

Lesson 2.2: Data Persistence

ACTIVITY 2.2.1

Exceptions and Scope

INTRODUCTION

In this lesson, you will learn the Java® tools you need to gather and maintain persistent data within the College App. You will also create Android™ ListView components to handle data.

Before you can do these things, it will be helpful for you to further your understanding of what exceptions are and how they are used in Java. As you first learned in Lesson 1.2, an exception is a predefined error condition for Java programs.

Another skillset you will develop in this lesson is working with variable scope. **Scope** is the property of a variable that deals with what parts of a program that variable is accessible from.

scope

The accessibility level of a variable that determines where it can be referenced and used within a program.

Materials

- Computer with Android™ Studio
- Android™ Tablet and USB cable, or device emulator

RESOURCES



Activity 2.2.1 Visual Aid

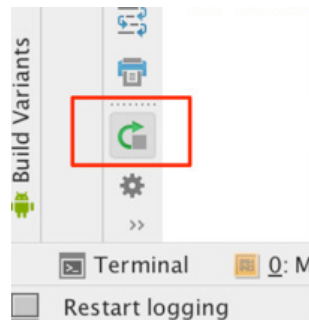
Resources available online

Procedure

Part I: Exceptions

- 1 In Android Studio, open your College App from *Activity 2.1.5 User Input*. If you were unable to finish the activity, open *2.1.5CollegeApp_Solution* as directed by your teacher.

- 2 Open the **Android Monitor** at the bottom left of Android Studio. Over the course of this activity, if you are not seeing the messages that you expect, try to restart logging using the Restart button shown below. (If you do not see the Restart button at first, make the monitor window taller by dragging its top bar higher. Or, click >> to see additional tools.)



- 3 Review the slideshow.



Activity 2.2.1 Exceptions and Scope

Computer Science A

© 2016 Project Lead The Way, Inc.

Exceptions are errors that cause a program to crash unless they are handled appropriately by the programmer. Examples of some of the most common exceptions are found on the next slide.

To find out more about exceptions in Java: <https://docs.oracle.com/javase/tutorial/essential/exceptions/definition.html>

Exceptions

Cause a program to crash...
...unless handled appropriately

Two types:

- Checked
- Unchecked

Computer Science A

© 2016 Project Lead The Way, Inc.

Programs must handle or respond to “checked” exceptions. Why? Problems that occur outside of a programmer’s control can cause apps to crash. Most often, these problems relate to input/output:

These types of exceptions should not crash the app; instead, your app should handle the error gracefully. Java requires your app to handle checked exceptions to ensure that problems outside of your control will not crash your app.

To find out more about exceptions in Java: <https://docs.oracle.com/javase/tutorial/essential/exceptions/definition.html>

Checked Exceptions

Examples:

An app tries to open a file that cannot be found or does not exist

An app tries to access a network resource that is unavailable, such as a web server that has gone offline

Computer Science A

© 2016 Project Lead The Way, Inc.

Programs must handle or respond to “checked” exceptions. Why? Problems that occur outside of a programmer’s control can cause apps to crash. Most often, these problems relate to input/output:

These types of exceptions should not crash the app; instead, your app should handle the error gracefully. Java requires your app to handle checked exceptions to ensure that problems outside of your control will not crash your app.

To find out more about exceptions in Java: <https://docs.oracle.com/javase/tutorial/essential/exceptions/definition.html>

Unchecked Exceptions

Examples:

Exception

`NullPointerException`

Programatic solution

```
if (<yourObject> != null) {...}
```

Exception

`ArrayIndexOutOfBoundsException`

Programatic solution

```
If (index < lengthOfArray) {...}
```

Computer Science A

© 2016 Project Lead The Way, Inc.

Programs are not required to “handle” unchecked exceptions since they do not usually occur and can be avoided with good programming practices.

<https://docs.oracle.com/javase/tutorial/essential/exceptions/definition.html>

Common Unchecked Exceptions

ArithmeticException

NullPointerException

ArrayIndexOutOfBoundsException

IndexOutOfBoundsException

IllegalArgumentException

Computer Science A

© 2016 Project Lead The Way, Inc.

The predefined error conditions above are unchecked exceptions and are part of the AP subset. There are many other unchecked exceptions. All unchecked exceptions are derived from the RuntimeException class.

You will learn more about each of these as you develop the College App, when they are relevant to the kind of code that you are working on. In this activity, you will artificially trigger the Arithmetic and NullPointerExceptions to gain familiarity with the conditions under which they occur.

Checked exceptions are exceptions that provide code to allow your program to gracefully fail under given conditions. All checked exceptions are derived from the Exception class. These are not part of the AP subset.

To find out more about exceptions in Java: <https://docs.oracle.com/javase/tutorial/essential/exceptions/definition.html>

ArithmeticException

Typically the result of a division by zero.

```
public Profile() {  
    mFirstName = new String("Wyatt");  
    mLastName = new String("Dumas");  
    mDateOfBirth = new Date(83, 0, 24/0);  
}  
IMS  
Devices logcat →  
Caused by: java.lang.ArithmeticException: divide by zero  
    at org.pltw.examples.collegeapp.Profile.<init>(Profile.java:55)
```

Computer Science A

© 2016 Project Lead The Way, Inc.

An Arithmetic Exception usually happens not because you intentionally add a division by zero to your program, but because some variable has acquired the value zero through a series of operations that were not checked closely enough.

The highlighted line shows code from Line 55 of the Profile class that has a divide by zero error. This will cause an Arithmetic Exception.

NullPointerException

The program is trying to use an instance of a class, but that instance does not exist.

```
rate class DoBButtonOnClickListener implements View.OnClickListener {
    public void onClick(View v) {
        FragmentManager fm = null; /* getActivity()
        .getSupportFragmentManager(); */
        DoBPickerFragment dialog = DoBPickerFragment
            .newInstance(mProfile.getDateOfBirth());
        dialog.setTargetFragment(ProfileFragment.this, REQUEST_DOB);
        dialog.show(fm, DIALOG_DATE);
    }
}

java.lang.NullPointerException
    at android.support.v4.app.DialogFragment.show(DialogFragment.java:136)
    at org.pltw.examples.collegeapp.ProfileFragment$DoBButtonOnClickListener.onClick(ProfileFragment.java:144)
```

Computer Science A © 2016 Project Lead The Way, Inc.

A `NullPointerException` will usually occur when you attempt to call a method of a class from an instance of that class that has yet to be initialized.

These may also occur when you pass a non-initialized instance of a class to another method that then tries to invoke one of that argument's methods. The latter is shown above. The `show(FragmentManager, int)` method of the `dialog` class attempts to use the argument "fm" only to find that it is null.

Line 144 in `ProfileFragment` is highlighted.

Other Exceptions

Many other exceptions must be included in method signatures to be used.

```
public class FamilyJSONSerializer {
    private static final String TAG = "FamilyJSONSerializer";
    protected Context mContext;
    public ArrayList<FamilyMember> loadFamily() throws IOException, JSONException {
        BufferedReader reader = null;
        ArrayList<FamilyMember> familyList = new ArrayList<FamilyMember>();
        try {
            Log.d(TAG, "Opening an input stream from: " + mFilename + " with Context:" + mContext);
            InputStream in = mContext.openFileInput(mFilename);
            reader = new BufferedReader(new InputStreamReader(in));
            StringBuilder jsonString = new StringBuilder();
            String line = null;
            while ((line = reader.readLine()) != null) {
                jsonString.append(line);
            }

            JSONArray array = (JSONArray) new JSONTokener(jsonString.toString()).nextValue();
            for (int i = 0; i < array.length(); i++) {
                JSONObject temp = array.getJSONObject(i);
            }
        } catch (FileNotFoundException e) {
            Log.e(TAG, "Error loading family from: " + mFilename, e);
        } finally {
            if (reader != null)

```

Computer Science A © 2016 Project Lead The Way, Inc.

Shown is an excerpt of code from a class you will see later in this lesson. The class has the job of converting `FamilyMember` data into a format that can be stored and retrieved when the `CollegeApp` is paused or terminated, or when switching between `Fragments`.

The method `loadFamily` makes use of `IOException`, `JSONException`, and `FileNotFoundException`, all of which must be imported. All of these exceptions are checked exceptions. You must account for your checked exceptions in one of two ways. Either append the keyword "throws" followed by a comma delimited (separated) list of exceptions to the method signature, or add a "catch" block to the method definition.

Note that "try", "catch", "finally", and "throws" are not tested on the AP exam, but you will see them frequently in Android source code.

Checked Exceptions: try, catch, throws, and finally

try, catch, and finally are all part of one block sequence, much like if, else if, and else.

Either enclose the code that may throw an exception in a try block followed by a catch block that handles that exception

OR

Add throws to the method signature followed by a list of exceptions.

Computer Science A

© 2016 Project Lead The Way, Inc.

Though these keywords are not tested on the AP exam, having some understanding of what they do will help you read Android source code.

If a method explicitly catches an exception with a catch statement, then that exception does not need to be declared to be thrown in the method signature.

If a method does not explicitly catch a checked exception and that method may produce that exception, then it must declare that exception to be thrown in its signature.

Advanced:

The finally key word will not show up as often as the others and is included here as an advanced topic. Statements enclosed within a “finally” block execute regardless of what happens within the “try” and “catch” blocks.

Checked Exceptions: try, catch, throws, and finally

```
private void sendResult(int resultCode) throws NumberOutOfRangeException {  
    if (getTargetFragment() == null) return;  
  
    if (mDoB.getYear() <= WITHIN_8_YEARS) {  
        String message = NumberOutOfRangeException.joinMessageAndYear(  
            "Who are you, Michael Kearney?", mDoB.getYear());  
        throw new NumberOutOfRangeException(message);  
    }  
}
```

```
private void sendResult(int resultCode) {  
    if (getTargetFragment() == null) return;  
    try {  
        if (mDoB.getYear() <= WITHIN_8_YEARS) {  
            String message = NumberOutOfRangeException.joinMessageAndYear(  
                "Who are you, Michael Kearney?", mDoB.getYear());  
            throw new NumberOutOfRangeException(message);  
        }  
    } catch (NumberOutOfRangeException s) {  
        Log.e(TAG, s.getMessage());  
    }  
}
```

Computer Science A

© 2016 Project Lead The Way, Inc.

(Accompanies Step 14)

These are two examples of handling the same exception.

The method will throw the exception up to whatever method called it in the stack trace. Note that in this case, the calling method must either catch the exception or declare it to be thrown and so on and so on.

In the example on the bottom, the method handles the exception internally using a try catch block, and so the method does not need to declare it to be thrown.

The code that is different is enclosed in red boxes.

- 4 After you observe the exceptions in each step of this section, restore your code to the original version.
- 5 Find an integer value anywhere in your code and append /0 to the end of it. Run your app to verify that you have created an `ArithmeticException`.

NOTE

Note that you will have to interact with the app to make the exception happen, because the part of the code where you put the 0 may not run automatically upon startup. For example, if you put a divide by 0 in the code for the `ProfileFragment` class, then open the Profile menu in the app to cause that fragment to load and the code to run.

- 6 Restore the original code after you have observed the exception, *and remember to do so after each of the steps in this section.*
- 7 Find an initialization statement anywhere in your code, other than for a primitive type, and comment out the right side of the = operator. When temporarily removing code, it is best to comment it out so that you can return it to normal afterward.
- 8 Set the value of the right-hand side to `null` and remember to include a new terminating semicolon.
- 9 Run your app to verify that this causes a `NullPointerException`. You might not get the expected exception if the variable you set to `null` is never used; either make the app run that section of code, or be sure to choose a variable that will be used.

The `NullPointerException` will not occur on the same line as this initialization statement. Why?

- 10 Now you will create your own exception to check whether the date of birth that the user selects is reasonable. In `ProfileFragment.java`, create a constant named `WITHIN_8_YEARS` that contains the year that is eight years before the present year. For example, if the current year is 2016, then the value stored in `WITHIN_8_YEARS` should be 2008.
- 11 Create a new class called `AgeException`. Add the following code beneath the package declaration:

```
1: public class AgeException extends RuntimeException {
2:     public AgeException(String message){
3:         super(message);
4:     }
5: }
```

- 12 Add a method, `joinMessageAndYear`, to this class that will help client classes convert the parameters, a `String` named `message` and an `int` named `year`, into a single `String` containing the two parameters separated by a space. For example, if you pass

`joinMessageAndYear` "Who are you, Michael Kearney?" and the `int` 2008, the return value should be the String "Who are you, Michael Kearney? 2008".

To help you debug your app, you will create a TAG that you will use when you log any kind of information to the Android Monitor. This will help you track which class, and therefore which Java file, logged the information.

- 13 In *ProfileFragment.java*, create a constant named TAG with a value of "ProfileFragment".

You will check the value of your applicant's date of birth. The best place for this addition is in the Presenter layer; the `ProfileFragment` method that sets the date of birth.

- 14 To do this, you will add a `try` block around the code that handles updating the Presenter layer with date-of-birth information, followed by a `catch` block that specifies your new `AgeException` and triggers a Log message. If a user enters an abnormal date, you should trigger the log message and not update the Model or View layers:

```
1: try {
2:     if (date.getYear() + 1900 <= WITHIN_8_YEARS) {
3:         // change the model and view
4:     }
5:     else throw new AgeException("Who are you, Michael Kearney?");
6: } catch (AgeException e) {
7:     Log.i(TAG, e.joinMessageAndYear(e.getMessage(), date.
        getYear() + 1900));
8: }
```

- 15 Test your application. Now, if you attempt to set the date of birth to indicate a person younger than the age of 8, you should see a log message appear and the button text should remain unchanged.

CHALLENGE: Instead of using a constant for the conditional check, create a method within `AgeException` that automates the comparison of the date entered versus the current date.

The result should be a program that still outputs the correct exception a year from now with no need to change the source code. This method should return a boolean value and be called directly within the conditional that created in your `joinMessageAndYear` method.

Part II: Scope

Scope is the property of a variable that deals with what parts of a program that variable is accessible from. Review the slideshow.

scope

The accessibility level of a variable that determines where it can be referenced and used within a program.

Scope

Class level scope

- Instance variables
- Static variables

Method level scope

- Parameters
- Local variables

Loop scope and block scope

Computer Science A

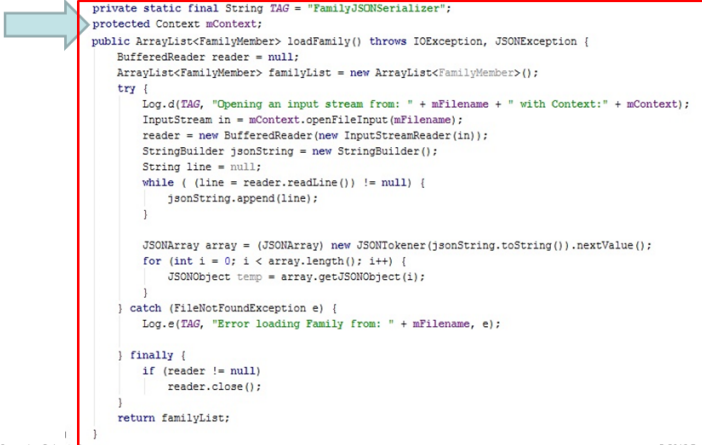
© 2016 Project Lead The Way, Inc.

Scope defines in which parts of a program a variable can be used.

Generally speaking, a variable can be used in the scope indicated by the block in which it is defined, as well as any blocks contained within that block.

Instance Variables

Can be used anywhere within the class definition.



```
public class FamilyJSONSerializer {
    private static final String TAG = "FamilyJSONSerializer";
    protected Context mContext;
    public ArrayList<FamilyMember> loadFamily() throws IOException, JSONException {
        BufferedReader reader = null;
        ArrayList<FamilyMember> familyList = new ArrayList<FamilyMember>();
        try {
            Log.d(TAG, "Opening an input stream from: " + mFilename + " with Context:" + mContext);
            InputStream in = mContext.openFileInput(mFilename);
            reader = new BufferedReader(new InputStreamReader(in));
            StringBuilder jsonString = new StringBuilder();
            String line = null;
            while ( (line = reader.readLine()) != null) {
                jsonString.append(line);
            }

            JSONArray array = (JSONArray) new JSONTokener(jsonString.toString()).nextValue();
            for (int i = 0; i < array.length(); i++) {
                JSONObject temp = array.getJSONObject(i);
            }
        } catch (FileNotFoundException e) {
            Log.e(TAG, "Error loading Family from: " + mFilename, e);
        } finally {
            if (reader != null)
                reader.close();
        }
        return familyList;
    }
}
```

Computer Science A

© 2016 Project Lead The Way, Inc.

Instance variables, also known as instance fields and class variables, can be used anywhere within the scope of the class in which they are declared.

Here, mContext is an instance variable of the FamilyJSONSerializer class. Because it is declared at the class level, it can be used throughout the entire class.


Notes:

A variable cannot be used before it is declared.

Though Android developers use the convention of an “m” prefix in naming instance variables to denote membership to that class, other Java developers do not.

Static Class Variables

Can be used anywhere within the class definition.



```
public class FamilyJSONSerializer {
    private static final String TAG = "FamilyJSONSerializer";
    protected Context mContext;

    public ArrayList<FamilyMember> loadFamily() throws IOException, JSONException {
        BufferedReader reader = null;
        ArrayList<FamilyMember> familyList = new ArrayList<FamilyMember>();
        try {
            Log.d(TAG, "Opening an input stream from: " + mFilename + " with Context:" + mContext);
            InputStream in = mContext.openFileInput(mFilename);
            reader = new BufferedReader(new InputStreamReader(in));
            StringBuilder jsonString = new StringBuilder();
            String line = null;
            while ( (line = reader.readLine()) != null) {
                jsonString.append(line);
            }

            JSONArray array = (JSONArray) new JSONTokener(jsonString.toString()).nextValue();
            for (int i = 0; i < array.length(); i++) {
                JSONObject temp = array.getJSONObject(i);
            }
        } catch (FileNotFoundException e) {
            Log.e(TAG, "Error loading Family from: " + mFilename, e);
        } finally {
            if (reader != null)
                reader.close();
        }
        return familyList;
    }
}
```

Computer Science A


© 2016 Project Lead The Way, Inc.

Static class variables can be used anywhere within the scope of the class in which they are declared.

Here, TAG is an instance variable of the FamilyJSONSerializer class. Because it is declared at the class level, it can be used throughout the entire class.

Method Variables

Can be used anywhere within the class definition.



```
public class FamilyJSONSerializer {
    private static final String TAG = "FamilyJSONSerializer";
    protected Context mContext;

    public ArrayList<FamilyMember> loadFamily() throws IOException, JSONException {
        BufferedReader reader = null;
        ArrayList<FamilyMember> familyList = new ArrayList<FamilyMember>();
        try {
            Log.d(TAG, "Opening an input stream from: " + mFilename + " with Context:" + mContext);
            InputStream in = mContext.openFileInput(mFilename);
            reader = new BufferedReader(new InputStreamReader(in));
            StringBuilder jsonString = new StringBuilder();
            String line = null;
            while ( (line = reader.readLine()) != null) {
                jsonString.append(line);
            }

            JSONArray array = (JSONArray) new JSONTokener(jsonString.toString()).nextValue();
            for (int i = 0; i < array.length(); i++) {
                JSONObject temp = array.getJSONObject(i);
            }
        } catch (FileNotFoundException e) {
            Log.e(TAG, "Error loading Family from: " + mFilename, e);
        } finally {
            if (reader != null)
                reader.close();
        }
        return familyList;
    }
}
```

Computer Science A

© 2016 Project Lead The Way, Inc.

If a variable is declared within a method, it may only be used within that method.

Here, the variable familyList is of type ArrayList<FamilyMember> and is declared and usable within the loadFamily method.

Variables Initialized in a Loop or Block

Can be used only within the block for which they are defined.

```
public class FamilyJSONSerializer {
    private static final String TAG = "FamilyJSONSerializer";
    protected Context mContext;

    public ArrayList<FamilyMember> loadFamily() throws IOException, JSONException {
        BufferedReader reader = null;
        ArrayList<FamilyMember> familyList = new ArrayList<FamilyMember>();
        try {
            Log.d(TAG, "Opening an input stream from: " + mFilename + " with Context:" + mContext);
            InputStream in = mContext.openFileInput(mFilename);
            reader = new BufferedReader(new InputStreamReader(in));
            StringBuilder jsonString = new StringBuilder();
            String line = null;
            while ( (line = reader.readLine()) != null) {
                jsonString.append(line);
            }

            JSONArray array = (JSONArray) new JSONTokener(jsonString.toString()).nextValue();
            for (int i = 0; i < array.length(); i++) {
                JSONObject temp = array.getJSONObject(i);
            }

        } catch (FileNotFoundException e) {
            Log.e(TAG, "Error loading Family from: " + mFilename, e);
        } finally {
            if (reader != null)
                reader.close();
        }
        return familyList;
    }
}
```

Computer Science A

© 2016 Project Lead The Way, Inc.

Especially in for loops, it is common to declare and initialize a variable whose only purpose is to store data within that loop. Variables like `i` shown here can only be used in the loop within which they are declared and cease to be accessible to the program once the loop has terminated.

- 16** For this part of the activity, you may reference and even modify your own code to determine your answers.
- In the `AgeException` class shown in step 10, where could you insert a statement referencing the `message` variable?
 - If you were to add a third method to the `AgeException` class with no parameters, would you be able to use the `year` parameter from `joinMessageAndYear`? Why or why not?
 - Name at least two variables in `FamilyMember` that could be used in the `getFirstName` method.
 - In the method definition below, which variable(s) has/have the most limited scope? Justify your answer.

```

1: public Profile load() throws IOException {
2:     BufferedReader reader = null;
3:     Profile profile = null;
4:     try {
5:         Log.d(TAG, "Opening an input stream from: " +
            mFilename + " with Context:" + mContext);
6:         InputStream in = mContext.openFileInput(mFilename);
7:         reader = new BufferedReader(new InputStreamReader(in));
8:         StringBuilder myString = new StringBuilder();
9:         String line = null;
10:        while ((line = reader.readLine()) != null) {
11:            myString.append(line);
12:        }
13:
14:        profile = new Profile(new StringObject(myString.
            toString()));
15:    } catch (FileNotFoundException e) {
16:        Log.e(TAG, "Error loading Profile from: " +
            mFilename, e);
17:
18:    } finally {
19:        if (reader != null)
20:            reader.close();
21:    }
22:    return profile;
23: }

```

CONCLUSION

1. In your own words, explain the most important things to remember about scope.
2. In Part I, you created an exception. It allowed you to detect when a user had entered a date of birth that might not be realistic for a college applicant. What might you want to modify your program to do when this exception is thrown?
3. In your opinion, when are exceptions the most useful?
4. Based on your experience in this activity, how has your definition of what makes for a successful piece of software changed?

Activity 2.2.1 Visual Aid

