Chapter 5
Comparison operators

• Operate on two numeric values; result in a boolean value.

• (Again, note operator precedences.)

• You know most of these from secondary school algebra.
## Basic comparisons

<table>
<thead>
<tr>
<th>Java</th>
<th>name</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>&lt;</code></td>
<td>less than</td>
</tr>
<tr>
<td><code>&gt;</code></td>
<td>greater than</td>
</tr>
<tr>
<td><code>==</code></td>
<td>equal to</td>
</tr>
<tr>
<td><code>!=</code></td>
<td>not equal to</td>
</tr>
<tr>
<td><code>&lt;=</code></td>
<td>less than or equal to</td>
</tr>
<tr>
<td><code>&gt;=</code></td>
<td>greater than or equal to</td>
</tr>
</tbody>
</table>
Example

```java
public static boolean areEqual(int x, int y) {
    return x == y;
}

(Of course, a real program would just use \texttt{x==y} to get the same result!)
```
Comparing objects

- Three ways to compare objects
  - `==`
  - `.equals()`
  - `.compareTo()`

- Each is useful for different things!
Referential equality

- A test for objects

- \( x == y \) is true iff \( x \) and \( y \) refer to the exact same object

- In this case, \( \text{otto} == \text{myDog} \) is true

- Why is this kind of test useful?
Warning!

- Two different objects will never have referential equality, even if they have the same contents!

- Why is this the case?

```java
public boolean refTest() {
    Dog otto =
        new Dog("Otto");
    Dog myDog =
        new Dog("Otto");
    // always false!
    return otto == myDog;
}
```
Comparing contents

- All objects support a method called `equals`
- `equals` takes an `Object` and returns a `boolean`: `true` if the base object is “equal to” the parameter, and `false` otherwise
- What does “equal to” mean?
Comparing contents

• What does “equal to” mean?

• Answer: whatever we want it to!

• By default, “equal to” means referential equality; we can provide an equals method that does better than that.
Example

public class NumHolder {
    private int num;

    public NumHolder(int n) {
        num = n;
    }

    // ... other methods

    public boolean equals(Object o) {
        if (o instanceof NumHolder) {
            return ((NumHolder)o).num == this.num;
        }
        return false;
    }
}

Self-check questions

• Say you have the following block of code:
  
  \[
  \begin{align*}
  x &= y; \\
  \text{return} & \ x == y;
  \end{align*}
  \]

  Assuming it compiles, will it always return true?

• How about if that second line reads
  
  \[
  \text{return} \ x.\text{equals}(y);
  \]

Comparing objects

- Many objects support a `compareTo` method.
- `x.compareTo(y)` returns:
  - the `int 0` if `x` and `y` have the same contents
  - some `int i`, `i < 0` if `x` is "less than" `y`
  - some `int i`, `i > 0` if `x` is "greater than" `y"
Logic basics

• We’re often interested in combining boolean values in interesting ways

• Logical connectives provide us with various means to do this
Connectives

- **AND:** \( x \text{ AND } y \) is true
  \( \text{iff } x \text{ is true and } y \text{ is true} \)

- **OR:** \( x \text{ OR } y \) is true
  \( \text{iff } x \text{ is true or if } y \text{ is true} \)

- **NOT:** \( \neg x \) is true
  \( \text{iff } x \text{ is false} \)

- **Note difference between logical OR and vernacular or!**
DeMorgan’s Law

“A negated conjunction is equivalent to a disjunction of negations, and a negated disjunction is equivalent to a conjunction of negations.”

(Say that five times quickly!)
DeMorgan’s Law

• NOT (x AND y) == (NOT x) OR (NOT y)

• NOT (x OR y) == (NOT x) AND (NOT y)
There are many other useful logical equivalences like this; how can we find them?
Truth tables provide a systematic, exhaustive way to explore the results of boolean functions.
| $x$ | $y$ | $x$ && $y$ | $x$ || $y$ | !$y$ |
|-----|-----|-----------|----------|------|
| F   | F   | F         | F        | T    |
| F   | T   | F         | T        | F    |
| T   | F   | F         | T        | T    |
| T   | T   | T         | T        | F    |
Self-check

• Write the following expressions in equivalent ways (p, q, and r are booleans, x and y are ints):
  • p = !(q && r) && !(!q && !r)
  • p = !(x < 4) && !(y > 5)
  • p = !((x < 2) && (y <= 4))
Binary boolean connectives don’t always evaluate both their arguments.
“I’m going to buy a cola if it costs less than ten cents and if I can climb Mt. Everest in under two weeks.”
If an and expression is *doomed to be false*, Java won’t evaluate the right hand side.

```
false && someExpensiveMethod()
```
Likewise, if an or expression is guaranteed to be true...

true || someExpensiveMethod()
This property is called *short-circuited evaluation*.
Boolean expressions can be used to make decisions in your programs.
Java programs can make decisions in a few ways: with an expression, or with one of several statements.
The *ternary operator* is an unusual 3-valued operator.

\[
\text{exp}_{\text{test}} \ ? \ \text{exp}_{\text{cons}} : \ \text{exp}_{\text{alt}}
\]

if \( \text{exp}_{\text{test}} \) is true

if \( \text{exp}_{\text{test}} \) is false
int x, y;

// ...

System.out.print("The larger number is: ");
System.out.println((x > y) ? x : y);
The ternary operator is also called the “conditional operator” and the “selection operator.”
The *if statement* executes a statement if its condition is true

\[
\text{if } (\text{exp}_{\text{test}}) \quad \text{stmt}_{\text{cons}}
\]

if \(\text{exp}_{\text{test}}\) is true
What is a statement?
A statement is either

- a single line of Java code terminated by a semicolon, or
- a sequence of statements enclosed in curly braces
x++;

System.out.println("foo");
{
    int y = 4;
    System.out.println(y);
}
{
    {  foo();  }
    System.out.println(5);
}

Copyright © 2005-2007 William C. Benton
The *if* statement executes a statement if its condition is true

\[
\text{if } (\exp_{test}) \quad stmt_{cons} \\
\text{if } \exp_{test} \text{ is true}
\]
if \((\text{exp}_{test})\)

\textit{stmt}_{cons}
The *if statement* executes a statement if its condition is true.

```plaintext
if (exp_{test})
  stmt_{cons}
else
  stmt_{alt}
```

- If \( exp_{test} \) is true
- If \( exp_{test} \) is false
if \( (exp_{test}) \)

\emph{stmt}_{cons}

else

\emph{stmt}_{alt}
The *if statement* executes a block of code if its condition is true.

\[
\text{if } (\exp_{test1}) \quad \text{stmt}_{cons1} \\
\text{else if } (\exp_{test2}) \quad \text{stmt}_{cons2} \\
\text{else if } (\exp_{test3}) \quad \text{stmt}_{cons3} \\
\text{else } \quad \text{stmt}_{alt}
\]
if \((\text{exp}_{t1})\) 
\(\text{stmt}_{c1}\) 
else if \((\text{exp}_{t2})\) 
\(\text{stmt}_{c2}\) 
else 
\(\text{stmt}_{alt}\)
switch is useful when you need to execute one (or more) of several statements depending on the value of an expression

```
switch (exp) {
    case v_1: stmt_1
    case v_2: stmt_2
    case v_3: stmt_3
    default: stmt_d
}
```

If \( exp \) has value \( v_1 \), then the `switch` statement will start executing at `stmt_1` (and so on). **Where does execution stop?**
Think of *switch* statements as *jumping into the middle of a block of code*.

(You can escape with *break*.)
Translate the following code into equivalent code that uses an if statement:

```java
switch (x % 2) {
    case 0:  System.out.println("E");
    case 1:  System.out.println("O");
    default:  System.out.println("D");
}
```
Review: switch

• Are there any restrictions on the type of the expression?

• Choices in `switch` statements aren’t mutually exclusive; why not?

• What is the term for the behavior that enables `switch` statements to execute more than one statement?