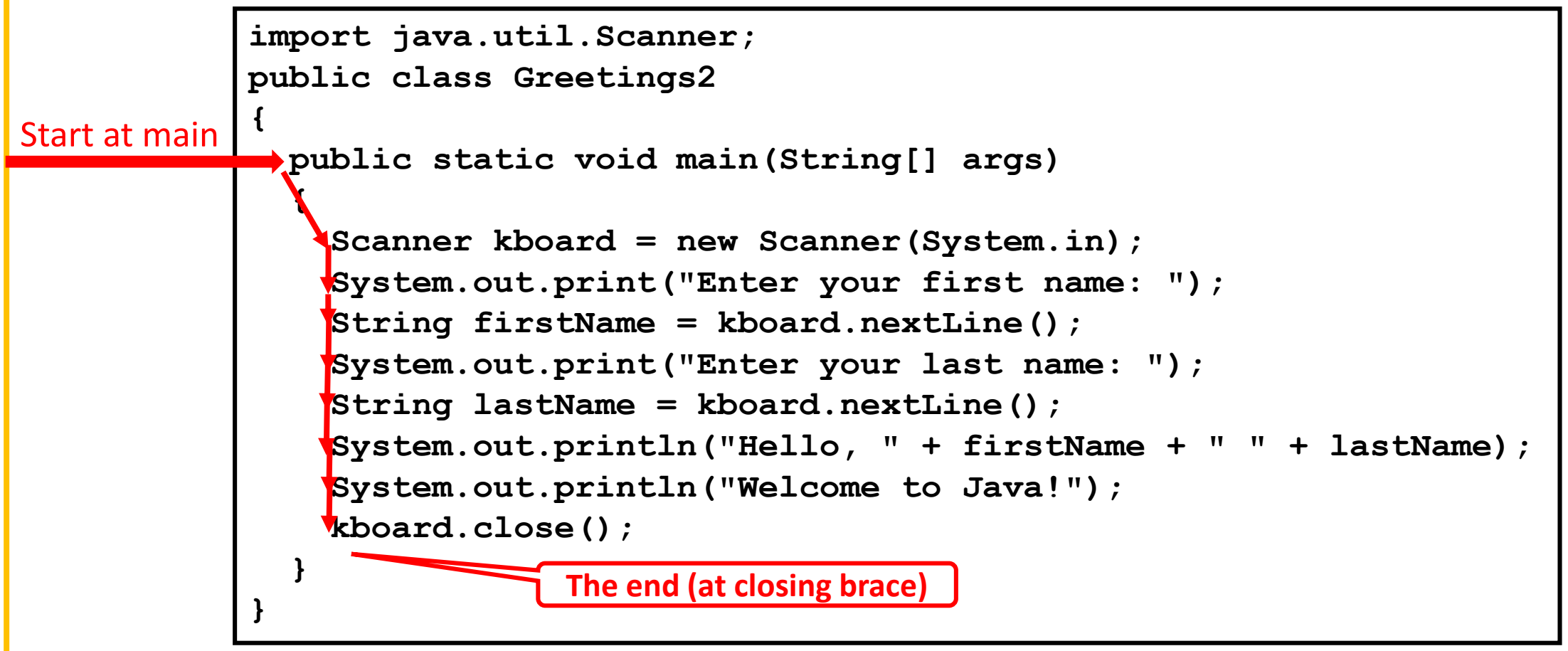
Chapter 5

JC08-5

(A. Nguyen)

Flow of control

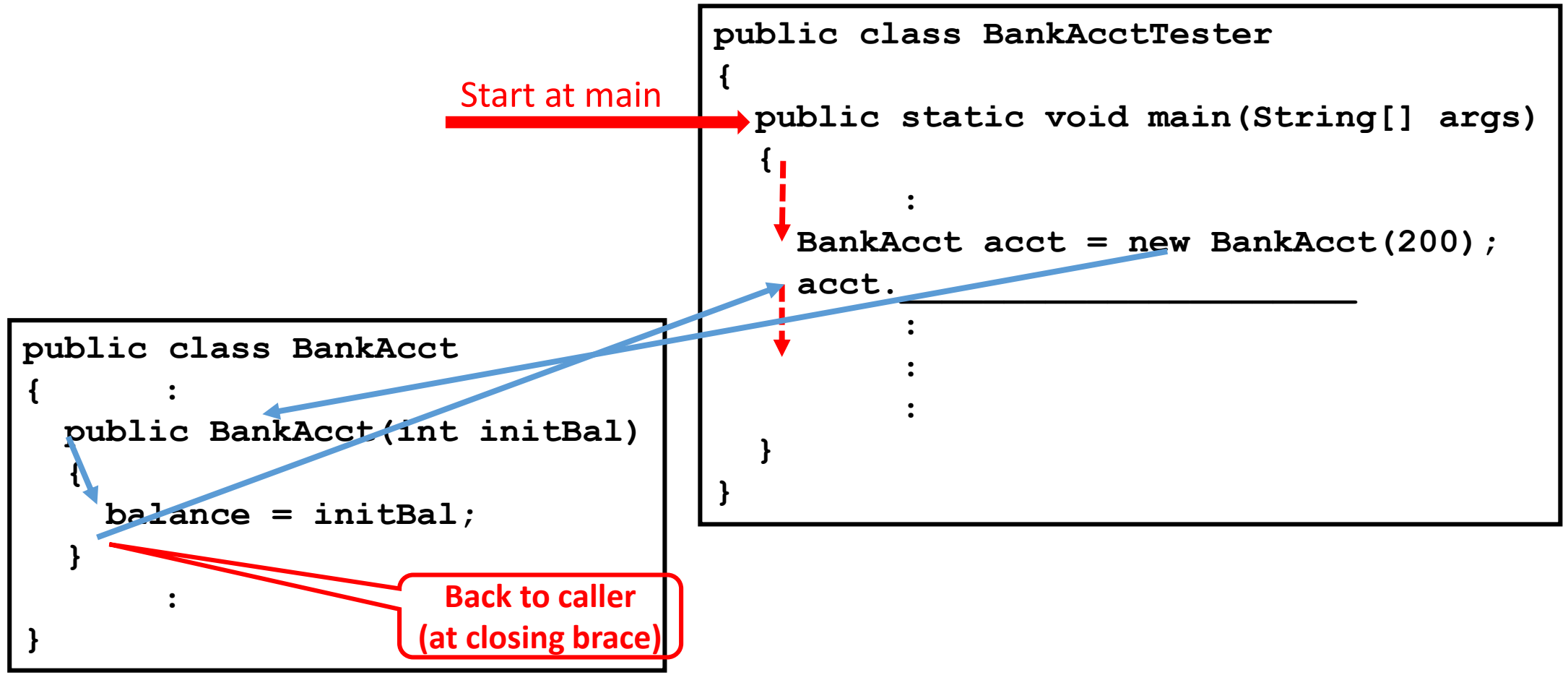
Control flow: normal flow (w/ method call in each statement)



Control flow

- Control flow or flow of control is the sequence of instructions that get executed when a program is run
- Normally, the flow is from one instruction to the instruction below it, except for:
  - A call to a method
  - Conditional statements (`if-else`)
  - Branching statements (`switch`)
  - Iterative statements (i.e., loop)
  - Exceptions
- In the debugger, the "step" function can help identify the control flow

Control flow: method call



Control flow: `return` statement

`return` statement does 2 things:

- Go back to the caller
- Give the caller the output

Start at main

```
public class someTester
{
   public static void main(String[] args)
   {
         :
      double area = calcCircleArea(5.0);
         :
         :
         :
   }
}
```

```
public double calcCircleArea (double r)
{
   return Math.PI() * r * r;
}
```

Back to caller (at `return`)

NOTE: Because control goes back to the caller at the `return` statement, there should NOT be another statement after it.

Control flow: `if-else` statement (& method calls)

Start at main

```
public class someTester
{
   public static void main(String[] args)
   {
         :
      int fact1 = calcFactorial(1);
         :
         :
      int fact5 = calcFactorial(5);
         :
         :
         :
   }
}
```

```
public int calcFactorial(int n)
{
   if (n <= 1)
      return 1;
   else
      return CalcFactorial(n-1);
}
```

NOTE: There is 1 `return` statement for each case. There should NOT be other statements after each or both.

if, if-else

if & if-else: flowchart

Flowchart with one branches

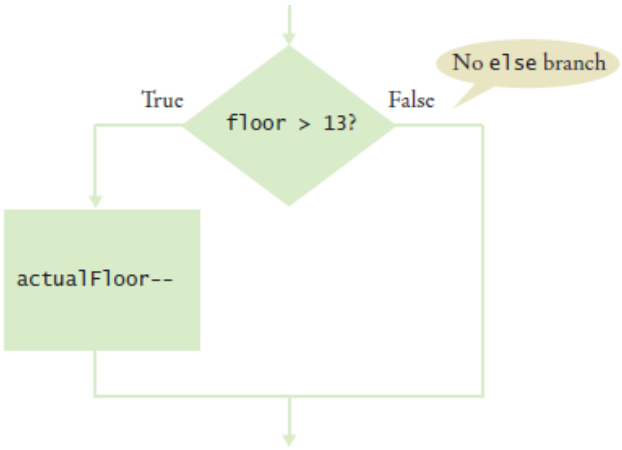
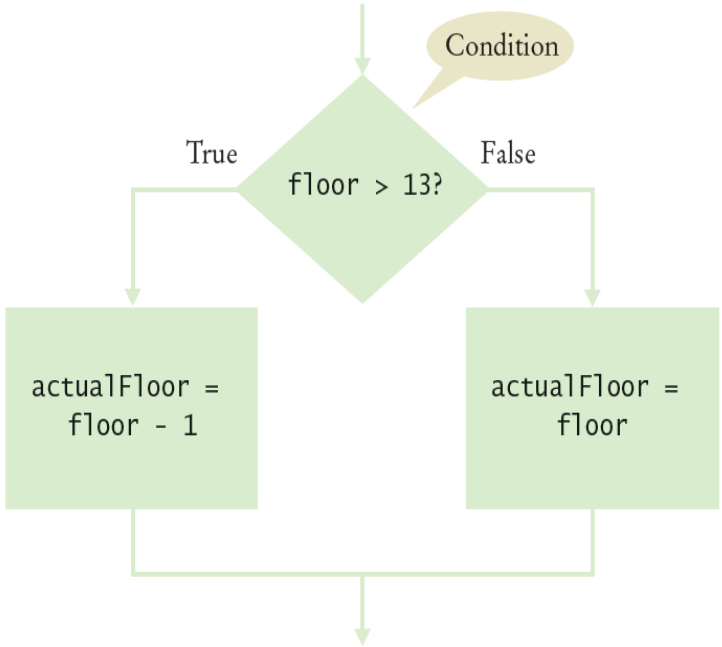


Figure 2
Flowchart for if Statement with No else Branch

Flowchart with two branch



You can include as many statements in each branch as you like.

if & if-else: syntax

```
if ( <condition> )
{
    < statements >
}
```

else clause is optional

Use this if there is 1 case to do.

```
if ( <condition> )
{
    < statements >
}
else
{
    < other statements >
}
```

Use this if there are 2 cases to do.

- If there is only 1 statement for a case, the braces are not required
- **`else` is matched w/ closest `if` above it**

Avoid duplicate code (in branch)

- Don't do this:

```
if (floor > 13) { // higher than floor 13
  actualFloor = floor - 1;
  System.out.println("Actual floor: " + actualFloor);
}
else { // at or lower than floor 13
  actualFloor = floor;
  System.out.println("Actual floor: " + actualFloor);
}
```

- Do this:

```
if (floor > 13) {// higher than floor 13
  actualFloor = floor - 1;
}
else {// at or lower than floor 13
  actualFloor = floor;
}
System.out.println("Actual floor: " + actualFloor);
```

Relational operators

Relational Operators

- A relational operator is used to check ***one condition***
- The **6 operators** are `<`, `>`, `<=`, `>=`, `==`, `!=` (to compare values of ***same/similar data types***; e.g., cannot compare numeric to string)
- `==` means "equal to"; and `!=` means "not equal to"
- Example: `int a, b;` assigned to some values

MEANING	CORRECT SYNTAX	ERROR
Is equal to	**if (a == b) {…}**	**if (a = b) {…}**
Is less than or equal to	**if (a <= b) {…}**	**if (a =< b) {…}**

Comparing double values

- Do NOT use `==` or `!=` with `doubles` because they may have rounding errors; i.e.,
  - 0 may be 0.1×10^{-21}
  - Verifying a right triangle (w/ converse of Pythagorean Thm) may "never" be true
- Example: given `double x, y;` assigned to some values
- Don't do this, which is almost always false:

```
if (x == y) {…}
```

- Do this:

```
final double EPSILON = 1E-14; // a very small #
if (Math.abs(x - y) <= EPSILON) {…}
```

Comparing strings or other objects

- Assume: given `String s1, s2;` assigned to some values
- Do this to compares the ***contents*** of 2 strings:

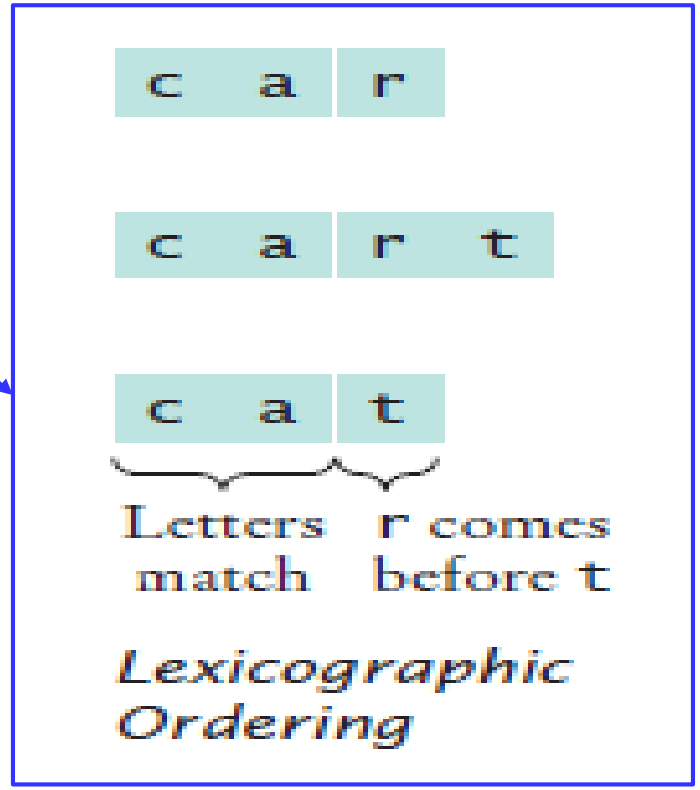
```
if (s1.equals(s2)) {…}
```

- Do this to check lexicographic ordering:

```
if (s1.compareTo(s2)) {…}
```

OR

```
if (s1.compareToIgnoreCase(s2)) {…}
```



Comparing strings or other objects (cont.)

- Do this to see whether the 2 strings/objects are the same – Do NOT do this to compare contents:

```
if (s1 == s2) {…}
```

- Do this to see whether the object exists (i.e., was created with new & constructor call):

```
if (s1 != null)) {…}
```

- Example in reading lines of text from a file, and detecting the end:

```
String text = file.readLine( );
if ( text != null )    {...}
```

Boolean Expressions

- In `if (`<*condition*>`)`, <*condition*> is a Boolean expression. A boolean expression evaluates to either `true` or `false`, and may be assigned/saved to a variable of type `boolean`
- Example: given `int length, width;` assigned to some values:

```
boolean isSquare = (length == width);
```

- Boolean expressions are written with `boolean` variables and relational & logical operators.

Avoid beginners' code

- Assume: given `int length, width;` assigned to some values
- Don't do this:

```
boolean isSquare;
        :
if (length == width) {
  isSquare = true;
}
else {
  isSquare = false;
}
```

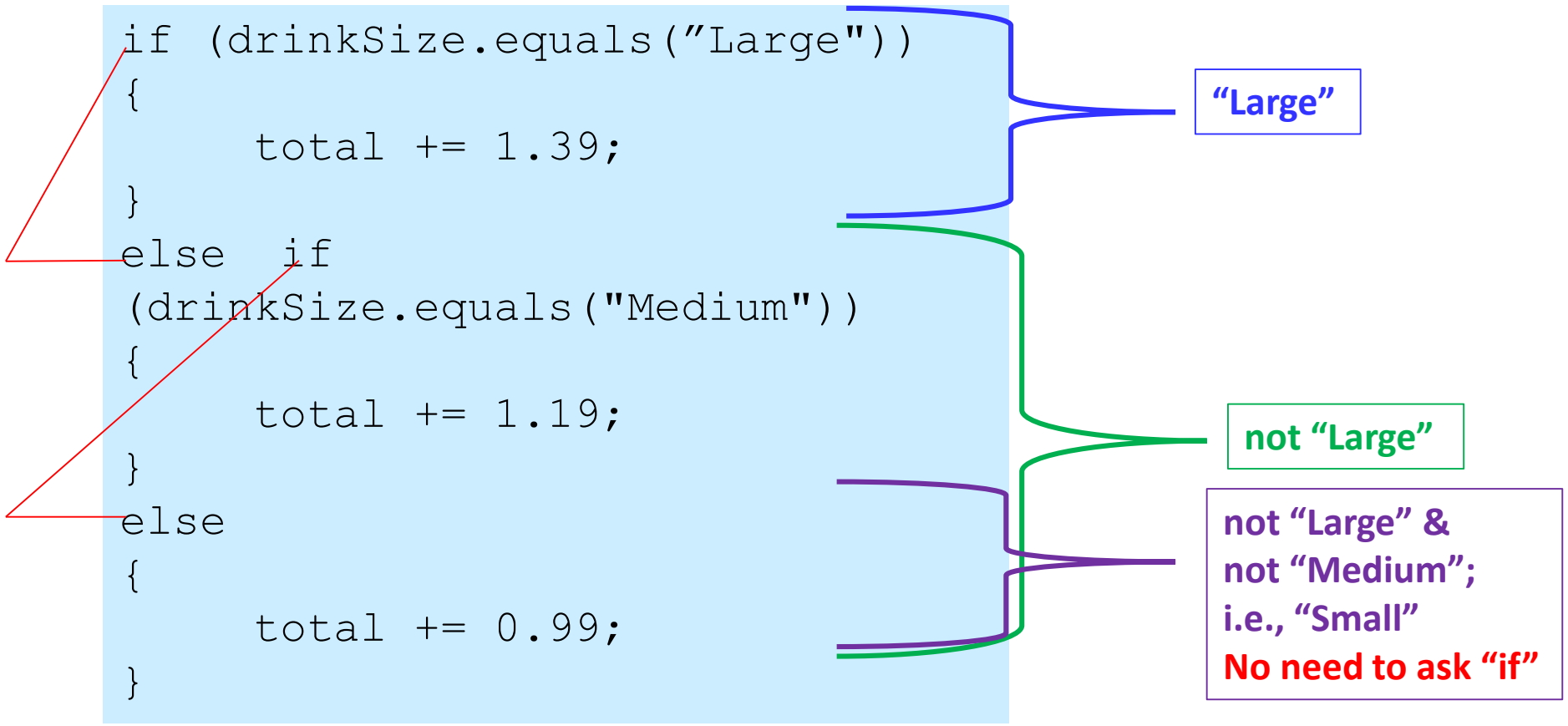
- Do this:

```
boolean isSquare;
        :
isSquare = (length == width);
```

Multiple-way “if”

if-else-if (for more than 2 cases)

```
if (drinkSize.equals(”Large"))
{
      total += 1.39;
}
else  if
(drinkSize.equals("Medium"))
{
      total += 1.19;
}
else
{
      total += 0.99;
}
```



Nested if-else

```
if ("forward".equals(cmd))
{
     if (slide >= numSlides)
          beep.play();
     else
          slide++;
}
else
{
     if (slide <= 1)
          beep.play();
     else
          slide--;
}
```

Common if-else Errors

Extra semicolon:

```
if  (...) ;
{
    statements;
   ...
}
```

Missing braces, so only statement1 is executed if condition is true:

```
if  (...)
    statement1;
    statement2;
...
```

else is matched with the most recent **if** above it:

```
if  (...)
   if  (...)
       statement1;
else
   statement2;
...
```

It is safer to always use braces in **if-else**.

Indentation is for humans, not compiler.

switch

The switch Statement

switch
case
default
break

Reserved words

```
switch  (expression)
{
   case value1:
      ...
      break;

   case value2:
      ...
      break;
   ...
   ...
   default:
      ...
      break;
}
```

Don't forget **break**s!

The switch Statement (cont.)

- The same case can have two or more labels. For example:

```
switch  (num)
{
    case 1:  // if (num == 1)
    case 2:  // if (num == 2)
       System.out.println ("Buckle your shoe");
       break;
    case 3:  // if (num == 3)
       ...
}
```

enum

[7.11]

enum Data Types

- Used when an object's attribute or state can have only one of a small set of values,

  for example:

```
private enum Speed { LOW, MEDIUM, HIGH };

private enum InkColor { BLACK, RED };

private enum DayOfWeek { sunday, monday,
      tuesday, wednesday, thursday, friday, saturday };
```

- enum variables **do not** represent numbers, characters, or strings.

enum Data Types (cont.)

- Use == or != to compare enum values

```
private enum Speed { LOW, MEDIUM, HIGH };
...
   Speed currentSpeed = Speed.LOW; // type-dot here
   ...
   if (currentSpeed == Speed.LOW) ...
```

- Can be used in a switch:

```
switch (currentSpeed)
{
   case LOW: // type-dot not used in switch statement
      ...
      break;
   case MEDIUM:
      ...
```

Logical operators

3 logical operators

- Logical operator “and” or “or” is used when ***2 conditions*** must be tested (binary operators, ***requiring 2 operands***)
- Logical operator “not” is used to check the opposite (unary operator, ***requiring 1 operand***)
- && means “and”; || means “or”; ! means “not”
- Example: `int a, b;` assigned to some values

MEANING	CORRECT SYNTAX	ERROR
and: result is true if both conditions are true	**if (a == 0 && b == 0) {…}**	**if (a && b == 0) {…}**
or: result is true if one or both condition(s) is/are true	**if (a == 0 || b == 0) {…}**	**if (a || b == 0) {…}**
not: result is the opposite: true → false, & vice-versa		

De Morgan's Laws

- &&, ||, and ! obey the laws of formal logic called *De Morgan's Laws*:

`!(p && q)`	**is same as**	`( !p || !q )`
`!(p || q)`	**is same as**	`( !p && !q )`

- Example:

```
if ( ! ( x => -10 && x <= 10 ) ) …
```

is same as

```
if ( x < -10 || x > 10 ) ...
```

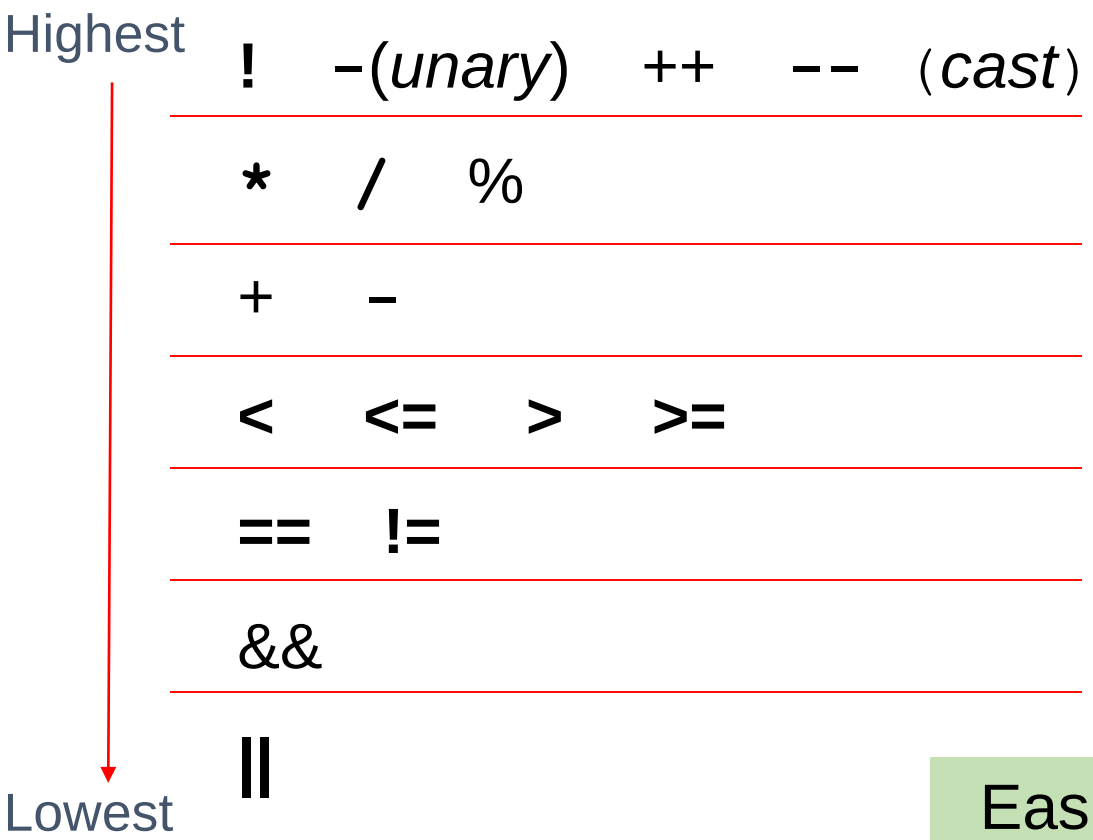
- Example:
- `!(asleep && inBed)` is same as `(!asleep || !inBed)`

Summary of relational & logical operators

Relational & logical operators

- Relational operators: >, <, >=, <=, ==, !=
  - For comparing values of the same type, so the operands are 2 values of the same type
  - >, <, >=, <= are not applicable for `boolean` type
  - Not used for `String` class
  - Takes 2 operands of same type; e.g., `if (i > k)` … (where `i` & `k` are `int`)
- Logical operators: &&, ||, !
  - The operands are 2 `boolean` values or expressions
  - Takes 2 boolean operands; e.g., `if (i < k || !isFound)` … (where `i` & `k` are `int`; and `isFound` is `boolean`)

Ranks of Operators



Easier to read

```
if  ( ( ( year % 4 ) == 0 ) && ( month == 2 ) ) ...
if  ( year % 4 == 0 && month == 2 ) ...
```

Short-Circuit Evaluation

if (*condition1* **&&** *condition2*) ...

If *condition1* is false, then *condition2* is not evaluated (the result is false anyway)

---

if (*condition1* **||** *condition2*) ...

If *condition1* is true, then *condition2* is not evaluated (the result is true anyway)

```
if ( x >= 0  &&  Math.sqrt (x) < 15.0) ...
```

Always OK: won't get to sqrt if x < 0

Short-circuit evaluation

- For OR, as soon as an operand is TRUE, we can stop evaluating and declare the result TRUE. The OR condition in real life: “must meet one of the requirements”. **Have you done short-circuit OR evaluation in real life by stop reading/caring about other requirements after knowing that 1 is met?**
- For AND, as soon as an operand is FALSE, we can stop evaluating and declare the result FALSE. The AND condition in real life: “must meet all of the requirements”. **Have you done short-circuit AND evaluation in real life by deciding not to pursue something when knowing that 1 requirement is not met?**

THE END