Chapter 3: Branches

Motivating Example

- Devise an algorithm to calculate a person's weekly pay. Assume that
  - the person gets paid a fixed *rate* for regular time pay (≤40 hours)
  - the person makes time and a half for working overtime (>40 hours)

Branch Example

- The program must choose among two decisions (with or without overtime pay)
  - A **branch** lets a program choose between two (or more) alternatives
  - The **flow of control** refers to the order in which statements are executed

- To calculate hourly wages there are two choices
  - Regular time (up to and including 40 hours)
    - gross_pay = rate * hours;
  - Overtime (over 40 hours)
    - gross_pay = rate * 40 + 1.5 * rate * (hours - 40);

If-else syntax

```
// Statements that execute before the branches

if (expression) {
        // Statements to execute when the
        // expression is true (first branch)
} else {
        // Statements to execute when the
        // expression is false (second branch)
}

// Statements that execute after the branches
```

Boolean Expressions

- Boolean expressions are expressions that are either true or false
- comparison operators such as '>' (greater than) are used to compare variables and/or numbers
  - (hours > 40) is the boolean expression from the wages example
  - A few of the comparison operators use two symbols (there are no spaces between the symbols!)
    - >= greater than or equal to
    - != not equal or inequality
    - = = equal or equivalent

Relational and equality operators*

Table 3.2.1: Relational (first four) and equality (last two) operators.

Relational and equality operators	Description
a **<** b	a is ***less-than*** b
a **>** b	a is ***greater-than*** b
a **<=** b	a is ***less-than-or-equal-to*** b
a **>=** b	a is ***greater-than-or-equal-to*** b
a **==** b	a is ***equal to*** b
a ***!=*** b	a is ***not equal to*** b

*Different operators are used for String values

Table 3.4.1: Logical operators.

Logical operator	Description
a **&&** b	Logical AND: true when *both* of its operands are true
a **//** b	Logical OR: true when *at least one* of its two operands are true
!a	Logical NOT (opposite): true when its single operand is false (and false when operand is true)

Precedence rules for logical operators

Table 3.4.3: Precedence rules for logical and relational operators.

Convention	Description	Explanation
( )	Items within parentheses are evaluated first.	In `!(age > 16)`, age > 16 is evaluated first, then the logical NOT.
!	Next to be evaluated is **!**.	
*** / % + -**	Arithmetic operator are then evaluated using the precedence rules for those operators.	`z - 45 < 53` is evaluated as `(z - 45) < 53`.
< <= > >=	Then, relational operators **< <= > >=** are evaluated.	`x < 2 || x >= 10` is evaluated as `(x < 2) || (x >= 10)` because < and >= have precedence over ||.
== !=	Then, the equality and inequality operators **== !=** are evaluated.	`x == 0 && x >= 10` is evaluated as `(x == 0) && (x >= 10)` because < and >= have precedence over &&.
&	Then, the bitwise AND operator is evaluated.	`x == 5 | y == 10 & z != 10` is evaluated as `(x == 5) | ((y == 10) & (z != 10))` because & has precedence over |.
|	Then, the bitwise OR operator is evaluated.	`x == 5 | y == 10 && z != 10` is evaluated as `((x == 5) | (y == 10)) && (z != 10))` because | has precedence over &&.
&&	Then, the logical AND operator is evaluated.	`x == 5 || y == 10 && z != 10` is evaluated as `(x == 5) || ((y == 10) && (z != 10))` because && has precedence over ||.
||	Finally, the logical OR operator is evaluated.	

Examples

Evaluate the following Boolean expressions in Java (assume that x = 6 and y = 3):

- x > 6
- x > 6 || y < 5
- x == y
- !(x<10)
- x = 5
- 5 > 3 || 1 > 2 && 10 > 20
- (5 > 3 || 1 > 2) && 10 > 20

Short circuit evaluation

Table 3.13.1: Short circuit evaluation.

Operator	Example	Short circuit evaluation
`operand1 && operand2`	`true && operand2`	If the first operand evaluates to true, operand2 is evaluated.
	`false && operand2`	If the first operand evaluates to false, the result of the AND operation is always false, so operand2 is not evaluated.
`operand1 || operand2`	`true || operand2`	If the first operand evaluates to true, the result of the OR operation is always true, so operand2 is not evaluated.
	`false || operand2`	If the first operand evaluates to false, operand2 is evaluated.

- Short-circuit evaluation can be used to prevent run time errors
  - Consider this if-statement

```
if ((kids != 0) && (pieces / kids >= 2) ) {
    System.out.println("Each child may have two pieces!");
}
```

  - If the value of kids is zero, short-circuit evaluation prevents evaluation of (pieces / 0 >= 2)
    - If not, then this division by zero would cause a run-time error

Multi branch If-else syntax

```
if (expression_1) {
        // Statements to execute when expression_1 is true
} else if (expression_2) {
        // Statements to execute when expression_2 is true
        // and all previous expressions are false
}

........

else if (expression_n) {
        // Statements to execute when expression_n is true
        // and all previous expressions are false
} else {
        // Statements to execute when no previous expression
        // is true
}
```

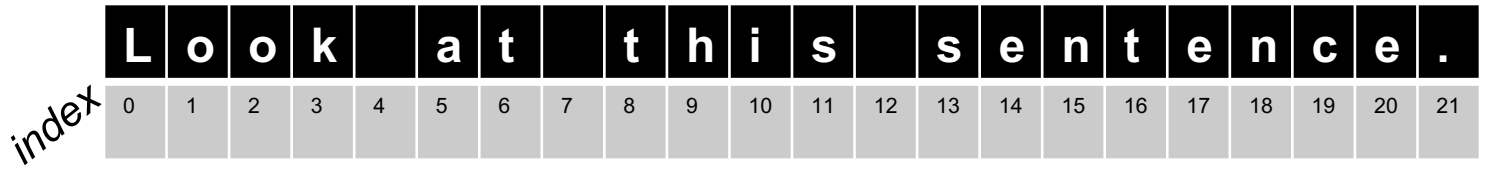
String comparisons

Comparison	Returns
str1.equals(str2)	*true* if str1 is *equal to* str2; otherwise, returns *false*
str1.compareTo(str2)	A *negative* number, if str1 < str2 (alphabetically) Zero, if str1 is equal to str2 A *positive* number, if str1 > str2 (alphabetically)

Example expression	Description
str1.equals(str2)	returns *true* if str1 is *equal* to str2
!str1.equals(str2)	retruns *true* if str1 is *not* equal to str2
str1.compareTo(str2) < 0	returns *true* if str1 is *less* than str2
str1.compareTo(str2) == 0	returns *true* if str1 is *equal* to str2
str1.compareTo(str2) > 0	returns *true* if str1 is *greater* than str2

Additional String methods

Definition: For a particular string, the index value *i* refers to the character at position *i +1*



Method	Description
str.indexOf(item)	returns the index of *item* in string *str*, or returns -1 if the item is not found
str.contains(item)	returns *true* if the string *str* contains the *item*; or *false* otherwise
str.replace(findStr,replaceStr)	Returns a string with all occurences of *findStr* replaced by *replaceStr* in the string *str*