Operations

Making Things Happen

(Chap. 3)

Expressions

Our Temperature Code

```
/* Temperature.java converts Celsius
 *                          heit.
                                  02
                                  & Keyboard classes
                                  c
{
  public static voi       tring [] args)
  {
    Screen theScreen =       creen();
    theScreen.print("We       to the temperature converter!\n" +
            "Please enter       temperature in Celsius: ");

    Keyboard theKeyboard = n    Keyboard();
    double celsius = theKeyboa rd.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");
  }
}
```

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.

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:

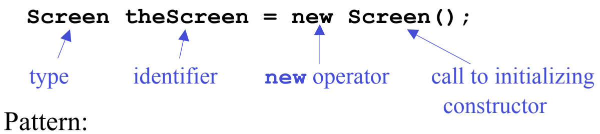
```
    double fahrenheit =
                  ((9.0/5.0)*celsius) + 32;
```

- 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 **new** 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

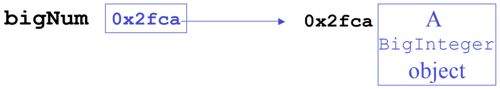
All variables refer to memory locations:

– for primitive types, the locations store the value:

```
int age = 18;
```

– for reference types, the locations store an *address:*

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



Wrapper Classes

- *Wrapper 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

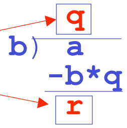
Sec. 3.3

- Java provides four familiar arithmetic operators: **+**, **-**, *****, **/**.
- They can be used with both reals and integers, but division (/) behaves differently:

  3/4 □ 0 3.0/4.0 □ 0.75

  3.0/4 □ 0.75 3/4.0 □ 0.75

- If `a` and `b` are integers:
  a / b returns the ***quotient***
  a % b returns the ***remainder***



The "guzinta" (goes-into) operation

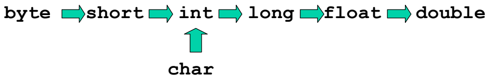
Implicit Type Conversion

- 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 promotion.
  Legal promotions are:



Explicit Type Conversion

- Using type casting:

```
double dubVar = (double)intVar * dubValue;
```

- Using methods in wrapper classes:

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

The **`Math`** Class

See Tables 3.2 & 3.3

- Contains *static constants,* e.g.:

```
PI = 3.14159 …
E = 2.71828 …
```

- Contains *static methods,* 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)`**

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 14.
- Parentheses can be used to override default precedence; e.g., **`(2 + 3) * 4`**

Operator Precedence

HIGHER

`( )`

`+` (positive), `-` (negative), `!` (NOT)

`*`, `/`, `%`

`+`, `-`

`<`, `<=`, `>`, `>=`

`==`, `!=`

`&&`

`||`

LOWER

See Appendix C for a complete list.

Associativity

- **Question:** Is the value of the expression `8 - 4 - 2` `(8 - 4) - 2 □ 2` or `8 - (4 - 2) □ 6`?
- *Associativity* governs the order of execution of operators that have equal precedence.
  `–` is *left-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.

Assignment Expressions

Sec. 3.4

Assignment is an operation; 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*.

Assignment Chaining

The assignment operator is *right-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).

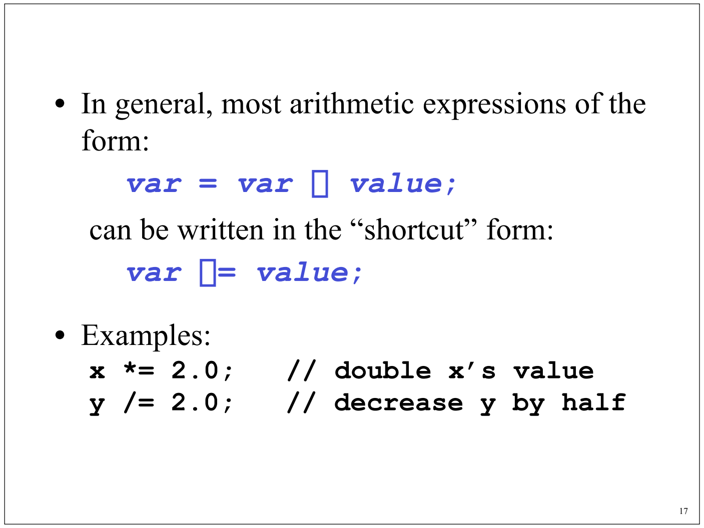
Assignment Shortcuts

- 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:

```
    var += x;  // add x to var
    var -= y;  // sub y from var
```



- In general, most arithmetic expressions of the form:

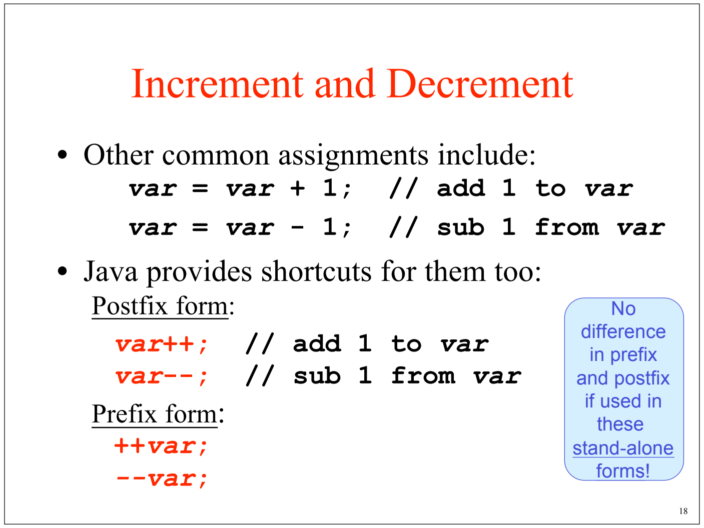
  `var = var □ value;`

  can be written in the “shortcut” form:

  `var □= value;`

- Examples:

```
x *= 2.0;    // double x’s value
y /= 2.0;    // decrease y by half
```



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:

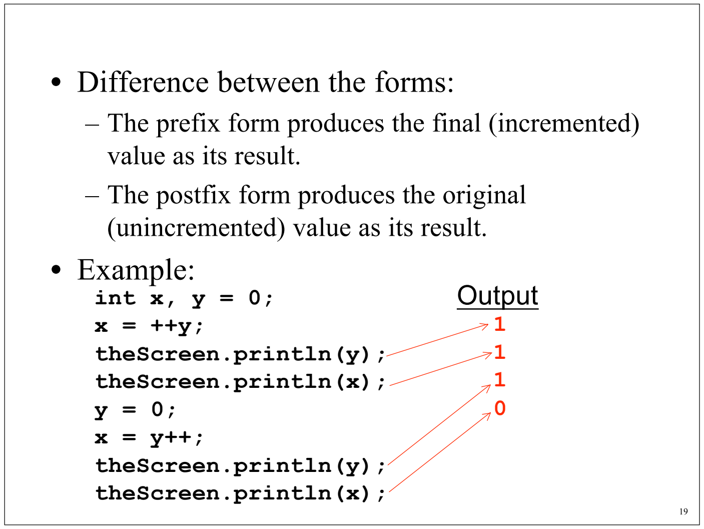
Postfix form:

```
var++;  // add 1 to var
var--;  // sub 1 from var
```

Prefix form:

```
++var;
--var;
```

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

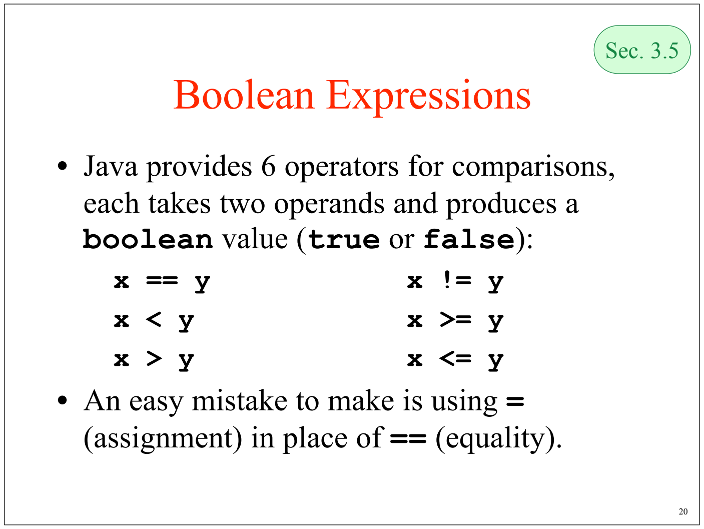


- Difference between the forms:
  - The prefix form produces the final (incremented) value as its result.
  - The postfix form produces the original (unincremented) value as its result.
- Example:

```
int x, y = 0;
x = ++y;
theScreen.println(y);
theScreen.println(x);
y = 0;
x = y++;
theScreen.println(y);
theScreen.println(x);
```

Output

```
1
1
1
0
```



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).

- 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)
```

- *Short-circuit 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)
```

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 65 < 66
```

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

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

See Table 3.7

String Expressions

- Concatenation:

```
"Jo " + "Doe" □ "Jo Doe”
```

- Strings are made up of individual characters:

```
String name = "Jo Doe";
```

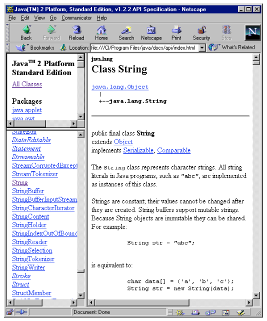
name

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

– `name.charAt(3)` results in `'D'`

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.*


Java(TM) 2 Platform, Standard Edition, v1.2.2 API Specification - Netscape
File Edit View Go Communicator Help
Back Forward Reload Home Search Netscape Print Security Stop
Bookmarks Location: file:///C|/Program Files/java/docs/api/index.html
What's Related
Document: Done

Java™ 2 Platform Standard Edition

All Classes

Packages

java.applet
java.awt

StateEdit
StateEditable
Statement
Streamable
StreamCorruptedExcept
StreamTokenizer
String
StringBuffer
StringBufferInputStream
StringCharacterIterator
StringContent
StringHolder
StringIndexOutOfBounc
StringReader
StringSelection
StringTokenizer
StringWriter
Stroke
Struct
StructMember

java.lang
Class String

```
java.lang.Object
  |
  +--java.lang.String
```

public final class **String**
extends Object
implements Serializable, Comparable

The `String` class represents character strings. All string literals in Java programs, such as `"abc"`, are implemented as instances of this class.

Strings are constant; their values cannot be changed after they are created. String buffers support mutable strings. Because String objects are immutable they can be shared. For example:

```
     String str = "abc";
```

is equivalent to:

```
     char data[] = {'a', 'b', 'c'};
     String str = new String(data);
```

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