Flow of Control

OOP

Lecture 3

Objectives

- Use Java branching statements
- Compare values of primitive types
- Compare objects such as strings
- Use the primitive type `boolean`

Outline

- The `if-else` Stetement
- The Type `boolean`
- The `switch` statement

Flow of Control

- *Flow of control* is the order in which a program performs actions.
  - Up to this point, the order has been sequential.
- A *branching statement* chooses between two or more possible actions.
- A *loop statement* repeats an action until a stopping condition occurs.

The `if-else` Statement: Outline

- Basic `if-else` Statement
- Boolean Expressions
- Comparing Strings
- Nested `if-else` Statements
- Multibranch `if-else` Statements
- The `switch` Statament
- (optional) The Conditional Operator
- The `exit` Method

The `if-else` Statement

- A branching statement that chooses between two possible actions.
- Syntax

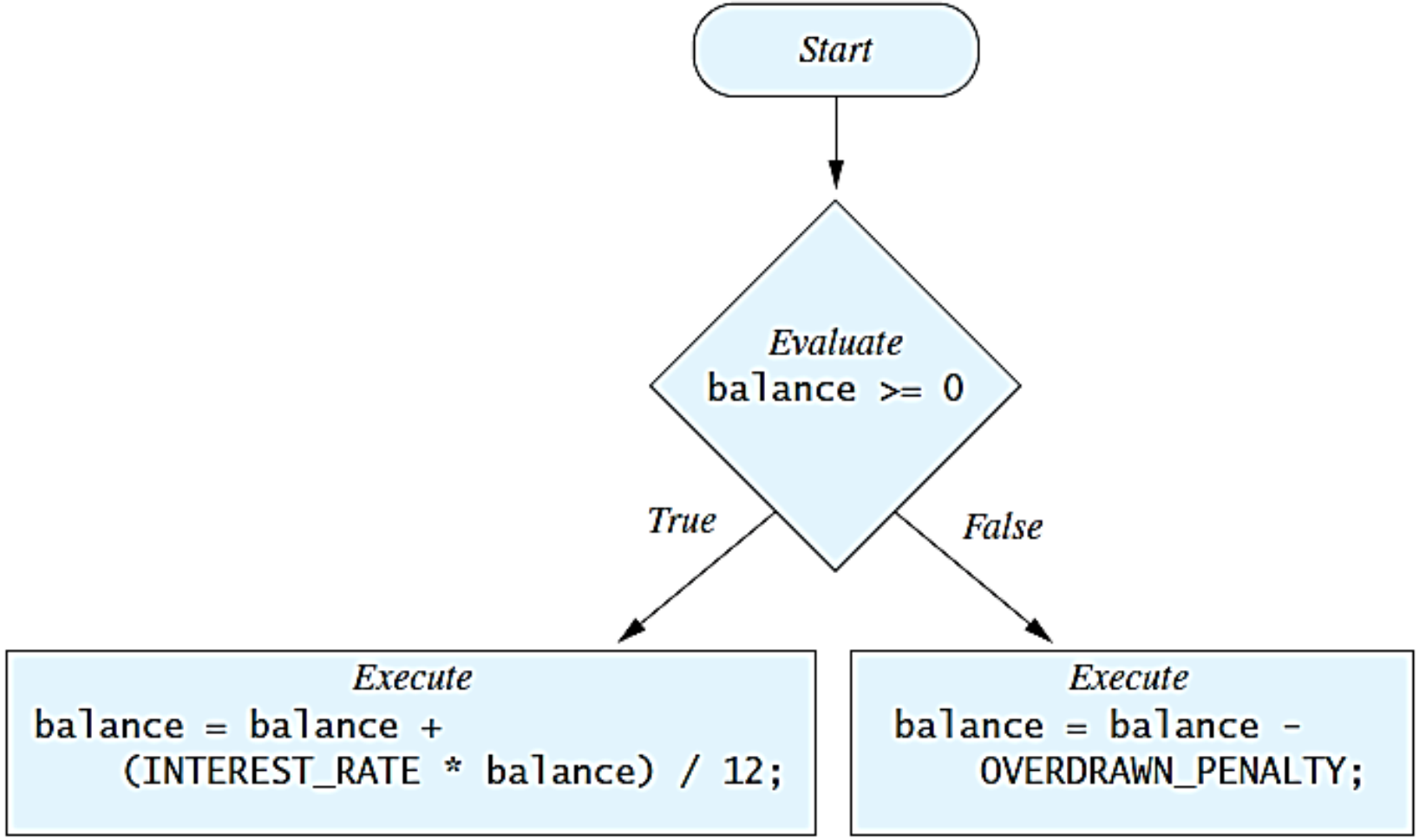
```
if (Boolean_Expression)
  Statement_1
else
  Statement_2
```

The `if-else` Statement

- Example

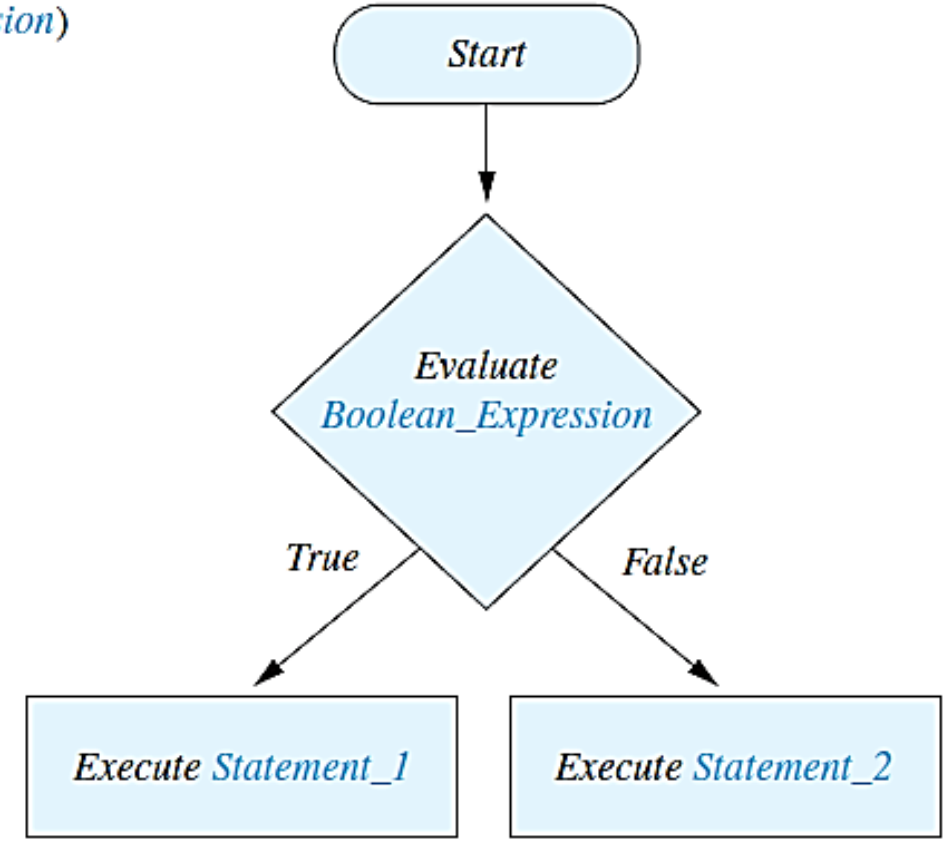
```
if (balance >= 0)
   balance = balance + (INTEREST_RATE * balance) / 12;
else
   balance = balance - OVERDRAWN_PENALTY;
```

The `if-else` Statement



Semantics of the `if-else` Statement

```
if (Boolean_Expression)
    Statement_1
else
    Statement_2
```



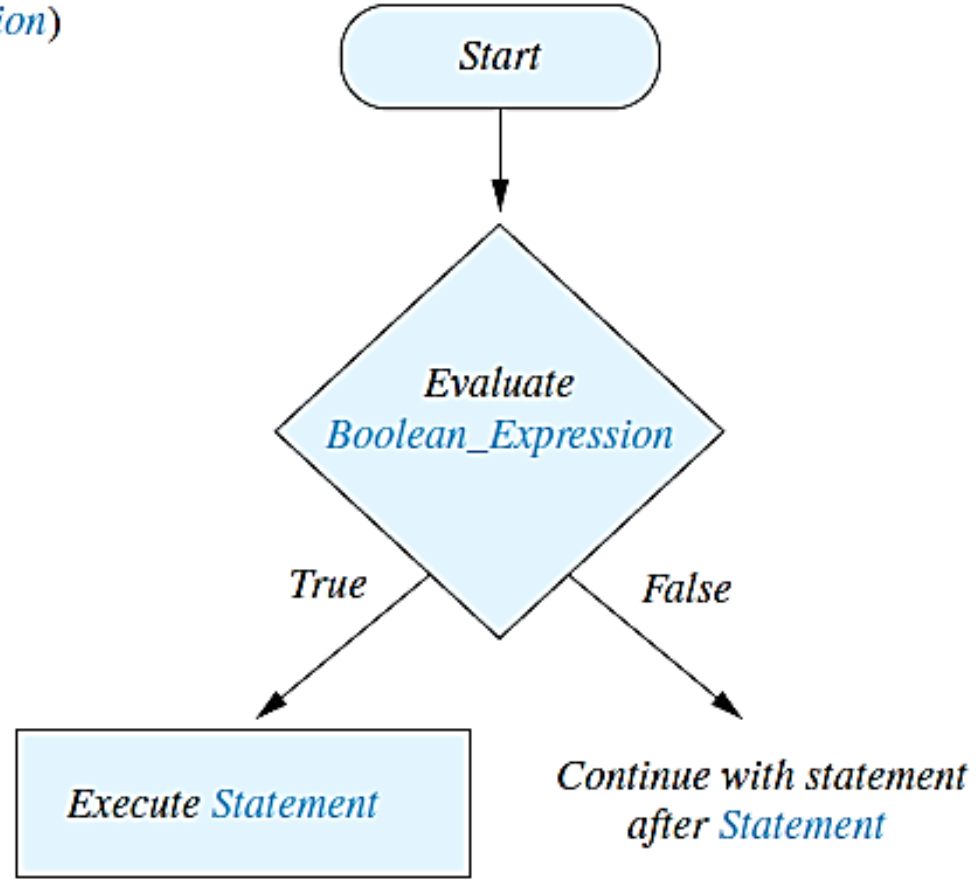
Compound Statements

- To include multiple statements in a branch, enclose the statements in braces.

```
if (count < 3)
{
        total = 0;
        count = 0;
}
```

Omitting the `else` Part

```
if (Boolean_Expression)
    Statement
```



Introduction to Boolean Expressions

- The value of a *boolean expression* is either `true` or `false`.
- Examples

```
time < limit
balance <= 0
```

Java Comparison Operators

Math Notation	Name	Java Notation	Java Examples
=	Equal to	==	`balance == 0` `answer == 'y'`
≠	Not equal to	!=	`income != tax` `answer != 'y'`
>	Greater than	>	`expenses > income`
≥	Greater than or equal to	>=	`points >= 60`
<	Less than	<	`pressure < max`
≤	Less than or equal to	<=	`expenses <= income`

Compound Boolean Expressions

- Boolean expressions can be combined using the "and" `(&&)` operator.
- Example

```
if ((score > 0) && (score <= 100))
...
```

- Not allowed

```
if (0 < score <= 100)
...
```

Compound Boolean Expressions

- Syntax

```
(Sub_Expression_1) && (Sub_Expression_2)
```

- Parentheses often are used to enhance readability.
- The larger expression is true only when both of the smaller expressions are true.

Compound Boolean Expressions

- Boolean expressions can be combined using the "or" `(||)` operator.
- Example

```
if ((quantity > 5) || (cost < 10))
...
```

- Syntax

```
(Sub_Expression_1) || (Sub_Expression_2)
```

Compound Boolean Expressions

- The larger expression is true
  - When either of the smaller expressions is true
  - When both of the smaller expressions are true.
- The Java version of "or" is the *inclusive or* which allows either or both to be true.
- The *exclusive or* allows one or the other, but not both to be true.

Negating a Boolean Expression

- A boolean expression can be negated using the "not" (`!`) operator.
- Syntax

  ```
  !(Boolean_Expression)
  ```

- Example

  ```
  (a || b) && !(a && b)
  ```

  which is the *exclusive or*

Negating a Boolean Expression

! (*A Op B*) Is Equivalent to (*A Op B*)	
<	>=
<=	>
>	<=
>=	<
==	!=
!=	==

Java Logical Operators

Name	Java Notation	Java Examples
Logical *and*	&&	`(sum > min) && (sum < max)`
Logical *or*	||	`(answer == 'y') || (answer == 'Y')`
Logical *not*	!	`!(number < 0)`

Boolean Operators

Value of *A*	Value of *B*	Value of *A* && *B*	Value of *A* || *B*	Value of ! (*A*)
true	true	true	true	false
true	false	false	true	false
false	true	false	true	true
false	false	false	false	true

Using ==

- == is appropriate for determining if two integers or characters have the same value.

  `if (a == 3)`

  where `a` is an integer type

- == is not appropriate for determining if two objects have the same value.
  - `if (s1 == s2),` where `s1` and `s2` refer to strings, determines only if s1 and s2 refer the a common memory location.
  - If `s1` and `s2` refer to strings with identical sequences of characters, but stored in different memory locations, `(s1 == s2)` is false.

Using ==

- To test the equality of objects of class String, use method **`equals`**.

  ```
  s1.equals(s2)
  ```

  or

  ```
  s2.equals(s1)
  ```

- To test for equality ignoring case, use method **`equalsIgnoreCase`**.

  ```
  ("Hello".equalsIgnoreCase("hello"))
  ```

- Syntax

  ```
  String.equals(Other_String)
  String.equalsIgnoreCase(Other_String)
  ```

Nested `if-else` Statements

- An `if-else` statement can contain any sort of statement within it.
- In particular, it can contain another `if-else` statement.
  - An `if-else` may be nested within the "if" part.
  - An `if-else` may be nested within the "else" part.
  - An `if-else` may be nested within both parts.

Nested Statements

- Syntax

```
if (Boolean_Expression_1)
    if (Boolean_Expression_2)
        Statement_1)
    else
        Statement_2)
else
   if (Boolean_Expression_3)
        Statement_3)
    else
        Statement_4);
```

Nested Statements

- Each `else` is paired with the nearest unmatched `if`.
- **If used properly**, indentation communicates which `if` goes with which `else`.
- Braces can be used like parentheses to group statements.

Nested Statements

- Subtly different forms

First Form

```
if (a > b)
{
    if (c > d)
        e = f
}
    else
        g = h;
```

Second Form

```
if (a > b)
   if (c > d)
        e = f
    else
        g = h;
```

Compound Statements

- When a list of statements is enclosed in braces (`{}`), they form a single *compound statement*.
- Syntax

```
{
    Statement_1;
    Statement_2;
    …
}
```

- A compound statement can be used wherever a statement can be used.
- Example

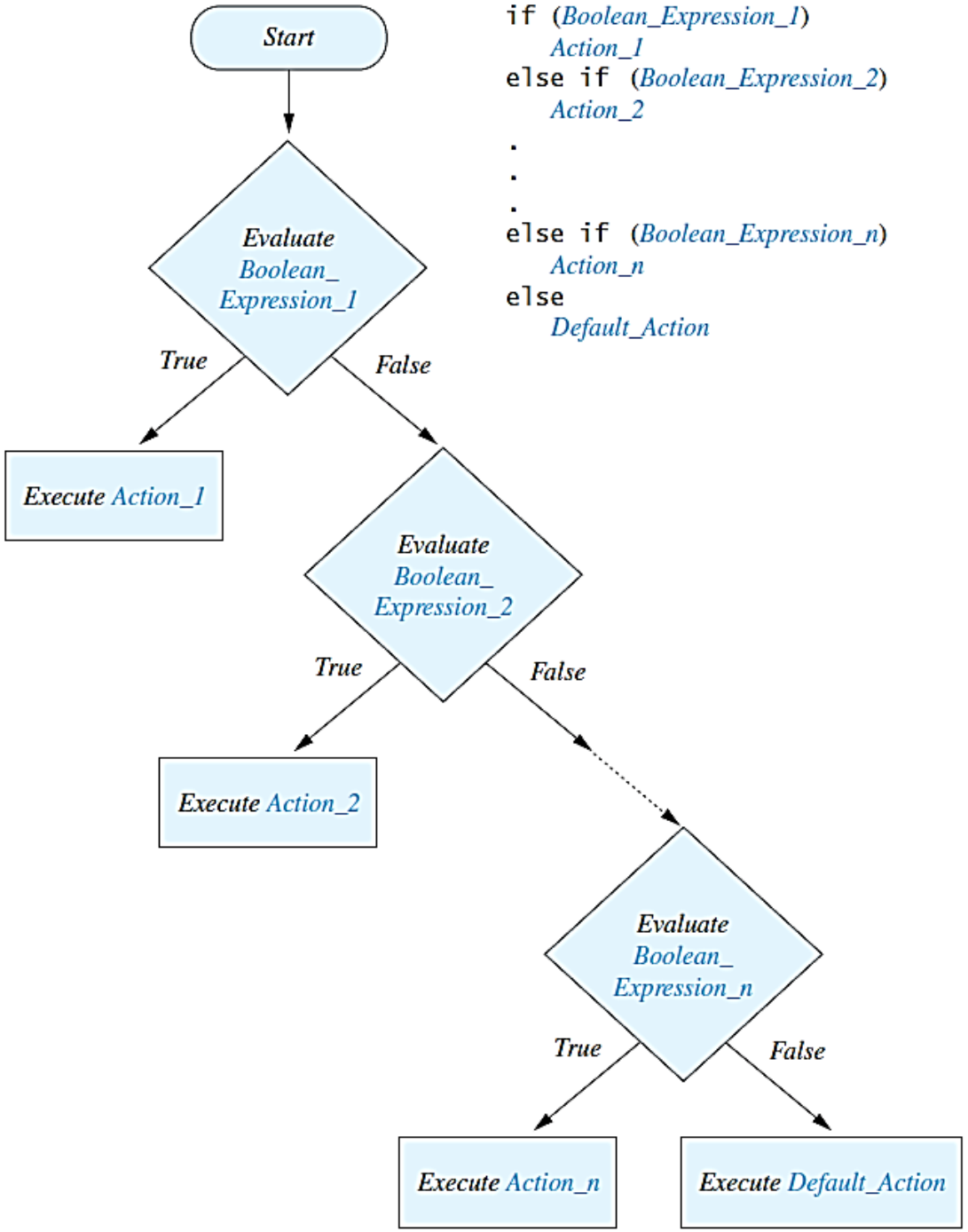
```
if (total > 10)
{
    sum = sum + total;
    total = 0;
}
```

Multibranch `if-else` Statements

- Syntax

```
if (Boolean_Expression_1)
    Statement_1
else if (Boolean_Expression_2)
    Statement_2
else if (Boolean_Expression_3)
    Statement_3
else if …
else
    Default_Statement
```

Multibranch `if-else` Statements




```
if (Boolean_Expression_1)
    Action_1
else if (Boolean_Expression_2)
    Action_2
.
.
.
else if (Boolean_Expression_n)
    Action_n
else
    Default_Action
```

Multibranch `if-else` Statements

- Equivalent code

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

The Conditional Operator

```
if (n1 > n2)
   max = n1;
else
   max = n2;
```

can be written as

```
max = (n1 > n2) ? n1 : n2;
```

- The `?` and `:` together are call the *conditional operator* or *ternary operator*.

The `exit` Method

- Sometimes a situation arises that makes continuing the program pointless.
- A program can be terminated normally by

```
System.exit(0).
```

The `exit` Method

- Example

```
if (numberOfWinners == 0)
{
    System.out.println ("Error: Dividing by
  zero.");
    System.exit (0);
}
else
{
    oneShare = payoff / numberOfWinners;
    System.out.println ("Each winner will receive
  $" + oneShare);
}
```

The Type `boolean`

- The type `boolean` is a primitive type with only two values: `true` and `false`.
- Boolean variables can make programs more readable.

```
if (systemsAreOK)
```

instead of

```
if((temperature <= 100) && (thrust >= 12000) &&
   (cabinPressure > 30) && …)
```

Boolean Expressions and Variables

- Variables, constants, and expressions of type **`boolean`** all evaluate to either **`true`** or **`false`**.
- A boolean variable can be given the value of a boolean expression by using an assignment operator.

```
boolean isPositive = (number > 0);
...
if (isPositive) ...
```

Precedence Rules

- Parentheses should be used to indicate the order of operations.
- When parentheses are omitted, the order of operation is determined by *precedence rules*.
- Operations with *higher precedence* are performed before operations with *lower precedence*.
- Operations with *equal precedence* are done left-to-right (except for unary operations which are done right-to-left).

Precedence Rules

Highest Precedence

First: the unary operators +, −, ++, −−, and !

Second: the binary arithmetic operators *, /, %

Third: the binary arithmetic operators +, −

Fourth: the boolean operators <, >, <=, >=

Fifth: the boolean operators ==, !=

Sixth: the boolean operator &

Seventh: the boolean operator |

Eighth: the boolean operator &&

Ninth: the boolean operator ||

Lowest Precedence

Short-circuit Evaluation

- Sometimes only part of a boolean expression needs to be evaluated to determine the value of the entire expression.
  - If the first operand associated with an `||` is `true`, the expression is `true`.
  - If the first operand associated with an `&&` is `false`, the expression is `false`.
- This is called *short-circuit* or *lazy* evaluation.
- Short-circuit evaluation is not only efficient, sometimes it is essential!
- A run-time error can result, for example, from an attempt to divide by zero.

```
if ((number != 0) && (sum/number > 5))
```

Input and Output of Boolean Values

- Example

```
boolean booleanVar = false;
System.out.println(booleanVar);
System.out.println("Enter a boolean
  value:");
Scanner keyboard = new
  Scanner(System.in);
booleanVar = keyboard.nextBoolean();
System.out.println("You entered " +
  booleanVar);
```

The `switch` Statement

- The `switch` statement is a mutltiway branch that makes a decision based on an *integral* (integer or character) expression.
- The `switch` statement begins with the keyword `switch` followed by an integral expression in parentheses and called the *controlling expression*.
- A list of cases follows, enclosed in braces.
- Each case consists of the keyword `case` followed by
  - A constant called the *case label*
  - A colon
  - A list of statements.
- The list is searched for a case label matching the controlling expression.

The `switch` Statement

- The action associated with a matching case label is executed.
- If no match is found, the case labeled `default` is executed.
  - The `default` case is optional, but recommended, even if it simply prints a message.
- Repeated case labels are not allowed.

The **`switch`** Statement

- Syntax

```
switch (Controlling_Expression)
{
    case Case_Label:
        Statement(s);
        break;
    case Case_Label:
    …
    default:
    …
}
```

The `switch` Statement

- The action for each case typically ends with the word `break`.
- The optional `break` statement prevents the consideration of other cases.
- The controlling expression can be anything that evaluates to an integral type.