Operations

Making Things Happen

(Chap. 3)

Expressions

Our Temperature Code

> We've looked at statements that declare objects (variables) and constants) and assignment statements. Now we look at the operations that can be applied to objects to produce expressions.

```
/* Temperature.java converts Celsius
   temperature                 Fahrenheit.
                            02

                       & Keyboard classes

  public static voi        String [] args)
  {
    Screen theScreen =        creen();
    theScreen.print("Wel       to the temperature converter!\n" +
          "Please enter     temperature in Celsius: ");

    Keyboard theKeyboard = n   Keyboard();
    double celsius = theKeyboard.readDouble();

    double fahrenheit = ((9.0/5.0)*celsius) + 32;

    theScreen.print(celsius + " degrees Celsius is " +
          fahrenheit + " degrees Fahrenheit.\n" +
          "It's been a pleasure!\n");
  }
}
```

Expressions

- As we noted in the last chapter, any sequence of *objects* and *operations* that combine to produce a value is called an *expression*.

- Example:
  ```
  double fahrenheit =
                ((9.0/5.0)*celsius) + 32;
  ```

- Now we focus on C++ *operations*

- But first, a little more about types.

Constructors

Sec. 3.2

- Primitive types use literals built into the compiler for their values.

- Reference types must use the _____operation:
  ```
  Screen theScreen = new Screen();
  ```
      type   identifier   **new** operator   call to initializing constructor

- Pattern:
  ```
  ClassName identifier =
             new ClassName(arguments);
  ```

- The **String** class is an exception; e.g.,
  ```
  String myName = "John Q. Doe";
  ```

Primitive vs Reference Types

All variables refer to memory locations:

– for primitive types, the locations store the value:

```
int age = 18;
```

```
age  18
```

– for reference types, the locations store an _____:

```
BigInteger bigNum = new BigInteger();
```

```
bigNum  0x2fca  ─────►  0x2fca   A
                                 BigInteger
                                 object
```

Wrapper Classes

- _____ *classes* are reference types that add capabilities to the primitive types:

  Byte Short Integer Long Float Double Boolean Character

- Examples:
  - Constants:
    ```
    Integer.MAX_VALUE
    Integer.MIN_VALUE
    ```
  - Methods:
    ```
    String digits = Integer.toString(intVal)
    ```
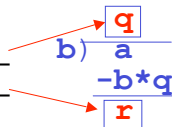

Numeric Expressions

Sec. 3.3

- Java provides four familiar arithmetic operators: **+, –, *, /**.

- They can be used with both reals and integers, but division (/) behaves differently:

  3/4 → _____   3.0/4.0 →_____

  3.0/4 → _____   3/4.0 → _____

- If a and b are integers:
  a / b returns the _____
  a % b returns the _____

  ```
  q
  b)  a
     -b*q
      r
  ```

  The "guzinta" (goes-into) operation

Implicit Type Conversion

- When types are mixed in an expression, the "narrower" type is "widened" to the larger type;. e.g.,

  ```
  (9.0/5.0)*celsius + 32;
  ```

  ```
  (9.0/5.0)*celsius + 32.0;
  ```

- These is known as_____.
  Legal promotions are:

  ```
  byte ⇒short ⇒ int ⇒ long⇒float ⇒ double
                 ⬆
                char
  ```

Explicit Type Conversion

- Using _____:

  ```
  double dubVar = _____intVar * dubValue;
  ```

- Using methods in wrapper classes:

  ```
  Integer intVarObject =  new Integer(intVar);
  double dubVar =
          intVarObject.doubleValue() * dubValue;
  ```

9

The **Math** Class

See Tables 3.2 & 3.3

- Contains _____, e.g.:

  ```
  PI = 3.14159 …
  E = 2.71828 …
  ```

- Contains _____, e.g.:

  ```
  abs(x)          sqrt(x)
  pow(x,y)        max(x,y)
  e(x)            log(x)
  ```

- To use these, attach **Math.** as a prefix;
  e.g.,   **Math.sqrt(x)**

10

Precedence/Priority

- **Question:** Is the value of the expression:
  ```
  2 + 3 * 4
  (2 + 3) * 4 → 20
  ```
  or **2 + (3 * 4) → 14**?

- Operator *precedence (or priority)* governs the evaluation order of operations in an expression. ***** has *higher precedence* than **+**, so it is applied first, making the answer _____.

- Parentheses can be used to override default precedence; e.g., **(2 + 3) * 4**

11

Operator Precedence

```
( )                              HIGHER
+ (positive), − (negative), ! (NOT)
*, /, %
+, −
<, <=, >, >=
==, !=
&&
||                               LOWER
```

See Appendix C for a complete list.

12

Associativity

- **Question:** Is the value of the expression **8 – 4 – 2**
  (8 – 4) – 2 → 2 or **8 – (4 – 2) → 6**?

- A*ssociativity* governs the order of execution of operators
  that have equal precedence.
  - **–** is _____-*associative*, so the left **–** is evaluated first

- Again, we can use parentheses to override the default;
  e.g., **8 – (4 – 2)**.

- Most (but not all) C++ operators associate left.

  See Appendix C for a complete list.

13

Assignment Expressions

Assignment is an _____; an expression
$$variable = expr$$
1. Assigns the value of *expr* to *variable*
   (side effect), and
2. Produces this value assigned to *variable*
   as the value of this expression

Appending a semicolon produces an
assignment statement.

14

Assignment Chaining

The assignment operator is _____-*associative*,
which supports expressions like
```
int w, x, y, z;
w = x = y = z = 0;
```
which is evaluated as
```
w = (x = (y = (z = 0)));
```
The rightmost = is applied first, assigning 0 to **z**
and producing 0; the next = thus assigns **y** the
value of z (0) and produces 0; then **x** is assigned
the value of **y** (0), and finally **w** is assigned the
value of **x** (0).

15

Assignment Shortcuts

- Some assignments are so common,
  ```
  var = var + x;  // add x to var
  var = var - y;  // sub y from var
  ```
  that Java provides shortcuts for them:
  ```
  _____;  // add x to var
  _____;  // sub y from var
  ```

16

4

- In general, most arithmetic expressions of the form:

  var = var Δ value;

  can be written in the "shortcut" form:

  var Δ= value;

- Examples:
  ```
  x *= 2.0;    // double x's value
  y /= 2.0;    // decrease y by half
  ```

17

Increment and Decrement

- Other common assignments include:
  ```
  var = var + 1;  // add 1 to var
  var = var - 1;  // sub 1 from var
  ```
- Java provides shortcuts for them too:

  Postfix form:
  ```
  _____;  // add 1 to var
  _____;  // sub 1 from var
  ```
  Prefix form:
  ```
  _____;
  _____;
  ```

  > No difference in prefix and postfix if used in these stand-alone forms!

18

- Difference between the forms:
  - The prefix form produces the final (incremented) value as its result.
  - The postfix form produces the original (unincremented) value as its result.
- Example:
  ```
  int x, y = 0;          Output
  x = ++y;               __
  theScreen.println(y);  __
  theScreen.println(x);  __
  y = 0;                 __
  x = y++;
  theScreen.println(y);
  theScreen.println(x);
  ```

19

Sec. 3.5

Boolean Expressions

- Java provides 6 operators for comparisons, each takes two operands and produces a **boolean** value (**true** or **false**):
  ```
  x == y            x != y

  x < y             x >= y

  x > y             x <= y
  ```
- An easy mistake to make is using **=** (assignment) in place of **==** (equality).

20

5

- More complex boolean expressions can be built using the logical operators:

```
a && b  // true iff a,b are both true
a || b  // true iff a or b is true
!a      // true iff a is false
```

- Examples:

```
(0 <= score) && (score <= 100)
done || (count > 1000)
```

- _____ *evaluation*: Second operand isn't evaluated unless necessary (e.g., if score is negative; if done is true.) This is useful in guarding potentially unsafe operation; e.g.,

```
(x >= 0) && (Math.sqrt(x) < 10)
```

21

Character Expressions

Sec. 3.6

- **char** objects can be used in comparisons:

```
'A' < 'B'
('a' <= letter) && (letter <= 'z')
```

- They are compared using their numeric (Unicode) codes:

```
'A' < 'B' // true because _____
```

- The **Character** wrapper class provides additional methods, including:

See Table 3.7

```
digit(ch, b)        isLetter(ch)
getNumericValue(ch) isUpperCase(ch)
```

22

String Expressions

- Concatenation:

```
"Jo " __ "Doe" ⇒ "Jo Doe"
```

- Strings are made up of individual characters:
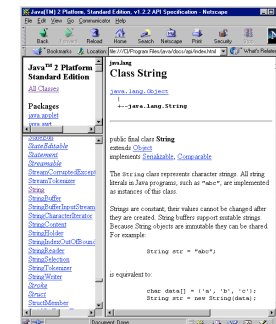
```
String name = "Jo Doe";
```

name

J	o		D	o	e	\0
0	1	2	3	4	5	6

– **name.charAt(3)** results in _____

23

Java's API Documentation

You'll never remember all the features of these reference types; e.g., **String**. *Use Java's online reference manual instead.*

http://java.sun.com/j2se/1.4.1/docs/api

24

6