CheatSheet

Design Patterns

Builder Design Pattern

The Builder design pattern is a creational design pattern that provides a flexible solution for constructing complex objects. This pattern is particularly useful when an object requires many steps to create and when these steps need to be executed in a specific sequence. The Builder pattern helps in producing different types and representations of an object using the same construction code.

Components of the Builder Pattern

The Builder pattern consists of several key components:

1. **Product**: The complex object that is being built.
2. **Builder**: An abstract interface for creating parts of the Product object.
3. **Concrete Builder**: Implements the Builder interface and provides an implementation for the steps necessary to construct a product. Each Concrete Builder corresponds to a different representation of the product.
4. **Director**: Constructs an object using the Builder interface. The Director is responsible for managing the correct sequence of object creation. It is isolated from the concrete implementation of each component of the product.
5. **Client**: The part of the application that calls the Director to construct the Product. The Client must associate the appropriate Builder with the Director.

How It Works

Here's a high-level overview of how the Builder pattern works:

1. **Client**: The process begins with the client choosing a specific Builder and passing it to the Director to initiate the building process.
2. **Director**: The Director guides the Builder on what parts to build and in what order. The Director is aware of the steps needed to construct the product but doesn't know how these steps are implemented.
3. **Builder**: Each Builder implements the building instructions for constructing parts of the product. Builders handle the details of creating and assembling each part of the product. They also provide a method to retrieve the final product.
4. **Product**: Once all parts are assembled, the product is returned to the client.

Example Scenario

Imagine a scenario where you need to build various types of computers (e.g., desktop, laptop, tablet). Each type of computer has different components (like CPU, memory, storage), but the assembly process might have similar steps.

- **Product**: Different types of computers.
- **Concrete Builders**: Each builder represents a different type of computer, handling specific component assembly.
- **Director**: Orchestrates the construction of computers by invoking the appropriate steps on the builders.
- **Client**: Might be a configuration system for a computer store allowing users to select computer specifications.

Benefits of Using the Builder Pattern

- **Encapsulation and Control**: The Builder pattern encapsulates the construction logic for the product. The director controls the order of construction.
- **Flexibility**: It allows for constructing different representations of the product using the same construction code.
- **Separation of Concerns**: It separates the construction of a complex object from its representation, which allows for better modularity and easier scalability.

This pattern is widely used in scenarios where multiple parts need to be assembled to form a complex object, especially when the construction steps might vary or require a specific sequence.

Nested static class so we don't have to account for all sequences

```
public class TeaTwo {
        private boolean sugar;
        private boolean milk;
        private TeaTwo(TeaBuilder teaBuilder) {
        this.sugar = teaBuilder.sugar;
        this.milk = teaBuilder.milk;
        }
        static class TeaBuilder {
                private boolean sugar;
                private boolean milk;
                TeaBuilder(){};
                public TeaBuilder setSugar (boolean sugar) {
                this.sugar = sugar;
                return this;
                }
                public TeaBuilder setMilk (boolean milk) {
                this.milk = milk;
                return this;
                }
                public TeaTwo bulid () {
                return new TeaTwo(this);
                }
        }
}
```

How its used:

```
TeaTwo teaTwo = new TeaTwo.TeaBuilder().setSugar(true).setMilk(true).build();
```

Observer Design Pattern

The Observer design pattern is a widely used design pattern in software engineering that defines a one-to-many dependency between objects. This means when one object changes its state, all of its dependents are notified and updated automatically. It's particularly useful in scenarios where changes to one object require changes to others, and you don't want these objects to be tightly coupled.

Components of the Observer Pattern

The Observer pattern involves two primary types of participants:

1. **Subject**: This is the object that holds the state that, when changed, needs to notify all attached observers. The Subject typically maintains a list of its dependents, the Observers, and provides an interface for attaching and detaching Observer objects.
2. **Observers**: These are the objects that need to be notified of changes in the Subject. They implement an update interface that will be called when the Subject changes its state.

How It Works

Here's a step-by-step explanation of how the Observer pattern works:

1. **Subject Interface**: This interface allows attaching and detaching Observers to the Subject. It also includes the method to notify all observers of any state changes.
2. **Concrete Subject**: Implements the Subject interface. It maintains the state of the object and when changes occur, it notifies the attached Observers by calling their update method.
3. **Observer Interface**: This interface declares the update method that all Observers must implement in order to receive updates from the Subject.
4. **Concrete Observer**: Implements the Observer interface. Each observer registers with a concrete subject to receive updates.

Example Scenario

Imagine you have a weather station (the Subject) that tracks weather data like temperature, humidity, and pressure. You have multiple display elements (Observers) such as current conditions display, statistics display, and forecast display that need to show updated weather data when the weather station records new measurements.

Benefits of the Observer Pattern

- **Decoupling**: The pattern promotes loose coupling between the Subject and the Observers. The only thing the subject knows about an observer is that it implements a certain interface.
- **Support for Broadcast Communication**: Unlike regular communication between objects, the Observer pattern allows the subject to send broadcast notifications to observers without needing to know who these observers are.

- **Dynamic Relationships**: You can add and remove observers at any time without modifying the subject or other observers.

Implementation Example in Java

Here's a simple Java example illustrating the Observer pattern:

```
import java.util.ArrayList;
import java.util.List;

// Subject interface
interface Subject {
    void attach(Observer o);
    void detach(Observer o);
    void notifyUpdate(Message m);
}

// Concrete Subject
class ConcreteSubject implements Subject {
    private List<Observer> observers = new ArrayList<>();

    @Override
    public void attach(Observer o) {
        observers.add(o);
    }

    @Override
    public void detach(Observer o) {
        observers.remove(o);
    }

    @Override
    public void notifyUpdate(Message m) {
        for (Observer o : observers) {
            o.update(m);
        }
    }
}

// Observer interface
interface Observer {
    void update(Message m);
}

// Concrete Observer A
class ConcreteObserverA implements Observer {
    @Override
    public void update(Message m) {
        System.out.println("ConcreteObserverA is updated with message: " +
m.getMessageContent());
    }
```

```java
}

// Concrete Observer B
class ConcreteObserverB implements Observer {
    @Override
    public void update(Message m) {
        System.out.println("ConcreteObserverB is updated with message: " +
m.getMessageContent());
    }
}

// Message class used to communicate information from subject to observers
class Message {
    private String messageContent;

    public Message(String m) {
        this.messageContent = m;
    }

    public String getMessageContent() {
        return messageContent;
    }
}

// Main class to demonstrate Observer Pattern
public class Main {
    public static void main(String[] args) {
        ConcreteSubject sub = new ConcreteSubject();
        Observer obs1 = new ConcreteObserverA();
        Observer obs2 = new ConcreteObserverB();

        sub.attach(obs1);
        sub.attach(obs2);
        sub.notifyUpdate(new Message("New Data Available!"));
    }
}
```

This Java code effectively demonstrates the Observer pattern, where `ConcreteSubject` is the Subject, and `ConcreteObserverA` and `ConcreteObserverB` are the Observers. When the Subject's state changes (here simulated by calling `notifyUpdate`), all registered observers are notified.

Singleton

The Singleton Design Pattern is a creational pattern used to ensure that a class has only one instance while providing a global point of access to this instance. It is commonly used when exactly one object is needed to coordinate actions across the system. Examples of such cases include configurations, logging, and access to resources like database connections.

Motivation

Often in software engineering, certain components (like configuration managers) are ideally only instantiated once. This ensures consistent behavior across the system, efficient use of resources, and controlled access.

Implementation

The Singleton pattern can be implemented in Java in several ways. Here are the key elements in making a class a Singleton:

1. **Private Constructor**: This prevents the instantiation of the class from outside to ensure that the class can't be instantiated more than once.
2. **Private Static Instance**: This is the only instance of the class that will ever exist.
3. **Public Static Method**: This method provides the global point of access to the instance and ensures that the same instance is returned each time it is called.

Example of Singleton Pattern in Java

Here is a simple implementation of the Singleton pattern:

```
public class Singleton {
    // Private static variable of the same class that is the only instance of
the class
    private static Singleton instance;

    // Private constructor to prevent instantiation from other classes
    private Singleton() {}

    // Public static method that returns the instance of the class, the global
access point
    public static Singleton getInstance() {
        if (instance == null) {
            // If the instance is null, initialize it
            instance = new Singleton();
        }
        return instance;
    }
}
```

Thread-Safe Singleton

The basic implementation shown above is not safe for use in a multithreaded scenario. When multiple threads can access the creation logic at the same time, it might create multiple instances. To solve this issue, you can synchronize the instance creation process.

```
public class Singleton {
    private static Singleton instance;

    private Singleton() {}

    // Synchronized method to control simultaneous access
```

```java
    public static synchronized Singleton getInstance() {
        if (instance == null) {
            instance = new Singleton();
        }
        return instance;
    }
}
```

However, synchronization can lead to performance issues if it is done in the method that gets the instance. An alternative and better approach for thread safety is the double-checked locking principle.

```java
public class Singleton {
    // The volatile keyword ensures that multiple threads handle the
uniqueInstance variable correctly
    private static volatile Singleton instance;

    private Singleton() {}

    public static Singleton getInstance() {
        if (instance == null) { // First check (no locking)
            synchronized (Singleton.class) { // Synchronize on the class object
                if (instance == null) { // Second check (with locking)
                    instance = new Singleton();
                }
            }
        }
        return instance;
    }
}
```

Singleton with Enum (Effective Java by Joshua Bloch)

Another approach to implement Singleton in a thread-safe way is using an enum, which provides implicit support for thread safety and only one instance is guaranteed by the JVM.

```java
public enum Singleton {
    INSTANCE;

    public void doSomething() {
        System.out.println("Doing something...");
    }
}
```

Pros and Cons of Singleton

Pros:

- Controlled access to sole instance.
- Reduced namespace.

- Allows for refinement of operations and representation.
- Permits a variable number of instances.

Cons:

- Often considered an anti-pattern as it can lead to code that is harder to test.
- It reduces the flexibility in changing the instantiation strategy.
- Can lead to resource contention in multithreaded scenarios if not properly handled.

In summary, the Singleton pattern is useful for managing global state but should be used carefully due to its impact on code testability and its potential to create undesirable code coupling.

Android

JSON Files

Interpreting a JSON file in Java involves reading the file, parsing the JSON content, and then accessing the data as needed in your Java program. This can be achieved by using various libraries available in Java for handling JSON, such as Jackson, Gson, or the JSON.org library. I'll guide you through the process using each of these popular libraries.

Using JSON.org

The JSON.org library is lightweight and easy to use for simple JSON processing:

1. **Add JSON.org Dependency**:

   If using Maven, include this in your `pom.xml`:

   ```xml
   <dependency>
       <groupId>org.json</groupId>
       <artifactId>json</artifactId>
       <version>20210307</version>
   </dependency>
   ```

2. **Read and Parse JSON**:

   ```java
   import org.json.JSONObject;
   import org.json.JSONTokener;

   import java.io.FileInputStream;
   import java.io.FileNotFoundException;
   import java.io.InputStream;

   public class JsonOrgExample {
       public static void main(String[] args) {
           try (InputStream is = new FileInputStream("data.json")) {
               JSONTokener tokener = new JSONTokener(is);
               JSONObject object = new JSONObject(tokener);
               System.out.println(object);
           } catch (FileNotFoundException e) {
   ```

```
                e.printStackTrace();
            }
        }
    }
```

General Steps to Work with JSON in Java

1. **Add the appropriate library to your project** via Maven or Gradle.
2. **Create a Java class that matches the JSON structure** if you're using a binding library like Jackson or Gson.
3. **Read the JSON file using a `FileReader`, `InputStream`, or similar**.
4. **Parse the JSON content to Java objects** using the library methods.

These libraries provide robust tools for handling JSON, including converting complex, nested JSON into Java objects and vice versa. Choose one based on your specific needs and the complexity of the JSON data you're dealing with.

XML Files

An Android XML file is crucial in Android application development. XML (eXtensible Markup Language) files in Android serve various purposes such as defining user interfaces, resources, and configurations. Here's a breakdown of how Android XML files are used and structured:

Types of Android XML Files

1. **Layout XML Files**: These files, typically stored in the `res/layout` directory, define the user interface of an activity, fragment, or any UI component. They specify the UI elements (like buttons, text views, layouts) and their arrangement on the screen.
2. **Drawable XML Files**: Located in `res/drawable`, these files define shapes, selectors, and state lists that can be used as backgrounds, icons, or other graphical assets.
3. **Values XML Files**: These files, stored in the `res/values` directory, contain simple values in a key-value pair format. They can define strings, dimensions, colors, styles, integers, and arrays that can be reused throughout the application.
4. **Manifest File**: The `AndroidManifest.xml` file at the root of the project source set contains essential information about the Android application, such as its components (activities, services, broadcast receivers, content providers), required permissions, hardware and software features, API level requirements, and more.

Anatomy of a Layout XML File

To illustrate, let's explore a basic layout XML file used in Android:

```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    android:orientation="vertical"
    android:padding="16dp">
    <TextView
        android:id="@+id/sample_text"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:text="@string/hello_world"
        android:layout_marginBottom="20dp" />

    <Button
        android:id="@+id/sample_button"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:text="@string/click_me" />

</LinearLayout>
```

Explanation:

- **XML Declaration**: `<?xml version="1.0" encoding="utf-8"?>` This line declares the XML version and the character encoding used in the file.
- **Root Element**: In this case, `<LinearLayout>` acts as the root of the UI structure. This tag supports attributes like `layout_width`, `layout_height`, `orientation`, and `padding`, which control the overall layout behavior.
  - `xmlns:android`: Defines the Android namespace URL, which is required to access Android-specific attributes.
  - `android:layout_width` and `android:layout_height`: These are mandatory attributes for all views and layouts. They can be set to specific dimensions, `match_parent` (occupy the space equal to the parent), or `wrap_content` (size needed to fit the content).
  - `android:orientation`: Specifies the stacking direction of the child views, either `horizontal` or `vertical`.
- **TextView**: Represents a UI element that displays text to the user.
  - `android:id`: Provides a unique identifier for the view, which can be used to reference it programmatically.
  - `android:text`: Sets the text displayed in the TextView, here referencing a string resource.
- **Button**: Represents a clickable UI element.
  - Similar attributes as TextView for dimensions and text.

How XML Files Are Used in Android

When an Android app runs, these XML files are parsed by the Android runtime to create view objects in memory. The attributes specified in XML configure the properties of these view objects. For example, setting a button’s `onClick` attribute in XML allows you to define what happens when the user taps the button without needing to configure it programmatically.

XML files are favored in Android for separating the presentation of the app from the business logic written in Java or Kotlin. This separation enhances code readability and maintainability and allows changing UI elements without altering the underlying codebase. Additionally, it simplifies the process of creating different layouts for various screen sizes and orientations by using different resources.

Android Activity Lifecycle

The Android lifecycle refers to the set of states an Android app or its components (like Activities, Fragments, Services, etc.) go through from the start of their creation to their destruction. Understanding these lifecycle states is crucial for building efficient, effective, and bug-free Android apps. Here, we'll primarily focus on the lifecycle of an Activity, which is one of the fundamental components of any Android application.

Activity Lifecycle

An Activity in Android is a single screen with a user interface. The lifecycle of an Activity is managed by a series of callback methods that the Android system calls. Here is a brief overview of each stage in the lifecycle of an Activity:

1. **`onCreate(Bundle savedInstanceState)`**
   - **Called when the activity is first created.**
   - This is where you should do all static setup: create views, bind data to lists, etc. This method also provides you with a `Bundle` containing the activity's previously frozen state, if there was one.
   - Always followed by `onStart()`.
2. **`onStart()`**
   - **Called when the activity becomes visible to the user.**
   - Follows `onCreate(Bundle savedInstanceState)` and precedes `onResume()`.
3. **`onResume()`**
   - **Called when the activity starts interacting with the user.**
   - At this point, your activity is at the top of the activity stack, and captures all user input. Most of your activity's normal activity processing is done during the `onResume()` phase.
   - Followed by `onPause()`.
4. **`onPause()`**
   - **Called when the system is about to start resuming another activity.**
   - This callback is mostly used for pausing ongoing activities that should not continue (like animations or video playback) while the activity is not in the foreground. It is also a good place to save data that should be persisted beyond the current user session.
   - May be followed either by `onResume()` if the activity returns back to the front, or by `onStop()` if it becomes invisible to the user.
5. **`onStop()`**
   - **Called when the activity is no longer visible to the user.**
   - This may happen because the activity is being destroyed, a new activity is starting, or an existing one is being brought in front of the current activity.
   - Followed by either `onRestart()` if the activity is coming back to interact with the user, or by `onDestroy()` if this activity is going away.
6. **`onRestart()`**

    - **Called after the activity has been stopped, just before it starts again.**
    - Always followed by `onStart()`.
7. **`onDestroy()`**
    - **Called before the activity is destroyed.**
    - This can be because the activity is finishing (due to the user completely dismissing the activity or due to `finish()` being called on the activity), or because the system is temporarily destroying this instance of the activity to save space.

Managing State Across Lifecycle Events

Handling the lifecycle correctly is essential for maintaining state and ensuring a good user experience. For example, you might start a background task in `onResume()` and stop it in `onPause()` to make sure it only runs while the activity is active. Similarly, you might commit unsaved changes to persistent storage in `onPause()` or `onStop()` because the changes the user expects to persist should be saved when the user leaves the activity.

Lifecycle-aware components, a newer addition with Android Architecture Components, can help manage lifecycle events better and ensure that your Android apps are well-behaved in various scenarios like configuration changes (e.g., screen rotations) or system-initiated process death. These components provide a way to create classes that are automatically subscribed to appropriate lifecycle events, simplifying lifecycle management significantly.

Understanding and managing these lifecycle events correctly is crucial to making Android applications that behave well across different devices and changing user conditions.

R class

In Android programming, the `R` class is an automatically generated class that acts as a bridge between your application's resources and your application code. This class is created by the Android build system when your app is compiled. It contains a series of static nested classes, each corresponding to a different type of resource that you have in your app, such as layouts, strings, colors, drawables, etc.

Structure of the R class

Each type of resource defined in your app’s `res/` directory gets an inner class within `R`. For example:

- `R.layout` contains constants for all layouts defined in the `res/layout/` folder.
- `R.string` contains constants for all strings defined in the `res/values/strings.xml`.
- `R.drawable` contains constants for all drawable assets located in the `res/drawable/` folder.
- `R.id` contains constants for all unique resource IDs defined in your XML files. These IDs are typically assigned in layout files using the `@+id/` syntax and are used to reference UI components (like buttons, text views, etc.) from your code.

Usage of the R class

The `R` class facilitates referencing resources from your Java or Kotlin code. Here’s an example of how it’s used in an Android application:

```
setContentView(R.layout.activity_main); // Setting the layout for an Activity

TextView textView = findViewById(R.id.my_text_view); // Accessing a TextView by its ID
textView.setText(R.string.greeting); // Setting text from string resources
```

In this example:

- `R.layout.activity_main` refers to an XML layout file named `activity_main.xml` in the `res/layout/` directory.
- `R.id.my_text_view` refers to a `TextView` in your layout file that has the ID `my_text_view`.
- `R.string.greeting` refers to a string resource named `greeting` in `strings.xml`.

Importance of the R class

Using the `R` class to reference resources has several benefits:

1. **Type Safety**: Resources accessed via the `R` class are checked at compile-time, which means you'll catch any missing resources when you compile your app, rather than at runtime.
2. **Ease of Use**: It simplifies the access to various resources, avoiding the need for manual handling of resource identifiers.
3. **Maintenance**: When resources are updated (e.g., adding, removing, or renaming resources), the `R` class is automatically regenerated to reflect these changes. This ensures that references in your code are always synchronized with actual resources.

Overall, the `R` class is a fundamental component in Android development, providing a structured and error-resistant way to work with resources in your applications.

Intents

In Android development, an **Intent** is a fundamental concept that facilitates communication between components in an application or between different applications. It acts as a messaging object you can use to request an action from another app component, whether within your own application or from another application.

Types of Intents

There are two types of Intents in Android:

1. **Explicit Intents**
    - **Purpose**: Used to start a specific component (such as an Activity or Service) within your own application. When using explicit intents, you specify the Java class of the component you want to start.
    - **Example**: Starting a new activity within your app to display a message.

    ```
    Intent intent = new Intent(this, DisplayMessageActivity.class);
    startActivity(intent);
    ```

2. **Implicit Intents**

- **Purpose**: Do not name a specific component; instead, they declare a general action to perform, which allows a component from another app to handle it.
- **Example**: If you want to show the user a location on a map, you can use an implicit intent to request that another capable app show a specified location on a map.

```
Uri location = Uri.parse("geo:0,0?
q=1600+Amphitheatre+Parkway,+Mountain+View,+California");
Intent mapIntent = new Intent(Intent.ACTION_VIEW, location);
if (mapIntent.resolveActivity(getPackageManager()) != null) {
    startActivity(mapIntent);
}
```

Common Uses of Intents

- **Starting Activities**: An Intent can be used to start an activity, specifying the activity to start and carrying necessary data.
- **Starting Services**: Similar to activities, Intents can start services.
- **Delivering Broadcasts**: You can send and receive broadcast messages from other applications using Intents. These broadcasts can be system-defined (like battery low, screen turned off) or custom defined.
- **Data Transfer**: Intents are often used to transfer data between activities. Data can be added to Intents in key-value pairs using the `putExtra()` method, and retrieved in the started activity using `getIntent().getExtras()`.

Intent Filters

- **Purpose**: Used in the manifest file or in component code to specify the types of intents that a component can respond to. When an implicit intent is broadcasted, the Android system filters it through all components based on the intent filters and starts the component that matches the intent filter.
- **Manifest Declaration Example**: An activity might declare that it handles all web URLs.

```
<activity android:name=".ExampleActivity">
    <intent-filter>
        <action android:name="android.intent.action.VIEW" />
        <category android:name="android.intent.category.DEFAULT" />
        <data android:scheme="http" />
        <data android:scheme="https" />
    </intent-filter>
</activity>
```

Intent Extras

- **Purpose**: Intents can carry bundled key-value pairs, where keys are generally strings. These pairs are known as extras and are used to pass data between activities.
- **Example**: Passing a string from one activity to another.

```java
Intent intent = new Intent(this, NextActivity.class);
intent.putExtra("message_key", "Hello, NextActivity!");
startActivity(intent);
```

In summary, Intents are a versatile mechanism for communication both within an app and between apps, supporting everything from simple actions like opening a web page to complex interactions such as setting an alarm. They are a central part of how apps manage transitions and share data in Android.

Shared Preferences

`SharedPreferences` in Android is a mechanism for storing and retrieving small amounts of data in the form of key-value pairs. This data persists across user sessions, meaning it remains available even after the application is closed and reopened. `SharedPreferences` is commonly used for storing preferences or other data that needs to be persistent but doesn't require the use of a database.

Key Characteristics of SharedPreferences

- **Persistence**: Data stored in `SharedPreferences` persists across user sessions.
- **Data Type Support**: Supports basic data types such as `boolean`, `float`, `int`, `long`, and `String`.
- **Ease of Use**: Provides a simple API for reading and writing data.
- **Not for Sensitive Data**: It's not secure storage, so sensitive information should not be stored in `SharedPreferences` without encryption.
- **Mode of Operation**: Can be private (default) or shared with other apps depending on the mode selected when opening the `SharedPreferences` file.

Common Use Cases

- Storing user settings, such as notification preferences or display options.
- Keeping track of whether the user has seen an introductory screen or tutorial.
- Storing simple user data, such as user's high score in a game or recent searches.

How to Use SharedPreferences

Here’s a step-by-step guide on how to use `SharedPreferences`:

1. Getting SharedPreferences Instance

You can get a `SharedPreferences` instance using one of these methods:

- **getDefaultSharedPreferences**: If you need just one shared preference file for your activity, you can use this method from the PreferenceManager class. It creates a preference file in private mode with a default name.

```
SharedPreferences prefs =
PreferenceManager.getDefaultSharedPreferences(context);
```

- **getSharedPreferences**: If you need multiple shared preference files identified by name, or you need a specific file with a mode other than private, you use this method from your Context.

```
SharedPreferences prefs = context.getSharedPreferences("MyPrefsFile",
Context.MODE_PRIVATE);
```

2. Writing to SharedPreferences

To write data to `SharedPreferences`, you need an editor object to make changes and commit those changes.

```
SharedPreferences.Editor editor = prefs.edit();
editor.putString("key_string", "Hello, World!");
editor.putInt("key_int", 123);
editor.commit(); // or editor.apply() for asynchronous write
```

- **commit() vs apply()**:
  - `commit()`: Synchronously saves the data to persistent storage and returns a boolean value indicating success or failure.
  - `apply()`: Asynchronously saves the data to persistent storage and doesn't return a result.

3. Reading from SharedPreferences

Reading data from `SharedPreferences` is straightforward; you simply provide the key and a default value if the key doesn't exist.

```
String stringValue = prefs.getString("key_string", "DefaultString");
int intValue = prefs.getInt("key_int", 0);
```

4. Removing or Clearing SharedPreferences

You can also remove specific entries or clear all the entries in the `SharedPreferences` file.

```
// To remove a specific entry
editor.remove("key_string");
editor.commit();

// To clear all the data from SharedPreferences
editor.clear();
editor.commit();
```

Security Considerations

Since `SharedPreferences` are stored in plain text, it is not suitable for storing sensitive information like passwords or personal identification numbers (PINs) without proper encryption mechanisms in place.

`SharedPreferences` provides a simple yet powerful option for data storage and retrieval of small quantities of data in Android applications. Its ease of use and persistence make it a popular choice for managing user preferences and other similar data requirements.

Concurrent Programming

Concurrent programming in Android refers to performing multiple tasks in parallel to make optimal use of system resources and improve the overall efficiency and performance of applications. In Android, concurrent programming is essential because the main thread (also known as the UI thread) is responsible for handling user interface operations and user interactions. If time-consuming tasks such as network operations, complex calculations, or database queries are performed on the UI thread, the application can become unresponsive and may even lead to an Application Not Responding (ANR) error.

Key Concepts in Android Concurrency:

1. **UI Thread/Main Thread**: This is the thread on which user interface operations are performed. Any updates to the UI must be done on this thread. Long-running operations on this thread can block the UI, making the application seem unresponsive.
2. **Background Thread**: Tasks that take a considerable amount of time should be offloaded to background threads. This prevents the UI thread from being blocked.
3. **Thread-safe Operations**: When multiple threads access shared resources, synchronization mechanisms must be used to avoid data corruption or inconsistent states.

Tools and APIs for Concurrent Programming in Android:

1. Threads and Runnables

- **Threads** can be directly used for executing operations in the background. However, managing raw threads can be cumbersome and error-prone.
- **Runnables** are tasks that can be executed by a thread. They encapsulate the code that should execute on a thread.

```
new Thread(new Runnable() {
    @Override
    public void run() {
        // Code that will run in new Thread
    }
}).start();
```

2. AsyncTask (Deprecated)

- This was a popular choice for performing background operations and publishing results on the UI thread without having to manage threads manually. However, as of API level 30, `AsyncTask` is deprecated and Google recommends using modern alternatives like `java.util.concurrent` package classes or Kotlin coroutines.

3. Handlers and Looper

- **Handlers** are used to schedule messages and runnables to be executed at some point in the future. They are bound to the thread from which they are created and can help in communicating with the main thread from a background thread.
- **Looper** is used to create a message loop in a thread, allowing it to process messages until the loop is ended.

```
Handler mainHandler = new Handler(Looper.getMainLooper());
mainHandler.post(new Runnable() {
    @Override
    public void run() {
        // This will be executed on the main thread.
    }
});
```

4. Executors

- Provides a framework for managing a pool of threads and running tasks asynchronously. It abstracts the thread management, making it easier to execute tasks concurrently.

```
ExecutorService executor = Executors.newFixedThreadPool(5);
executor.execute(new Runnable() {
    @Override
    public void run() {
        // Background work
    }
});
```

5. Future and Callable

- **Callable** is similar to `Runnable` but can return a result and throw a checked exception. **Future** holds the result provided by a `Callable` and can be used to check if the operation is complete, wait for completion, and retrieve the result.

```
Future<Integer> future = executor.submit(new Callable<Integer>() {
    @Override
    public Integer call() throws Exception {
        return someComputation();
    }
});
```

Best Practices:

- **Avoid Blocking the UI Thread**: Keep the UI thread for UI operations only.

- **Use Appropriate Concurrency Constructs**: Choose the right tool based on the complexity and requirements of the task.
- **Handle Configuration Changes**: Manage threads and tasks properly during configuration changes (like device rotation) to avoid memory leaks or crashes.

Android provides a rich set of tools and frameworks to handle concurrency effectively. Developers should leverage these to enhance app responsiveness and maintain a smooth user experience.

Java

Generics

Generics are a feature in Java that allows type (classes and interfaces) to be parameters when defining classes, interfaces, and methods. Introduced in Java 5, generics enhance the type safety of your code by allowing you to enforce stricter type checks at compile time. This reduces bugs and errors in your code by catching invalid types at compile time, thereby avoiding potential ClassCastException at runtime.

Benefits of Using Generics

1. **Stronger Type Checks at Compile Time**: Generics enforce type safety by allowing you to specify the exact types your collections can contain, which prevents runtime errors.
2. **Elimination of Casts**: Casting is not needed when retrieving an object from a generic collection. The compiler automatically handles it for you, which makes the code cleaner and easier to read.
3. **Enabling Programmers to Implement Generic Algorithms**: By using generics, programmers can implement algorithms that work on collections of different types, customizable by the types of data they operate upon.

How Generics Work

Generics work with various types of Java elements, but they are most commonly used with classes, interfaces, and methods.

Generic Classes and Interfaces

A class or an interface is generic if it declares one or more type variables. These type variables are known as type parameters. A generic class is typically declared as follows:

```
class Box<T> {
    private T t; // T stands for "Type"

    public void set(T t) {
        this.t = t;
    }

    public T get() {
        return t;
    }
}
```

In this example, `T` is a type parameter that will be replaced by a real type when an object of `Box` is created.

Example Usage of Generic Class

```
Box<Integer> integerBox = new Box<>();
integerBox.set(10);
Integer someInteger = integerBox.get();

Box<String> stringBox = new Box<>();
stringBox.set("Hello World");
String someString = stringBox.get();
```

Generic Methods

Generic methods are methods that introduce their own type parameters. This is similar to declaring a generic type, but the type parameter's scope is limited to the method where it is declared. Such methods are especially useful for static methods because they cannot access the type parameters of the class they belong to.

```
public class Util {
    public static <T> void fill(List<T> list, T val) {
        for (int i = 0; i < list.size(); i++) {
            list.set(i, val);
        }
    }
}
```

Wildcards

Wildcards in generics are unknown type arguments when declaring a variable. They are useful in situations where you don’t know or care about the type specifics of the objects you need to work with. There are three types of wildcards:

- `?`: Represents an unknown type.
- `? extends Type`: Represents an unknown type that extends `Type` or implements `Type`.
- `? super Type`: Represents an unknown type that is a superclass of `Type` (inclusive of `Type` itself).

Example of Wildcards

```
public void printList(List<?> list) {
    for (Object elem : list) {
        System.out.println(elem); // elem is treated as an Object
    }
}
```

```
List<Integer> li = Arrays.asList(1, 2, 3);
printList(li);
```

Type Erasure

Generics were introduced to improve type safety, but the way Java implements generics does not create new classes for parameterized types; this is handled by a mechanism known as Type Erasure. Type Erasure ensures that no new classes are created for parameterized types; consequently, generics incur no runtime overhead. This process removes all information related to type parameters and type arguments within a class or method to ensure binary compatibility with Java libraries and applications that were created before generics.

Generics are a powerful feature in Java that provide stronger type safety and clearer code, but they require a good understanding to be used effectively. They are integral to modern Java programming, particularly in the collections framework.

Also can be bounded by extends or implements

```
Pair <T extends Comparable<T>, S extends Comparable<S>> implements
Comparable<Pair<T,S>>
```

Adapters

In the context of software design, an adapter is a structural design pattern that allows objects with incompatible interfaces to work together by converting the interface of one class into an interface expected by the clients. Adapters are often used to make existing classes work with others without modifying their source code.

Purpose and Use Cases

The primary purpose of the Adapter pattern is to achieve compatibility between different interfaces. Here are some common scenarios where adapters are used:

1. **Integration of Third-Party Libraries**: When integrating third-party libraries, often the library's interface may not align with the existing application code. An adapter can be used to bridge this gap without altering the original code.
2. **Legacy System Integration**: In systems where new components need to interact with old systems, adapters help unify the interface between the new and old systems without rewriting the legacy code.
3. **API Version Upgrades**: When an API evolves, adapters can be used to support applications that were built against older versions of the API, providing backward compatibility.

Types of Adapters

There are two common ways to implement the Adapter pattern:

1. **Class Adapter**: This implementation uses inheritance and extends the "adaptee" class. It inherits the interface of the adaptee while implementing the target interface. Here, the adapter

gets access to the adaptee's methods and properties.

2. **Object Adapter**: This type of adapter uses composition and holds an instance of the class it adapts. This is more flexible than class adapters as it can work with subclasses of the adaptee without inheritance.

Example in Java

Let’s illustrate the Adapter pattern with a simple example in Java. Suppose we have a legacy system that operates with `Rectangle` objects, but we need to integrate new functionality that works with `Circle` objects.

Step 1: Define the Target Interface

```java
public interface Circle {
    void drawCircle();
}
```

Step 2: Adaptee Class

```java
public class Rectangle {
    public void drawRectangle() {
        System.out.println("Drawing Rectangle");
    }
}
```

Step 3: Adapter Class

We'll use an object adapter approach here.

```java
public class CircleAdapter implements Circle {
    private Rectangle rectangle;

    public CircleAdapter(Rectangle rectangle) {
        this.rectangle = rectangle;
    }

    @Override
    public void drawCircle() {
        rectangle.drawRectangle();  // Adapting the interface
    }
}
```

Step 4: Usage

```java
public class AdapterDemo {
    public static void main(String[] args) {
        Rectangle rectangle = new Rectangle();
        Circle circle = new CircleAdapter(rectangle);
        circle.drawCircle();  // Internally draws a rectangle
    }
}
```

Advantages and Disadvantages

Advantages:

- **Flexibility**: Adapters provide flexibility by allowing existing classes to work with others without modifying their source code.
- **Reusability**: Facilitates the reuse of existing code even if their interfaces do not match the requirements of new systems.

Disadvantages:

- **Complexity**: Use of adapters can add complexity to the code, particularly if the number of interfaces to adapt increases.
- **Maintenance**: Overuse of adapters can lead to maintenance challenges as the project evolves, making the system architecture harder to understand.

In software development, adapters are a crucial design solution for maintaining and extending the functionality of applications as they evolve. They enable seamless interactions between components that would otherwise require significant changes to work together, preserving existing codebases and promoting modularity.