SWEN 6301 Software Construction

Module 7: Statements

Ahmed Tamrawi

Organizing Straight-Line Code

Outline

- Statements That Must Be in a Specific Order
- Statements Whose Order Doesn’t Matter

Statements That Must Be in a Specific Order

- The easiest sequential statements to order are those in which the order counts.
- Here's an example:

Java Example of Statements in Which Order Counts

```
data = ReadData();
results = CalculateResultsFromData( data );
PrintResults( results );
```

Statements That Must Be in a Specific Order

- The statement must be executed in the order shown:
  - The data must be read before the results can be calculated
  - The results must be calculated before they can be printed.

Java Example of Statements in Which Order Counts

```
data = ReadData();
results = CalculateResultsFromData( data );
PrintResults( results );
```

Statements That Must Be in a Specific Order

- The underlying concept in this example is that of dependencies.
  - *The third statement depends on the second*
  - *The second statements depends on the first*
- In this example, the fact that one statement depends on another is obvious from the routine names.

Java Example of Statements in Which Order Counts

```
data = ReadData();
results = CalculateResultsFromData( data );
PrintResults( results );
```

Statements That Must Be in a Specific Order

- The quarterly revenue calculation assumes that the monthly revenues have already been calculated. The same for quarterly and annual.
- What about these statements?

```
revenue.ComputeMonthly();
revenue.ComputeQuarterly();
revenue.ComputeAnnual();
```

- A familiarity with accounting—or even common sense—might tell you that quarterly revenues have to be calculated before annual revenues.
- There is a **dependency**, but it's not obvious merely from reading the code.

Statements That Must Be in a Specific Order

- The dependencies aren't obvious - they're literally hidden:

```
ComputeMarketingExpense
ComputeSalesExpense
ComputeTravelExpense
ComputePersonnelExpense
DisplayExpenseSummary
```

- Suppose that *ComputeMarketingExpense()* initializes the class member variables that all the other routines put their data into. In such a case, it needs to be called before the other routines. *How could you know that?*
- Because the routine calls don't have any parameters, you might be able to guess that each of these routines accesses class data. But you can't know for sure from reading this code.

Statements That Must Be in a Specific Order

Organize code so that dependencies are obvious

- In the earlier example, *ComputeMarketingExpense()* shouldn't initialize the class member variables.
- The routine names suggest that *ComputeMarketingExpense()* is similar to *ComputeSalesExpense(), ComputeTravelExpense(),* and the other routines except that it works with marketing data rather than with sales data or other data.
- Having *ComputeMarketingExpense()* initialize the member variable is an arbitrary practice you should avoid.
- Why should initialization be done in that routine instead of one of the other two? Unless you can think of a good reason, you should write another routine, *InitializeExpenseData(),* to initialize the member variable. The routine's name is a clear indication that it should be called before the other expense routines.
  - Also refer to **COHESION**

Statements That Must Be in a Specific Order

Name routines so that dependencies are obvious

- *ComputeMarketingExpense()* is misnamed since it does more than compute marketing expenses; it also initializes member data.
- In this case, *ComputeMarketingExpenseAndInitializeMemberData()* would be an adequate name yet a bit long.
  - Again, refer to **COHESION**
- The routine itself is terrible!

Statements That Must Be in a Specific Order

- ***Use routine parameters to make dependencies obvious***
- In the earlier example, since no data is passed between routines, *you don't know whether any of the routines use the same data*.
- A better version:

Visual Basic Example of Data That Suggests an Order Dependency

```
InitializeExpenseData( expenseData )
ComputeMarketingExpense( expenseData )
ComputeSalesExpense( expenseData )
ComputeTravelExpense( expenseData )
ComputePersonnelExpense( expenseData )
DisplayExpenseSummary( expenseData )
```

Statements That Must Be in a Specific Order

Use routine parameters to make dependencies obvious

- Because all the routines use *expenseData*, you have a hint that they might be working on the same data and that the order of the statements might be important. How to make it even clearer?
- A better approach might be to convert the functions such that take *expenseData* as inputs and return updated *expenseData* as outputs, which makes it even clearer for order dependencies.

Visual Basic Example of Data and Routine Calls That Suggest an Order Dependency

```
expenseData = InitializeExpenseData( expenseData )
expenseData = ComputeMarketingExpense( expenseData )
expenseData = ComputeSalesExpense( expenseData )
expenseData = ComputeTravelExpense( expenseData )
expenseData = ComputePersonnelExpense( expenseData )
DisplayExpenseSummary( expenseData )
```

Statements That Must Be in a Specific Order

- Data can also indicate that execution order is or isn't important:

Visual Basic Example of Data That Doesn't Indicate an Order Dependency

```
ComputeMarketingExpense( marketingData )
ComputeSalesExpense( salesData )
ComputeTravelExpense( travelData )
ComputePersonnelExpense( personnelData )
DisplayExpenseSummary( marketingData, salesData, travelData, personnelData )
```

- Since the routines in the first four lines don't have any data in common, the code implies that the order doesn't matter. Because the last line uses data from each of the first four routines, you can assume

Statements That Must Be in a Specific Order

Document unclear dependencies with comments

- Try first to write code without order dependencies.
- Try second to write code that makes dependencies obvious.
- *If you're still concerned that an order dependency isn't explicit enough, document it*.
- Documenting unclear dependencies is one aspect of documenting coding assumptions, which is critical to writing maintainable, modifiable code.
- In the example, comments along these lines would be helpful:

Statements That Must Be in a Specific Order

- ***Document unclear dependencies with comments***
  - Still, better not to rely on just comments

Visual Basic Example of Statements in Which Order Dependencies Are Hidden but Clarified with Comments

```
' Compute expense data. Each of the routines accesses the
' member data expenseData. DisplayExpenseSummary
' should be called last because it depends on data calculated
' by the other routines.
InitializeExpenseData
ComputeMarketingExpense
ComputeSalesExpense
ComputeTravelExpense
ComputePersonnelExpense
DisplayExpenseSummary
```

Statements That Must Be in a Specific Order

Check for dependencies with assertions or error-handling code

- If the code is critical enough, you might use status variables and error-handling code or assertions to document critical sequential dependencies.
- ***For example***, in the class's constructor, you might initialize a class member variable *isExpenseDataInitialized* to *false*.
- Then in *InitializeExpenseData(),* you can set *isExpenseDataInitialized* to *true*. Each function that depends on *expenseData* being initialized can then check whether *isExpenseDataInitialized* has been set to *true* before performing additional operations on *expenseData*.
- Depending on how extensive the dependencies are, you might also need variables like *isMarketingExpenseComputed*, *isSalesExpenseComputed*, and so on.

Statements That Must Be in a Specific Order

Check for dependencies with assertions or error-handling code

- This technique creates new variables, new initialization code, and new error-checking code, all of which create additional possibilities for error.
- *The benefits of this technique should be weighed against the additional complexity and increased chance of secondary errors that this technique creates*.

Statements Whose Order Doesn't Matter

- You might encounter cases in which it seems as if the order of a few statements or a few blocks of code doesn't matter at all.
- But ordering affects readability, performance, and maintainability, and *in the absence of execution-order dependencies, you can use secondary criteria to determine the order of statements or blocks of code*.
- The guiding principle is the Principle of Proximity: *Keep related actions together*.

Statements Whose Order Doesn’t Matter

Making Code Read from Top to Bottom

- As a general principle, make the program read from top to bottom rather than jumping around.
- Simply making the control flow from top to bottom at run time isn’t enough.
- If someone who is reading your code has to search the whole program to find needed information, you should reorganize the code.

Statements Whose Order Doesn't Matter

Making Code Read from Top to Bottom

- Suppose that you want to determine how *marketingData* is calculated.
- You have to start at the last line and track all references to *marketingData* back to the first line. *marketingData* is used in only a few other places, but you have to keep in mind how *marketingData* is used everywhere between the first and last references to it.

C++ Example of Bad Code That Jumps Around

```
MarketingData marketingData;
SalesData salesData;
TravelData travelData;

travelData.ComputeQuarterly();
salesData.ComputeQuarterly();
marketingData.ComputeQuarterly();

salesData.ComputeAnnual();
marketingData.ComputeAnnual();
travelData.ComputeAnnual();

salesData.Print();
travelData.Print();
marketingData.Print();
```

Statements Whose Order Doesn’t Matter

Making Code Read from Top to Bottom

C++ Example of Good, Sequential Code That Reads from Top to Bottom

```
MarketingData marketingData;
marketingData.ComputeQuarterly();
marketingData.ComputeAnnual();
marketingData.Print();

SalesData salesData;
salesData.ComputeQuarterly();
salesData.ComputeAnnual();
salesData.Print();

TravelData travelData;
travelData.ComputeQuarterly();
travelData.ComputeAnnual();
travelData.Print();
```

Statements Whose Order Doesn't Matter

Making Code Read from Top to Bottom

- This code is better in several ways. References to each object are kept close together; they're "localized."
- The number of lines of code in which the objects are "live" is small.
- The code now looks as if it could be broken into separate routines (remember **COHESION**) for marketing, sales, and travel data. The first code fragment gave no hint that such a decomposition was possible.

Statements Whose Order Doesn’t Matter

Grouping Related Statements

- They can be related because they operate on the same data, perform similar tasks, or depend on each other’s being performed in order.
- An easy way to test whether related statements are grouped well is to print out a listing of your routine and then draw boxes around the related statements.
- If the statements are ordered well, you’ll get a picture like that shown in the figure, in which the boxes don’t overlap.

Statements Whose Order Doesn't Matter

Grouping Related Statements

- If statements aren't ordered well, you'll get a picture something like that shown in the figure, in which the boxes do overlap.
- If you find that your boxes overlap, reorganize your code so that related statements are grouped better.
- Once you've grouped related statements, you might find that they're strongly related and have no meaningful relationship to the statements that precede or follow them. In such a case, *you might want to refactor the* ***strongly related statements into their own routine***.

Using Conditionals

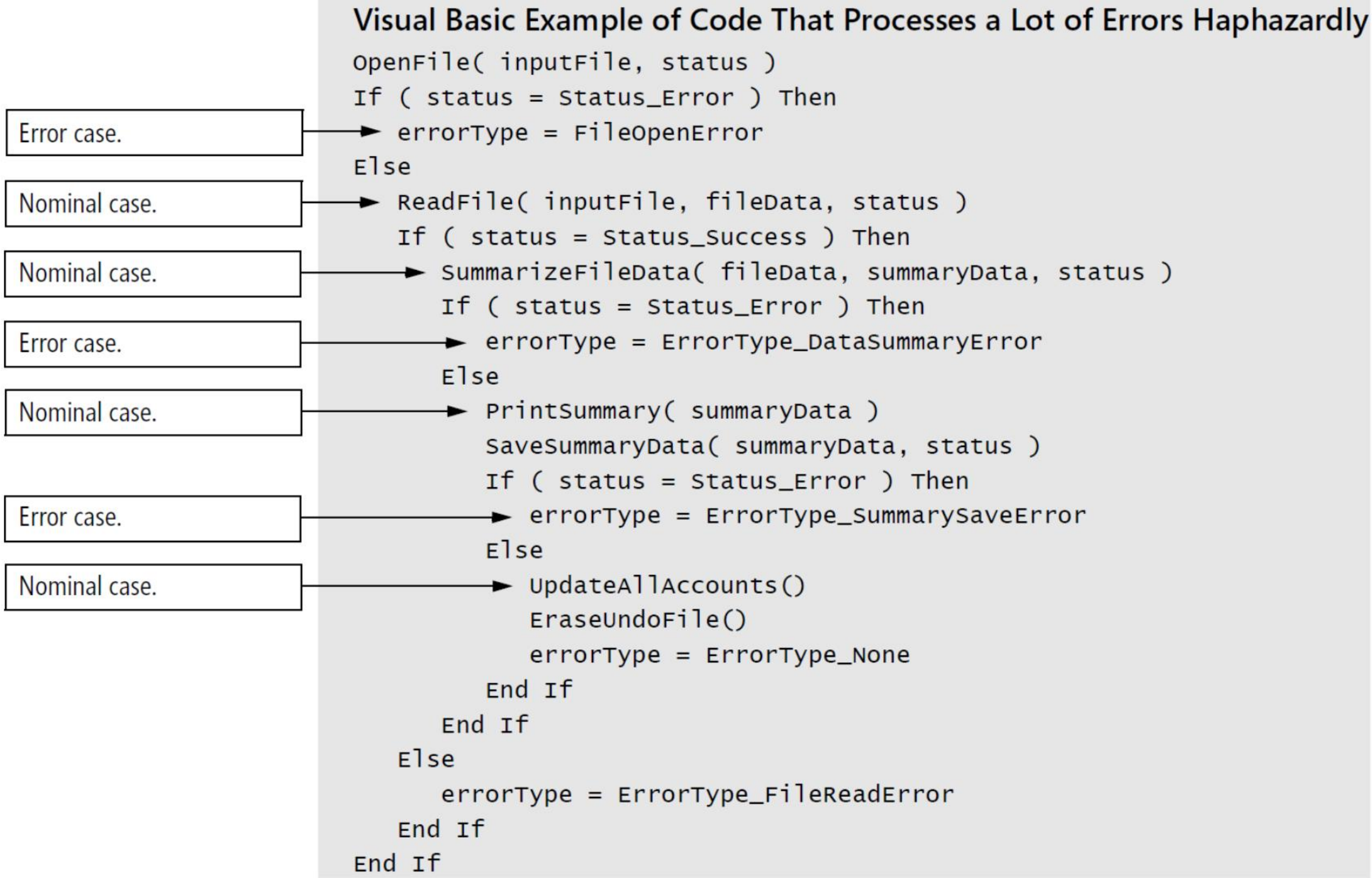
Outline

- *if* Statements
- *case* Statements

if Statements - Plain *if-then* Statements

- ***Put the normal case after the if rather than after the else***
- Put the case you normally expect to process first.
- This is in line with the general principle of putting code that results from a decision as close as possible to the decision.
- Here's a code example that does a lot of error processing, haphazardly checking for errors along the way:

Visual Basic Example of Code That Processes a Lot of Errors Haphazardly

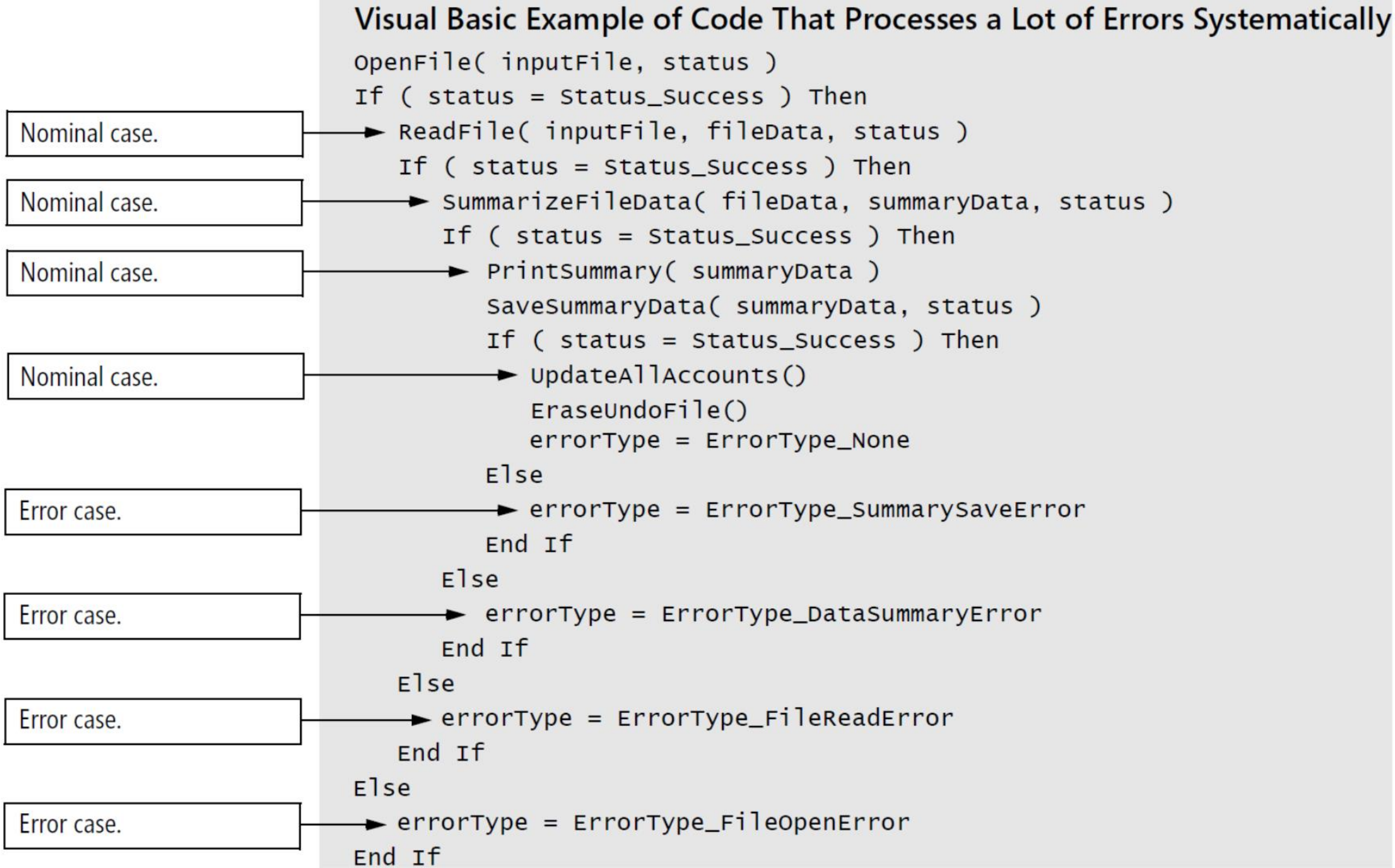



```
OpenFile( inputFile, status )
If ( status = Status_Error ) Then
   errorType = FileOpenError
Else
   ReadFile( inputFile, fileData, status )
   If ( status = Status_Success ) Then
      SummarizeFileData( fileData, summaryData, status )
      If ( status = Status_Error ) Then
         errorType = ErrorType_DataSummaryError
      Else
         PrintSummary( summaryData )
         SaveSummaryData( summaryData, status )
         If ( status = Status_Error ) Then
            errorType = ErrorType_SummarySaveError
         Else
            UpdateAllAccounts()
            EraseUndoFile()
            errorType = ErrorType_None
         End If
      End If
   Else
      errorType = ErrorType_FileReadError
   End If
End If
```

if Statements - Plain ***if-then*** Statements

- ***Put the normal* case *after the* if *rather than after the* else**
- This code is hard to follow because the normal cases and the error cases are all mixed together. It's hard to find the path that is normally taken through the code.
- In addition, because the error conditions are sometimes processed in the *if* clause rather than the *else* clause, it's hard to figure out which *if* test the normal case goes with.
- In the rewritten code, *on the next slide*, the normal path is consistently coded first and all the error cases are coded last. This makes it easier to find and read the nominal case.

Visual Basic Example of Code That Processes a Lot of Errors Systematically




```
OpenFile( inputFile, status )
If ( status = Status_Success ) Then
   ReadFile( inputFile, fileData, status )
   If ( status = Status_Success ) Then
      SummarizeFileData( fileData, summaryData, status )
      If ( status = Status_Success ) Then
         PrintSummary( summaryData )
         SaveSummaryData( summaryData, status )
         If ( status = Status_Success ) Then
            UpdateAllAccounts()
            EraseUndoFile()
            errorType = ErrorType_None
         Else
            errorType = ErrorType_SummarySaveError
         End If
      Else
         errorType = ErrorType_DataSummaryError
      End If
   Else
      errorType = ErrorType_FileReadError
   End If
Else
   errorType = ErrorType_FileOpenError
End If
```

if Statements - Plain *if-then* Statements

- ***Make sure that you branch correctly on equality***
- Using > instead of >= or < instead of <= is analogous to making an off-by-one error in accessing an array or computing a loop index.
- In a conditional statement, think through the equals case to avoid one.

if Statements - Plain *if-then* Statements

- ***Consider the* else *clause***
- *If you think you need a plain if statement, consider whether you don't actually need an if-then-else statement*.
- A classic General Motors analysis found that 50 to 80 percent of *if* statements should have had an *else* clause (Elshoff 1976).

if Statements - Plain *if-then* Statements

- ***Consider the* else *clause***
- One option is to code the ***else*** clause—with a **null statement** if necessary—to show that the ***else*** case has been considered.
- Coding null *else*s just to show that that case considered might be excessive, but at the very least, take the *else* case into account. When you have an *if* test without an *else*, unless the reason is obvious, use comments to explain why the *else* clause isn't necessary, like so:
    - Remember Assertions

Java Example of a Helpful, Commented *else* Clause

```
// if color is valid
if ( COLOR_MIN <= color && color <= COLOR_MAX ) {
   // do something
   ...
}
else {
   // else color is invalid
   // screen not written to -- safely ignore command
}
```

if Statements - Plain *if-then* Statements

- ***Test the* else *clause for correctness***
- When testing your code, you might think that the main clause, the *if*, is all that needs to be tested.
- If it's possible to test the *else* clause, however, be sure to do that.

if Statements - Plain ***if-then*** Statements

- ***Check for reversal of the* if *and* else *clauses***
- A common mistake in programming *if-then*s is to flip-flop the code that's supposed to follow the *if* clause and the code that's supposed to follow the *else* clause or to get the logic of the *if* test backward.
- Check your code for this common error.

if Statements - Plain ***if-then*** Statements

- *What is the output for x=10, y=5?*
  - *Dangling-else Problem: "x is <= 5"*

    The Java compiler always associates an else with the immediately preceding if unless told to do otherwise by the placement of braces ({ and }).

```
if ( x > 5 )
   if ( y > 5 )
      System.out.println( "x and y are > 5" )
else
   System.out.println( "x is <= 5" );
```

- *The compiler interprets the statement as*

```
if ( x > 5 )
   if ( y > 5 )
      System.out.println( "x and y are > 5" );
   else
      System.out.println( "x is <= 5" );
```

if Statements - Chains of ***if-then-else*** Statements

- In languages that don't support *case* statements—or that support them only partially—you'll often find yourself writing chains of *if-then-else* tests.

if Statements - Chains of *if-then-else* Statements

- How to make it better?

C++ Example of Using an *if-then-else* Chain to Categorize a Character

```
if ( inputCharacter < SPACE ) {
   characterType = CharacterType_ControlCharacter;
}
else if (
   inputCharacter == ' ' ||
   inputCharacter == ',' ||
   inputCharacter == '.' ||
   inputCharacter == '!' ||
   inputCharacter == '(' ||
   inputCharacter == ')' ||
   inputCharacter == ':' ||
   inputCharacter == ';' ||
   inputCharacter == '?' ||
   inputCharacter == '-'
   ) {
   characterType = CharacterType_Punctuation;
}
else if ( '0' <= inputCharacter && inputCharacter <= '9' ) {
   characterType = CharacterType_Digit;
}
else if (
   ( 'a' <= inputCharacter && inputCharacter <= 'z' ) ||
   ( 'A' <= inputCharacter && inputCharacter <= 'Z' )
   ) {
   characterType = CharacterType_Letter;
}
```

if Statements - Chains of *if-then-else* Statements

- ***Simplify complicated tests with boolean function calls***
- One reason the code in the previous example is hard to read is that the tests that categorize the character are complicated.
- To improve readability, you can replace them with calls to Boolean functions.

if Statements - Chains of *if-then-else* Statements

- ***Simplify complicated tests with boolean function calls***
- Here's how the example's code looks when the tests are replaced with boolean functions:

C++ Example of an *if-then-else* Chain That Uses Boolean Function Calls

```
if ( IsControl( inputCharacter ) ) {
   characterType = CharacterType_ControlCharacter;
}
else if ( IsPunctuation( inputCharacter ) ) {
   characterType = CharacterType_Punctuation;
}
else if ( IsDigit( inputCharacter ) ) {
   characterType = CharacterType_Digit;
}
else if ( IsLetter( inputCharacter ) ) {
   characterType = CharacterType_Letter;
}
```

if Statements - Chains of *if-then-else* Statements

- ***Put the most common cases first***
- Minimize the amount of exception-case handling code someone has to read to find the usual cases. You improve efficiency because you minimize the number of tests the code does to find the most common cases.
- In the example just shown, letters would be more common than punctuation but the test for punctuation is made first. Here's the code revised so that it tests for letters first:

if Statements - Chains of *if-then-else* Statements

This test, the most common, is now done first.

This test, the least common, is now done last.

C++ Example of Testing the Most Common Case First

```
if ( IsLetter( inputCharacter ) ) {
   characterType = CharacterType_Letter;
}
else if ( IsPunctuation( inputCharacter ) ) {
   characterType = CharacterType_Punctuation;
}
else if ( IsDigit( inputCharacter ) ) {
   characterType = CharacterType_Digit;
}
else if ( IsControl( inputCharacter ) ) {
   characterType = CharacterType_ControlCharacter;
}
```

if Statements - Chains of *if-then-else* Statements

- ***Make sure that all cases are covered***
- Code a final *else* clause with an error message or assertion to catch cases you didn't plan for.
- This error message is intended for you rather than for the user, so word it appropriately. Here's how you can modify the character classification example to perform an "other cases" test:

C++ Example of Using the Default Case to Trap Errors

```
if ( IsLetter( inputCharacter ) ) {
   characterType = CharacterType_Letter;
}
else if ( IsPunctuation( inputCharacter ) ) {
   characterType = CharacterType_Punctuation;
}
else if ( IsDigit( inputCharacter ) ) {
   characterType = CharacterType_Digit;
}
else if ( IsControl( inputCharacter ) ) {
   characterType = CharacterType_ControlCharacter;
}
else {
   DisplayInternalError( "Unexpected type of character detected." );
}
```

case Statements

- A variation of multiple *if-else* statements
- A switch statement allows a variable to be tested for equality against a list of values.
- Each value is called a case, and the variable being switched on is checked for each case.

```
switch(expression) {
   case value :
      // Statements
      break; // optional

   case value :
      // Statements
      break; // optional

   // You can have any number of case statements
   default : // Optional
      // Statements
}
```

case Statements

- Deciding whether to use if-then-else statements or a switch statement is based on readability and the expression that the statement is testing.
- An if-then-else statement can test expressions based on ranges of values or conditions, *whereas* a switch statement tests expressions based only on a single integer, enumerated value, or String object.

case Statements

- ***Order cases alphabetically or numerically***
- If cases are equally important, putting them in A-B-C order improves readability. That way a specific case is easy to pick out of the group

case Statements

- ***Put the normal case first***
- If you have one normal case and several exceptions, put the normal case first.
- Indicate with comments that it's the normal case and that the others are unusual.

case Statements

- ***Order cases by frequency***
- Put the most frequently executed cases first and the least frequently executed last.
- This way, human readers can find the most common cases easily. Readers scanning the list for a specific case are likely to be interested in one of the most common cases, and putting the common ones at the top of the code makes the search quicker.

case Statements

- ***Keep the actions of each case simple***
- Keep the code associated with each case short.
- Short code following each case helps make the structure of the *case* statement clear.
- If the actions performed for a case are complicated, write a routine and call the routine from the case rather than putting the code into the case itself.

case Statements

- ***Don't make up misleading variables to be able to use the* case *statement***
- A *case* statement should be used for simple data that's easily categorized. If your data isn't simple, use chains of *if-then-else*s instead.
- Misleading variables are confusing, and you should avoid them. For example, don't do this:

Java Example of Creating a Phony *case* Variable—Bad Practice

```
action = userCommand[ 0 ];
switch ( action ) {
   case 'c':
      Copy();
      break;
   case 'd':
      DeleteCharacter();
      break;
   case 'f':
      Format();
      break;
   case 'h':
      Help();
      break;
   ...
   default:
      HandleUserInputError( ErrorType.InvalidUserCommand );
}
```

case Statements

- ***Don't make up misleading variables to be able to use the* case *statement***
- In general, when you manufacture a variable to use in a *case* statement, the real data might not map onto the *case* statement as you like.
- In this example, if the user types **copy**, the *case* statement peels off the first "c" and correctly calls the *Copy()* routine.
- On the other hand, if the user types **cement overshoes**, **clambake**, or **cellulite**, the *case* statement also peels off the "c" and calls *Copy()*.
- The test for an erroneous command in the *case* statement's *else* clause won't work very well because it will miss only erroneous first letters rather than erroneous commands.

Java Example of Using *if-then-elses* Instead of a Phony *case* Variable—Good Practice

```
if ( UserCommand.equals( COMMAND_STRING_COPY ) ) {
   Copy();
}
else if ( UserCommand.equals( COMMAND_STRING_DELETE ) ) {
   DeleteCharacter();
}
else if ( UserCommand.equals( COMMAND_STRING_FORMAT ) ) {
   Format();
}
else if ( UserCommand.equals( COMMAND_STRING_HELP ) ) {
   Help();
}
...
else {
   HandleUserInputError( ErrorType_InvalidCommandInput );
}
```

case Statements

- ***Use the default clause only to detect legitimate defaults***
- You might sometimes have only one case remaining and decide to code that case as the default clause.
- When you do that, you lose the automatic documentation provided by *case*-statement labels, and you lose the ability to detect errors with the default clause.
- Such *case* statements break down under modification. If you use a legitimate default, adding a new case is trivial—you just add the case and the corresponding code. Otherwise, you have to add the new case, possibly making it the new default, and then change the case previously used as the default so that it's a legitimate case.

case Statements

- ***Use the default clause to detect errors***
- If the default clause in a *case* statement isn't being used for other processing and isn't supposed to occur, put a diagnostic message in it:

Java Example of Using the Default Case to Detect Errors—Good Practice

```
switch ( commandShortcutLetter ) {
   case 'a':
      PrintAnnualReport();
      break;
   case 'p':
      // no action required, but case was considered
      break;
   case 'q':
      PrintQuarterlyReport();
      break;
   case 's':
      PrintSummaryReport();
      break;
   default:
      DisplayInternalError( "Internal Error 905: Call customer support." );
}
```

case Statements

- What is the output?
  - Fall Through

```
public class SwitchDemoFallThrough {

    public static void main(String[] args) {
        java.util.ArrayList<String> futureMonths =
            new java.util.ArrayList<String>();

        int month = 8;

        switch (month) {
            case 1:  futureMonths.add("January");
            case 2:  futureMonths.add("February");
            case 3:  futureMonths.add("March");
            case 4:  futureMonths.add("April");
            case 5:  futureMonths.add("May");
            case 6:  futureMonths.add("June");
            case 7:  futureMonths.add("July");
            case 8:  futureMonths.add("August");
            case 9:  futureMonths.add("September");
            case 10: futureMonths.add("October");
            case 11: futureMonths.add("November");
            case 12: futureMonths.add("December");
                     break;
            default: break;
        }

        if (futureMonths.isEmpty()) {
            System.out.println("Invalid month number");
        } else {
            for (String monthName : futureMonths) {
               System.out.println(monthName);
            }
        }
    }
}
```

case Statements

- ***In C++ and Java, avoid dropping through the end of a* case *statement***
- C-like languages (C, C++, and Java) don't automatically break out of each case. Instead, you have to code the end of each case explicitly.
- If you don't code the end of a case, the program drops through the end and executes the code for the next case.
- This can lead to some particularly egregious coding practices, including the following horrible example:

case Statements

- ***In C++, clearly and unmistakably identify flow-throughs at the end of a* case *statement***
- If you intentionally write code to drop through the end of a case, clearly comment the place at which it happens and explain why it needs to be coded that way.

C++ Example of Documenting Falling Through the End of a *case* Statement

```
switch ( errorDocumentationLevel ) {
   case DocumentationLevel_Full:
      DisplayErrorDetails( errorNumber );
      // FALLTHROUGH -- Full documentation also prints summary comments

   case DocumentationLevel_Summary:
      DisplayErrorSummary( errorNumber );
      // FALLTHROUGH -- Summary documentation also prints error number

   case DocumentationLevel_NumberOnly:
      DisplayErrorNumber( errorNumber );
      break;

   default:
      DisplayInternalError( "Internal Error 905: Call customer support." );
}
```

Controlling Loops

Outline

- Loop Types
- Controlling Loops
- Creating Loops

Selecting the Kind of Loop

- *In most languages, you'll use a few kinds of loops*:
- The counted loop is performed a specific number of times, perhaps one time for each employee.
- The continuously evaluated loop doesn't know ahead of time how many times it will be executed and tests whether it has finished on each iteration. ***For example***, it runs while money remains, until the user selects quit, or until it encounters an error.
- The endless loop executes forever once it has started. It's the kind you find in embedded systems such as pacemakers, microwave ovens, and cruise controls.
- The iterator loop performs its action once for each element in a container class.

Selecting the Kind of Loop

- The kinds of loops are differentiated first by flexibility—whether the loop executes a specified number of times or whether it tests for completion on each iteration.
- The kinds of loops are also differentiated by the location of the test for completion.
- You can put the test at the beginning, the middle, or the end of the loop. This characteristic tells you whether the loop executes at least once.
  - If the loop is tested at the beginning, its body isn't necessarily executed.
  - If the loop is tested at the end, its body is executed at least once.
  - If the loop is tested in the middle, the part of the loop that precedes the test is executed at least once, but the part of the loop that follows the test isn't necessarily executed at all.

Selecting the Kind of Loop

- Loops w.r.t. flexibility and test location

Language	Kind of Loop	Flexibility	Test Location
C, C++, C#, Java	*for*	flexible	beginning
	while	flexible	beginning
	do-while	flexible	end

Selecting the Kind of Loop - ***while*** Loop

- Although it's not quite that flexible, a *while* loop is a flexible loop choice.
- *If you don't know ahead of time exactly how many times you'll want the loop to iterate, use a while loop*.
- The test for the loop exit is performed only once each time through the loop, and the main issue with respect to *while* loops is deciding whether to test at the beginning or the end of the loop.

Selecting the Kind of Loop - ***while*** Loop

- **Loop with Test at the Beginning**

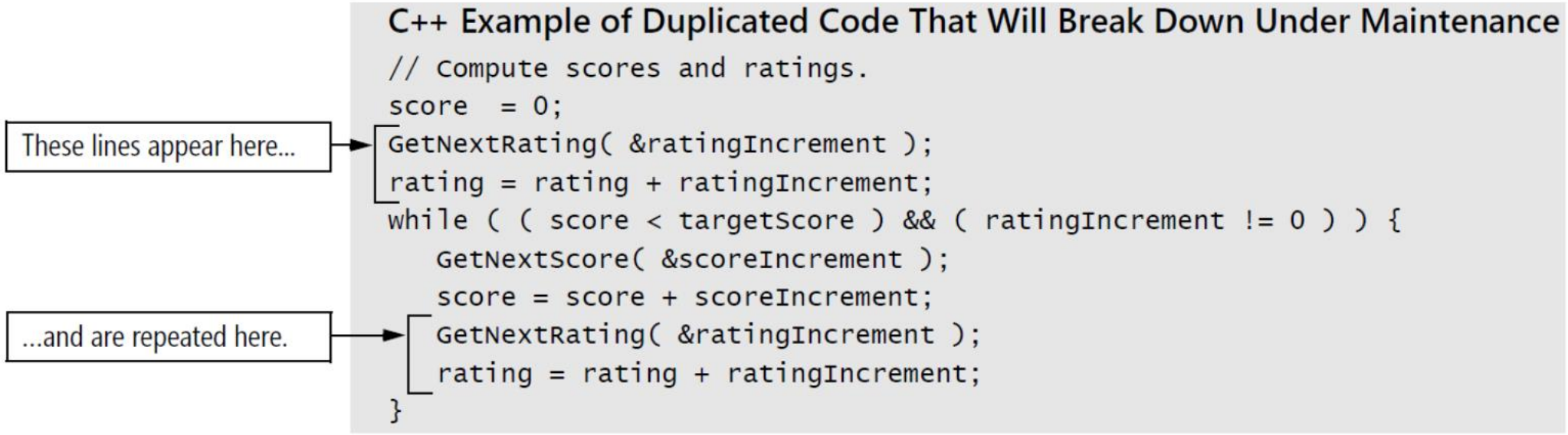
  For a loop that tests at the beginning, you can use a *while* loop in C++, C#, Java, Visual Basic, and most other languages. You can emulate a *while* loop in other languages.

- **Loop with Test at the End**

  You might occasionally have a situation in which you want a flexible loop, but the loop needs to execute at least one time. In such a case, you can use a *while* loop that is tested at its end. You can use *do-while* in C++, C#, and Java, *Do-Loop-While* in VisualBasic, or you can emulate end-tested loops in other languages.

Selecting the Kind of Loop - Loop-With-Exit Loop

- What is wrong?



C++ Example of Duplicated Code That Will Break Down Under Maintenance

```
// Compute scores and ratings.
score  = 0;
GetNextRating( &ratingIncrement );
rating = rating + ratingIncrement;
while ( ( score < targetScore ) && ( ratingIncrement != 0 ) ) {
   GetNextScore( &scoreIncrement );
   score = score + scoreIncrement;
   GetNextRating( &ratingIncrement );
   rating = rating + ratingIncrement;
}
```

Selecting the Kind of Loop - Loop-With-Exit Loop

- The two lines of code at the top of the example are repeated in the last two lines of code of the *while* loop.
- During modification, you can easily forget to keep the two sets of lines parallel.
- Another programmer modifying the code probably won't even realize that the two sets of lines are supposed to be modified in parallel. Either way, the result will be errors arising from incomplete modifications.
- Here's how you can rewrite the code more clearly:

Selecting the Kind of Loop - Loop-With-Exit Loop

This is the loop-exit condition

C++ Example of a Loop-With-Exit Loop That's Easier to Maintain

```
// Compute scores and ratings. The code uses an infinite loop
// and a break statement to emulate a loop-with-exit loop.
score = 0;
while ( true ) {
   GetNextRating( &ratingIncrement );
   rating = rating + ratingIncrement;

   if ( !( ( score < targetScore ) && ( ratingIncrement != 0 ) ) ) {
      break;
   }

   GetNextScore( &scoreIncrement );
   score = score + scoreIncrement;
}
```

Selecting the Kind of Loop - Loop-With-Exit Loop

- Convert it to a Loop-With-Exit Loop

```
goto Start;
while ( expression ) {
   // do something
   ...

   Start:

   // do something else
   ...
}
```

The blocks before and after the *break* have been switched.

C++ Example of Code Rewritten Without a *goto*—Better Practice

```
while ( true ) {
   // do something else
   ...

   if ( !( expression ) ) {
      break;
   }

   // do something
   ...
}
```

Selecting the Kind of Loop - Loop-With-Exit Loop

- Another kind of loop-with-exit loop that's used to avoid a loop-and-a-half is shown here:

C++ Example of Entering the Middle of a Loop with a *goto*—Bad Practice

```
goto Start;
while ( expression ) {
   // do something
   ...

   Start:

   // do something else
   ...
}
```

Selecting the Kind of Loop - Loop-With-Exit Loop

- At first glance, this seems to be similar to the previous loop-with-exit examples.
- It's used in simulations in which // *do something* doesn't need to be executed at the first pass through the loop but // *do something* else does. It's a one-in, one-out control construct:
- The only way into the loop is through the *goto* at the top, and the only way out of the loop is through the *while* test.
- This approach has two problems: it uses a *goto*, and it's unusual enough to be confusing.

Selecting the Kind of Loop - Loop-With-Exit Loop

- In C++, you can accomplish the same effect without using a *goto*, as demonstrated in the following example. If the language you're using doesn't support a *break* command, you can emulate one with a *goto*.

The blocks before and after the *break* have been switched.

C++ Example of Code Rewritten Without a *goto*—Better Practice

```
while ( true ) {
   // do something else
   ...

   if ( !( expression ) ) {
      break;
   }

   // do something
   ...
}
```

Selecting the Kind of Loop – for vs. while Loop

- A *for* loop is a good choice when you need a loop that executes a specified number of times.
- Use *for* loops for simple activities that don't require internal loop controls.
- Use them when the loop control involves simple increments or simple decrements, such as iterating through the elements in a container.
- The point of a *for* loop is that you set it up at the top of the loop and then forget about it. You don't have to do anything inside the loop to control it.
- If you have a condition under which execution has to jump out of a loop, use a *while* loop instead.
- Likewise, don't explicitly change the index value of a *for* loop to force it to terminate. Use a *while* loop instead. The *for* loop is for simple uses.
- Most complicated looping tasks are better handled by a *while* loop.

Controlling the Loop

- What can go wrong with a loop?
  - incorrect or omitted loop initialization,
  - omitted initialization of accumulators or other variables related to the loop,
  - improper nesting,
  - incorrect termination of the loop,
  - forgetting to increment a loop variable or incrementing the variable incorrectly, and
  - indexing an array element from a loop index incorrectly.

Controlling the Loop

- You can prevent these problems by observing two practices.
- First, minimize the number of factors that affect the loop. Simplify!
- Second, treat the inside of the loop as if it were a routine—keep as much of the loop control as possible outside the loop.

Controlling the Loop

- Explicitly state the conditions under which the body of the loop is to be executed.
- Don’t make the reader look inside the loop to understand the loop control.
- Think of a loop as a black box: the surrounding program knows the control conditions but not the contents.

C++ Example of Treating a Loop as a Black Box

```
while ( !inputFile.EndOfFile() && moreDataAvailable ) {
    [redacted]
}
```

Controlling the Loop – Entering the Loop

- ***Enter the loop from one location only***
- A variety of loop-control structures allows you to test at the beginning, middle, or end of a loop.
- These structures are rich enough to allow you to enter the loop from the top every time. You don’t need to enter at multiple locations.

Controlling the Loop – Entering the Loop

- ***Put initialization code directly before the loop***
- The Principle of Proximity advocates putting related statements together.
- Keep loop-initialization statements with the loop they're related to.
- If you don't, you're more likely to cause errors when you generalize the loop into a bigger loop and forget to modify the initialization code.

Controlling the Loop – Entering the Loop

- ***Use* while(true) *for infinite loops***
- You might have a loop that runs without terminating— ***for example***, a loop in firmware such as a pacemaker or a microwave oven.
- Or you might have a loop that terminates only in response to an event—an "event loop."
- You could code such an infinite loop in several ways. Faking an infinite loop with a statement like *for i = 1 to 99999* is a poor choice because the specific loop limits muddy the intent of the loop—*99999* could be a legitimate value. *Such a fake infinite loop can also break down under maintenance*.

Controlling the Loop – Entering the Loop

- ***Use* while(true) *for infinite loops***
- The *while(true)* idiom is considered a standard way of writing an infinite loop in C++, Java, Visual Basic, and other languages that support comparable structures.
- Can you do this with a ***for loop***?

  for(;;) is an acceptable alternative, yet an unnatural one.

Controlling the Loop – Entering the Loop

- ***Prefer* for *loops when they're appropriate***
- The *for* loop packages loop-control code in one place, which makes for easily readable loops.
- One mistake programmers commonly make when modifying software is changing the loop-initialization code at the top of a *while* loop but forgetting to change related code at the bottom.
- In a *for* loop, all the relevant code is together at the top of the loop, which makes correct modifications easier.
- If you can use the *for* loop appropriately instead of another kind of loop, do it.

Controlling the Loop – Entering the Loop

- ***Don't use a for loop when a while loop is more appropriate***
- A common abuse of the flexible *for* loop structure in C++, C#, and Java is haphazardly cramming the contents of a *while* loop into a *for* loop header.
- The following example shows a *while* loop crammed into a *for* loop header:

C++ Example of a *while* Loop Abusively Crammed into a *for* Loop Header

```
// read all the records from a file
for ( inputFile.MoveToStart(), recordCount = 0; !inputFile.EndOfFile();
   recordCount++ ) {
   inputFile.GetRecord();
}
```

Controlling the Loop – Entering the Loop

- ***Don't use a* for *loop when a* while *loop is more appropriate***
- Reserve the *for* loop header for loop-control statements—statements that initialize the loop, terminate it, or move it toward termination.
- ***In the example***, the *inputFile.GetRecord()* statement in the body of the loop moves the loop toward termination, but the *recordCount* statements don't; they're housekeeping statements that don't control the loop's progress.
- Putting the *recordCount* statements in the loop header and leaving the *inputFile.GetRecord()* statement out is misleading; it creates the false impression that *recordCount* controls the loop.

Controlling the Loop – Entering the Loop

- ***Don't use a* for *loop when a* while *loop is more appropriate***
- If you want to use the *for* loop rather than the *while* loop in this case, put the loop-control statements in the loop header and leave everything else out. Here's the right way to use the loop header:

C++ Example of Logical if Unconventional Use of a *for* Loop Header

```
recordCount = 0;
for ( inputFile.MoveToStart(); !inputFile.EndOfFile(); inputFile.GetRecord() ) {
   recordCount++;
}
```

Controlling the Loop – Entering the Loop

- ***Don't use a* for *loop when a* while *loop is more appropriate***
- The contents of the loop header in this example are all related to control of the loop.
- The *inputFile.MoveToStart()* statement initializes the loop, the *!inputFile.EndOfFile()* statement tests whether the loop has finished, and the *inputFile.GetRecord()* statement moves the loop toward termination.
- The statements that affect *recordCount* don't directly move the loop toward termination and are appropriately not included in the loop header. The *while* loop is still more appropriate for this job, but at least this code uses the loop header logically.

Controlling the Loop – Entering the Loop

- ***Don't use a* for *loop when a* while *loop is more appropriate***
- Here's how the code looks when it uses a *while* loop:

C++ Example of Appropriate Use of a *while* Loop

```
// read all the records from a file
inputFile.MoveToStart();
recordCount = 0;
while ( !inputFile.EndOfFile() ) {
   inputFile.GetRecord();
   recordCount++;
}
```

Controlling the Loop – Processing the Middle of the Loop

- ***Use { and } to enclose the statements in a loop***
- Use code brackets every time.
- They don't cost anything in speed or space at run time, they help readability, and they help prevent errors as the code is modified.
- *They're a good defensive-programming practice*.

Controlling the Loop – Processing the Middle of the Loop

- Is this a good while loop?

```
while ( ( inputChar = dataFile.GetChar() ) != CharType_Eof ) {
    ;
}
```

Controlling the Loop – Processing the Middle of the Loop

- ***Avoid empty loops***
- It's possible to create an empty loop, one in which the work the loop is doing is coded on the same line as the test that checks whether the work is finished.
- The loop is empty because the *while* expression includes two things: the work of the loop—*inputChar = dataFile.GetChar()*—and a test for whether the loop should terminate—*inputChar != CharType_Eof*.

C++ Example of an Empty Loop

```
while ( ( inputChar = dataFile.GetChar() ) != CharType_Eof ) {
    ;
}
```

Controlling the Loop – Processing the Middle of the Loop

- ***Avoid empty loops***
- The loop would be clearer if it were recoded so that the work it does is evident to the reader:

C++ Example of an Empty Loop Converted to an Occupied Loop

```
do {
   inputChar = dataFile.GetChar();
} while ( inputChar != CharType_Eof  );
```

- The new code takes up three full lines rather than one line and a semicolon, which is appropriate since it does the work of three lines rather than that of one line and a semicolon.

Controlling the Loop – Processing the Middle of the Loop

- ***Keep loop-housekeeping chores at either the beginning or the end of the loop***
- Loophousekeeping chores are expressions like *i = i + 1* or *j++*, expressions whose main purpose isn't to do the work of the loop but to control the loop.
- The housekeeping is done at the end of the loop in the example on the next slide:

Controlling the Loop – Processing the Middle of the Loop

C++ Example of Housekeeping Statements at the End of a Loop

```
nameCount = 0;
totalLength = 0;
while ( !inputFile.EndOfFile() ) {
   // do the work of the loop
   inputFile >> inputString;
   names[ nameCount ] = inputString;
   ...

   // prepare for next pass through the loop--housekeeping
   nameCount++;
   totalLength = totalLength + inputString.length();
}
```

Here are the housekeeping statements.

One Class = One Purpose

One Routine = One Purpose

One Variable = One Purpose

~~*One Loop = One Purpose*~~

Controlling the Loop – Processing the Middle of the Loop

- ***Make each loop perform only one function***
- The mere fact that a loop can be used to do two things at once isn't sufficient justification for doing them together.
- Loops should be like routines in that each one should do only one thing and do it well.
- *If it seems inefficient to use two loops where one would suffice, write the code as two loops, comment that they could be combined for efficiency, and then wait until benchmarks show that the section of the program poses a performance problem before changing the two loops into one*.

Controlling the Loop – Exiting the Loop

- ***Assure yourself that the loop ends***
- Mentally simulate the execution of the loop until you are confident that, in all circumstances, it ends.
- Think through the normal cases, the endpoints, and each of the exceptional cases.

Controlling the Loop – Exiting the Loop

- ***Make loop-termination conditions obvious***
- If you use a *for* loop and don't fool around with the loop index and don't use a *goto* or *break* to get out of the loop, the termination condition will be obvious.
- Likewise, if you use a *while* loop and put all the control in the *while* clause, the termination condition will be obvious.
- The key is putting the control in one place.

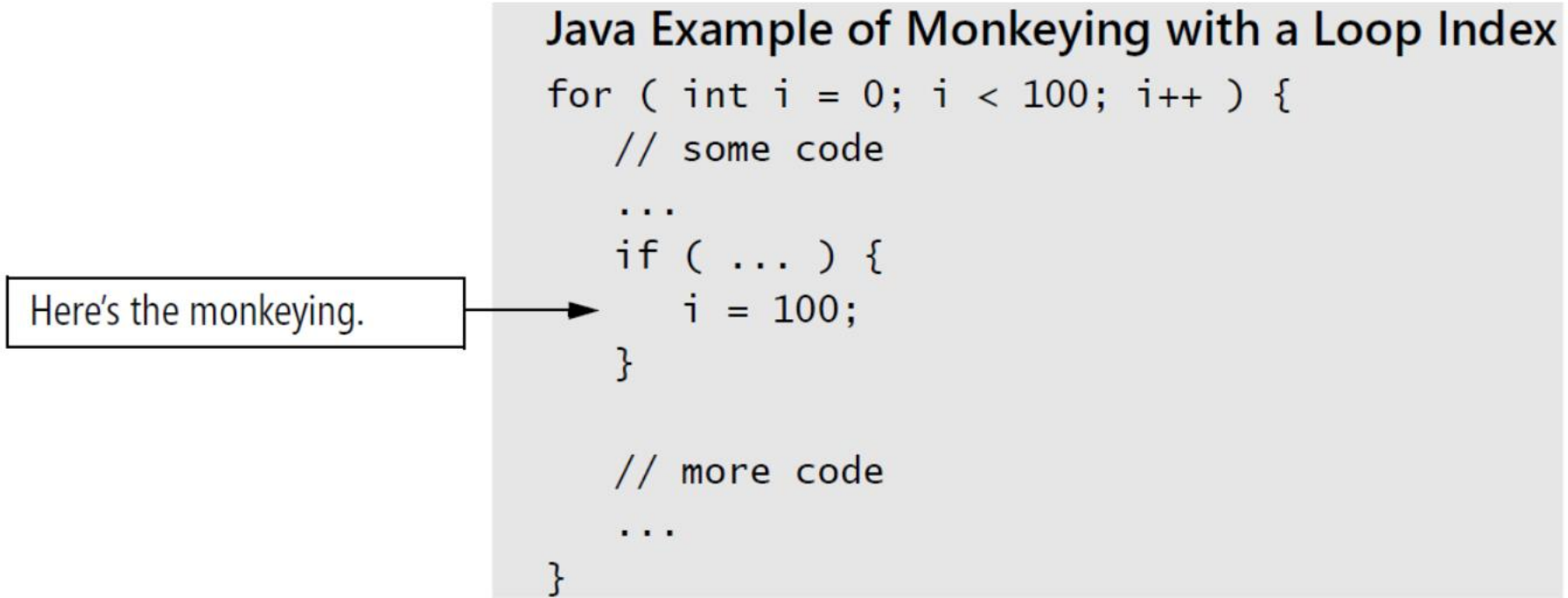
Controlling the Loop – Exiting the Loop

- ***Any comments on this loop?***

```
for ( int i = 0; i < 100; i++ ) {
    // some code
    ...
    if ( ... ) {
        i = 100;
    }

    // more code
    ...
}
```

Controlling the Loop – Exiting the Loop

- ***Don't monkey with the loop index of a* for *loop to make the loop terminate***
- Some programmers change the value of a *for* loop index to make the loop terminate early.



Java Example of Monkeying with a Loop Index

```
for ( int i = 0; i < 100; i++ ) {
   // some code
   ...
   if ( ... ) {
      i = 100;
   }

   // more code
   ...
}
```

Controlling the Loop – Exiting the Loop

- ***Don't monkey with the loop index of a* for *loop to make the loop terminate***
- The intent in this example is to terminate the loop under some condition by setting *I* to *100*, a value that's larger than the end of the *for* loop's range of *0* through *99*.
- Use a *while* loop to provide more control over the loop's exit conditions.

Controlling the Loop – Exiting the Loop

- ***Avoid code that depends on the loop index's final value***
- It's bad form to use the value of the loop index after the loop.
- The value is different when the loop terminates normally and when it terminates abnormally.
- Even if you happen to know what the final value is without stopping to think about it, the next person to read the code will probably have to think about it.
- It's better form and more self-documenting if you assign the final value to a variable at the appropriate point inside the loop.

Controlling the Loop – Exiting the Loop

- ***Avoid code that depends on the loop index's final value***

C++ Example of Code That Misuses a Loop Index's Terminal Value

```
for ( recordCount = 0; recordCount < MAX_RECORDS; recordCount++ ) {
   if ( entry[ recordCount ] == testValue ) {
      break;
   }
}
// lots of code
...
if ( recordCount < MAX_RECORDS ) {
   return( true );
}
else {
   return( false );
}
```

Here's the misuse of the loop index's terminal value.

Controlling the Loop – Exiting the Loop

- ***Avoid code that depends on the loop index's final value***
- In this fragment, the second test for *recordCount < MaxRecords* makes it appear that the loop is supposed to loop though all the values in *entry[]* and return *true* if it finds the one equal to *testValue* and *false* otherwise.
- It's hard to remember whether the index gets incremented past the end of the loop, so it's easy to make an off-by-one error.

Controlling the Loop – Exiting the Loop

- ***Avoid code that depends on the loop index's final value***
- You're better off writing code that doesn't depend on the index's final value. Here's how to rewrite the code:

C++ Example of Code That Doesn't Misuse a Loop Index's Terminal Value

```
found = false;
for ( recordCount = 0; recordCount < MAX_RECORDS; recordCount++ ) {
   if ( entry[ recordCount ] == testValue ) {
      found = true;
      break;
   }
}
// lots of code
...
return( found );
```

Controlling the Loop – Exiting the Loop

- ***Consider using safety counters***
- A safety counter is a variable you increment each pass through a loop to determine whether a loop has been executed too many times.
- If you have a program in which an error would be disastrous, you can use safety counters to ensure that all loops end. This C++ loop could profitably use a safety counter:

C++ Example of a Loop That Could Use a Safety Counter

```
do {
   node = node->Next;
   ...
} while ( node->Next != NULL );
```

Controlling the Loop – Exiting the Loop

- ***Consider using safety counters***

Here's the safety-counter code.

C++ Example of Using a Safety Counter

```
safetyCounter = 0;
do {
   node = node->Next;
   ...
   safetyCounter++;
   if ( safetyCounter >= SAFETY_LIMIT ) {
      Assert( false, "Internal Error: Safety-Counter Violation." );
   }
   ...
} while ( node->Next != NULL );
```

Controlling the Loop – Exiting the Loop

- ***Consider using safety counters***
- Safety counters are not a cure-all. Introduced into the code one at a time, safety counters increase complexity and can lead to additional errors.
- Because they aren't used in every loop, you might forget to maintain safety-counter code when you modify loops in parts of the program that do use them.
- If safety counters are instituted as a project-wide standard for critical loops, however, you learn to expect them and the safety-counter code is no more prone to produce errors later than any other code is.

Controlling the Loop – Exiting the Loop

- ***Be wary of a loop with a lot of* break*s scattered through it***
- A loop's containing a lot of *break*s can indicate unclear thinking about the structure of the loop or its role in the surrounding code.
- A proliferation of *break*s raises the possibility that the loop could be more clearly expressed as a series of loops rather than as one loop with many exits.
- Multiple *breaks* don't necessarily indicate an error, but their existence in a loop is a warning sign,

Controlling the Loop – Exiting the Loop

- ***Any possible issues?***

```
do {
    ...
    switch
        ...
        if () {
            ...
             break;
            ...
        }
        ...
} while ( ... );
```

Controlling the Loop – Exiting the Loop

- ***Be wary of a loop with a lot of* breaks *scattered through it***

C++ Example of Erroneous Use of a *break* Statement Within a *do-switch-if* Block

```
do {
   ...
   switch
      ...
      if () {
         ...
         break;
         ...
      }
      ...
} while ( ... );
```

This *break* was intended for the *if* but broke out of the *switch* instead.

Controlling the Loop – Exiting the Loop

- ***Use continue for tests at the top of a loop***
- A good use of *continue* is for moving execution past the body of the loop after testing a condition at the top.
- ***For example***, if the loop reads records, discards records of one kind, and processes records of another kind, you could put a test like this one at the top of the loop:

Controlling the Loop – Exiting the Loop

- ***Use continue for tests at the top of a loop***
- Using *continue* in this way lets you avoid an *if* test that would effectively indent the entire body of the loop. If, on the other hand, the *continue* occurs toward the middle or end of the loop, use an *if* instead.

Pseudocode Example of a Relatively Safe Use of *continue*

```
while ( not eof( file ) ) do
   read( record, file )
   if ( record.Type <> targetType ) then
      continue

   -- process record of targetType
   ...
end while
```

Controlling the Loop – Exiting the Loop

- ***Use the labeled break structure if your language supports it***
- Java supports use of labeled *break*s to prevent the kind of problem experienced.
- A labeled *break* can be used to exit a *for* loop, an *if* statement, or any block of code enclosed in braces (Arnold, Gosling, and Holmes 2000).
- ***An unlabeled break statement terminates the innermost switch, for, while, or do-while statement, but a labeled break terminates an outer statement.***

Controlling the Loop – Exiting the Loop

- ***Use the labeled* break *structure if your language supports it***

Java Example of a Better Use of a Labeled *break* Statement Within a *do-switch-if* Block

```
do {
   ...
   switch
      ...
      CALL_CENTER_DOWN:
      if () {
         ...
         break CALL_CENTER_DOWN;
         ...
      }
      ...
} while ( ... );
```

The target of the labeled *break* is unambiguous.

Controlling the Loop – Exiting the Loop

- ***What happens if***
  - ***xxx = first***
  - ***xxx = second***
  - ***xxx = third***
- ***xxx = first -> breaks the outer loop***
- ***xxx = second -> breaks the inner loop***
- ***xxx = third -> cannot compile***

```
first:
for( int i = 0; i < 10; i++) {
  second:
  for(int j = 0; j < 5; j ++ )
  {
    break xxx;
  }
}

third:
for( int a = 0; a < 10; a++) {

}
```

Controlling the Loop – Exiting the Loop

- ***Use* break *and* continue *only with caution***
- *Use of break eliminates the possibility of treating a loop as a black box*.
- Limiting yourself to only one statement to control a loop's exit condition is a powerful way to simplify your loops.
- Using a *break* forces the person reading your code to look inside the loop for an understanding of the loop control. That makes the loop more difficult to understand.
- Considering the alternatives, if you can't defend a *break* or a *continue*, don't use it.

Controlling the Loop – Checking Endpoints

- *A single loop usually has three cases of interest*:

  the first case, an arbitrarily selected middle case, and the last case. When you create a loop, mentally run through the first, middle, and last cases to make sure that the loop doesn't have any off-by- one errors.

- If you have any special cases that are different from the first or last case, check those too.
- If the loop contains complex computations, get out your calculator and manually check the calculations.

Controlling the Loop – Checking Endpoints

- Inefficient programmers tend to experiment randomly until they find a combination that seems to work. If a loop isn't working the way it's supposed to, the inefficient programmer changes the < sign to a <= sign.
- If that fails, the inefficient programmer changes the loop index by adding or subtracting 1. Eventually the programmer using this approach might stumble onto the right combination or simply replace the original error with a more subtle one.
- Even if this random process results in a correct program, it doesn't result in the programmer's knowing why the program is correct.

Controlling the Loop – Loop Variables

- Anything could go wrong?

```
for(index = 0.1; index < 10; index+=0.1){
    //do something
}
```

- Double/float variables may not work as expected

```
0.1
0.2
0.30000000000000004
0.4
0.5
0.6
0.7
0.7999999999999999
0.8999999999999999
0.9999999999999999
```

```
double nominal = 1.0;
double sum = 0.0;

for ( int i = 0; i < 10; i++ ) {
→ sum += 0.1;
}

if ( nominal == sum ) {
   System.out.println( "Numbers are the same." );
}
else {
   System.out.println( "Numbers are different." );
}
```

Controlling the Loop – Loop Variables

- ***Use ordinal or enumerated types for limits on both arrays and loops***
- *Generally, loop counters should be integer values*.
- Floating-point values don't increment well. For example, you could add 1.0 to 26,742,897.0 and get 26,742,897.0 instead of 26,742,898.0.
- If this incremented value were a loop counter, you'd have an infinite loop.

Controlling the Loop – Loop Variables

- ***Use meaningful variable names to make nested loops readable***
- Arrays are often indexed with the same variables that are used for loop indexes. If you have a one dimensional array, you might be able to get away with using *i*, *j*, or *k* to index it.
- But if you have an array with two or more dimensions, you should use meaningful index names to clarify what you're doing.
- Meaningful array-index names clarify both the purpose of the loop and the part of the array you intend to access.

Controlling the Loop – Loop Variables

- ***Use meaningful variable names to make nested loops readable***

Java Example of Bad Loop Variable Names

```
for ( int i = 0; i < numPayCodes; i++ ) {
   for ( int j = 0; j < 12; j++ ) {
      for ( int k = 0; k < numDivisions; k++ ) {
         sum = sum + transaction[ j ][ i ][ k ];
      }
   }
}
```

Controlling the Loop – Loop Variables

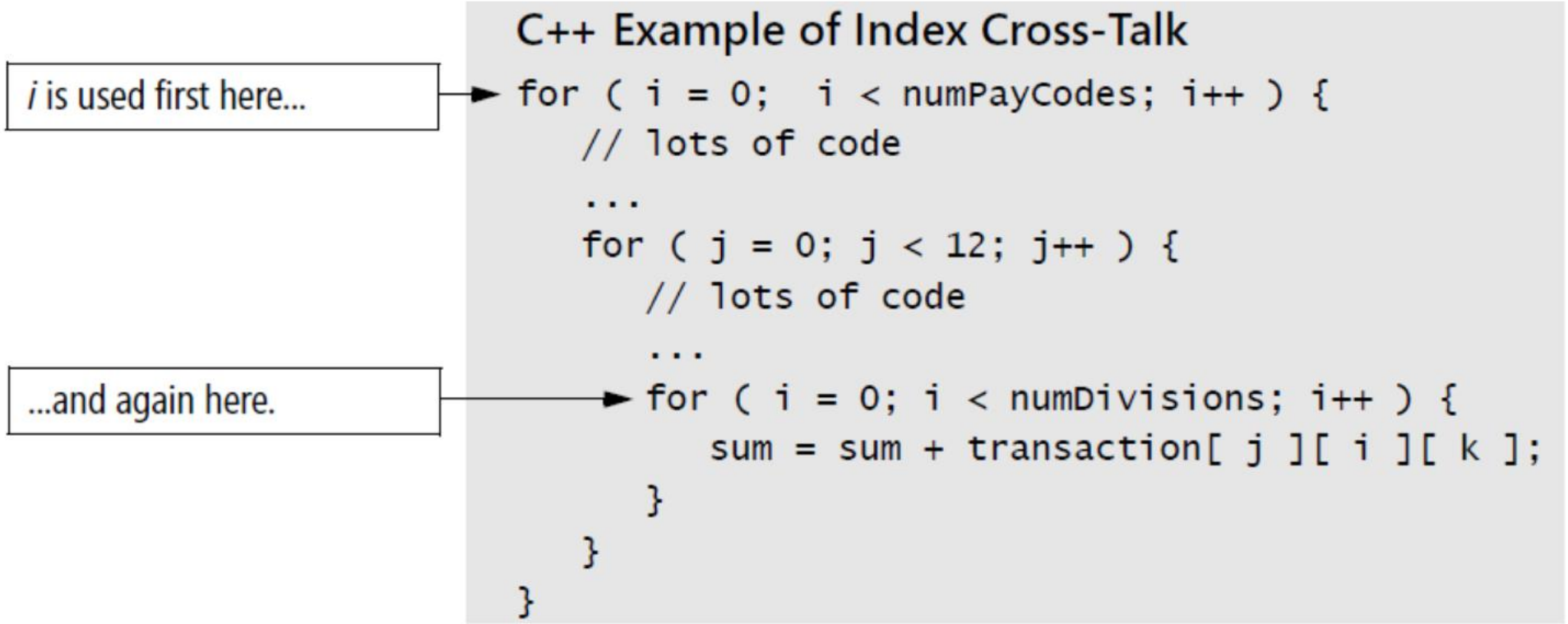
- ***Use meaningful variable names to make nested loops readable***
- What do you think the array indexes in *transaction* mean? Do *i*, *j*, and *k* tell you anything about the contents of *transaction*? If you had the declaration of *transaction*, could you easily determine whether the indexes were in the right order?

Java Example of Good Loop Variable Names

```
for ( int payCodeIdx = 0; payCodeIdx < numPayCodes; payCodeIdx++ ) {
   for (int month = 0; month < 12; month++ ) {
      for ( int divisionIdx = 0; divisionIdx < numDivisions; divisionIdx++ ) {
         sum = sum + transaction[ month ][ payCodeIdx ][ divisionIdx ];
      }
   }
}
```

Controlling the Loop – Loop Variables

- ***Use meaningful names to avoid loop-index cross-talk***
- Habitual use of *i*, *j*, and *k* can give rise to index cross-talk—using the same index name for two different purposes.



Controlling the Loop – Loop Variables

- ***Limit the scope of loop-index variables to the loop itself***
- Loop-index cross-talk and other uses of loop indexes outside their loops is such a significant problem that the designers of Ada decided to make *for* loop indexes invalid outside their loops; trying to use one outside its *for* loop generates an error at compile time.

C++ Example of Declaring a Loop-Index Variable Within a *for* loop

```
for ( int recordCount = 0; recordCount < MAX_RECORDS; recordCount++ ) {
   // looping code that uses recordCount
}
```

Controlling the Loop – Loop Variables

- ***Limit the scope of loop-index variables to the loop itself***
- In principle, this technique should allow creation of code that redeclares *recordCount* in multiple loops without any risk of misusing the two different *recordCounts*.

C++ Example of Declaring Loop-Indexes Within *for* loops and Reusing Them Safely—Maybe!

```
for ( int recordCount = 0; recordCount < MAX_RECORDS; recordCount++ ) {
   // looping code that uses recordCount
}
// intervening code
for ( int recordCount = 0; recordCount < MAX_RECORDS; recordCount++ ) {
   // additional looping code that uses a different recordCount
}
```

Controlling the Loop – Loop Variables

- ***Limit the scope of loop-index variables to the loop itself***
- This technique is helpful for documenting the purpose of the *recordCount* variable; *however, don't rely on your compiler to enforce recordCount's scope*.
- Compiler implementations can vary:
  - The first compiler flagged *recordCount* in the second *for* loop for multiple variable declarations and generated an error.
  - The second compiler accepted *recordCount* in the second *for* loop but allowed it to be used outside the first *for* loop.
  - The third compiler allowed both usages of *recordCount* and did not allow either one to be used outside the *for* loop in which it was declared.

Controlling the Loop – Loop Length

- Loop length can be measured in lines of code or depth of nesting.

```
public void printGrid()
{
    for ( int i = 0 ; i < height ; i++ )
    {
        // PRINT a row
        for ( int i = 0 ; i < width ; i++ )
        {
            System.out.print( "*" ) ;
        }
        // PRINT newline
        System.out.println( "" ) ;
    }
}
```

Controlling the Loop – Loop Length

- ***Make your loops short enough to view all at once***
- If you usually look at loops on your monitor and your monitor displays 50 lines, that puts a 50-line restriction on you.
- Experts have suggested a loop-length limit of one page. When you begin to appreciate the principle of writing simple code, however,
- *You'll rarely write loops longer than 15 or 20 lines*.

Controlling the Loop – Loop Length

- ***Limit nesting to three levels***
- Studies have shown that the ability of programmers to comprehend a loop deteriorates significantly beyond three levels of nesting (Yourdon 1986a).
- If you're going beyond that number of levels, make the loop shorter (conceptually) by breaking part of it into a routine or simplifying the control structure.

Controlling the Loop – Loop Length

- ***Move loop innards of long loops into routines***
- If the loop is well designed, the code on the inside of a loop can often be moved into one or more routines that are called from within the loop.

Controlling the Loop – Loop Length

- ***Make long loops especially clear***
- Length adds complexity.
- If you write a short loop, you can use riskier control structures such as *break* and *continue*, multiple exits, complicated termination conditions, and so on.
- If you write a longer loop and feel any concern for your reader, you'll give the loop a single exit and make the exit condition unmistakably clear.

Creating Loops Easily

- Suppose you're writing a program for an insurance company.
- It has life-insurance rates that vary according to a person's age and gender.
- Your job is to write a routine that computes the total life-insurance premium for a group.
- You need a loop that takes the rate for each person in a list and adds it to a total.

Creating Loops Easily

- **First**, in comments, write the steps the body of the loop needs to perform. It's easier to write down what needs to be done when you're not thinking about details of syntax, loop indexes, array indexes, and so on.

Step 1: Creating a Loop from the Inside Out (Pseudocode Example)

```
-- get rate from table
-- add rate to total
```

Creating Loops Easily

Creating Loops Easily

- Second, convert the comments in the body of the loop to code, as much as you can without actually writing the whole loop. In this case, get the rate for one person and add it to the overall total.
- Use concrete, specific data rather than abstractions.

table doesn't have any indexes yet.

Step 2: Creating a Loop from the Inside Out (Pseudocode Example)

```
rate = table[ ]
totalRate = totalRate + rate
```

- The example assumes that *table* is an array that holds the rate data. You don't have to worry about the array indexes at first. *rate* is the variable that holds the rate data selected from the rate table. Likewise, *totalRate* is a variable that holds the total of the rates.

Creating Loops Easily

- Next, put in indexes for the *table* array:

Step 3: Creating a Loop from the Inside Out (Pseudocode Example)

```
rate = table[ census.Age ][ census.Gender ]
totalRate = totalRate + rate
```

Creating Loops Easily

- The array is accessed by age and gender, so *census.Age* and *census.Gender* are used to index the array.
- The example assumes that *census* is a structure that holds information about people in the group to be rated.

Creating Loops Easily

- The next step is to build a loop around the existing statements.
- Since the loop is supposed to compute the rates for each person in a group, the loop should be indexed by person.

Step 4: Creating a Loop from the Inside Out (Pseudocode Example)

```
For person = firstPerson to lastPerson
   rate = table[ census.Age, census.Gender ]
   totalRate = totalRate + rate
End For
```

Creating Loops Easily

- All you have to do here is put the *for* loop around the existing code and then indent the existing code and put it inside a *begin-end* pair. Finally, check to make sure that the variables that depend on the *person* loop index have been generalized.
- In this case, the *census* variable varies with *person*, so it should be generalized appropriately. **Anything else?**

Step 5: Creating a Loop from the Inside Out (Pseudocode Example)

```
For person = firstPerson to lastPerson
   rate = table[ census[ person ].Age, census[ person ].Gender ]
   totalRate = totalRate + rate
End For
```

Creating Loops Easily

- Finally, write any initializations that are needed.
- In this case, the *totalRate* variable needs to be initialized.

Final Step: Creating a Loop from the Inside Out (Pseudocode Example)

```
totalRate = 0
For person = firstPerson to lastPerson
   rate = table[ census[ person ].Age, census[ person ].Gender ]
   totalRate = totalRate + rate
End For
```

Creating Loops Easily

- If you had to put another loop around the *person* loop, you would proceed in the same way.
- You don't need to follow the steps rigidly. The idea is to start with something concrete, worry about only one thing at a time, and build up the loop from simple components.
- Take small, understandable steps as you make the loop more general and complex.
- *That way, you minimize the amount of code you have to concentrate on at any one time and therefore minimize the chance of error*.

Unusual Control Structures

Outline

- Multiple Returns from a Routine
- Recursion

Multiple Returns from a Routine

- The *return* and *exit* statements are control constructs that enable a program to exit from a routine.

```
// a method for computing the area of the rectangle
public int getArea() {
    return width * height;
}
```

Multiple Returns from a Routine

- ***Use a* return *when it enhances readability***
- In certain routines, once you know the answer, you want to return it to the calling routine immediately.
- *If the routine is defined in such a way that it doesn't require any further cleanup*, not returning immediately means that you have to write more code.

Multiple Returns from a Routine

- ***Use a* return *when it enhances readability***

This routine returns a *Comparison* enumerated type.

C++ Example of a Good Multiple Return from a Routine

```
Comparison Compare( int value1, int value2 ) {
   if ( value1 < value2 ) {
      return Comparison_LessThan;
   }
   else if ( value1 > value2 ) {
      return Comparison_GreaterThan;
   }
   return Comparison_Equal;
}
```

Multiple Returns from a Routine

- ***Use guard clauses (early returns or exits) to simplify complex error processing***
- Code that has to check for numerous error conditions before performing its normal/main actions can result in ***deeply indented code*** and can make the main case difficult to understand, as shown on the next slide:

Multiple Returns from a Routine

- ***Use guard clauses (early returns or exits) to simplify complex error processing***

Visual Basic Code That Obscures the Nominal Case

```
If file.validName() Then
   If file.Open() Then
      If encryptionKey.valid() Then
         If file.Decrypt( encryptionKey ) Then
            ' lots of code
            ...
         End If
      End If
   End If
End If
```

This is the code for the nominal case.

Multiple Returns from a Routine

- ***Use guard clauses (early returns or exits) to simplify complex error processing***
- Indenting the main body of the routine inside four *if* statements is aesthetically ugly, especially if there's much code inside the innermost *if* statement.
- *In such cases, the flow of the code is sometimes clearer if the erroneous cases are checked first*, clearing the way for the main path through the code. Here's how that might look:

Multiple Returns from a Routine

- ***Use guard clauses (early returns or exits) to simplify complex error processing***

Simple Visual Basic Code That Uses Guard Clauses to Clarify the Nominal Case

```
' set up, bailing out if errors are found
If Not file.validName() Then Exit Sub
If Not file.Open() Then Exit Sub
If Not encryptionKey.valid() Then Exit Sub
If Not file.Decrypt( encryptionKey ) Then Exit Sub

' lots of code
...
```

Multiple Returns from a Routine

- ***Use guard clauses (early returns or exits) to simplify complex error processing***
- This simple code makes this technique look like a tidy solution, but production code often requires more extensive housekeeping or cleanup when an error condition is detected.
- Here is a more realistic example:

More Realistic Visual Basic Code That Uses Guard Clauses to Clarify the Nominal Case

```
' set up, bailing out if errors are found
If Not file.validName() Then
   errorStatus = FileError_InvalidFileName
   Exit Sub
End If

If Not file.Open() Then
   errorStatus = FileError_CantOpenFile
   Exit Sub
End If

If Not encryptionKey.valid() Then
   errorStatus = FileError_InvalidEncryptionKey
   Exit Sub
End If

If Not file.Decrypt( encryptionKey ) Then
   errorStatus = FileError_CantDecryptFile
   Exit Sub
End If

' lots of code
...
```

This is the code for the nominal case.

Multiple Returns from a Routine

- ***Use guard clauses (early returns or exits) to simplify complex error processing***
- With production-size code, the *Exit Sub* approach creates a noticeable amount of code before the main case is handled.
- The *Exit Sub* approach does avoid the deep nesting of the first example, and, if the code in the first example were expanded to show setting an *errorStatus* variable, the *Exit Sub* approach would do a better job of keeping related statements together.
- The *Exit Sub* approach does appear more readable and maintainable, just not by a very wide margin.

Multiple Returns from a Routine

- ***Minimize the number of returns in each routine***
- It's harder to understand a routine when, reading it at the bottom, you're unaware of the possibility that it returned somewhere above.
- For that reason, use returns only when they improve readability.

Recursion

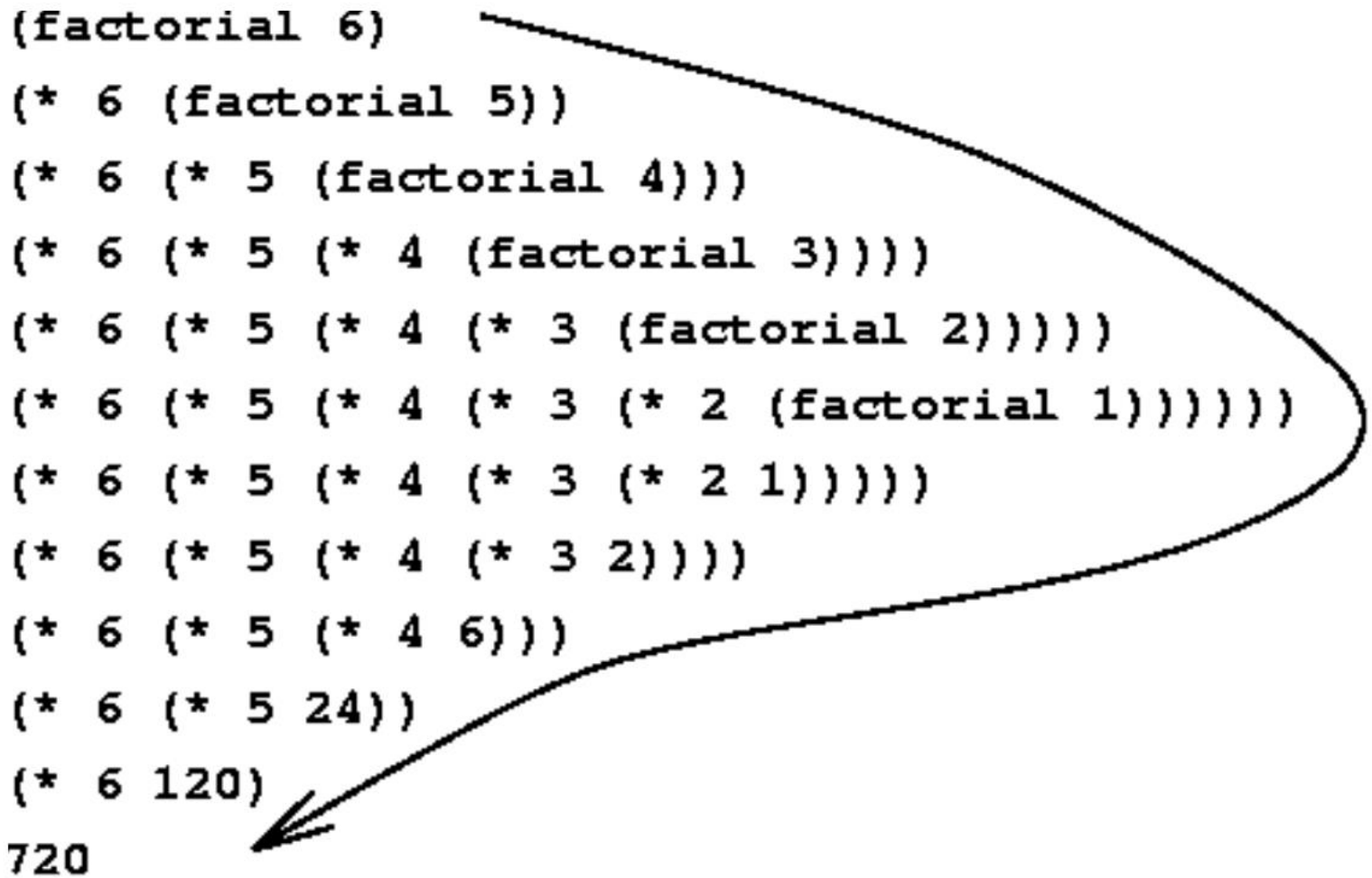
- In recursion, a routine solves a small part of a problem itself, divides the problem into smaller pieces, and then calls itself to solve each of the smaller pieces.
- Recursion is usually called into play when a small part of the **problem is easy to solve** and a **large part is easy to decompose** into smaller pieces.

```java
// Fig. 18.3: FactorialCalculator.java
// Recursive factorial method.

public class FactorialCalculator
{
   // recursive method factorial (assumes its parameter is >= 0)
   public long factorial( long number )
   {
      if ( number <= 1 ) // test for base case
         return 1; // base cases: 0! = 1 and 1! = 1
      else // recursion step
         return number * factorial( number - 1 );
   } // end method factorial

   // output factorials for values 0-21
   public static void main( String[] args )
   {
      // calculate the factorials of 0 through 21
      for ( int counter = 0; counter <= 21; counter++ )
         System.out.printf( "%d! = %d\n", counter, factorial( counter ) );
   } // end main
} // end class FactorialCalculator
```

Recursive method call solves simpler problem

Nonrecursive method call

Recursion


(factorial 6)
(* 6 (factorial 5))
(* 6 (* 5 (factorial 4)))
(* 6 (* 5 (* 4 (factorial 3))))
(* 6 (* 5 (* 4 (* 3 (factorial 2)))))
(* 6 (* 5 (* 4 (* 3 (* 2 (factorial 1))))))
(* 6 (* 5 (* 4 (* 3 (* 2 1)))))
(* 6 (* 5 (* 4 (* 3 2))))
(* 6 (* 5 (* 4 6)))
(* 6 (* 5 24))
(* 6 120)
720

Recursion

- Recursion isn't useful often (**WHY?**), but when used it can produce elegant solutions:
  - performance and memory issues, also complexity problems

Here are the recursive calls.

Java Example of a Sorting Algorithm That Uses Recursion

```
void QuickSort( int firstIndex, int lastIndex, String [] names ) {
   if ( lastIndex > firstIndex ) {
      int midPoint = Partition( firstIndex, lastIndex, names );
      QuickSort( firstIndex, midPoint-1, names );
      QuickSort( midPoint+1, lastIndex, names )
   }
}
```

Recursion

- For a small group of problems, recursion can produce simple, elegant solutions.
- For a slightly larger group of problems, it can produce simple, elegant, yet hard-to-understand solutions.
- For most problems, it produces massively complicated solutions—in those cases, simple iteration is usually more understandable.
- Use recursion selectively.

Recursion

- ***Make sure the recursion stops***
- Check the routine to make sure that it includes a nonrecursive path. That usually means that the routine has a test that stops further recursion when it's not needed.
- *For example*, *AlreadyTried()* and *ThisIsTheExit()* ensure that the recursion stops.

Recursion

- ***Use safety counters to prevent infinite recursion***
- If you're using recursion in a situation that doesn't allow a simple test, use a safety counter to prevent infinite recursion.
- The safety counter has to be a variable that's not re-created each time you call the routine. Use a class member variable or pass the safety counter as a parameter. Here's an example:

Recursion

- ***Use safety counters to prevent infinite recursion***

The recursive routine must be able to change the value of *safetyCounter*, so in Visual Basic it's a *ByRef* parameter.

Visual Basic Example of Using a Safety Counter to Prevent Infinite Recursion

```
Public Sub RecursiveProc( ByRef safetyCounter As Integer )
   If ( safetyCounter > SAFETY_LIMIT ) Then
      Exit Sub
   End If
   safetyCounter = safetyCounter + 1
   ...
   RecursiveProc( safetyCounter )
End Sub
```

Recursion

- ***Limit recursion to one routine***
- Cyclic recursion (A calls B calls C calls A) is dangerous because it's hard to detect.
- Mentally managing recursion in one routine is tough enough; understanding recursion that spans routines is too much.
- If you have cyclic recursion, you can usually redesign the routines so that the recursion is restricted to a single routine. If you can't and you still think that recursion is the best approach, use safety counters as a recursive insurance policy.

Recursion

- ***Keep an eye on the stack***
- With recursion, you have no guarantees about how much stack space your program uses and it's hard to predict in advance how the program will behave at run time.
- You can take a couple of steps to control its run-time behavior, however.
  - First, if you use a safety counter, one of the considerations in setting a limit for it should be how much stack you're willing to allocate to the recursive routine.
  - Second, watch for allocation of local variables in recursive functions, especially memory-intensive objects. In other words, use *new* to create objects on the heap rather than letting the compiler create *auto* objects on the stack.

Recursion

- ***Don't use recursion for factorials or Fibonacci numbers***
- The typical examples are computing a factorial or computing a Fibonacci sequence (1,1,2,3,5,8,13,…). Recursion is a powerful tool, yet not very practical in either of those cases.

Java Example of an Inappropriate Solution: Using Recursion to Compute a Factorial

```
int Factorial( int number ) {
   if ( number == 1 ) {
      return 1;
   }
   else {
      return number * Factorial( number - 1 );
   }
}
```

Recursion

- ***Don't use recursion for factorials or Fibonacci numbers***
- In addition to being slow and making the use of run-time memory unpredictable, the **recursive** version of this routine is harder to understand than the **iterative** version, which follows:

Java Example of an Appropriate Solution: Using Iteration to Compute a Factorial

```
int Factorial( int number ) {
   int intermediateResult = 1;
   for ( int factor = 2; factor <= number; factor++ ) {
      intermediateResult = intermediateResult * factor;
   }
   return intermediateResult;
}
```

General Control Issues

Outline

- Boolean Expressions
- Deep Nesting

Boolean Expressions

- *Use the identifiers true and false in boolean expressions rather than using values like 0 and 1*.
- Most modern languages have a boolean data type and provide predefined identifiers for true and false. They make it easy—they don't even allow you to assign values other than *true* or *false* to boolean variables.
- Languages that don't have a Boolean data type require you to have more discipline to make boolean expressions readable.
- Here's an example of the problem:

Boolean Expressions

Visual Basic Examples of Using Ambiguous Flags for Boolean Values

```
Dim printerError As Integer
Dim reportSelected As Integer
Dim summarySelected As Integer
...
If printerError = 0 Then InitializePrinter()
If printerError = 1 Then NotifyUserOfError()

If reportSelected = 1 Then PrintReport()
If summarySelected = 1 Then PrintSummary()

If printerError = 0 Then CleanupPrinter()
```

Boolean Expressions

- *What's wrong with using 0-1 as Boolean?*

  It's not clear from reading the code whether the function calls are executed when the tests are true or when they're false.

- What is reportSelected = 1?
  - Nothing in the code fragment itself tells you whether *1* represents true and *0* false or whether the opposite is true. It's not even clear that the values *1* and *0* are being used to represent true and false.
  - ***For example***, in the *If reportSelected = 1* line, the *1* could easily represent the first report, a *2* the second, a *3* the third; nothing in the code tells you that *1* represents either true or false. It's also easy to write *0* when you mean *1* andvice versa.

Boolean Expressions

- Use terms named *true* and *false* for tests with boolean expressions. The previous code example is rewritten here using Visual Basic's built-in *True* and *False*:

Good, but Not Great Visual Basic Examples of Using *True* and *False* for Tests Instead of Numeric Values

```
Dim printerError As Boolean
Dim reportSelected As ReportType
Dim summarySelected As Boolean
...
If ( printerError = False ) Then InitializePrinter()
If ( printerError = True ) Then NotifyUserOfError()

If ( reportSelected = ReportType_First ) Then PrintReport()
If ( summarySelected = True ) Then PrintSummary()

If ( printerError = False ) Then CleanupPrinter()
```

Boolean Expressions

- Use of the *True* and *False* constants makes the intent clearer.
- You don't have to remember what *1* and *0* represent, and you won't accidentally reverse them.
- Moreover, in the rewritten code, it's now clear that some of the *1*s and *0*s in the original Visual Basic example weren't being used as boolean flags.
- The *If reportSelected = 1* line was not a boolean test at all; it tested whether the first report had been selected.

Boolean Expressions

- This approach tells the reader that you're making a boolean test.
- It's also harder to write *true* when you mean *false* than it is to write *1* when you mean *0*, and you avoid spreading the magic numbers *0* and *1* throughout your code.
- Here are some tips on defining *true* and *false* in boolean tests:

Boolean Expressions

- ***Compare boolean values to* true *and* false *implicitly***
- You can write clearer tests by treating the expressions as Boolean expressions. For example, write

```
while ( not done ) ...
while ( a > b ) ...
```

rather than

```
while ( done = false ) ...
while ( (a > b) = true ) ...
```

Boolean Expressions

- ***Compare boolean values to* true *and* false *implicitly***
- Using implicit comparisons reduces the number of terms that someone reading your code has to keep in mind, and the resulting expressions read more like conversational English.
- The previous example could be rewritten with even better style like this:

Boolean Expressions

Better Visual Basic Examples of Testing for *True* and *False* Implicitly

```
Dim printerError As Boolean
Dim reportSelected As ReportType
Dim summarySelected As Boolean
...
If ( Not printerError ) Then InitializePrinter()
If ( printerError ) Then NotifyUserOfError()

If ( reportSelected = ReportType_First ) Then PrintReport()
If ( summarySelected ) Then PrintSummary()

If ( Not printerError ) Then CleanupPrinter()
```

Boolean Expressions

- ***Compare boolean values to* true *and* false *implicitly***
- If your language doesn't support boolean variables and you have to emulate them, you might not be able to use this technique because emulations of *true* and *false* can't always be tested with statements like *while (not done)*.

Boolean Expressions

- ***Break complicated tests into partial tests with new Boolean variables***
- Rather than creating a monstrous test with half a dozen terms, assign intermediate values to terms that allow you to perform a simpler test.

Boolean Expressions

- ***Move complicated expressions into boolean functions***
- If a test is repeated often or distracts from the main flow of the program, move the code for the test into a function and test the value of the function. ***For example***, here's a complicated test:

Visual Basic Example of a Complicated Test

```
If ( ( document.AtEndOfStream ) And ( Not inputError ) ) And _
   ( ( MIN_LINES <= lineCount ) And ( lineCount <= MAX_LINES ) ) And _
   ( Not ErrorProcessing(  ) ) Then
   ' do something or other
   ...
End If
```

Boolean Expressions

- ***Move complicated expressions into boolean functions***
- This is an ugly test to have to read through if you're not interested in the test itself.
- By putting it into a boolean function, you can isolate the test and allow the reader to forget about it unless it's important.
- Here's how you could put the *if* test into a function:

Boolean Expressions

Visual Basic Example of a Complicated Test Moved into a Boolean Function, with New Intermediate Variables to Make the Test Clearer

```
Function DocumentIsValid( _
   ByRef documentToCheck As Document, _
   lineCount As Integer, _
   inputError As Boolean _
   ) As Boolean

   Dim allDataRead As Boolean
   Dim legalLineCount As Boolean

   allDataRead = ( documentToCheck.AtEndOfStream ) And ( Not inputError )
   legalLineCount = ( MIN_LINES <= lineCount ) And ( lineCount <= MAX_LINES )
   DocumentIsValid = allDataRead And legalLineCount And ( Not ErrorProcessing() )

End Function
```

Intermediate variables are introduced here to clarify the test on the final line, below.

Boolean Expressions

- ***Move complicated expressions into boolean functions***
- This example assumes that *ErrorProcessing()* is a boolean function that indicates the current processing status.
- Now, when you read through the main flow of the code, you don't have to read the complicated test:

Visual Basic Example of the Main Flow of the Code Without the Complicated Test

```
If ( DocumentIsValid( document, lineCount, inputError ) ) Then
    ' do something or other
    ...
End If
```

Boolean Expressions

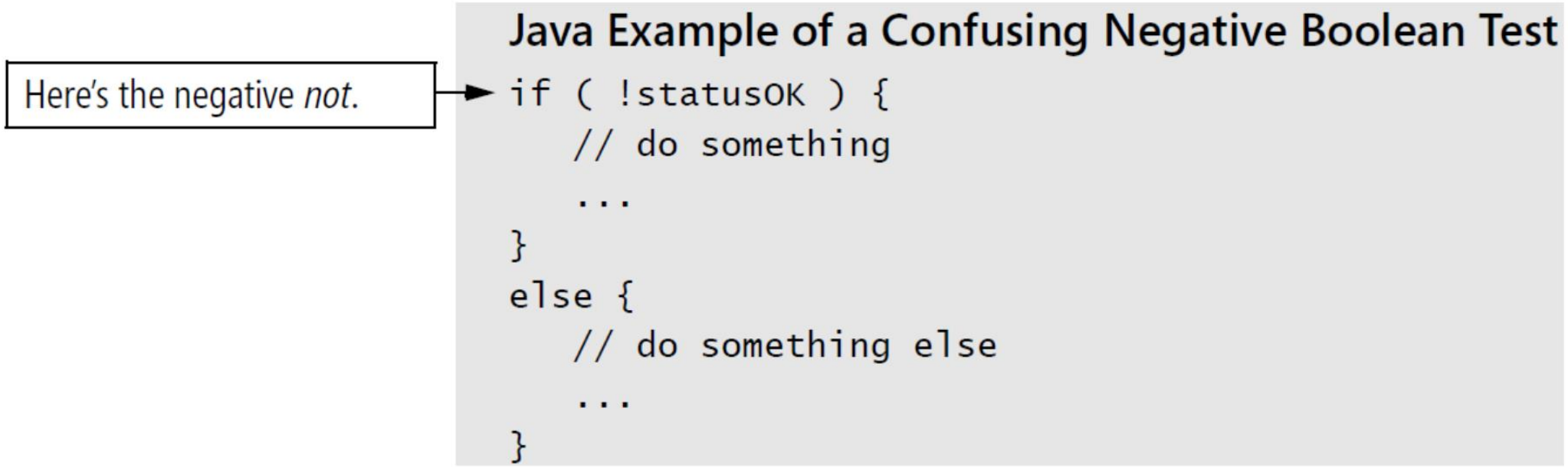
- ***Move complicated expressions into boolean functions***
- If you use the test only once, you might not think it's worthwhile to put it into a routine.
- But putting the test into a well-named function improves readability and makes it easier for you to see what your code is doing, and that's a sufficient reason to do it.
- The new function name introduces an abstraction into the program that documents the purpose of the test *in code*. That's even better than documenting the test with comments because the code is more likely to be read than the comments and it's more likely to be kept up to date, too.

Boolean Expressions

- ***Use decision tables to replace complicated conditions***
- Sometimes you have a complicated test involving several variables. It can be helpful to use a decision **table** to perform the test rather than using *if*s or *case*s.
- A decision-table lookup is easier to code initially, having only a couple of lines of code and no tricky control structures.
- *This minimization of complexity minimizes the opportunity for mistakes*. If your data changes, you can change a decision table without changing the code; you only need to update the contents of the data structure.

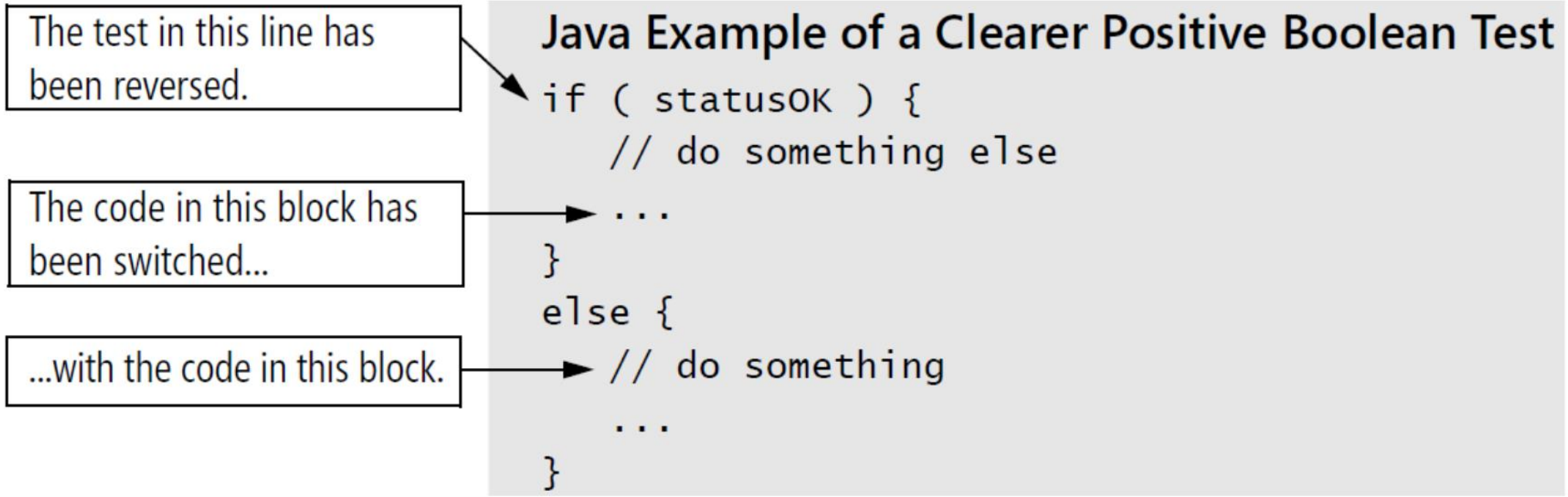
Boolean Expressions

- ***In* if *statements, convert negatives to positives and flip-flop the code in the* if *and* else *clauses***
- Here's an example of a negatively expressed test:



Boolean Expressions

- ***In* if *statements, convert negatives to positives and flip-flop the code in the* if *and* else *clauses***
- You can change this to the following positively expressed test:



Boolean Expressions

- ***Apply DeMorgan's Theorems to simplify boolean tests with negatives***
- DeMorgan's Theorems let you exploit the logical relationship between an expression and a version of the expression that means the same thing because it's doubly negated.
- ***For example***, you might have a code fragment that contains the following test:

Java Example of a Negative Test

```
if ( !displayOK || !printerOK ) ...
```

Boolean Expressions

- ***Apply DeMorgan's Theorems to simplify boolean tests with negatives***
- For example, you might have a code fragment that contains the following test:

Java Example of a Negative Test

```
if ( !displayOK || !printerOK ) ...
```

- This is logically equivalent to the following:

Java Example After Applying DeMorgan's Theorems

```
if ( !( displayOK && printerOK ) ) ...
```

Boolean Expressions

- ***Apply DeMorgan's Theorems to simplify boolean tests with negatives***
- Here you don't have to flip-flop *if* and *else* clauses; the expressions in the last two code fragments are logically equivalent.
- To apply DeMorgan's Theorems to the logical operator *and* or the logical operator *or* and a pair of operands, you negate each of the operands, switch the *and*s and *or*s, and negate the entire expression.

Boolean Expressions

Table 19-1 Transformations of Logical Expressions Under DeMorgan's Theorems

Initial Expression	Equivalent Expression
not A and not B	not (A or B)
not A and B	not (A or not B)
A and not B	not (not A or B)
A and B	not (not A or not B)
not A or not B*	not (A and B)
not A or B	not (A and not B)
A or not B	not (not A and B)
A or B	not (not A and not B)

Boolean Expressions

- If you have a complicated boolean expression, rather than relying on the language's evaluation order, parenthesize to make your meaning clear.
- Using parentheses makes less of a demand on your reader, who might not understand the subtleties of how your language evaluates boolean expressions.
- If you're smart, you won't depend on your own or your reader's in-depth memorization of evaluation precedence—especially when you have to switch among two or more languages.

Boolean Expressions

- Here's an expression with too few parentheses:

Java Example of an Expression Containing Too Few Parentheses

```
if ( a < b == c == d ) ...
```

- This is a confusing expression to begin with, and it's even more confusing because it's not clear whether the coder means to test *(a < b) == (c == d)* or *((a < b) == c) == d*.
- The following version of the expression is still a little confusing, but the parentheses help:

Java Example of an Expression Better Parenthesized

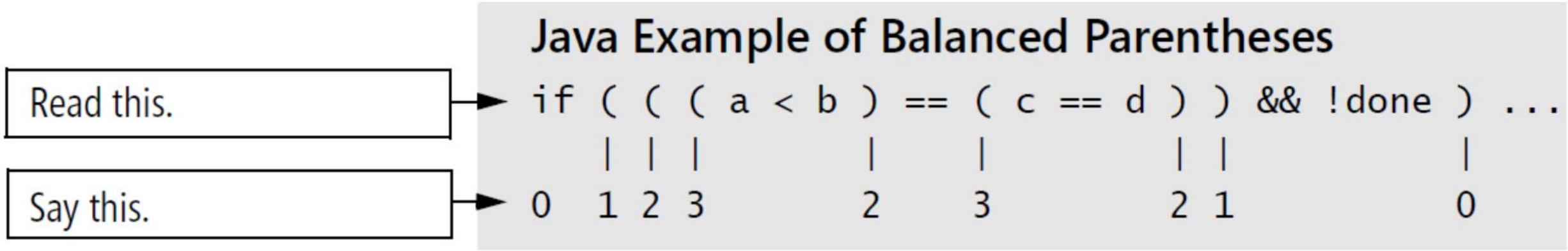
```
if ( ( a < b ) == ( c == d ) ) ...
```

Boolean Expressions

- ***Use a simple counting technique to balance parentheses***
- *If you have trouble telling whether parentheses balance, here's a* ~~*simple counting trick that helps*~~.
- Start by saying "zero."
- Move along the expression, left to right. When you encounter an opening parenthesis, say "one."
- Each time you encounter another opening parenthesis, increase the number you say. Each time you encounter a closing parenthesis, decrease the number you say. If, at the end of the expression, you're back to 0, your parentheses are balanced.

Boolean Expressions

- ***Use a simple counting technique to balance parentheses***
- *If you have trouble telling whether parentheses balance, here's a simple counting trick that helps*.



Boolean Expressions – How Evaluated

- Compilers for some languages evaluate each term in a Boolean expression before combining the terms and evaluating the whole expression.
- Compilers for other languages have “short-circuit” or “lazy” evaluation, evaluating only the pieces necessary.
- This is particularly significant when, depending on the results of the first test, you might not want the second test to be executed.

Boolean Expressions – How Evaluated

- ***For example***, suppose you're checking the elements of an array and you ~~have the follo~~wing test:

Pseudocode Example of an Erroneous Test

```
while ( i < MAX_ELEMENTS and item[ i ] <> 0 ) ...
```

- If this whole expression is evaluated, you'll get an error on the last pass through the loop. The variable *i* equals *maxElements*, so the expression *item[i]* is equivalent to *item[maxElements]*, which is an array-index error.
- You might argue that it doesn't matter since you're only looking at the value, not changing it. But it's sloppy programming practice and could confuse someone reading the code. In many environments it will also generate either a run-time error or a protection violation.

Boolean Expressions – How Evaluated

- In *pseudocode*, you could restructure the test so that the error doesn't occur:

  This is correct because *item[i]* isn't evaluated unless *i* is less than *maxElements*

Pseudocode Example of a Correctly Restructured Test

```
while ( i < MAX_ELEMENTS )
   if ( item[ i ] <> 0 ) then
      ...
```

Boolean Expressions – How Evaluated

- Many modern languages provide facilities that prevent this kind of error from happening in the first place.
- ***For example***, C++ uses short-circuit evaluation: if the first operand of the *and* is false, the second isn't evaluated because the whole expression would be false anyway. In other words, in C++ the only part of

```
if ( somethingFalse && someCondition ) ...
```

- that's evaluated is *SomethingFalse*. Evaluation stops as soon as *SomethingFalse* is identified as false.

Boolean Expressions – How Evaluated

- Evaluation is similarly short-circuited with the *or* operator. In C++ and Java, the only part of

```
if ( somethingTrue || someCondition ) ...
```

- that is evaluated is *somethingTrue*. The evaluation stops as soon as *somethingTrue* is identified as true because the expression is always true if any part of it is true. As a result of this method of evaluation, the following statement is a fine, legal statement.

Java Example of a Test That Works Because of Short-Circuit Evaluation

```
if ( ( denominator != 0 ) && ( ( item / denominator ) > MIN_VALUE ) ) ...
```

Boolean Expressions – How Evaluated

- Any problem?

```
if ( ( ( item / denominator ) > MIN_VALUE ) && ( denominator != 0 ) ) ...
```

- In this case, *item / denominator* is evaluated before *denominator != 0*. Consequently, this code commits the divide-by-zero error.

Boolean Expressions – How Evaluated

- Java further **complicates** this picture by providing **"logical" operators**. Java's logical & and | operators guarantee that all terms will be fully evaluated regardless of whether the truth or falsity of the expression could be determined without a full evaluation. In other words, in Java, this is safe:

Java Example of a Test That Works Because of Short-Circuit (Conditional) Evaluation

```
if ( ( denominator != 0 ) && ( ( item / denominator ) > MIN_VALUE ) ) ...
```

- But this is not safe:

Java Example of a Test That Doesn't Work Because Short-Circuit Evaluation Isn't Guaranteed

```
if ( ( denominator != 0 ) & ( ( item / denominator ) > MIN_VALUE ) ) ...
```

Boolean Expressions – How Evaluated

- Bitwise operators, not short-circuit for Boolean variables

```
int a = 60;          /* 60 = 0011 1100 */
int b = 13;          /* 13 = 0000 1101 */
int c = 0;

c = a & b;           /* 12 = 0000 1100 */
c = a | b;           /* 61 = 0011 1101 */
```

Boolean Expressions – Number Line Order

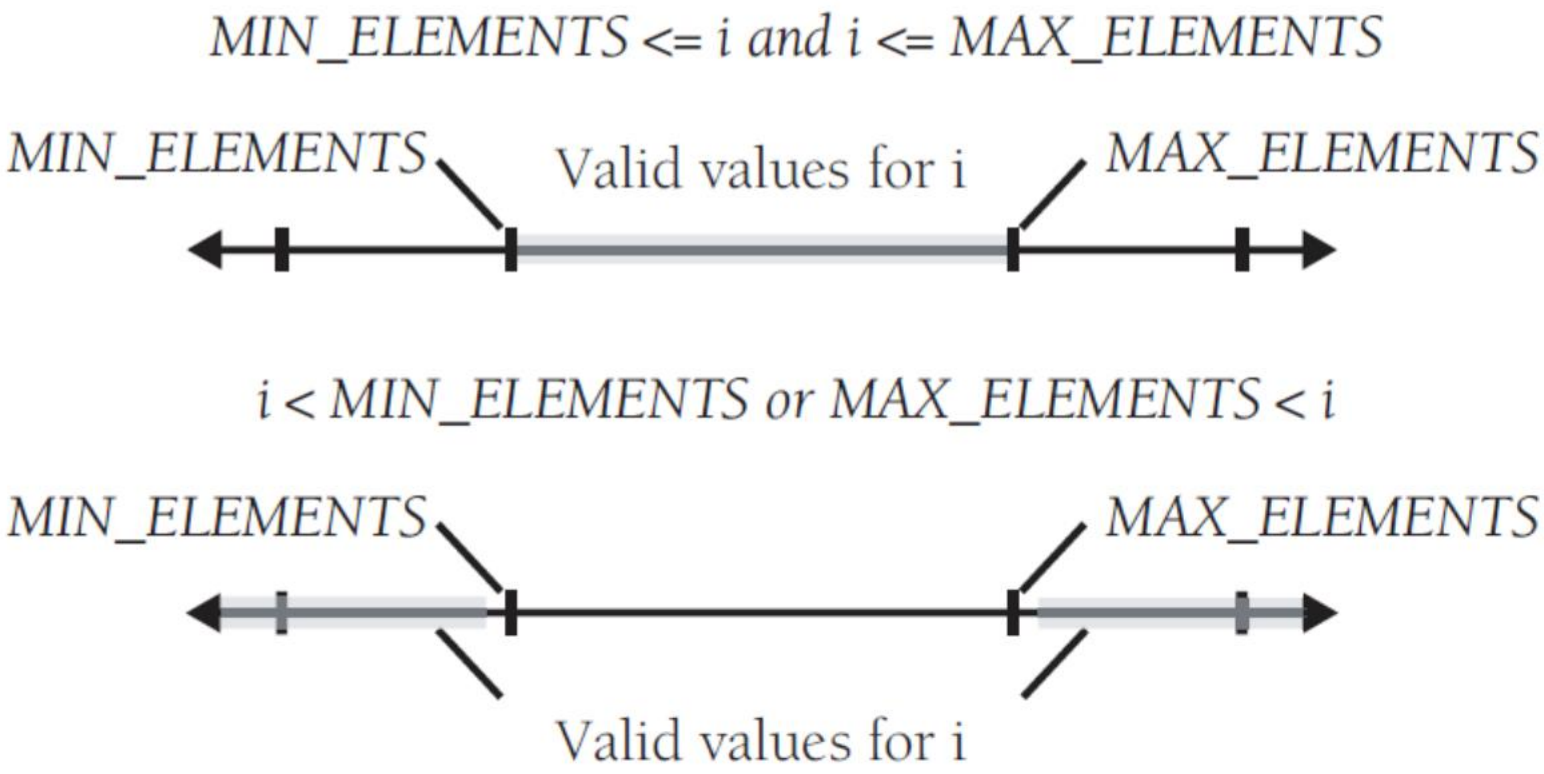
- Organize numeric tests so that they follow the points on a number line. In general, structure your numeric tests so that you have comparisons like these:

```
MIN_ELEMENTS <= i and i <= MAX_ELEMENTS
i < MIN_ELEMENTS or MAX_ELEMENTS < i
```

- The idea is to order the elements left to right, from smallest to largest. In the ***first example***, *MIN_ELEMENTS* and *MAX_ELEMENTS* are the two end~~points, so th~~ey go at the ends.
- The variable *i* is supposed to be between them, so it goes in the middle. In the ***second example***, you're testing whether *i* is outside the range, so *i* goes on t~~he outside of th~~e test at either end and *MIN_ELEMENTS* and *MAX_ELEMENTS* go on the inside.

Boolean Expressions – Number Line Order

- This approach maps easily to a visual image of the comparison



Boolean Expressions – Comparisons to Zero

- Programming languages use *0* for several purposes.
  - It's a numeric value.
  - It's a null terminator in a string.
  - It's the value of a null pointer.
  - It's the value of the first item in an enumeration.
  - It's *false* in logical expressions.
- Because it's used for so many purposes, you should write code that highlights the specific way *0* is used

Boolean Expressions – Comparisons to Zero

- ***Compare logical variables implicitly***
- As mentioned earlier, it's appropriate to write logical expressions such as

```
while(!done) ...
```

- This implicit comparison to *0* is appropriate because the comparison is in a logical expression.

Boolean Expressions – Comparisons to Zero

- ***Compare numbers to* 0**
- Although it's appropriate to compare logical expressions implicitly, you should compare numeric expressions explicitly. For numbers, write

```
while ( balance != 0 ) ...
```

rather than

```
while ( balance ) ...
```

Boolean Expressions – Comparisons to Zero

- ***Compare characters to the null terminator ('\0') explicitly in C***
- Characters, like numbers, aren't logical expressions. Thus, for characters, write

```
while ( *charPtr != '\0' ) ...
```

rather than

```
while ( *charPtr ) ...
```

Boolean Expressions – Comparisons to Zero

- ***Compare pointers to NULL***
- For pointers, write

```
while ( bufferPtr != NULL ) ...
```

rather than

```
while ( bufferPtr ) ...
```

Deep Nesting

- ***Simplify a nested if by retesting part of the condition***
- If the nesting gets too deep, you can decrease the number of nesting levels by retesting some of the conditions.
- This code example has nesting that's deep enough to warrant restructuring:

C++ Example of Bad, Deeply Nested Code

```
if ( inputStatus == InputStatus_Success ) {
   // lots of code
   ...
   if ( printerRoutine != NULL ) {
   // lots of code
   ...
   if ( SetupPage() ) {
      // lots of code
      ...
      if ( AllocMem( &printData ) ) {
         // lots of code
         ...
      }
   }
}
```

Deep Nesting

- ***Simplify a nested if by retesting part of the condition***
- Here's the code revised to use retesting rather than nesting:
- This is a particularly realistic example because it shows that you can't reduce the nesting level for free; you have to put up with a ***more complicate*** test in return for the ~~reduced level of nesting~~.
- A reduction from four levels to two is a big improvement in readability, and is worth considering.

C++ Example of Code Mercifully Unnested by Retesting

```
if ( inputStatus == InputStatus_Success ) {
   // lots of code
   ...
   if ( printerRoutine != NULL ) {
      // lots of code
      ...
   }
}

if ( ( inputStatus == InputStatus_Success ) &&
   ( printerRoutine != NULL ) && SetupPage() ) {
   // lots of code
   ...
   if ( AllocMem( &printData ) ) {
      // lots of code
      ...
   }
}
```

Deep Nesting

- ***Simplify a nested* if *by using a* break *block***
- *An alternative to the approach just described is to define a section of code that will be executed as a block*.
- If some condition in the middle of the block fails, execution skips to the end of the block.
- This technique is uncommon that it should be used only when your entire team is familiar with it and when it has been adopted by the team as an accepted coding practice.

C++ Example of Using a *break* Block

```
do {
   // begin break block
   if ( inputStatus != InputStatus_Success ) {
      break; // break out of block
   }
   // lots of code
   ...
   if ( printerRoutine == NULL ) {
      break; // break out of block
   }

   // lots of code
   ...
   if ( !SetupPage() ) {
      break; // break out of block
   }

   // lots of code
   ...
   if ( !AllocMem( &printData ) ) {
      break; // break out of block
   }

   // lots of code
   ...
} while (FALSE); // end break block
```

Deep Nesting

- Any Comments?

```
if ( 10 < quantity ) {
   if ( 100 < quantity ) {
      if ( 1000 < quantity ) {
         discount = 0.10;
      }
      else {
         discount = 0.05;
      }
   }
   else {
      discount = 0.025;
   }
}
else {
   discount = 0.0;
}
```

Deep Nesting

- ***Convert a nested* if *to a set of* if-then-elses**
- If you think about a nested *if* test critically, you might discover that you can reorganize it so that it uses *if-then-else*s rather than nested *if*s. Suppose you have a decision tree like this:
- This test is poorly organized in several ways, one of which is that the tests are redundant.
- When you test whether *quantity* is greater than *1000*, you don't also need to test whether it's greater than *100* and greater than *10*. Consequently, you can reorganize the code:

Java Example of an Overgrown Decision Tree

```
if ( 10 < quantity ) {
   if ( 100 < quantity ) {
      if ( 1000 < quantity ) {
         discount = 0.10;
      }
      else {
         discount = 0.05;
      }
   }
   else {
      discount = 0.025;
   }
}
else {
   discount = 0.0;
}
```

Deep Nesting

- ***Convert a nested* if *to a set of* if-then-elses**

Java Example of a Nested *if* Converted to a Set of *if-then-elses*

```
if ( 1000 < quantity ) {
   discount = 0.10;
}
else if ( 100 < quantity ) {
   discount = 0.05;
}
else if ( 10 < quantity ) {
   discount = 0.025;
}
else {
   discount = 0;
}
```

Deep Nesting

- ***Convert a nested* if *to a set of* if-then-elses**
- Difference to the previous one?

```
if ( 1000 < quantity ) {
   discount = 0.10;
}
else if ( ( 100 < quantity ) && ( quantity <= 1000 ) ) {
   discount = 0.05;
}
else if ( ( 10 < quantity ) && ( quantity <= 100 ) ) {
   discount = 0.025;
}
else if ( quantity <= 10 ) {
   discount = 0;
}
```

Deep Nesting

- ***Convert a nested* if *to a* case *statement***
- The main difference between this code and the previous code is that the expressions in the *else-if* clauses don't rely on previous tests.
- This code doesn't need the *else* clauses to work, and the tests actually could be performed in any order.
- The code could consist of four *if*s and no *else*s. The only reason the *else* version is preferable is that it avoids repeating tests unnecessarily.

Deep Nesting

- ***Convert a nested* if *to a* case *statement***
- You can recode some kinds of tests, particularly those with integers, to use a *case* statement rather than chains of *if*s and *else*s.
- You can't use this technique in some languages, but it's a powerful technique for those in which you can.
- The example in Visual Basic:

```
Select Case quantity
   Case 0 To 10
      discount = 0.0
   Case 11 To 100
      discount = 0.025
   Case 101 To 1000
      discount = 0.05
   Case Else
      discount = 0.10
End Select
```

Deep Nesting

- ***Factor deeply nested code into its own routine***
- If deep nesting occurs inside a loop, you can often improve the situation by putting the inside of the loop into its own routine.
- This is especially effective if the nesting is a result of both conditionals and iterations.
- **Leave the *if-then-else* branches in the main loop** to show the decision branching, and then move the statements within the branches to their own routines.
- The code on the next slide needs to be improved by such a modification:

C++ Example of Nested Code That Needs to Be Broken into Routines

```
while ( !TransactionsComplete() ) {
   // read transaction record
   transaction = ReadTransaction();

   // process transaction depending on type of transaction
   if ( transaction.Type == TransactionType_Deposit ) {
      // process a deposit
      if ( transaction.AccountType == AccountType_Checking ) {
         if ( transaction.AccountSubType == AccountSubType_Business )
            MakeBusinessCheckDep( transaction.AccountNum, transaction.Amount );
         else if ( transaction.AccountSubType == AccountSubType_Personal )
            MakePersonalCheckDep( transaction.AccountNum, transaction.Amount );
         else if ( transaction.AccountSubType == AccountSubType_School )
            MakeSchoolCheckDep( transaction.AccountNum, transaction.Amount );
      }
      else if ( transaction.AccountType == AccountType_Savings )
         MakeSavingsDep( transaction.AccountNum, transaction.Amount );
      else if ( transaction.AccountType == AccountType_DebitCard )
         MakeDebitCardDep( transaction.AccountNum, transaction.Amount );
      else if ( transaction.AccountType == AccountType_MoneyMarket )
         MakeMoneyMarketDep( transaction.AccountNum, transaction.Amount );
      else if ( transaction.AccountType == AccountType_Cd )
         MakeCDDep( transaction.AccountNum, transaction.Amount );
   }
   else if ( transaction.Type == TransactionType_Withdrawal ) {
      // process a withdrawal
      if ( transaction.AccountType == AccountType_Checking )
         MakeCheckingWithdrawal( transaction.AccountNum, transaction.Amount );
```

Deep Nesting

- ***Factor deeply nested code into its own routine***
- Although it's complicated, this isn't the worst code you'll ever see.
- In spite of its adequacy, however, you can improve it by breaking the contents of the inner *if* tests into their own routines.

C++ Example of Good, Nested Code After Decomposition into Routines

```
while ( !TransactionsComplete() ) {
   // read transaction record
   transaction = ReadTransaction();

   // process transaction depending on type of transaction
   if ( transaction.Type == TransactionType_Deposit ) {
      ProcessDeposit(
         transaction.AccountType,
         transaction.AccountSubType,
         transaction.AccountNum,
         transaction.Amount
      );
   }
   else if ( transaction.Type == TransactionType_Withdrawal ) {
      ProcessWithdrawal(
         transaction.AccountType,
         transaction.AccountNum,
         transaction.Amount
      );
   }
   else if ( transaction.Type == TransactionType_Transfer ) {
      MakeFundsTransfer(
         transaction.SourceAccountType,
         transaction.TargetAccountType,
         transaction.AccountNum,
         transaction.Amount
      );
   }
```

Deep Nesting

- Easier to read:
- **Even better**: try an object-oriented approach

C++ Example of Good, Nested Code After Decomposition and Use of a *case* Statement

```
while ( !TransactionsComplete() ) {
   // read transaction record
   transaction = ReadTransaction();

   // process transaction depending on type of transaction
   switch ( transaction.Type ) {
      case ( TransactionType_Deposit ):
         ProcessDeposit(
            transaction.AccountType,
            transaction.AccountSubType,
            transaction.AccountNum,
            transaction.Amount
            );
         break;

      case ( TransactionType_Withdrawal ):
         ProcessWithdrawal(
            transaction.AccountType,
            transaction.AccountNum,
            transaction.Amount
            );
         break;

      case ( TransactionType_Transfer ):
         MakeFundsTransfer(
            transaction.SourceAccountType,
            transaction.TargetAccountType,
            transaction.AccountNum,
            transaction.Amount
            );
         break;
```

つづく