Using Control Structures in Methods

Chapter 5
Chapter Contents

Objectives
5.1 Example: Improved Payroll Program
5.2 Methods That Use Selection
5.3 Methods That Use Repetition
5.4 Graphical/Internet Java: Old MacDonald ... Applet Revisited

PART OF THE PICTURE:
Computability Theory

PART OF THE PICTURE:
Numerical Computing
Objectives

- Give problem example requiring new control structures
- Take first look at basic control structures
  - sequential
  - selection
  - repetition
- Study the if statement used for selection
Objectives

- See use of \texttt{for} statement for counter-controlled repetitions
- See use of \texttt{for} statement used as “forever” loops
- Give applet example to generate output
- Brief indication of area of computability theory
- Describe use of numerical methods
5.1 Example: Improved Payroll Program

- Previous program (Figure 2.1) now must be upgraded
- Need capability of including overtime pay
- Desire for program to handle multiple employees, not just one
# Additional Objects

<table>
<thead>
<tr>
<th>Objects</th>
<th>Type</th>
<th>Kind</th>
<th>Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>In addition to previous objects …</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>regular wages</td>
<td>double</td>
<td>variable</td>
<td>regularPay</td>
</tr>
<tr>
<td>overtime pay factor</td>
<td>double</td>
<td>constant</td>
<td>OVERTIME_FACTOR</td>
</tr>
<tr>
<td>overtime wages</td>
<td>double</td>
<td>variable</td>
<td>overtimePay</td>
</tr>
<tr>
<td>combined wages</td>
<td>double</td>
<td>variable</td>
<td>wages</td>
</tr>
</tbody>
</table>
Additional Operations

Previous Operations ...
... plus ...

Compute regularPay, overtimePay, wages

Display real values (wages)

Repeat steps for each employee
Calculating Wages

More complicated than before:

if hoursWorked ≤ 40, calculate:
    regularPay = hoursWorked x
                hourlyRate;
    overtimePay = 0;
Otherwise, calculate:
    regularPay = 40 x hourlyRate
    overtimePay = OVERTIME_FACTOR x
                  (hoursWorked - 40) x
                  hourlyRate
wages = regularPay + overtimePay
Algorithm for New Payroll Program

- Construct Screen and Keyboard objects
- Display prompt for number of employees
- Read integer into numEmployees
- Loop from 1 through numEmployees

- For each employee ...
  - Display prompts for hours, rate
  - Read doubles into hoursWorked, hourlyRate
  - Calculate wages according to previous algorithm
  - Display results with message
Coding and Testing

Note source code Figure 5.1

looping structure
for( int count = 1 ;
    count <= numEmployees ;
    count++ )
    { ... }

Selection structure
if( hours worked <= 40 )
    { ... }
else
    { ... }

Note sample runs
5.2 Methods That Use Selection

Problem:
Given two real values, return the minimum of the two values

Behavior for our method
- receive two real values from caller
- if first less than second, return first
- otherwise return second
## Objects

<table>
<thead>
<tr>
<th>Object</th>
<th>Type</th>
<th>Kind</th>
<th>Movement</th>
<th>Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>1st value</td>
<td>double</td>
<td>variable</td>
<td>received</td>
<td>first</td>
</tr>
<tr>
<td>2nd value</td>
<td>double</td>
<td>variable</td>
<td>received</td>
<td>second</td>
</tr>
<tr>
<td>minimum value</td>
<td>double</td>
<td>variable</td>
<td>returned</td>
<td></td>
</tr>
</tbody>
</table>

- **Objects**: Values and their properties such as type, kind, and movement are listed. The values are handled as variables in the context of received or returned actions.
Operations

- Receive two real values from method's caller
- Compare the two values to see if one is less than the other
- Do one (but not both of the following)
  - Return the first value
  - Return the second value
public static double minimum (double first, double second) {
    if (first < second)
        return first;
    else
        return second;
}

Note driver program source code with sample runs, Figure 5.3
Programming Structures

- **Sequential Execution**
  - Like traveling down a straight road
  - Pass through a sequence of points or locations

- **Selective Execution**
  - Like coming to a fork in the road
  - We either take one direction or the other
Programming Structures
Selective Execution

Sequential Execution

Statement
Statement
Statement

Selective Execution

true
false

Stmt 1
Stmt 2
Alternate Graphical Representation

Sequential Execution

Statement_1
Statement_2...
Statement_n

Selective Execution

true ? false

Statement_1 Statement_2
IF Statement

Two basic forms

if( boolean_expression )
  statement  Statement is only executed if boolean_expression is true

if( boolean_expression )
  statement1 else
  statement2  Statement1 is executed if boolean_expression is true; otherwise statement2 is executed
Blocks

- An if statement may need to control several statements.
- A group or “block” of statements can be specified with braces:

```c
{ statement1 statement2 . . . }
```

- Note use in wage calculation.
Checking Preconditions

Some algorithms work correctly only if certain conditions are true
- no zero in a denominator
- non negative value for square root
- if statement enables checking

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

Key issue is how well humans (not computers) can read the source code

Form for if statements

- Align the if and the else
- Use indentation to mark statements being selected (controlled) by the if and else
Nested if(s)

- Note the syntax of the if statement.
- It controls whether a statement will be executed.
- This statement could be another if.
- Referred to as a “nested” if:
  ```
  if( boolean_expression1 )
      statement1
  else if( boolean_expression2 )
      statement2
  ```
Method Signature

Signature (unique identification) of a method made up of
- the name of the method
- the list of the types of parameters

This means we could have two methods with the same name but different types and/or numbers of parameters

```java
public static double minimum (double first, double second) ...
public static int minimum (int first, int second)
```
Method Overloading

Two different methods with the same name are said to be “overloaded”

The name of a method can be overloaded, provided no two definitions of the method have the same signature
5.3 Methods That Use Repetition

Problem: Computing factorials

\[ n! = \begin{cases} 
1 & n = 0 \\
1 \times 2 \times \ldots \times n & n > 0 
\end{cases} \]

Write a method that given an integer \( n \geq 0 \), computes \( n \) factorial (\( n! \))
## Object-Centered Design

**Behavior**– repeated multiplication

**Objects**

<table>
<thead>
<tr>
<th>Object</th>
<th>Type</th>
<th>Kind</th>
<th>Movement</th>
<th>Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>integer &gt;=0</td>
<td>variable</td>
<td>int</td>
<td>received</td>
<td>n</td>
</tr>
<tr>
<td>running product</td>
<td>variable</td>
<td>int</td>
<td>returned</td>
<td>product</td>
</tr>
<tr>
<td>counter</td>
<td>variable</td>
<td>int</td>
<td>(local)</td>
<td>count</td>
</tr>
</tbody>
</table>
Operations

1. Check precondition \( n \geq 0 \)
2. Define, initialize two integer variables
   \( \text{product} \) and \( \text{count} \)
3. Multiply \( \text{product} \times \text{count} \), assign result to \( \text{product} \)
4. Increment \( \text{count} \)
5. Repeat 3. and 4. so long as \( \text{count} \leq n \)
Algorithm

- Receive \( n \) from caller, check precondition
- Initialize \texttt{product} to 1
- Repeat following for each value of \texttt{count} in range 2 through \( n \)
  - Multiply \texttt{product} by \texttt{count}
- Return \texttt{product}
Coding

- Note `factorial()` method, Figure 5.4 in text
- Note Driver for Method `factorial()`, Figure 5.5 in text
- Note test runs
  - with legal arguments
  - with invalid argument
Repeated Execution: The `for` Statement

- Make analogy to a roadway
  - Think of a race track
  - Enter the track
  - Circle for a set number of times
  - Leave the track
- Three parts to the repetition mechanism
  - Initialization
  - Repeated execution
  - Termination
/* given */
for (int count=2; count <= n; count++)
    product *= count

int count = 2
while count <= n
    product *= count;
    count++;
for Statement Syntax

`for (initExpression; booleanExpression; stepExpression) statement;

for is a keyword
initExpression: usually an assignment
booleanExpression: usually a comparison (think “while”)
stepExpression: usually an increment`
Typical for Execution

```java
for (initExpression;
     booleanExpression
     stepExpression)
    statement;
```

1. Loop control variable given initial value
2. booleanExpression checked
   1. If it is true, statement executed
   2. If false, loop terminates
3. Increment of loop control variable
4. Back to step 2
Alternate Version of for

Specifications inside the parentheses are not required only the two semicolons

for ( ; ; )
{
    if ( ... ) break;
}

break statement jumps flow of control out of for loop (See Figure 5.6 in text)
Sentinel Based Loop

- Often user asked to enter a sentinel value
- When sentinel value found in `if ( )`, loop terminates

```c
for ( ; ; )
    {
        . . .
        if ( value is sentinel ) break;
    }
```

- Called “sentinel-based” input processing
Forever Loops

Using `for ( )`

```java
for ( ; ; )
{
    . . .
    if (booleanExpression) break;
    . . . }
```

Using `while ( )`

```java
while ( true )
{
    . . .
    if (booleanExpression) break;
    . . . }
```

Note: something in the loop must cause `booleanExpression` to evaluate to true

Otherwise the loop does go forever
Testing, Maintaining
factorial()

- Method works correctly for values 1 – 12
- Incorrect value for 13!
  - Algorithm is correct
  - Problem is use if type int
  - 13! exceeds maximum int value
- Solution is to change type returned (and received) by the method to
- Note new version and test runs,
  Figure 5.7 of text
5.4 Graphical/Internet Java: Old MacDonald... Applet Revisited

- Write versions of the applet using more flexible structure
- Write for ( ) loop to receive inputs from user
  - name of animal
  - sound of animal
- See source code Figure 5.8, Text
Note the capabilities now available to us:

- Sequential execution
- Selection (branching)
- Repetition (looping)

These operations provide more capability.
Computability Theory Considerations

- What kinds of operations can/cannot be computed?
- How can be operations be classified?
- What relationships exist among classes?
- What is most efficient algorithm for solving a particular problem?
Computability Theory

- Represent programs abstractly
- Use mathematical model
- Provides language and hardware independence
- Gives theory with timelessness
Part of the Picture: Numerical Methods

- Mathematical models used to solve a variety of problems
- Often involve solutions to different kinds of equations

Examples:
- Curve fitting
- Equation solving
- Integration
- Differential equations
- Solving linear systems
The sum of the areas of these trapezoids is approximately the area under the graph of \( f(x) \) between the points \( x_0 \) and \( x_n \). The approximation improves as \( \Delta x \) gets smaller.
Trapezoidal Method

Use this formula as an algorithm for calculating approximation of area.

\[
\text{area} = \Delta x \left( \frac{f(x_0) + f(x_n)}{2} + \sum_{i=1}^{n-1} f(x_i) \right)
\]
Trapezoidal Area Method

- Note source code Figure 5.9 in text

Tasks
- Screen prompts for y values
- Inside for() loop sums the successive f(x) values
- Calculates and returns total area under curve

Method applied to road construction
- Determine total volume of dirt removed for highway (Figure 5.10, text)
- Cross section is trapezoid