示图像。

每个视图都包含在一个更大的视图中，即**父视图**，假定最大视图的父视图为矩形玻璃屏幕。

要想水平扩展视图以占用父视图的全部宽度，需要将视图的 `layout_width` 属性设置为特殊值 **match_parent**。同样，要想垂直扩展视图以占用父视图的全部高度，需要将 `layout_height` 设置为 **match_parent**。

代码示例

```
<ImageView
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    android:src="@drawable/mountains"
    android:scaleType="centerCrop"/>
```



`android:layout_width="match_parent"`



`android:layout_width="200dp"`

Method Signature（方法签名）

定义

计算机是按照一系列称为**程序**的指令运行的机器。**Android 设备**便是计算机，**应用**是程序。设备内部是称为**变量**的容器，用于存储数字或文本片段等**值**。

对象是变量，但在以下两个方面特殊。第一，对象中可包含更小的变量，即对象的**域**。例如，跟踪采购的对象可能包含数量域。第二，我们可向对象附加称为**方法**的一系列指令，实际上是小程序。例如，采购对象可能具有 `calculatePrice` 方法。

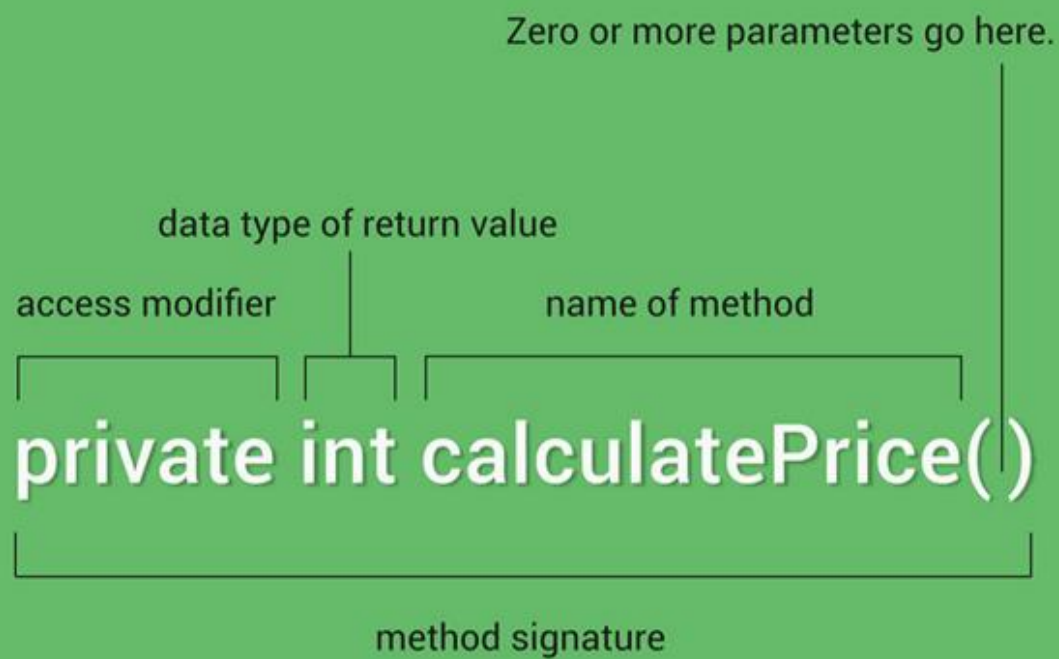
对象分多种**类**。针对每个类，我们必须编写**定义**：即属于各个类对象的域和方法列表。类的定义必须包含每个方法的**签名**，其中汇总了该方法的最重要的数据。签名包含方法的名称、方法**参数**（反馈到方法中的数据块）的名称和数据类型以及方法**返回值**（由方法生成的数据）的数据类型。签名还包含方法的**访问修饰符**，用于指定应用的哪些部分可以使用该方法。

编写完方法签名后，我们需要编写一系列构成方法的指令。签名和指令共同构成**方法的定义**。

代码示例

```
// This is the definition of a class named MainActivity. Although the class has many
// methods, we show the definition of only one of them.
public class MainActivity extends AppCompatActivity {
    int mQuantity;

    // This is the definition of a method named calculatePrice.
    // The first line of the definition, not including the {, is the signature of the method.
    private int calculatePrice() {
        int price = mQuantity * 5;
        return price;
    }
}
```



Method（方法）

定义

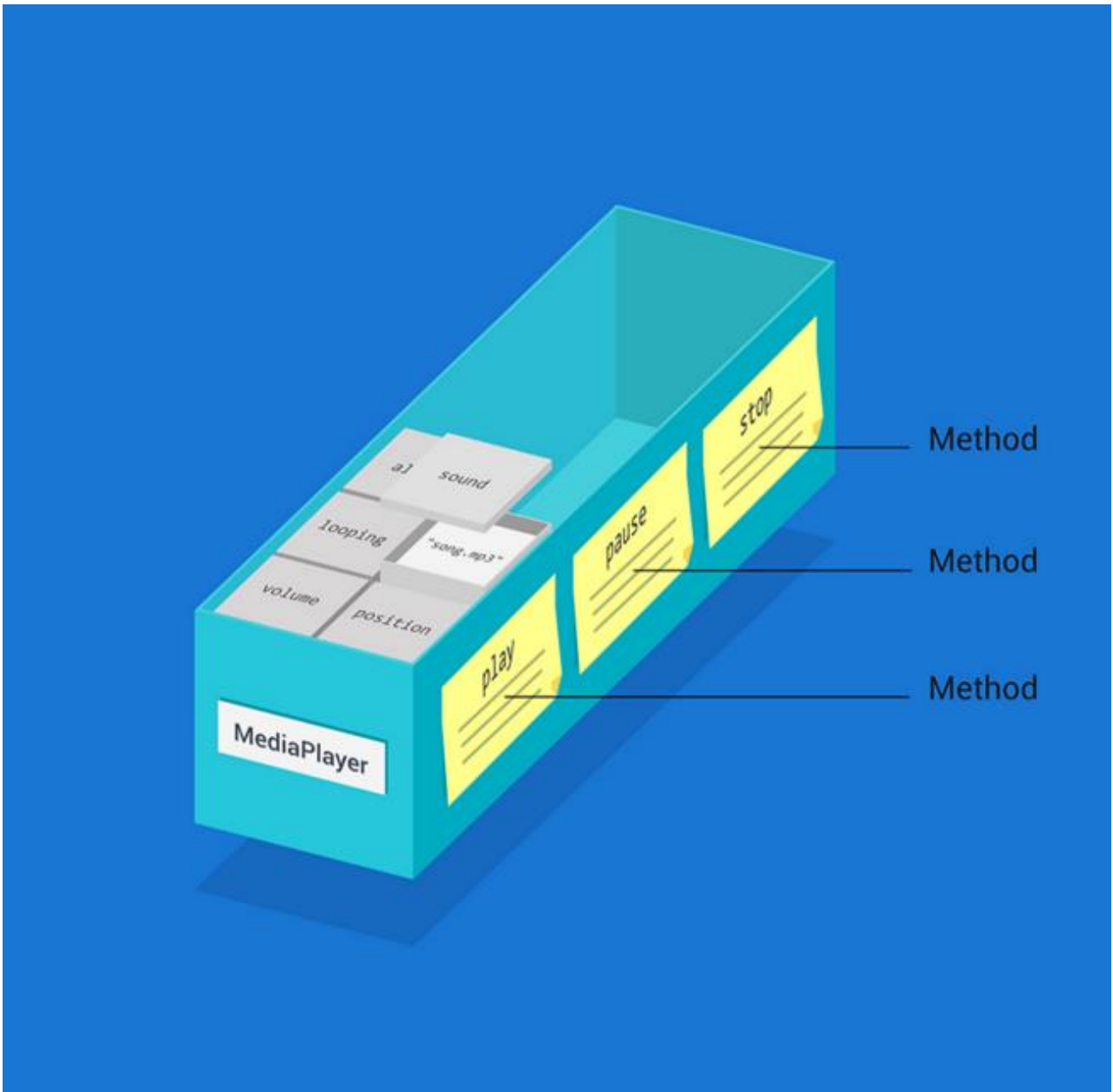
计算机是按照一系列称为**程序**的指令运行的机器。**Android 设备**便是计算机，**应用**是程序。

设备内部是称为变量的容器，用于存储数字或文本片段等值。

对象是变量，但在以下两个方面特殊。第一，对象中可包含更小的变量，即对象的**域**。例如，在 **MediaPlayer** 对象中可能包含多个域，用于存储正在播放的声音文件的名称、音量等级、文件回放的当前位置以及指示是否采用无限循环方式播放文件。

第二，我们可向对象附加称为**方法**的一系列指令，实际上是小程序。例如，我们的 **MediaPlayer** 对象可能具有 **play**、**pause** 和 **stop** 方法。对象的方法可使用对象内部这些方法所附加到的域。例如，**play** 方法需要使用全部四个域。

当指示计算机执行对象的方法时，便是在**调用该方法**。例如，我们可以调用 **MediaPlayer** 的 **play** 方法，让其执行播放声音文件的指令。



Nested ViewGroups（嵌套式 ViewGroup）

定义

视图是屏幕上的矩形区域。例如，**TextView** 显示文本，**ImageView** 显示图像。

ViewGroup 是大视图，其中可包含小视图。小视图称为 ViewGroup 的**子视图**，可以是 TextView 或 ImageView。ViewGroup 称为其子视图的**父视图**。每个子视图都**嵌套**（完全包含）在其父视图内。

子视图可具有其自己的子视图。例如，图例显示了一个包含两个子视图的 **vertical LinearLayout**。第一个子视图是包含三个子视图的 **horizontal LinearLayout**，第二个子视图是包含四个子视图的 **RelativeLayout**。这四个子视图的其中一个子视图自身还有一个子视图。



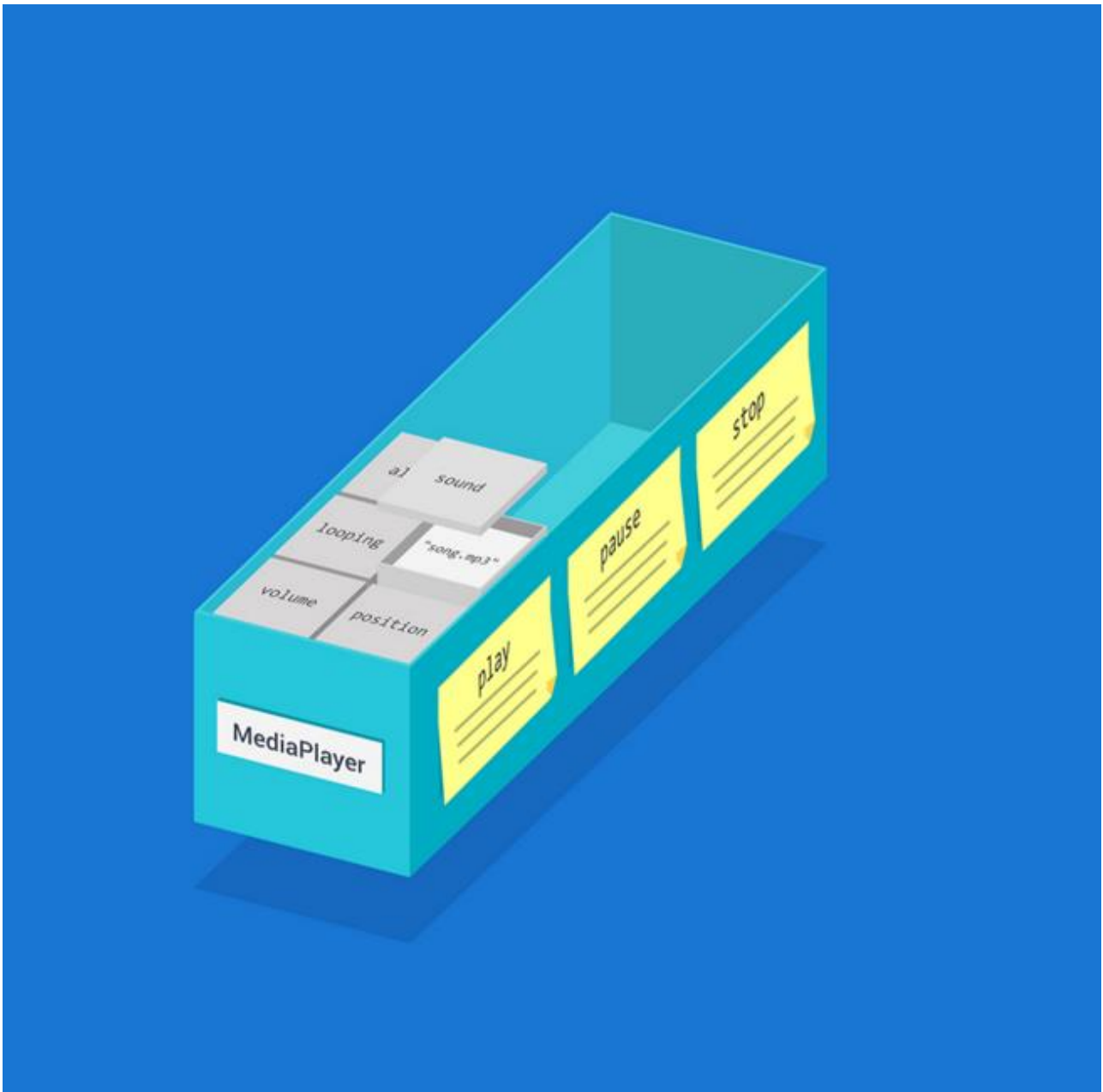
Object（对象）

定义

计算机是按照一系列称为**程序**的指令运行的机器。**Android 设备**便是计算机，**应用**是程序。设备内部是称为**变量**的容器，用于存储数字或文本片段等**值**。

对象是变量，但在以下两个方面特殊。第一，对象中可包含更小的变量，即对象的**域**。例如，在 **MediaPlayer** 对象中可能包含多个域，用于存储正在播放的声音文件的名称、音量等级、文件回放的当前位置以及指示是否采用无限循环方式播放文件。

第二，我们可向对象附加称为**方法**的一系列指令，实际上是小程序。我们的 **MediaPlayer** 对象可能具有 **play**、**pause** 和 **stop** 方法。对象的方法可使用对象内部这些方法所附加到的域。例如，**play** 方法需要使用全部四个域。



OnClickListener

定义

计算机是按照一系列称为**程序**的指令运行的机器。**Android 设备**便是计算机，**应用**是程序。

设备内部是称为**变量**的容器，用于存储数字或文本片段等值。变量中的值很容易改变，因此称之为“变量”。

能够包含小变量的大变量称为**对象**。小变量称为对象的**域或成员**，且小变量往往自身便是对象。也就是说我们可以将小对象存储在大对象中。还可向对象附加称为**方法**的一系列指令，实际上是小程序。。不能更改对象的方法。这些方法永久附加到该对象上。

对象分多种**类**。例如，**ImageView** 类的对象显示图像，触摸 **Button** 类的对象时，对象会作出响应。给定类的所有对象都附有完全相同的方法集。例如，每个 **Button** 类对象都有一个 **setEnabled(boolean)** 方法，确定是否可以触摸 **Button**。

但通常，我们需要执行不同操作（例如“采购”和“取消”）的两个或多个 **Button**。通过将不同的方法附加到各个 **Button** 无法实现这一要求。所有的 **Button** 都属于同一个类，因此必须具有完全相同的方法。类似地，从 "play" 变为 "pause" 的 **Button** 必须在不改变方法的前提下进行。

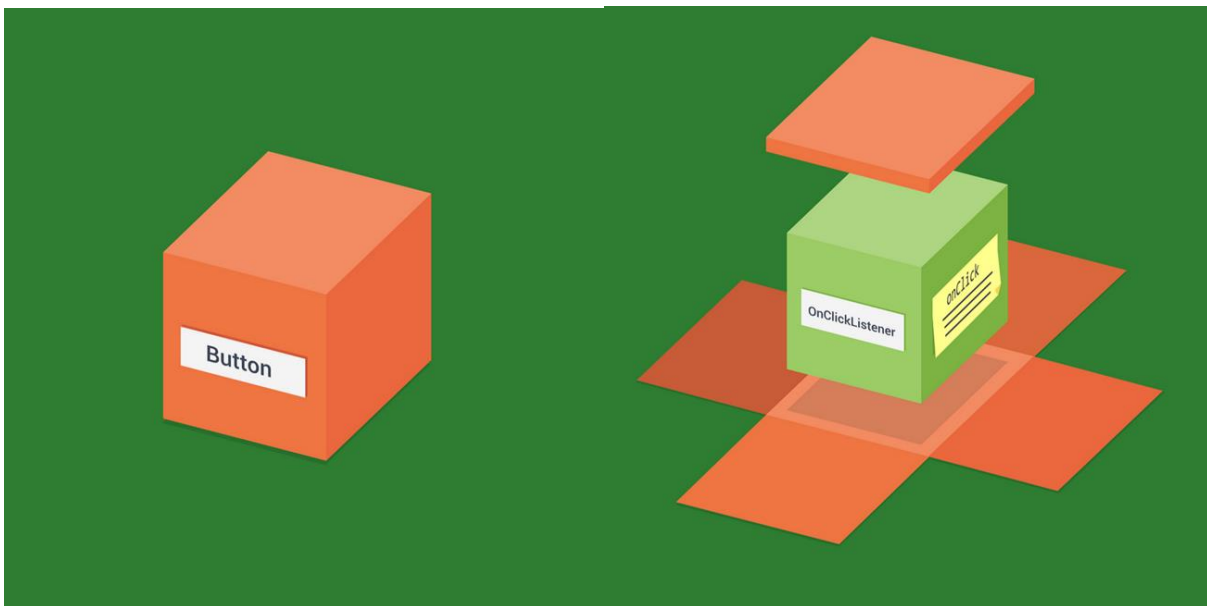
解决方案是创建一个名为 **OnClickListener** 的单独对象。附加到监听器的方法是 **onClick(View)**，其中包含要在触摸 **Button** 后执行的指令。（在 **Android** 的早期版本中，“单击”一词即表示“触摸”。）监听器存储在 **Button** 的一个域中，在触摸发生之前一直处于睡眠状态。触摸时，会自动执行附加到监听器的 **onClick** 方法。

Button 有许多个类，每个类都有一组不同的方法，这让我们很难进行区分。但 **OnClickListener** 是一种非常简单的对象类型。我们可以创建这样两个监听器：其中一个监听器的 **onClick** 方法包含“采购”指令，另外一个监听器的 **onClick** 执行“取消”指令。然后，可将这两个监听器存储到不同的 **Button** 中，从而在触摸每个 **Button** 时产生不同的行为。由于可更改域的内容，因此可将 **Button** 中的“播放”监听器替换为“暂停”监听器。

另外，还可使用另外一种方式来设置要在触摸 **Button** 时执行的方法。如果使用 XML 布局文件中的 **<Button>** 元素创建 **Button**，则可以使用该元素的 **android:onClick** 属性指定方法。这一方便的备选方案专为不改变行为的 **Button** 提供。

代码示例

```
Button button = (Button) findViewById(R.id.button);  
button.setOnClickListener(new View.OnClickListener() {  
    @Override  
    public void onClick(View v) {  
        String s = "Thanks for touching the button.";  
        Toast t = Toast.makeText(MainActivity.this, s, Toast.LENGTH_LONG);  
        t.show();  
    }  
});
```



Operator（操作符）

定义

计算机是按照一系列称为**程序**的指令运行的机器。**Android 设备**便是计算机，**应用**是程序。在应用中，可以写入**文本**（如 10 或 "John"）来表示**值**，如数字或文本片段。

变量是设备内部用于存储值的容器。每个变量都有一个**名称**，如 "x"、"y" 或 "greeting"。

通过在应用指令中编写**表达式**，指示设备对数字和文本进行操控。表达式中的值以文本或变量的形式编写：

$x + y$

$x - 10$

`greeting + "John!"`

其中，加号和减号称为**操作符**，所操作的值称为**操作数**。通过操作符计算出的结果称为表达式的**值**。



The diagram shows the expression "10 + 20" in large white text on a blue background. Below the expression, three brackets are used to identify the components: the first bracket under "10" is labeled "operand", the second bracket under "+" is labeled "operator", and the third bracket under "20" is labeled "operand".

Override（覆盖）

定义

计算机是按照一系列称为**程序**的指令运行的机器。**Android 设备**便是计算机，**应用**是使用 **Java** 语言编写的程序。设备内部是称为**变量**的容器，用于存储数字或文本片段等**值**。

对象是变量，但在以下两个方面特殊。第一，对象中可包含更小的变量，即对象的**域**。例如，**TextView** 对象在屏幕上显示文本，并且可能包含称为 **mText** 的域。第二，我们可向对象附加称为**方法**的一系列指令，实际上是小程序。**TextView** 对象可能具有称为 **setText** 的方法，可将文本片段放入对象的 **mText** 域中。

对象分多种**类**（类型）。针对每个类，我们必须编写**定义**：即属于各个类对象的域和方法列表。每个给定类的对象都有一组相同的域和方法。例如，每个 **TextView** 对象必须具有称为 **mText** 的域和称为 **setText** 的方法。但是每个 **TextView** 都可在其 **mText** 域中包含不同的值：一个 **TextView** 可能会说“您好”而另一个说“再见”。

定义类即意味着通过编写类定义来创建类。之后我们可以创建类的对象，但这是单独的操作。我们完全可以在不创建类对象的情况下定义类。

可通过列出属于各个类对象的每个域和方法，从头开始定义一个新类。我们也可以使新类自动具有某现有类的所有域和方法，并在此基础上加入新类定义中列出的附加域和方法，以此来创建新类。在此场景中，现有类称为**超类或基类**，而新类是其**子类**。我们称子类继承了其超类的所有域和方法，并且子类是使用**继承**通过超类创建的。之所以称为“子类”和“超类”，是因为在图表中，我们始终在超类下方绘制子类。请注意，子类通常会比其超类具有更多方法和域。

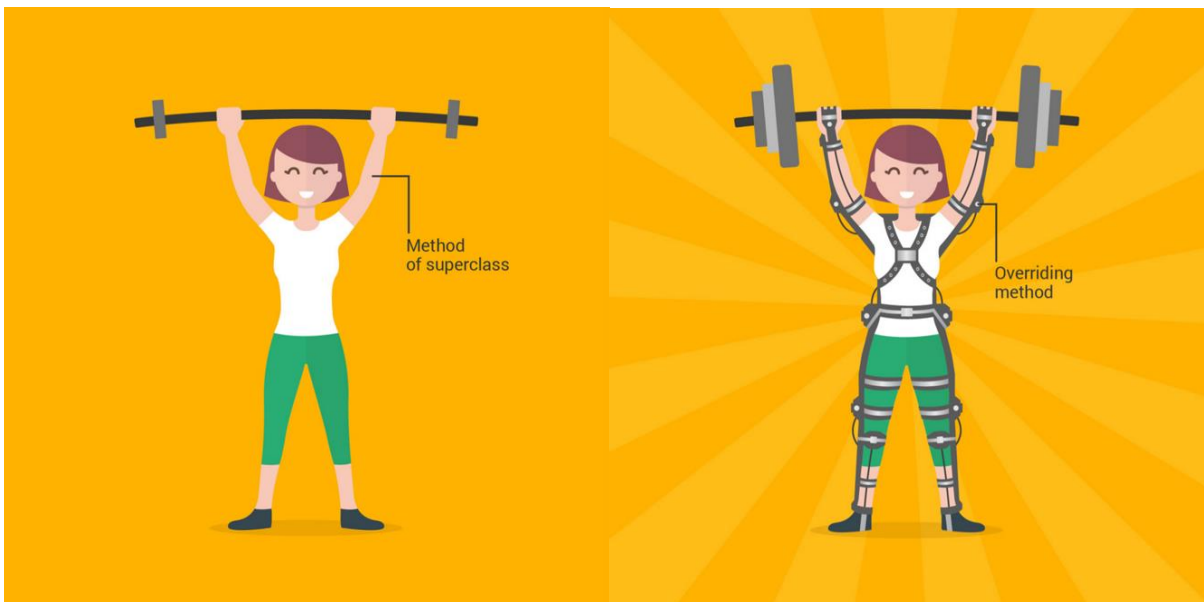
让我们看个例子。**EditText** 类的对象不仅可以执行 **TextView** 类的对象能够执行的所有操作，还可以执行其他操作。它可以在屏幕上显示文本，还允许用户编辑该文本。因此，**EditText** 类是使用继承根据 **TextView** 类创建的，并且具有支持编辑的附加域和方法。**EditText** 是 **TextView** 的子类，而 **TextView** 是 **EditText** 的超类。

除了添加新的方法，子类还可以添加现有方法的新版本。例如，之前曾提到过 **TextView** 类包含 **setText** 方法，可以将文本片段放入 **TextView** 以便在屏幕上显示。子类 **EditText** 具有一个同名的 **setText** 方法，但此方法可以进行更多操作。将文本片段放入 **EditText** 以后，可编辑和显示这些文本。我们可以说 **EditText** 类的 **setText** 方法**覆盖**了超类 **TextView** 中的同名方法。

在 Java 语言中，使用特殊单词 **@Override** 标记覆盖方法，覆盖方法通常具备被覆盖方法的所有功能，同时还有更多其他功能。事实上，覆盖方法通常通过调用（执行）被覆盖方法，然后执行附加指令来提供此功能。

代码示例

```
// A simplified superclass (TextView) and subclass (EditText).  
// The subclass overrides one method of its superclass.  
class TextView extends View {  
    // the text to be displayed  
    String mText;  
    public void setText(String text) {  
        mText = text;  
    }  
}  
  
public class EditText extends TextView {  
    @Override  
    public void setText(String text) {  
        // Call the overridden method of the superclass.  
        super.setText(text);  
        // Do additional work here to make the text editable.  
    }  
}
```

Package Name（包名）

定义

计算机是按照一系列称为**程序**的指令运行的机器。**Android 设备**便是计算机，**应用**是程序。应用中包含 `TextView` 类和 `ImageView` 类等许多不同**类**的**对象**。

两个人来自不同的家族，因此可以使用同一个名字，例如 **Bill Clinton** 和 **Bill Gates**。两个分别属于不同**包**的类也可以共用同一个名称，例如 `widget.TextView` 和 `thirdParty.TextView`。

包是类的集合。将包当作一个家族，为类提供共同的姓氏，即包本身的名称。

在英文中，姓氏写在名字的**右侧**，中间以空格分隔：

Bill Clinton

在 **Java** 语言中，包名写在**左侧**，中间以圆点分隔：

`widget.TextView`

一个包可以属于一个更大的包。例如，类 **TextView** 属于 **widget** 包，而此包属于更大的包 **android**。因此，类 `TextView` 的全名或**完全限定名称**为

`android.widget.TextView`

并且，必须在提到该类的每个 **Java** 文件顶部附近的 **import 语句**中写入这个完全限定名称。

每个应用都有一个包，其中包含在该应用中创建的类。应用的包名由组织名称后加应用名称构成，所有名称均为小写字母，并且在每个 **Java** 文件顶部的 **package 语句**中进行声明。

```
package com.mycompany.myapp;
```

如果您的应用创建了 `MainActivity` 类，其完全限定名称为

`com.mycompany.myapp.MainActivity`

代码示例

```
// Because of the package statement, the fully qualified name of class MainActivity is  
// com.mycompany.myapp.MainActivity. The same is true for every class we create.
```

```

package com.mycompany.myapp;

// Because of the import statement for class android.widget.TextView, every mention of
// class TextView in this file is assumed to refer to class android.widget.TextView.

import android.os.Bundle;
import android.support.v7.app.AppCompatActivity;
import android.widget.TextView;

// Create a class named MainActivity.
public class MainActivity extends AppCompatActivity {

    @Override

    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_main);
        TextView textView = (TextView) findViewById(R.id.textView);
        textView.setText("Hello");
    }

}

```

name of package

name of class

android.widget.TextView

fully qualified name of class

Padding（内边距）

定义

视图是屏幕上的矩形区域。例如，**TextView** 包含文本，**ImageView** 包含图像。

如果我们将视图的宽度和/或高度设置为特殊值 **wrap_content**，视图将收缩并包围内容。为防止视图包围得过紧，我们可以在每一边指定一个**内边距**。

代码示例

```
<TextView
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:background="#C0C0C0"
    android:text="CLAUSTROPHOBIA"/>
<TextView
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:paddingLeft="16dp"
    android:paddingRight="16dp"
    android:paddingTop="8dp"
    android:paddingBottom="8dp"
    android:background="#C0C0C0"
    android:text="CLAUSTROPHOBIA"/>
```

CLAUSTROPHOBIA

Without padding

CLAUSTROPHOBIA

With padding

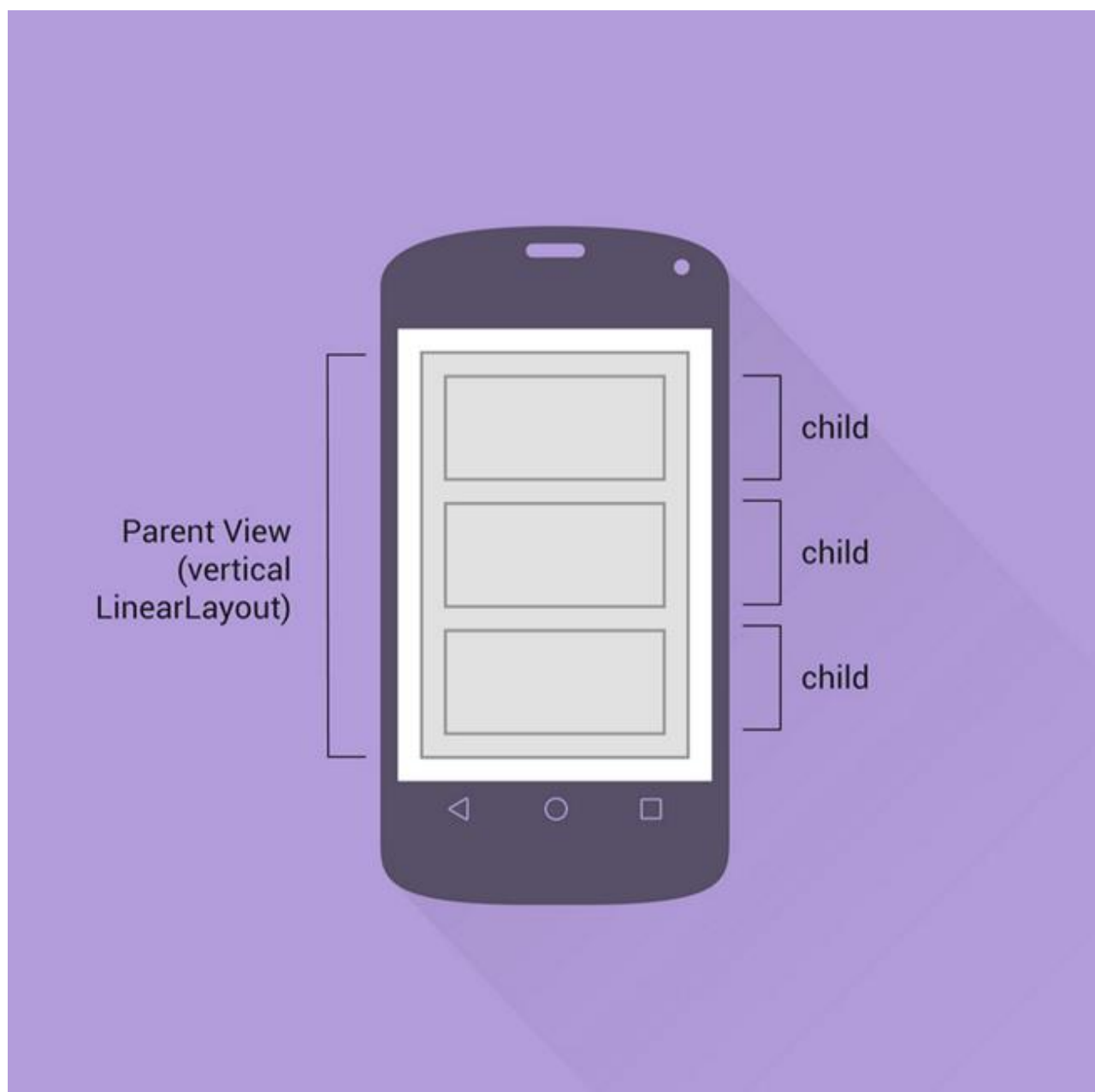
Parent View（父视图）

定义

视图是屏幕上的矩形区域。例如，**TextView** 显示文本，**ImageView** 显示图像。

ViewGroup 是大视图，其中可包含小视图。小视图称为 ViewGroup 的**子视图**，可以是 TextView 或 ImageView。ViewGroup 称为其子视图的**父视图**。图例显示了一个 vertical **LinearLayout** 父视图，其内部**嵌套**了三个子视图。

子视图可具有其自己的子视图。例如，大 ViewGroup 可以包含更小的 ViewGroup，更小的 ViewGroup 可以包含 TextView。



Parse（解析）

定义

计算机是按照一系列称为**程序**的指令运行的机器。**Android 设备**便是计算机，**应用**是程序。

由于 Android 设备还不能可靠地理解英语，因此我们必须以更简单的语言（如 Java 和 XML）编写应用。然后，桌面应用程序 Android Studio 将应用的文件从这些**编程语言**转换为 Android 设备能够理解的语言。

每种编程语言均有其自己的语法、标点和组织规则。例如，XML 文件始终包含一个数据项，该数据项中可能包含更小的项，而每个更小的项还可能包含再小的项。代码样例显示了一个 `LinearLayout`，其中包含两个子项，而第二个子项自身还有一个子项。

读取 XML 文件的第一步是**解析**该文件：分解该文件，确保所有组成部分均存在且顺序和关系均正确。解析 XML 或 Java 文件类似于将人类语言句子解析为各个组成部分。XML 解析将生成信息树，显示要在屏幕上创建和显示的对象之间的关系。

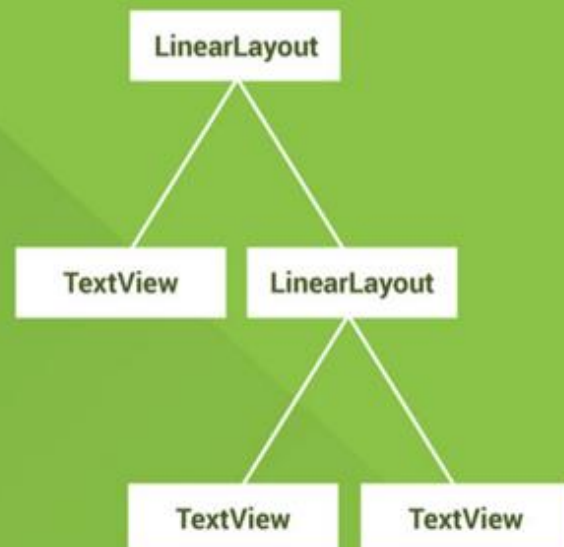
代码示例

```
<!-- A fuller version of the XML file parsed in the illustration. -->
<LinearLayout
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    android:orientation="vertical">
    <TextView
        android:layout_width="match_parent"
        android:layout_height="wrap_content"
        android:text="1"/>
    <LinearLayout
        android:layout_width="match_parent"
        android:layout_height="wrap_content"
        android:orientation="vertical"/>
```

```
<TextView
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:text="2a"/>
<TextView
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:text="2b"/>
</LinearLayout>
</LinearLayout>
```

```
<LinearLayout>
  <TextView/>
  <LinearLayout>
    <TextView/>
    <TextView/>
  </LinearLayout>
</LinearLayout>
```

XML file



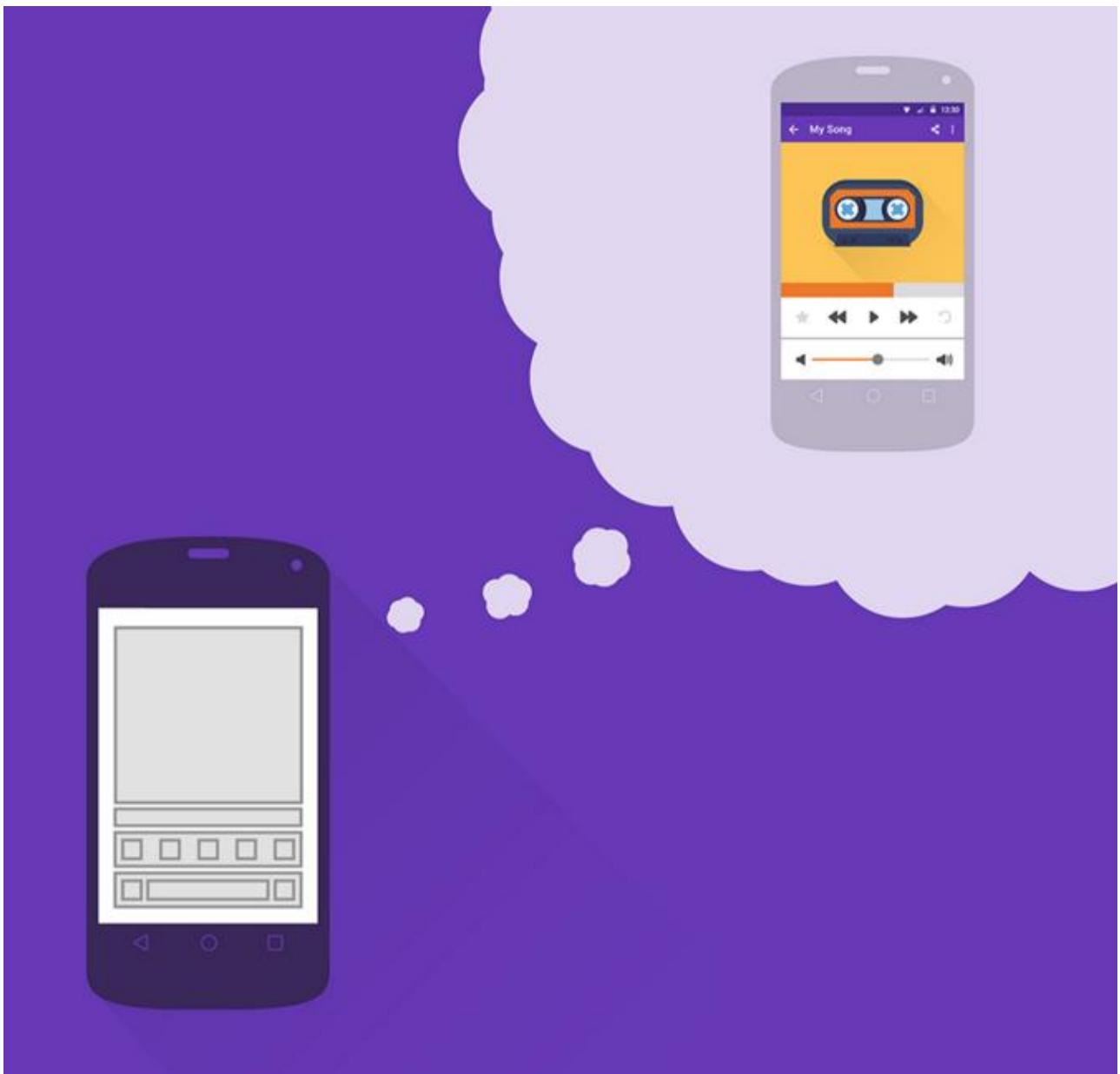
Tree of information about the objects to be created

Prototype（原型）

定义

计算机是按照一系列称为程序的指令运行的机器。**Android 设备**便是计算机，**应用**是程序。

通常，我们编写应用的**原型**，一个没有任何装饰的简单版本。原型相当于测试，用于查看应用的基本设计是否合理。如果合理，我们可在以后添加**附属项目**。



Pseudocode（伪代码）

定义

计算机是按照一系列称为**程序**的指令运行的机器。**Android 设备**便是计算机，**应用**是程序。

由于 Android 设备尚未可靠地理解人类语言，因此必须以较简单的语言（例如 **Java**）编写应用。使用 **Java** 编写的指令称为**代码**，并且能为设备所理解。

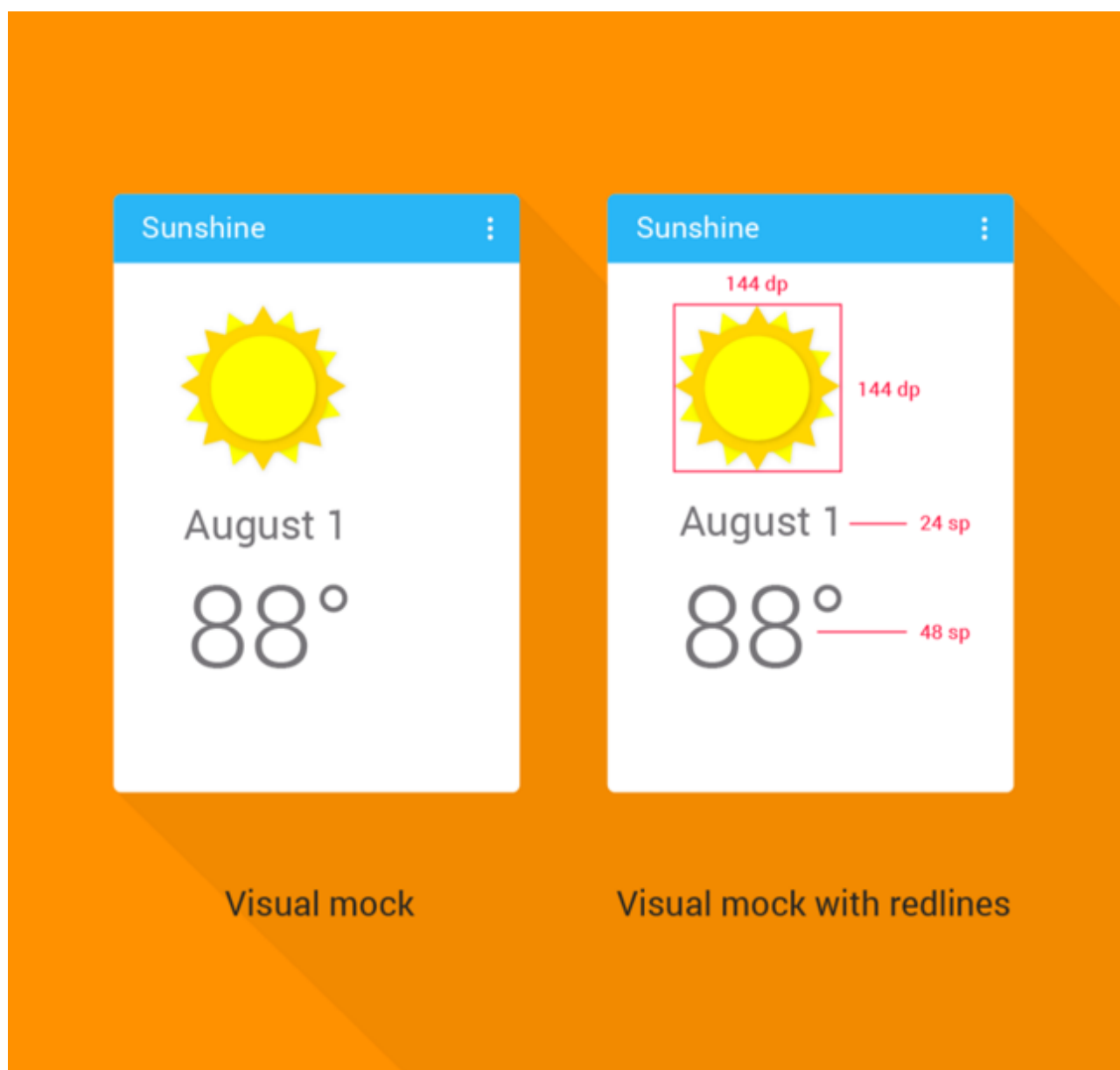
但是，采用以 **Java** 描述的细小步骤编写应用可能会相当冗长，因此我们首先用人类语言勾绘出应用的轮廓。这种复述称为**伪代码**，只能为人类所理解。

| PSEUDOCODE | JAVA CODE |
|------------------------------|-------------------------|
| Create a variable named "i". | <code>int i;</code> |
| Put 10 into it. | <code>i = 10;</code> |
| Then subtract 1 from it. | <code>i = i - 1;</code> |

Redlines（红线）

定义

视觉模拟用于描绘应用的预期布局。通常，设计师通过创建视觉模拟将项目在屏幕上的尺寸传达给开发人员：项目之间的间距、项目的字体和字体大小。由于这些注释以红色绘制，部分屏幕项目的周围还有线条，因此将其称为**红线**。



RelativeLayout

定义

视图是屏幕上的矩形区域。例如，**TextView** 显示文本，**ImageView** 显示图像。

ViewGroup 是大视图，其中可包含小视图。小视图称为 ViewGroup 的**子视图**，可以是 TextView 或 ImageView。ViewGroup 称为其子视图的**父视图**。

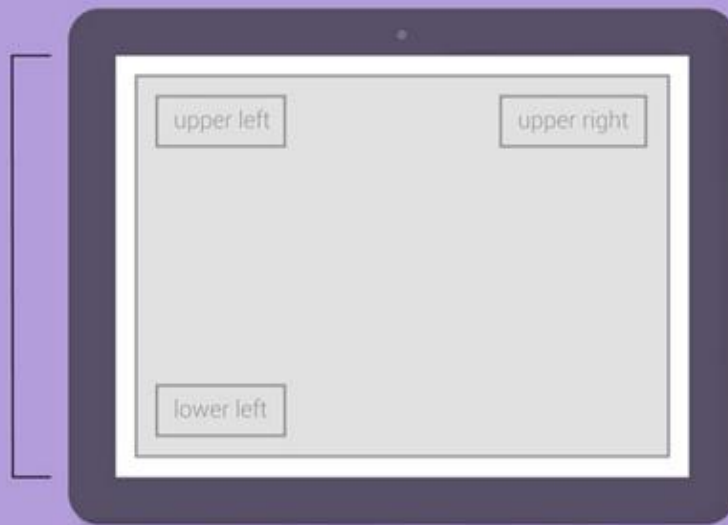
RelativeLayout 是一种常见的 ViewGroup，允许我们相对于其自身的边缘放置其子项。例如，图例中的三个子项分别放置在 RelativeLayout 的三个角上。RelativeLayout 还允许我们相对于彼此之间的关系来安排子项：可将一个子项放置在另一个子项的右侧，甚至可以重叠。

代码示例

```
<RelativeLayout
    android:layout_width="match_parent"
    android:layout_height="match_parent"/>
<TextView
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:layout_alignParentLeft="true"
    android:layout_alignParentTop="true"
    android:text="upper left"/>
<TextView
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:layout_alignParentRight="true"
    android:layout_alignParentTop="true"
    android:text="upper right"/>
<TextView
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
```

```
android:layout_alignParentLeft="true"  
android:layout_alignParentBottom="true"  
android:text="lower left"/>  
</RelativeLayout>
```

RelativeLayout



RelativeLayout with 3 children

Return Value（返回值）

定义

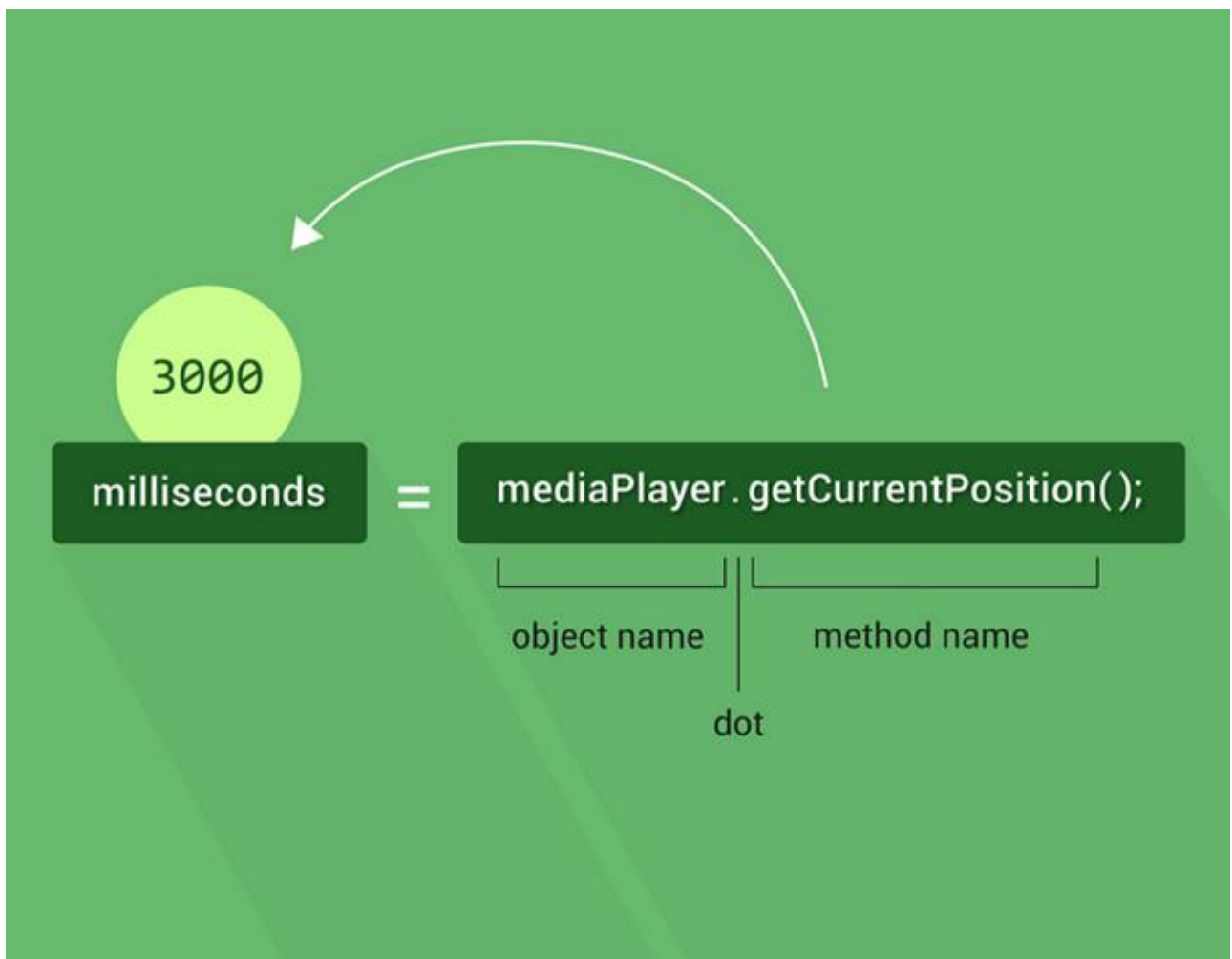
计算机是按照一系列称为**程序**的指令运行的机器。**Android 设备**便是计算机，**应用**是程序。设备内部是称为**变量**的容器，用于存储数字或文本片段等**值**。

对象是变量，但在以下两个方面特殊。第一，对象中可包含更小的变量，即对象的**域**。例如，在 **MediaPlayer** 对象中可能包含多个域，用于存储正在播放的声音文件的名称、音量等级、文件回放的当前位置以及指示是否采用无限循环方式播放文件。

第二，我们可向对象附加称为**方法**的一系列指令，实际上是小程序。例如，我们的 **MediaPlayer** 对象可能具有 **play**、**pause** 和 **stop** 方法。对象的方法可使用对象内部这些方法所附加到的域。例如，**play** 方法需要使用全部四个域。

当应用中的指令指示设备执行某对象的方法时，便是在**调用该方法**。例如，我们可以调用 **MediaPlayer** 的 **play** 方法，让其执行播放声音文件的指令。

某个方法的指令可计算得出一个值（称为该方法的**返回值**），以传回到调用该方法的指令。例如，**MediaPlayer** 对象有一个 **getCurrentPosition()** 方法，用于返回播放器当前播放的文件位置（以毫秒计）。给定方法始终返回相同类型的值。例如，**getCurrentPosition()** 始终返回 **int** 类型的值，即整数，没有小数点和分数。



Robust（可靠）

定义

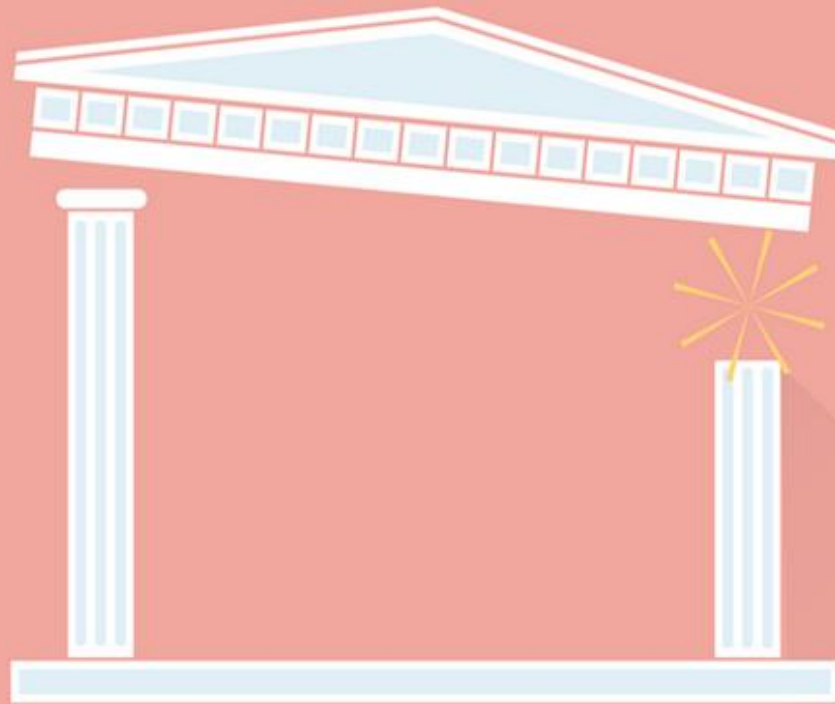
计算机是按照一系列称为**程序**的指令运行的机器。**Android 设备**便是计算机，**应用**是程序。

通常，开发人员在更改应用的一条指令时并不检查是否有任何其他指令需要因此更改而进行修改。通过让应用的不同部分彼此独立，应用会变得更**可靠**。其中一种方法是避免使用**硬编码值**。

代码示例

```
<!-- This TextView has its width hardcoded into it. -->
<LinearLayout
    android:layout_width="100dp"
    android:layout_height="wrap_content"
    android:orientation="vertical">
    <TextView
        android:layout_width="100dp"
        android:layout_height="wrap_content"
        android:text="Hello"/>
</LinearLayout>

<!-- This TextView is more robust. It gets its width from its parent. -->
<LinearLayout
    android:layout_width="100dp"
    android:layout_height="wrap_content"
    android:orientation="vertical">
    <TextView
        android:layout_width="match_parent"
        android:layout_height="wrap_content"
        android:text="Hello World"/>
</LinearLayout>
```

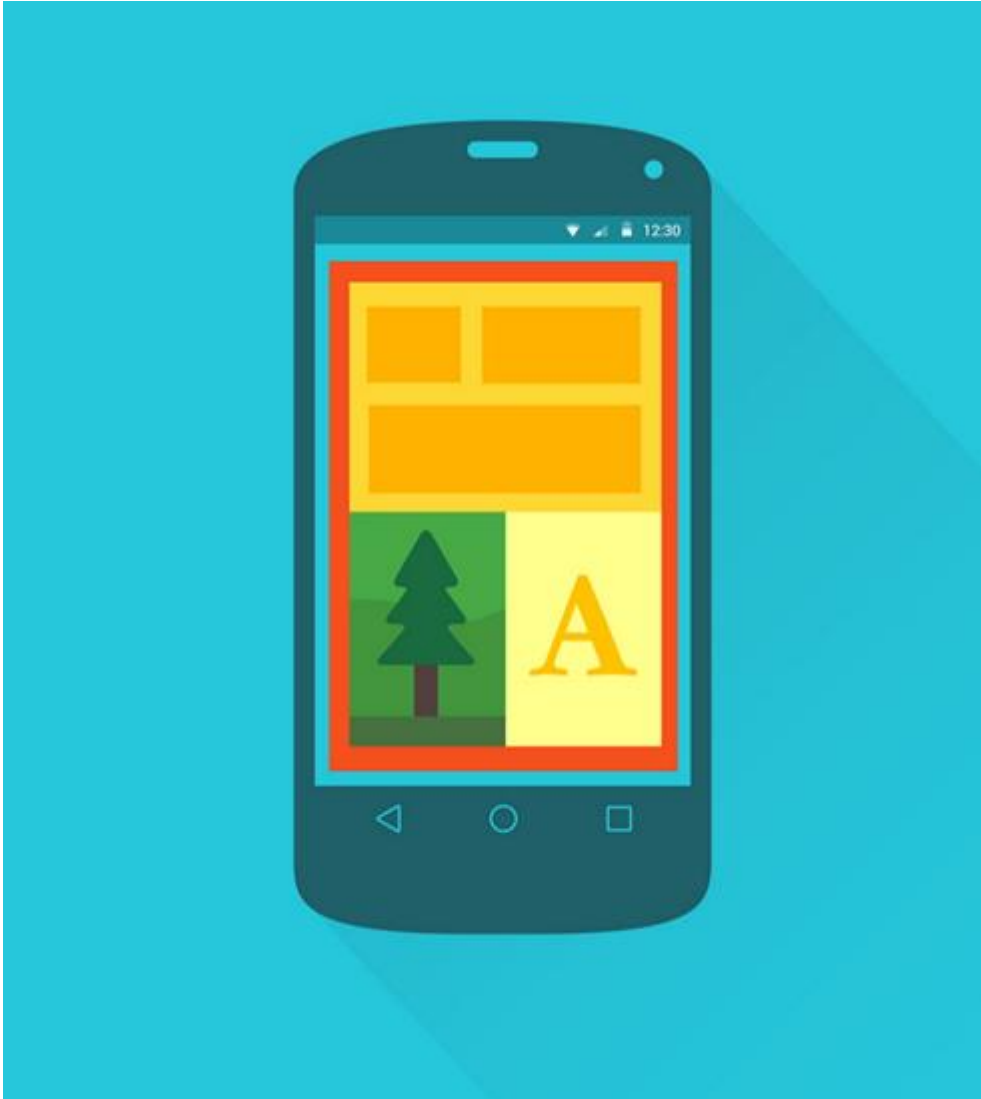


Program is not **robust** if a change in one place requires a change in another.

Root View（根视图）

定义

视图是屏幕上的矩形区域。大视图可以包含较小视图，较小视图依次还可以包含更小的视图。在任意给定时刻，全部视图中的最大视图（即包含所有其他视图的视图）称为**根视图**。



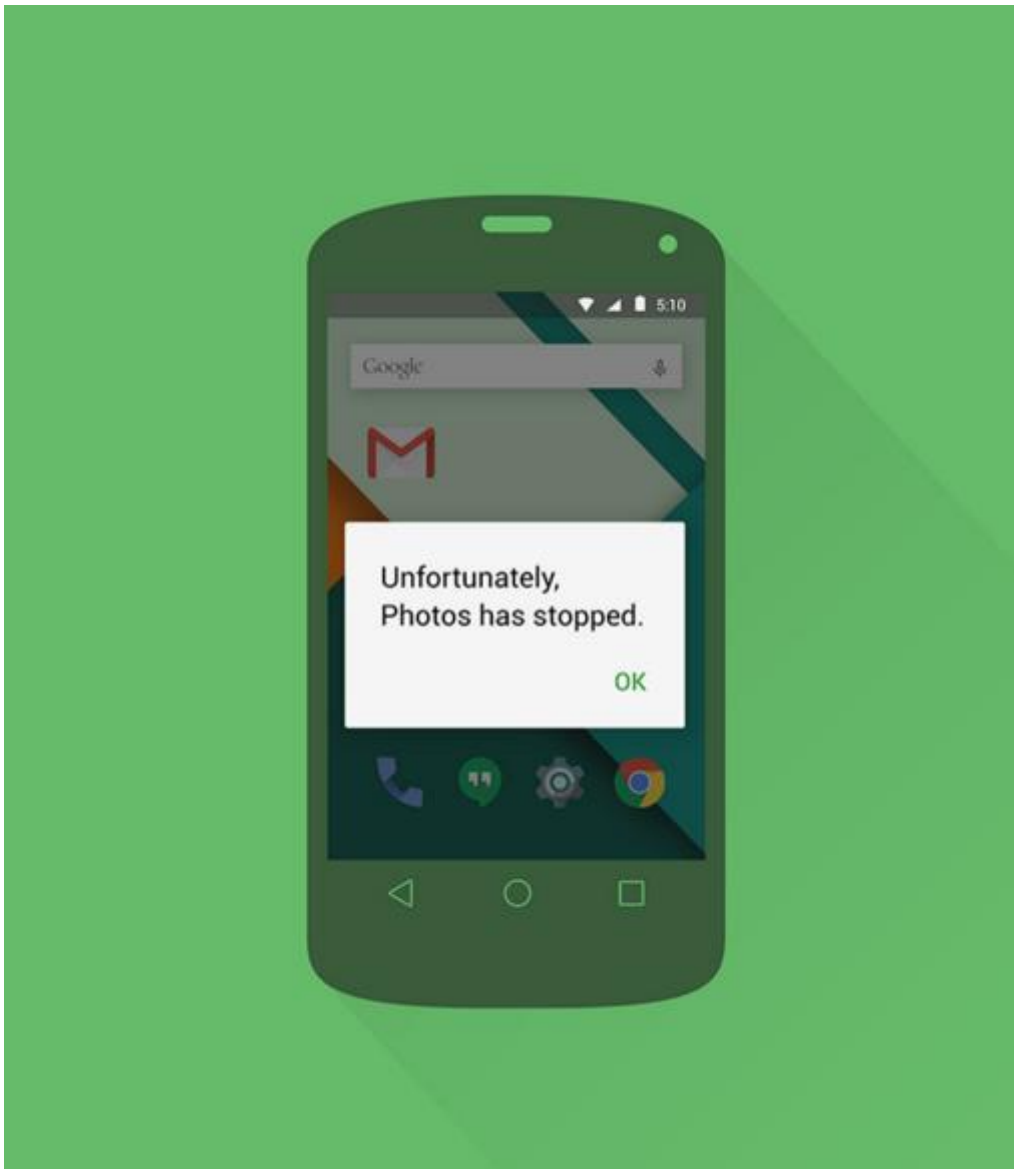
Runtime Error（运行时错误）

定义

计算机是按照一系列称为**程序**的指令运行的机器。**Android 设备**便是计算机，**应用**是程序。应用采用**Java** 语言编写。

在将应用**编译**（翻译）成设备自己的**本机语言**（即 1 和 0）前，无法执行应用。编译过程期间可检测应用中特定种类的错误，而非全部。

在**运行时**执行应用时，应用指令不会告知设备应用的发起人意图做什么。这些指令甚至可以令设备进入到一种无法继续执行下一条指令的状态。当这种情况发生时，将显示**运行时错误消息**，我们称此应用已崩溃。



Setter Method（Setter 方法）

定义

计算机是按照一系列称为**程序**的指令运行的机器。**Android 设备**便是计算机，**应用**是程序。设备内部是称为**变量**的容器，用于存储数字或文本片段等**值**。

对象是变量，但在以下两个方面特殊。第一，对象中可包含更小的变量，即对象的**域**。例如，表示房屋的对象可能包含一个 **color** 域。第二，我们可向对象附加称为**方法**的一系列指令，实际上是小程序。

House 对象中可具有 **setColor** 方法，用于将房屋设置为给定颜色，还可以具有 **getColor** 方法，用于**返回**（至开发人员）房屋的当前颜色。用于设置和获取域值的一对方法称为 **getter** 和 **setter**。

对象分多种**类**（类型）。针对每个类，我们必须编写**定义**：即属于各个类对象的域和方法列表。每个给定类的对象都有一组相同的域和方法。例如，每个 **house** 对象必须具有称为 **color** 的域和称为 **setColor** 的方法。但是每个 **house** 对象都可在其 **color** 域中包含不同的值：可以一个房屋为红色，而另一个为蓝色。

可以指定对象类各个域和方法的**访问修饰符**。例如，可以将颜色字段设为 **private**，即除 **House** 类的方法外，任何其他类的方法都不能引用。可以将域的 **getter** 和 **setter**（**getColor** 和 **setColor** 方法）设为 **public**，即应用中任意类的方法都可以引用。这样一来，如果 **House** 类中再无其他可以获取和设置字段的 **public** 方法，则字段已**封装**，即只能通过 **getter** 和 **setter** 进行访问。

sp (Scale-Independent Pixel)（与比例无关的像素）

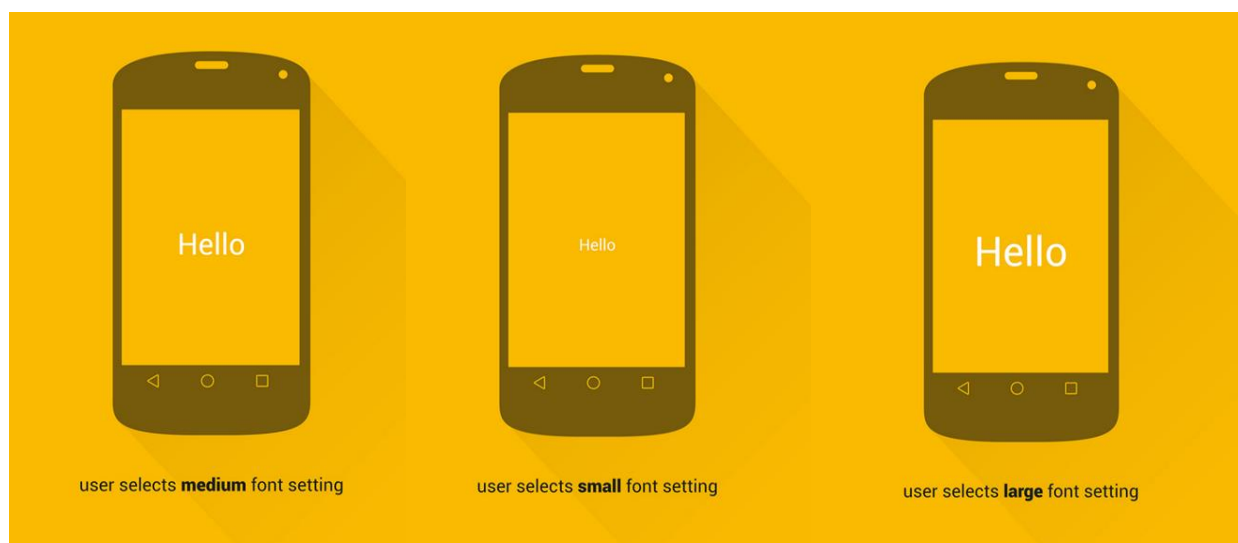
定义

与比例无关的像素 (sp) 是用于指定字体类型大小的长度单位。其长度取决于用户的字体大小首选项。该首选项在 Android 设备的“设置”应用中设置。

为尊重用户的首选项，应使用与比例无关的像素指定所有字体大小。应在**与设备无关的像素 (dp)** 中给定所有其他尺寸。

代码示例

```
<TextView  
    android:layout_width="wrap_content"  
    android:layout_height="wrap_content"  
    android:padding="8dp"  
    android:textSize="20sp"  
    android:text="Hello"/>
```



Stack Trace（堆栈跟踪）

定义

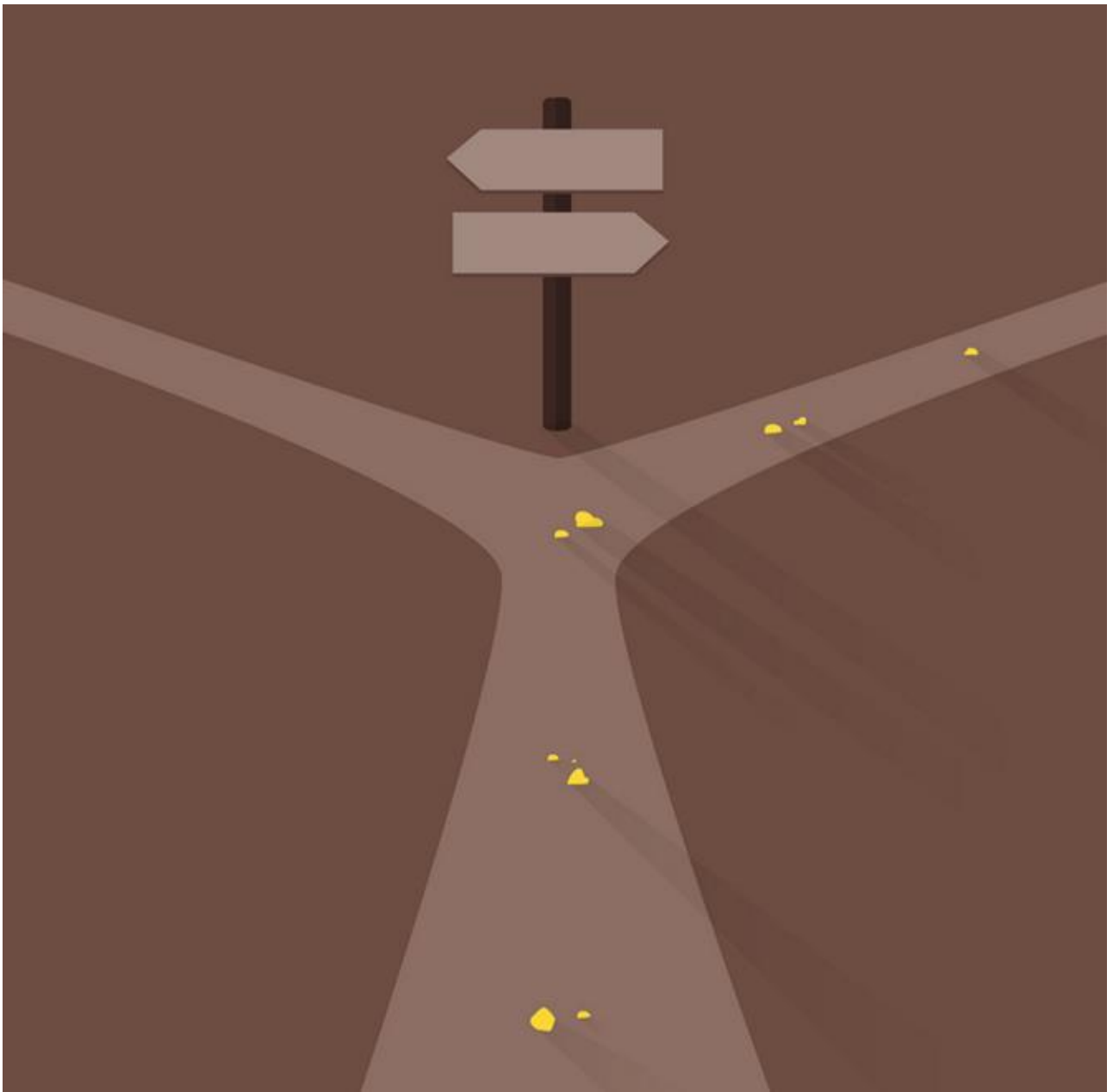
计算机是按照一系列称为**程序**的指令运行的机器。**Android 设备**便是计算机，**应用**是程序。

应用划分为多个部分，称为**方法**。一个方法中的指令可指挥设备另一个方法中的指令。出现这种情况时，我们称第一个方法已**调用**第二个方法。完全执行第二个方法后，设备将从离开第一个方法时所处的点开始继续执行第一个方法中的指令。出现这种情况时，我们称第二个方法已**返回**到第一个方法的特定点。

第一个方法可以调用第二个方法，而第二个方法也可以调用第三个方法。（事实上，**方法调用**的深度可远超出此。）出现这种情况时，设备必须记住在完成第三个方法后将返回到的第二个方法中的点。设备还必须记住在完成第二个方法后将返回到的第一个方法中的点。

设备通过将这些信息堆叠成称为**堆栈**的堆来记住这些信息片段。调用每个方法时，计算机最终必须返回到的点存储在堆栈顶层，因此将会越来越高。设备从各个方法返回后，系统会从堆栈中移除最高点并返回到该目标。在任意给定时刻，堆栈均包含所有方法的踪迹，即设备通过传递哪些方法到达当前正在执行的方法，以及必须通过返回到哪些方法才能最终回到起点。

堆栈的内容显示称为**堆栈跟踪**。它显示了设备当前执行的方法以及到达该方法所经过的路径，也就是设备将来返回到起点要经过的路径（但顺序相关）。若设备到达错误的方法，跟踪可帮助我们了解问题所在。



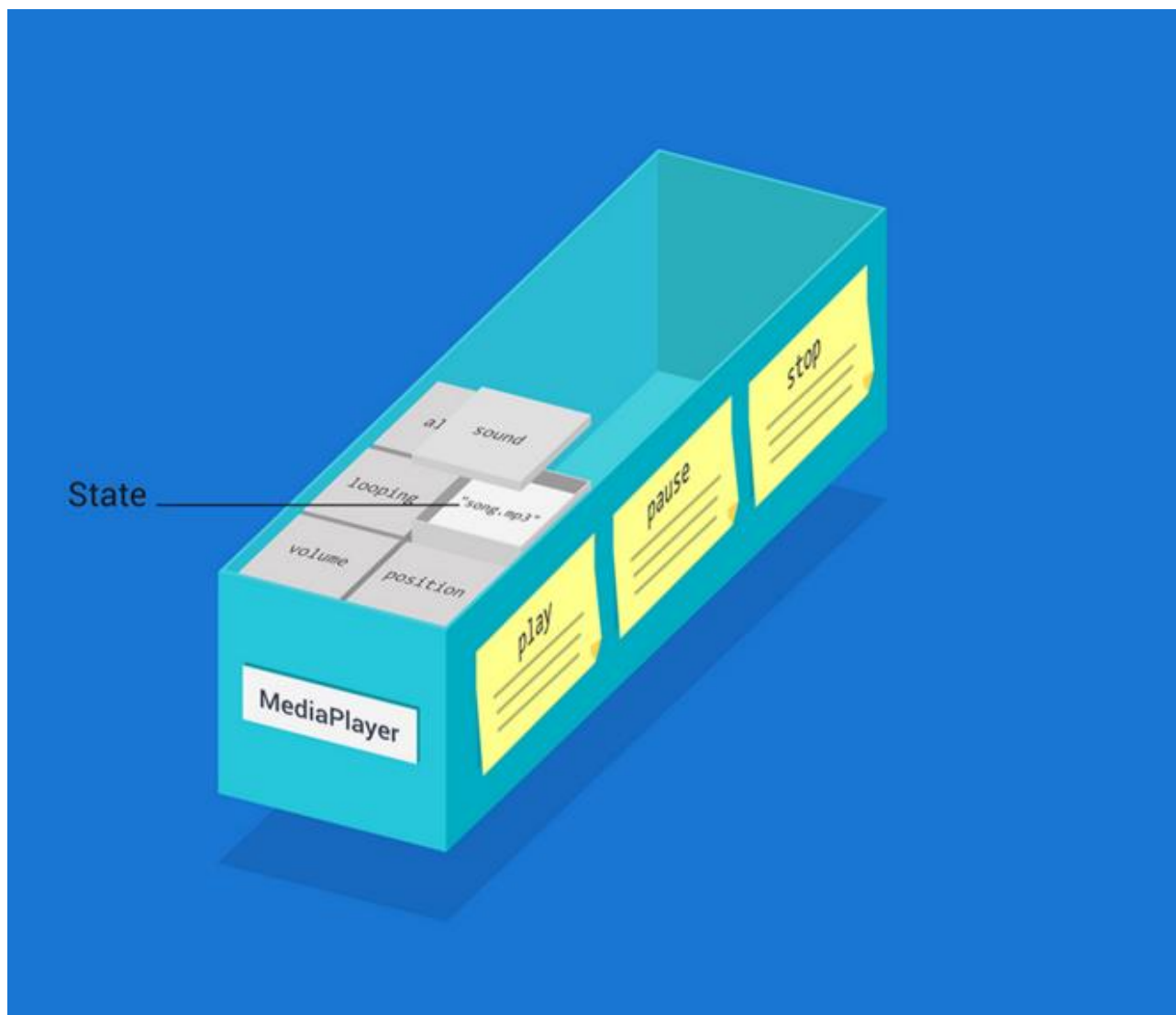
State（状态）

定义

计算机是按照一系列称为**程序**的指令运行的机器。**Android 设备**便是计算机，**应用**是程序。

设备内部是称为**变量**的容器，用于存储数字或文本片段等**值**。内部可包含若干个较小变量的变量称为对象，这些较小变量称为对象的**域**。例如，在 **MediaPlayer** 对象中可能包含多个域，用于存储正在播放的声音文件的名称、音量等级、文件回放的当前位置以及指示是否采用无限循环方式播放文件。

对象域的内容构成了对象的**状态**。在 **MediaPlayer** 播放声音文件时，其状态将会发生变化：随着文件的向前播放，用于存储回放位置的域将增加，而且用户可能会更改音量。



String（字符串）

定义

由字母、数字、标点符号、空格组成的任意字符系列称为字符的**字符串**。字符串中的字符数称为字符串的**长度**。字符串中可能包括单词或随机选取的若干字符。也可以只包括单一字符，甚至根本不包括字符。后者称为**空字符串**，是唯一长度为零的字符串。

计算机是按照一系列称为**程序**的指令运行的机器。**Android 设备**便是计算机，**应用**是程序。**变量**是设备内部的一种容器。很多变量都用于存储数字，但称为**字符串变量**的变量类型能够存储字符串。

代码示例

```
// Create a variable named firstName that can hold a string, and put a string into it.
String firstName = "JOE";
// Display the string in a piece of Toast.
Toast.makeText(this, firstName, Toast.LENGTH_SHORT).show();
// Create a variable that holds the shortest possible string.
String emptyString = "";
// Display the string in a TextView.
textView.setText(emptyString);
```



Style（样式）

定义

视图是屏幕上的矩形区域。例如，**TextView** 显示文本，**ImageView** 显示图像。

当 **TextView** 未指定前景或背景颜色时，将使用默认设置，如透明背景加灰色文本。这是因为 **TextView** 应用了默认属性集，我们称之为**样式**。

插图中显示了 **BlackOnYellow** 和 **WhiteOnGreen** 两种样式，样式可在 **styles.xml** 文件中进行创建。之后可轻松应用到另一 XML 文件的 **TextView** 中。

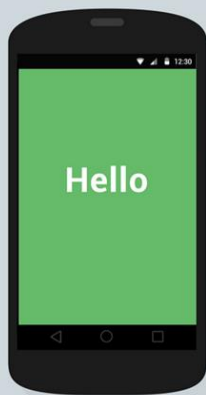
代码示例

```
<!-- Create the styles in the file styles.xml. -->
<resources>
    <style name="BlackOnYellow">
        <item name="android:background">#FFFF00</item>
        <item name="android:textColor">#000000</item>
    </style>
    <style name="WhiteOnGreen">
        <item name="android:background">#00FF00</item>
        <item name="android:textColor">#FFFFFF</item>
    </style>
</resources>

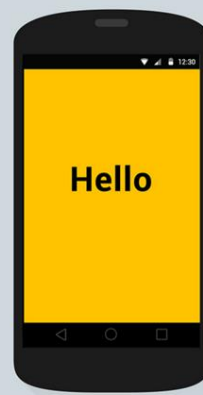
<!-- Use the styles in the file activity_main.xml. -->
<TextView
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    android:gravity="center"
    style="@style/BlackOnYellow"
    android:text="Hello"/>
```

<TextView

```
android:layout_width="match_parent"  
android:layout_height="match_parent"  
android:gravity="center"  
style="@style/WhiteOnGreen"  
android:text="Hello"/>
```



WhiteOnGreen style



BlackOnYellow style

Subclass（子类）

定义

计算机是按照一系列称为**程序**的指令运行的机器。**Android 设备**便是计算机，**应用**是使用 **Java** 语言编写的程序。设备内部是称为**变量**的容器，用于存储数字或文本片段等**值**。

对象是变量，但在以下两个方面特殊。第一，对象中可包含更小的变量，即对象的**域**。例如，**TextView** 对象在屏幕上显示文本，并且可能包含称为 **mText** 的域。第二，我们可向对象附加称为**方法**的一系列指令，实际上是小程序。**TextView** 对象可能具有称为 **setText** 的方法，可将文本片段放入对象的 **mText** 域中。

对象分多种**类**（类型）。针对每个类，我们必须编写**定义**：即属于各个类对象的域和方法列表。每个给定类的对象都有一组相同的域和方法。例如，每个 **TextView** 对象必须具有称为 **mText** 的域和称为 **setText** 的方法。但是每个 **TextView** 都可在其 **mText** 域中包含不同的值：一个 **TextView** 可能会说“您好”而另一个说“再见”。

定义类即意味着通过编写类定义来创建类。之后我们可以创建类的对象，但这是单独的操作。我们完全可以在不创建类对象的情况下定义类。

可通过列出属于各个类对象的每个域和方法，从头开始定义一个新类。我们也可以使新类自动具有某现有类的所有域和方法，并在此基础上加入新类定义中列出的附加域和方法，以此来创建新类。在此场景中，现有类称为**超类**或**基类**，而新类是其**子类**。我们称子类继承了其超类的所有域和方法，并且子类是使用**继承**通过超类创建的。之所以称为“子类”和“超类”，是因为在图表中，我们始终在超类下方绘制子类。请注意，子类通常会比其超类具有更多方法和域。

让我们看个例子。**EditText** 类的对象不仅可以执行 **TextView** 类的对象能够执行的所有操作，还可以执行其他操作。它可以在屏幕上显示文本，还允许用户编辑该文本。因此，**EditText** 类是使用继承根据 **TextView** 类创建的，并且具有支持编辑的附加域和方法。

代码示例

```
// A simplified superclass (class TextView in the file TextView.java).  
// Each object of the class has a simple setText method that allows  
// a String to be stored in the object.
```

```
public class TextView extends View {  
    // the text to be displayed  
    private String mText;  
    public void setText(String text) {  
        mText = text;  
    }  
}  
  
// A simplified subclass (class EditText in the file EditText.java).  
// Each object of this class has a more elaborate setText method  
// that allows a String to be edited as well as stored.  
public class EditText extends TextView {  
  
    @Override  
    public void setText(String text) {  
        // Call the original setText method inherited from class TextView.  
        super.setText(text);  
        // Additional instructions to allow the text to be edited go here.  
    }  
}
```

| | occupies memory | appears on screen | displays text | and lets you edit the text |
|----------------|--------------------|----------------------|------------------|-------------------------------|
| class Object | ✓ | | | |
| class View | ✓ | ✓ | | |
| class TextView | ✓ | ✓ | ✓ | |
| class EditText | ✓ | ✓ | ✓ | ✓ |

Class Object is the **superclass** of class View.
Class View is a **subclass** of class Object.

Superclass or Base Class（超类或基类）

定义

计算机是按照一系列称为**程序**的指令运行的机器。**Android 设备**便是计算机，**应用**是使用 **Java** 语言编写的程序。设备内部是称为**变量**的容器，用于存储数字或文本片段等**值**。

对象是变量，但在以下两个方面特殊。第一，对象中可包含更小的变量，即对象的**域**。例如，**TextView** 对象在屏幕上显示文本，并且可能包含称为 **mText** 的域。第二，我们可向对象附加称为**方法**的一系列指令，实际上是小程序。**TextView** 对象可能具有称为 **setText** 的方法，可将文本片段放入对象的 **mText** 域中。

对象分多种**类**（类型）。针对每个类，我们必须编写**定义**：即属于各个类对象的域和方法列表。每个给定类的对象都有一组相同的域和方法。例如，每个 **TextView** 对象必须具有称为 **mText** 的域和称为 **setText** 的方法。但是每个 **TextView** 都可在其 **mText** 域中包含不同的值：一个 **TextView** 可能会说“您好”而另一个说“再见”。

定义类即意味着通过编写类定义来创建类。之后我们可以创建类的对象，但这是单独的操作。我们完全可以在不创建类对象的情况下定义类。

可通过列出属于各个类对象的每个域和方法，从头开始定义一个新类。我们也可以使新类自动具有某现有类的所有域和方法，并在此基础上加入新类定义中列出的附加域和方法，以此来创建新类。在此场景中，现有类称为**超类或基类**，而新类是其**子类**。我们称子类继承了其超类的所有域和方法，并且子类是使用**继承**通过超类创建的。之所以称为“子类”和“超类”，是因为在图表中，我们始终在超类下方绘制子类。请注意，子类通常会比其超类具有更多方法和域。

让我们看个例子。**EditText** 类的对象不仅可以执行 **TextView** 类的对象能够执行的所有操作，还可以执行其他操作。它可以在屏幕上显示文本，还允许用户编辑该文本。因此，**EditText** 类是使用继承根据 **TextView** 类创建的，并且具有支持编辑的附加域和方法。

代码示例

```
// A simplified superclass (class TextView in the file TextView.java).
```

```
// Each object of the class has a simple setText method that allows  
// a String to be stored in the object.
```

```
public class TextView extends View {
```

```
    // the text to be displayed
```

```
    private String mText;
```

```
    public void setText(String text) {
```

```
        mText = text;
```

```
    }
```

```
}
```

```
// A simplified subclass (class EditText in the file EditText.java).
```

```
// Each object of this class has a more elaborate setText method
```

```
// that allows a String to be edited as well as stored.
```

```
public class EditText extends TextView {
```

```
    @Override
```

```
    public void setText(String text) {
```

```
        // Call the original setText method inherited from class TextView.
```

```
        super.setText(text);
```

```
        // Additional instructions to allow the text to be edited go here.
```

```
    }
```

```
}
```

| | occupies memory | appears on screen | displays text | and lets you edit the text |
|----------------|--------------------|----------------------|------------------|-------------------------------|
| class Object | ✓ | | | |
| class View | ✓ | ✓ | | |
| class TextView | ✓ | ✓ | ✓ | |
| class EditText | ✓ | ✓ | ✓ | ✓ |

Class Object is the **superclass** of class View.
Class View is a **subclass** of class Object.


System Log（系统日志）

定义

手机或平板电脑等 **Android 设备** 也会写日记，称为**系统日志**，用于记录在设备上执行的操作和已发生的事件。这些事件包括启用应用、打开数据库和建立网络连接等。

同样重要的是，日志也记录执行失败的事件：应用启动失败、数据库打开失败以及失败的原因。系统日志还可以显示通过应用调用 [Log](#) 方法打印出的信息。

请参阅[查看系统日志](#)相关说明。



```
12:30:32 I/ActivityManager: Start proc com.google.myactivity
12:30:33 I/PackageManager: Copying native libraries
12:20:34 I/PowerManagerService: Waking up from sleep...
12:20:35 D/ConnectivityService: handleConnectivityChange
```

Text Localization（文本本地化）

定义

若用户转到设置应用，将设备语言更改为“西班牙语”，此时应用会察觉到此[配置更改](#)并做出相应响应。针对[更改区域设置](#)准备相应的应用，称为**本地化**。接下来我们对语言和地区为 "Español (Estados Unidos)" 的应用进行本地化，例如“西班牙语（美国）”。

文本片段称为**字符串**。应用在屏幕上显示的所有字符串均应收集到名为 `strings.xml` 的[资源文件](#)中。此文件已经是应用的一部分，位于文件夹 `app/res/values` 中。在 **Android Studio** 左侧面板中可以看到该文件夹和文件，这个面板称为[项目视图](#)。

接下来我们将新建名为 `app/res/values-es-rUS` 的文件夹，在该文件夹中新建名为 `strings.xml` 的文件。新文件称为[备选资源](#)，其中包含原始 `strings.xml` 文件中字符串的西班牙语译文。

在项目视图中，选择文件夹 `app/res`。从菜单栏中，下拉

文件 (File) → 新建 (New) → Android 资源目录 (resource directory)

填写随即出现的新建资源目录 (New Resource Directory) 表单。

目录名称 (Directory name): `values`

资源类型 (Resource type): `values`

源集 (Source set): `main`

在“可用限定符 (Available qualifiers)”下，选择区域设置 (Locale) 然后按下 >> 按钮。在语言 (Language) 下，选择 "es:Spanish"。在仅具体地区 (Specific Region Only) 下，选择 "US:United States"。

可以观察到表单顶部的目录名称已更改为 `values-es-rUS`。按下确定 (OK) 按钮。

返回项目视图，选择文件夹 `app/res/values`。从菜单栏中，下拉

文件 (File) → 新建 (New) → Values 资源文件 (Values resource file)

填写随即出现的新建资源文件 (New Resource File) 表单。

文件名 (File name): `strings`

源集 (Source set): `main`

目录名称 (Directory name): `values`

在“可用限定符 (Available qualifiers)”下，选择区域设置 (Locale) 然后按下 >> 按钮。在语言 (Language) 下，选择 "es:Spanish"。在仅具体地区 (Specific Region Only) 下，选择 "US:United States"。可以观察到目录名称已更改为 values-es-rUS。按下确定 (OK) 按钮。

在 Android Studio 的中央面板中，单击新标签 es-rUS/strings.xml。编辑新文件 strings.xml，使文件中的每条字符串资源均变成原始 strings.xml 文件中相应资源的译文。例如，如果原始 strings.xml 包含资源

```
<string name="hello_world">Hello world!</string>
```

则新文件 strings.xml 应包含

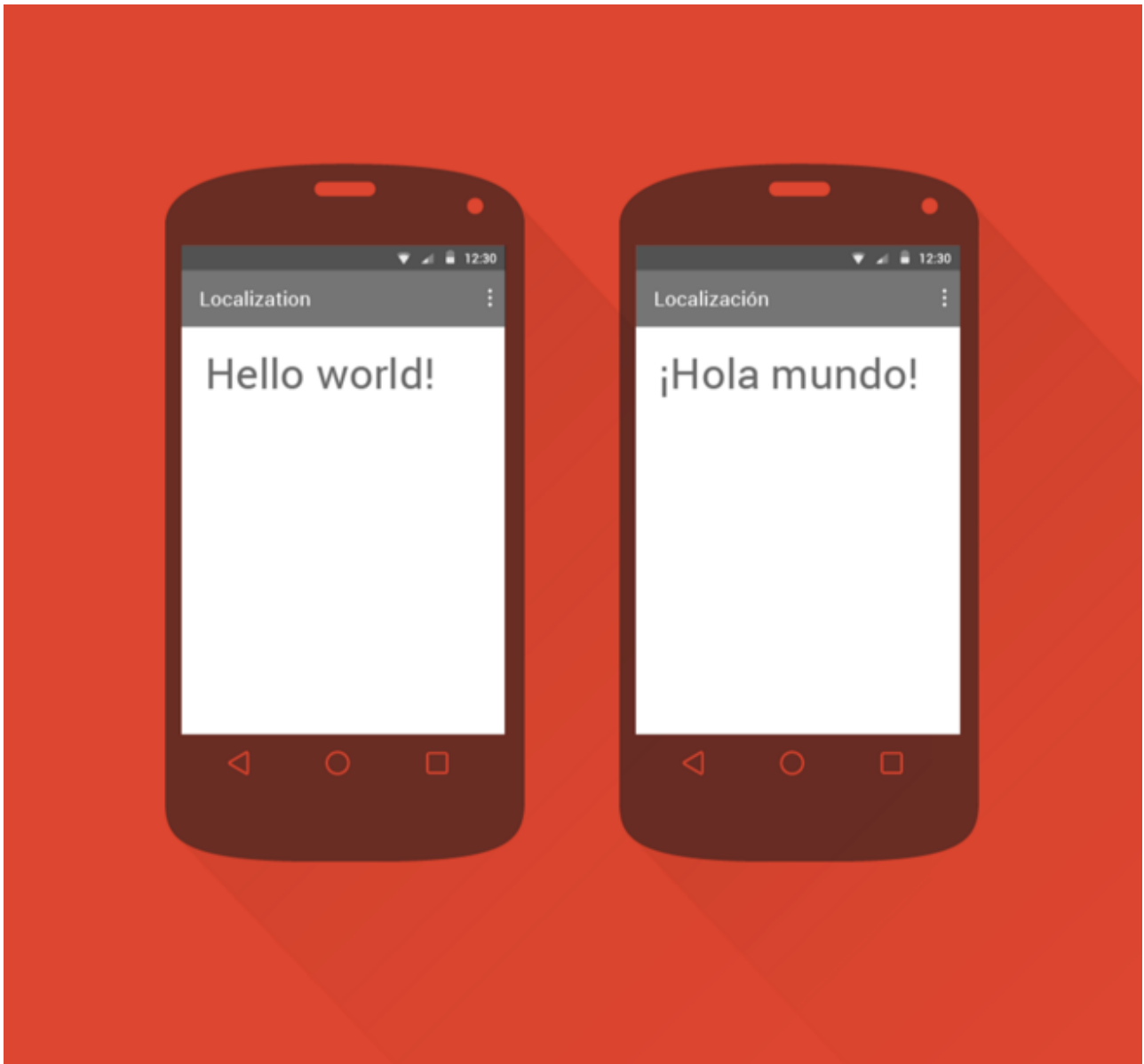
```
<string name="hello_world">¡Hola mundo!</string>
```

现在，应用将根据用户在设置应用中的语言首选项，选择从 res/values/strings.xml 或 res/values-es-rUS/strings.xml 中获取字符串。当然，这只是本地化的第一步。我们还应该对数字、日期、时间等显示格式进行本地化。而且从右到左的语言首选右对齐。

代码示例

```
<!-- This is the original strings.xml in the folder app/res/values. -->
<resources>
    <string name="app_name">Localization</string>
    <string name="hello_world">Hello world!</string>
    <string name="action_settings">Settings</string>
</resources>

<!-- This is the new strings.xml in the folder app/res/values-es-rUS. -->
<resources>
    <?xml version="1.0" encoding="utf-8"?>
    <string name="app_name">Localización</string>
    <string name="hello_world">¡Hola mundo!</string>
    <string name="action_settings">Configuración</string>
</resources>
```



TextView

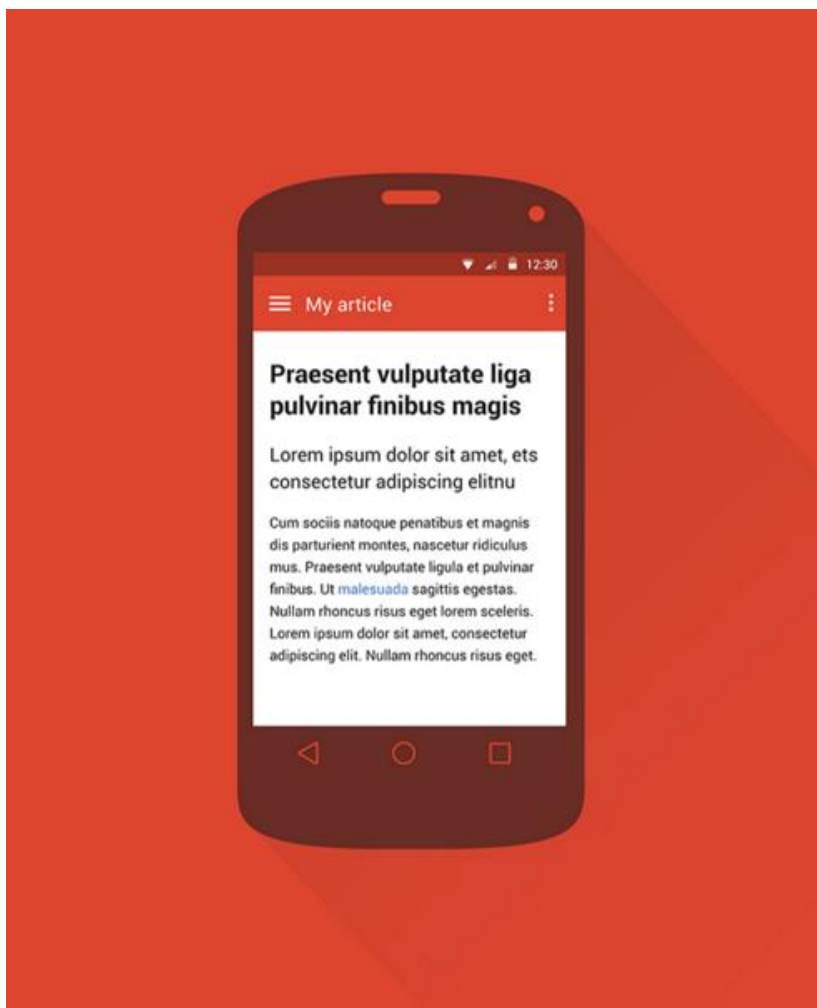
定义

视图是屏幕上的矩形区域。**TextView** 是一种视图类型，显示一行或多行文本。

屏幕上的 **TextView** 由 Android 设备内部的 **Java** 对象绘制。事实上，**Java** 对象是真正的 **TextView**。但是在谈论用户所见时，将屏幕上的矩形区域视为 "**TextView**" 比较方便。

代码示例

```
<TextView
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:padding="8dp"
    android:textSize="24sp"
    android:text="Hello"/>
```

Theme（主题）

定义

视图是屏幕上的矩形区域。例如，**TextView** 显示文本，**ImageView** 显示图像。

当 **TextView** 未指定文本颜色或背景颜色时，将使用默认设置，如透明背景加灰色文本。这是因为 **TextView** 应用了默认属性集，我们称之为**样式**。

可以轻松为各 **TextView** 应用不同的样式。但通常我们希望应用中的所有 **TextView** 以及按钮等其他视图都使用相同的样式。这可以通过**主题**来实现，主题是可以自动应用到多种视图的样式。一个主题可以应用到活动对象创建的所有视图，甚至应用中的所有视图。

当然，并非每个属性都适用于所有视图。例如，**TextView** 和 **Button** 具有某种字体，但 **ImageView** 则不具有。主题指定的各个默认属性仅会应用到适用的视图。

Android 提供若干个已设置好的主题。例如，**Theme.Material** 主题为大多数视图（如 **TextView** 的文本）的**内容**赋予亮色。而使大多数视图的背景呈暗色，可能是因为背景实际为暗色，也可能是因为背景是透明的而背景后面的视图是暗色的。整体结果将使应用的**内容区域**主要呈暗色。

而 **Theme.Material.Light** 主题恰好相反：使内容为暗色，而内容区域为亮色。

Theme.Material.Light.DarkActionBar 主题与 **Theme.Material.Light** 主题一样，只不过其应用栏（最近更名为操作栏）使用的是亮色内容和暗色背景。

代码示例

```
<!-- In the file styles.xml, create a theme named MyTheme. It will be the same as the existing theme Theme.AppCompat.Light.DarkActionBar, except that its textColor will be green. -->
<resources>
  <style name="MyTheme"
    parent="Theme.AppCompat.Light.DarkActionBar">
    <item name="android:textColor">#00FF00</item>
  </style>
</resources>
```

<!-- In the file AndroidManifest.xml, apply the theme to all the Views created by the app. -->

<application

```
    android:allowBackup="true"
    android:icon="@mipmap/ic_launcher"
    android:label="@string/app_name"
    android:theme="@style/MyTheme"/>
```

<!-- In the file activity_main.xml, the MyTheme is automatically applied to the TextView. -->

<TextView

```
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:text="Hello"/>
```



User Interface（用户界面）

定义

应用的用户界面是我们在 Android 设备屏幕上所看到的内容。用户界面包括一个或多个用于显示信息的矩形区域，称为视图。部分视图（如按钮）还可以对触摸做出响应。



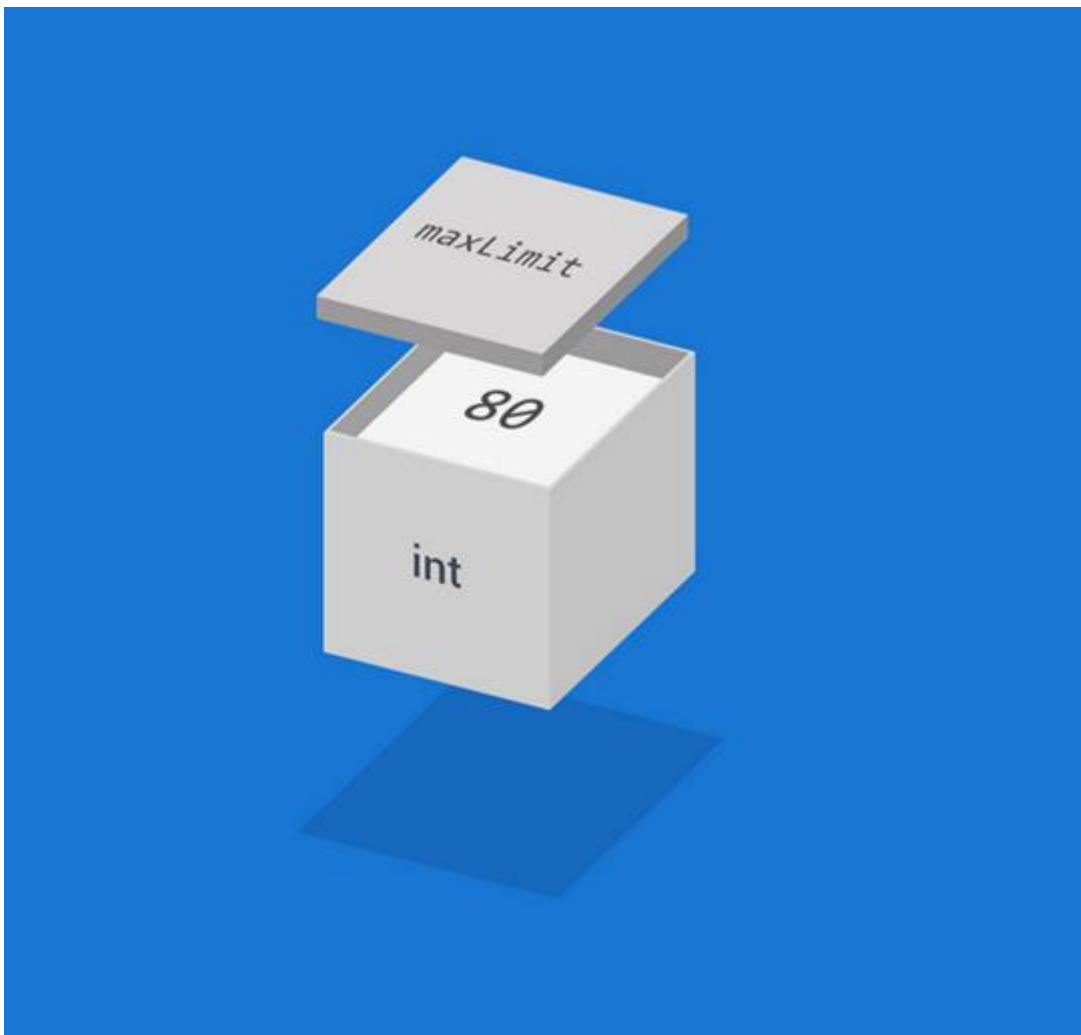
Variable（变量）

定义

计算机是按照一系列称为**程序**的指令运行的机器。**Android 设备**便是计算机，**应用**是程序。

在 Android 设备内部，**变量**是一种保存数量或文本等**值**的容器。例如，存储比赛当前分数或餐馆名称的变量。每个变量都有一个**名称**，例如 "currentScore" 或 "restaurantName"。插图中显示了名为 maxLimit 的变量，包含值 80。

当应用运行时，变量的内容可由 Android 设备更改。这也是将其称为“变量”的原因。



Variable Declaration（变量声明）

定义

计算机是按照一系列称为**程序**的指令运行的机器。**Android 设备**便是计算机，**应用**是程序。

变量是 Android 设备中用于存储数字或文本片段等**值**的容器。数字和文本片段只是许多允许的变量**类型**中的两种。当应用运行时，变量的内容可由设备更改。这也是将其称为“变量”的原因。

每个变量都有一个**名称**，例如 "currentScore"、"restaurantName" 或 "textView"。由于变量名称是我们发明的单词，并且设备之前从未见过，因此我们必须在应用中编写指令，通知设备该单词是变量的名称以及变量所属的类型。这个指令称为**变量声明**。变量声明创建变量，也使得可以在后续应用指令中提及变量名称。插图显示了名为 "firstName" 的变量的声明。



Variable Name（变量名）

定义

变量是我们在 Android 设备内部创建的容器。每个变量均具有一个名称，称为其**标识符**。在 Java 语言中，标识符可由字母、数字和下划线组成，按照惯例，变量的标识符必须以小写字母开头。标识符中不允许空格字符，这意味着标识符必须为单个单词。

标识符中的字母可以是大写或小写，但该选择区分大小写。换言之，"jamesbond" 和 "jamesBond" 无法用作同一变量的名称。我们只能选用其中的一个名称。



Variable Scope（变量范围）

定义

计算机是按照一系列称为**程序**的指令运行的机器。**Android 设备**便是计算机，**应用**是程序。由于设备尚未可靠地理解人类语言，所以我们必须以**编程语言**（例如 **Java**）编写应用。

设备内部是称为**变量**的容器，用于存储数字或文本片段等值。每个变量都有一个**名称**，例如 "x"、"y" 或 "shoppingCart"。

Java 语言的规则允许仅在应用的一个部分中提及给定变量的名称。这个部分称为变量的**范围**，类似于动植物物种的栖息地或范围。在变量范围之外试图提及变量名称将会出现错误。

一些变量具有非常有限的范围。例如，应用的指令分组到称为**方法**的部分，而在方法中创建的变量只能在该方法内提及。其余在所有方法之外创建的变量适用范围更广。

代码示例

```
// The scope of a variable such as textView, created  
// inside of the curly braces of a method, extends from  
// the statement in which the variable was created to  
// the curly brace that closes the method.  
  
@Override  
protected void onCreate(Bundle savedInstanceState) {  
    super.onCreate(savedInstanceState);  
    setContentView(R.layout.activity_main);  
    TextView textView = (TextView) findViewById(R.id.textView);  
    textView.setText("Hello");  
}
```



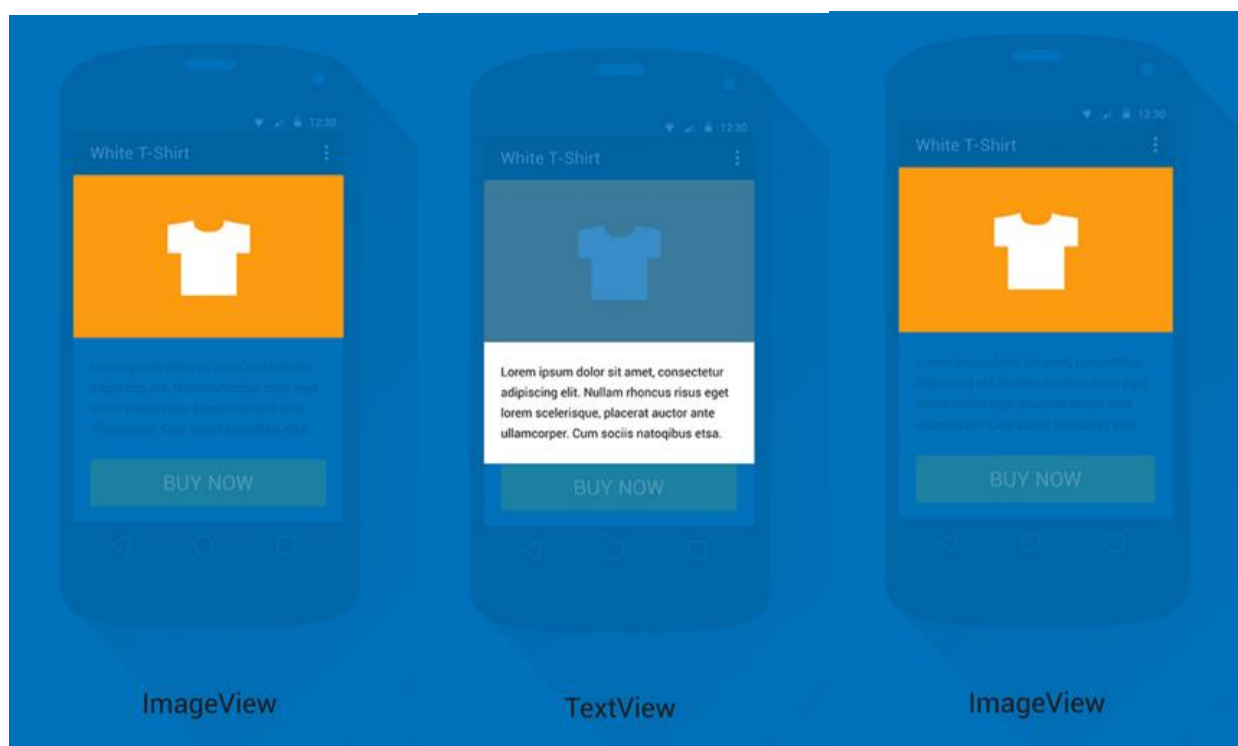
View（视图）

定义

视图是屏幕上可见的矩形区域。视图具有宽度和高度，有时还具有背景色。

插图显示了三种不同类型的视图。**ImageView** 显示图像，如图标或照片。**TextView** 显示文本。**Button** 是对触摸敏感的 **TextView**：用手指点击时即会做出响应。**ViewGroup** 是大视图，通常不可见，其内部包含并可放置较小视图。

屏幕上的视图由 **Android** 设备内部的 **Java** 对象绘制。事实上，**Java** 对象是真正的**视图**。但是在谈论用户所见时，将屏幕上的矩形区域视为“视图”比较方便。



ViewGroup

定义

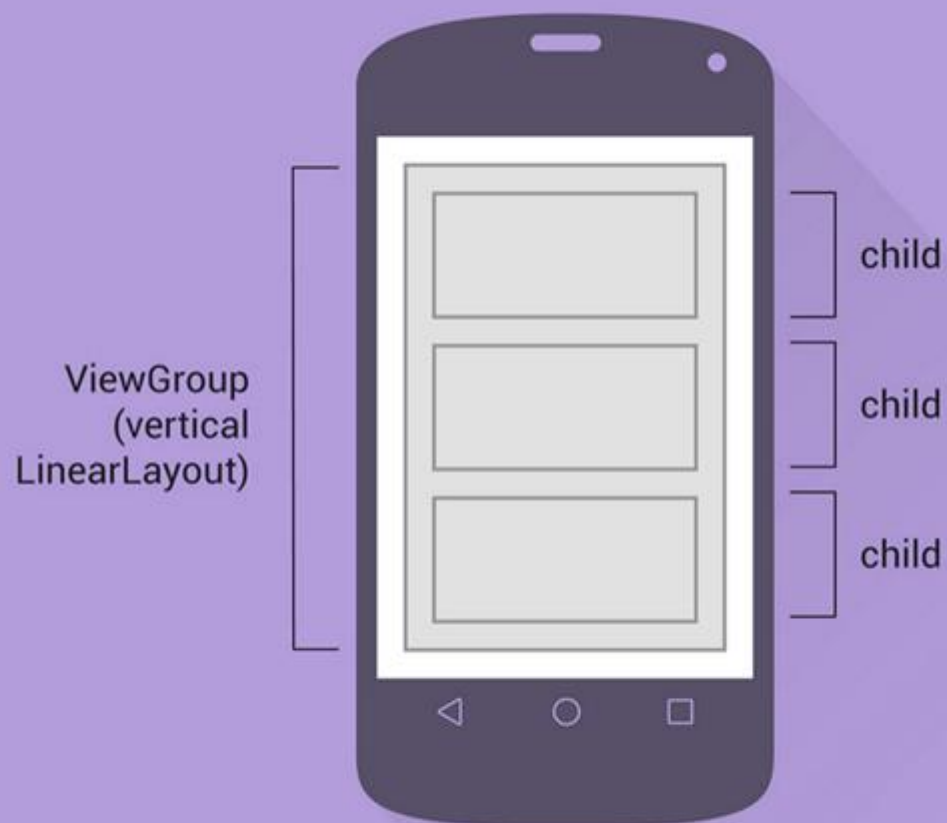
视图是屏幕上的矩形区域。例如，**TextView** 显示文本，**ImageView** 显示图像。

ViewGroup 是大视图，其中可包含小视图。小视图称为 ViewGroup 的**子视图**，可以是 **TextView** 或 **ImageView**。ViewGroup 称为其子视图的**父视图**。插图显示了最常见的 ViewGroup 之一，即垂直的 **LinearLayout**。

ViewGroup 本身可能是透明的，仅用于包含及放置其子视图。但其子视图几乎始终可见。

代码示例

```
<LinearLayout
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    android:orientation="vertical"/>
    <TextView
        android:layout_width="match_parent"
        android:layout_height="wrap_content"
        android:text="some content"/>
    <TextView
        android:layout_width="match_parent"
        android:layout_height="wrap_content"
        android:text="some content"/>
    <TextView
        android:layout_width="match_parent"
        android:layout_height="wrap_content"
        android:text="some content"/>
</LinearLayout>
```



View Hierarchy（视图层次结构）

定义

视图是屏幕上的矩形区域。例如，**TextView** 显示文本，**ImageView** 显示图像。

ViewGroup 是大视图，其中可包含小视图。小视图称为 ViewGroup 的**子视图**，可以是 TextView 或 ImageView。ViewGroup 称为其子视图的**父视图**。每个子视图都**嵌套**（完全包含）在其父视图内。

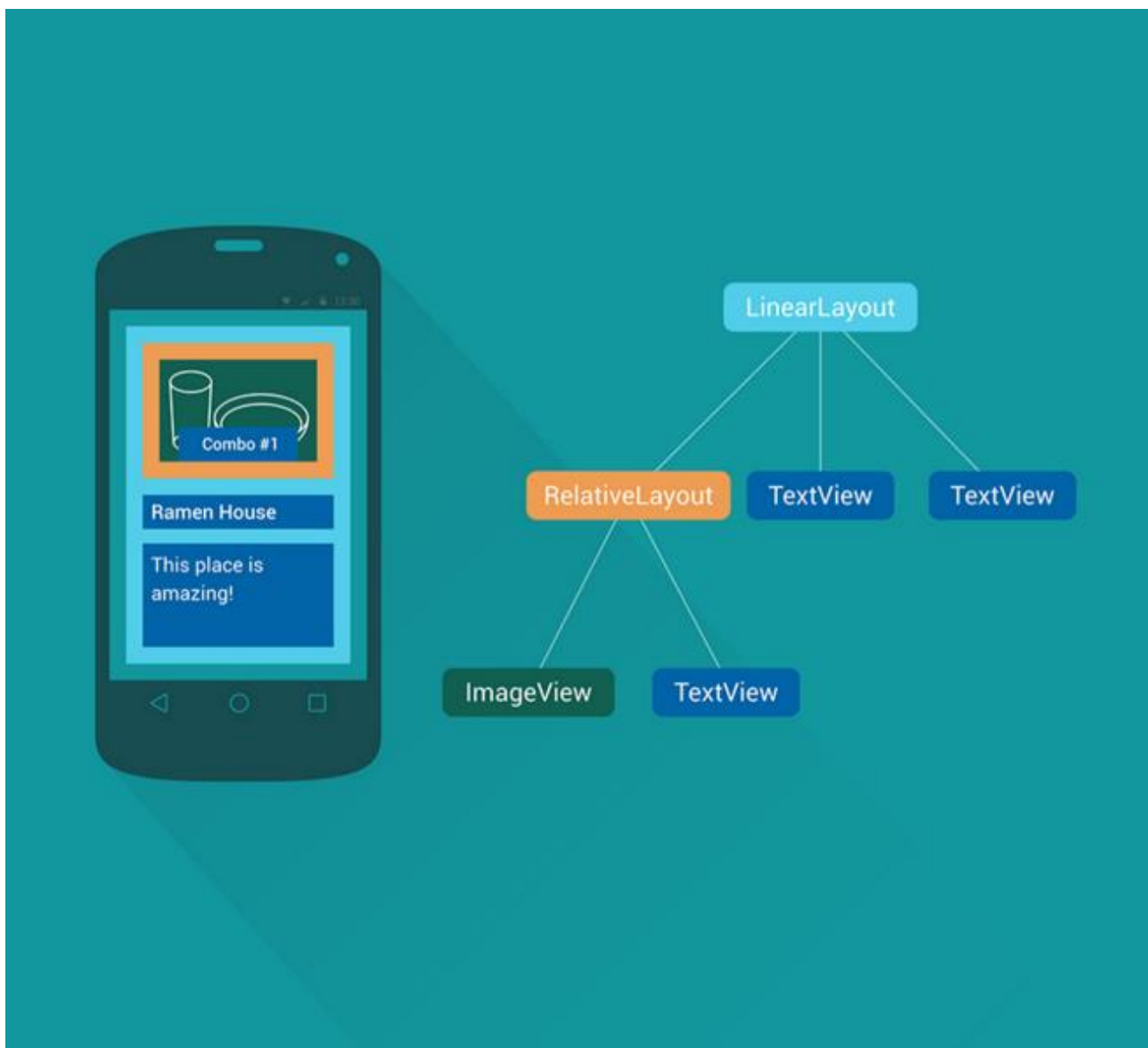
子视图可具有其自己的子视图。例如，插图中显示了包含三个子视图的垂直 **LinearLayout**。第一个是具有两个子视图的 **RelativeLayout**。

始终存在一个包含其余所有视图（如果存在）的视图，称为**根视图**。其余视图是根视图的子视图、二级子视图或三级子视图等等。因此这些视图构成了家族树，称为**视图层次结构**。

代码示例

```
<!-- Create the View hierarchy in the picture. -->
<LinearLayout
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    android:orientation="vertical">
    <RelativeLayout
        android:layout_width="match_parent"
        android:layout_height="wrap_content">
        <ImageView
            android:layout_width="wrap_content"
            android:layout_height="wrap_content"
            android:src="@drawable/ramen"/>
        <TextView
            android:layout_width="wrap_content"
            android:layout_height="wrap_content"
            android:layout_alignParentBottom="true"
            android:layout_centerHorizontal="true"
```

```
        android:text="Combo #1"/>
    </RelativeLayout>
    <TextView
        android:layout_width="match_parent"
        android:layout_height="wrap_content"
        android:text="Ramen House"/>
    <TextView
        android:layout_width="match_parent"
        android:layout_height="wrap_content"
        android:text="This place is amazing!"/>
</LinearLayout>
```



void

定义

计算机是按照一系列称为**程序**的指令运行的机器。**Android 设备**便是计算机，**应用**是程序。设备内部是称为**变量**的容器，用于存储数字或文本片段等**值**。

对象是变量，但在以下两个方面特殊。第一，对象中可包含更小的变量，即对象的**域**。例如，跟踪采购的对象可能包含数量域。第二，我们可向对象附加称为**方法**的一系列指令，实际上是小程序。例如，采购对象可能具有名为 `displayQuantity` 的方法。当指令指示设备执行方法时，我们称指令正在**调用**该方法。

对象分多种**类**。针对每个类，我们必须编写**定义**：即属于各个类对象的域和方法列表。类的定义必须包括每个方法的**签名**，其中概括了关于方法的最重要事实。签名包含方法的名称、方法**参数**（反馈到方法中的数据块）的名称和数据类型以及方法**返回值**（由方法生成的数据）的数据类型。签名还包含方法的**访问修饰符**，用于指定应用的哪些部分可以使用该方法。

部分方法可能不存在返回值。这种情况下，我们编写命令 **void** 以表明此方法不会向它的调用指令回送任何值。

代码示例

```
// This is the definition of a class named MainActivity. Although the class has many
// methods, we show the definition of only one of them.
public class MainActivity extends AppCompatActivity {

    // The "void" in the signature (first line) of the definition of this method
    // indicates that the method does not produce a return value.
    private void displayQuantity(int number) {
        TextView quantityTextView = (TextView) findViewById(R.id.quantity_text_view);
        quantityTextView.setText("" + number);
    }
}
```


indicates that the method returns no value

access modifier

method name

parameter

`private void displayQuantity(int number)`

method signature

wrap_content

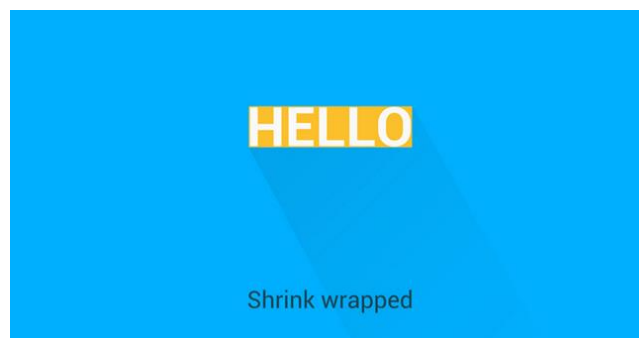
定义

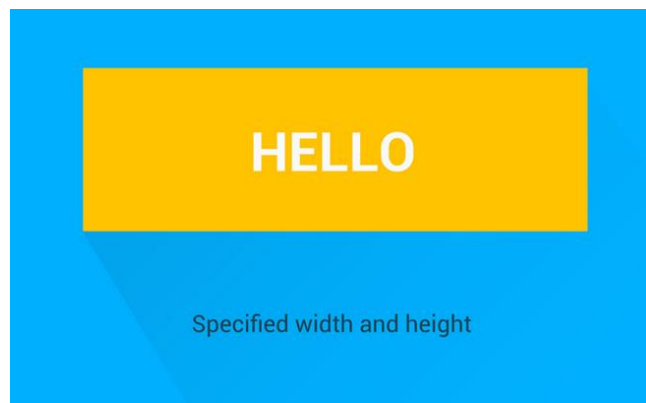
视图是屏幕上的矩形区域，通常包含一些内容。例如，**TextView** 包含文本，**ImageView** 包含图像，而称为 **ViewGroup** 的特殊类型视图内部包含较小的视图。

我们可以用给定距离指定视图的宽度或高度。我们也可以将其指定为特殊值 **wrap_content**，以围绕其内容压缩视图。为防止视图把自身包围得过紧，我们还可以指定特定的**内边距**量。

代码示例

```
<TextView
    android:layout_width="120dp"
    android:layout_height="40dp"
    android:background="#FFC300"
    android:text="HELLO"/>
<TextView
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:background="#FFC300"
    android:text="HELLO"/>
```





XML

定义

XML 代表“可扩展标记语言”。它是一种表示法，用于编写以**层次结构**或家族树形式组织的信息。示例包括主题的罗马数字轮廓、部门和分支的企业组织结构图或州、县和市的列表。

一个州可以包含许多县，且一个县可以包含许多市。但是每个市只能属于一个县，且每个县只能属于一个州。在 XML 中，我们称每个数据项目可包含许多**子项**，但每个子项只能包含在一个**父项**中。

家族树结构使 XML 成为描述 Android 应用的屏幕布局的理想语言，应用的屏幕布局由称为**视图**的矩形区域组成。该布局总是以大视图包含小视图，小视图继而包含更小视图的形式存在。

代码示例

```
<LinearLayout
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    android:orientation="vertical">
    <TextView
        android:layout_width="match_parent"
        android:layout_height="wrap_content"
        android:text="Hello"/>
    <ImageView
        android:layout_width="match_parent"
        android:layout_height="wrap_content"
        android:src="@drawable/mountain"/>
    <Button
        android:layout_width="match_parent"
        android:layout_height="wrap_content"
        android:text="Press me"
        android:onClick="doSomething"/>
</LinearLayout>
```

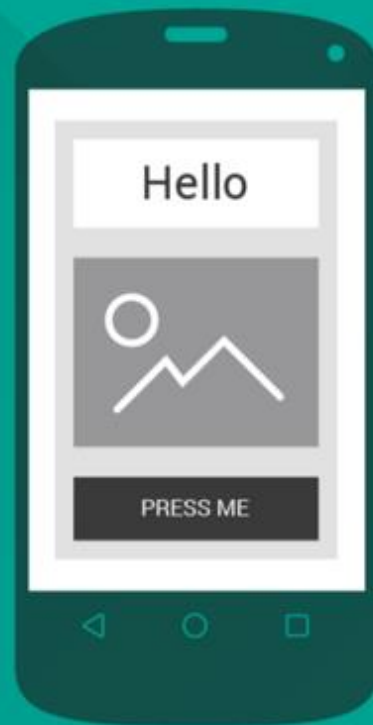
```
<LinearLayout
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    android:orientation="vertical">

    <TextView
        android:layout_width="match_parent"
        android:layout_height="wrap_content"
        android:text="Hello"/>

    <ImageView
        android:layout_width="match_parent"
        android:layout_height="wrap_content"
        android:src="@drawable/mountain"/>

    <Button
        android:layout_width="match_parent"
        android:layout_height="wrap_content"
        android:text="Press me"
        android:onClick="doSomething"/>

</LinearLayout>
```



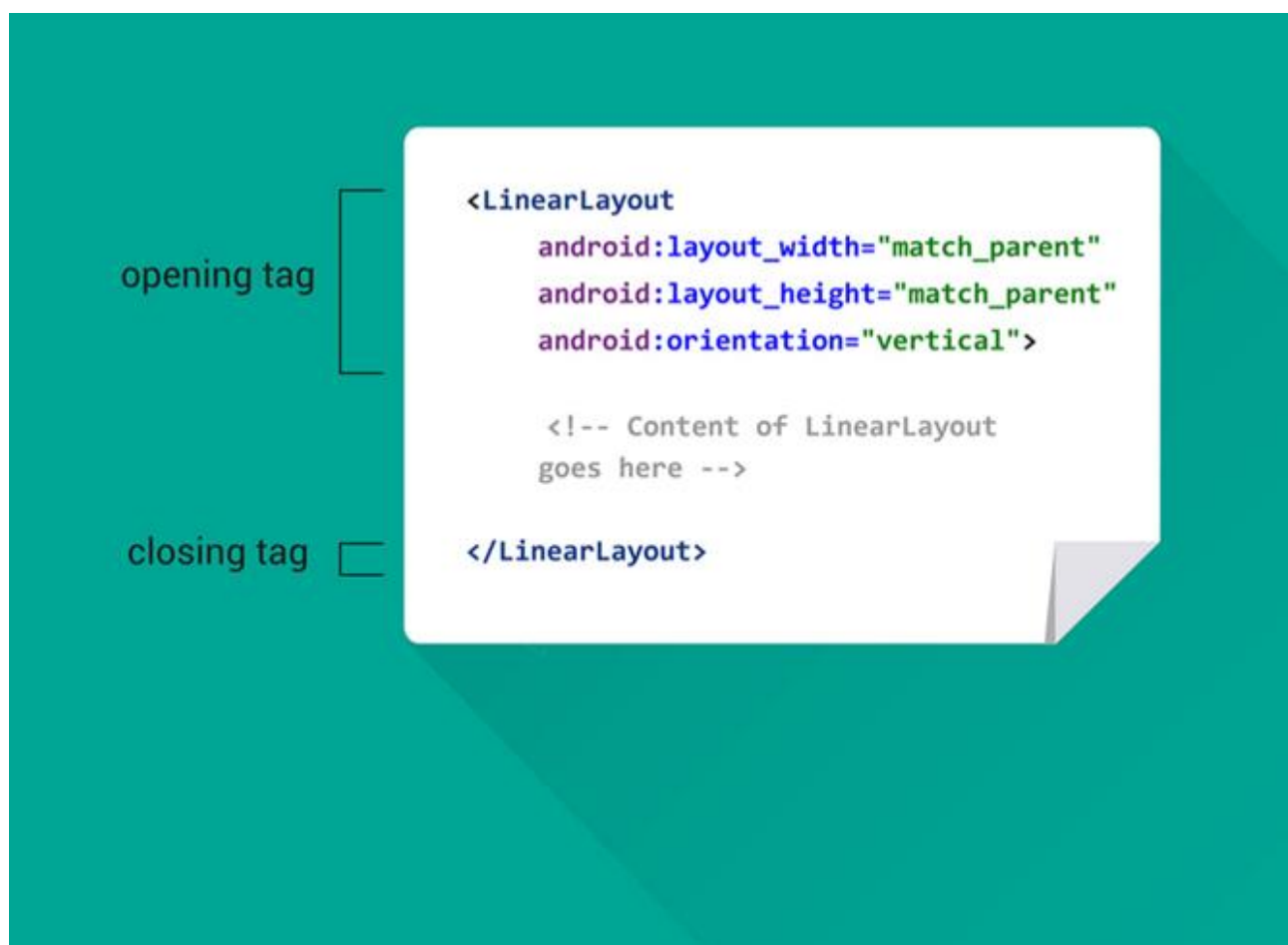
XML Tag (XML 标记)

定义

可扩展标记语言 (XML) 是一种表示法，编写的文件包含称为**元素**的信息片段。为表明元素的开始和结束，我们编写**标记**。标记易于识别，因为它始终以字符 `<` 和 `>` 开始和结束。标记还包括所标记开始和结束元素（如 `LinearLayout`）的名称。

元素通常由一对标记加上两个标记间的所有内容组成。这种情况下，一对标记中的第二个标记以字符 `</` 开始，我们称第二个标记**闭合**第一个标记。

不需要括任何内容的元素可由单一标记组成。这种情况下，标记以字符 `/>` 结束，我们称这种标记为**自闭合**标记。



本术语表是对优达学城的 Google: Android 基础 纳米学位项目（适用于想要着手构建 Android 应用的编程初学者）的补充资料，由优达学城翻译提供。

如果你对其中某些术语的翻译有更好的建议，欢迎你在[论坛](#)中提出。

优达学城

2016 年 9 月