1. What is an extension function?

It is a member function of a class that is defined outside the class.

Example

you need to use a method to the String class that returns a new string with first and last character removed; this method is not already available in String class. You can use extension function to accomplish this task.

```
fun String.removeFirstLastChar(): String =  this.substring(1, this.length - 1)

fun main(args: Array<String>) {
   val myString= "Hello Everyone"
   val result = myString.removeFirstLastChar()
   println("First character is: $result")
}
```

Output:

```
First character is: ello Everyon
```

2. What does Kotlin have for null safety

Kotlin compiler by default doesn't allow any types to have a value of null at compile-time.
For Kotlin, Nullability is a type. At a higher level, a Kotlin type fits in either of the two.

- Nullability Type
- Non-Nullability Type

Types:

- Safe Calls **?.**
    - Executes the relevant call only when the value is non-null. Else it prints a null for you.
- !! Operator(not-null assertion) **!!**
    - Converts the nullable reference into its non-nullable type irrespective of whether it's a null or not. This should be used only when you're certain that the value is NOT NULL. Else it'll lead to an NPE exception.
- Safe Casting **as?**
    - Can be used for preventing ClassCastException as well(another commonly seen exception).

- Elvis Operator **?:**
    - Allows us to set a default value instead of the null
- Using let()
    - Executes the lamda function specified only when the reference is non-nullable, and allow us to use **it** → contains the non-null value of newString
- Using also()
    - Also behaves the same way as let except that it's generally used to log the values. It can't assign the value of it to another variable. The statement present inside let can't be placed in also. Vice-versa can work though.
- Filtering Out Null Values **.filterNotNull()**
    - We can filter out Null Values from a collection type using the function
- Java Interoperability
    - Allow you to set the Java Annotation **@Nullable** or **@NotNull**, the Kotlin compiler would consider them as Nullable or Not Nullable References.
    - Remember : Nullable references require a ? in Kotlin.

3. What advantages does Kotlin have over Java?

DATA CLASSES

In simple terms, ‘data classes’ are containers for data.

JAVA
If If you come from a Java background, you’ll know how much code you have to write to get these set up. Say we are building an app that has two screens, initial authentication and then a home screen with a ‘social feed’ style list of posts. For this, we’d need a container for the ‘User Object’, as the social feed items: Picture, Video and PhotoAlbum.

If we were building this in Java, it would be a fairly involved process:

- First, we’d have to define class fields,
- Then we would need a ‘getter’ and a ‘setter’ for each property,
- Lastly, we would employ ‘equals’ and ‘hashCode’ methods.

A simple data class with few properties would easily reach dozens of lines of code.

KOTLIN
Replaces this excess with a data class: a perfect fit for our use case.

```
1. data class User(val name: String, val email: String)
2. data class Picture(val url: String)
3. data class Video(val url: String)
4. data class PhotoAlbum(val photos: List<Photo>)
```

That’s it, in just 4 data classes and 4 lines of code! In fact, we can actually further condense this into just one Kotlin file. When we write a data class in Kotlin, ‘getters’, ‘setters’, ‘equals’ and ‘hashCode’ methods are automatically generated once this code is converted to JVM byte-code.

COROUTINES

JAVA
Java async execution on Android is not fun to do. You have to juggle managing the UI thread and a network call at the same time. There are two main ways to do this:

- Callbacks. These aren’t pretty, and they don’t allow a request to be canceled.
- rxJava. If you can get over the steep learning curve, rxJava does enable a request to be canceled but it has no callbacks.

KOTLIN

Kotlin's coroutines help us write async code in a blocking fashion. This way we can just focus on the data rather than how to fetch it. Coroutines are lightweight threads. They don't block the execution of the thread they work in, but they can suspend their own execution.

EXTENSION FUNCTIONS

For this explication we're going to use the example to show a 'toast' message in Android we need to specify the context, message, and duration of the toast. But all we typically care about is showing the message itself. So, wouldn't it be nice if we could somehow just pass a message to a function and magically get our 'toast' shown on the screen? Let's build that magic function.

JAVA

The Java style is to write a static utility function that can display a 'toast'. One of the downsides of utility classes is that they have to be directly referenced from all over your code base. This directly couples your code with that utility class and, in turn, goes against SOLID principles of quality software.

KOTLIN

```
1. fun Context.toast(message: String, short: Boolean = true) {
2.     val duration = if (short) Toast.LENGTH_SHORT else Toast.LENGTH_LONG
3.     Toast.makeText(this, message, duration).show()
4. }
```

To use our 'toast extension' function, we can simply write toast(message = "woow") in Activities or Fragments. This is much cleaner than a method call on a utility class. What Kotlin does, behind the scenes, is to create a utility function and injects it to the caller's site without you doing it manually.

NULL SAFETY

Kotlin there is the concept of nullable and non-nullable variables. For non-nullable variables, you have to specify a value immediately upon creating an object (the exception is the late init var modifier). There is no way to have a null pointer exception and there is no need for null checks. The rule of thumb is this: If you know something may be null (such as data coming from a network call), use nullable variables to protect yourself from null pointer exception.

DATA SERIALIZATION

JAVA
What if you want to share objects between screens while developing an Android app? The recommended way is to utilize the Parcelable serialization provided in the Android platform, but to do this we would need to write custom code for each individual property to 'marshal' and 'unmarshal' it. Not to mention maintaining that code as the class changes.

KOTLIN
All it takes is a single annotation – @Parcelize – on the desired class. Better still, if you have a nested structure of objects Kotlin will automatically serialize the whole tree! Since serializing data is cheap and easy, we can utilize this feature to create a better and more consistent user experience. For example, when transferring data from a list screen to a detailed view, it would be nice to show title and description right away while the rest of the data is being fetched.

4. How do you implement headless fragments?

They are fragments without any UI) for executing long running tasks and then passing the data back to Activity using WeakReferences. Headless Fragments are basically Fragments with setRetainInstance() set to true.

1. attach the Headless Fragment to the Activity

```
public class ExampleActivity extends AppCompatActivity {
private ExampleActivityStateFragment mStateFragment;
@Override
  protected void onCreate(Bundle savedInstanceState) {
     super.onCreate(savedInstanceState);

     FragmentManager fm = getSupportFragmentManager();
     mStateFragment = (ExampleActivityStateFragment)
fm.findFragmentByTag(KEY_STATE_FRAGMENT);
     if (mStateFragment == null) {
        mStateFragment = new ExampleActivityStateFragment();
        fm.beginTransaction().add(mStateFragment, KEY_STATE_FRAGMENT).commit();
     }
  }
}
```

2. The above block checks to see if we have already created a StateFragment (we called it StateFragment in code) and if so, we can just find that Fragment by the unique TAG for each Fragment. If not, we create one and attach it to the Activity.
3. Set the SetUp to save the data with the StateFragment, fetches the data

4. When the Activity is rotated and getData() is called again, the StateFragment's mList won't be null, because the StateFragment retained its instance and is not killed like its parent Activity. And bingo! we save the data across rotation.
5. setting the views to null in onDestroy()