Version 1.0

2018-JuL-01



Interview Feedback

Homework

Presenter: Arturo Sanchez

Mobile Apps Company

3301 Windy Ridge Parkway, Suite 400, Atlanta, GA 30339

# **Interview Feedback**

## Introduction

### Elevator

My name is Arturo Sanchez Chavarria, I am 28 years old and I have been working as an Analyst Developer since 2012.

I like to walk in the park, listen to music, play video games, and watch some movies.

I consider myself as a friendly, responsible, cheerful, optimistic and hardworking person.

I really like to be a developer and working as a team.

### Previous Experience

I have mostly worked with .NET technologies using SQL Server as the main Database server, maintaining, updating and creating new applications for the companies that I have been assign.

I have worked with Visual Studio 2010 up to 2017 using C# as primary language (also used VB .NET about 1 year).

In Visual Studio, I have created Web Applications (WebForms, MVC, and Classic ASP), Windows Applications (WinForms, WPF) and Web Services (WCF, ASMX).

In the information storage, I have mostly used SQL Server (2000 up to 2014), but also worked with Progress (10.1 b), XML, SQL LITE.

I have been a code auditor as one of my responsibilities in previous jobs so I need to work with the best practices in code implementation (OOP), that also includes validating design in databases, web services, and interfaces.

I have been involved in almost all of the steps in the creation of an application:

* Creating work plan
* Requirements survey
* Clients interviews
* Creating prototypes
* Designing the application
  + database solution
  + interfaces design (if applies)
  + Web Services Design (if applies)
* Developing the application
* Unit testing
* Publishing the application
  + Initial loads (Database)
* Post-implementation support.

I have also been involved in optimizing the performance of certain existing process as part of the developing phase.

## Questions Asked

### Android Platform architecture



|  |  |
| --- | --- |
| Layer | Description |
| Linux Kernel | It is the foundation of the Android platform, because all of the functionality it is done by this layer (passed through the Android Runtime (ART).  Allows device manufacturers to develop hardware drivers for a well-known kernel. |
| Hardware Abstraction Layer (HAL) | Provides standard interfaces that expose device hardware capabilities to the higher-level Java API framework.  Consists of multiple modules, each of which implements an interface for a specific type of hardware component (Example camera or Bluetooth) |
| Android Runtime | It is an application runtime environment used by the Android Operating System.  It’s optimizes the execution of applications by continually profiling application each time they run and dynamically compiling frequently executed short segments of their bytecode into native machine code.  Some of the major features of ART include the following:   * Ahead-of-time (AOT) and just-in-time (JIT) compilation * Optimized garbage collection (GC) * Better debugging support, including a dedicated sampling profiler, detailed diagnostic exceptions and crash reporting, and the ability to set watchpoints to monitor specific fields |
| Native C/C++ Libraries | It contains the libraries exposed for the Java Framework API to use.  You can use the Android NDK to access some of these native platform libraries directly from your native code. |
| Java API Framework | All of the Android OS features are available through APIs written in the Java language.  The Apps use this API Framework based on Managers to get the desired functionality.  Example.- If you want to use the Location of the device you use the Location Manager. |
| System Apps | Android comes with a set of core apps for several basic functionality.  A third-party app can become the user’s default app because there is no special status among the system apps. |

### Supporting multiple languages

Android supports multiple languages based on the primary language configured in the device and that is by the proper use of the resource values.

When setting a text in the application be sure to use the reference to a resource value, that way you can create a new resource file configuring this resource value file as a location, then selecting the location and region you want it. When working with this way, Android can automatically change these texts with the users default language.

### Units of measure that we can use in android? SP vs DP?

Android runs on a variety of devices that have different screen sizes and pixel densities. The system performs basic scaling and resizing to adapt your user interface to different screens.

A dimension value defined in XML. A dimension is specified with a number followed by a unit of measure. For example: 10px, 2in, 5sp. The following units of measure are supported by Android:

dp

Density-independent Pixels - An abstract unit that is based on the physical density of the screen. These units are relative to a 160 dpi (dots per inch) screen, on which 1dp is roughly equal to 1px. When running on a higher density screen, the number of pixels used to draw 1dp is scaled up by a factor appropriate for the screen's dpi. Likewise, when on a lower density screen, the number of pixels used for 1dp is scaled down. The ratio of dp-to-pixel will change with the screen density, but not necessarily in direct proportion. Using dp units (instead of px units) is a simple solution to making the view dimensions in your layout resize properly for different screen densities. In other words, it provides consistency for the real-world sizes of your UI elements across different devices.

sp

Scale-independent Pixels - This is like the dp unit, but it is also scaled by the user's font size preference. It is recommend you use this unit when specifying font sizes, so they will be adjusted for both the screen density and the user's preference.

pt

Points - 1/72 of an inch based on the physical size of the screen, assuming a 72dpi density screen.

px

Pixels - Corresponds to actual pixels on the screen. This unit of measure is not recommended because the actual representation can vary across devices; each devices may have a different number of pixels per inch and may have more or fewer total pixels available on the screen.

mm

Millimeters - Based on the physical size of the screen.

in

Inches - Based on the physical size of the screen.

### View / View group

#### View

* View objects are the basic building blocks of User Interface (UI) elements in Android.
* View is a simple rectangle box, which responds to the user's actions.
* Examples are EditText, Button, CheckBox, etc...
* View refers to the android.view.View class, which is the base class of all UI classes.

#### ViewGroup

* ViewGroup is the invisible container. It holds View and ViewGroup
* For example, LinearLayout is the ViewGroup that contains Button (View) and other Layouts also.
* ViewGroup is the base class for Layouts.

### When to use which View group?

Depending on the necessity of you layout you need to identify which viewgroup would help you.

Linear Layout

* LinearLayout is a view group that aligns all children in a single direction, vertically or horizontally.

Relative Layout

* RelativeLayout is a view group that displays child views in relative positions.

Table Layout

* TableLayout is a view that groups views into rows and columns.

Absolute Layout

* AbsoluteLayout enables you to specify the exact location of its children.

Frame Layout

* The FrameLayout is a placeholder on screen that you can use to display a single view.

List View

* ListView is a view group that displays a list of scrollable items.

Grid View

* GridView is a ViewGroup that displays items in a two-dimensional, scrollable grid.

### What is Context? Types?

It is the context of current state of the application/object. It lets newly created objects understand what has been going on. Typically, you call it to get information regarding another part of your program (activity and package/application).

You can get the context by invoking getApplicationContext(), getContext(), getBaseContext() or this (when in a class that extends from Context, such as the Application, Activity, Service and IntentService classes).

Typical uses of context:

* Creating new objects: Creating new views, adapters, listeners:

TextView tv = new TextView(getContext());

ListAdapter adapter = new SimpleCursorAdapter(getApplicationContext(), ...);

* Accessing standard common resources: Services like LAYOUT\_INFLATER\_SERVICE, SharedPreferences:

context.getSystemService(LAYOUT\_INFLATER\_SERVICE)

getApplicationContext().getSharedPreferences(\*name\*, \*mode\*);

* Accessing components implicitly: Regarding content providers, broadcasts, intent

getApplicationContext().getContentResolver().query(uri, ...);

#### Application Context

It is an instance which is the singleton and can be accessed in an activity via getApplicationContext(). This context is tied to the lifecycle of an application. The application context can be used where you need a context whose lifecycle is separate from the current context or when you are passing a context beyond the scope of an activity.

#### Activity Context

This context is available in an activity. This context is tied to the lifecycle of an activity. The activity context should be used when you are passing the context in the scope of an activity or you need the context whose lifecycle is attached to the current context.

### What are the components in android?

App components are the essential building blocks of an Android app. Each component is an entry point through which the system or a user can enter your app. Some components depend on others.

There are four different types of app components:

* Activities
* Services
* Broadcast receivers
* Content providers

Each type serves a distinct purpose and has a distinct lifecycle that defines how the component is created and destroyed.

#### Activities

An activity is the entry point for interacting with the user.

It represents a single screen with a user interface.

An activity facilitates the following key interactions between system and app:

* Keeping track of what the user currently cares about (what is on screen) to ensure that the system keeps running the process that is hosting the activity.
* Knowing that previously used processes contain things the user may return to (stopped activities), and thus more highly prioritize keeping those processes around.
* Helping the app handle having its process killed so the user can return to activities with their previous state restored.
* Providing a way for apps to implement user flows between each other, and for the system to coordinate these flows. (The classic example here being share.)

#### Services

A service is a general-purpose entry point for keeping an app running in the background for all kinds of reasons.

A Service is a component that runs in the background to perform long-running operations or to perform work for remote processes.

A service does not provide a user interface.

There are two type of services:

* Started Services
  + Tell the system to keep them running until their work is completed
* Bound Services
  + Run because some other app (or the system) has said that it wants to make use of the service.
* A service is implemented as a subclass of Service.

#### BroadCast Receivers

A broadcast receiver is a component that enables the system to deliver events to the app outside of a regular user flow, allowing the app to respond to system-wide broadcast announcements.

Because broadcast receivers are another well-defined entry into the app, the system can deliver broadcasts even to apps that are not currently running

Although broadcast receivers do not display a user interface, they may create a status bar notification to alert the user when a broadcast event occurs. More commonly, though, a broadcast receiver is just a gateway to other components and is intended to do a very minimal amount of work

A broadcast receiver is implemented as a subclass of BroadcastReceiver and each broadcast is delivered as an Intent object

#### Content Providers

A content provider manages a shared set of app data that you can store in the file system, in a SQLite database, on the web, or on any other persistent storage location that your app can access.

Through the content provider, other apps can query or modify the data if the content provider allows it.

To the system, a content provider is an entry point into an app for publishing named data items, identified by a URI scheme. Thus, an app can decide how it wants to map the data it contains to a URI namespace, handing out those URIs to other entities, which can in turn use them to access the data.

Content providers are also useful for reading and writing data that is private to your app and not shared.

A content provider is implemented as a subclass of ContentProvider and must implement a standard set of APIs that enable other apps to perform transactions.

### What is the activity lifecycle?

As a user navigates through, out of, and back to your app, the Activity instances in your app transition through different states in their lifecycle. The Activity class provides a number of callbacks that allow the activity to know that a state has changed: that the system is creating, stopping, or resuming an activity, or destroying the process in which the activity resides.



#### onCreate()

You must implement this callback, which fires when the system first creates the activity. On activity creation, the activity enters the Created state. In the onCreate() method, you perform basic application startup logic that should happen only once for the entire life of the activity.

#### onStart()

When the activity enters the Started state, the system invokes this callback. The onStart() call makes the activity visible to the user, as the app prepares for the activity to enter the foreground and become interactive.

#### onResume()

When the activity enters the Resumed state, it comes to the foreground, and then the system invokes the onResume() callback. This is the state in which the app interacts with the user. The app stays in this state until something happens to take focus away from the app. Such an event might be, for instance, receiving a phone call, the user’s navigating to another activity, or the device screen’s turning off.

#### onPause()

The system calls this method as the first indication that the user is leaving your activity (though it does not always mean the activity is being destroyed); it indicates that the activity is no longer in the foreground (though it may still be visible if the user is in multi-window mode). Use the onPause() method to pause or adjust operations that should not continue (or should continue in moderation) while the Activity is in the Paused state, and that you expect to resume shortly. There are several reasons why an activity may enter this state.

#### onStop()

When your activity is no longer visible to the user, it has entered the Stopped state, and the system invokes the onStop() callback. This may occur, for example, when a newly launched activity covers the entire screen. The system may also call onStop() when the activity has finished running, and is about to be terminated.

#### onDestroy()

onDestroy() is called before the activity is destroyed. The system invokes this callback either because:

* The activity is finishing (due to the user completely dismissing the activity or due to finish() being called on the activity), or
* The system is temporarily destroying the activity due to a configuration change (such as device rotation or multi-window mode)

When the activity moves to the stopped state, any lifecycle-aware component tied to the activity's lifecycle will receive the ON\_DESTROY event. This is where the lifecycle components can clean up anything it needs to before the Activity is destroyed.

### How to handle configuration changes?

There are various situations such as when the screen orientation is rotated where the Activity can actually be destroyed and removed from memory and then re-created from scratch again. In these situations, the best practice is to prepare for cases where the Activity is re-created by properly saving and restoring the state.

#### Saving and Restoring Activity State

As your activity begins to stop, the system calls onSaveInstanceState() so your activity can save state information with a collection of key-value pairs. The default implementation of this method automatically saves information about the state of the activity's view hierarchy, such as the text in an EditText widget or the scroll position of a ListView.

To save additional state information for your activity, you must implement onSaveInstanceState() and add key-value pairs to the Bundle object.

#### Saving and Restoring Fragment State

Fragments also have a onSaveInstanceState() method which is called when their state needs to be saved.

#### Retaining Fragments

In many cases, we can avoid problems when an Activity is re-created by simply using fragments. If your views and state are within a fragment, we can easily have the fragment be retained when the activity is re-created

### How to avoid activity from getting destroyed on rotation?

If your application does not need to update resources during a specific configuration change and you have a performance limitation that requires you to avoid the activity restart, then you can declare that your activity handles the configuration change itself, which prevents the system from restarting your activity.

However, this technique should be considered a last resort when you must avoid restarts due to a configuration change and is not recommended for most applications. To take this approach, we must add the android:configChanges node to the activity within the AndroidManifest.xml:

<activity android:name=".MyActivity"

android:configChanges="orientation|screenSize|keyboardHidden"

android:label="@string/app\_name">

Now, when one of these configurations change, the activity does not restart but instead receives a call to onConfigurationChanged():

// Within the activity which receives these changes

// Checks the current device orientation, and toasts accordingly

@Override

public void onConfigurationChanged(Configuration newConfig) {

super.onConfigurationChanged(newConfig);

// Checks the orientation of the screen

if (newConfig.orientation == Configuration.ORIENTATION\_LANDSCAPE) {

Toast.makeText(this, "landscape", Toast.LENGTH\_SHORT).show();

} else if (newConfig.orientation == Configuration.ORIENTATION\_PORTRAIT){

Toast.makeText(this, "portrait", Toast.LENGTH\_SHORT).show();

}

}

### What is an intent?

An intent is an abstract description of an operation to be performed. It can be used with startActivity to launch an Activity, broadcastIntent to send it to any interested BroadcastReceiver components, and Context.startService(Intent) or Context.bindService(Intent, ServiceConnection, int) to communicate with a background Service.

An Intent provides a facility for performing late runtime binding between the code in different applications. Its most significant use is in the launching of activities, where it can be thought of as the glue between activities. It is basically a passive data structure holding an abstract description of an action to be performed.

#### Intent Structure

The primary pieces of information in an intent are:

* Action
  + The general action to be performed, such as ACTION\_VIEW, ACTION\_EDIT, ACTION\_MAIN, etc.
* data
  + The data to operate on, such as a person record in the contacts database, expressed as a Uri.

Some examples of action/data pairs are:

* ACTION\_VIEW content://contacts/people/1
  + Display information about the person whose identifier is "1".
* ACTION\_DIAL content://contacts/people/1
  + Display the phone dialer with the person filled in.
* ACTION\_VIEW tel:123
  + Display the phone dialer with the given number filled in. Note how the VIEW action does what is considered the most reasonable thing for a particular URI.
* ACTION\_DIAL <tel:123>
  + Display the phone dialer with the given number filled in.
* ACTION\_EDIT content://contacts/people/1
  + Edit information about the person whose identifier is "1".
* ACTION\_VIEW content://contacts/people/
  + Display a list of people, which the user can browse through. This example is a typical top-level entry into the Contacts application, showing you the list of people. Selecting a particular person to view would result in a new intent { ACTION\_VIEW content://contacts/people/N } being used to start an activity to display that person.

### How to pass data among activities?

Android Intents are objects used to trigger actions from other Android Activities. One of the most common uses of Intents is to open a new Activity in your app. Often, you will want to pass information to the new Activity.

#### Intent Extras

Intent objects all have an extras property. This is a Bundle, storing key value pairs in which keys are Strings. It’s the same structure you see every time you override an Activity’s onCreate method. The bundle is one of the mechanisms through which Intents transfer data to other activities, and it is my main focus in this post.

#### Primitives, Strings, and Arrays

Primitives and Strings can easily be put into Intents. For example, the following code will store an int variable called intToSend in an Intent, giving the int the key “my-int.”

#### Serializables

One way to pass objects in Intents is for the object’s class to implement Serializable. This interface doesn’t require you to implement any methods; simply adding implements Serializable should be enough. To get the object back from the Intent, just call intent.getSerializableExtra. You’ll probably want to cast the return value to the expected value type.

This is a very simple approach. Unfortunately, it can be slow.

#### Parcelable

Another approach to sending objects is to implement Android’s Parcelable interface. This interface requires three things: a public void writeToParcel method, a public int describeContents, and a non-static field called CREATOR that implements Parcelable.Creator. writeToParcel is responsible for serializing the data, and the Creator is responsible for deserializing it to reconstruct the original Parcelable object.

### What is serializable? How does it work?

Serializable is a standard Java interface. It is not a part of the Android SDK. It’s simplicity is it’s beauty. Just by implementing this interface your plain old Java object will be ready to jump from one Activity to another.

Because Serializable is a marker interface, we do not have to implement tons of extra methods. When we ‘mark’ our plan old Java object with it, Java will try it is best to serialize it.

Of course, this simple approach has its price. Reflection is used during the process and many additional objects are created along the way. This can cause lots of garbage collection. The result is poor performance and battery drain.

### What is parcelable? How does it work?

Due to Android's memory management scheme, you will often find yourself needing to communicate with different components of your application, system components, or other applications installed on the phone. Parcelable will help you pass data between these components.

Android uses Binder to facilitate such communication in a highly optimized way. The Binder communicates with Parcels, which is a message container. The Binder marshals the Parcel to be sent, sends and receives it, and then unmarshals it on the other side to reconstruct a copy of the original Parcel.

Parcelable is an interface for classes whose instances can be written to and restored from a Parcel. It is a part of the Android SDK. Now, Parcelable was specifically designed in such a way that there is no reflection when using it. That is because, we are being really explicit for the serialization process.

Of course, there is a price we have to pay when using Parcelable as well! Because of the boilerplate code, it is much harder to maintain and understand the plain old Java object.

### What is the difference between them?

Serializable is a standard Java Interface. This interface relies on the automatic serialization process of Java, the process is apparently not custom at all and creates lots of garbage because it implements reflection, which makes it easy to implement but also makes it slower in runtime.

The beauty of serializable is that you only need to implement the Serializable interface on a class and its children. It is a marker interface, meaning that there is no method to implement; Java will simply do its best effort to serialize it efficiently.

The problem with this approach is that reflection is used and it is a slow process. This mechanism also tends to create many temporary objects and cause quite a bit of garbage collection.

Parcelable is a custom interface that it is specially made so that it does not implement reflection, this makes it faster in runtime but with the downside that it involves more work when implementing it into your code.

According to google engineers, this code will run significantly faster. One of the reasons for this is that we are being explicit about the serialization process instead of using reflection to infer it. It also stands to reason that the code has been heavily optimized for this purpose.

However, it is obvious here that implementing Parcelable is not free. There is a significant amount of boilerplate code and it makes the classes harder to read and maintain.

### What is Java Reflection?

Reflection is then the ability to make modifications at runtime by making use of introspection. The ability to inspect the code in the system and see object types is Type Introspection.

For example, say you have an object of an unknown type in Java, and you would like to call a 'doSomething' method on it if one exists. Java's static typing system isn't really designed to support this unless the object conforms to a known interface, but using reflection, your code can look at the object and find out if it has a method called 'doSomething' and then call it if you want to.

So, to give you a code example of this in Java (imagine the object in question is foo) :

Method method = foo.getClass().getMethod("doSomething", null);

method.invoke(foo, null);

One very common use case in Java is the usage with annotations. JUnit 4, for example, will use reflection to look through your classes for methods tagged with the @Test annotation, and will then call them when running the unit test.

### How does Garbage Collection work?

A managed memory environment, like the ART or Dalvik virtual machine, keeps track of each memory allocation. Once it determines that a piece of memory is no longer being used by the program, it frees it back to the heap, without any intervention from the programmer. The mechanism for reclaiming unused memory within a managed memory environment is known as garbage collection. Garbage collection has two goals: find data objects in a program that cannot be accessed in the future; and reclaim the resources used by those objects.

Android’s memory heap is a generational one, meaning that there are different buckets of allocations that it tracks, based on the expected life and size of an object being allocated. For example, recently allocated objects belong in the Young generation. When an object stays active long enough, it can be promoted to an older generation, followed by a permanent generation.

Each heap generation has its own dedicated upper limit on the amount of memory that objects there can occupy. Any time a generation starts to fill up, the system executes a garbage collection event in an attempt to free up memory. The duration of the garbage collection depends on which generation of objects it's collecting and how many active objects are in each generation.

Even though garbage collection can be quite fast, it can still affect your app's performance. You don’t generally control when a garbage collection event occurs from within your code. The system has a running set of criteria for determining when to perform garbage collection. When the criteria are satisfied, the system stops executing the process and begins garbage collection. If garbage collection occurs in the middle of an intensive processing loop like an animation or during music playback, it can increase processing time. This increase can potentially push code execution in your app past the recommended 16ms threshold for efficient and smooth frame rendering.

Additionally, your code flow may perform kinds of work that force garbage collection events to occur more often or make them last longer-than-normal. For example, if you allocate multiple objects in the innermost part of a for-loop during each frame of an alpha blending animation, you might pollute your memory heap with a lot of objects. In that circumstance, the garbage collector executes multiple garbage collection events and can degrade the performance of your app.

### Difference between ArrayList and LinkedList?

LinkedList and ArrayList are two different implementations of the List interface. LinkedList implements it with a doubly-linked list. ArrayList implements it with a dynamically re-sizing array.

As with standard linked list and array operations, the various methods will have different algorithmic runtimes.

For LinkedList<E>

* get(int index) is O(n) (with n/4 steps on average)
* add(E element) is O(1)
* add(int index, E element) is O(n) (with n/4 steps on average), but O(1) when index = 0 <--- main benefit of LinkedList<E>
* remove(int index) is O(n) (with n/4 steps on average)
* Iterator.remove() is O(1). <--- main benefit of LinkedList<E>
* ListIterator.add(E element) is O(1) This is one of the main benefits of LinkedList<E>

Note: Many of the operations need n/4 steps on average, constant number of steps in the best case (e.g. index = 0), and n/2 steps in worst case (middle of list)

For ArrayList<E>

* get(int index) is O(1) <--- main benefit of ArrayList<E>
* add(E element) is O(1) amortized, but O(n) worst-case since the array must be resized and copied
* add(int index, E element) is O(n) (with n/2 steps on average)
* remove(int index) is O(n) (with n/2 steps on average)
* Iterator.remove() is O(n) (with n/2 steps on average)
* ListIterator.add(E element) is O(n) (with n/2 steps on average)

Note: Many of the operations need n/2 steps on average, constant number of steps in the best case (end of list), n steps in the worst case (start of list)

LinkedList<E> allows for constant-time insertions or removals using iterators, but only sequential access of elements. In other words, you can walk the list forwards or backwards, but finding a position in the list takes time proportional to the size of the list. Javadoc says "operations that index into the list will traverse the list from the beginning or the end, whichever is closer", so those methods are O(n) (n/4 steps) on average, though O(1) for index = 0.

ArrayList<E>, on the other hand, allow fast random read access, so you can grab any element in constant time. But adding or removing from anywhere but the end requires shifting all the latter elements over, either to make an opening or fill the gap. Also, if you add more elements than the capacity of the underlying array, a new array (1.5 times the size) is allocated, and the old array is copied to the new one, so adding to an ArrayList is O(n) in the worst case but constant on average.

So depending on the operations you intend to do, you should choose the implementations accordingly. Iterating over either kind of List is practically equally cheap. (Iterating over an ArrayList is technically faster, but unless you're doing something really performance-sensitive, you shouldn't worry about this -- they're both constants.)

The main benefits of using a LinkedList arise when you re-use existing iterators to insert and remove elements. These operations can then be done in O(1) by changing the list locally only. In an array list, the remainder of the array needs to be moved (i.e. copied). On the other side, seeking in a LinkedList means following the links in O(n) (n/2 steps) for worst case, whereas in an ArrayList the desired position can be computed mathematically and accessed in O(1).

Another benefit of using a LinkedList arise when you add or remove from the head of the list, since those operations are O(1), while they are O(n) for ArrayList. Note that ArrayDeque may be a good alternative to LinkedList for adding and removing from the head, but it is not a List.

Also, if you have large lists, keep in mind that memory usage is also different. Each element of a LinkedList has more overhead since pointers to the next and previous elements are also stored. ArrayLists don't have this overhead. However, ArrayLists take up as much memory as is allocated for the capacity, regardless of whether elements have actually been added.

The default initial capacity of an ArrayList is pretty small (10 from Java 1.4 - 1.8). But since the underlying implementation is an array, the array must be resized if you add a lot of elements. To avoid the high cost of resizing when you know you're going to add a lot of elements, construct the ArrayList with a higher initial capacity.

### Difference between HashSet vs HashMap?

They are entirely different constructs. A HashMap is an implementation of Map. A Map maps keys to values. The key look up occurs using the hash.

On the other hand, a HashSet is an implementation of Set. A Set is designed to match the mathematical model of a set. A HashSet does use a HashMap to back its implementation, as you noted. However, it implements an entirely different interface.

HashSet

* HashSet class implements the Set interface
* In HashSet, we store objects(elements or values) e.g. If we have a HashSet of string elements then it could depict a set of HashSet elements: {“Hello”, “Hi”, “Bye”, “Run”}
* HashSet does not allow duplicate elements that mean you can not store duplicate values in HashSet.
* HashSet permits to have a single null value.

HashSet is not synchronized which means they are not suitable for thread-safe operations until unless synchronized explicitly.

### How does a HashMap work?

HashMap

* HashMap class implements the Map interface
* HashMap is used for storing key & value pairs. In short, it maintains the mapping of key & value (The HashMap class is roughly equivalent to Hashtable, except that it is unsynchronized and permits nulls.) This is how you could represent HashMap elements if it has integer key and value of String type: e.g. {1->”Hello”, 2->”Hi”, 3->”Bye”, 4->”Run”}
* HashMap does not allow duplicate keys however it allows having duplicate values.
* HashMap permits single null key and any number of null values.

HashMap is not synchronized which means they are not suitable for thread-safe operations until unless synchronized explicitly.

### What is the difference between HashTable and HashMap?

There are several differences between HashMap and Hashtable in Java:

* Hashtable is synchronized, whereas HashMap is not. This makes HashMap better for non-threaded applications, as unsynchronized Objects typically perform better than synchronized ones.
* Hashtable does not allow null keys or values. HashMap allows one null key and any number of null values.
* One of HashMap's subclasses is LinkedHashMap, so in the event that you'd want predictable iteration order (which is insertion order by default), you could easily swap out the HashMap for a LinkedHashMap. This wouldn't be as easy if you were using Hashtable.

If synchronization is not an issue for you, you can use HashMap. If synchronization becomes an issue, you may also look at ConcurrentHashMap.