Controlling Behavior

Chap.5

Study Sections 5.1 – 5.3

The `if` and `for` Statements

Method Behavior

The behavior of a method is determined by the statements within the method.

Statements fall into one of three categories called *control structures*:

Statements that simply execute in________________.

Statements that ___________ one of several alternatives.

Statements that ___________ another statement.

EVERY PROGRAM CAN BE WRITTEN USING THESE 3 CONTROL STRUCTURES.

Sequential execution

In a standard von Neumann architecture, statements are executed one at a time in sequence.

The Java ________________________ (or ___________) can be thought of as a statement that produces sequential execution of a series of statements.

```
{
   Statement1
   Statement2
   ...
   StatementN
}
```

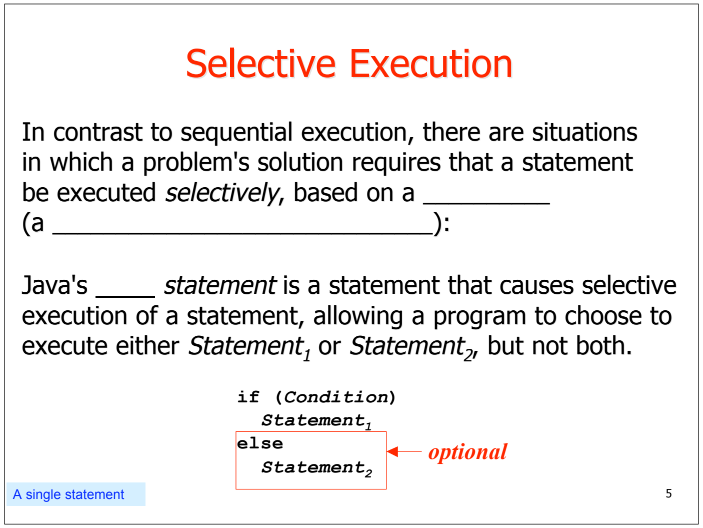
Scope

A variable declared in a block is called a ___________ ___________. It exists only from its declaration to the end of the block. We say that its ___________ extends from its declaration to the end of the block.

For example, in the code

```
if (...)
{
  int i = 1;
  ...
}
theScreen.println("Value of i = " : i);
```

the last line won't compile because local variable i is out of scope.



Selective Execution

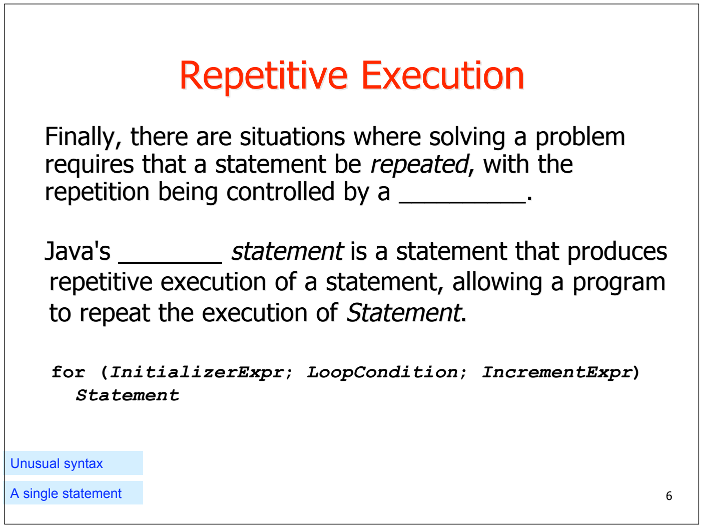
In contrast to sequential execution, there are situations in which a problem's solution requires that a statement be executed *selectively*, based on a __________ (a ______________________________):

Java's _____ *statement* is a statement that causes selective execution of a statement, allowing a program to choose to execute either *Statement*$_1$ or *Statement*$_2$, but not both.

```
if (Condition)
  Statement1
else
  Statement2
```

optional

A single statement



Repetitive Execution

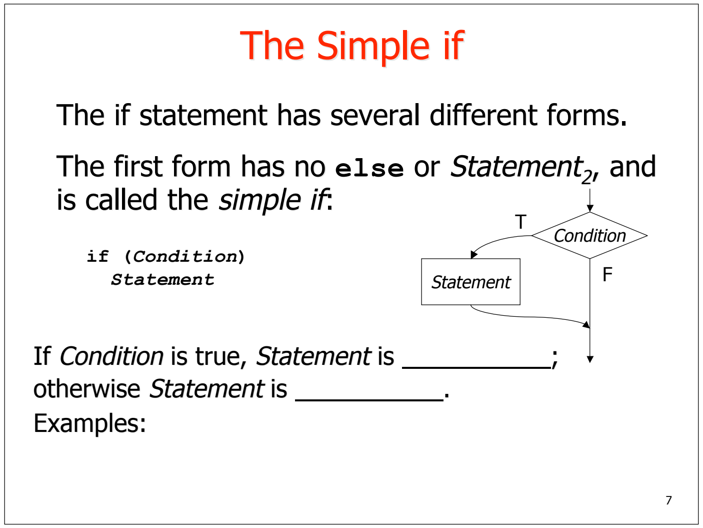
Finally, there are situations where solving a problem requires that a statement be *repeated*, with the repetition being controlled by a __________.

Java's ________ *statement* is a statement that produces repetitive execution of a statement, allowing a program to repeat the execution of *Statement*.

```
for (InitializerExpr; LoopCondition; IncrementExpr)
  Statement
```

Unusual syntax

A single statement



The Simple if

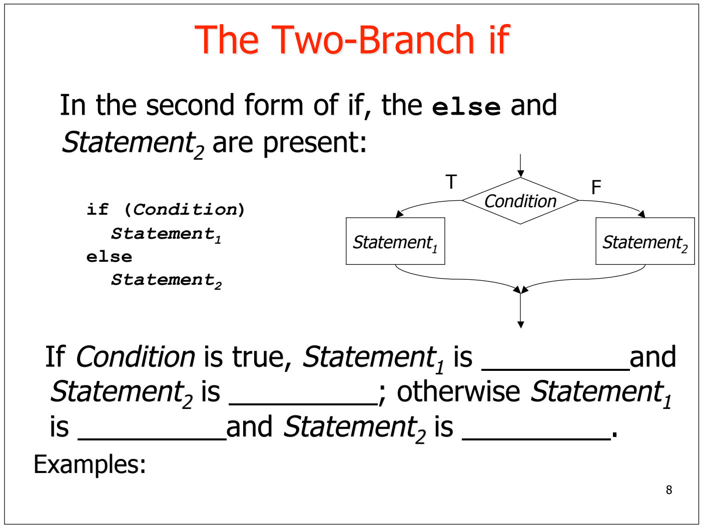
The if statement has several different forms.

The first form has no **else** or *Statement*$_2$, and is called the *simple if*:

```
if (Condition)
  Statement
```

If *Condition* is true, *Statement* is ____________; otherwise *Statement* is ____________.

Examples:



The Two-Branch if

In the second form of if, the **else** and *Statement*$_2$ are present:

```
if (Condition)
  Statement1
else
  Statement2
```

If *Condition* is true, *Statement*$_1$ is ___________and *Statement*$_2$ is ___________; otherwise *Statement*$_1$ is ___________and *Statement*$_2$ is ___________.

Examples:

Java Statements

Note that a *Statement* can be either a single statement or a sequence of statements enclosed in __________ __________:

```
if (score > 100 || score < 0)
{
   System.err.println("Invalid score!");
   System.exit(1);
}
else if (score >= 60)
  grade = 'P';
else
  grade = 'F';
```

Statements wrapped in curly braces form a single statement, called a ______________________________.
Note also that the above if statement is a *single statement*!

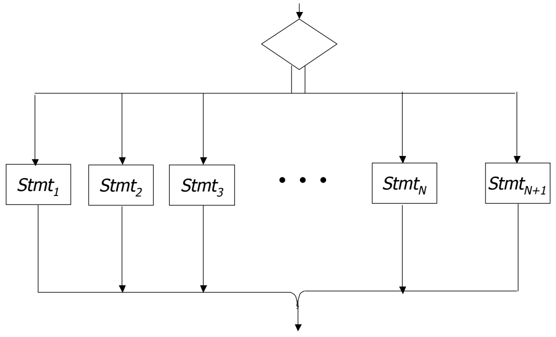
The Multi-branch if

The final form of the if statement is:

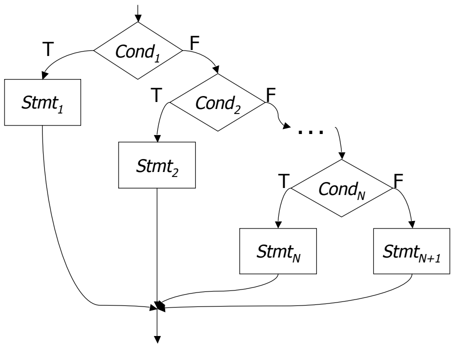
```
if (Cond_1)
   Stmt_1
else if (Cond_2)
   Stmt_2
...
else if (Cond_N)
   Stmt_N
else
   Stmt_N+1
```

__________________________of the statements $Stmt_i$ will be selected and executed, namely, the one corresponding to the first $Cond_i$ that is true.

The intent is to implement a multi-alternative selection structure of the following form, where exactly one of the alternatives is selected and executed:



Actually, however, it implements a "waterfall" selection structure of the following form:



And it is treated by the compiler as a sequence of ___________ _____s in which each else clause (except the last) is another if-else statement:

```
if (Cond_1)
  Stmt_1
else
  if (Cond_2)
    Stmt_2
  else
    if (Cond_3)
      Stmt_3
          . . .
             else
               if (Cond_N)
                 Stmt_N
               else
                 Stmt_N+1
```

This form is surely more difficult to type with all its staggered indents. It also does not display as clearly the different alternatives and that exactly one of them will be selected.

If *Condition*$_1$ is true, *Statement*$_1$ is executed and the remaining statements are skipped;

otherwise, control moves to *Condition*$_2$;
if *Condition*$_2$ is true, *Statement*$_2$ is executed and the remaining statements are skipped;

otherwise control goes to the next condition
...

if *Condition*$_N$ is true *Statement*$_N$ is executed and *Statement*$_{N+1}$ is skipped;

otherwise, *Statement*$_{N+1}$ is executed.

Example: Assigning letter grades:

Using the nested-if form:

```
if (score >= 90)
  grade = 'A';
else
  if (score >= 80)
    grade = 'B';
  else
    if (score >= 70)
      grade = 'C';
    else
      if (score >= 60)
        grade = 'D';
      else
        grade = 'F';
```

... or the preferred if-else-if form:

```
_____________________
_____________________
_____________________
_____________________
else if (score >= 70)
  grade = 'C';
else if (score >= 60)
  grade = 'D';
else
  grade = 'F';
```

Note the simple conditions; don't need `(score < 90 && score >= 80)`

Checking Preconditions

Some algorithms work correctly only if certain conditions (called *preconditions*) are true; e.g.,

- nonzero denominator
- nonnegative value for square root

We can use an if statement to check these:

```
public static double f(double x)
{
  if (x < 0)
  {
    System.err.println("invalid x");
    return 0.0;
  }
  else
    return 3.5*Math.sqrt(x);
}
```

Alternative to **Assertion.check()** in Lab Exercise 4

Repetition

There are three parts to the repetition mechanism:

- Initialization
- Repeated execution
- Termination

Now we look at one repetition statement in Java, the for statement:

Does the initialization

Causes termination — think "while this is true, do the following"

Usually modifies something each time through the loop

```
for (InitializerExpr; LoopCondition; IncrementExpr)
  Statement
```

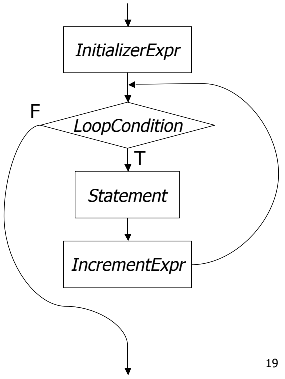
where *Statement* can be either a single statement, or a compound statement.

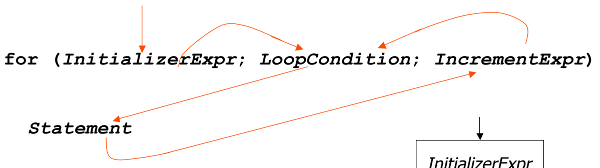
The for Loop

```
for (InitializerExpr; LoopCondition; IncrementExpr)
  Statement
```

Statement will be executed so long as *LoopCondition* is _______.

This *statement* (usually compound) is called the _______ of the loop.





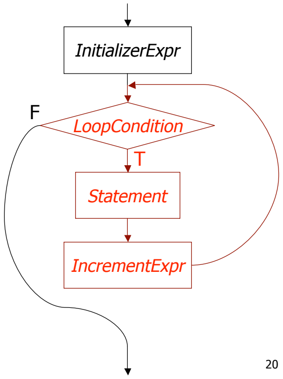
```
for (InitializerExpr; LoopCondition; IncrementExpr)

  Statement
```

Each execution of

LoopCondition
Statement
IncrementExpr

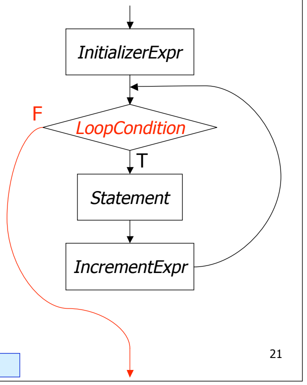
is called one *repetition* or __________ of the loop.



```
for (InitializerExpr; LoopCondition; IncrementExpr)
  Statement
```

When *LoopCondition* becomes false,control proceeds to the statement ______________________.

Note: if the *LoopCondition* is initially false, then the body of the loop will ______________________.



A *pretest* loop

Counting

The "normal" use of the for loop is to *count*:

Declare and initialize the *loop-control variable*

Check if loop-control variable has gone through all its values

Inc-/dec-rement loop-control variable

```
for (int count = 1; count <= limit; count++)
  theScreen.println(count);
```

Output (suppose limit = 5):

What if limit = 1? → __________

limit = 0? → __________

How get 1, 3, 5, . . . ? → ________________

The for-loop Scope Rule: The scope of a variable declared in a for loop extends from its declaration to the end of the for loop.

For example, the statements

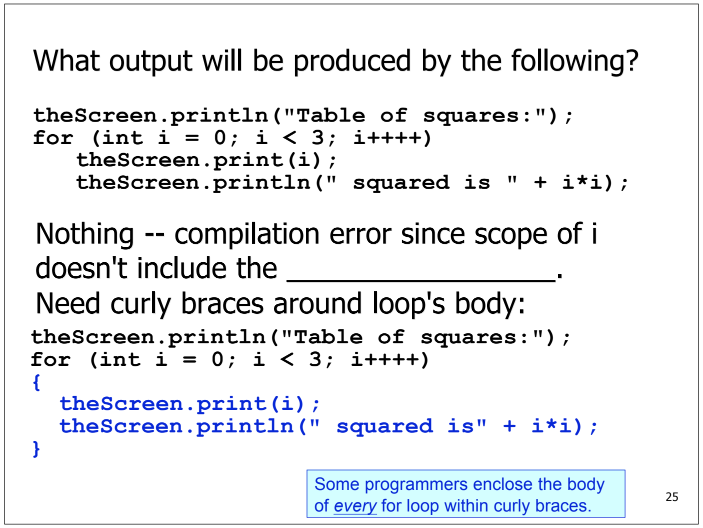
```
int sum = 0;
for (int count = 1; count <= limit; count++)
  sum += count;
theScreen.println("Sum = " + sum +
                  "when count is " + count);
```

result in an error because the variable **count** in the last statement is ______________________.

One solution would be to declare **count** ____________ the for loop:

```
int sum = 0,
____________
for (____________; count <= limit; count++)
  sum += count;
theScreen.println("Sum = " + sum +
                  "when count is " + count);
```

Output for limit = 3?



What output will be produced by the following?

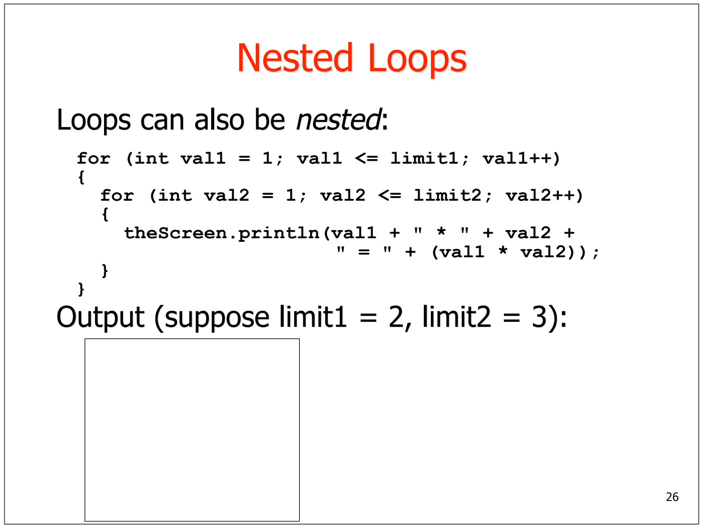
```
theScreen.println("Table of squares:");
for (int i = 0; i < 3; i++++)
   theScreen.print(i);
   theScreen.println(" squared is " + i*i);
```

Nothing -- compilation error since scope of i doesn't include the __________________.

Need curly braces around loop's body:

```
theScreen.println("Table of squares:");
for (int i = 0; i < 3; i++++)
{
  theScreen.print(i);
  theScreen.println(" squared is" + i*i);
}
```

Some programmers enclose the body of *every* for loop within curly braces.

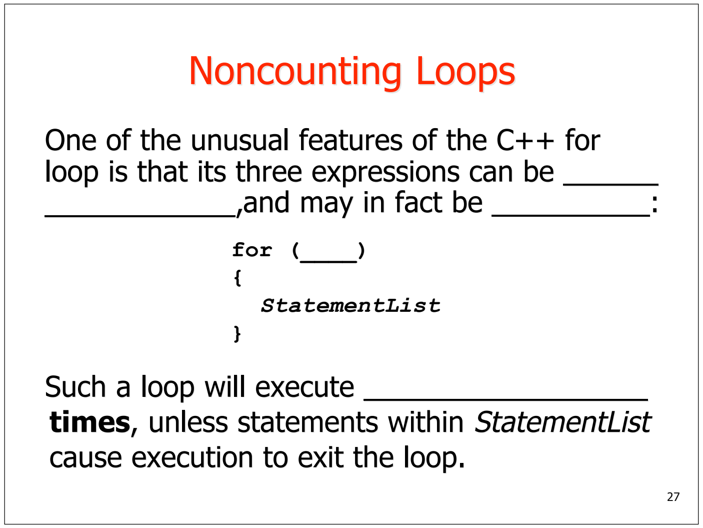


Nested Loops

Loops can also be *nested*:

```
for (int val1 = 1; val1 <= limit1; val1++)
{
  for (int val2 = 1; val2 <= limit2; val2++)
  {
    theScreen.println(val1 + " * " + val2 +
                      " = " + (val1 * val2));
  }
}
```

Output (suppose limit1 = 2, limit2 = 3):

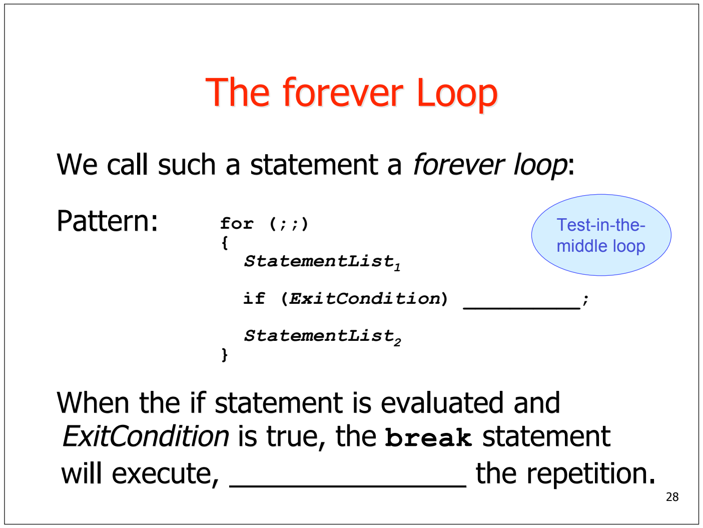


Noncounting Loops

One of the unusual features of the C++ for loop is that its three expressions can be ______ _____________,and may in fact be ___________:

```
for (____)
{
  StatementList
}
```

Such a loop will execute ___________________ **times**, unless statements within *StatementList* cause execution to exit the loop.



The forever Loop

We call such a statement a *forever loop*:

Pattern:

```
for (;;)
{
  StatementList_1

  if (ExitCondition) __________;

  StatementList_2
}
```

Test-in-the-middle loop

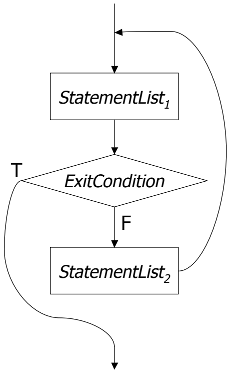
When the if statement is evaluated and *ExitCondition* is true, the **break** statement will execute, ________________ the repetition.

Forever Behavior

```
for (;;)
{
  StatementList1

  if (ExitCondition) break;

  StatementList2
}
```



Note: we are guaranteed that *StmtList1* will execute at least once, but *StmtList2* may not execute.

Input Loops

The forever loop is ideal for reading a list of values whose end is marked by a ___________ (i.e., a value that signals the end of input).

Pattern:

```
for (;;)
{
  Prompt for value
  Read value

  if (value is the sentinel) break;

  Process value
}
```

Example

Read and average a list of test scores:

```
public static double ReadAndAverage()
{
  double score, sum   = 0.0;
  int count = 0;

  ______________
  {
    theScreen.print("Enter a test score (-1 to quit): ");
    score = theKeyboard.readDouble();

    _________________________________;   // test for sentinel

    count++;
    sum += score;
  }
  if (count > 0)
    return sum / count;
  else
  {
    System.err.println("\n* no scores to average!\n");
    System.exit(1);
  }
}
```

Error Handling

A forever loop is also useful for fool-proofing input.

Pattern:

```
for (;;)
{
  Prompt for value
  Read value

  if (value is valid) break;

  Display error message
}
```

This is good because control will only leave the loop if/when the user enters a valid value.