Programming Fundamentals for Android

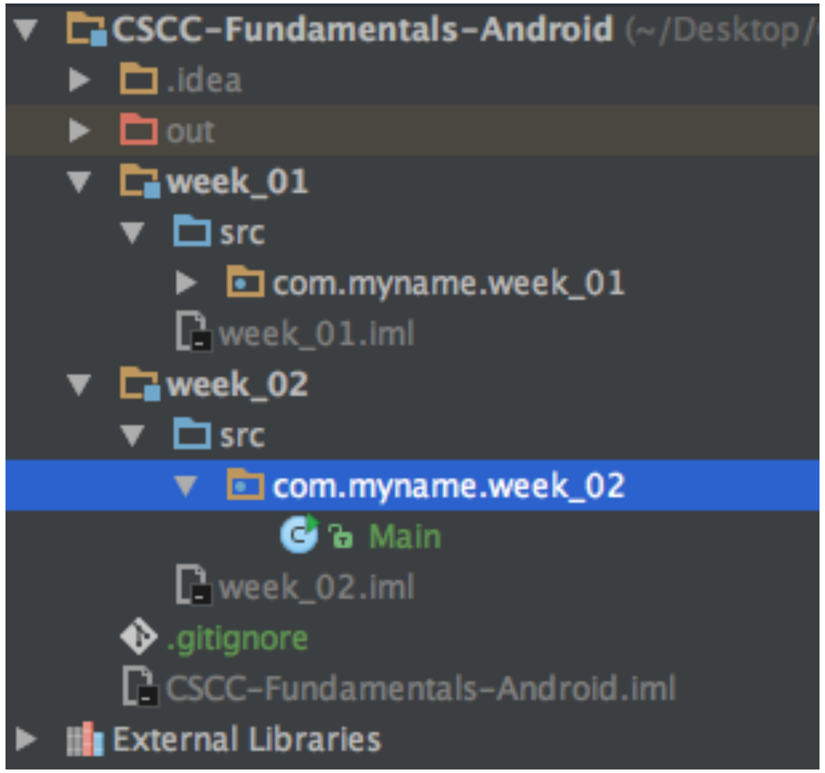
Week 2: Flow Control

January 28, 2016

Corresponding Text

- *Learn Java for Android Development*, pp. 71-84

Adding to our repository



```
▼ CSCC-Fundamentals-Android (~/Desktop/
    ▶ .idea
    ▶ out
    ▼ week_01
        ▼ src
            ▶ com.myname.week_01
          week_01.iml
    ▼ week_02
        ▼ src
            ▼ com.myname.week_02
                  Main
          week_02.iml
      .gitignore
      CSCC-Fundamentals-Android.iml
▶ External Libraries
```

- Want to keep last week’s code
- Create a new module within the existing project for this week’s code
- Create a package and Java class file for the new module
- Update run configurations to easily run our new code

Boolean Operators

Boolean operators can be used to make compound expressions with one or more Boolean operands.

Operator	Symbol	Description
Conditional AND	&&	*operand1* && *operand2*: true when both operands are true; if operand1 is false, operand2 isn't evaluated
Conditional OR	||	*operand1* || *operand2*: true when either operand is true; if operand1 is true, operand2 isn't evaluated
Logical AND	&	*operand1* & *operand2*: true when both operands are true
Logical complement	!	!*operand*: opposite Boolean value of operand
Logical exclusive OR	^	*operand1* ^ *operand2*: true when only one operand is true
Logical inclusive OR	|	*operand1* | *operand2*: true when at least one operand is true

Decision Statements

Decision statements can be used to choose which of several sets of statements is executed based on the evaluation of a Boolean expression.

If and If-Else Statements

- If statement evaluates a Boolean expression and executes another statement if the Boolean expression is true.
- Boolean expression can be a simple expression or a compound expression using Boolean operators.
- We can use if-else statements to execute one statement when the Boolean expression is true and another when its false.
- We can use the conditional operator, ?:, to write simple if-else statements

```
if (Boolean expression)
    statement
```

```
if (Boolean expression)
    statement
else
    statement
```

```
Boolean expression ?
    value_if_true : value_if_false
```

Switch Statements

- We can choose from among several different execution paths based on the value of a variable.
- Specify different paths with *case*.
- Use *break* after each case to avoid unexpected execution.

```
switch (selector expression)
{
    case value1: statement1 [break;]
    case value2: statement2 [break;]
    ...
    case valueN: statementN [break;]
    [default: statement]
}
```

Loops

Loops allow us to execute a statement (or set of statements) multiple times.

For Loops

- Let us loop over a statement a specific number of times or indefinitely
- Three parts: initialize, test, update
- Initialize: comma-separated list of variable declarations or assignments; some variables are loop-control variables
- Test: Boolean expression, loop continues as long as this is true
- Update: comma-separated list of expressions evaluated after each iteration

```
for ([initialize]; [test]; [update])
    statement
```

For Each Loops

- Can be used when you want to iterate over elements in a collection
- Often easier to read than a standard for loop

```
for (item: collection)
    statement
```

While Loops

- Repeatedly executes a statement as long as the Boolean expression evaluates to true

```
while (Boolean expression)
    statement
```

Do-While Loops

- Similar to while loops but always execute the statement at least once – before evaluation of the Boolean expression

```
do
    statement
while (Boolean expression);
```

Break Statements

- The **break** statement terminates execution of a loop or evaluation of a switch statement and transfers execution to the first statement following the loop or switch statement.
- Can be used to avoid computation of unnecessary loop iterations or switch cases.
- Common example is searching for a specific value.

Labeled Break Statements

- A **labeled break** statement transfers execution to the first statement after a loop that has been prefixed by a label.
- A label is an identifier followed by a colon, e.g. “*label*:”.

Continue Statements

- The **continue** statement skips the remainder of a loop’s current iteration, re-evaluates the loop’s Boolean expression, and perform the next iteration of the Boolean expression is true or terminates the loop.

Labeled Continue Statements

- A **labeled continue** statement skips the remaining iterations of one or more nested loops and transfers execution to the loop prefixed with a a label.
- Example with positive difference of two numbers.

First Number	Second Number	Difference	Outcome
1	1	Not Positive	Don't display this and move on to the next value for both first and second numbers
2	1	Positive	Display this and move on to the next value for the second number
2	2	Not Positive	Don't display this and move on to the next value for both first and second numbers
3	1	Positive	Display this and move on to the next value for the second number
3	2	Positive	Display this and move on to the next value for the second number
3	3	Not Positive	Don't display this and move on to the next value for both first and second numbers
4	1	Positive	Display this and move on to the next value for the second number

Scope

- A variable’s **scope** is where, in code, the variable exists and is accessible.
- All variables aren’t accessible from any part of a program.

Exercise

Suppose the high temperature (in degrees Fahrenheit) for each of next week's days are 45, 29, 10, 22, 41, 28, and 33 and the probability of precipitation for each of the next five days is 95%, 60%, 25%, 5%, 0%, 75%, and 90%. Write a program using a loop that displays the day of the week if that day's high temperature is less than or equal to 32 and the probability of precipitation is greater than 50% (meaning it's likely to snow).