Advanced Programming Language

MSc. Nguyen Cao Dat

dat@hcmut.edu.vn

Module I (cont’d)

BASIC JAVA PROGRAMMING

Content

- Components of the Java Environment
- Your First Java Program
- Variables and Primitive Data Types
- Selection Statements
- Loop Statements
- Methods

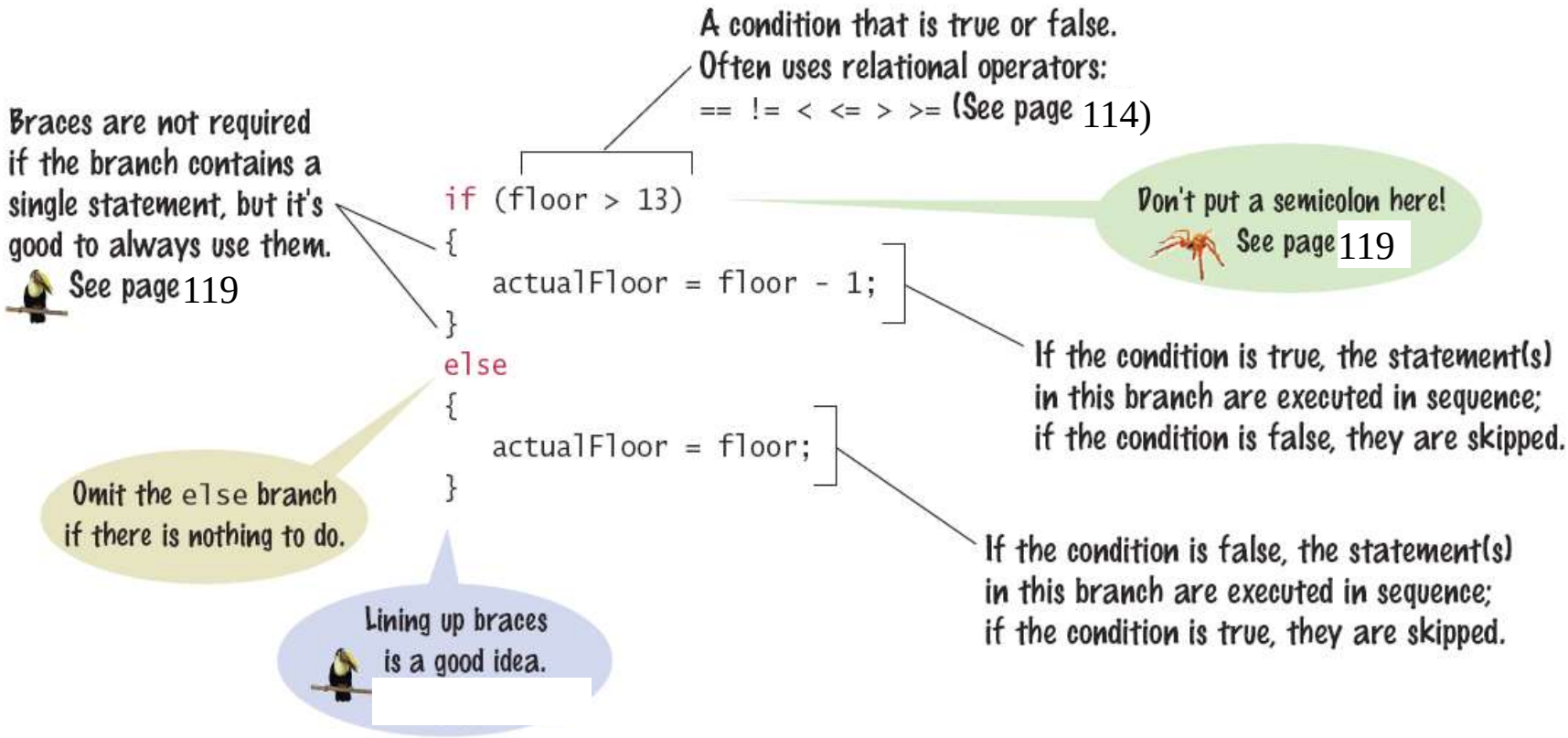
CREATING IF STATEMENTS

Recap: Simple Selection With IF Statements

- **Selection statements are used when a decision needs to be made based on a condition**
- **Selection statements can be created using IF statements**
- **The evaluation of the condition returns a boolean (true or false)**
- **Pseudocode Example**

```
IF account_balance < 300 THEN
   service_charge = 5.00
ELSE
   service_charge = 2.00
ENDIF
```

Syntax: The IF Statement



Example: IF Statement

```
1  import javax.swing.JOptionPane;
2
3  /*
4     This program simulates an elevator panel that skips the 13th floor
5  */
6
7  public class ElevatorSimulator {
8     public static void main(String[] args) {
9        int floor = Integer.parseInt(JOptionPane.showInputDialog("Enter your floor:"));
10
11       // Adjust the floor if necessary
12
13       int actualFloor;
14
15       if (floor > 13) {
16          actualFloor = floor - 1;
17       }
18       else {
19          actualFloor = floor;
20       }
21
22       JOptionPane.showMessageDialog(
23          null, "The elevator will travel to the actual floor: " + actualFloor);
24    }
25 }
```

Program Run

```
Floor: 20
The elevator will travel to the actual floor 19
```

Ways to Denote Braces

- **Line up all pairs of braces**
  - Most IDEs have the ability to automatically align matching braces

```
if (floor > 13)
{
   floor--;
}
```

```
if (floor > 13) {
   floor--;
}
```

- **Always use braces**
  - Although single statement clauses do not require them, you should always include them because without braces the clause consists of only the first statement

```
if (floor > 13)
   floor--;
```

```
if (floor > 13)
{
   floor--;
}
```

Tips on Indenting Blocks

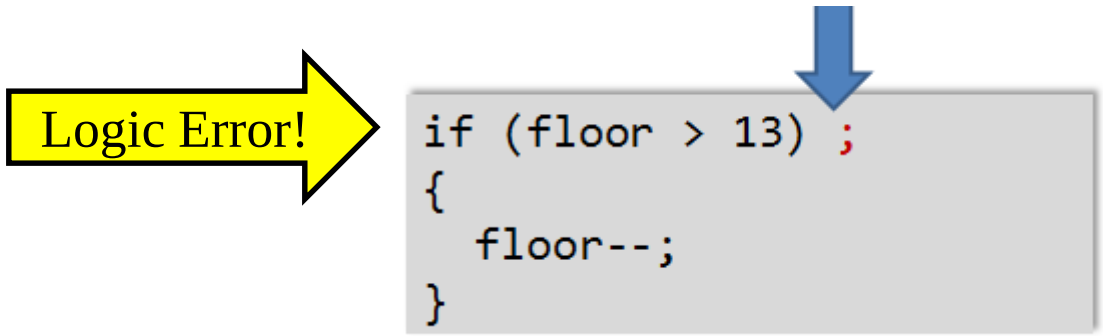
- **Use tab to indent a consistent number of spaces**

```
public class ElevatorSimulation
{
|  public static void main(String[] args)
|  {
|  |  int floor;
|  |  . . .
|  |  if (floor > 13)
|  |  {
|  |  |  floor--;
|  |  }
|  |  . . .
|  }
|  |  |  |
0  1  2  3    Indentation level
```

This is referred to as 'block- structured' code. Indenting consistently makes code much easier for humans to follow.

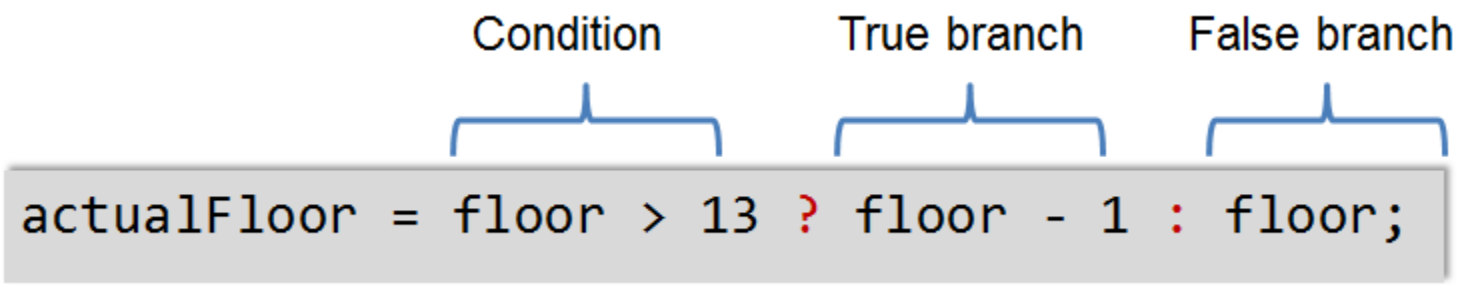
Common Error: Misplaced Semicolon

- **It is easy to make a mistake and include a semicolon at the beginning of an if statement**
  - The TRUE condition would now be the space just before the semicolon, which is empty. Nothing would execute and orphan braces are a logic error



The Conditional (Ternary) Operator

- **Shortcut used to denote an IF control structure**
- **Includes all parts of an if/else clause, but uses**
  - ? To begin the true branch
  - : To end the true branch and start the false branch



COMPARING NUMBERS AND STRINGS

Comparing Numbers

- **Every IF statement has a condition**
  - Usually compares two values with an operator

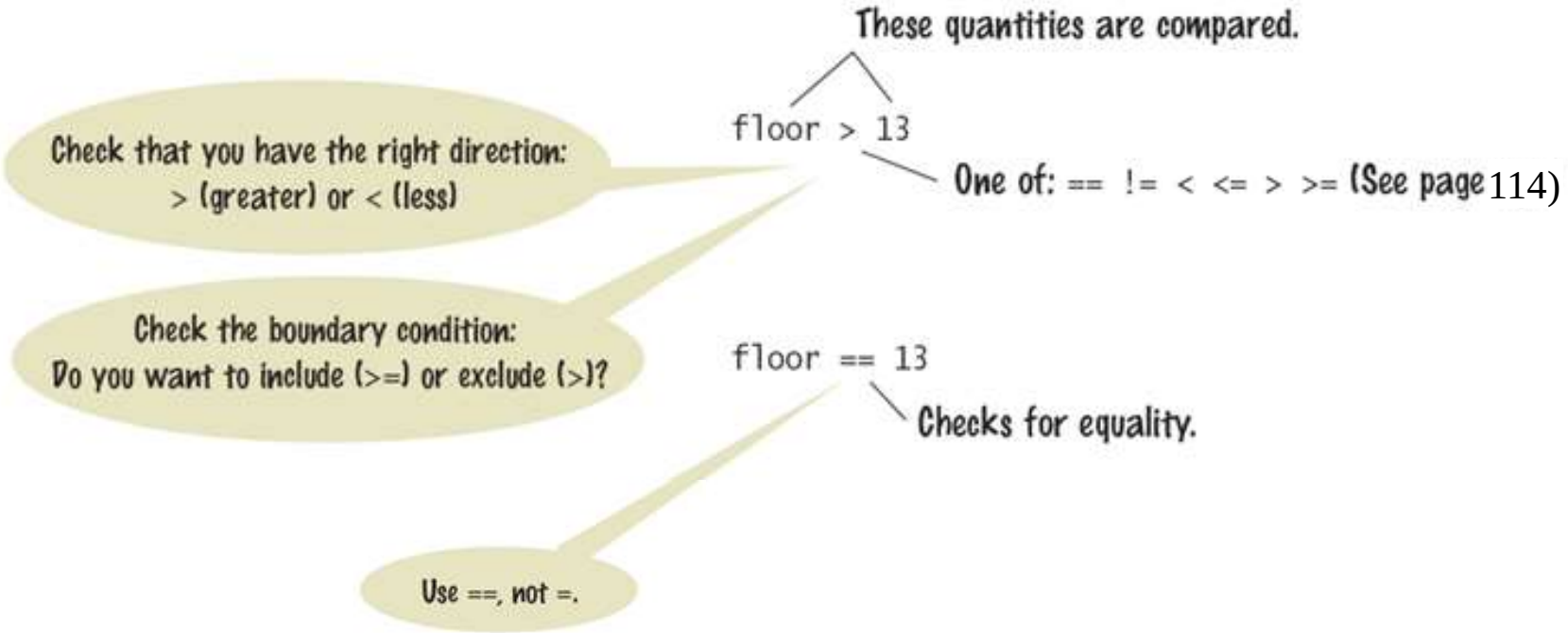
```
if (floor > 13)
   ..
if (floor >= 13)
   ..
if (floor < 13)
   ..
if (floor <= 13)
   ..
if (floor == 13)
   ..
```

Beware!

Table 1 Relational Operators

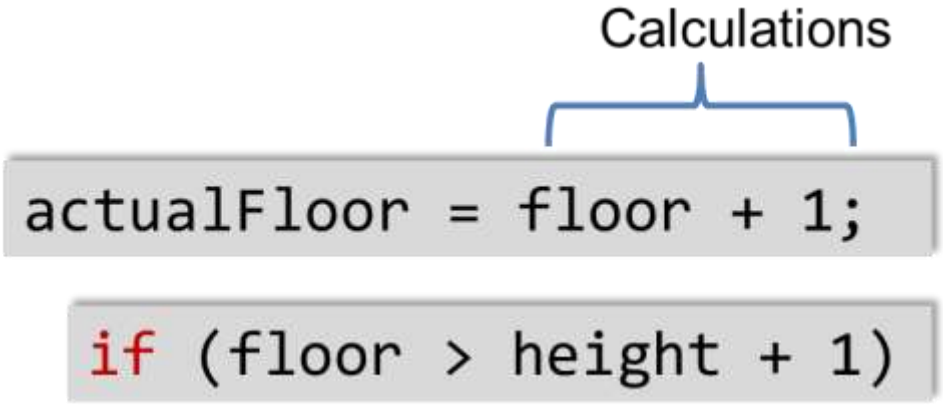
Java	Math Notation	Description
>	>	Greater than
>=	≥	Greater than or equal
<	<	Less than
<=	≤	Less than or equal
==	=	Equal
!=	≠	Not equal

Syntax: Numerical Comparisons



Operator Precedence

- **Comparison operators, such as: < and > have a lower precedence than arithmetic operators**
  - Calculations are done before the comparison
  - Normally, calculations are on the "right side" of the comparison or assignment operator (=)



Comparing Strings

- **Strings are a bit "special" in Java**
- **You cannot use == to compare Strings**
  - Using == compares the memory locations, instead
- **Use the String's equals or equalsIgnoreCase method**

```
if (string1 == string2) ...
```

```
if (string1.equals(string2)) ...
```

Relational Operator Use Examples

Table 2 Relational Operator Examples

Expression	Value	Commment
`3 <= 4`	`true`	3 is less than 4; <= tests for "less than or equal".
`3 =< 4`	**Error**	The "less than or equal" operator is <=, not =<. The "less than" symbol comes first.
`3 > 4`	`false`	> is the opposite of <=.
`4 < 4`	`false`	The left-hand side must be strictly smaller than the right-hand side.
`4 <= 4`	`true`	Both sides are equal; <= tests for "less than or equal".
`3 == 5 - 2`	`true`	== tests for equality.
`3 != 5 - 1`	`true`	!= tests for inequality. It is true that 3 is not 5 – 1.

Relational Operator Use Examples (Cont'd)

`3 = 6 / 2`	**Error**	Use == to test for equality.
`1.0 / 3.0 == 0.333333333`	`false`	Although the values are very close to one another, they are not exactly equal because the expression results in a repeating decimal not exactly equivalent to 0.333333333
`"10" > 5`	**Error**	You cannot compare a string to a number.
`"Tomato".substring(0, 3).equals("Tom")`	`true`	Always use the `equals` method to check whether two strings have the same contents.
`"Tomato".substring(0, 3) == ("Tom")`	`false`	Never use == to compare strings; it only checks whether the strings are stored in the same location. See "Comparing `String` Objects" on page 145.

SOLVING PROBLEMS USING IF STATEMENTS

Implementing an IF Statement

1) **Decide on a branching condition**

original price < 128?

2) **Write pseudocode for the true branch**

discounted price = 0.92 x original price

3) **Write pseudocode for the false branch**

discounted price = 0.84 x original price

4) **Double-check relational operators**
   - Test values below, at, and above the comparison (127, 128, 129)

5) **Remove duplication**

6) **Test both branches**

7) **Write the code in Java**

Implemented Example

- **The university bookstore has a Kilobyte Day sale every October 24, giving an 8 percent discount on all computer accessory purchases if the price is less than $128, and a 16 percent discount if the price is at least $128.**

```
if (originalPrice < 128) {
  discountRate = 0.92;
}
else {
  discountRate = 0.84;
}
discountedPrice = discountRate * originalPrice;
```

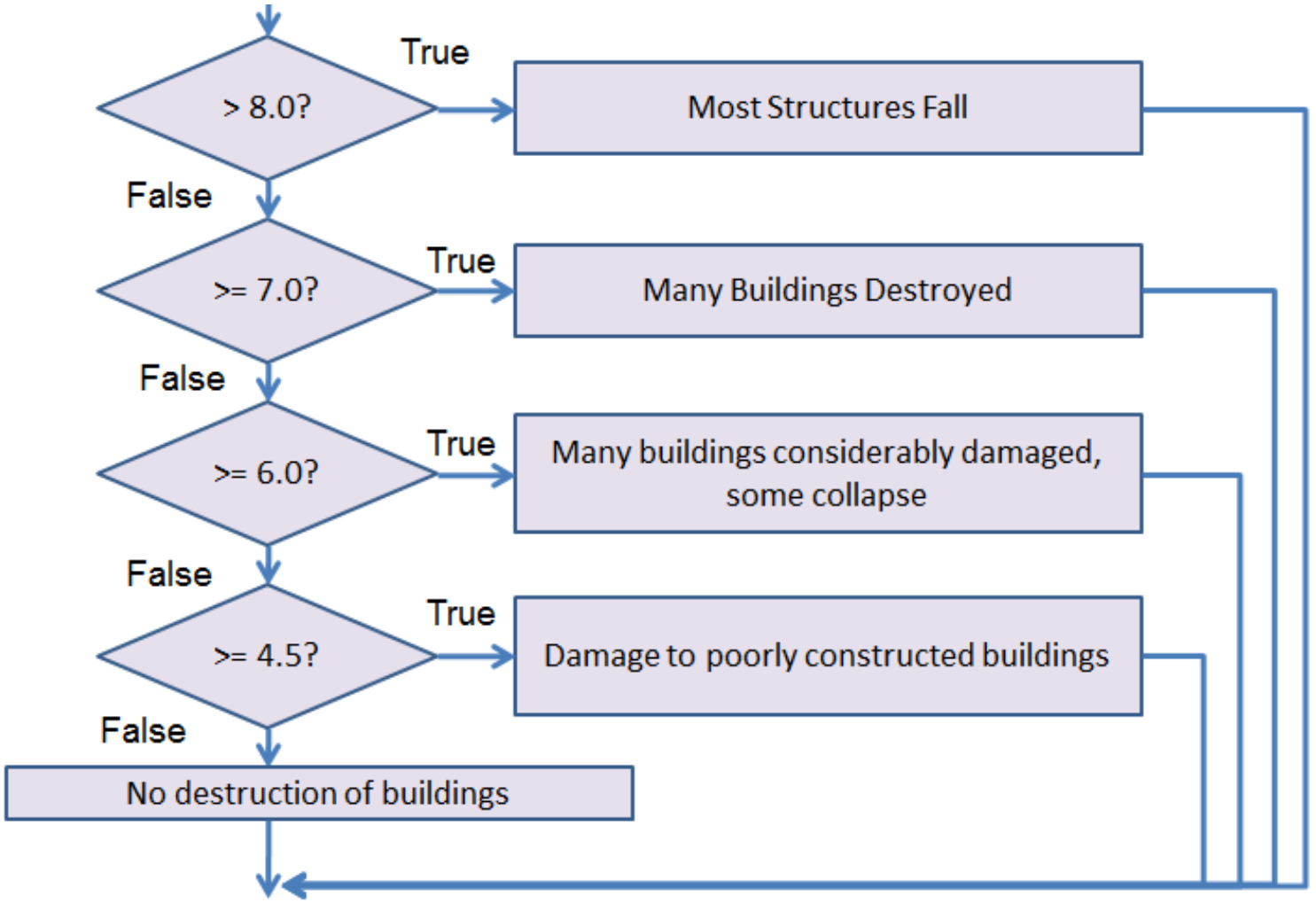
Multiway Branching

- **Used when there are more than two available decision branches, such as for the Richter scale:**
  - 8 (or greater)
  - 7 to 7.99
  - 6 to 6.99
  - 4.5 to 5.99
  - Less than 4.5

Table 3 Richter Scale

Value	Effect
8	Most structures fall
7	Many buildings destroyed
6	Many buildings considerably damaged, some collapse
4.5	Damage to poorly constructed buildings

Flowchart of Multiway Branching



If/Else If Multiway Branching

- **When using multiway branching, only one branch will be selected**

```
 7 public class Richter {
 8    public static void main(String[] args) {
 9       double scaleValue = Double.parseDouble(JOptionPane.showInputDialog("Enter the Richter value:"));
10       String message;
11
12       if (scaleValue >= 8.0) { // Handle this 'special case' first
13          message = "Most structures Fall";
14       }
15       else if (scaleValue >= 7.0) {
16          message = "Many buildings destroyed";
17       }
18       else if (scaleValue >= 6.0) {
19          message = "Many buildings damaged, some collapse";
20       }
21       else if (scaleValue >= 4.5) {
22          message = "Damage to poorly constructed buildings";
23       }
24       else { // so that the 'general case' can be handled last
25          message = "No destruction of buildings";
26       }
27
28       JOptionPane.showMessageDialog(null, message);
29    }
30 }
```

Question

- **What is wrong with this code?**

```
if (scaleValue >= 8.0) {
   message = "Most structures Fall";
}
if (scaleValue >= 7.0) {
   message = "Many buildings destroyed";
}
if (scaleValue >= 6.0) {
   message = "Many buildings damaged, some collapse";
}
if (scaleValue >= 4.5) {
   message = "Damage to poorly constructed buildings";
}
```

Using CASE to Multiway Branch

- **The switch statement chooses a case based on an integer value**
  - **break** ends each **case**
  - **default** catches all other values

If the `break` is missing, the case *falls through* to the next case's statements.

switch statements can also work with character and String values, but not double or boolean values.

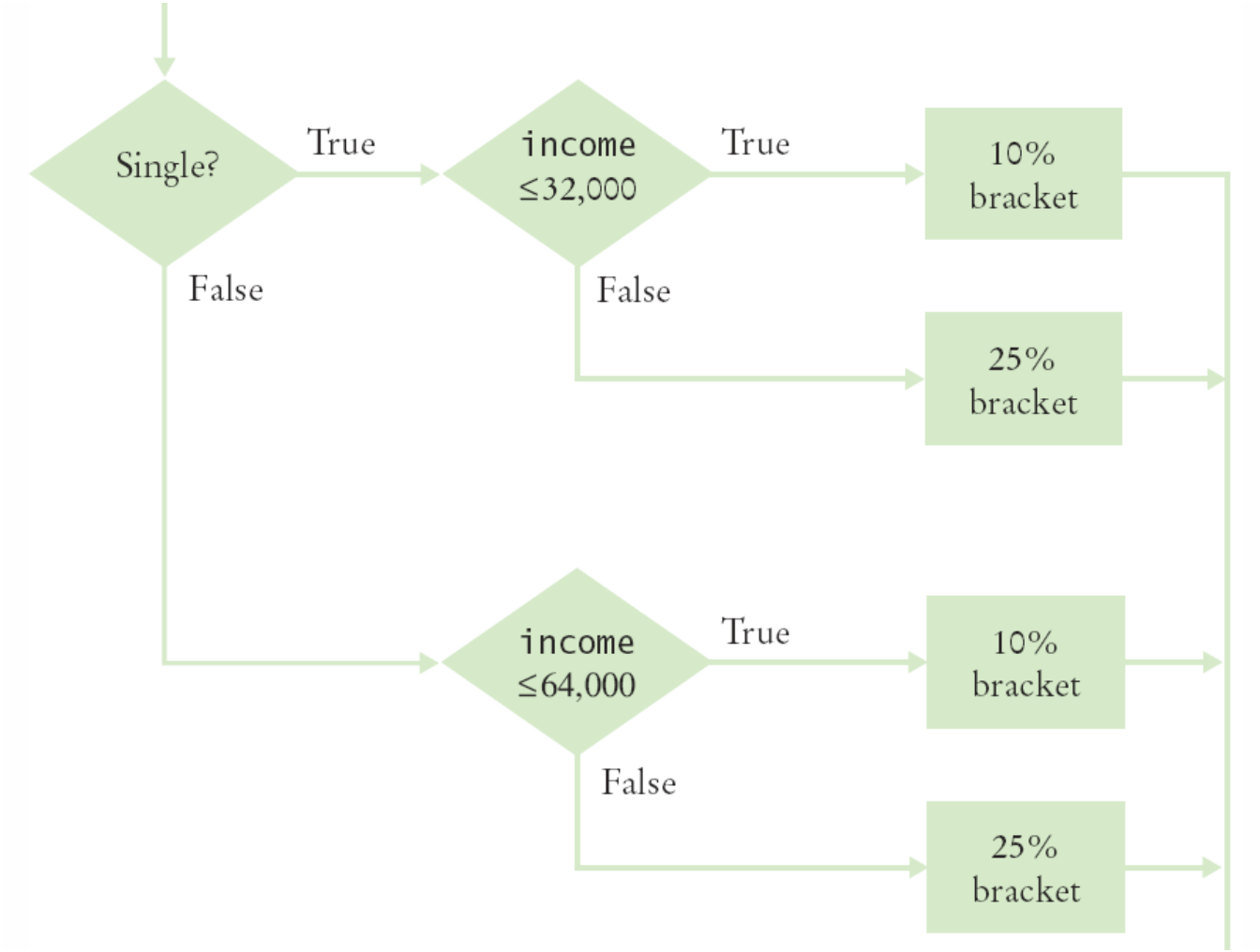
```
int digit = . . .;
switch (digit)
{
   case 1: digitName = "one";   break;
   case 2: digitName = "two";   break;
   case 3: digitName = "three"; break;
   case 4: digitName = "four";  break;
   case 5: digitName = "five";  break;
   case 6: digitName = "six";   break;
   case 7: digitName = "seven"; break;
   case 8: digitName = "eight"; break;
   case 9: digitName = "nine";  break;
   default: digitName = "";     break;
}
```

Nested IF Example: Calculating Taxes

- **Four outcomes**
  - Single
    - <= $32,000
    - > $32,000
  - Married
    - <= $64,000
    - > $64,000

Table 4 Federal Tax Rate Schedule		
If your status is Single and if the taxable income is	the tax is	of the amount over
at most $32,000	10%	$0
over $32,000	$3,200 + 25%	$32,000
If your status is Married and if the taxable income is	the tax is	of the amount over
at most $64,000	10%	$0
over $64,000	$6,400 + 25%	$64,000

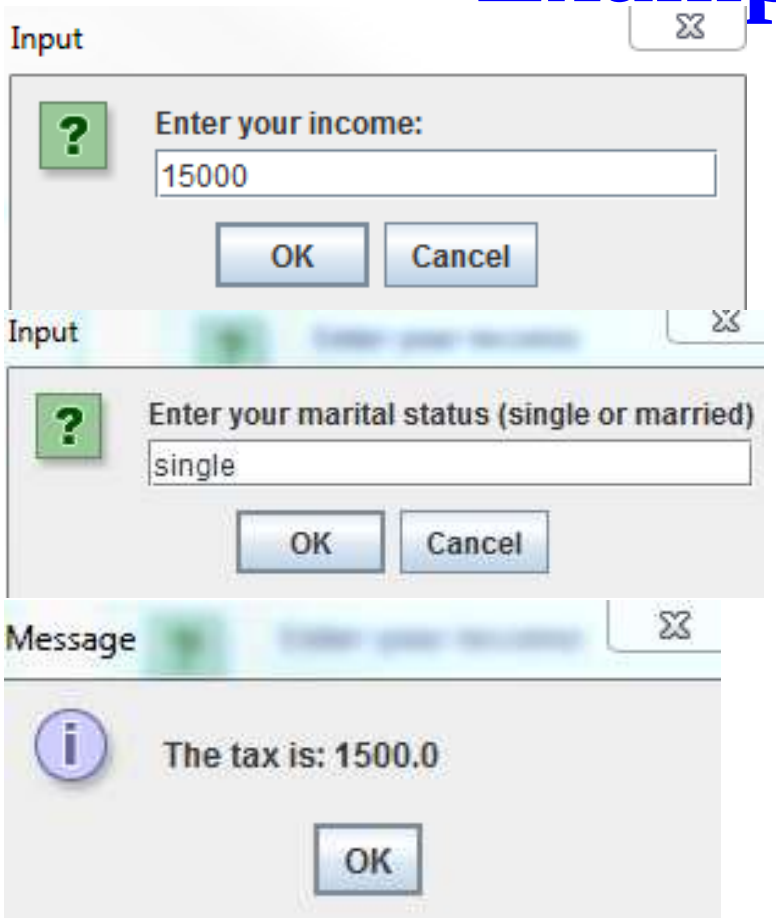
Flowchart for Tax Example

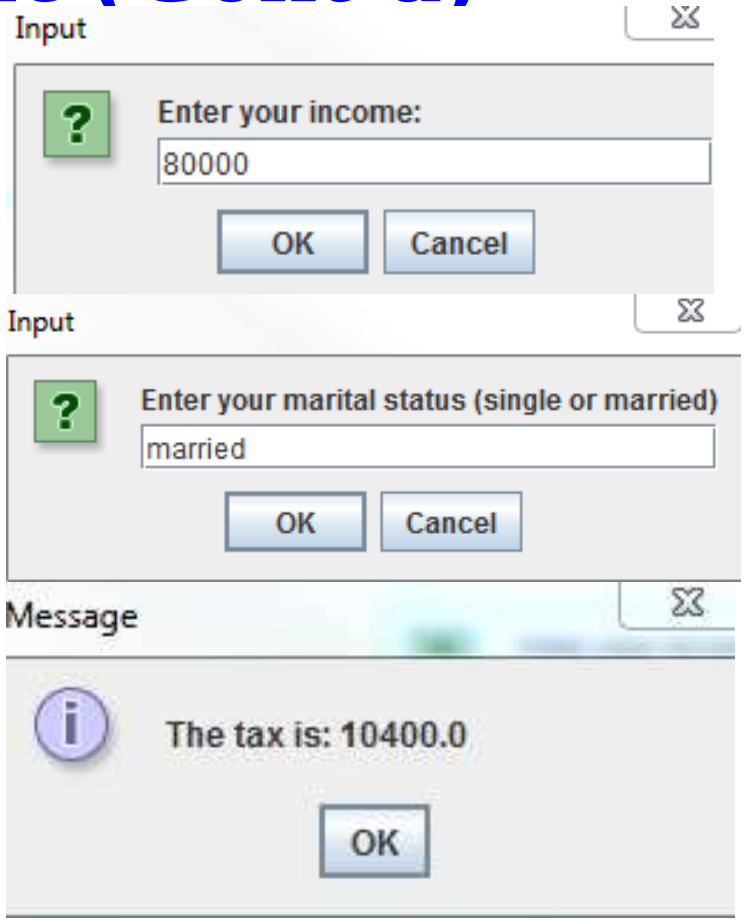


Completed Java Program for Tax Example

```
 8 public class TaxCalculator {
 9    public static void main(String[] args) {
10       // The tax code provided two rates
11       final double RATE1 = 0.10;
12       final double RATE2 = 0.25;
13       // The lower rate has limits depending on marital status
14       final double RATE1_SINGLE_LIMIT = 32000;
15       final double RATE1_MARRIED_LIMIT = 64000;
16       double income_tax;
17
18       // Read income and marital status
19       double income = Double.parseDouble(JOptionPane.showInputDialog("Enter your income:"));
20       String maritalStatus = JOptionPane.showInputDialog("Enter your marital status (single or married)");
21
22       if (maritalStatus.equals("single")) {
23          if (income <= RATE1_SINGLE_LIMIT) {
24             income_tax = RATE1 * income;
25          }
26          else {
27             //income_tax = $3200 + 25% of income over $32000
28             income_tax = (RATE1 * RATE1_SINGLE_LIMIT) + (RATE2 * (income - RATE1_SINGLE_LIMIT));
29          }
30       }
31       else {
32          if (income <= RATE1_MARRIED_LIMIT) {
33             income_tax = RATE1 * income;
34          }
35          else {
36             //income_tax = $6400 + 25% of income over $64000
37             income_tax = (RATE1 * RATE1_MARRIED_LIMIT) + (RATE2 * (income - RATE1_MARRIED_LIMIT));
38          }
39       }
40       JOptionPane.showMessageDialog(null, "The tax is: " + income_tax);
41    }
42 }
```

Completed Java Program for Tax Example (Cont'd)





Choosing Test Cases

A *boundary case* is a value that is tested in the code.

- **Choose input values that:**
  - Test boundary cases and 0 values

Test Case		Expected Output	Comment
30,000	s	3,000	10% bracket
72,000	s	13,200	3,200 + 25% of 40,000
50,000	m	5,000	10% bracket
104,000	m	16,400	6,400 + 25% of 40,000
32,000	m	3,200	boundary case
0		0	boundary case

Common Error: Dangling Else Problem

- **When an if statement is nested inside another if statement, errors can occur without proper {}**
- **The indentation suggests the else is related to the if country("USA")**
  - **Else** clause is always associated to the closest **if**

```
double shippingCharge = 5.00; // $5 inside continental U.S.
if (country.equals("USA"))
  if (state.equals("HI"))
    shippingCharge = 10.00;   // Hawaii is more expensive
else // Pitfall!
  shippingCharge = 20.00;     // As are foreign shipment
```

Missing {. Would solve problem

What happens when the state is Virginia?

Boolean Variables and Operators

- **Boolean variables are often called flags because they can either be up (true) or down (false)**
  - Think of a light switch
- **boolean is a Java data type**
- **Boolean operators: &&, | |, and !**
  - **&&** is the *and* operator
  - || is the *or* operator
  - **!** (not) is used to invert a boolean value

Combined Conditions: &&

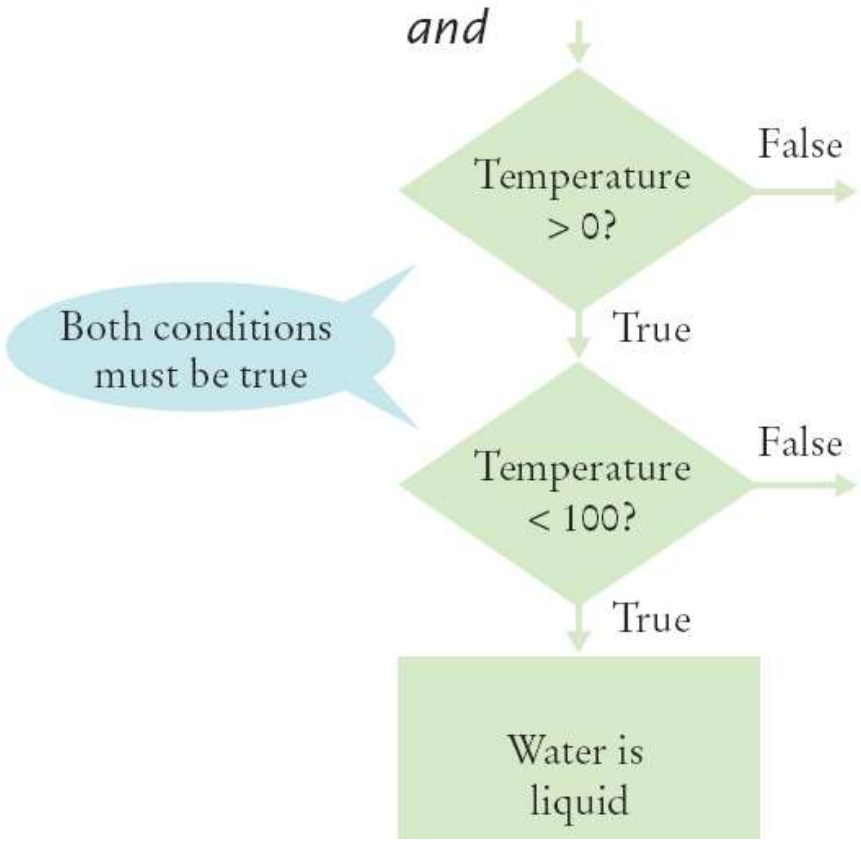
- **Combining two conditions is often used in range checking**
  - Is a value between two other values?
- **<u>Both</u> sides of the *and* must be true for the result to be true**

```
if (temp > 0 && temp < 100) {
  state = "Liquid";
}
```

A	B	A && B
true	true	true
true	false	false
false	true	false
false	false	false

Flowchart Using AND

```
if (temp > 0 && temp < 100) {
  state = "Liquid";
}
```



Lazy Evaluation: &&

- **Java uses lazy evaluation**
- **Combined conditions are evaluated from left to right**
  - If the left half of an *and* condition is false, why look further?
  - Put conditions more likely to fail first

```
if (GPA > 3.9 && numCredits > 0) {
  awardEligible = true;
}
```

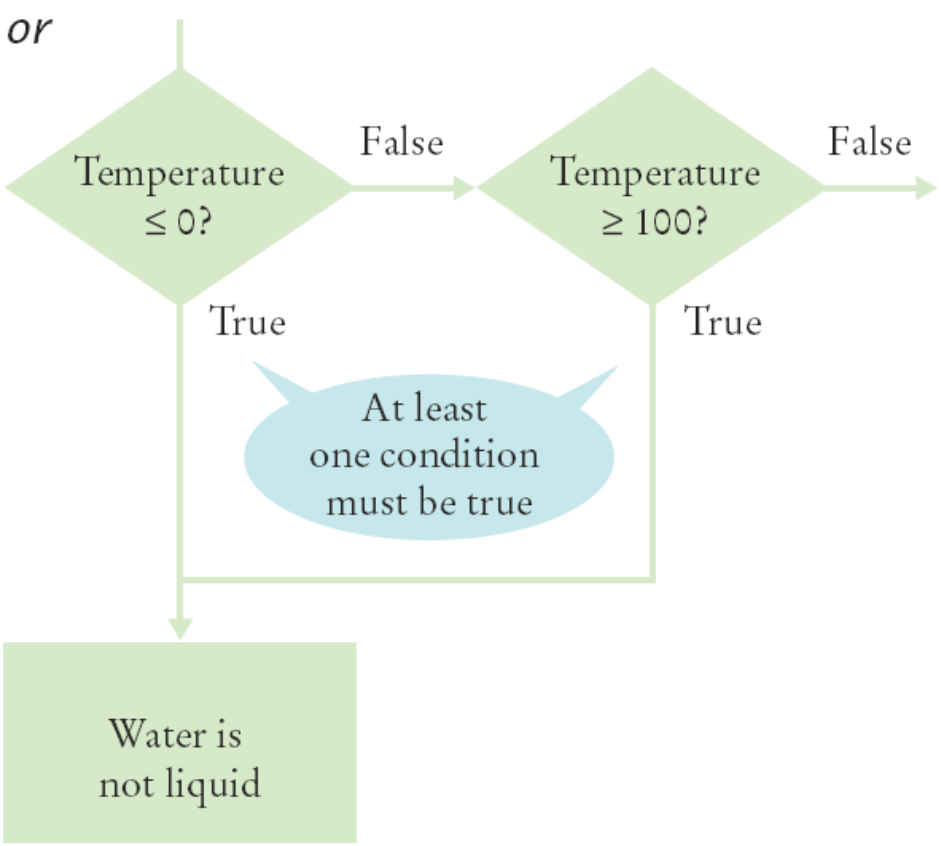
Combined Conditions: ||

- **Combining two conditions is often used in range checking**
  - Is a value outside two other values?
- **Only one side of the *or* must be true for the result to be true**

```
if (temp <= 0 || temp >= 100) {
  state = "Not Liquid";
}
```

A	B	A || B
true	true	true
true	false	true
false	true	true
false	false	false

Flowchart Using OR




```
if (temp <= 0 || temp >= 100) {
  state = "Not Liquid";
}
```

Lazy Evaluation: ||

- **Java uses lazy evaluation**
- **Combined conditions are evaluated from left to right**
  - If the left half of an *or* condition is true, why look further?
  - Put conditions more likely to be true first

```
if (GPA > 3.9 || numCredits > 0) {
  awardEligible = true;
}
```

The NOT Operator: !

- If you need to invert a boolean variable or comparison, precede it with !

```
if (!attending || grade < 60) {
  action = "Drop Student";
}
```

```
if (attending && !(grade >= 60)) {
  action = "Tutor Student";
}
```

A	!A
true	false
false	true

Boolean Operator Examples

Table 6 Boolean Operator Examples

Expression	Value	Comment
`0 < 200 && 200 < 100`	`false`	Only the first condition is true.
`0 < 200 || 200 < 100`	`true`	The first condition is true.
`0 < 200 || 100 < 200`	`true`	The `||` is not a test for "either-or". If both conditions are true, the result is true.
`0 < x && x < 100 || x == -1`	`(0 < x && x < 100) || x == -1`	The `&&` operator has a higher precedence than the `||` operator see Page 143
`0 < x < 100`	`Error`	**Error:** This expression does not test whether `x` is between `0` and `100`. The expression `0 < x` is a Boolean value. You cannot compare a Boolean value with the integer 100.

Boolean Operator Examples (Cont'd)

	`x && y > 0`	`Error`	**Error:** This expression does not test whether `x` and `y` are positive. The left hand side `x` of `&&` is an integer, the right hand side `y > 0` is a Boolean value. You cannot use `&&` with an integer argument.
	`!(0 < 200)`	`false`	`0 < 200` is `true`, therefore its negation is `false`.
	`frozen == true`	`frozen`	There is no need to compare a Boolean variable with `true`.
	`frozen == false`	`!frozen`	It is clearer to use `!` than to compare with `false`.

Common Errors

```
if (0 <= temp <= 100)  // Syntax error!
```

- **Combining multiple relational operators**
  - Can be used in math, not in Java
  - Requires two comparisons
  - Use: `if (temp >= 0 && temp <= 100)`

```
if (input == 1 || 2)  // Syntax error!
```

- **Also bad**
  - Use: `if (input == 1 || input == 2)`

Common Errors (Cont'd)

- **Confusing && and ||**
  - Surprisingly common error to confuse *and* and *or* conditions
    - A value lies between 0 and 100 if it is at least 0 **and** at most 100
    - A value lies outside that range if it is less than 0 **or** greater than 100
  - There is no golden rule, you just have to think carefully, and check your pseudocode
    - Remember Desk Checking!

Practice Problem

- **Scenario**
  - You have been asked to create a program that allows the user to input two integers and then displays their sum, difference, product, and quotient with a message for each (e.g. The sum is). The quotient calculation should only be performed if the second integer does not equal zero. The user should be informed if an error occurs.
- **To Do:**
  - Create a defining diagram
  - Create a solution algorithm using pseudocode
  - Write the program with Java

Questions?

Content

- Components of the Java Environment
- Your First Java Program
- Variables and Primitive Data Types
- Selection Statements
- Loop Statements
- Methods

Overview: Types of Loops

- **Java has three types of loops:**
  - **while** loops
  - **for** loops
  - **do** loops
- **Each loop requires the following steps**
  - Initialization (get ready to start looping)
  - Condition (test if we should execute loop body)
  - Update (change something each time through)

THE WHILE LOOPS

The while loop

- **Examples of loop applications**
  - Calculating compound interest
  - Simulations, event driven programs
- **Compound interest algorithm (from earlier in the class)**

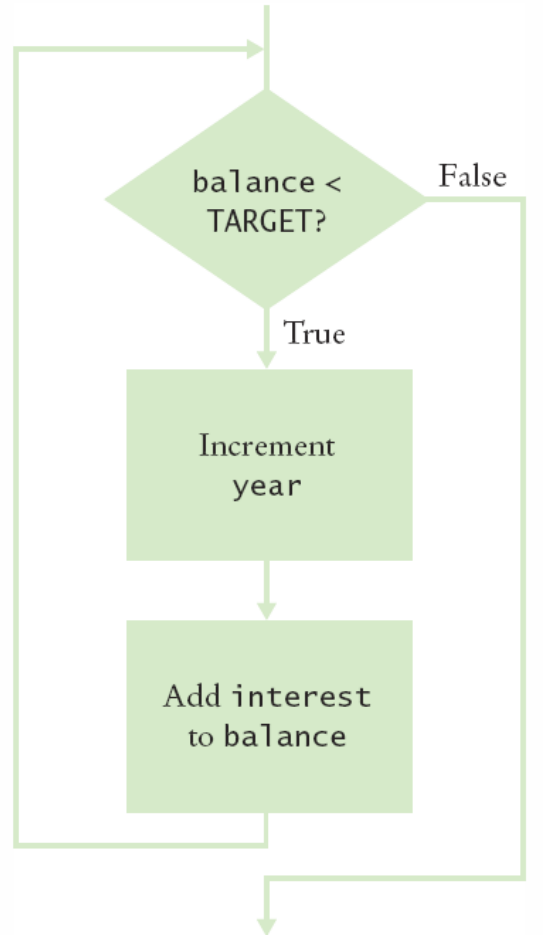
Start with a year value of 0 and a balance of $10,000.

year	balance
0	$10,000

How?

```
DOWHILE balance < 20000
   year = year + 1
   balance = balance * 1.05 (a 5% increase)
ENDDO
```

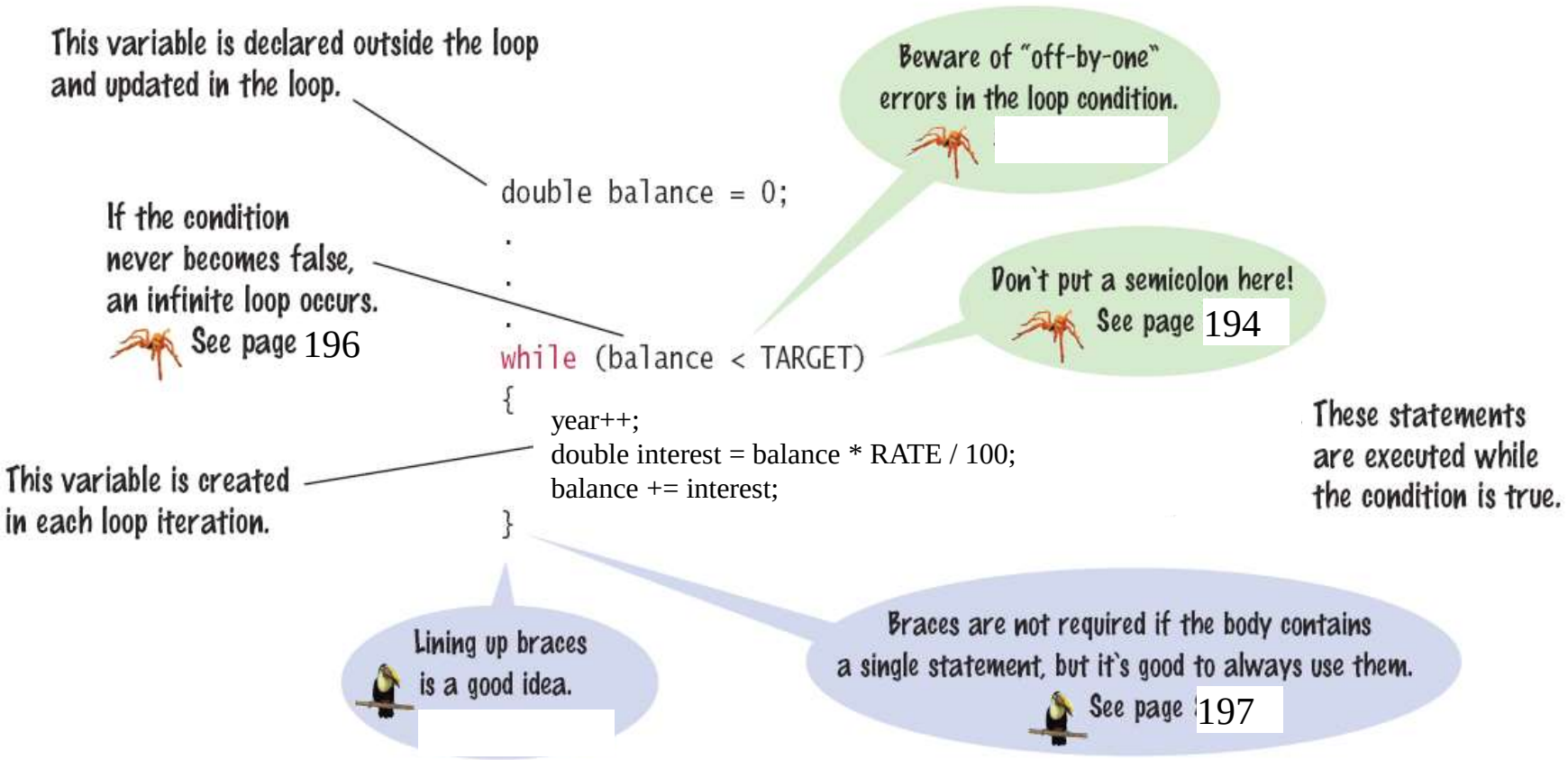
Planning the while loop



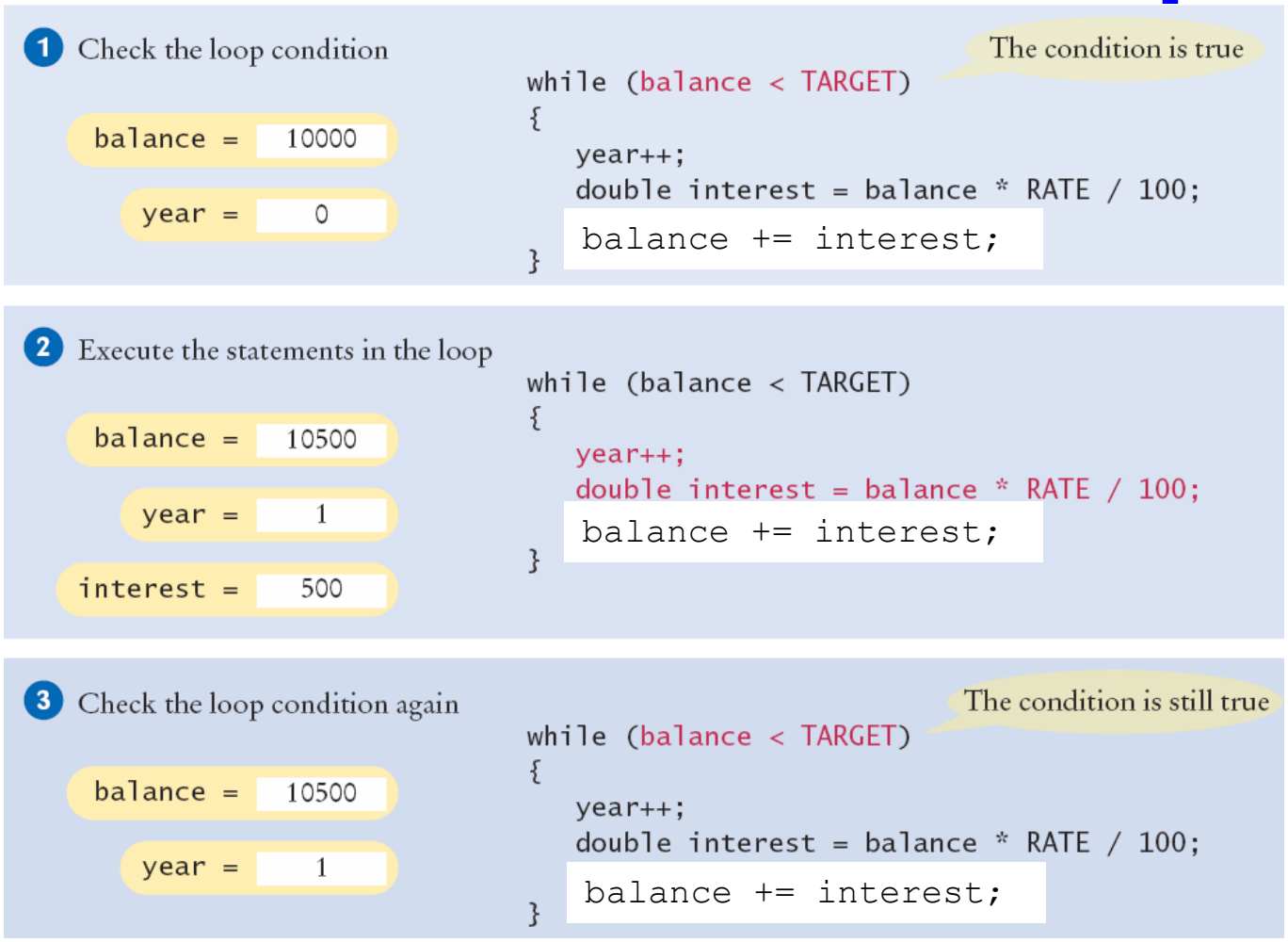
A loop executes instructions repeatedly while a condition is true.

```
while (balance < TARGET) {
   year++;
   double interest = balance * RATE/100;
   balance += interest;
}
```

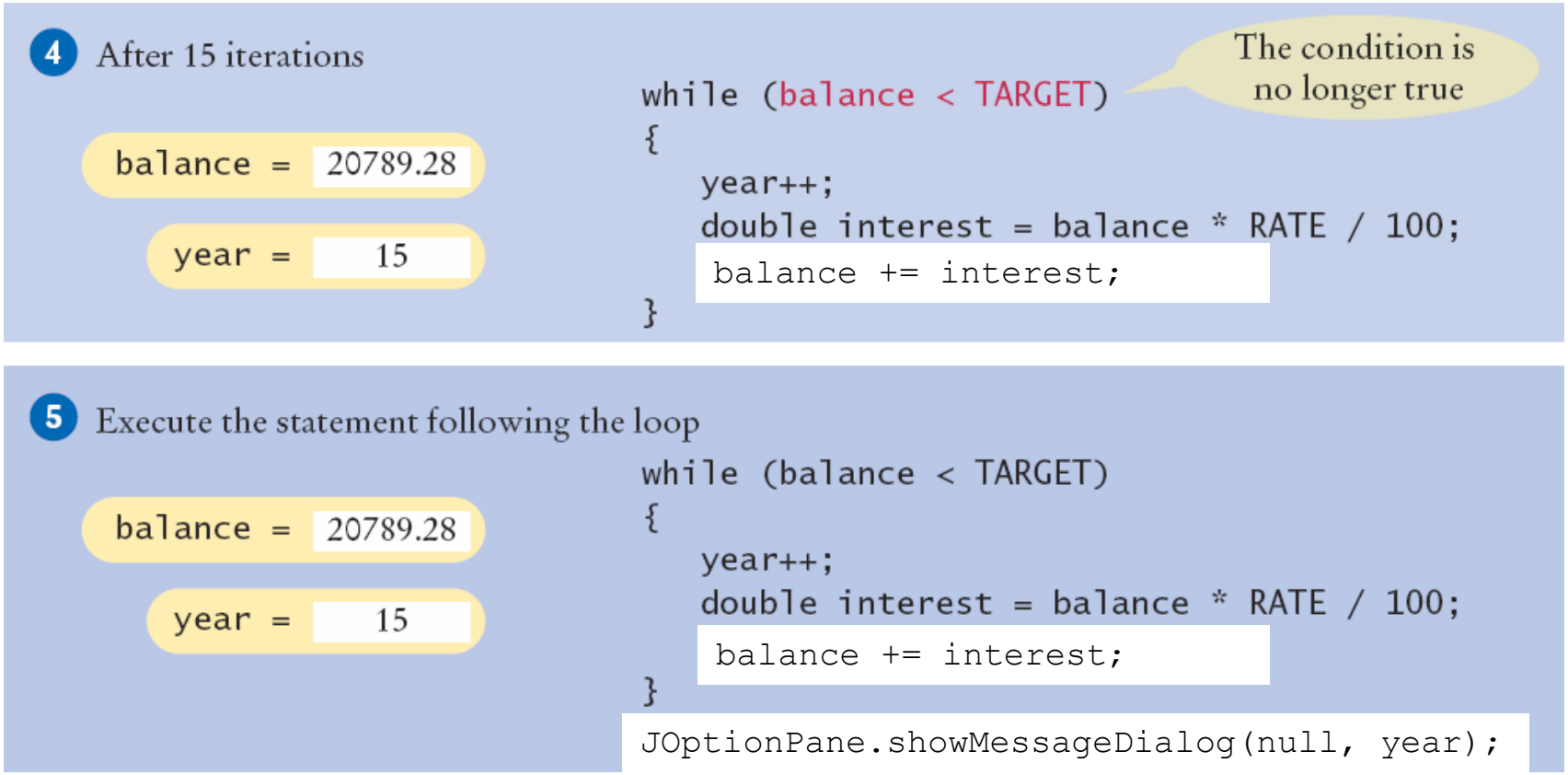
Syntax: while Statement



Execution of the Loop



Execution of the Loop (Cont'd)



Example: Double Investment Computation

```
 7 public class DoubleInvestment {
 8    public static void main(String[] args) {
 9       final double RATE = 5;
10       final double INITIAL_BALANCE = 10000;
11       final double TARGET = INITIAL_BALANCE * 2;
12
13       double balance = INITIAL_BALANCE;
14       int year = 0;
15
16       // Count the years required for the investment to double
17
18       while (balance < TARGET) {
19          year++;
20          double interest = balance * RATE / 100;
21          balance += interest;
22       }
23
24       JOptionPane.showMessageDialog(null,
25          "The investment doubled after: " + year + " years.");
26    }
27 }
```

Declare and initialize a variable outside of the loop to count `years`

Increment the `years` variable each time through

while Loop Examples

Loop	Output	Explanation

`i = 0; sum = 0;`
`while (sum < 10)`
`{`
`  i++; sum = sum + i;`
`  Print i and sum;`
`}`
	1 1
	2 3
	3 6
	4 10
		When sum is 10, the loop condition is false, and the loop ends.

`i = 0; sum = 0;`
`while (sum < 10)`
`{`
`  i++; sum = sum - i;`
`  Print i and sum;`
`}`
	1 -1
	2 -3
	3 -6
	4 -10
	. . .
		Because sum never reaches 10, this is an "infinite loop" (see page 196)

`i = 0; sum = 0;`
`while (sum < 0)`
`{`
`  i++; sum = sum - i;`
`  Print i and sum;`
`}`
	(No output)	The statement sum < 0 is false when the condition is first checked, and the loop is never executed.

while Loop Examples (Cont'd)

Loop	Output	Explanation

`i = 0; sum = 0;`
`while (sum >= 10)`
`{`
`  i++; sum = sum + i;`
`  Print i and sum;`
`}`
	(No output)	The programmer probably thought, "Stop when the sum is at least 10." However, the loop condition controls when the loop is executed, not when it ends

`i = 0; sum = 0;`
`while (sum < 10) ;`
`{`
`  i++; sum = sum + i;`
`  Print i and sum;`
`}`
	(No output, program does not terminate)	Note the semicolon before the `{`. This loop has an empty body. It runs forever, checking whether `sum < 0` and doing nothing in the body.

Common Error: No Loop Execution

- **Are we there yet?**
  - The loop body will only execute if the test condition is **TRUE**
  - If **bal** is supposed to grow until **TARGET**
    - Which version will execute the loop body?

```
while (bal < TARGET) {
  year++;
  interest = bal * RATE;
  bal += interest;
}
```

```
while (bal >= TARGET) {
  year++;
  interest = bal * RATE;
  bal += interest;
}
```

Common Error: Infinite Loops (Cont'd)

- **Infinite Loops**
  - A loop body will execute until the test condition becomes **FALSE**
  - What if you forget to update the test variable?
    - **bal** in this case (**TARGET** doesn't change)
    - This will loop forever until the program is stopped or crashes from running out of memory

```
while (bal < TARGET) {
  year++;
  interest = bal * RATE;
}
```

Common Error: Off-By-One

- A "counter" variable is often used in the condition test
- Counters can start at 0 or 1, but typically start at 0
- If you wanted to paint all 5 fingers, when are you done?

```
int finger = 0;
final int FINGERS = 5;
while (finger < FINGERS) {
  // paint finger
  ++finger;
}
```

```
int finger = 1;
final int FINGERS = 5;
while (finger <= FINGERS) {
  // paint finger
  ++finger;
}
```

Common Error: Empty Body

- **You probably have developed the habit of typing a semicolon at the end of each line**
- **Don't do this with loop statements!**
  - The loop body becomes very short
    - Between the closing) and the ;
    - Infinite loop!

```
while (bal < TARGET);
{
   year++;
   interest = bal * RATE;
   bal += interest;
}
```

Summary of the while Loop

- **while loops are very commonly used**
- **Initialize variables before you test**
- **The condition is tested BEFORE the loop body**
  - This is called pre-test
  - The condition often uses a counter variable
- **Something inside the loop must change the variable used in the test**
- **Watch out for infinite loops!**

THE FOR LOOPS

The for Loop

- **Use a for loop when you:**
  - Can use an integer counter variable
  - Have a constant increment (or decrement)
  - Have a fixed starting and ending value for the counter

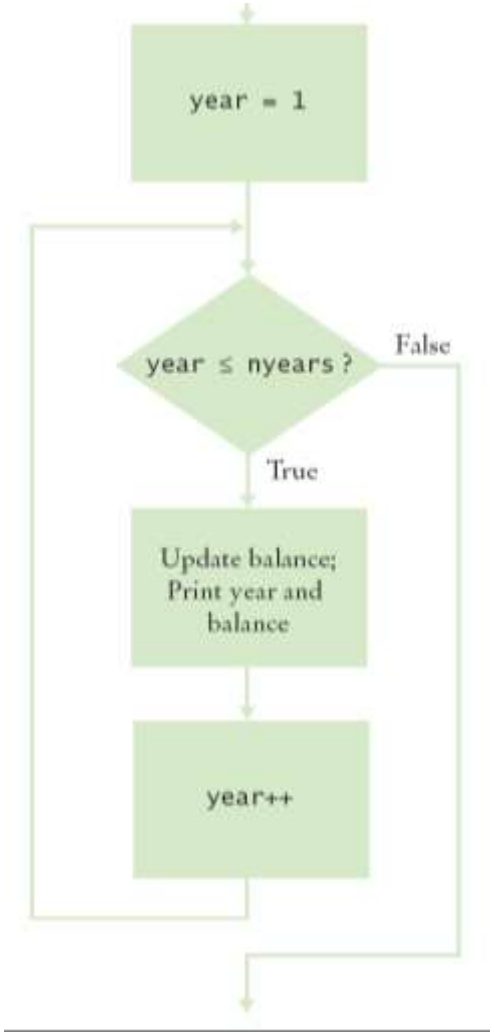
```
int i = 5;  // initialize
while (i <= 10) {  // test
  sum += i;
  i++; // update
}
```

while loop version

```
for (int i = 5; i <= 10; i++) {
  sum += i;
}
```

for loop version

Planning a for Loop



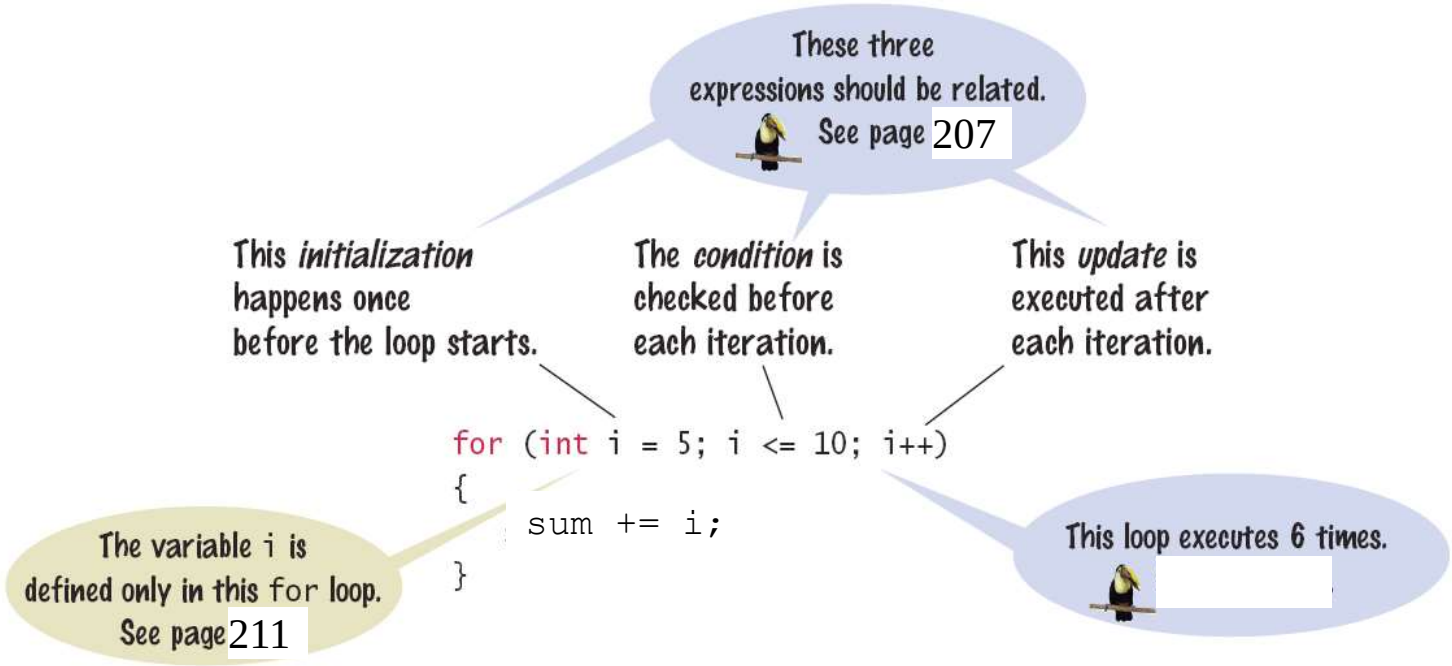
- **Print the balance at the end of each year for a number of years**

```
for (int year = 1; year <= nyears; year++)
{
   Update balance.
   Print year and balance.
}
```

Year	Balance
1	10500.00
2	11025.00
3	11576.25
4	12155.06
5	12762.82

Syntax: for Statement

- **Semicolons separate each part**
  - Initialization
  - Condition
  - Update



Why use a for Loop?

- **while loops can do everything a for loop, can do, but many programmers prefer them because it is concise**
  - Initialization, condition, and update, all on one line

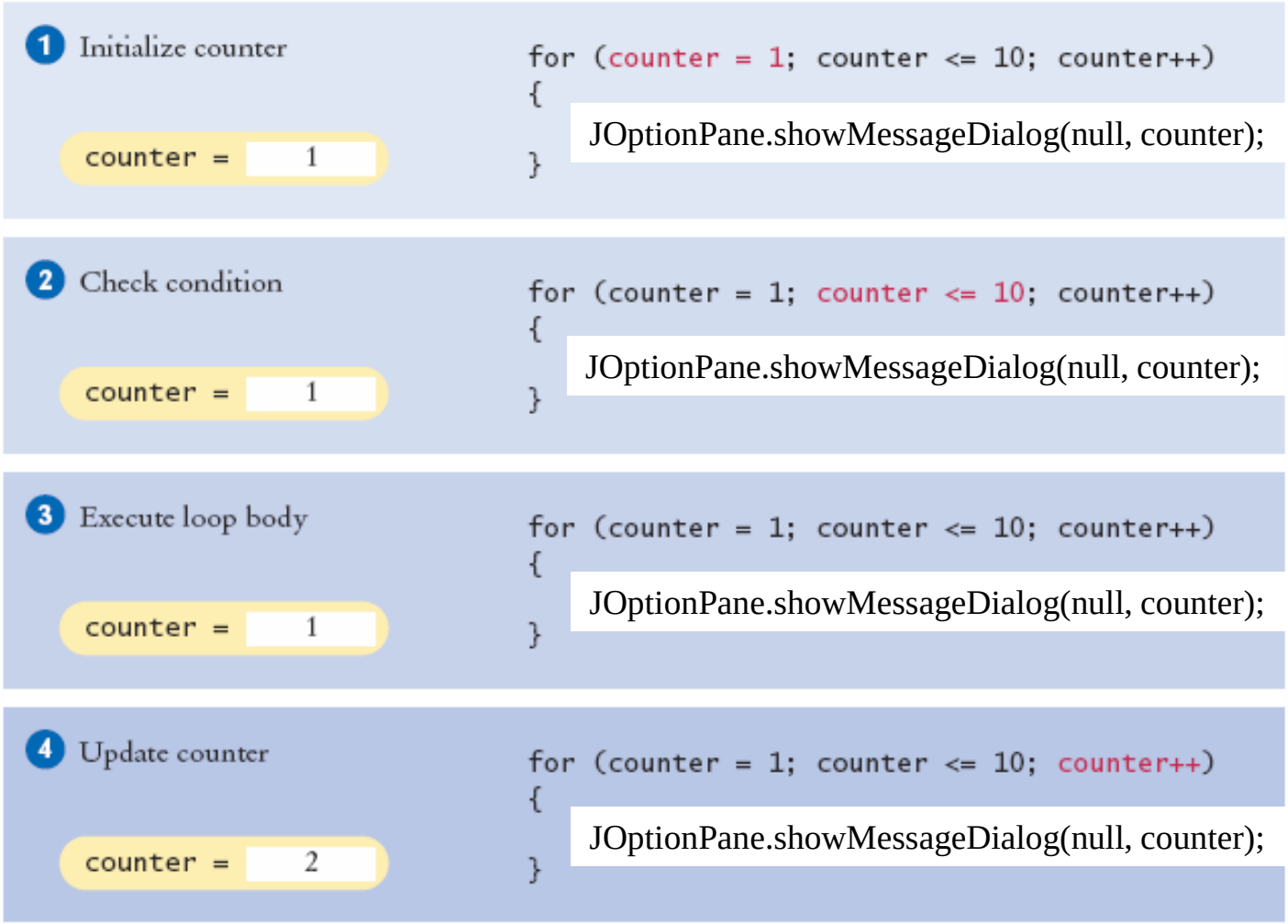
In general, the `for` loop:

```
for (initialization; condition; update)
{
    statements
}
```

has exactly the same effect as the `while` loop:

```
initialization;
while (condition)
{
    statements
    update
}
```

Execution of a for Loop



for Loop Examples

Table 2 for Loop Examples

Loop	Values of i	Comment
`for (i = 0; i <= 5; i++)`	0 1 2 3 4 5	Note that the loop is executed 6 times.
`for (i = 5; i >= 0; i--)`	5 4 3 2 1 0	Use `i--` for decreasing values.
`for (i = 0; i < 9; i = i + 2)`	0 2 4 6 8	Use `i = i + 2` for a step size of 2.
`for (i = 0; i != 9; i = i + 2)`	0 2 4 6 8 10 12 14 ... (infinite loop)	You can use `<` or `<=` instead of `!=` to avoid this problem.
`for (i = 1; i <= 20; i = i * 2)`	1 2 4 8 16	You can specify any rule for modifying `i`, such as doubling it in every step.

for Loop Variable Scope

- **Scope is the "lifetime" of a variable**
  - When "x" is declared in the **for** statement, it only exists inside the "block" of the loop

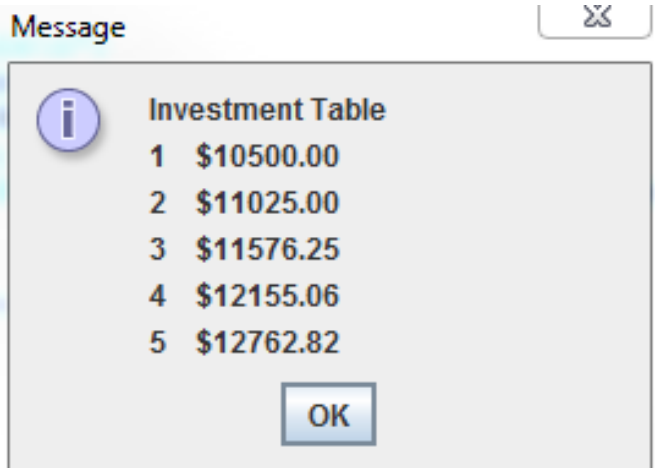
```
for(int x = 1; x < 10; x++) {
  // steps to do inside the loop
  // You can use ‘x’ anywhere in this box
}
if (x > 100)   // Error! x is out of scope!
```

  - If the value of "x" is needed after the for loop, you should declare "x" before the for loop

```
int x;
for(x = 1; x < 10; x++)
```

Example: Creating an Investment Table

```
7  public class InvestmentTable {
8     public static void main(String[] args) {
9        // Setup variables
10       final double RATE = 5;
11       final double INITIAL_BALANCE = 10000;
12       double balance = INITIAL_BALANCE;
13       String table = "";
14
15       // Get number of years to run table
16       int numYears;
17       do {
18          try {
19             numYears = Integer.parseInt(JOptionPane.showInputDialog(
20             "Enter number of years:"));
21          }
22          catch (NumberFormatException e) {
23             numYears = 0;
24          }
25          if (numYears <= 0) {
26             JOptionPane.showMessageDialog(null, "Invalid number of years!");
27          }
28       }
29       while (numYears <= 0);
30
31
32       // Store table header in variable
33       table = table + "Investment Table\n";
34       // Loop and store the balances of each year in a variable
35       for (int year = 1; year <= numYears; year++) {
36          double interest = balance * RATE / 100;
37          balance = balance + interest;
38          table += year + "    $" + String.format("%.2f", balance) + "\n";
39       }
40
41       // Print the table
42       JOptionPane.showMessageDialog(null, table);
43    }
44 }
```



Tips on for Loops

- **Use for loops for their intended purpose only**
  - Increment (or decrement) by a constant value
  - Do not update the counter inside the body
    - Poor design!
    - Only update in the third section of the header
  - Do not use 'break' inside the body
    - Poor design!
    - As a pre-test loop, the loop should only end once the condition evaluates to false
  - Most counters start at 0 or 1
    - Use an integer (int) named "counter", "i", or "x" for index variable

```
for (int counter = 1; counter <= 100; counter++) {
  if (counter % 10 == 0) { // Skip values divisible by 10
    counter++; // Bad style: Do NOT update counter inside loop
  }
  JOptionPane.showMessageDialog(null, counter);
}
```

Tips on for Loops (Cont'd)

- **Count iterations**
  - Many bugs are created by "off by one" issues
- **Example: Are you trying to show 4 or 5**

```
final int RAILS = 5;
for (int i = 1; i < RAILS; i++ ) {
   JOptionPane.showMessageDialog(null, "Painting rail
    " + i);
}
```

```
Painting rail 1
Painting rail 2
Painting rail 3
Painting rail 4
```

```
final int RAILS = 5;
for (int i = 1; i <= RAILS; i++ ) {
   JOptionPane.showMessageDialog(null, "Painting rail
    " + i);
}
```

```
Painting rail 1
Painting rail 2
Painting rail 3
Painting rail 4
Painting rail 5
```

Summary of the for Loop

- **for loops are very commonly used when a specific number of iterations is known**
- **They have a very concise notation**
  - Initialization; condition; increment
  - Initialization happens once at start
- **The condition is tested <u>BEFORE</u> executing the loop body (pre-test)**
  - Loop body only executes if the condition is true
  - Increment is done at the end of the body
- **Great for integer counting**
- **Watch out for infinite loops!**

THE DO LOOPS

The do Loop

- **Use a do loop when you want to ensure the loop runs at least once**
- **Example: Range check a value entered**
  - User must enter a valid value before proceeding

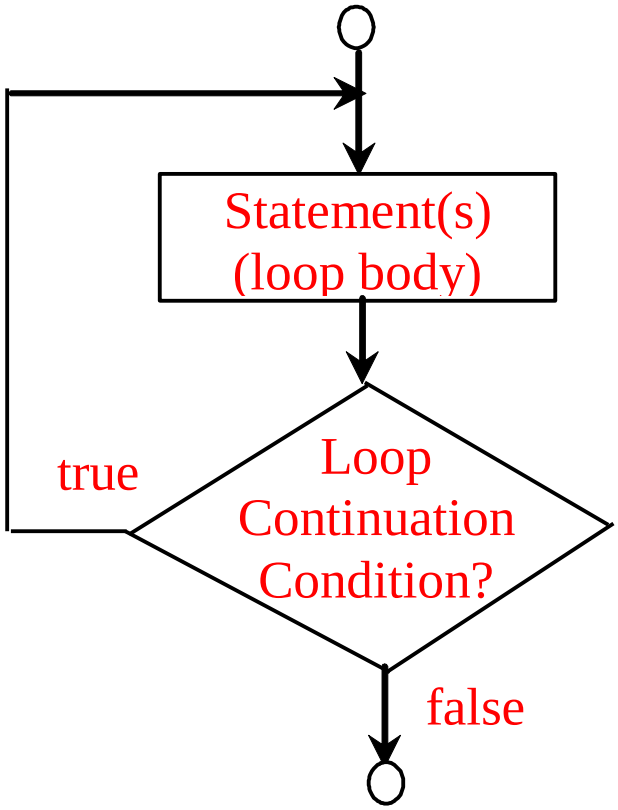
```
 7 public class EnterIntegerBelowValue {
 8    public static void main(String[] args) {
 9       final int MAXIMUM_VALUE = 100;
10
11       int value;
12
13       do {
14          try {
15             value = Integer.parseInt(JOptionPane.showInputDialog("Enter an integer < " + MAXIMUM_VALUE));
16          }
17          // Catch any non-numeric values entered, and set the value equal to something out of range
18          catch (NumberFormatException e) {
19             value = MAXIMUM_VALUE;
20          }
21          if (value >= MAXIMUM_VALUE) {
22             JOptionPane.showMessageDialog(null, "Sorry, you did not enter a valid value. Please try again.");
23          }
24       } while (value >= MAXIMUM_VALUE);
25    }
26 }
```

Syntax: do Loop

- **Syntax:**

```
do {
   // Loop body;
   Statement(s);
} while (loop-condition-
continuation);
```

Note the use
of the semicolon!



Which Loop to Use?

- **Interchangeable, but only one is the right choice!**
  - Definitive number of iterations: **for loop**
  - Indefinite number of iterations, but at least one: **do loop**
  - Indefinite number of iterations, and not necessarily at least one iteration: **while loop**

The correct choice will be expressly implied by your flowchart and program requirements!

COMMON LOOP ALGORITHMS

Processing With a Sentinel

- **Sentinel values are often used**
  - When you don't know how many items are in a list, you can use a special character or value to signify you are finished entering items
  - For numeric input of positive numbers, it is common to use the value **-1**

```
 7 public class ProcessingWithSentinel {
 8    public static void main(String[] args) {
 9       final String SENTINEL = "-1";
10
11       // Priming read
12       String inputValue = JOptionPane.showInputDialog(null, "Enter any number, or " + SENTINEL + " to quit");
13
14       while (!inputValue.equals(SENTINEL)) {
15          int value;
16          try {
17             value = Integer.parseInt(inputValue);
18          }
19          // Catch any non-numeric values entered
20          catch (NumberFormatException e) {
21             JOptionPane.showMessageDialog(null, "Sorry, you did not enter a valid value. Please try again.");
22          }
23
24          // Read in the next value
25          inputValue = JOptionPane.showInputDialog(null, "Enter any number, or -1 to quit");
26       }
27    }
28 }
```

Averaging a Set of Values

- **Pseudocode**

```
SET sum = 0
SET count = 0
PROMPT user for input
READ input
DOWHILE input <> -1
    sum = sum + input
    count = count + 1
    PROMPT user for input
    READ input
ENDDO
IF count > 0 THEN
    SET average = sum/count
    PRINT 'The average is ', average
ELSE
    PRINT 'No data!'
ENDIF
```

Java Example: Calculate Average Salary

```
3  public class AverageSalary {
4     public static void main(String[] args) {
5        final double MINIMUM_SALARY = 0.00;
6        double sum = 0;
7        int count = 0;
8
9        // Priming read for salary
10       String salaryInput = JOptionPane.showInputDialog("Enter salaries, or Q to quit");
11       while (!salaryInput.equalsIgnoreCase("Q")) {
12          double salary;
13          try {
14             salary = Double.parseDouble(salaryInput);
15          }
16          // Catch any bad values entered, and set the value equal to something out of range
17          catch (NumberFormatException e) {
18             salary = MINIMUM_SALARY - 1;
19          }
20
21          // If the salary is valid, process it. Otherwise, inform the user of an error.
22          if (salary >= MINIMUM_SALARY) {
23             sum += salary;
24             count++;
25             JOptionPane.showMessageDialog(null, "The salary has been recorded.");
26          }
27          else {
28             JOptionPane.showMessageDialog(null, "You entered an invalid salary. Please try again.");
29          }
30
31          // Read next salary
32          salaryInput = JOptionPane.showInputDialog("Enter salaries, or Q to quit");
33       }
34       if (count > 0) {
35          double average = sum/count;
36          JOptionPane.showMessageDialog(
37             null, "Average salary: $" + String.format("%.2f", average));
38       }
39       else {
40          JOptionPane.showMessageDialog(null, "No data!");
41       }
42    }
43 }
```

Outside the while loop, declare and initialize variables to use

Prompt the user for the first salary. This is known as a *priming read*

If a valid value is entered, update `sum` and `count` for average later

Prevent divide by 0

Inputting Numeric Values

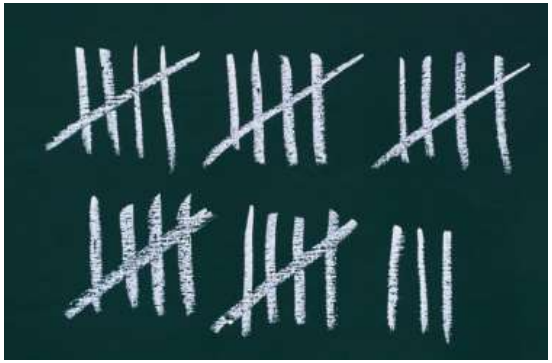
- **When valid values can be positive or negative numbers**
  - You cannot use -1 (or any other number) as a sentinel
- **One solution: Use a non-numeric sentinel**

```
9       // Priming read for salary
10      String salaryInput = JOptionPane.showInputDialog("Enter salaries, or Q to quit");
11      while (!salaryInput.equalsIgnoreCase("Q")) {
12         double salary;
13         try {
14            salary = Double.parseDouble(salaryInput);
15         }
16         // Catch any bad values entered, and set the value equal to something out of range
17         catch (NumberFormatException e) {
18            salary = MINIMUM_SALARY - 1;
19         }
20
21         // If the salary is valid, process it. Otherwise, inform the user of an error.
22         if (salary >= MINIMUM_SALARY) {
23            sum += salary;
24            count++;
25            JOptionPane.showMessageDialog(null, "The salary has been recorded.");
26         }
27         else {
28            JOptionPane.showMessageDialog(null, "You entered an invalid salary. Please try again.");
29         }
30
31         // Read next salary
32         salaryInput = JOptionPane.showInputDialog("Enter salaries, or Q to quit");
33      }
```

Counting Matches

- **Initialize a count to 0**
- **Use a for loop**
- **Increment count per match**

```
int upperCaseLetters = 0;
for (int i = 0; i < str.length(); i++) {
  char ch = str.charAt(i);
  if (Character.isUpperCase(ch)) {
    upperCaseLetters++;
  }
}
```



Finding the First Match

- **Example: Finding the first lower case letter**
- **Initialize boolean sentinel to false**
- **Initialize index counter to walk through string**
- **Use compound conditional within *while* loop**

```
boolean found = false;
int position = 0;
while (!found && position < str.length()) {
  if (Character.isLowerCase(str.charAt(position))) {
    found = true;
  }
  else {
    position++;
  }
}
```

Finding the Maximum (or Minimum)

- **Get the first input value**
  - This is the largest (or smallest) that you have seen (so far)
- **Loop while you have a valid number (non-sentinel)**
  - Get another input value
  - Compare the new input
  - Update largest (or smallest) if necessary

```
import javax.swing.JOptionPane;

public class FindMaximum {
   public static void main(String[] args) {
      int highestNumber;
      boolean firstNumberEntered = false;

      int firstNumber = 0;

      // Continue to prompt the user until a valid first number is entered
      do {
         // Get the first number. This program assumes at least one number must be entered.
         // If this is not the case, the structure will need to change
         String firstNumberInput = JOptionPane.showInputDialog("Enter the first number: ");

         try {
            firstNumber = Integer.parseInt(firstNumberInput);
            // Once any number is entered by the user, set a boolean flag
            // [Instructor Note: A boolean flag must be used here and in the
            // loop condition because if firstNumber was set to a value, the
            // loop condition would not know if that value was entered by the
            // user or the result of the user entering a bad value and the value being
            // set automatically since this problem states all numbers are valid]
            firstNumberEntered = true;
         }
         catch (NumberFormatException e) {
            JOptionPane.showMessageDialog(null, "Invalid number! Please try again.");
         }
      } while (!firstNumberEntered);

      // Set the highestNumber to be the first number entered
      highestNumber = firstNumber;

      // Priming read to gather additional numbers until the user indicates they are finished
      String numberInput = JOptionPane.showInputDialog("Enter another number, or Q to quit");
      while (!numberInput.equalsIgnoreCase("Q")) {
         int number;

         try {
            number = Integer.parseInt(numberInput);
            if (number > highestNumber) {
               highestNumber = number;
            }
         }
         // Catch any bad values entered
         catch (NumberFormatException e) {
            JOptionPane.showMessageDialog(null, "Invalid number!");
         }
         // Prompt again before going back to while loop
         numberInput = JOptionPane.showInputDialog("Enter another number, or Q to quit");
      }
      JOptionPane.showMessageDialog(null, "The highest number was: " + highestNumber);
   }
}
```

Simulating Die Tosses

- Goal: Get a random integer between 1 and 6

```
 3 public class Dice {
 4    public static void main(String[] args) {
 5       final int NUM_DIE_ROLLS = 10;
 6       String table = "";
 7
 8       // Store table header in variable
 9       table += "Dice Rolls\n";
10       // Loop through a set number of die rolls
11       for (int i=1; i <= NUM_DIE_ROLLS; i++) {
12
13          // Generate two random numbers between 1 and 6
14          int d1 = (int) (Math.random() * 6) + 1;
15          int d2 = (int) (Math.random() * 6) + 1;
16          table += d1 + " " + d2 + "\n";
17       }
18
19       // Print the table
20       JOptionPane.showMessageDialog(null, table);
21    }
22 }
```

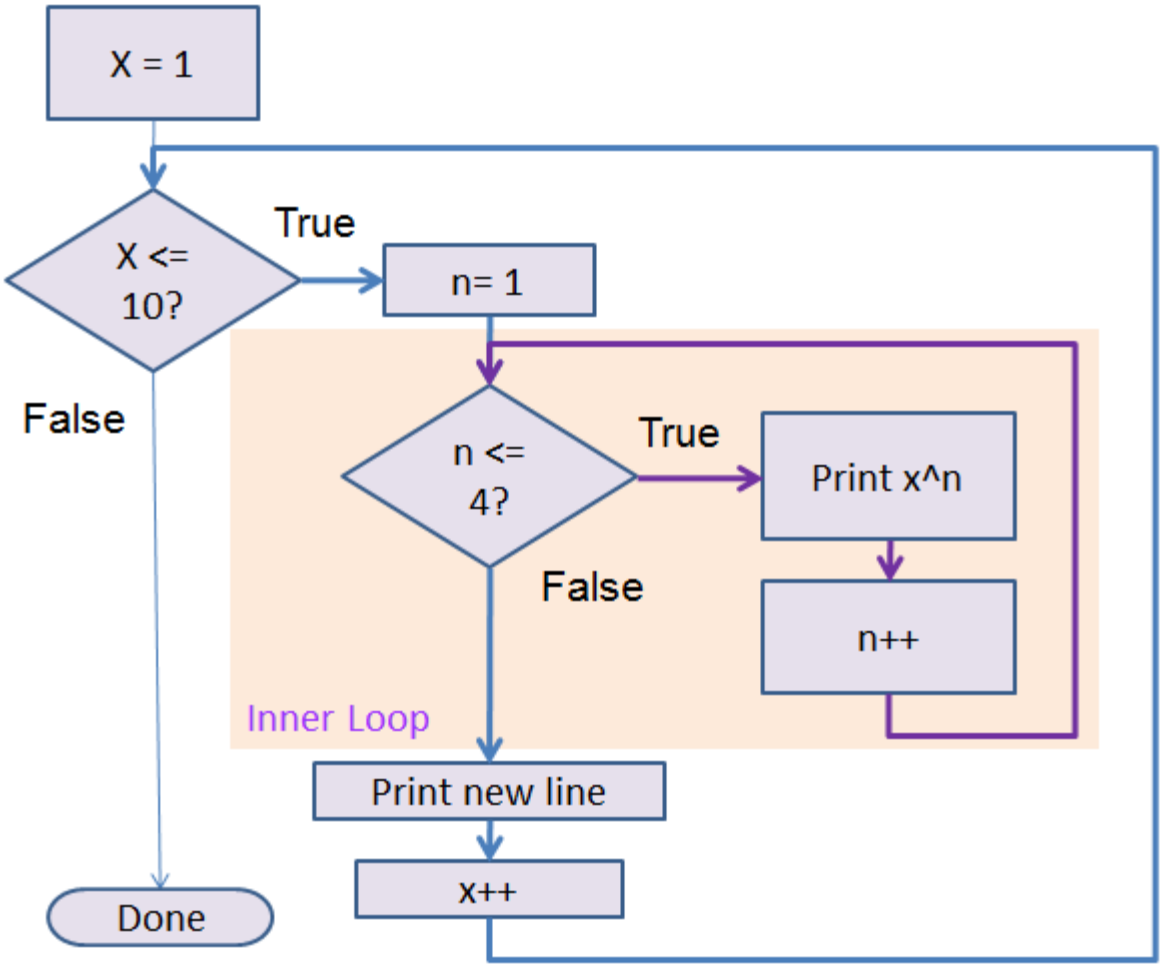
NESTED LOOPS

Nested Loops

- **How would you print a table with rows and columns?**
  - Print top line (header)
    - Use a for loop
  - Print table body
    - How many rows?
    - How many columns?
  - Loop per row
    - Loop per column

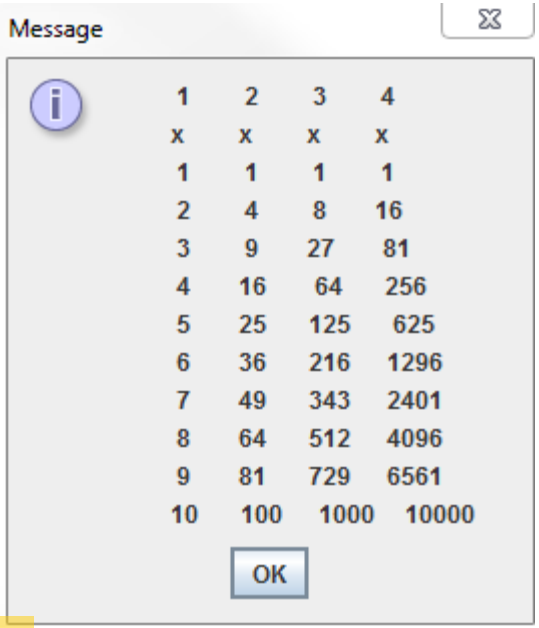
x^1	x^2	x^3	x^4
1	1	1	1
2	4	8	16
3	9	27	81
…	…	…	…
10	100	1000	10000

Flowchart of a Nested Loop



Java Example: Printing a Table of Powers

```
5  public class PowerTable {
6     public static void main(String[] args) {
7        final int NMAX = 4;
8        final double XMAX = 10;
9
10       String report = "";
11
12       //Print table head
13       // First row
14       for (int n=1; n<=NMAX; n++) {
15          report += String.format("%10d", n);
16       }
17       report += "\n";
18
19       // Second row
20       for (int n=1; n<=NMAX; n++) {
21          report += String.format("%10s", "x ");
22       }
23       report += "\n";
24
25       //Print table body
26
27       //Loop through each row
28       for (double x = 1; x <= XMAX; x++) {
29          //Loop through each column in each row
30          for (int n = 1; n<= NMAX; n++) {
31             report += String.format("%10.0f", Math.pow(x, n));
32          }
33          report += "\n";
34       }
35       JOptionPane.showMessageDialog(null, report);
36    }
37 }
```



Body of outer loop

Body of inner loop

Nested Loop Examples

`for (i = 1; i <= 3; i++)`
`{`
`  for (j = 1; j <= 4; j++)  { print "*" }`
`  print new line`
`}`
	****
	****
	****
		Prints 3 rows of 4 asterisks each.

`for (i = 1; i <= 4; i++)`
`{`
`  for (j = 1; j <= 3; j++) { print "*" }`
`  print new line`
`}`
	***
	***
	***
	***
		Prints 4 rows of 3 asterisks each.

`for (i = 1; i <= 4; i++)`
`{`
`  for (j = 1; j <= i; j++) { print "*" }`
`  print new line`
`}`
	*
	**
	***
	****
		Prints 4 rows of lengths 1, 2, 3, and 4.

Practice Problem

- **Scenario**
  - Based on a collection of payroll records, each containing a payroll amount, from user input, design an algorithm to process all of these amounts. This should continue until an amount of -1 is entered. At the end of the program, display the total payroll amount to the screen and the number of payroll amounts entered.
- **To Do:**
  - Create a defining diagram
  - Create a solution algorithm using pseudocode
  - Write the program with Java

Questions?

Content

- Components of the Java Environment
- Your First Java Program
- Variables and Primitive Data Types
- Selection Statements
- Loop Statements
- **Methods**

PROGRAMMING WITH MODULES (METHODS)

Review of Modules

- **A module is a section of an algorithm dedicated to a single function**
  - In Java, modules are referred to as methods
- **Allows the programmer to break a large task into manageable pieces**
  - Debugging is easier
  - Maintenance is easier
- **Allows for code reuse**
  - Method is called whenever the activity is required

Identifying Methods

- **After determining what needs to be done in processing (IPO Chart!), identify the different logical units of work/subtasks by grouping processes**
  - These will become the algorithm's modules, converted to Java methods
- **Java's main() method, serves as the mainline and functions as a master control, calling other modules when their actions are required**

Calling/Invoking Methods

- **Methods are *called/invoked* by referencing a named block of code**
- **The instructions of the method are then executed**

```
public static void main(String[] args)
{
  double z = Math.pow(2, 3);
  . . .
}
```

Examples of Methods

- **Some methods that have already been used**
  - Math.pow()
  - JOptionPane.showInputDialog()
  - JOptionPane.showMessageDialog()
  - main()
- **All methods have:**
  - A name – follows the same rules as variable names
  - () at the end – a place to receive input information

Methods – Two Parts

- **Declaration/Definition**
  - Describes what to be done when the method is called/invoked
  - Example: public static void **main**(String [] args)
    - main() is invoked by the JVM
- **Call/Invocation**
  - Tells Java to execute the statements within the method's definition
  - Example:
    - String name = JOptionPane.showInputDialog("Enter Your Name");
    - int result = Math.pow(2, 3);

Method Declaration

- **Method declarations must include:**
  - Method header
  - An opening curly brace
  - Method body
  - A closing curly brace
- **Methods are specified in the same class, but outside of other methods, including main()**
- **Example:**

```
public class HelloPrinter {
        public static void main (String[] args) {
                // code for main
        }
        public static void printHelloMessage() {
                // method body
        }
}
```

Method Header - Dissected

```
public static void main (String[] args)
```

1. **Access specifier – determines access to method**
   - public – method is accessible everywhere
   - private – method is only accessible inside of class
   - protected – method is only accessible within classes of the same package or from subclasses that inherit from the class
2. **Access modifier**
   - static – doesn't belong to a particular instance of an object
   - final – deals with restricting object inheritance
3. **Method return data type – data type of value returned from method**
   - can be any of the data types (int, double, String, etc.) or an object
   - must be void if the method does not return a value
4. **Method name – must be a valid identifier**
5. **Parameter list – list of values sent *into* the method**
   - Note in the main() method, values sent into the method would occur at the command line of running the Java program

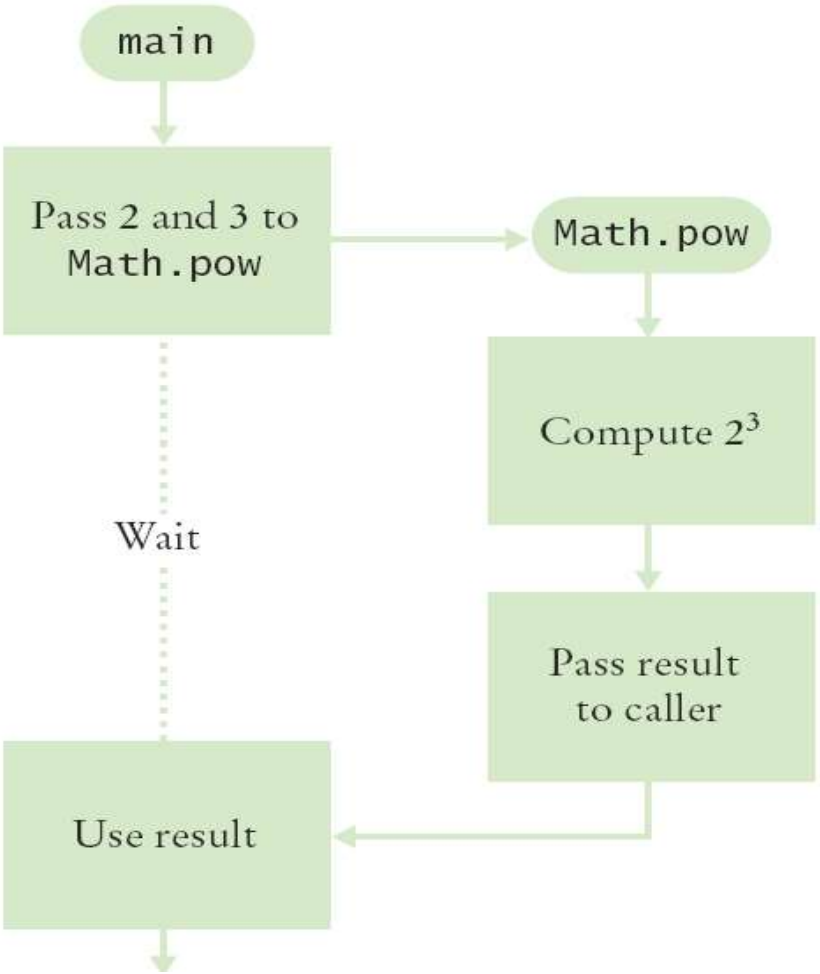
Method Calls/Invocations

- **You have already used methods written by others**
  - JOptionPane.showMessageDialog(null, message);

Method Name
Note: if method is not defined in the same class – include where to find it **JOptionPane.**

Parameter List
- Provide values INTO method
- Must have a corresponding parameter in the header to 'catch' the value to store locally
- Must match:
  - Number of parameters
  - Datatype(s) of parameters
  - Logical order

Calling/Invoking a Method

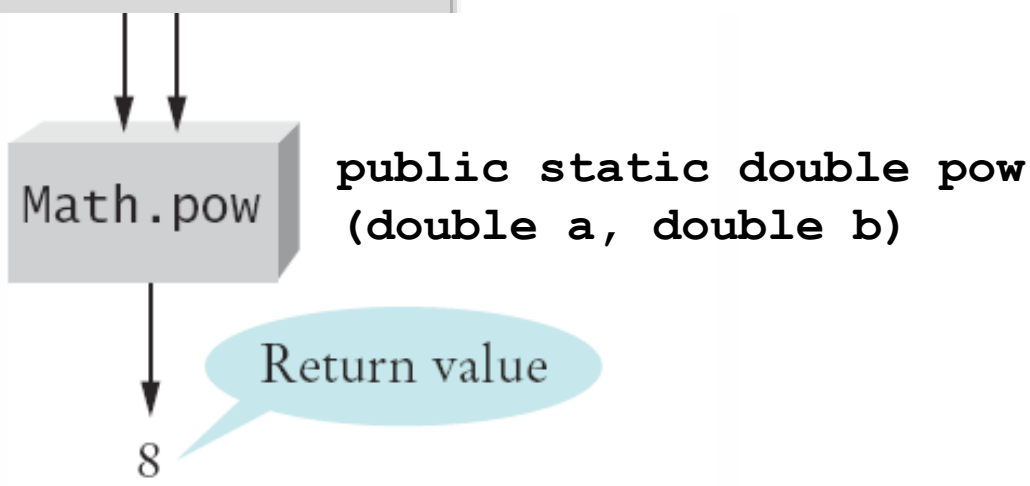



```
public static void main(String[] args) {
  double z = Math.pow(2.0, 3.0);
  . . .
}
```

- **One method "calls" (or invokes) another**
  - Example: main() calls Math.pow(x, y)
  - Passes two values (2.0 and 3.0)
    - Values can be variables or literals
  - Math.pow(x, y) starts
    - Uses values (2.0, 3.0)
    - Does its job
    - Returns the answer
  - main() uses result
    - Because the method returns a value, you must **DO** something with it

Parameters and Return Values

```
public static void main(String[]
   args) {
   double z = Math.pow(2.0, 3.0);
   . . .
}
```



- **main() "passes" two actual parameters (arguments) to Math.pow**
- **Math.pow calculates and returns a value**
- **Main stores the return value to variable 'z'**

Black Box Analogy

- **A thermostat is a "black box"**
  - Set a desired temperature
  - Thermostat turns on heater/AC as required
  - How the thermostat works is irrelevant
    - How does it know the current temperature?
    - What signals/commands does it send to the heater/AC?
- **Use methods like "black boxes"**
  - Pass the method what it needs to do its job
  - Receive the answer

IMPLEMENTING METHODS

Example: Implementing Methods Calculating Cube Volume

- **Problem: Create a method that shows the calculation for the volume of a cube**
  - What does it need to do its job?
  - What does it answer with?
- **When writing this method:**
  - Pick a name for the method (`cubeVolume`)
  - Give a data type and name for each parameter variable (`double sideLength`)
  - Specify the type of the return value (`double`)
  - Add modifiers such as `public static`

When declaring a method, you provide a name for the method, a name and type for each parameter, and a type for the result

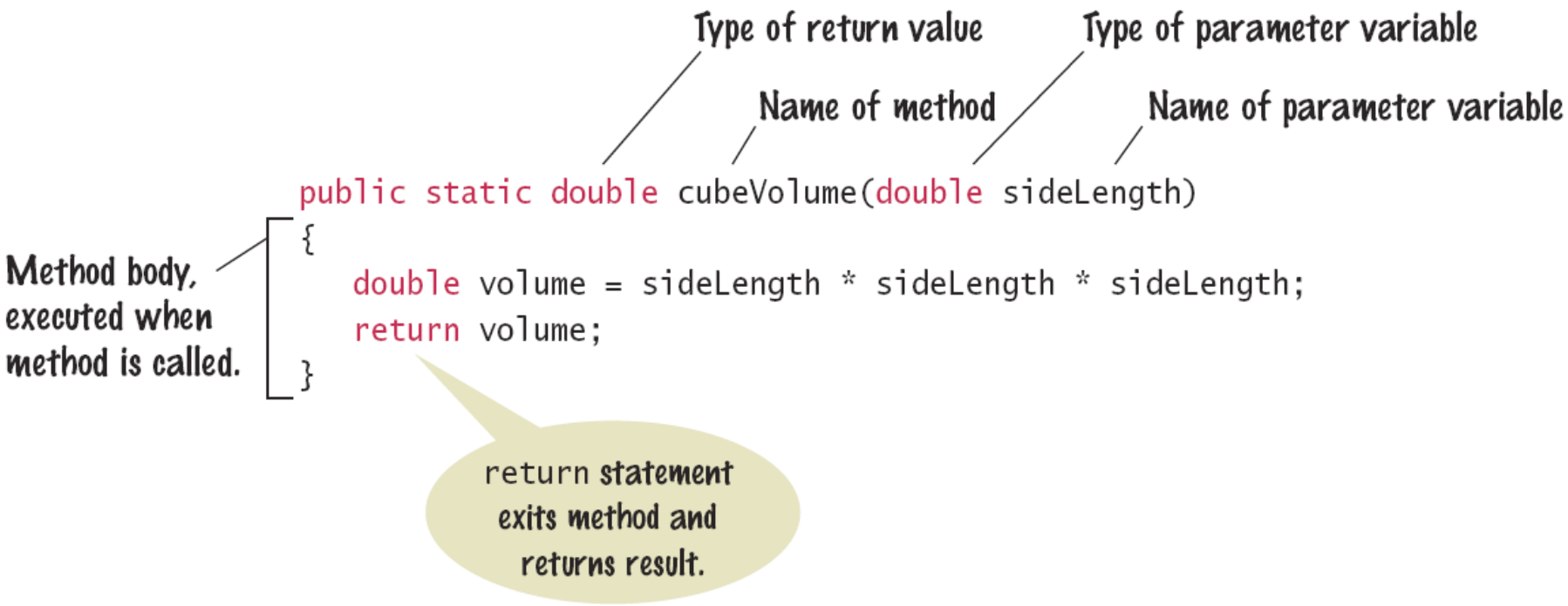
```
public static double cubeVolume(double sideLength)
```

Example: Implementing Methods Calculating Cube Volume (Cont'd)

- **Writing the body of the method**
  - The body is surrounded by curly braces { }
  - The body contains the variable declarations and statements that are executed when the method is called
  - The body returns the calculated answer

```
public static double cubeVolume(double sideLength) {
  double volume = sideLength * sideLength * sideLength;
  return volume;
}
```

Syntax: Method Declaration



Example: Implementing Methods Calculating Cube Volume (Cont'd)

```
1  import javax.swing.JOptionPane;
2
3  /**
4     This program computes the volume of two cubes.
5  */
6
7  public class Cube {
8     public static void main(String[] args) {
9        final double CUBE_SIDE_LENGTH1 = 2.0;
10       final double CUBE_SIDE_LENGTH2 = 10.0;
11
12       double cubeVolume1 = cubeVolume(CUBE_SIDE_LENGTH1);
13       double cubeVolume2 = cubeVolume(CUBE_SIDE_LENGTH2);
14       printVolume(CUBE_SIDE_LENGTH1, cubeVolume1);
15       printVolume(CUBE_SIDE_LENGTH2, cubeVolume2);
16    }
17
18    /**
19       Computes the volume of a cube
20       @param sideLength the side length of the cube
21       @return the volume
22    */
23    public static double cubeVolume(double sideLength) {
24       // Could also use double volume = Math.pow(sideLength, 3);
25       double volume = sideLength * sideLength * sideLength;
26       return volume;
27    }
28
29    /**
30       Prints the volume of a cube based on a side length
31       @param sideLength the side length of the cube
32       @param volume the volume of the cube
33    */
34    public static void printVolume(double sideLength, double volume) {
35       JOptionPane.showMessageDialog(null,
36          "A cube with side length " + sideLength + " has volume " + volume);
37    }
38 }
```

Method Comments

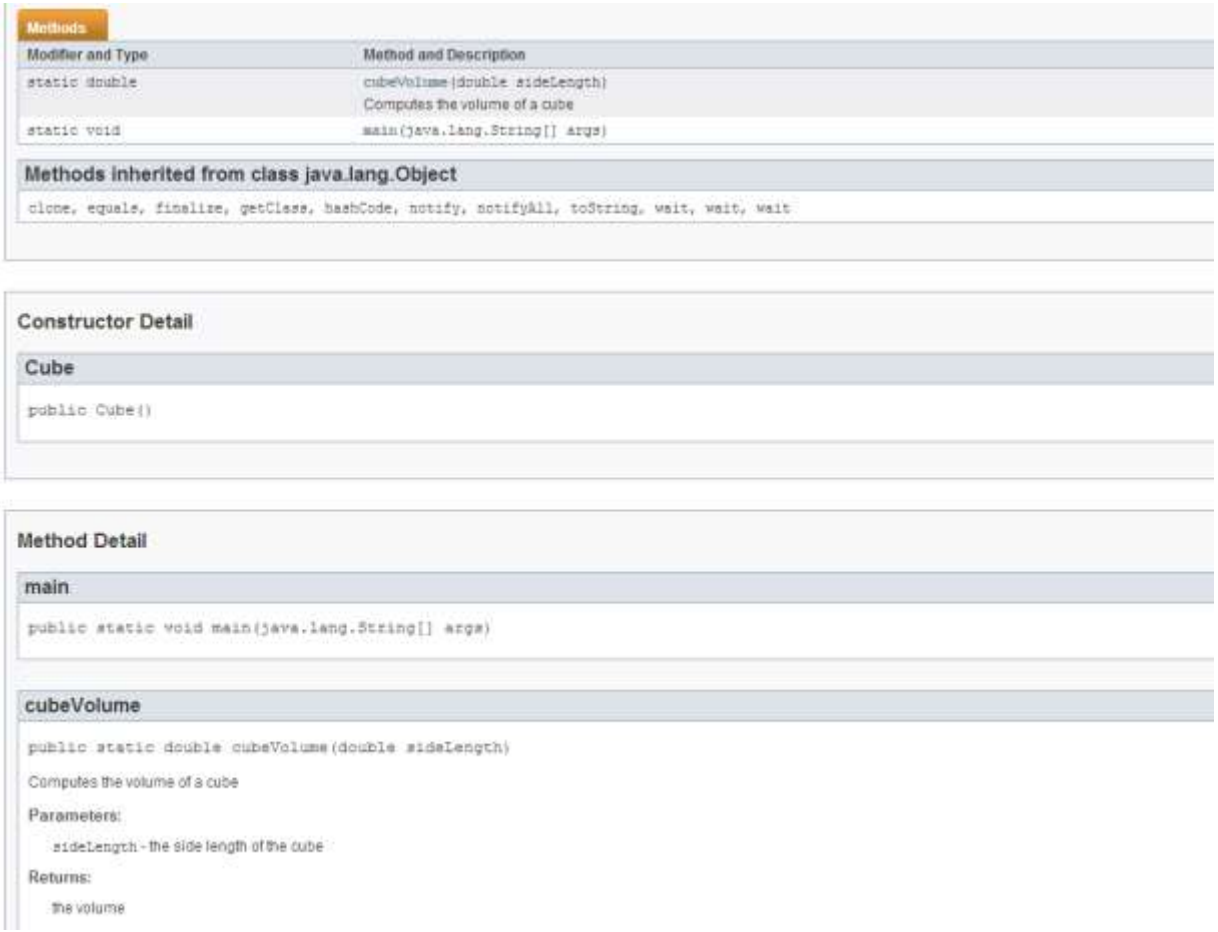
- **Start with /** (Javadoc multiline)**
  - Note the purpose of the method
  - Describe each parameter
  - Describe the return value
- **End with */**

```
/**
   Computes the volume of a cube
   @param sideLength the side length of the cube
   @return the volume
*/
public static double cubeVolume(double sideLength) {

/**
   Prints the volume of a cube based on a side length
   @param sideLength the side length of the cube
   @param volume the volume of the cube
*/
public static void printVolume(double sideLength, double volume) {
```

Method Comments (Cont'd)

- **Code documentation can be generated automatically using Javadoc**



Methods

Modifier and Type	Method and Description
static double	cubeVolume(double sideLength) Computes the volume of a cube
static void	main(java.lang.String[] args)

Methods inherited from class java.lang.Object

clone, equals, finalize, getClass, hashCode, notify, notifyAll, toString, wait, wait, wait

Constructor Detail

Cube

```
public Cube()
```

Method Detail

main

```
public static void main(java.lang.String[] args)
```

cubeVolume

```
public static double cubeVolume(double sideLength)
```

Computes the volume of a cube

Parameters:

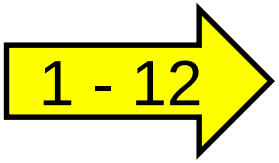
sideLength - the side length of the cube

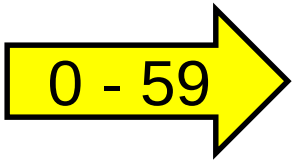
Returns:

the volume

Converting Repeated Code into Methods

- **Keep an eye out for repetitive code**
  - May have different values, but same logic



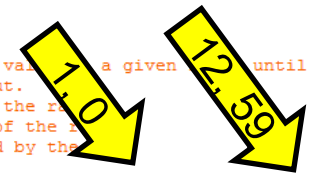



```
int hours;
do {
   hours = Integer.parseInt(JOptionPane.showInputDialog(
      "Enter a value between 1 and 12"));
}
while (hours < 1 || hours > 12);

int minutes;
do {
   minutes = Integer.parseInt(JOptionPane.showInputDialog(
      "Enter a value between 0 and 59"));
}
while (minutes < 0 || minutes > 59);
```

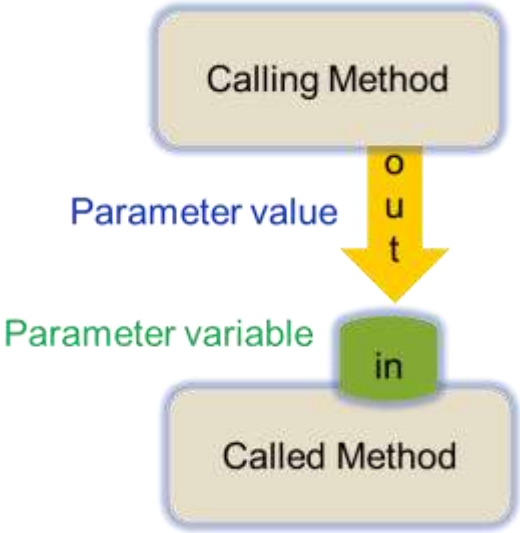
Writing a "Parameterized" Method

```
1  import javax.swing.JOptionPane;
2
3  /**
4     This program is a stub to simulate the running of a clock.
5  */
6  public class Clock {
7     public static void main(String[] args) {
8        final int MIN_HOURS = 1;
9        final int MAX_HOURS = 12;
10       final int MIN_MINUTES = 0;
11       final int MAX_MINUTES = 59;
12
13       int hours = readValueBetween(MIN_HOURS, MAX_HOURS);
14       int minutes = readValueBetween(MIN_MINUTES, MAX_MINUTES);
15
16       printTime(hours, minutes);
17    }
18
19    /**
20     Prompts a user to enter a va      a given      until the
21     user provides a valid input.
22     @param low the low end of the r
23     @param high the high end of the
24     @return the value provided by th
25    */
26    public static int readValueBetween(int low, int high) {
27       int number;
28
29       do {
30          try {
31             number = Integer.parseInt(JOptionPane.showInputDialog(
32                "Enter a value between " + low + " and " + high));
33          }
34          catch (NumberFormatException e) {
35             number = low - 1;
36          }
37       }
38       while (number < low || number > high);
39
40       return number;
41    }
42
43    /**
44     Prints a time based on a provided number of hours and minutes
45     @param hours the hours portion of the time
46     @param minutes the minutes portion of the time
47    */
48    public static void printTime(int hours, int minutes) {
49       JOptionPane.showMessageDialog(null, "The time is " + hours
50          + ":" + (minutes < 10 ? "0" : "") + minutes);
51    }
52 }
```



Parameter Passing

- **Parameter variables hold the parameter values supplied in the method call**
  - They must both be the same data type
- **The parameter value may be:**
  - The contents of a variable or constant
  - A literal
  - Often described as an "argument"
- **The parameter variable is:**
  - Named in the called method declaration
  - Used as a variable inside the called method
  - Often described as a "formal parameter"

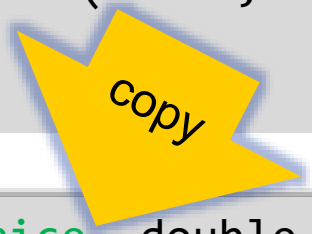


Common Error: Trying to Modify Parameters

- A copy of parameter values are passed
- Called method (addTax) can modify local copy (`price`), but not in original calling method (`total`)

```
public static void main(String[] args) {
  double total = 10.00;
  double tax = addTax(total, 7.5);
}
```

10.00




```
public static double addTax(double price, double rate) {
  double tax = price * (rate / 100);
  price = price + tax; // Has no effect outside the method
  return tax;
}
```

10.75

Methods with Return Values

- **Methods can (optionally) return one value**
  - The return type is specified in the method declaration
  - The return statement does two things:
    - The method terminates immediately
    - The return value is returned to the calling method

Return type (use "void" when a value is not returned)

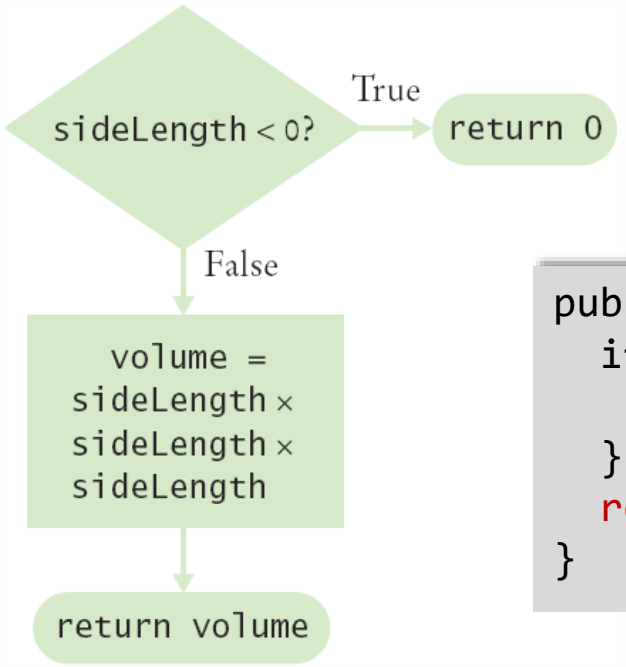
```
public static double addTax(double price, double rate)
{
  price += price * rate / 100;
  return price;
}
```

Return value

- The return value may be a value, a variable or a calculation
  - Type must match return type

Multiple `return` Statements

- **A method can use multiple `return` statements**
  - But, every branch must have a `return` statement
  - Not recommended
    - Consider keeping it simple. One return statement per method




```
public static double cubeVolume(double sideLength) {
  if (sideLength < 0.0) {
    return 0.0;
  }
  return sideLength * sideLength * sideLength;
}
```

Methods without Return Values

- **Methods are not required to return a value**
  - The return type of void means nothing is returned
  - No return statement is required
  - The method can generate output though

```
public static void main(String[] args) {
   starString(5);
}

public static void starString(int numStars) {
   String strStars = "";

   for (int x = 0; x < numStars; x++) {
      strStars += "*";
   }

   JOptionPane.showMessageDialog(null, strStars);
}
```

Using `return` Without a Value

- In methods with **void** return type
- The method will terminate immediately

```
public static void starString(int numStars) {
   String strStars = "";

   if (numStars == 0) {
      return; // Return immediately
   }

   for (int x = 0; x < numStars; x++) {
      strStars += "*";
   }

   JOptionPane.showMessageDialog(null, strStars);
}
```

Common Error: Missing `return` Statement

- **All conditions must be handled**
- **In this case, x could equal 0**
  - No return statement for this condition; Compiler will throw a syntax error if any branch has no return statement
  - Another reason to have one return value per method!

```
// Before
public static int sign(int x) {
  if (x < 0) { return -1; }
  else if (x > 0) { return 1; }
  // Error: missing return value if x equals 0
}

// After
public static int sign(int x) {
  int returnValue = 0;
  if (x < 0) { returnValue = -1; }
  else if (x > 0) { returnValue = 1; }
  else { returnValue = 0; }
  return returnValue;
}
```

Tip: Keep Methods Short!

- **Don't be lazy!**
  - New programmers often make their methods really long
  - Each method should do "one thing and one thing only"
  - Rule of thumb: If your method contains more than one screen of code, chances are, it may be appropriate to break it up into more methods
- **The effort is worth it!**
  - It may take a little longer to break up the code into methods, but the efficiency of debugging and maintenance more than make up for it

SAMPLE PROBLEM: DETERMINING SCHOLARSHIP ELIGIBILITY

Problem Scenario

- **Create a program to request the user's name, age and gender. If the user is a male, over the age of 18 issue a message that in order to apply for a scholarship, he must register for the selective service.**

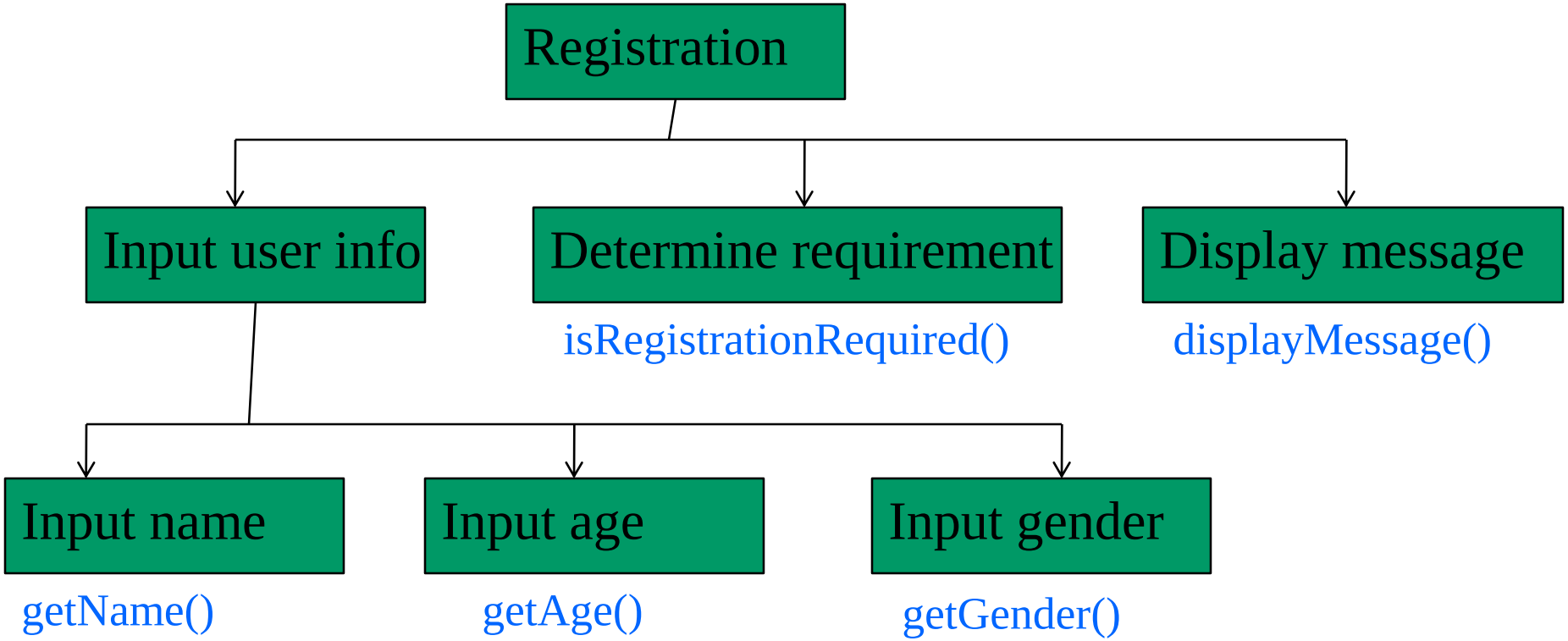
What do we do now?

Step 1: Define the Problem

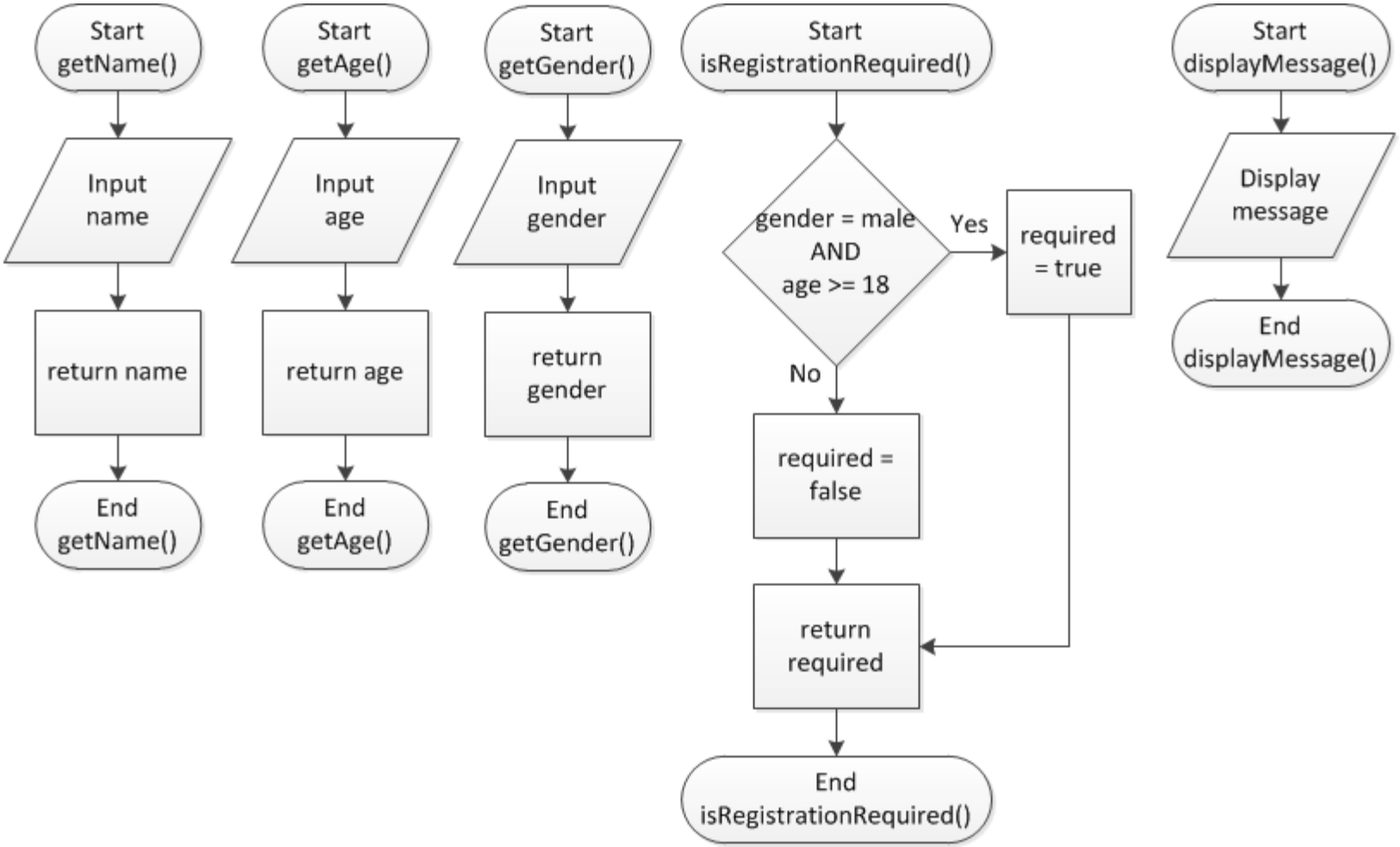
IPO Chart

Input	Processing	Output
• name • age • gender	• Prompt for name • Read name • Validate name • Prompt for age • Read age • Validate age • Prompt for gender • Read gender • Validate gender • Determine if selective service registration is required • Print registration message	• registration message

Step 2/3: Group the Activities & Create a Hierarchy Chart



Step 2/3: Group the Activities & Create a Hierarchy Chart (Cont'd)
Flowchart EACH Method



Step 4: Establish the mainline logic

BEGIN Registration

getName

getAge

getGender

isRegistrationRequired

displayMessage

END

Step 5: Create the Solution Algorithm
Must include pseudocode for **EACH** module

```
BEGIN Registration
1    getName
2    getAge
3    getGender
4    isRegistrationRequired
5    displayMessage
END

BEGIN getName
6    Prompt user for name
7    Read name
END

BEGIN getAge
8    REPEAT
9       Prompt user for age
10      Read age
11      IF age <= 0 OR age >= 130 THEN
           Display error message
        ENDIF
     UNTIL age > 0 AND age < 130
END
```

```
BEGIN getGender
     REPEAT
12      Prompt user for gender
13      Read gender
14      IF gender != 'M' AND gender != 'F' THEN
           Display error message
        ENDIF
     UNTIL gender = 'M' OR gender = 'F'
END

BEGIN isRegistrationRequired
15   IF gender = 'M' AND age >= 18 THEN
        SET required = true
     ELSE
        SET required = false
     ENDIF
END

BEGIN displayMessage
16   IF required = true
        Print '<Name>, selective service registration is
required'
     ELSE
        Print '<Name>, selective service registration is
not required'
     ENDIF
END
```

Step 6: Desk Check the Algorithm

Input Test Data

	Set 1	Set 2
name	Jack	Mary
age	19	20
gender	M	F

Expected Output

	Set 1	Set 2
registrationMessage	Selective service registration is required	Selective service registration is not required

Desk Checking Table

Statement Number	name	age	gender	required	registrationMessage
Set 1					
1, 6, 7	Jack				
2, 8, 9, 10, 11		19			
3, 12, 13, 14			M		
4, 15				true	
5, 16					Selective service registration is required
Set 1					
1, 6, 7	Mary				
2, 8, 9, 10, 11		20			
3, 12, 13, 14			F		
4, 15				false	
5, 16					Selective service registration is not required

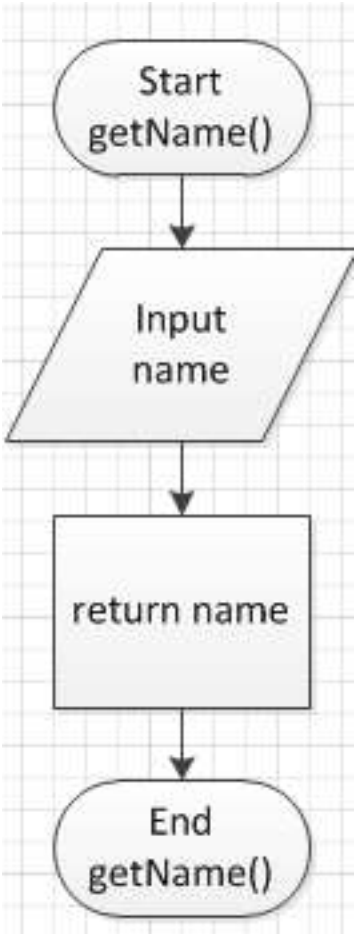
Step 7: Create the Program Determine Return Types and Create Stubs

```
import javax.swing.JOptionPane;
public class Registration {
        public static void main (String[] args) {}
        public static String getName() {}
        public static int getAge() {}
        public static char getGender() {}
        public static boolean isRegistrationRequired() {}
        public static void displayMessage() {}
}
```

Step 7: Create the Program (Cont'd) Create a Rough Draft of the main() Method

```
import javax.swing.JOptionPane;
public class Registration {
        public static void main (String[] args) {
                String name = getName();
                int age = getAge();
                char gender = getGender();
                boolean registrationRequired = isRegistrationRequired();
                displayMessage();
        }
        public static String getName() {}
        public static int getAge() {}
        public static char getGender() {}
        public static boolean isRegistrationRequired() {}
        public static void displayMessage() {}
}
```

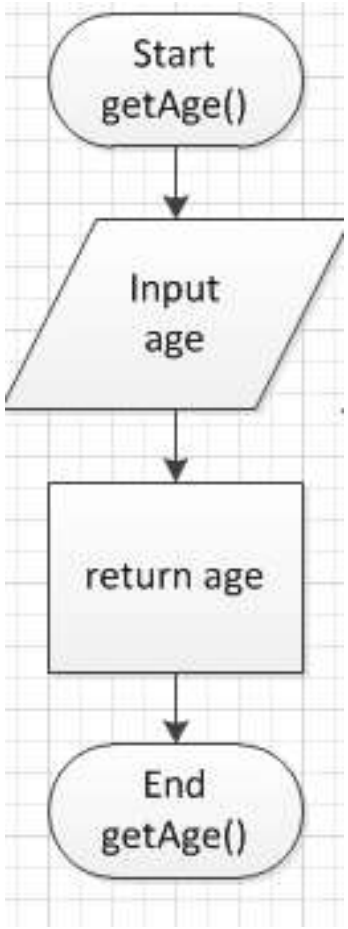
Step 7: Create the Program (Cont'd)
Work on the Methods: getName



No parameter needed

```
public static String getName() {
    String name = JOptionPane.showInputDialog("Enter name: ");
    return name;
}
```

Step 7: Create the Program (Cont'd)
Work on the Methods: getAge

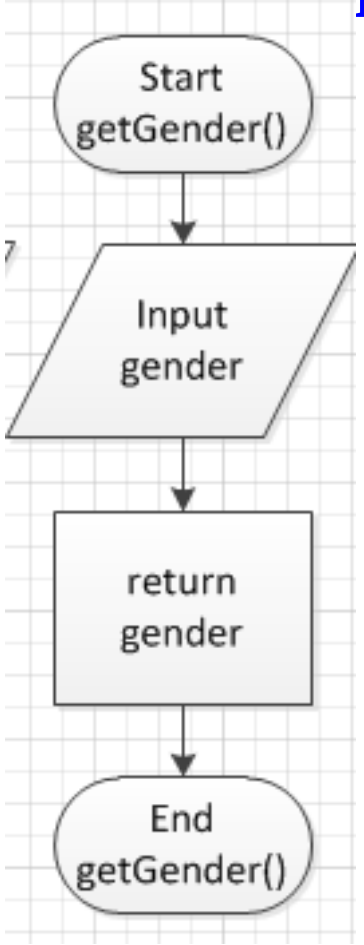


No parameter needed

```
public static int getAge() {
   int age;
   do {
      try {
         age = Integer.parseInt(JOptionPane.showInputDialog("Enter age: "));
      }
      catch (NumberFormatException e) {
         age = 0;
      }
      if (age <= 0 || age >= 130) {
         JOptionPane.showMessageDialog(null,
            "You have entered an invalid age. Please try again.");
      }
   }
   while (age <= 0 || age >= 130);

   return age;
}
```

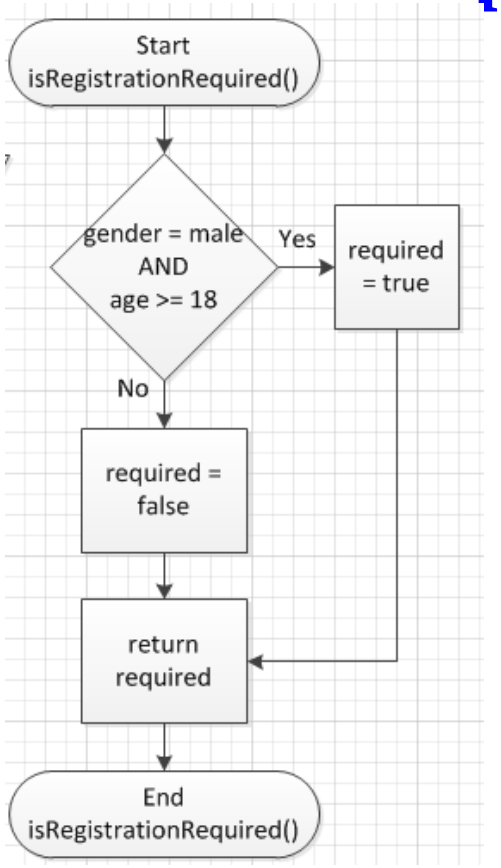
Step 7: Create the Program (Cont'd)
Work on the Methods: getGender




```
public static char getGender() {
    char gender;
    do {
        try {
            gender = JOptionPane.showInputDialog(
                "Enter gender (M or F): ").charAt(0);
        }
        catch (StringIndexOutOfBoundsException e) {
            gender = ' ';
        }
        if (gender != 'M' && gender != 'F') {
            JOptionPane.showMessageDialog(null,
                "You have entered an invalid gender. Please try again.");
        }
    }
    while (gender != 'M' && gender != 'F');
    return gender;
}
```

No parameter needed

Step 7: Create the Program (Cont'd)
Work on the Methods: isRegistrationRequired



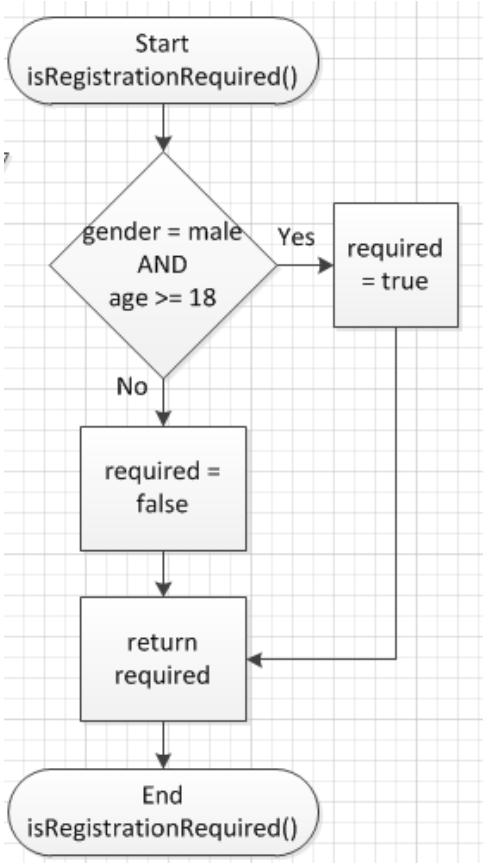
Parameters needed!
Need to know age and gender

```
public static boolean isRegistrationRequired(int age, char gender) {
   boolean required = true;

   if (gender == 'M' && age >= 18) {
      required = true;
   }
   else {
      required = false;
   }

   return required;
}
```

Step 7: Create the Program (Cont'd)
Work on the Methods: isRegistrationRequired (Cont'd)




```
boolean required = isRegistrationRequired(age, gender);
```

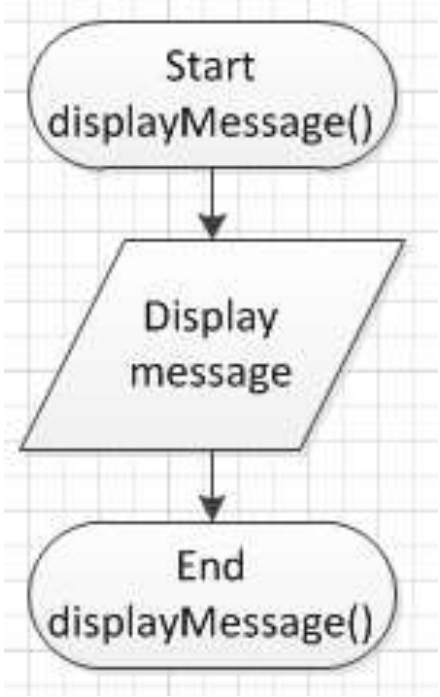
Parameters needed!
Need to know age and gender

```
public static boolean isRegistrationRequired(int age, char gender) {
   boolean required = true;

   if (gender == 'M' && age >= 18) {
      required = true;
   }
   else {
      required = false;
   }

   return required;
}
```

Step 7: Create the Program (Cont'd)
Work on the Methods: displayMessage

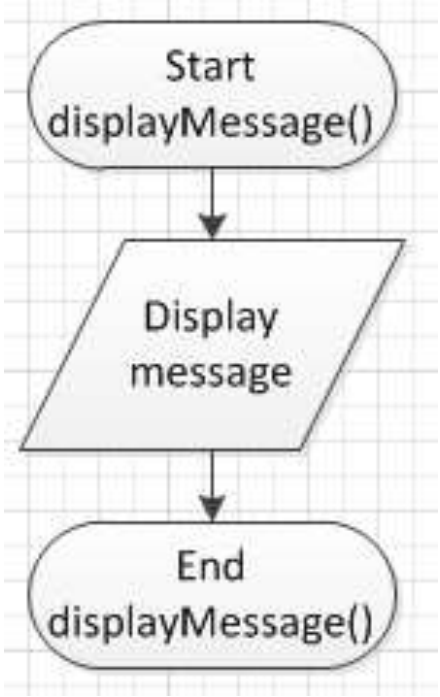


Parameters needed!
Need to know name and required

```
public static void displayMessage(String name, boolean required) {
   if (required == true) {
      JOptionPane.showMessageDialog(null, name +
         ", selective service registration is required.");
   }
   else {
      JOptionPane.showMessageDialog(null, name +
         ", selective service registration is not required.");
   }

}
```

Step 7: Create the Program (Cont'd)
Work on the Methods: displayMessage (Cont'd)




```
displayMessage(name, required);
```

Parameters needed!
Need to know name and required

```
public static void displayMessage(String name, boolean required) {
    if (required == true) {
        JOptionPane.showMessageDialog(null, name +
            ", selective service registration is required.");
    }
    else {
        JOptionPane.showMessageDialog(null, name +
            ", selective service registration is not required.");
    }

}
```

Steps 8/9: Test and Document the Code

- **Don't forget these important steps to make sure your program is correct and maintainable!**

Questions?