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 **sequence**.

Statements that **select** one of several alternatives.

Statements that **repeat** 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 ***compound statement*** (or ***block***) can be thought of as a statement that produces sequential execution of a series of statements.

{
  $Statement_1$
  $Statement_2$
  ...
  $Statement_N$
}

Scope

A variable declared in a block is called a local variable. It exists only from its declaration to the end of the block. We say that its scope 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 *condition* (a boolean expression):

Java's *if 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 *condition*.

Java's *for 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

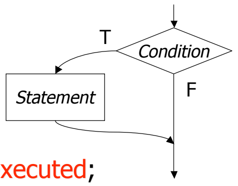
A single statement

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
```



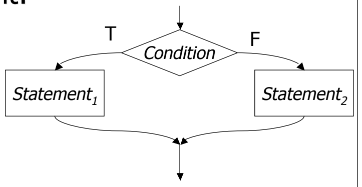
If *Condition* is true, *Statement* is executed; otherwise *Statement* is skipped.

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 executed and *Statement*$_2$ is skipped; otherwise *Statement*$_1$ is skipped and *Statement*$_2$ is executed.

Examples:

Java Statements

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

```
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 compound statement.
Note also that the above if statement is a *single statement*!

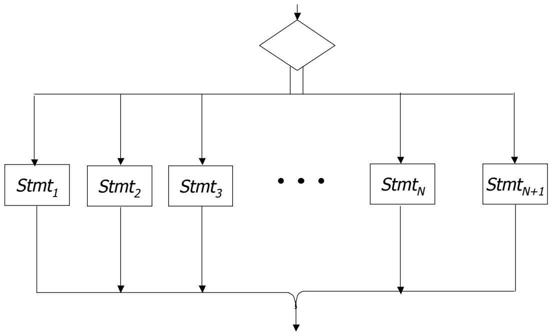
The Multi-branch if

The final form of the if statement is:

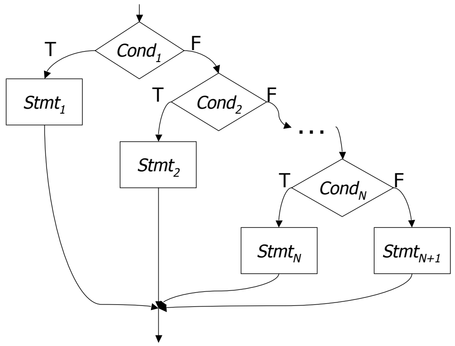
```
if ($Cond_1$)
  $Stmt_1$
else if ($Cond_2$)
  $Stmt_2$
...
else if ($Cond_N$)
  $Stmt_N$
else
  $Stmt_{N+1}$
```

Exactly one 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 nested ifs 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.

```
if ($Cond_1$)
  $Stmt_1$
else if ($Cond_2$)
  $Stmt_2$
...
else if ($Cond_N$)
  $Stmt_N$
else
  $Stmt_{N+1}$
```

Example: Assigning letter grades:

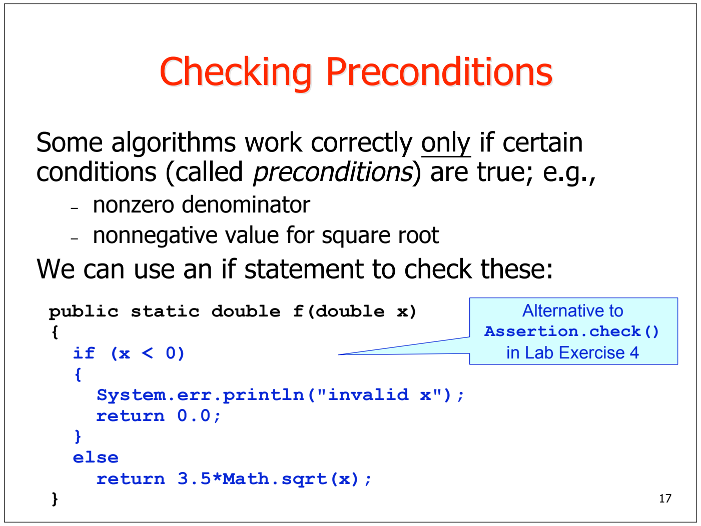
Using the nested-if form:

```
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:

```
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';
```

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



Checking Preconditions

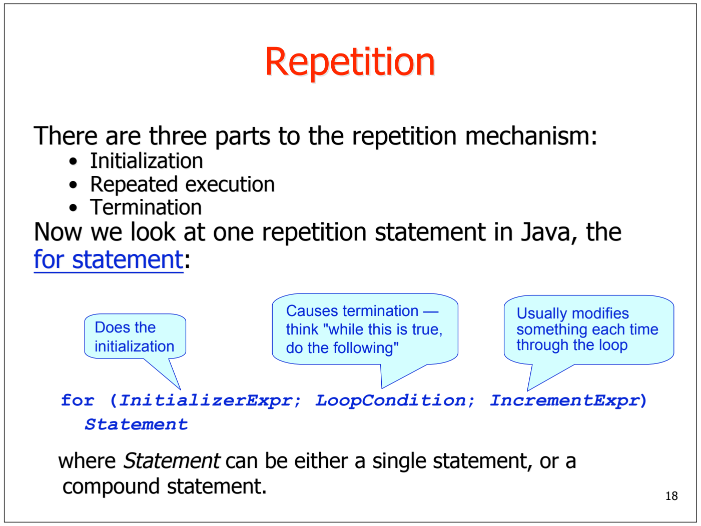
Some algorithms work correctly only if certain conditions (called *preconditions*) are true; e.g.,

- 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:

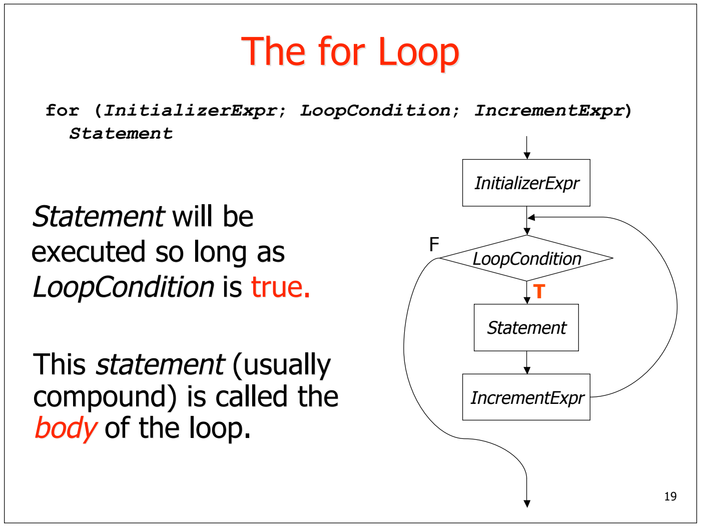
Does the initialization

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.

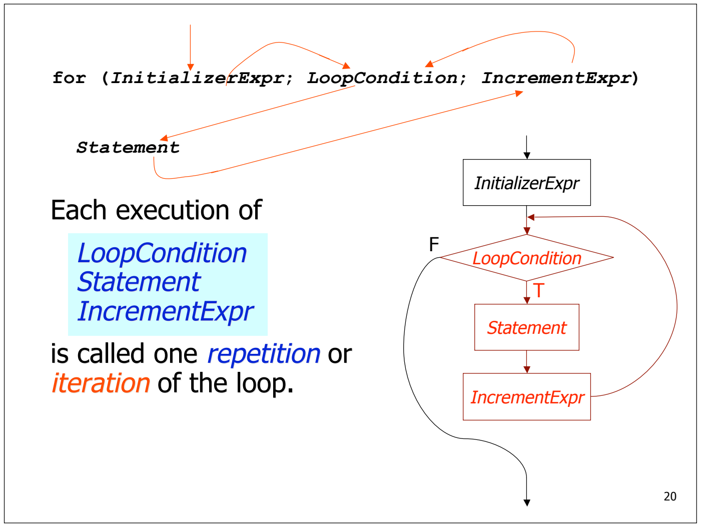


The for Loop

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

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

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



```
for (InitializerExpr; LoopCondition; IncrementExpr)

  Statement
```

Each execution of

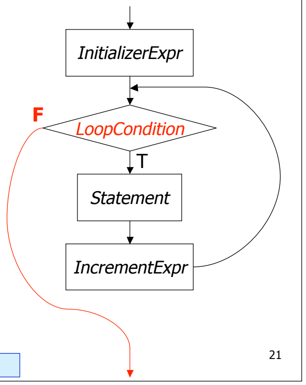
LoopCondition
Statement
IncrementExpr

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

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

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

Note: if the *LoopCondition* is initially false, then the body of the loop will not be executed.



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):

```
1
2
3
4
5
```

What if limit = 1? → 1

limit = 0? → no output

How get 1, 3, 5, . . . ? → `count += 2`

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

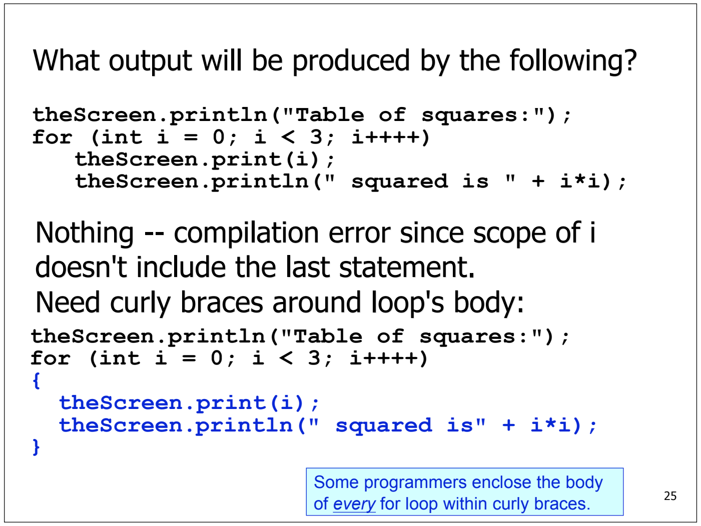
```
int sum = 0;
for (int count = 1; count <= limit; count++)
  sum += count;
theScreen.println("Sum = " + sum +
                  "when count is " + count);
```

result in an error because the variable **count** in the last statement is outside its scope.

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

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

Output for limit = 3? 6



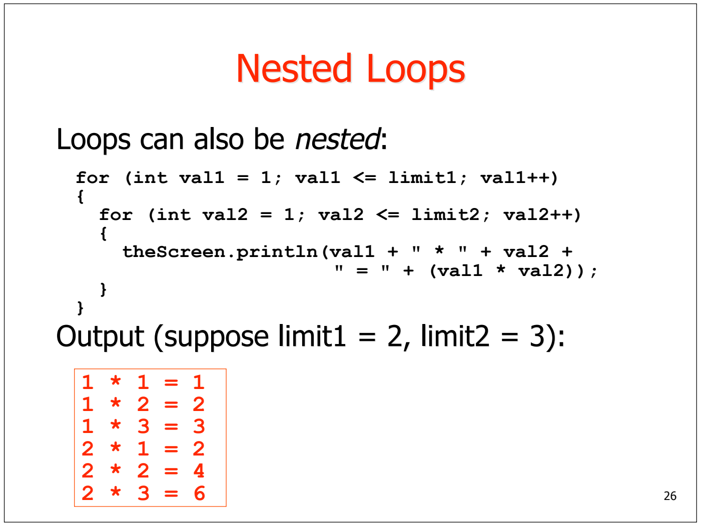
What output will be produced by the following?

```
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 last statement.
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.



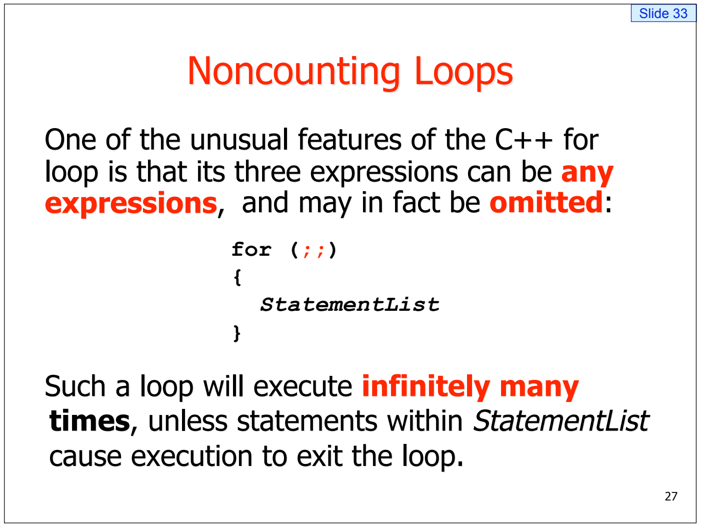
Nested Loops

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):

```
1 * 1 = 1
1 * 2 = 2
1 * 3 = 3
2 * 1 = 2
2 * 2 = 4
2 * 3 = 6
```



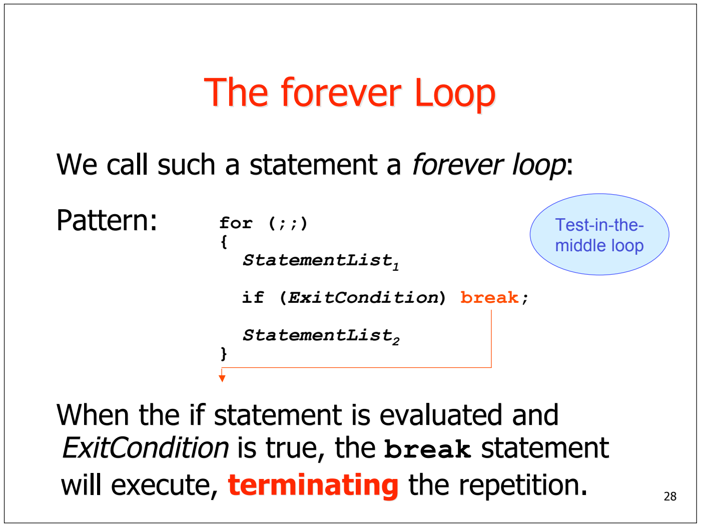
Slide 33

Noncounting Loops

One of the unusual features of the C++ for loop is that its three expressions can be **any expressions**, and may in fact be **omitted**:

```
for (;;)
{
  StatementList
}
```

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



The forever Loop

We call such a statement a *forever loop*:

Pattern:

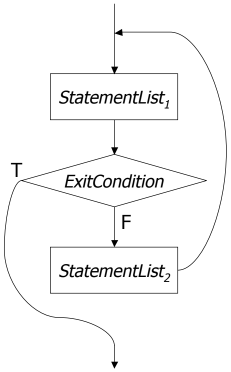
```
for (;;)
{
  StatementList1

  if (ExitCondition) break;

  StatementList2
}
```

Test-in-the-middle loop

When the if statement is evaluated and *ExitCondition* is true, the **break** statement will execute, **terminating** 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 ***sentinel*** (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;
  for (;;)
  {
    theScreen.print("Enter a test score (-1 to quit): ");
    score = theKeyboard.readDouble();
    if (score < 0) break;    // 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);
  }
}
```

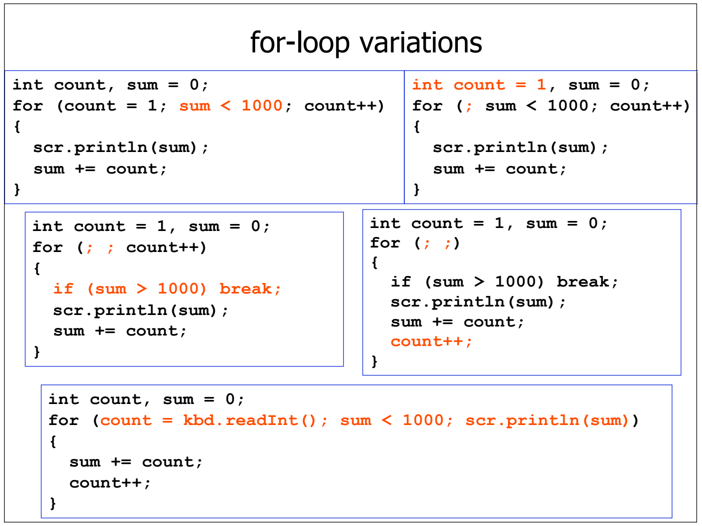
Error Handling

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.



for-loop variations

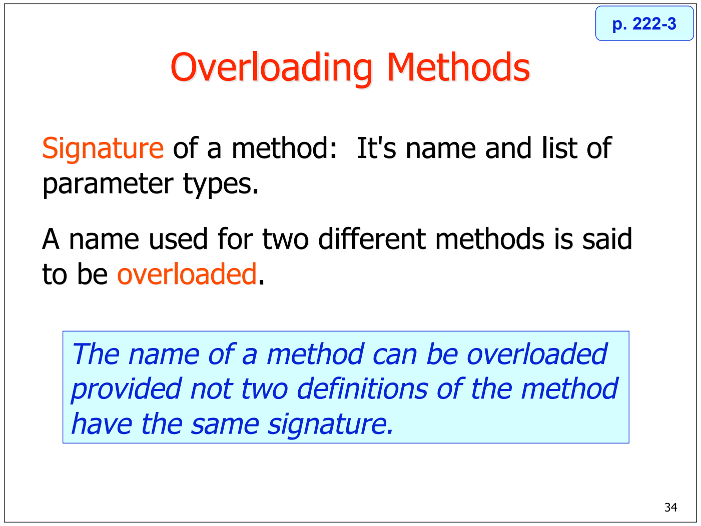
```
int count, sum = 0;
for (count = 1; sum < 1000; count++)
{
  scr.println(sum);
  sum += count;
}
```

```
int count = 1, sum = 0;
for (; sum < 1000; count++)
{
  scr.println(sum);
  sum += count;
}
```

```
int count = 1, sum = 0;
for (; ; count++)
{
  if (sum > 1000) break;
  scr.println(sum);
  sum += count;
}
```

```
int count = 1, sum = 0;
for (; ;)
{
  if (sum > 1000) break;
  scr.println(sum);
  sum += count;
  count++;
}
```

```
int count, sum = 0;
for (count = kbd.readInt(); sum < 1000; scr.println(sum))
{
  sum += count;
  count++;
}
```



p. 222-3

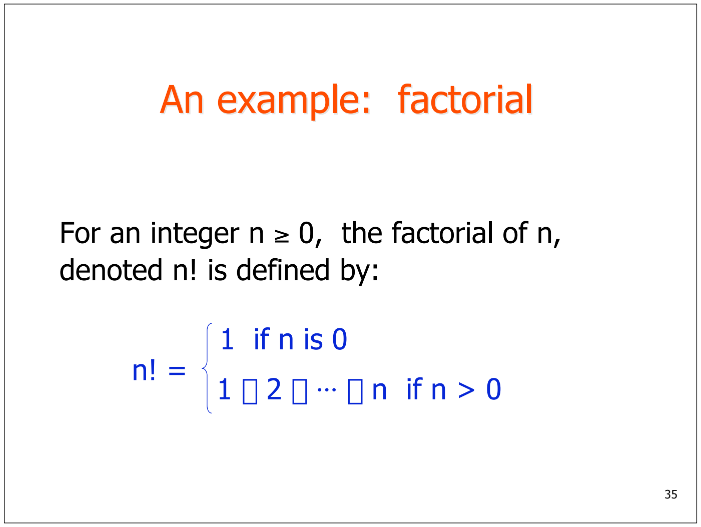
Overloading Methods

Signature of a method: It's name and list of parameter types.

A name used for two different methods is said to be overloaded.

The name of a method can be overloaded provided not two definitions of the method have the same signature.

34



An example: factorial

For an integer n ≥ 0, the factorial of n, denoted n! is defined by:

$$n! = \begin{cases} 1 & \text{if n is 0} \\ 1 \square 2 \square \cdots \square n & \text{if } n > 0 \end{cases}$$

35

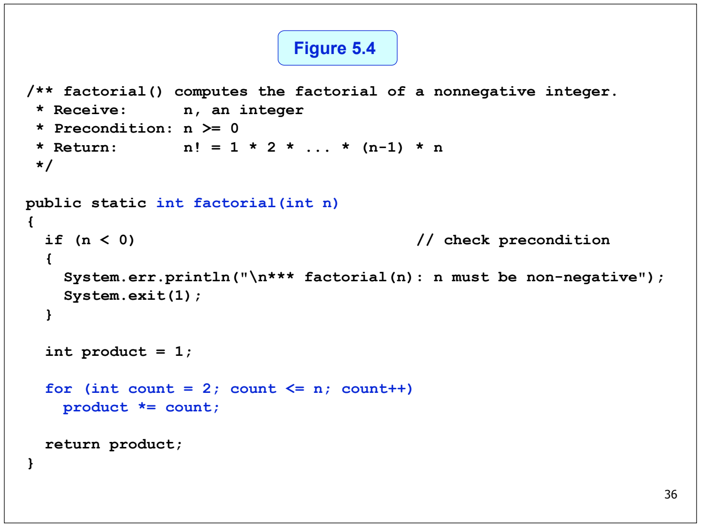


Figure 5.4

```
/** factorial() computes the factorial of a nonnegative integer.
 * Receive:      n, an integer
 * Precondition: n >= 0
 * Return:       n! = 1 * 2 * ... * (n-1) * n
 */

public static int factorial(int n)
{
  if (n < 0)                              // check precondition
  {
    System.err.println("\n*** factorial(n): n must be non-negative");
    System.exit(1);
  }

  int product = 1;

  for (int count = 2; count <= n; count++)
    product *= count;

  return product;
}
```

36

Figure 5.6

```
// FactorialLoopDriver.java computes any number of factorials.

import ann.easyio.*;

class FactorialLoopDriver
{
 // Insert the definition of factorial() from Figure 5.4 here

 public static void main(String [] args)
 {
   Screen theScreen = new Screen();
   Keyboard theKeyboard = new Keyboard();
   int theNumber;

   for (;;)
   {
     theScreen.print("To compute n!, enter n (-1 to quit): ");
     theNumber = theKeyboard.readInt();

     if (theNumber < 0) break;

     theScreen.println(theNumber + "! = "
                      + factorial(theNumber) + "\n");
   }
 }
}
```

Sample runs:

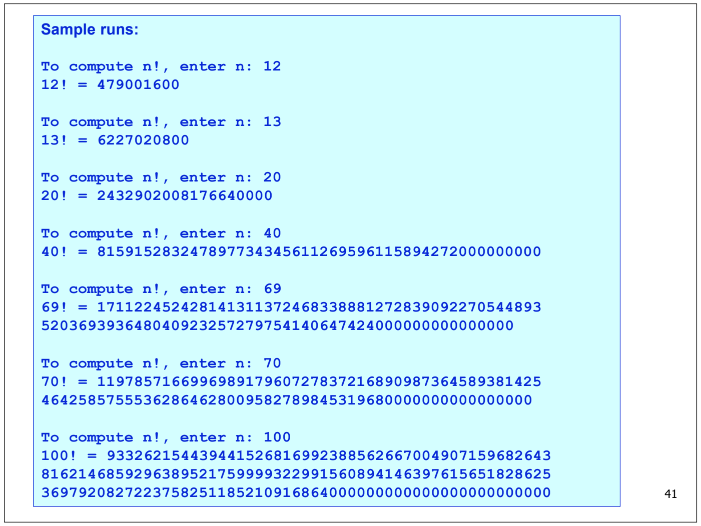
```
To compute n!, enter n (-1 to quit): 1
1! = 1

To compute n!, enter n (-1 to quit): 2
2! = 2

To compute n!, enter n (-1 to quit): 5
5! = 120

To compute n!, enter n (-1 to quit): 6
6! = 720

To compute n!, enter n (-1 to quit): -1
```

Gives wrong answers for n ≥ 13

Figure 5.7

```
/** factorial() computes n!, given a nonnegative BigInteger.
 *  Receive:      n, a BigInteger
 *  Precondition: n >= 0
 *  Return:       n! = 1 * 2 * ... * (n-1) * n
 */

public static BigInteger factorial(BigInteger n)
{
  if(n.compareTo(ZERO) < 0)                     // check precondition
  {
    System.err.println("\n*** factorial(n): invalid argument "
                       + n + "received");
    System.exit(1);
  }
  final BigInteger ONE = new BigInteger("1");   // constant 1
  BigInteger product = new BigInteger("1");

  for (BigInteger count = new BigInteger("2");  // initExpression
                  count.compareTo(n) <= 0;      // booleanExpression
                  count = count.add(ONE))       // stepExpression
    product = product.multiply(count);          // statement

  return product;
}
```

```
// BigIntegerFactorialDriver.java tests the factorial() method

import ann.easyio.*;                           // Screen, Keyboard
import java.math.BigInteger;

class BigIntegerFactorialDriver
{
  final static BigInteger ZERO = new BigInteger("0");

 // Insert definition of factorial() here

  public static void main(String [] args)
  {
    Screen theScreen = new Screen();
    Keyboard theKeyboard = new Keyboard();
    String numberString;
    BigInteger theNumber;
    for (;;)
    {
      theScreen.print("To compute n!, enter n (-1 to quit): ");
      numberString = theKeyboard.readWord();
          theNumber = new BigInteger(numberString);

      if (theNumber.compareTo(ZERO) < 0) break;

      theScreen.println(theNumber + "! = "
           + factorial(theNumber) + "\n");
    }
  }
}
```



Sample runs:

```
To compute n!, enter n: 12
12! = 479001600

To compute n!, enter n: 13
13! = 6227020800

To compute n!, enter n: 20
20! = 2432902008176640000

To compute n!, enter n: 40
40! = 815915283247897734345611269596115894272000000000

To compute n!, enter n: 69
69! = 171122452428141311372468338881272839092270544893
520369393648040923257279754140647424000000000000000

To compute n!, enter n: 70
70! = 119785716699698917960727837216890987364589381425
46425857555362864628009582789845319680000000000000000

To compute n!, enter n: 100
100! = 93326215443944152681699238856266700490715968264
381621468592963895217599993229915608941463976156518286253
697920827223758251185210916864000000000000000000000000
```