**DEV BOOK**

**CH 2:LINGO AND FIRST APP**

**Android Application Development Lingo**

Android uses very unique terminology to describe the various components or areas of its application development. In this section, I define what these areas are, and what they most closely equate to in other common programming languages such as C++ or Java. Plus, I’ll give you a little more insight as to what open source technologies besides Java and XML drive Android under the hood, as well as how Android optimizes its Java into binaries for use on Embedded (Portable Consumer Electronics) Devices.

**The Foundation of Android: The Linux 2.6 Kernel**

The foundation of the Android OS is the **Linux 2.6 Kernel**; just like the Eclipse IDE runs on top of Java, Android runs on top of a full version of the latest Linux operating system. So yes, any and all Android devices are essentially fully functional Linux computers, which is why you see Android devices these days with dual-core, quad-core, and octa-core processors, just like “real” computers feature. The same thing goes for memory—most Android devices have at least one full gigabyte (1GB) of RAM (random access memory), and many feature 2GB, and soon you will see Android devices that have 3GB or 4GB of RAM.

The Android OS uses the core libraries of the Linux OS to do all the low-level things with files, memory, and processing that are done using any computer OS, and because Linux is so highly optimized, it can do this with less memory (more efficiently) than other OSes can. Why is the new Windows 8 OS so memory efficient all of a sudden? Because it has to compete with Linux (Android), that’s why.

It is rumored that Google has Android 4.2.2 running on top of the much more recent Linux 3.8 Kernel and that Android 5.0 will run on top of this more modern Linux Kernel later on in 2013.

**Android Optimization: The Dalvik Virtual Machine (DVM)**

An Android application developer uses the Android SDK and its Java programming language to access lower-level OS functions, as well as other higher-level functions that Android provides for using various new media assets, SQLite databases, and hardware features of the Android devices, such as cameras, gyroscopes, GPS, and the like.

When you compile your Java code and other assets (XML, audio, images, etc.) Android uses a DVM, or Dalvik Virtual Machine, technology to optimize these into a highly optimized binary format, much like an executable file, only even more optimized to run on smaller, more portable devices. If you look inside your Android APK file, you will see a .DEX file for your application in the root of your project. This is stands a Dalvik executable file.

DVM is not something an Android developer needs to be concerned with during day-to-day development, I am just mentioning it here so that if you happen to come across it on an Android-related Google search someday, you’ll have the context to know what it is and what the DVM is doing in the overall development process. Next let’s get into the lingo of Android and learn about each of the four major components of an Android application and what they are and how they work together to form a single cohesive Android app.

**Android Activities: The Presentation Layer**

Android uses the term **Activity** to describe the “front-end” screens of your application that your users view.

An Activity is a collection of design elements that, working together, compose each screen view of your application. These would include such things as user interface elements, text, background graphics, 3D, foreground content, digital video, pop-up menus, animated elements, and other visual design components that serve to provide an interface between what your application does (its Activities), and the end-user who is utilizing your application. We learn about Activities in detail in the first two parts of this book.

In Android terms, Activities generally consist of a **Layout Container**, which organizes and arranges a series of user interface elements called **Widgets**, along with your application’s content, into predefined screen areas. If you want to include graphical elements, to make your designs more compelling, then you would use**Drawables**, which is Android’s terminology for graphic elements such as images or animation.

Animation in Android uses both Widgets and Drawables; Drawables are used for frame-based animation (raster animation), and Widgets are used in procedural or tween-based animation (vector animation), as drawables can be as well. Amazingly, Android calls animation exactly what it is: animation. There is one other term in Android that is the same as it is in other programming languages: **Events**. Events allow the many elements that comprise our Activities to be processed using**Event Handling** code, just like in many other programming languages. Because readers of this book are familiar with programming, and because Event Handling is needed to make user interface designs in the first half of this book functional, I am going to cover Events early on in this book, so that we can create more robust applications sooner.

We will learn more about the Android **Activity** Class, and designing screen layouts via Android **Layouts**, which leverage the Android **ViewGroup** Class, in [Chapter 4](http://my.safaribooksonline.com/9781430257462/9781430257462_Ch04_xhtml). We’ll learn more about User Interface Design via Android Widgets, which leverage the Android **View** Class, in [Chapter 5.](http://my.safaribooksonline.com/9781430257462/9781430257462_Ch05_xhtml) Finally, we will learn more about**Events** in [Chapter 6](http://my.safaribooksonline.com/9781430257462/9781430257462_Ch06_xhtml), and **Drawables** in [Chapters 7](http://my.safaribooksonline.com/9781430257462/9781430257462_Ch07_xhtml) through [10](http://my.safaribooksonline.com/9781430257462/9781430257462_Ch10_xhtml), when we cover advanced User Interface Design and Graphics Design using Digital Images, Digital Video and Animation within Android.

**Android Services: The Processing Layer**

Whereas Activities represent the front-end or foreground of your Android application, Android **Services** represent the back-end, or background processing, the heavy lifting if you will.

The Android **Service** Class is used to create Java classes that do repetitive processing tasks, usually via programming constructs called loops or via data-fetching processes, such as streaming, in the background, while your user is simultaneously uses your application via its the front-end user interface activities. A good example of this would be a user playing an MP3 audio file while using your app.

There are many uses for Service classes in Android, obvious and common uses would be playing audio MP3 and digital video MP4 files for entertainment, streaming new media files from a remote server, converting text to speech (speech synthesis), calculating game logic while playing the game, processing Android Live Wallpapers or Android Daydreams, using Bluetooth or NFC to transfer large new media files between end-users in real-time, running a spelling checking process, any overly complex mathematical calculations, 3D rendering, and anything else that requires so much processor overhead (power) that it makes the Activity user interface or content become stilted and unprofessional.

Fortunately for Android Developers, most modern-day Android devices, such as smartphones, tablets, e-readers, and iTV sets, come with a minimum hardware configuration of a dual-core (two central processing units, or CPUs) processor, and many more are now becoming available with quad-core (four CPU) processors, and now octa-core (eight CPU) processor Android devices are appearing on the market. This means that there is plenty of processing power for your Android Service classes to leverage!

One of the more powerful options in the Android Service class is to have your background processing Service placed into its own **thread** (I like to call it spawning, but it is known in programming terminology as a **process**), which, with so many CPUs present in an Android device, could actually translate into your Service classes being allocated their own CPU cores.

Android also features a number of specialized Service classes that are already sub-classed (we will learn about this Java terminology in [Chapter 3](http://my.safaribooksonline.com/9781430257462/9781430257462_Ch03_xhtml)) from the primary Android Service class. For example, there are SpellCheckerService, WallpaperService, TextToSpeechService, DreamService, IntentService, AccessibilityService, VpnService, and AbstractInputMethodService classes also available for use by Android Developers. We will be learning all about Services in[Chapter 17](http://my.safaribooksonline.com/9781430257462/9781430257462_Ch17_xhtml) of this book.

**Android Broadcast Receivers: The Communications Layer**

Android communicates inside its application infrastructure via **Broadcast Receivers**.

These are usually used for inter-application communication, as well as to provide alerts for your users, many times from the Android OS or Android device itself. For instance, if the phone is ringing, or if the tablet is about to run out of battery power, the Android OS sends out a Broadcast Receiver that your application programming logic can respond to with some sort of custom action. Other common messages relate to the camera being used to take a picture, a time-zone change, a data download completion, a language preference change, a video cache is complete and ready for playback, and so forth.

Similar to Android Services, Broadcast Receivers run in the background, and you can configure your app code to “trap” any type of Broadcast Receiver that you feel is necessary for your application and its end-users to be concerned with. Once your code detects a Broadcast Receiver that it needs to respond to, it can invoke the appropriate user interface element in the appropriate Activity, and alert the end-user to the change in the OS or device status. We will be taking a close look at Broadcast Receivers in [Chapter 18](http://my.safaribooksonline.com/9781430257462/9781430257462_Ch18_xhtml).

**Android Content Providers: The Data Storage Layer**

Android has a unique term for a common concept: Data Storage. Whether it is stored in memory, in files on an SD Card, or in a SQLite database, stored data in Android is accessed via a **Content Provider**.

The primary (and best) way to store and share data across Android applications to use the open source **SQL** technology, which is a part of the Android OS. This is the **SQLite** database management tool package. Android has an entire package dedicated to the SQLite database management paradigm, called**android.database.sqlite,** which contains everything you will need to create and access SQLite databases.

Android Content Provider SQLite databases are used for Android device user-defined data storage by the Android OS across the board. We will be learning more about some of the more important databases to an Android device user, such as the **Contacts** databases and the **Calendar** databases, later on in this book. There are also new media related databases that keep track of a user’s images, audio files, and video assets and playlists.

We’ll learn more about Android Content Providers and learn about creating, populating, modifying, and deleting Android SQLite databases, in [Chapter 19](http://my.safaribooksonline.com/9781430257462/9781430257462_Ch19_xhtml).

**Android Intents: Inter-Application Communications**

An Android application communicates between its primary components via an **Intent**.

For example, you can use Intents to communicate between your Activities and Services, and you can do everything from launching new tasks or activities, to sending out new task instructions to existing ones.

The Intent object holds information regarding which application component needs to perform the required task. This includes what actions need to be taken to complete that task, a definition of the data that the task uses to perform that action, and, optionally, that data’s MIME type and any flags (settings) and other optional data or information that may be needed to completely convey what exactly needs to be accomplished via the Intent object that is being sent from one Android app module to another. Yes, Intents are one of the more complicated areas of the Android OS, as far as their implementation goes.

Each type of of Android component, Activities, Services, or Broadcast Receivers, has its own type of Intent object. This assures that Intents do not get mixed up and keeps everything well organized. So to start up a Service via an Intent, you pass over a Context.startService( ); Java method call with your Intent. For an Activity, use a Context.startActivity( ); Java method call, and for a Broadcast Receiver use a Context.sendBroadcast( ); Java method call. We will learn all about this in [Chapter 3](http://my.safaribooksonline.com/9781430257462/9781430257462_Ch03_xhtml) (Java) and [Chapter 16](http://my.safaribooksonline.com/9781430257462/9781430257462_Ch16_xhtml) (Intents).

**Android Manifest: Application Permissions Definition**

Finally, Android utilizes the term **Manifest** to describe the XML file that “bootstraps,” or defines and launches, any given Android application. Android uses XML to define many things, and like the index.html file used to launch websites, Android apps launch via their project APK’s AndroidManifest.xml file.

In the root level of any Android application .**APK** (Android PacKage) file, you will find a file named: **AndroidManifest.xml** that contains XML mark-up tags that define everything that the Android OS needs to know about your application. This includes, but is not limited to, what Android OS versions the app supports, what Activities, Broadcast Receivers, and Services the app contains, what permissions the app needs to access secure databases and Internet resources, and Intent Filters that need to be established.

We’ll discuss Android Manifest in the chapters where we need to define special Activities, Services, Broadcast Receivers, or assign permissions to application components. This will turn out to be quite a few, by the time we hit the end of this book.

**XML:**

The reason that the Layout and Menu XML files have the same name is because normally you will have both UI elements and menus for each of your application (activity) screens, so the most logical way to group them is by their file names. 🡨The files are not the same in my version (main.xml vs activity\_main.xml)

The third **android:showAsAction** parameter determines whether your menu shows on the Action Icon Bar on Android OS version 3.x and 4.x (and soon 5.x) devices. In this case, we are using the **none** setting, because we want backward-compatibility to Android 1.6 and 2.x devices such as the Amazon Kindle Fire (2.3.7). A **none** setting pops-up the menu at the bottom of the screen as you will see when you run the Hello\_World app in the 4.2 Emulator later. If you are developing apps solely for Android 3.x and later, you can use the **ifRoom** parameter as a setting instead, and your menu items will appear in the Android Action Bar, if there is room for them, that is.

JAVA:

The first method call uses the **super** keyword to pass the**saveInstanceState** Bundle object up to the **android.app.Activity** superclass and its onCreate( ) method.

**R.layout.activity\_main** actually translates into (means to Android):

**C:\Users\Username\workspace\Hello\_World\res\layout\activity\_main.xml**

**CH 3: JAVA FOR ANDROID**

Java provides a method of organizing the code in your programming projects into logical modules, or collections of code, called**Packages**. In the Java programming language, a **Package** is a collection of Java **Classes**, which we will learn about in greater detail in the next section of this chapter.

**String** objects allow an **array** (collection) of text to be created via the **char** (character) **primitive data type**. Thus, String data type objects are really a collection (via an array) of characters (via the **char** data type), whereas all the other common data types in Java are called **primitives** or primitive data types. Lowercase string in your code will not represent a String data type to Java, and will be considered an object or variable name, so this is something to be aware of if you are using lots of text.

To summarize, just remember that the top-level **java.lang.Object** or Java Language Object class is the mother of all Java classes, as it provides the foundation for all Objects in Java, however big or small they might be.

The keyword placed immediately before the method name is called a **modifier** in Java. Modifiers in Java can be used with classes, methods, or variables. There can be more than one modifier, if needed, to define a class, method, or variable’s characteristics. There are two types of modifiers, **access control** modifiers, and modifiers which do not define any access control.

There are four levels of access control modification: **no modifier** (see all the preceding methods, except for the constructor method) where the method or variable is visible to the entire package that it is contained in; the **public** modifier, which means that any Java class, even classes outside of the package, can use that method or variable; the **private** modifier, which means that only the class that a method or variable is defined in can use that method or variable; and the **protected** modifier, which means that the method or variable is visible to the package, as well as to all subclasses which may be created from that class.

There are several other types of modifiers which do not affect access control, such as: **static**, **final**, **abstract,** and**synchronized** modifiers.

Another difference between a constructor and a method is that constructors cannot have non-access-control modifiers, so be sure not to declare your constructor as: **static**, **final**, **abstract**, or **synchronized**.

We will need to have all our methods coded before we develop public interfaces for any given class

The general format for coding a Java interface is as follows:

**<Access Control Modifier List> interface <Interface Name> { ... }**

All methods declared within an interface are **abstract** by definition, that is, they cannot be used directly, but must be subclassed to be used, which is what the**abstract** modifier signifies.

If we made WorldGen abstract like this:

public abstract class WorldGen { ... }

Then to create WorldGen objects we would have to subclass a concrete class from the now abstract WorldGen class, now designed as a WorldGen template:

public class WorldGenPlanets **extends** WorldGen { ... }

Because all **methods** declared within an interface are inherently abstract, they do not need to be declared as such using the Java abstract modifier. Similarly, because an interface is intended (again, inherently) to **expose** these methods to the public, the **public** access control modifier is also implicitly assumed, and thus also does not have to be explicitly declared.

Final variables differ from constants in that a constant value is known at compile time, whereas a final variable’s value might not be known at compile time.

A class declared as final cannot be subclassed; examples of Java classes declared as final, so that they cannot be changed, are **java.lang.String** and**java.lang.System**.

**CH 4: LAYOUTS & ACTIVITIES**

the **foundation**, for any screen design in Android. This is called the Android **Layout Container**, and it is implemented in Android using the **ViewGroup** class. The Android **ViewGroup** class is a subclass of the Android **View** class

the ViewGroup class is not used directly to implement any specific layout container in Android; it can thus be defined or classified as a **base** class to its many subclasses.

all the different types of screen layout container classes currently available to Android developers to create user interface designs under (with) are actually subclasses of the Android ViewGroup base class.

**RelativeLayout** (default layout type, created in the New Android Application Project dialogs), **FrameLayout** (used for single item layouts), and **LinearLayout** (for user interface elements that need to conform to a row or column format, such as in-line button strips and top-down lists)

API Level 13 (Honeycomb) and later layout containers include **Grid Layout**, **ViewPager,** and **PagerTitleStrip**, and we will not cover these.

Grid Layout sounds important indeed.

we also don’t discuss the **FragmentBreadCrumbs** layout container, as its complexity is outside the scope of an introductory book on learning Android application development.

Menus in Android usually have five or fewer entries

Every Activity you create needs to implement an onCreate( ) method so that it can create the Activity in system memory.

Eclipse ADT can, and will, make some mistakes, which you may need to fix later on!

**CH 5: INTENTS & EVENTS**

Both Intents and Events allow communication inside your application, which is why they allow interactivity to be implemented.

**Intents** allow communication at a more global level within the Android app.

**Events** allow communication at a much more localized level within an Android application.

An Android Intent is actually an object.

The primary areas of Android applications that can process Intent objects include Activities, which we have created already; Services, which are used for background processing, and which we cover later on; and Broadcast Receivers, which are used for sending messages to the Android end-user.

So what types of information does one of these Intent objects pass between application components? Functional areas of an Intent object include:

* The **component name** of the component (class) which needs to process the intent
* The **action** that needs to be performed
* The **data** that the action needs to operate on
* The **type** of data (a MIME type) that is being processed
* The **category** that this processing falls under
* Any **flags** and **extras** that are needed to further define this processing that needs to be performed

An Intent that contains a component name is termed an **explicit** Intent, whereas an Intent that does *not* specify a component name is termed an **implicit** Intent.

If you were going to launch our NewPlanet Activity class from another package, you would use**chapter.two.hello\_world.NewPlanet.class** as the complete path name to the class. If you were inside the chapter.two.hello\_world package (such as we will be, in the Intent examples later in this chapter) you would use**NewPlanet.class** as a **run-time reference** to our NewPlanet.java class code.

Some examples of commonly used actions used with **Activity** classes for the Android OS include:

* ACTION\_DIAL (Display a phone number to dial)
* ACTION\_CALL (Make a phone call)
* ACTION\_MAIN (Start the main activity for the app)
* ACTION\_EDIT (Edit a database)

Some commonly used actions with **Broadcast Receivers** include:

* ACTION\_TIMEZONE\_CHANGED (User has moved into a new time zone)
* ACTION\_POWER\_CONNECTED (User has plugged in the device)
* ACTION\_SHUTDOWN (User has shut down the Android device)

Data is usually specified using something called a **URI** or **Universal Resource Identifier**.

Some more popular **category constants** include:

* **CATEGORY\_DEFAULT**
* **CATEGORY\_BROWSABLE**
* **CATEGORY\_TAB**
* **CATEGORY\_LAUNCHER**
* **CATEGORY\_INFO**
* **CATEGORY\_HOME**
* **CATEGORY\_PREFERENCE**
* **CATEGORY\_CAR\_DOCK**
* **CATEGORY\_DESK\_DOCK**
* **CATEGORY\_CAR\_MODE**
* **CATEGORY\_APP\_MARKET**

Because we are not going to cover Intents at an advanced level in this book, we will not be using flags and extras in our Intent objects, but I wanted to let you know that they are included in the Intent object hierarchy and are available if your advanced Intent object design requires them.

**Implicit Intents** are Intent objects that do not specify any **component specifier** within the Intent object itself.

You can find much more detailed information regarding Intent Filters at [**developer.android.com/reference/android/content/IntentFilter.html**](http://developer.android.com/reference/android/content/IntentFilter.html)**.**

Intent filter structures are declared inside the AndroidManifest.xml file, by using the **<intent-filter>** tag. They filter implicit intents based on three of the seven attributes of an Intent object; the Intent **action**, its **data**, and possibly its **category**, if included and applicable.

Intent filter tags can be contained (nested) inside the **<activity>** and **<activity-alias>** XML tags, as well as inside the**<service>** and the **<receiver>** tags.

Intent filter tag structures are used to define a description of Intent object configurations that need to be matched. They also allow a **priority** attribute that can be implemented, if more than one match is encountered.

Intent filters are tested for **action** matches first, then for **data** matches**,** and finally for **category** matches.

For Intent filters that include data characteristics, a data parameter gets broken down into four categories, including the **data type** (MIME Type), the **data scheme** (such as http:// ), the **data authority** (server host and server port, specified as host:port)**,** and finally the **data path**.

Android event handling methods:

onClick(); 🡨most common

onKeyDown();

onLongClick();

onCreateContextMenu();

onFocusChange();

onTouch();

**CH 6: ANDROID UI**

Some of the most common user interface widget objects and layout container objects, including ViewGroup, ImageView, and TextView, are **direct sublasses** of the View class. Others that we have already utilized, such as Button, VideoView, and ImageButton, are **indirect subclasses** of the View class.

Android Toast objects are used to send messages onto your Activity screen that appear for a predefined period of time sending a message to the user.

Tag parameters need only be separated by a single space character; parameters do not need to be on their own line.

UI element widgets are subclassed from the venerable Android **View** class

**CH 7: ANDROID GRAPHICS**

the Android**Drawable** class is also the highest level class for **graphics design** related objects, which are called **drawables**

The **direct subclasses** of the Drawable class essentially equate to which types of graphics design elements we can use in our Android applications.

By far the most often used direct subclass of the Drawable class is called the **BitmapDrawable** subclass, which is used for**digital images**, such as the PNG files we have used so far in our app as well as for WEBP, JPEG or GIF.

The **ColorDrawable** is the most basic level Drawable class, and it’s used to define the **color** of a screen, like we did when we defined the **black** screen color for our Attack Planet Activity user interface screen in [Chapter 6](http://my.safaribooksonline.com/9781430257462/9781430257462_Ch06_xhtml).

The **GradientDrawable** is used to create a gradient of colors from one color into another, and can be defined by any shape. Shapes supported in Android GradientDrawables include: line, rectangle, ellipse (oval), circular, ring (hoop or torus), and a gradient can be drawn **linear** (straight, any angle), **radial** (emanate from a point), or **sweep** (linear but rotate around a point).

The **ShapeDrawable** is used to create a **vector** shape in Android. A vector or shape is a 2D line or curve that defines the outside of a 2D volume, like a heart or a star. Those familiar with Adobe Illustrator will be familiar with vector shapes, as well as gradients and color graphics tools.

The **LayerDrawable** lets us handle multiple layers of imagery, much like one sees in Photoshop or GIMP digital imaging software, and their image layers features. This is a more advanced drawable, for use in advanced gaming and real-time compositing applications. We’ll learn more about image **layers** and **compositing** later on in this chapter, as well as how they are used.

The **InsetDrawable** allows us to use an area or subset of the screen called an **inset** to display graphics (drawable) elements. This is used for things like widgets, which use only a part of the display screen as the required screen area for their application or purpose. A clock for the device home screen might be a good example of something that may use an InsetDrawable.

the Android **TransitionDrawable**, which **cross-fades** two images to create an image **transition**, is a subclass of**LayerDrawable**. Because these LayerDrawable objects handle multiple images in layers, they are a logical class to subclass to create the TransitionDrawable class, which takes two images and animates their **alpha channel** values

The **AnimationDrawable** is another key indirect subclass of Drawable that implements **Frame Animation** in Android.

The **LevelListDrawable** is used for progress bars and similar applications where a graphic element needs to be replaced on the screen based on a level of activity of some kind. A similar **StateListDrawable** class is used to replace graphic elements based on a different state change that may be encountered during an application runtime. A StateListDrawable can access graphics elements in any particular order, that is, out of order, whereas a LevelListDrawable accesses graphics in order, from one level to another.

Both the LevelListDrawable and StateListDrawable are subclassed from the Android **DrawableContainer** direct subclass

**aspect ratio** involves the **ratio** of how many pixels wide to how many pixels in height a screen resolution is

5:3 is the correct aspect ratio for an 800x480 screen

Other common aspect ratios include 16:9 (HDTV), since 1920 divided by 16 times 9 equals 1080, and 4:3 was the original computer monitor aspect ratio for all resolutions except for 1280x1024, which was a 5:4 aspect ratio. Another popular Android Smartphone resolution (and mini-tablet resolution) is 854x480, which is also 16:9.

There are two types of color display, **subtractive color** used in print, where ink colors subtract from each other, and**additive color**, which is used in lit display (display uses light) products, where color values are added to each other.

Subtractive color follows a **CMYK** color model

Additive color follows an **RGB** color model

Whereas a 24-bit image has three (RGB) color **channels**, a **32-bit image** has four (**ARGB**) color channels. The fourth channel in a 32-bit image is called the **alpha channel**.

We have already turned UI element parameters completely transparent, using an **ARGB** setting in XML of **#00000000**. The first two zeroes specify **opacity** to be 100% off, and thus transparency is 100% on.

Besides using image layers and their alpha channels, compositing involves using a **blending algorithm** on each layer that sums (or differences) pixel data values, based on complex algorithms that can create image compositing special effects such as**Overlay**, **Screen**, **Darken**, **Lighten**, **XOR**, and so on.

The format Android prefers over all others is **Portable Network Graphics** or **PNG** (pronounced **Ping**) file format. PNG comes in two flavors, **PNG8** or indexed color PNG and **PNG24** or truecolor PNG. Because PNG24 can also “carry” an alpha channel, technically a 24-bit PNG with an alpha would be a **PNG32**.

Android likes PNG format images because PNG use **lossless** image compression that yields the highest quality result because PNG images do not lose any of the original image quality (or data) during the compression process.

The next most desirable image format for use in Android is **JPEG**, which stands for **Joint Photographic Experts Group**. This image format uses **lossy** image compression, which “throws away” some of the original image data to achieve a better compression result, but at the expense of image quality. If you zoom into a JPEG image you will see areas that look dirty or discolored; these are compression **artifacts.** Another important aspect of JPEG image format is that it cannot carry an alpha channel, and thus it cannot be used in image compositing, unless it is the bottom layer in the image compositing layer stack, or unless the alpha information is attached to it later on, inside of an application.

The least desirable image format to use in Android is the Compuserve **GIF** format, which stands for **Graphics Information Format**. A GIF image only supports 8-bit indexed color, and has a larger data footprint (a weaker compression algorithm) than a PNG8 file will have.

WEBP stands for **Web Photo** and it is similar to PNG32 but has about a 25% better (smaller) data footprint; that is, its compression algorithm is superior to PNG24 and PNG32. (Only compatible with newer Android.)

Without compression, the raw data size for an image would be calculated as follows: Pixel Width times Pixel Height times 3 (for RGB) or 4 (for ARGB).

**Resolution** is the biggest factor to adjust to get better compression (file size) results of your image. Color depth is the next most important factor in image compression. This is because you have to factor the number of pixels by multiplying that against the number of color channels, and if you can reduce this multiplier, you also greatly reduce the resulting file size. So instead of using a 32-bit PNG we used an 8-bit PNG; by doing this we reduced the amount of pixels we were compressing from 36,864 to 9,216.

As you can see in top of the screenshot in [Figure 7-2](http://my.safaribooksonline.com/9781430257462/Sec6_9781430257462_Ch07_xhtml#Fig2_9781430257462_Ch07) the attacklaser.png is an 8-bit indexed color PNG image, yet it looks like it’s an even higher quality truecolor image. This is because I used **dithering**, a key option available for 8-bit image compression.

Dithering involves simulating more than the 256 allowed colors in an 8-bit Indexed color palette. This is done by using subtle dot patterns that mix two colors together, to form a third color halfway between the two.

In digital imaging, we always start with the highest resolution and work down to avoid any **pixellation**.

**CH 8: ANDROID COMPOSITING**

Name the **File:  attack\_invade**. Since Android does not use file extensions in XML and Java code, be sure not to name the file attackinvade as that is our PNG8 filename.

**CH 9-10: ANDROID ANIMATION**

Android features two different types of animation, the first type, which we’ll be focusing on in this chapter, is **frame-based** or **raster** animation. The second type of animation is **procedural** or **vector** animation.

Typical frame rates for new media assets that you’re familiar with in your everyday life are digital video, which runs at **30 FPS**; motion picture film, which runs at **24 FPS**; or video games, which run at **60 FPS** the **illusion of motion** can be achieved in as little as **12 FPS**

There are two types of looping: **seamless looping**, where the frames play in a circle, like this **0**,**1**,**2**,**3**,**4**,**0**,**1**,**2**,**3**,**4** and **pong looping**, where images go back and forth, like a good game of pong, like this **0**,**1**,**2**,**3**,**4**,**3**,**2**,**1**,**0**.

Android keeps frame animation assets and definitions in **drawable** asset folders, whereas Android keeps procedural animation assets and definitions in **animation** asset folders.

procedural animation uses less imagery and more processing power, and so yields a smaller app size (better compression), whereas frame animation uses more imagery and less processing power, as the CPU is simply taking data from memory and putting it on the screen.

Vector and raster are two completely different approaches; raster is **data heavy**, whereas vector is **data compact**. The reason for this is because text (math) compresses well, and arrays of pixels (and frames) simply do not. Vector uses more CPU resources (while it does the calculations), whereas raster uses more storage or bandwidth resources (transferring data). This is because a mathematical vector needs to be **rendered** to the display screen, which means the math is turned into graphical elements, usually motion graphics, or 2D animation.

**Note**   Animation concepts apply to both 2D and 3D imaging and animation. Both use vectors; 2D vectors in 2D, and 3D vectors in 3D, and both 2D and 3D involve the core concepts of **translation** (movement), **rotation**, and **scaling**. In 2D these concepts involve the**X** and **Y** axes, and in 3D, these involve the **X**, **Y**, and **Z** axes.

**CH 19: DBMS AND SQL**

In a real world database design, the theory of which is largely beyond the scope of an introductory book, you will want to have more than one single database structure, for both access (search) performance, as well as for organizational reasons. In fact the Android OS uses more than one database structure for its end-user information storage and access, as we will soon see later on in this chapter.

The way to have multiple databases is to have a unique **key** (unique index) for each record in each of the databases (tables). In that way information for a single data record can span more than one database table, using that key. In Android, this key is called an **ID** and is always designated via the constant **“\_ID”** in Android’s SQLite databases.

If you own your own server hardware, you can download and install MySQL, using the [MySQL.com](http://mysql.com/) website

SQLite is also in all the HTML5 browsers

has only a **quarter megabyte** memory footprint

SQLite supports three different data types: **TEXT** (known as a **String** value in Java), **INTEGER** (known as a **long** value in Java), and a **REAL** (known as a **double** value in Java) data type.

It is important to note that SQLite does not itself validate any data type that may be written to its fields

The general purpose of an Android Content Provider is to encapsulate data in a standardized fashion, while at the same time, providing the Android Developer with some sort of mechanism for enforcing their data security.

If you want to access data that is inside a Content Provider, you would use a **ContentResolver** object within your current application **Context** to communicate with that Content Provider as a database client.

A complete URI for an Android Content Provider follows this format:

content://Authority/Path/ID

An authority naming convention usually follows Java package naming conventions.

By using different path names, one single Content Provider can accommodate many different types of data that are in some way related, such as the New Media content types, for example, kept in the MediaStore Content Provider.

For totally unrelated data types, it is standard programming practice that you would want to utilize a different Content Provider subclass, as well as different data authority (and path, for that matter) for each database.

The **MediaStore** databases include 9 different new media asset databases, the **CalendarContract** databases include 11 different Calendar component databases, and the **ContactsContract** databases include the most databases, with 21 functional databases.

A READ operation on a database table is referred to in the industry as being a **non-destructive** database access operation.

Launch the Android emulator, by using the **Window****Android Virtual Device Manager** menu sequence. This work process launches the Nexus S emulator, without having the emulator load and run your Hello World application automatically.

It is also important to note here that if for some reason a **Run As Android Application** work process for any reason does not launch Android’s emulator functionality, that this is the way to “**hard launch**” the Android emulator.

GAMES BOOK

**Chapter 3: Game Design Basics**

Design documents serve a few different purposes.

First and foremost, they contain a functional specification of the game.

The second purpose of the design document is the technical specification.

Every game needs to tell a story.

Example design overview:

**Section 1 - Game Overview**

**Section 2 - Gameplay and Mechanics**

**Section 2.1 – Gameplay**

**Section 2.2 – Mechanics**

**Section 2.2.1 – Movement**

**Section 2.2.2 – Obstacles**

**Section 2.2.3 – Pickups**

**Pacing**

**Aesthetics**

**Scale**

**Technical Requirements**

**AGILE:**

*We are uncovering better ways of developing software by doing it and helping others do it. Through this work we have come to value:*

**Individuals and interactions** over Processes and tools

**Working software** over Comprehensive documentation

**Customer collaboration** over Contract negotiation

**Responding to change** over Following a plan

*That is, while there is value in the items on the right, we value the items on the left more.*[[1]](http://en.wikipedia.org/wiki/Agile_software_development#cite_note-Agile_Manifesto-1)

|  |  |  |
| --- | --- | --- |
| [Kent Beck](http://en.wikipedia.org/wiki/Kent_Beck) | James Grenning | [Robert C. Martin](http://en.wikipedia.org/wiki/Robert_Cecil_Martin) |
| Mike Beedle | [Jim Highsmith](http://en.wikipedia.org/wiki/Jim_Highsmith) | [Steve Mellor](http://en.wikipedia.org/wiki/Stephen_J._Mellor) |
| Arie van Bennekom | [Andrew Hunt](http://en.wikipedia.org/wiki/Andy_Hunt_(author)) | [Ken Schwaber](http://en.wikipedia.org/wiki/Ken_Schwaber) |
| [Alistair Cockburn](http://en.wikipedia.org/wiki/Alistair_Cockburn) | [Ron Jeffries](http://en.wikipedia.org/wiki/Ron_Jeffries) | [Jeff Sutherland](http://en.wikipedia.org/wiki/Jeff_Sutherland) |
| [Ward Cunningham](http://en.wikipedia.org/wiki/Ward_Cunningham) | Jon Kern | [Dave Thomas](http://en.wikipedia.org/wiki/Dave_Thomas_(programmer)) |
| [Martin Fowler](http://en.wikipedia.org/wiki/Martin_Fowler) | [Brian Marick](http://en.wikipedia.org/wiki/Brian_Marick) |  |

© 2001, the above authors. This declaration may be freely copied in any form, but only in its entirety through this notice.

The Agile Manifesto is based on twelve principles:[[8]](http://en.wikipedia.org/wiki/Agile_software_development#cite_note-manifestoprinciples-8)

1. Customer satisfaction by rapid delivery of useful software
2. Welcome changing requirements, even late in development
3. Working software is delivered frequently (weeks rather than months)
4. Working software is the principal measure of progress
5. Sustainable development, able to maintain a constant pace
6. Close, daily cooperation between business people and developers
7. Face-to-face conversation is the best form of communication (co-location)
8. Projects are built around motivated individuals, who should be trusted
9. Continuous attention to technical excellence and good design
10. Simplicity—the art of maximizing the amount of work not done—is essential
11. Self-organizing teams
12. Regular adaptation to changing circumstances

There are many specific agile development methods. Most promote development, teamwork, collaboration, and process adaptability throughout the life-cycle of the project.

**Iterative, incremental and evolutionary**

Agile methods break tasks into small increments with minimal planning and do not directly involve long-term planning. Iterations are short time frames ([timeboxes](http://en.wikipedia.org/wiki/Timeboxing)) that typically last from one to four weeks. Each iteration involves a [cross-functional team](http://en.wikipedia.org/wiki/Cross-functional_team) working in all functions: planning, [requirements analysis](http://en.wikipedia.org/wiki/Requirements_analysis), [design](http://en.wikipedia.org/wiki/Software_design), [coding](http://en.wikipedia.org/wiki/Computer_programming), [unit testing](http://en.wikipedia.org/wiki/Unit_testing), and [acceptance testing](http://en.wikipedia.org/wiki/Acceptance_testing). At the end of the iteration a working product is demonstrated to stakeholders. This minimizes overall risk and allows the project to adapt to changes quickly. An iteration might not add enough functionality to warrant a market release, but the goal is to have an available release (with minimal [bugs](http://en.wikipedia.org/wiki/Software_bug)) at the end of each iteration.[[10]](http://en.wikipedia.org/wiki/Agile_software_development#cite_note-embracing_change-10) Multiple iterations might be required to release a product or new features.

**Efficient and face-to-face communication**

No matter what development disciplines are required, each agile team will contain a [customer representative](http://en.wikipedia.org/wiki/Customer_representative), e.g. *Product Owner* in Scrum. This person is appointed by stakeholders to act on their behalf[[11]](http://en.wikipedia.org/wiki/Agile_software_development#cite_note-11) and makes a personal commitment to being available for developers to answer mid-iteration questions. At the end of each iteration, stakeholders and the customer representative review progress and re-evaluate priorities with a view to optimizing the [return on investment](http://en.wikipedia.org/wiki/Rate_of_return) (ROI) and ensuring alignment with customer needs and company goals.

In agile software development, an *information radiator* is a (normally large) physical display located prominently in an office, where passers-by can see it. It presents an up-to-date summary of the status of a software project or other product.[[12]](http://en.wikipedia.org/wiki/Agile_software_development#cite_note-Cockburn.2C_Information_radiator-12)[[13]](http://en.wikipedia.org/wiki/Agile_software_development#cite_note-Ambler-13) The name was coined by [Alistair Cockburn](http://en.wikipedia.org/wiki/Alistair_Cockburn), and described in his 2002 book *Agile Software Development*.[[13]](http://en.wikipedia.org/wiki/Agile_software_development#cite_note-Ambler-13) A [build light indicator](http://en.wikipedia.org/wiki/Build_light_indicator) may be used to inform a team about the current status of their project.

**Very short feedback loop and adaptation cycle**

A common characteristic of agile development are daily status meetings or "stand-ups", e.g. *Daily Scrum (Meeting)*. In a brief session, team members report to each other what they did the previous day, what they intend to do today, and what their roadblocks are.

**Quality focus**

Specific tools and techniques, such as [continuous integration](http://en.wikipedia.org/wiki/Continuous_integration), automated [unit testing](http://en.wikipedia.org/wiki/XUnit), [pair programming](http://en.wikipedia.org/wiki/Pair_programming), [test-driven development](http://en.wikipedia.org/wiki/Test-driven_development), [design patterns](http://en.wikipedia.org/wiki/Software_design_pattern), [domain-driven design](http://en.wikipedia.org/wiki/Domain-driven_design), [code refactoring](http://en.wikipedia.org/wiki/Code_refactoring) and other techniques are often used to improve quality and enhance project agility.