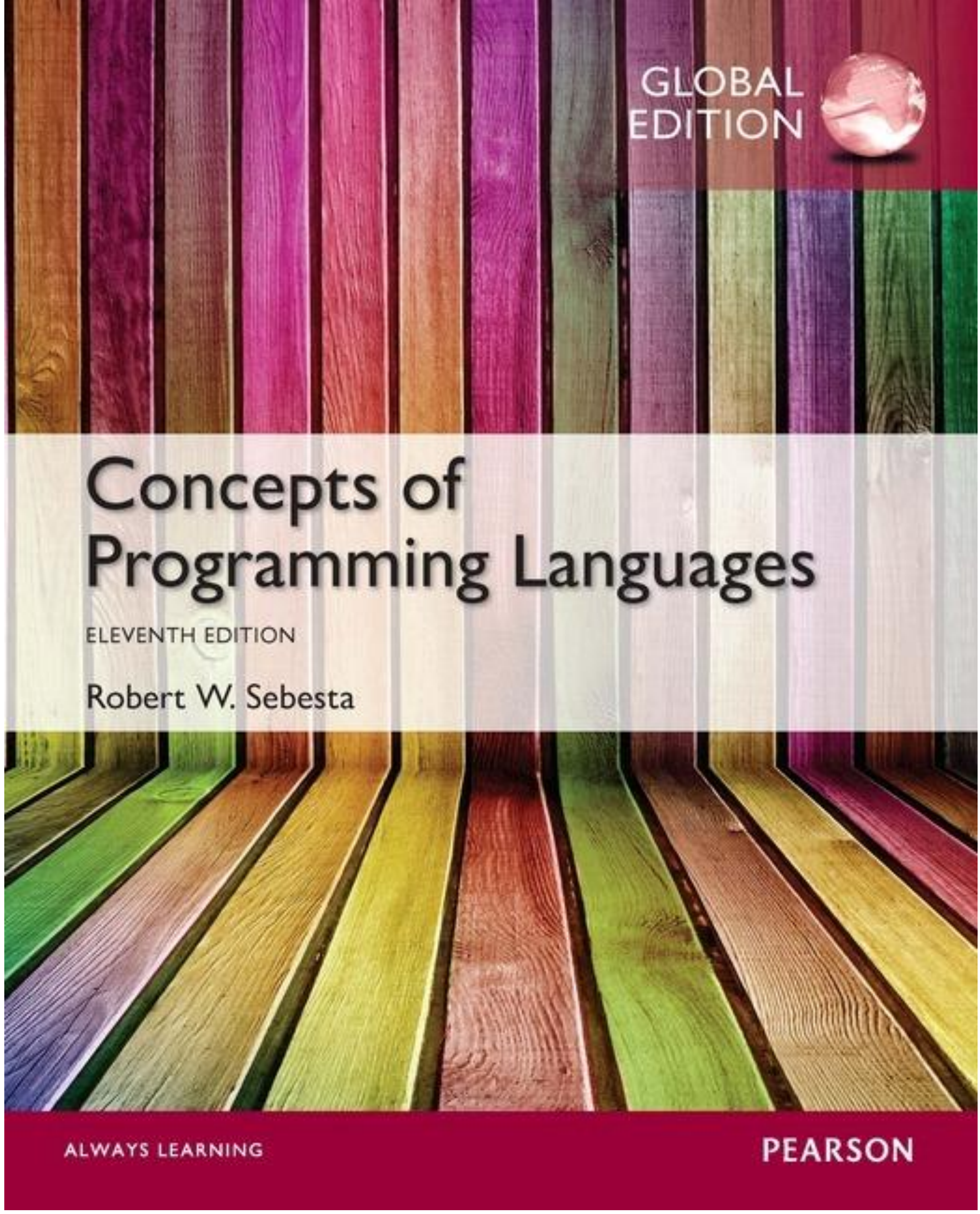
Chapter 8

Statement-Level Control Structures



Chapter 8 Topics

- Introduction
- Selection Statements
- Iterative Statements
- Unconditional Branching
- Guarded Commands
- Conclusions

Copyright © 2017 Pearson Education, Ltd. All rights reserved.

Levels of Control Flow

- Within expressions (Chapter 7)
- Among program units (Chapter 9)
- Among program statements (this chapter)

Copyright © 2017 Pearson Education, Ltd. All rights reserved.

Controlling Program Flows

- A control structure is a control statement and the statements whose execution it controls
- Most programming languages follow a single thread of control (or scheduling)
- Types of control statements:
  - Selection statements
  - Iterative statements
  - Unconditional branching statement

- Levels of Control Flow:
  1. Within expressions
  2. Among program units
  3. Among program statements

Copyright © 2017 Pearson Education, Ltd. All rights reserved.

Control Statements: Evolution

- FORTRAN I control statements were based directly on IBM 704 hardware
- Much research and argument in the 1960s about the issue
  - One important result: It was proven that all algorithms represented by flowcharts can be coded with only two-way selection and pretest logical loops

Copyright © 2017 Pearson Education, Ltd. All rights reserved.

Control Structure

- A *control structure* is a control statement and the statements whose execution it controls
- Design question
  - Should a control structure have multiple entries?

Copyright © 2017 Pearson Education, Ltd. All rights reserved.

Selection Statements

- A *selection statement* provides the means of choosing between two or more paths of execution
- Two general categories:
  - Two-way selectors
  - Multiple-way selectors

Two-Way Selection Statements

- General form:

```
if control_expression
    then clause
    else clause
```

- Design Issues:
  - What is the form and type of the control expression?
  - How are the **`then`** and **`else`** clauses specified?
  - How should the meaning of nested selectors be specified?

Copyright © 2017 Pearson Education, Ltd. All rights reserved.

The Control Expression

- If the then reserved word or some other syntactic marker is not used to introduce the then clause, the control expression is placed in parentheses
- In C89, C99, Python, and C++, the control expression can be arithmetic
- In most other languages, the control expression must be Boolean

```
#include <stdio.h>

int main()
{
    int x = -10;
    if (x+10)
       printf("Hello World");
    else
       printf("Hello Zero");
    return 0;
}
```

Clause Form

- In many contemporary languages, the then and else clauses can be single statements or compound statements
- In Perl, all clauses must be delimited by braces (they must be compound)
- In Python and Ruby, clauses are statement sequences
- Python uses indentation to define clauses

```
if x > y :
  x = y
  print " x was greater than y"
```

```
if (x > y) {
   x = y
   print " x was greater than y"
}
```

Two Way-Selection in Common Lisp

- Syntax

```
(if <test>     <do_if_true>
               <do_if_false>)
```

- Single/nested functions for **`<do_if_...>`**

- May contain multiple statements
  - if surrounded by a block structure
    - **`prog`**, **`let`**, **`do`**, etc.

Lisp – Two Way Selection

```
(when (= x 0)
   (terpri)
   (princ "It's now 0")
)


(if (> (aref a x) (aref b x))
   (progn                ; block,
   returns last value
       (terpri) (princ "Fixing")
       (setf (aref a x) (aref b x)))
   (if (< (aref a x) (aref b x))
       (format t "~%It's Less")
       (format t "~%It's Equal")))
```

- an array with 10- cells, named my-array
  (setf my-array (make-array '(10)))
- access the content of the tenth cell
  (aref my-array 9)

progn is a special form that causes each of its arguments to be evaluated in sequence and then returns the value of the last one.

Nesting Selectors

- Java example

```
    if (sum == 0)
      if (count == 0)
           result = 0;
  else result = 1;
```

- Which **`if`** gets the **`else`**?
- Java's static semantics rule: **`else`** matches with the nearest previous **`if`**

Nesting Selectors (continued)

- To force an alternative semantics, compound statements may be used:

```
if (sum == 0) {
 if (count == 0)
     result = 0;
}
else result = 1;
```

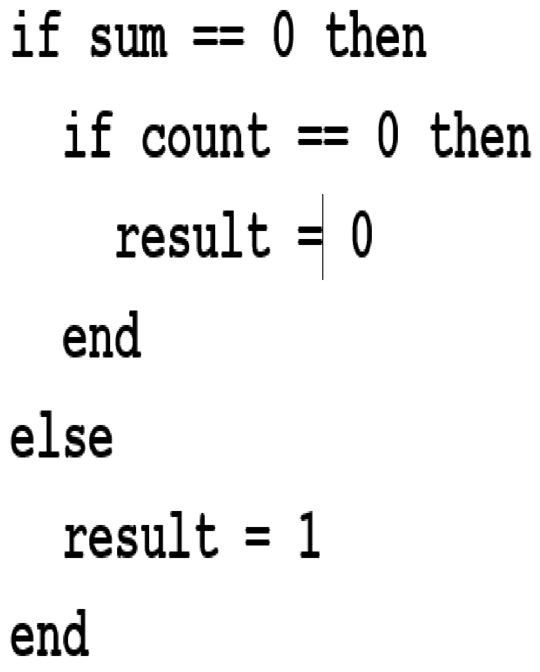
- The above solution is used in C, C++, and C#

Copyright © 2017 Pearson Education, Ltd. All rights reserved.

Nesting Selectors (continued)

- Statement sequences as clauses: Ruby

```
if sum == 0 then
   if count == 0 then
      result = 0
   else
      result = 1
   end
end
```



```
if sum == 0 then
  if count == 0 then
    result = 0
  end
else
  result = 1
end
```

Copyright © 2017 Pearson Education, Ltd. All rights reserved.

Nesting Selectors (continued)

- Python

```
if sum == 0 :
  if count == 0 :
    result = 0
  else :
    result = 1
```

Copyright © 2017 Pearson Education, Ltd. All rights reserved.

Selector Expressions

- In ML, F#, and Lisp, the selector is an expression; in F#:

```
let y =
  if x > 0 then x
  else 2 * x
```

  - If the `if` expression returns a value, there must be an else clause (the expression could produce a unit type, which has no value). The types of the values returned by then and else clauses must be the same.

Copyright © 2017 Pearson Education, Ltd. All rights reserved.

Multiple-Way Selection Statements

- Allow the selection of one of any number of statements or statement groups
- Design Issues:
  1. What is the form and type of the control expression?
  2. How are the selectable segments specified?
  3. Is execution flow through the structure restricted to include just a single selectable segment?
  4. How are case values specified?
  5. What is done about unrepresented expression values?

Copyright © 2017 Pearson Education, Ltd. All rights reserved.

Multiple-Way Selection: Examples

- C, C++, Java, and JavaScript

```
switch (expression) {
   case const_expr_1: stmt_1;
   …
   case const_expr_n: stmt_n;
   [default: stmt_n+1]
}
```

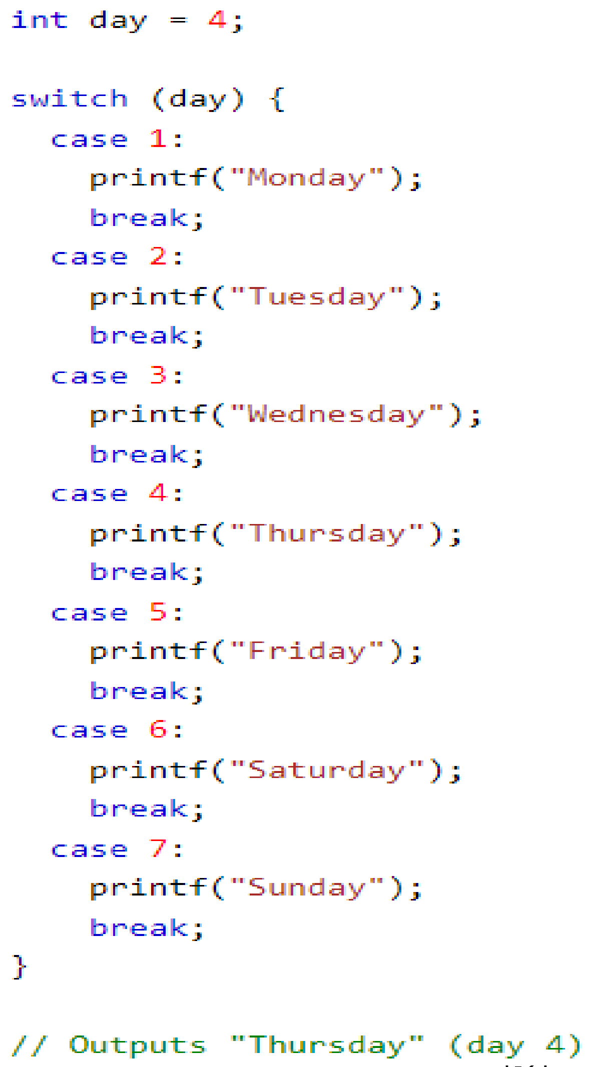
Switch in C, C++, Java

Note the use of case labels inside if statements is supported in C++, but not Java.

```
switch (x)
  default:
    if (prime(x))
      case 2: case 3: case 5: case 7:
        process_prime(x);
    else
      case 4: case 6: case 8:
      case 9: case 10:
        process_composite(x);
```

Multiple-Way Selection: Examples

- Design choices for C's **`switch`** statement
  1. Control expression can be only an integer type
  2. Selectable segments can be statement sequences, blocks, or compound statements
  3. Any number of segments can be executed in one execution of the construct (*there is no implicit branch at the end of selectable segments*)
  4. **`default`** clause is for unrepresented values (if there is no **`default`**, the whole statement does nothing)

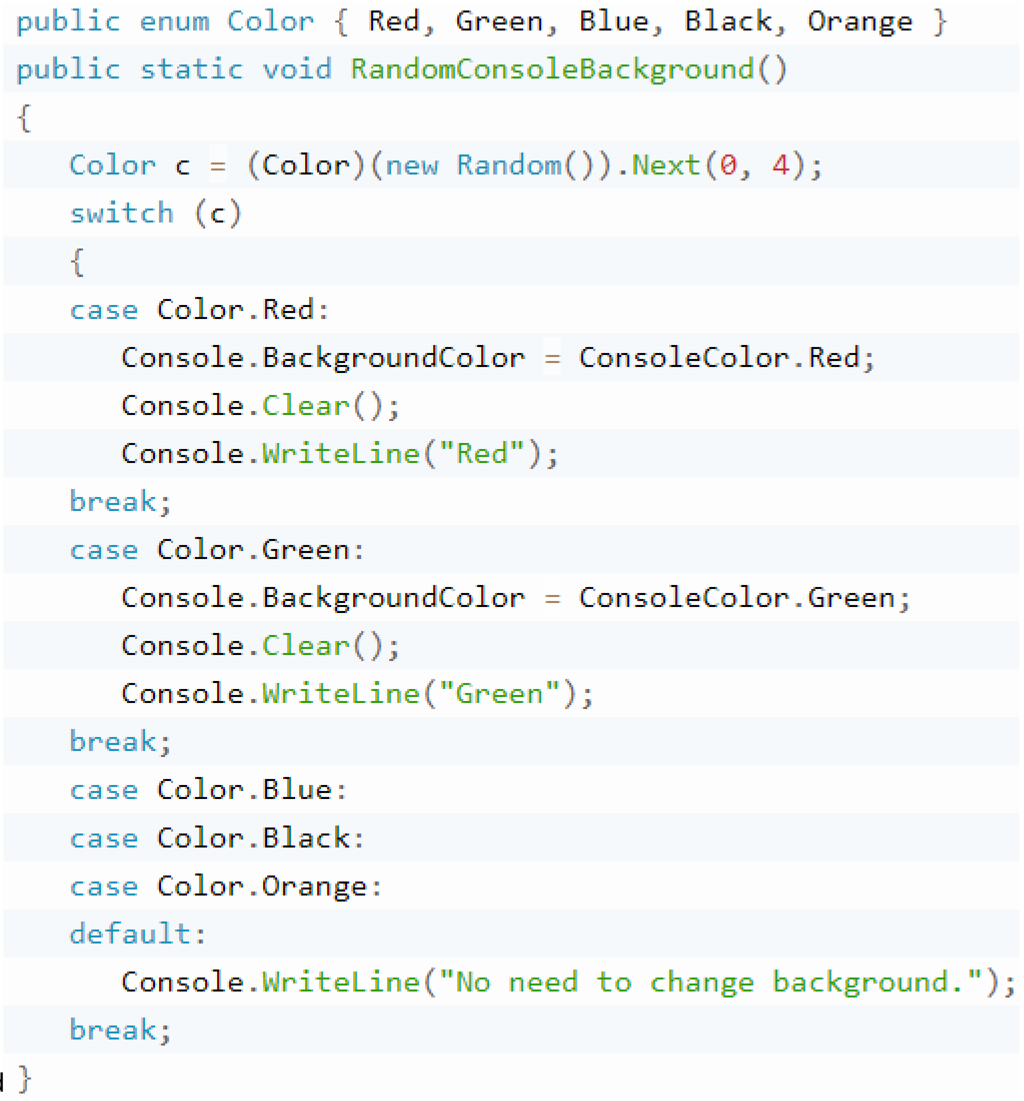


```
int day = 4;

switch (day) {
  case 1:
    printf("Monday");
    break;
  case 2:
    printf("Tuesday");
    break;
  case 3:
    printf("Wednesday");
    break;
  case 4:
    printf("Thursday");
    break;
  case 5:
    printf("Friday");
    break;
  case 6:
    printf("Saturday");
    break;
  case 7:
    printf("Sunday");
    break;
}

// Outputs "Thursday" (day 4)
```

Copyright © 2017 Pearson Education, Ltd. All rights reserved.

Multiple-Way Selection: Examples

- C#
  - Differs from C in that it has a static semantics rule that disallows the implicit execution of more than one segment
  - Each selectable segment must end with an unconditional branch (**goto** or **break**)
  - Also, in C# the control expression and the case constants can be strings



```
public enum Color { Red, Green, Blue, Black, Orange }
public static void RandomConsoleBackground()
{
   Color c = (Color)(new Random()).Next(0, 4);
   switch (c)
   {
   case Color.Red:
      Console.BackgroundColor = ConsoleColor.Red;
      Console.Clear();
      Console.WriteLine("Red");
   break;
   case Color.Green:
      Console.BackgroundColor = ConsoleColor.Green;
      Console.Clear();
      Console.WriteLine("Green");
   break;
   case Color.Blue:
   case Color.Black:
   case Color.Orange:
   default:
      Console.WriteLine("No need to change background.");
   break;
}
```

Copyright © 2017 Pearson Education, Ltd. All rights reserved

Multiple-Way Selection in C#

- It has a static semantics rule that disallows the implicit execution of more than one segment
  - Each selectable segment must end with an unconditional branch (**`goto`** or **`break`**)
- The control expression and the case constants can be strings

```
switch (value) {
  case -1:    Negatives++;     break;
  case 0:     Zeros++;         goto case 1;
  case 1:     Positives++;     break;
  default: Console.WriteLine(“!!!\n”); }
```

Multiple-Way Selection: Examples

- Ruby has two forms of case statements–we'll cover only one

```
leap = case
    when year % 400 == 0 then true
    when year % 100 == 0 then false
    else year % 4 == 0
    end
```

Multiple-Way Selection in Ada

- Ada

```
case expression is
 when choice list => stmt_sequence;
 …
 when choice list => stmt_sequence;
 when others => stmt_sequence;]
end case;
```

- More reliable than C’s **`switch`**
  - Once a **`stmt_sequence`** execution is completed, control is passed to the first statement after the **`case`** statement

Lisp Multiple Selection – `cond`

```
(defun test (x)
   (cond
   ((< x 0) ;test
        (terpri) (princ “It’s negative"))
               ; actions to perform if test is true
   ((> x 100)
        (terpri) (princ "It's huge"))
   (t
        (terpri) (princ “It’s reasonable"))))
```

- Only the 1st true test and following expressions within the corresponding clause are evaluated
  - Safe – like ADA
- GOTOs not needed

Implementing Multiple Selectors

- Approaches:
  - Multiple conditional branches
  - Store case values in a table and use a linear search of the table
  - When there are more than ten cases, a hash table of case values can be used
  - If the number of cases is small and more than half of the whole range of case values are represented, an array whose indices are the case values and whose values are the case labels can be used

Copyright © 2017 Pearson Education, Ltd. All rights reserved.

Multiple-Way Selection Using **`if`**

- Multiple Selectors can appear as direct extensions to two-way selectors, using else-if clauses, for example in Python:

```
if count < 10 :
  bag1 = True
elif count < 100 :
  bag2 = True
elif count < 1000 :
  bag3 = True
```

Copyright © 2017 Pearson Education, Ltd. All rights reserved.

Multiple-Way Selection Using **`if`**

- The Python example can be written as a Ruby **`case`**

```
case
  when count < 10 then bag1 = true
  when count < 100 then bag2 = true
  when count < 1000 then bag3 = true
end
```

Copyright © 2017 Pearson Education, Ltd. All rights reserved.

Scheme's Multiple Selector

- General form of a call to `COND`:

```
(COND
  (predicate₁ expression₁)
   …
  (predicateₙ expressionₙ)
  [(ELSE expressionₙ₊₁)]
 )
```

  - The `ELSE` clause is optional; `ELSE` is a synonym for true
  - Each predicate-expression pair is a parameter
  - Semantics: The value of the evaluation of `COND` is the value of the expression associated with the first predicate expression that is true

Copyright © 2017 Pearson Education, Ltd. All rights reserved.

Iterative Statements

- The repeated execution of a statement or compound statement is accomplished either by iteration or recursion
- General design issues for iteration control statements:
  1. How is iteration controlled?
  2. Where is the control mechanism in the loop?

Copyright © 2017 Pearson Education, Ltd. All rights reserved.

Counter-Controlled Loops

- A counting iterative statement has a loop variable, and a means of specifying the *initial* and *terminal*, and *stepsize* values
- Design Issues:

1. What are the type and scope of the loop variable?
2. Should it be legal for the loop variable or loop parameters to be changed in the loop body, and if so, does the change affect loop control?
3. Should the loop parameters be evaluated only once, or once for every iteration?

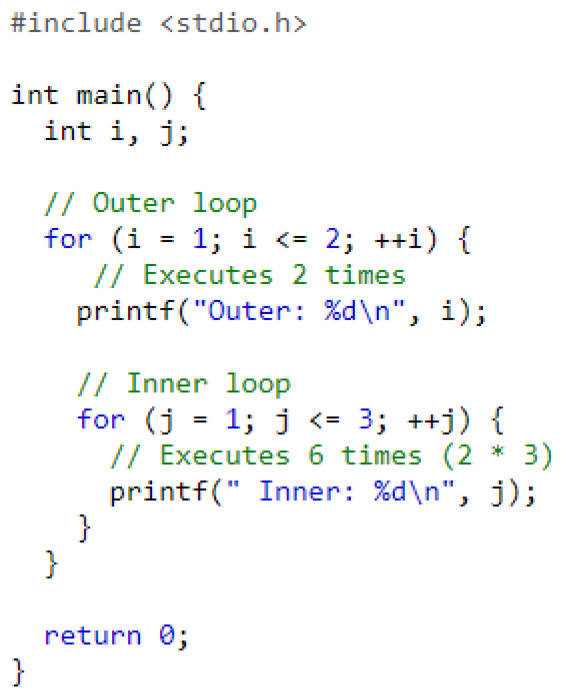
Copyright © 2017 Pearson Education, Ltd. All rights reserved.

Counter-Controlled Loops: Examples

- C-based languages

  ```
  for ([expr_1] ; [expr_2] ; [expr_3])
     statement
  ```

  – The expressions can be whole statements, or even statement sequences, with the statements separated by commas
  - The value of a multiple-statement expression is the value of the last statement in the expression
  - If the second expression is absent, it is an infinite loop
- Design choices:
  - There is no explicit loop variable
  - Everything can be changed in the loop
  - The first expression is evaluated once, but the other two are evaluated with each iteration
  - It is legal to branch into the body of a for loop in C



```
#include <stdio.h>

int main() {
  int i, j;

  // Outer loop
  for (i = 1; i <= 2; ++i) {
     // Executes 2 times
    printf("Outer: %d\n", i);

    // Inner loop
    for (j = 1; j <= 3; ++j) {
      // Executes 6 times (2 * 3)
      printf(" Inner: %d\n", j);
    }
  }

  return 0;
}
```

Counter-Controlled Loops: Examples

- C++ differs from C in two ways:
  1. The control expression can also be Boolean
  2. The initial expression can include variable definitions (scope is from the definition to the end of the loop body)
- Java and C#
  - Differs from C++ in that the control expression must be Boolean

Copyright © 2017 Pearson Education, Ltd. All rights reserved.

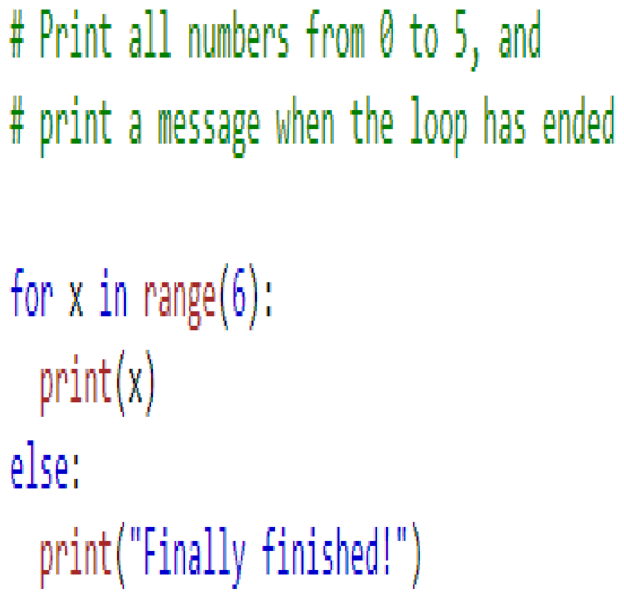
Counter-Controlled Loops: Examples

- Python

  for loop_variable **in** object:
  - loop body

  [**else**:
  - else clause]

  - The object is often a range, which is either a list of values in brackets (`[2, 4, 6]`), or a call to the range function (**range**`(5)`, which returns `0, 1, 2, 3, 4`
  - The loop variable takes on the values specified in the given range, one for each iteration
  - The else clause, which is optional, is executed if the loop terminates normally



```
# Print all numbers from 0 to 5, and
# print a message when the loop has ended

for x in range(6):
  print(x)
else:
  print("Finally finished!")
```

Copyright © 2017 Pearson Education, Ltd. All rights reserved.

Counter-Controlled Loops: Examples

- F#
  - Because counters require variables, and functional languages do not have variables, counter-controlled loops must be simulated with recursive functions

```
let rec forLoop loopBody reps =
  if reps <= 0 then ()
  else
    loopBody()
    forLoop loopBody, (reps - 1)
```

  - This defines the recursive function `forLoop` with the parameters `loopBody` (a function that defines the loop’s body) and the number of repetitions
  - `()` means do nothing and return nothing

Copyright © 2017 Pearson Education, Ltd. All rights reserved.

Logically-Controlled Loops

- Repetition control is based on a Boolean expression
- Design issues:
  - Pretest or posttest?
  - Should the logically controlled loop be a special case of the counting loop statement or a separate statement?

Copyright © 2017 Pearson Education, Ltd. All rights reserved.

Logically-Controlled Loops: Examples

- C and C++ have both pretest and posttest forms, in which the control expression can be arithmetic:

```
while (control_expr)          do
    loop body                     loop body
                              while (control_expr)
```

  - In both C and C++ it is legal to branch into the body of a logically-controlled loop

- Java is like C and C++, except the control expression must be Boolean (and the body can only be entered at the beginning -- Java has no `goto`

Copyright © 2017 Pearson Education, Ltd. All rights reserved.

Logically-Controlled Loops: Examples

- F#

```
// Recursive function defintion:
let rec function-name parameter-list =
function-body
// recursive fibonacci
let rec fib n =
        if n <= 2 then 1
        else fib (n - 1) + fib (n - 2)
```

  - As with counter-controlled loops, logically-controlled loops can be simulated with recursive functions

```
let rec whileLoop test body =
  if test() then
    body()
    whileLoop test body
  else ()
```

  - This defines the recursive function `whileLoop` with parameters `test` and `body`, both functions. `test` defines the control expression

User-Located Loop Control Mechanisms

- Sometimes it is convenient for the programmers to decide a location for loop control (other than top or bottom of the loop)
- Simple design for single loops (e.g., `break`)
- Design issues for nested loops

1. Should the conditional be part of the exit?
2. Should control be transferable out of more than one loop?

Copyright © 2017 Pearson Education, Ltd. All rights reserved.

User-Located Loop Control

- Exit statement:
  - ❖Unconditional unlabeled exit: **break** (C, C++)

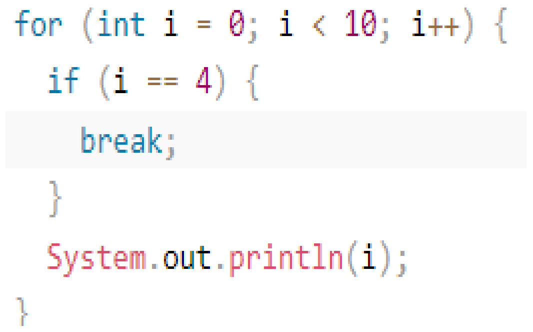
```
for (index=0; index<10; index++) {
        ...
    if (value < 0) break;
}
```

  - ❖Unconditional labeled exit: **break** (Java, C#), **last** (Perl)

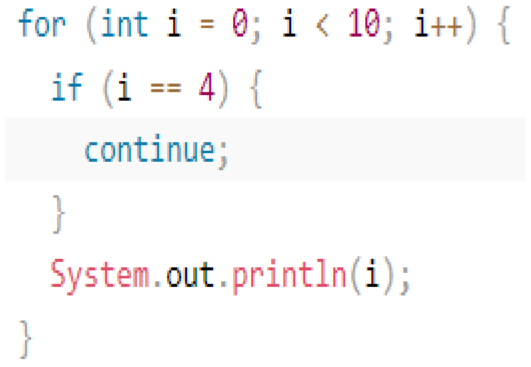
```
C#:  outerloop:  for (row=0; row<numRows; row++)
                     for (col = 0; col < numCols; col++) {
                        sum += matrix[row][col];
                        if (sum > 1000)
                           break outerLoop;
                     }

Perl:     LINE: while (<STDIN>) {
                   last LINE if /^$/;
                          ...
                }
```

User-Located Loop Control Mechanisms

- C , C++, Python, Ruby, and C# have unconditional unlabeled exits (**`break`**)
- Java and Perl have unconditional labeled exits (**`break`** in Java, **`last`** in Perl)
- C, C++, and Python have an unlabeled control statement, **`continue`**, that skips the remainder of the current iteration, but does not exit the loop
- Java and Perl have labeled versions of **`continue`**



```
for (int i = 0; i < 10; i++) {
  if (i == 4) {
    break;
  }
  System.out.println(i);
}
```

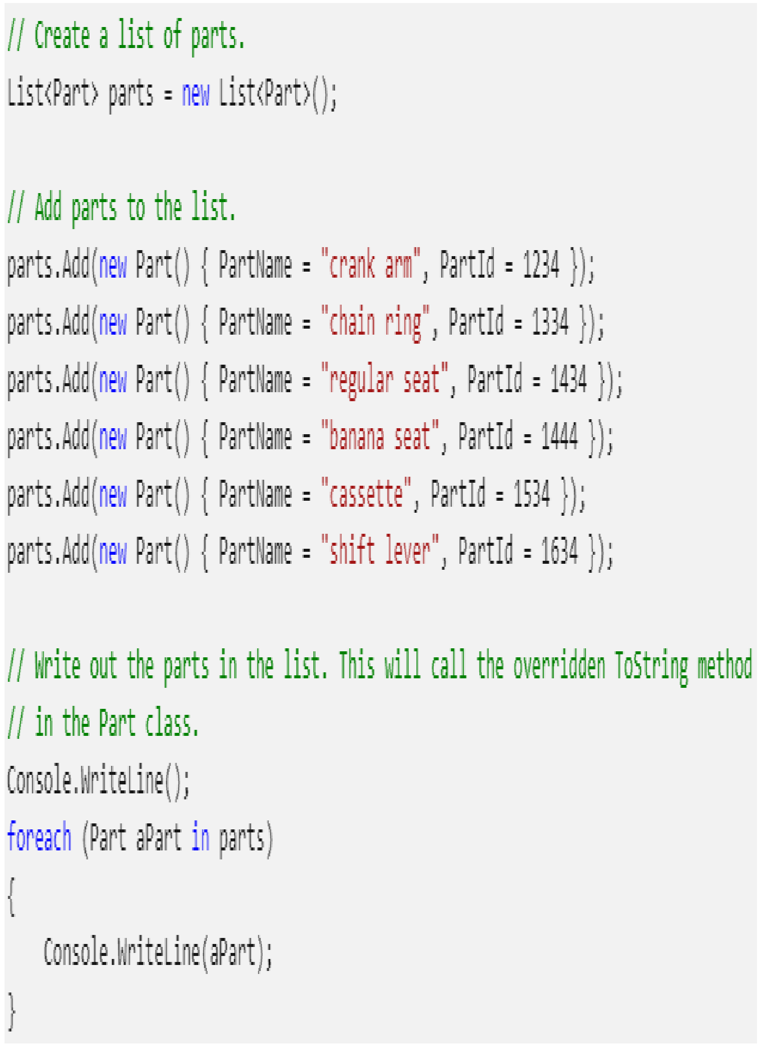


```
for (int i = 0; i < 10; i++) {
  if (i == 4) {
    continue;
  }
  System.out.println(i);
}
```

Iteration Based on Data Structures

- The number of elements in a data structure controls loop iteration
- Control mechanism is a call to an *iterator* function that returns the next element in some chosen order, if there is one; else loop is terminate
- C's **`for`** can be used to build a user-defined iterator:

```
for (p=root; p==NULL; traverse(p)){
   ...
}
```



```
// Create a list of parts.
List<Part> parts = new List<Part>();

// Add parts to the list.
parts.Add(new Part() { PartName = "crank arm", PartId = 1234 });
parts.Add(new Part() { PartName = "chain ring", PartId = 1334 });
parts.Add(new Part() { PartName = "regular seat", PartId = 1434 });
parts.Add(new Part() { PartName = "banana seat", PartId = 1444 });
parts.Add(new Part() { PartName = "cassette", PartId = 1534 });
parts.Add(new Part() { PartName = "shift lever", PartId = 1634 });

// Write out the parts in the list. This will call the overridden ToString method
// in the Part class.
Console.WriteLine();
foreach (Part aPart in parts)
{
    Console.WriteLine(aPart);
}
```

Iteration Based on Data Structures (continued)

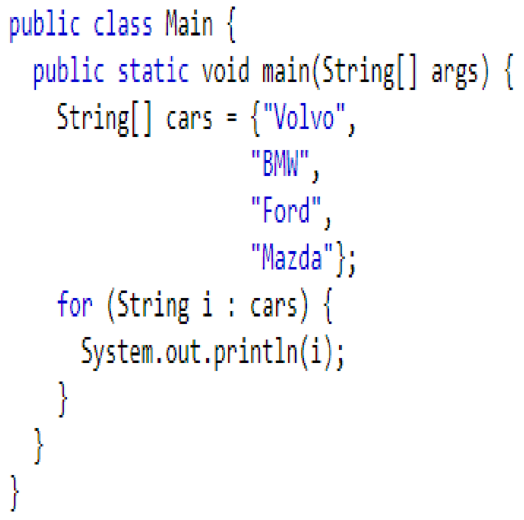
- PHP
  - `current` points at one element of the array
  - `next` moves `current` to the next element
  - `reset` moves `current` to the first element

```
reset $list;
print(“1st: “+current($list) + “<br />”);
while($current_value = next($list))
    print(“next: “+$current_value+”<br />”);
```

- Java 5.0 (uses **for**, although it is called foreach)

  For arrays and any other class that implements the `Iterable` interface, e.g., `ArrayList`

```
    for (String myElement : myList) { … }
```



```
public class Main {
  public static void main(String[] args) {
    String[] cars = {"Volvo",
                     "BMW",
                     "Ford",
                     "Mazda"};
    for (String i : cars) {
      System.out.println(i);
    }
  }
}
```

Copyright © 2017 Pearson Education, Ltd. All rights reserved.

Iteration Based on Data Structures (continued)

- C# and F# (and the other .NET languages) have generic library classes, like Java 5.0 (for arrays, lists, stacks, and queues). Can iterate over these with the **foreach** statement. User-defined collections can implement the IEnumerator interface and also use **foreach**.

```
List<String> names = new List<String>();
 names.Add("Bob");
 names.Add("Carol");
 names.Add("Ted");
 foreach (Strings name in names)
     Console.WriteLine ("Name: {0}", name);
```

String[] strList = {“Bob”, “John”, “Carol” };
foreach (String name in strList)

Copyright © 2017 Pearson Education, Ltd. All rights reserved.

Iteration Based on Data Structures (continued)

- Ruby *blocks* are sequences of code, delimited by either braces or **`do`** and **`end`**
  - Blocks can be used with methods to create iterators
  - Predefined iterator methods (`times`, `each`, `upto`):

    ```
    3.times {puts "Hey!"}
    list.each {|value| puts value}
    ```

    (`list` is an array; `value` is a block parameter)

    ```
    1.upto(5) {|x| print x, " "}
    ```

  - Iterators are implemented with blocks, which can also be defined by applications

```
>> 4.times {puts "Hey!"}
Hey!
Hey!
Hey!
Hey!
=> 4
```

```
>> list = [2, 4, 6, 8]
=> [2, 4, 6, 8]
>> list.each {|value| puts value}
2
4
6
8
=> [2, 4, 6, 8]
```

```
1.upto(5) {|x| print x, " "}
```

This produces the following output:

```
1 2 3 4 5
```

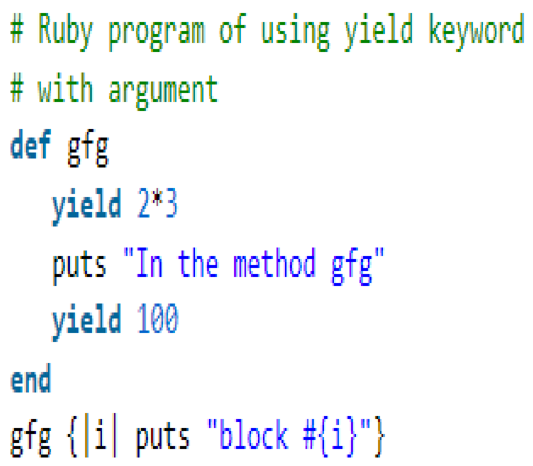
Copyright © 2017 Pearson Education, Ltd. All rights reserved.

Iteration Based on Data Structures (continued)

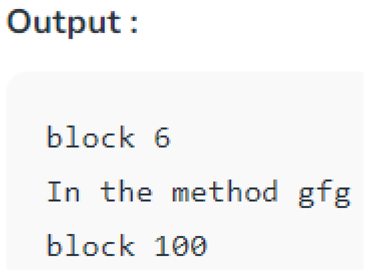
- Ruby blocks are attached methods calls; they can have parameters (in vertical bars); they are executed when the method executes a **yield** statement

```
def fibonacci(last)
  first, second = 1, 1
  while first <= last
    yield first
    first, second = second, first + second
  end
end
puts "Fibonacci numbers less than 100 are:"
fibonacci(100) {|num| print num, " "}
puts
```

– Ruby has a **for** statement, but Ruby converts them to `upto` method calls



```
# Ruby program of using yield keyword
# with argument
def gfg
   yield 2*3
   puts "In the method gfg"
   yield 100
end
gfg {|i| puts "block #{i}"}
```

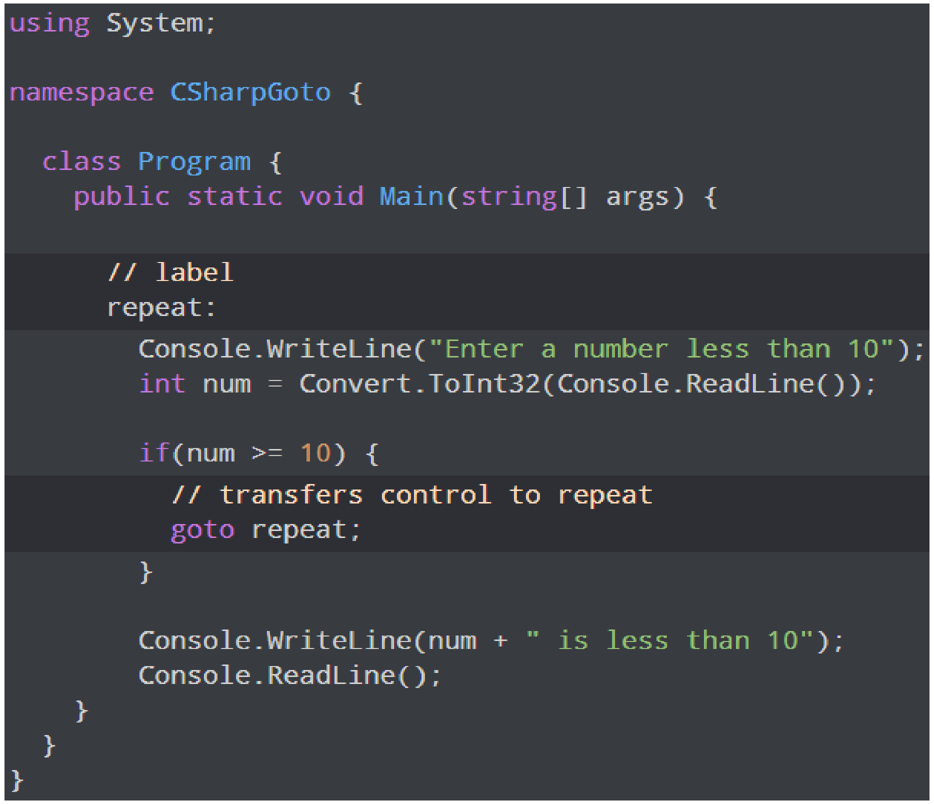


Output :

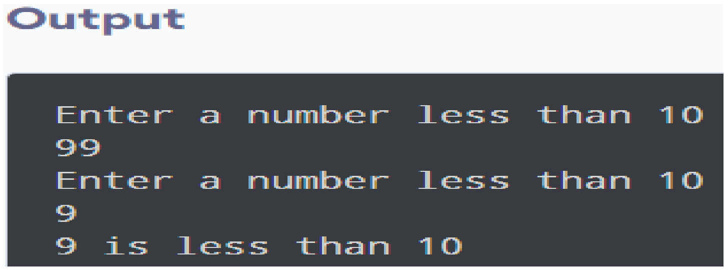
```
block 6
In the method gfg
block 100
```

Unconditional Branching

- Transfers execution control to a specified place in the program
- Represented one of the most heated debates in 1960's and 1970's
- Major concern: Readability
- Some languages do not support **goto** statement (e.g., Java)
- C# offers **goto** statement (can be used in **switch** statements)
- Loop exit statements are restricted and somewhat camouflaged **goto**'s



```
using System;

namespace CSharpGoto {

  class Program {
    public static void Main(string[] args) {

      // label
      repeat:
        Console.WriteLine("Enter a number less than 10");
        int num = Convert.ToInt32(Console.ReadLine());

        if(num >= 10) {
          // transfers control to repeat
          goto repeat;
        }

        Console.WriteLine(num + " is less than 10");
        Console.ReadLine();
    }
  }
}
```



Output

```
Enter a number less than 10
99
Enter a number less than 10
9
9 is less than 10
```

Copyright © 2017 Pearson Education, Ltd. All rights reserved.

Guarded Commands

- Designed by Dijkstra
- Purpose: to support a new programming methodology that supported verification (correctness) during development
- Basis for two linguistic mechanisms for concurrent programming (in CSP)
- Basic Idea: if the order of evaluation is not important, the program should not specify one

Copyright © 2017 Pearson Education, Ltd. All rights reserved.

Selection Guarded Command

- Form

```
if <Boolean expr> -> <statement>
[] <Boolean expr> -> <statement>
  ...
[] <Boolean expr> -> <statement>
fi
```

- Semantics: when construct is reached,
  - Evaluate all Boolean expressions
  - If more than one are true, choose one non-deterministically
  - If none are true, it is a runtime error

Copyright © 2017 Pearson Education, Ltd. All rights reserved.

Loop Guarded Command

- **Form**

```
do <Boolean> -> <statement>
 [] <Boolean> -> <statement>
  ...
 []  <Boolean> -> <statement>
od
```

- Semantics: for each iteration
  - Evaluate all Boolean expressions
  - If more than one are true, choose one non-deterministically; then start loop again
  - If none are true, exit loop

Copyright © 2017 Pearson Education, Ltd. All rights reserved.

Guarded Commands: Rationale

- Connection between control statements and program verification is intimate
- Verification is impossible with `goto` statements
- Verification is possible with only selection and logical pretest loops
- Verification is relatively simple with only guarded commands

Copyright © 2017 Pearson Education, Ltd. All rights reserved.

Conclusions

- Variety of statement-level structures
- Choice of control statements beyond selection and logical pretest loops is a trade-off between language size and writability
- Functional and logic programming languages use quite different control structures

Copyright © 2017 Pearson Education, Ltd. All rights reserved.