ry(), DNAME);

false);

ite

irectory and used that
l using the new directory

A NOTE ABOUT WRITING FILES

Android applications run inside a virtual machine environment. This has some effects to be aware of when working with certain aspects of the system, such as the filesystem. Java APIs like `FileOutputStream` do not share a 1:1 relationship with the native file descriptor inside the kernel. Typically, when data is written to the stream by using the `write()` method, that data is written directly into a memory buffer for the file and asynchronously written out to disk. In most cases, as long as your file access is strictly within the VM, you will never see this implementation detail. A file you just wrote could be opened and immediately read without issue, for example.

However, when dealing with removable storage such as an SD card on a mobile handset or tablet, we may often need to guarantee that the file data has made it all the way to the filesystem before returning an operation to the user, since the user has the ability to physically remove the storage medium. The following is a good standard code block to use when writing external files:

```
//Write the data
out.write();
//Clear the stream buffers
out.flush();
//Sync all data to the filesystem
out.getFD().sync();
//Close the stream
out.close();
```

The `flush()` method on an `OutputStream` is designed to ensure that all the data resident in the stream is written out of the VM's memory buffer. In the direct case of `FileOutputStream`, this method actually does nothing. However, in cases where that stream may be wrapped inside another (such as a `BufferedOutputStream`), this method can be essential in clearing out internal buffers, so it is a good habit to get into by calling it on every file write before closing the stream.

Additionally, with external files, we can issue a `sync()` to the underlying `FileDescriptor`. This method will block until all the data has been successfully written to the underlying filesystem, so it is the best indicator of when a user could safely remove physical storage media without file corruption.

External System Directories

There are additional methods in `Environment` and `Context` that provide standard locations on external storage where specific files can be written. Some of these locations have additional properties as well.

- `Environment.getExternalStoragePublicDirectory(String type)`
  - API Level 8
  - Returns a common directory where all applications store media files. The contents of these directories are visible to users and other applications. In particular, the media placed here will likely be scanned and inserted into the device's `MediaStore` for applications such as the Gallery.
  - Valid type values include `DIRECTORY_PICTURES`, `DIRECTORY_MUSIC`, `DIRECTORY_MOVIES`, and `DIRECTORY_RINGTONES`.

- `Context.getExternalFilesDir(String type)`
  - API Level 8
  - Returns a directory on external storage for media files that are specific to the application. Media placed here will not be considered public, however, and won't show up in `MediaStore`.
  - This is external storage, however, so it is still possible for users and other applications to see and edit the files directly: there is no security enforced.
  - Files placed here will be removed when the application is uninstalled, so it can be a good location in which to place large content files the application needs that one may not want on internal storage.
  - Valid type values include `DIRECTORY_PICTURES`, `DIRECTORY_MUSIC`, `DIRECTORY_MOVIES`, and `DIRECTORY_RINGTONES`.
- `Context.getExternalCacheDir()`
  - API Level 8
  - Returns a directory on internal storage for app-specific temporary files. The contents of this directory are visible to users and other applications.
  - Files placed here will be removed when the application is uninstalled, so it can be a good location in which to place large content files the application needs that one may not want on internal storage.
- `Context.getExternalFilesDirs()` and `Context.getExternalCacheDirs()`
  - API Level 19
  - Identical features as their counterparts described previously, but returns a list of paths for each storage volume on the device (primary and any secondary volumes)
  - For example, a single device may have a block of internal flash for primary external storage, and a removable SD card for secondary external storage.
- `Context.getExternalMediaDirs()`
  - API Level 21
  - Files placed in these volumes will be automatically scanned and added to the device's media store to expose them to other applications. These will generally also be visible to the user through core applications like the Gallery.

■ **Note** As of KitKat (API Level 19), permissions are no longer required to read and write the directory paths returned by `getExternalFilesDir()` and `getExternalCacheDir()` for your application. Primary volumes are still writable outside these directories with the aforementioned permissions. Secondary volumes (also new to the KitKat APIs) are fully write-protected outside these directories, even if the `WRITE_EXTERNAL_STORAGE` permission is granted.

5-5. Using Files as Resources

Problem

Your application must utilize resource files that are in a format Android cannot compile into a resource ID.

Solution

(API Level 1)

Use the `assets` directory to house files your application needs to read from, such as local HTML, comma-separated values (CSV), or proprietary data. The `assets` directory is a protected resource location for files in an Android application. The files placed in this directory will be bundled with the final APK but will not be processed or compiled. Like all other application resources, the files in `assets` are read-only.

How It Works

There are a few specific instances that we've seen already in this book, where `assets` can be used to load content directly into widgets, such as `WebView` and `MediaPlayer`. However, in most cases, `assets` is best accessed through a traditional `InputStream`. Listings 5-17 and 5-18 provide an example in which a private CSV file is read from `assets` and displayed onscreen.

Listing 5-17. `assets/data.csv`

```
John,38,Red
Sally,42,Blue
Rudy,31,Yellow
```

← set up your text file, save it in assets

Listing 5-18. Reading from an Asset File

```
public class AssetActivity extends Activity {

    @Override
    public void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        TextView tv = new TextView(this);
        setContentView(tv);
        try {
            //Access application assets
            AssetManager manager = getAssets();
            //Open our data file
            InputStream in = manager.open("data.csv");

            //Parse the CSV data and display
            ArrayList<Person> cooked = parse(in);
            StringBuilder builder = new StringBuilder();
            for(Person piece : cooked) {
                builder.append(String.format("%s is %s years old, and likes the color %s",
                        piece.name, piece.age, piece.color));
                builder.append('\n');
            }
```

start try

end try

Similar to our code

slight changes of syntax

← Part A goes here. (it is in a method in this code)

```
            tv.setText(builder.toString());
        } catch (FileNotFoundException e) {
            e.printStackTrace();
        } catch (IOException e) {
            e.printStackTrace();
        }

    }

    /* Simple CSV Parser */
    private static final int COL_NAME = 0;
    private static final int COL_AGE = 1;
    private static final int COL_COLOR = 2;

    private ArrayList<Person> parse(InputStream in) throws IOException {
        ArrayList<Person> results = new ArrayList<Person>();

        BufferedReader reader = new BufferedReader(new InputStreamReader(in));
        String nextLine = null;
        while ((nextLine = reader.readLine()) != null) {
            String[] tokens = nextLine.split(",");
            if (tokens.length != 3) {
                Log.w("CSVParser", "Skipping Bad CSV Row");
                continue;
            }

            //Add new parsed result
            Person current = new Person();
            current.name = tokens[COL_NAME];
            current.color = tokens[COL_COLOR];
            current.age = tokens[COL_AGE];

            results.add(current);
        }

        in.close();

        return results;
    }

    private class Person {
        public String name;
        public String age;
        public String color;

        public Person() { }
    }
}
```

The key to accessing files in `assets` lies in using `AssetManager`, which will allow the application to open any resource currently residing in the `assets` directory. Passing the name of the file we are interested in to `AssetManager.open()` returns an `InputStream` for us to read the file data. Once the stream is read into memory, the example passes the raw data off to a parsing routine and displays the results to the user interface.

Parsing the CSV

This example also illustrates a simple method of taking data from a CSV file and parsing it into a model object (called `Person` in this case). The method used here takes the entire file and reads it into a byte array for processing as a single string. This method is not the most memory efficient when the amount of data to be read is quite large, but for small files like this one it works just fine.

The raw string is passed into a `StringTokenizer` instance, along with the required characters to use as breakpoints for the tokens: comma and new line. At this point, each individual chunk of the file can be processed in order. Using a basic state machine approach, the data from each line is inserted into new `Person` instances and loaded into the resulting list.

5-6. Managing a Database

Problem

Your application needs to persist data that can later be queried or modified as subsets or individual records.

Solution

(API Level 1)

Create a `SQLiteDatabase` with the assistance of an `SQLiteOpenHelper` to manage your data store. SQLite is a fast and lightweight database technology that utilizes SQL syntax to build queries and manage data. Support for SQLite is baked into the Android SDK, making it very easy to set up and use in your applications.

How It Works

Customizing `SQLiteOpenHelper` allows you to manage the creation and modification of the database schema itself. It is also an excellent place to insert any initial or default values you may want in the database when it is created. Listing 5-19 is an example of how to customize the helper in order to create a database with a single table that stores basic information about people.

Listing 5-19. Custom `SQLiteOpenHelper`

```
public class MyDbHelper extends SQLiteOpenHelper {

    private static final String DB_NAME = "mydb";
    private static final int DB_VERSION = 1;

    public static final String TABLE_NAME = "people";
    public static final String COL_NAME = "pName";
    public static final String COL_DATE = "pDate";
```