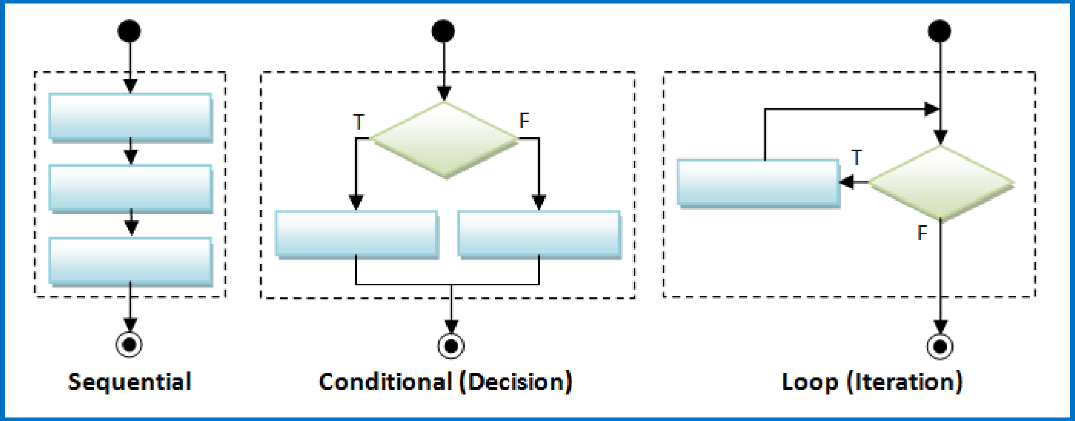


CSS 142

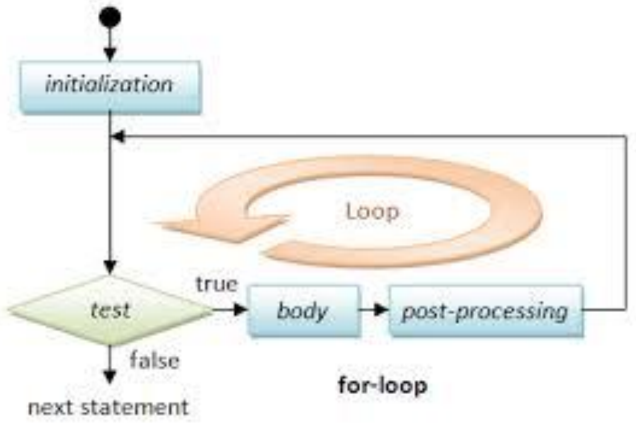
Lecture 6

Arkady Retik aretik@uw.edu

TODAY'S CONTENT

1. **HW1 & HoA2 – feedback**
2. **Midterm content and preps**
3. **Last Week recap**
   a. **Reading Quiz**
4. **Booleans**
5. **Loops**
6. **Next lecture (Wedn):**
   - **Read Savitch: 3.2, 3.3, 3.4**
   - **HoA3**
   - **HW2 is available: read and ask questions**

Follow Chapter 3 (3.1, 3.2, 3.3)
and examples



HoA2: feedback

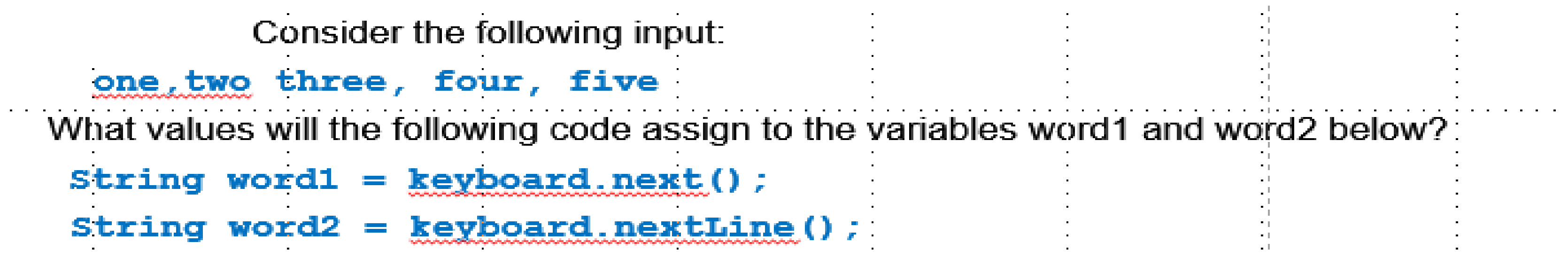
1. Most problems

   - problem 3: watch for spaces

   - problem 4 and 5: reading with a diff delimiter.

2. What if we change the last line (i.e. the way we read word2?

Demo: ideone



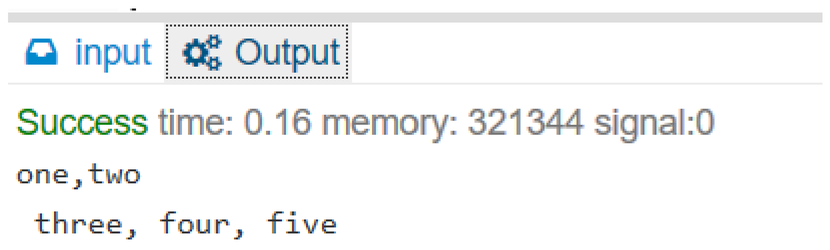
Consider the following input:

```
one,two three, four, five
```

What values will the following code assign to the variables word1 and word2 below?

```
String word1 = keyboard.next();
String word2 = keyboard.nextLine();
```

```
public static void main (String[] args) throws java.lang.Exception
{
Scanner keyboard = new Scanner (System.in);
String word1 = keyboard.next();
String word2 = keyboard.nextLine();
System.out.println (word1);
System.out.println (word2);
}
```



Success time: 0.16 memory: 321344 signal:0

```
one,two
 three, four, five
```

HW1: additional feedback

Results:

- Grader: Michael
- Miss some submissions
    - Talk to me, if you missed

See comments in the Rubric;

Reply with questions on Canvas if the comments are not clear

Midterm 1: content

- 3.5 weeks content:
  - Savitch Chapters 1,2,3
  - Lectures 1-7
  - HW1,2,3   && HoA 1,2,3 && Labs
- Logistics
  - Wed, Apr 18, 11.00 – 1.00
  - Closed books; paper based
  - Combination of Q&As, writing code; reading code and writing output
  - **Bring pencils and erasers**

Key Topics

- Assignments
- Primitives and String
- Statements
- Methods; Printing
- IO, Scanner
- Branching: IFs, switch
- Booleans
- Loops
  - while; do while; for

Preparations

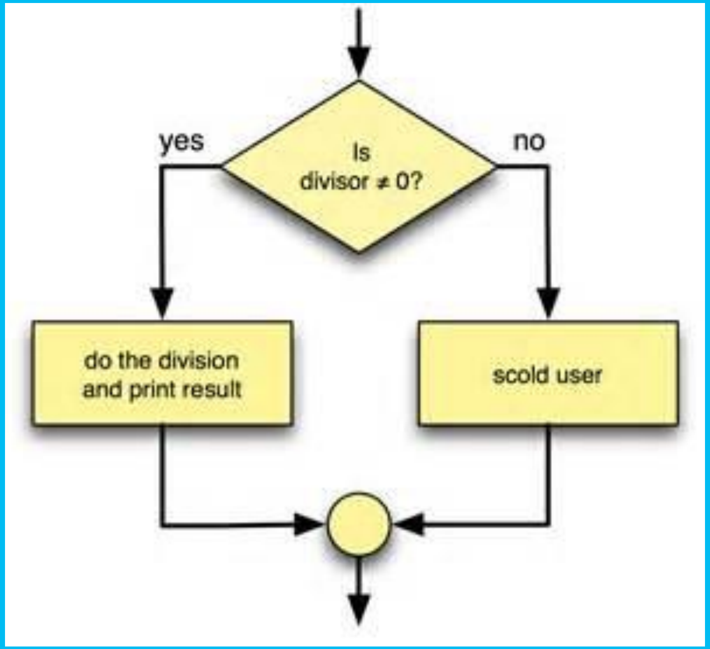
- Read, read, read
- Write, write, write
- Practice, practice, practice

individually

- Study groups
- CSS QSC help
- My Student Hours
- External Resources

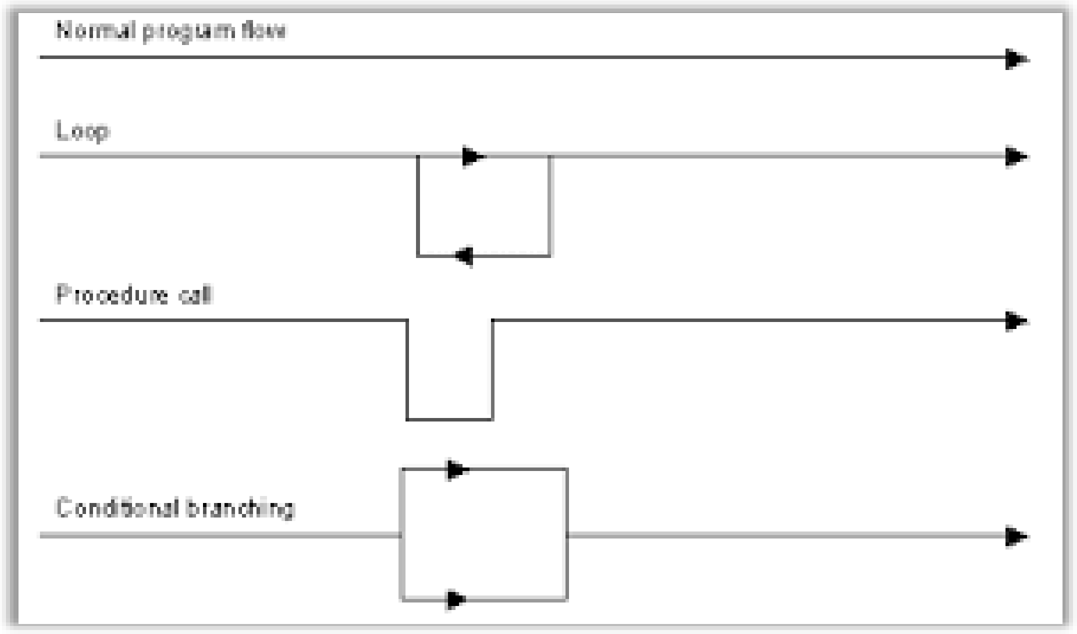
Extra-help

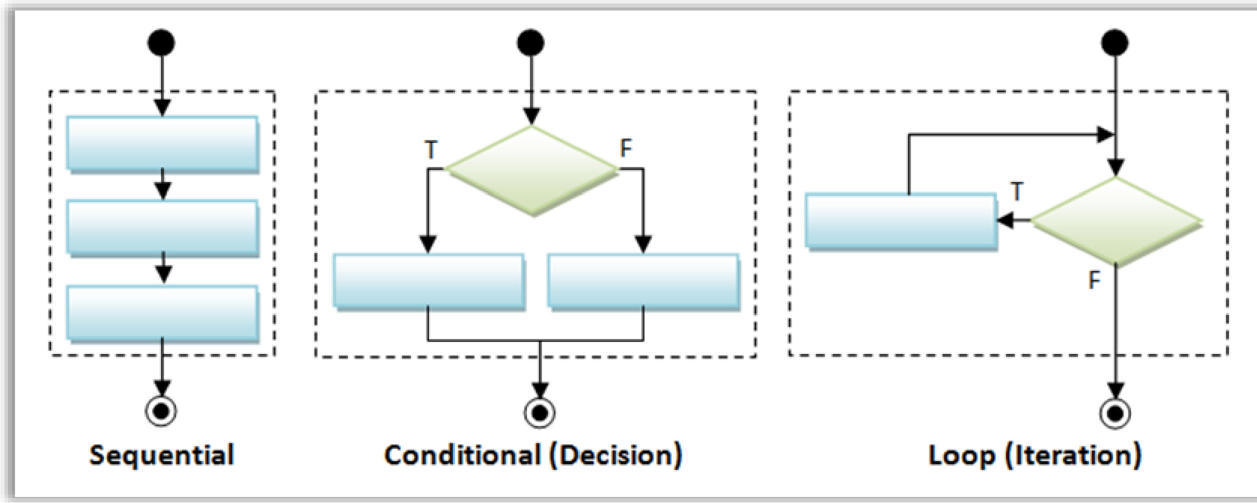
Continue: Flow of Control



Flow charts

Flow of Control

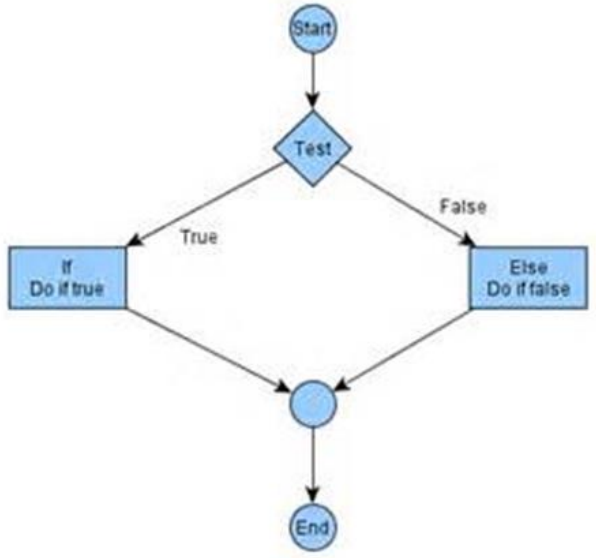




Flow charts

Compound Statements: pay attention

```
if (myScore > your Score)
{
    System.out.println("I win!");
    wager = wager + 100;
}
else
{
    System.out.println("I wish these were golf scores.");
    wager = 0;
}
```



Flow Chart for If/Else

Omitting the `else` Part

- The `else` part may be omitted to obtain what is often called an `if` statement

```
if (Boolean_Expression)
      Action_Statement;
```

Write down an example of a compound statement

  - If the `Boolean_Expression` is true, then the `Action_Statement` is executed
  - The `Action_Statement` can be a **single** or **compound** statement
  - **Otherwise**, nothing happens, and the program goes on to the next statement

```
if (weightNow > weightTarget)
{
    calorieIntake - = 700;
    exerciseTime + = 45;
}
```

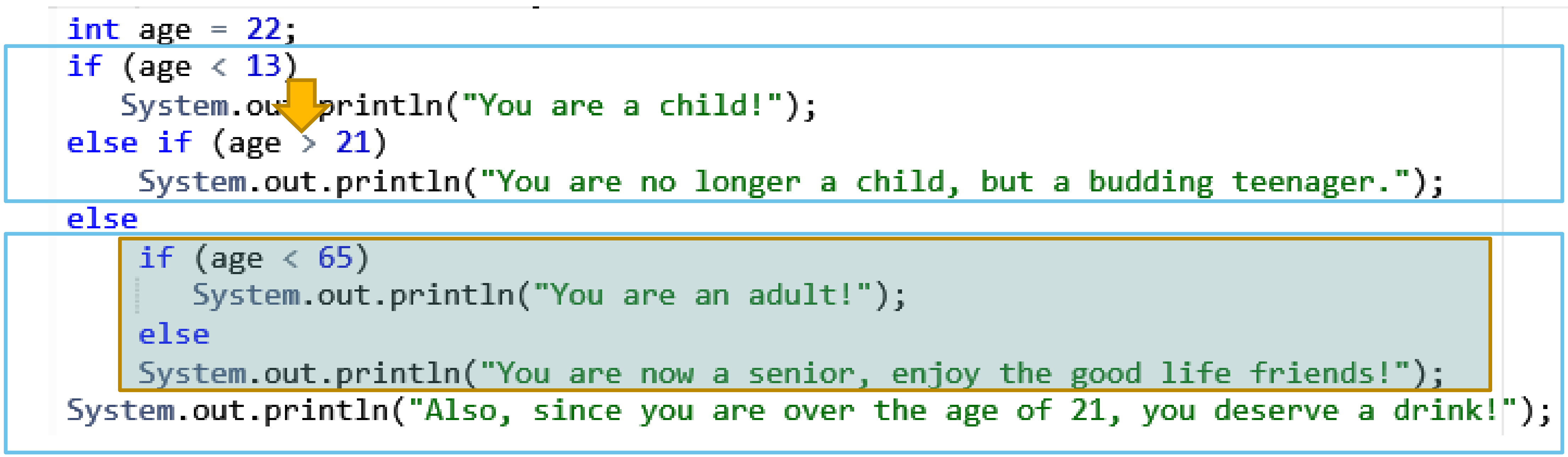
Nested Statements

- `if-else` statements and `if` statements both contain smaller statements within them
  - For example, single or compound statements
- In fact, any statement at all can be used as a subpart of an `if-else` or `if` statement, including another `if-else` or `if` statement
  - Each level of a nested `if-else` or `if` **should be indented further** than the previous level
  - ***Exception:*** *multiway* `if-else` statements

```
int a=1, b=3, c=4;
{
if (a==b)
    {
    a=a+1;
    if (a>c)
            System.out.println ("start");
    else
        System.out.println ("wait");
    }
 else
    {
     a=a -1;
     if (a<c)
            System.out.println ("stop");
     else
            System.out.println ("wait");
     }
}
```

Nested Statements: class

Write down the output this program will produce

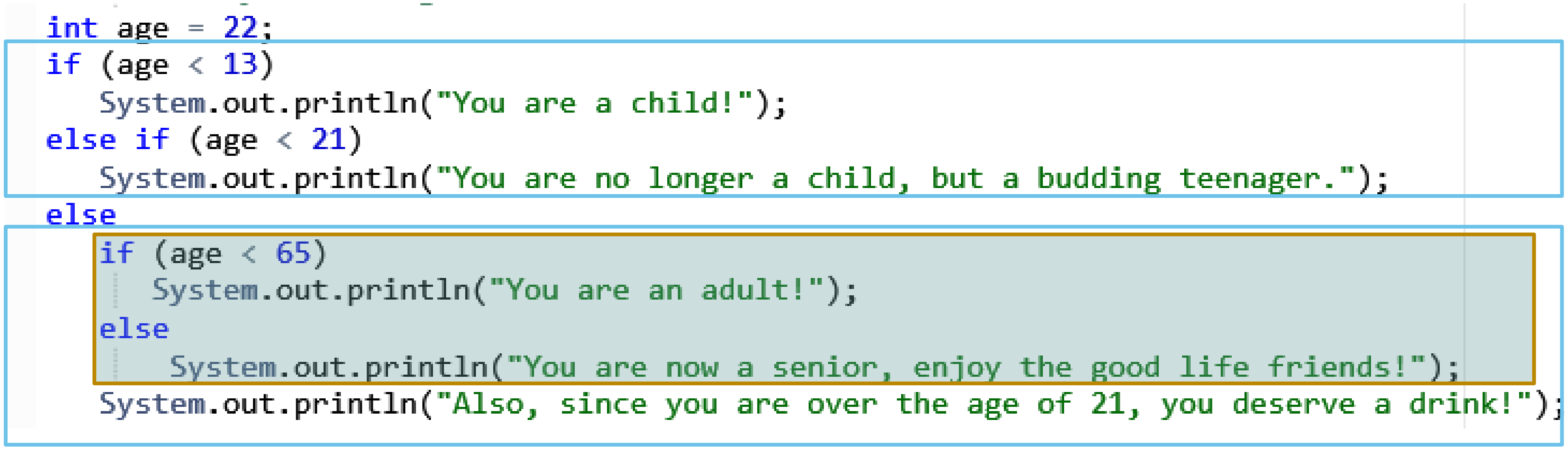


```
int age = 22;
if (age < 13)
   System.out.println("You are a child!");
else if (age > 21)
    System.out.println("You are no longer a child, but a budding teenager.");
else
    if (age < 65)
       System.out.println("You are an adult!");
    else
    System.out.println("You are now a senior, enjoy the good life friends!");
System.out.println("Also, since you are over the age of 21, you deserve a drink!");
```

Will making (age < 21) fix the problem we discovered in class?

Nested Statements: example

Write down the output this program will produce

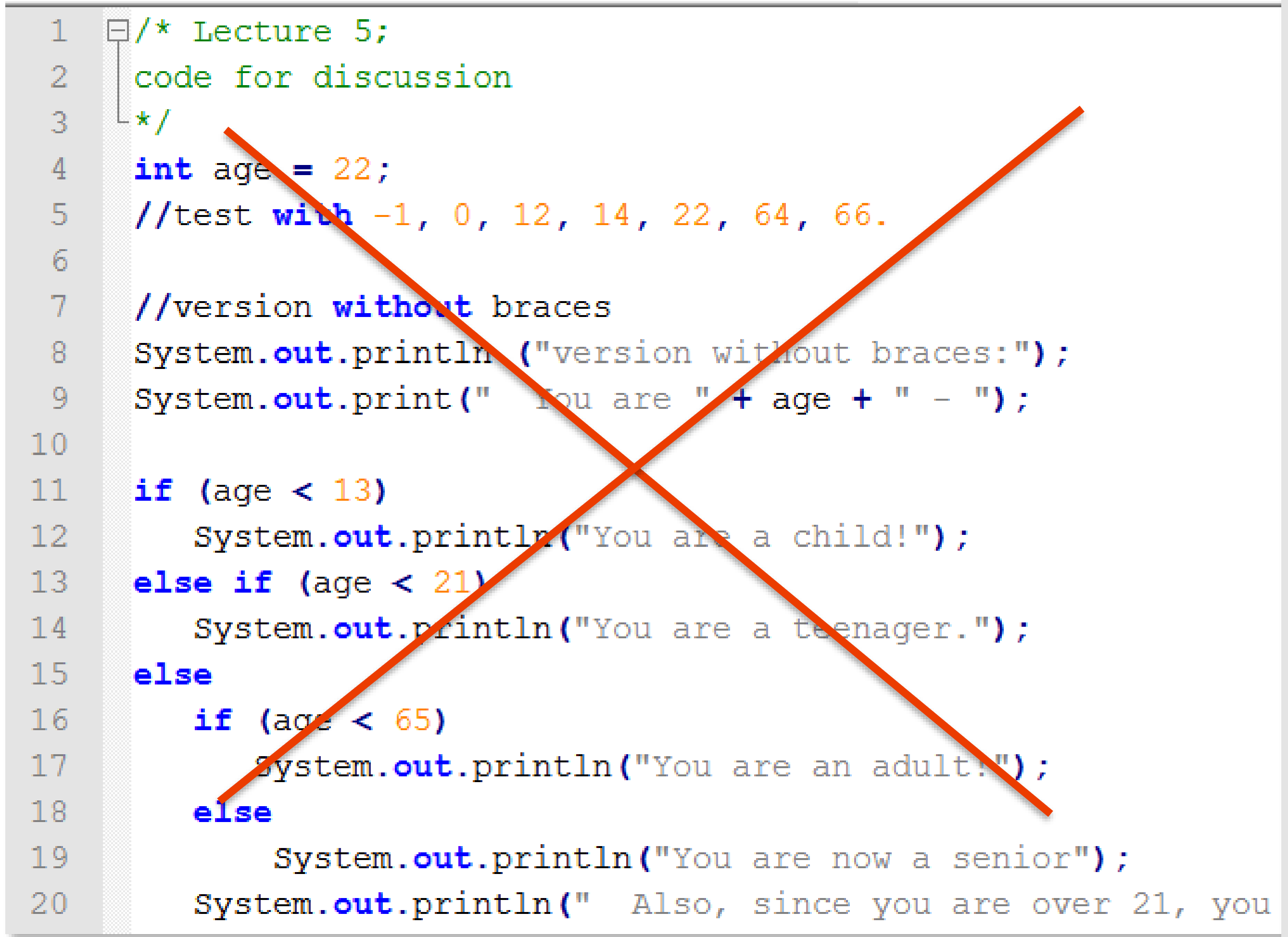


```
int age = 22;
if (age < 13)
   System.out.println("You are a child!");
else if (age < 21)
   System.out.println("You are no longer a child, but a budding teenager.");
else
   if (age < 65)
      System.out.println("You are an adult!");
   else
       System.out.println("You are now a senior, enjoy the good life friends!");
   System.out.println("Also, since you are over the age of 21, you deserve a drink!");
```

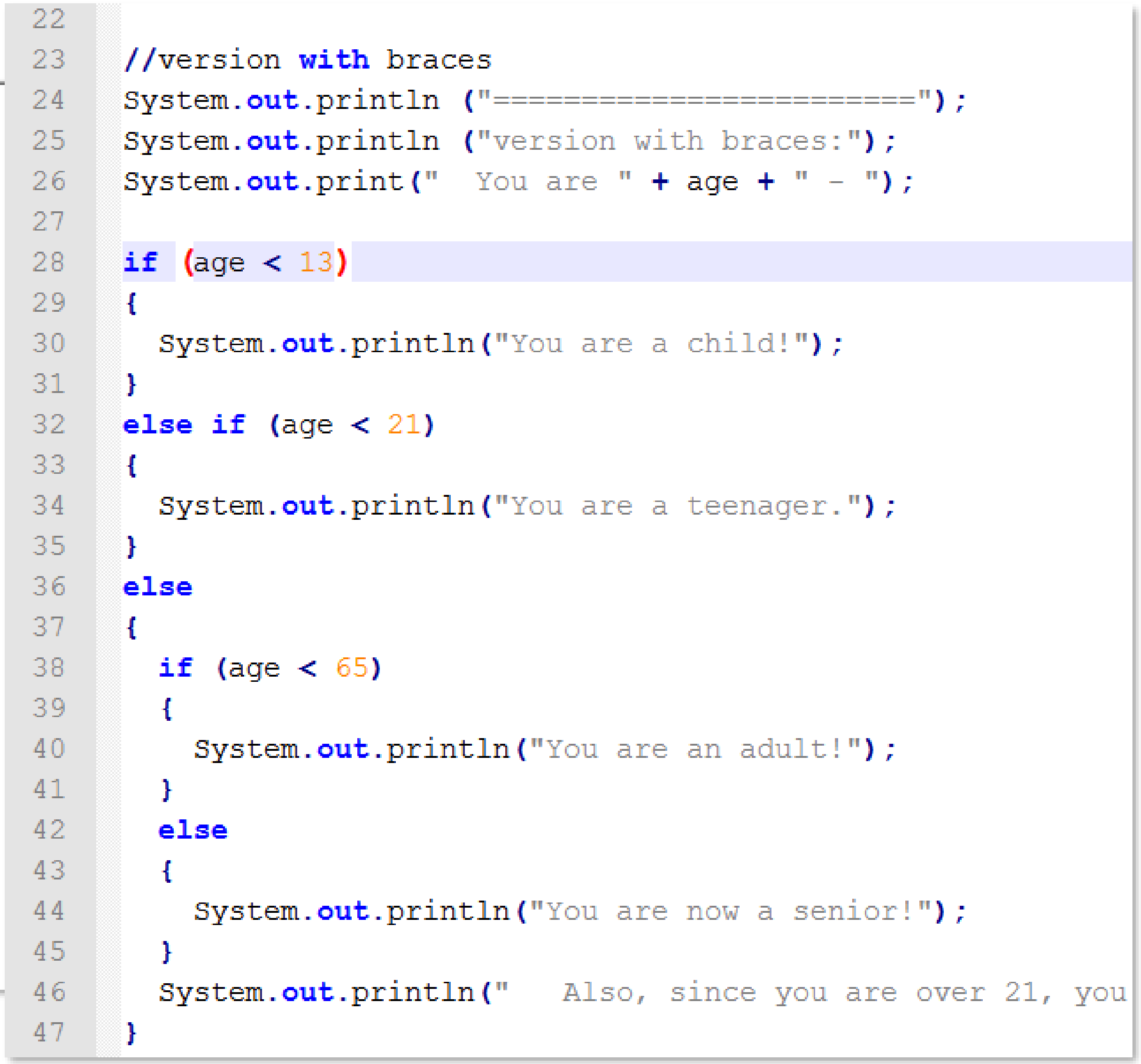
Write down the output this program will produce for
age = 12;
age = 67;

Check it out and see if there is anything wrong……

Compare:

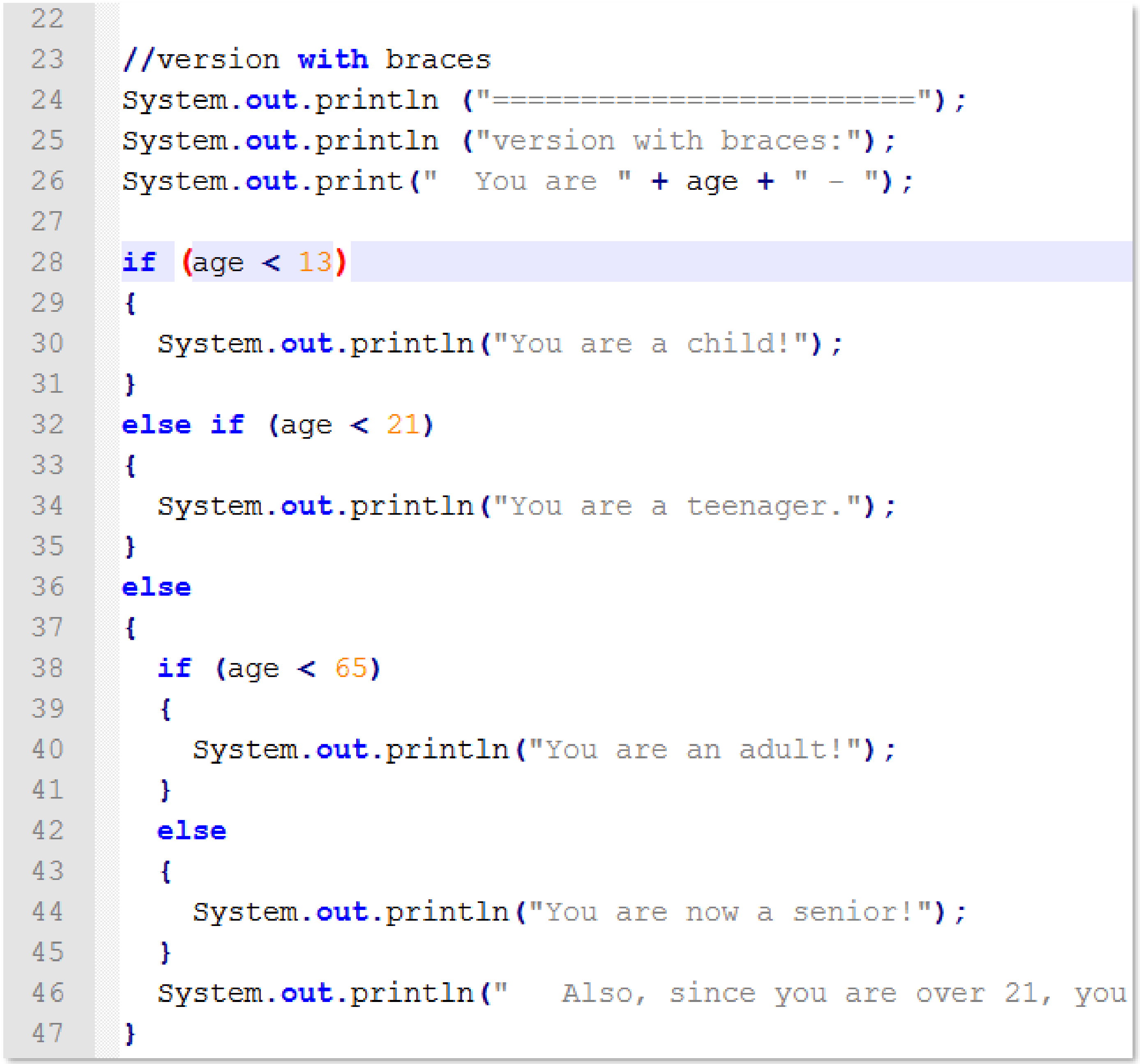


```
/* Lecture 5;
code for discussion
*/
int age = 22;
//test with -1, 0, 12, 14, 22, 64, 66.

//version without braces
System.out.println ("version without braces:");
System.out.print("  You are " + age + " - ");

if (age < 13)
   System.out.println("You are a child!");
else if (age < 21)
   System.out.println("You are a teenager.");
else
   if (age < 65)
      System.out.println("You are an adult!");
   else
       System.out.println("You are now a senior");
   System.out.println("  Also, since you are over 21, you
```



```

//version with braces
System.out.println ("========================");
System.out.println ("version with braces:");
System.out.print("  You are " + age + " - ");

if (age < 13)
{
  System.out.println("You are a child!");
}
else if (age < 21)
{
  System.out.println("You are a teenager.");
}
else
{
  if (age < 65)
  {
    System.out.println("You are an adult!");
  }
  else
  {
    System.out.println("You are now a senior!");
  }
  System.out.println("  Also, since you are over 21, you
}
```

More points…:



```
//version with braces
System.out.println ("========================");
System.out.println ("version with braces:");
System.out.print("  You are " + age + " - ");

if (age < 13)
{
  System.out.println("You are a child!");
}
else if (age < 21)
{
  System.out.println("You are a teenager.");
}
else
{
  if (age < 65)
  {
    System.out.println("You are an adult!");
  }
  else
  {
    System.out.println("You are now a senior!");
  }
  System.out.println("   Also, since you are over 21, you
}
```

Rule of thumb:
Use either > or <,
i.e. all conditions go TopDown or BottonUp,
i.e. age, grades

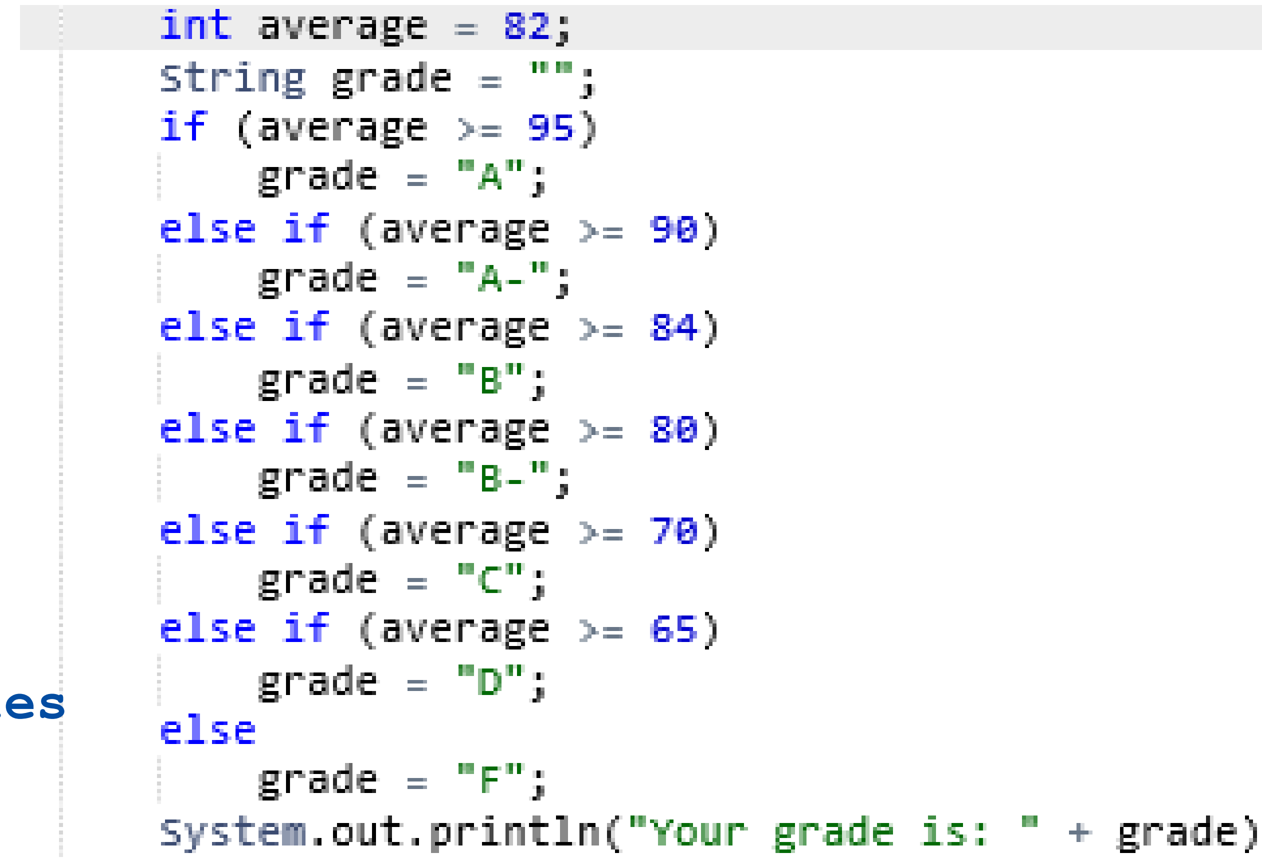
Why we start age from Bottom and grades from Up?

Multiway `if-else` Statements

- The multiway **`if-else`** statement is simply a normal **`if-else`** statement that **nests** another **`if-else`** statement at every **`else`** branch
    - It is indented differently from other nested statements
    - All of the **`Boolean_Expressions`** are aligned with one another, and their corresponding actions are also aligned with one another
    - The **`Boolean_Expressions`** are evaluated in order **until one that evaluates to** **`true`** is found
    - The final **`else`** is optional

Multiway **if-else** Statement

```
if (Boolean_Expression)
   Statement_1
else if (Boolean_Expression)
   Statement_2
     ..............................................
else if (Boolean_Expression_n)
   Statement_n
else
  Statement_For_All_Other_Possibilities
```



```
int average = 82;
String grade = "";
if (average >= 95)
    grade = "A";
else if (average >= 90)
    grade = "A-";
else if (average >= 84)
    grade = "B";
else if (average >= 80)
    grade = "B-";
else if (average >= 70)
    grade = "C";
else if (average >= 65)
    grade = "D";
else
    grade = "F";
System.out.println("Your grade is: " + grade)
```

Multiway `if-else` Statement

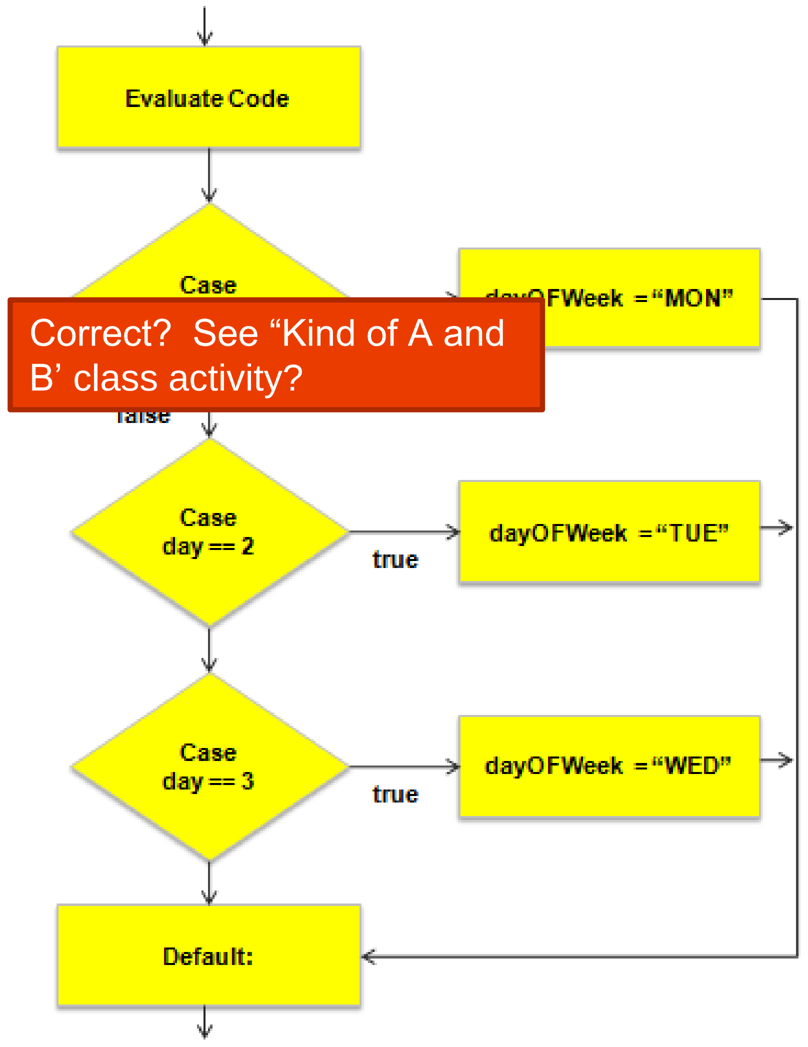
```
if ( condition_1 )
{
   //statements to execute when condition_1 is true
}
else if ( condition_2 )
{
   //statements to execute when condition_2 is true
}
//more else if blocks as necessary

else if ( last_condition )
{
   //statements to execute when last_condition is true
}
else
{
   //statements to execute when all conditions are false
}
```

else if is two words

The else branch is optional

The `switch` Statement

- The `switch` statement is the only other kind of Java statement that implements ***multiway*** branching
  - When a `switch` statement is evaluated, **one** of the branches is executed
  - The choice of which branch to execute is determined by a *controlling expression* enclosed in parentheses after the keyword `switch`
    - The controlling expression must evaluate to a `char`, `int`, `short`, `byte`, `or String*`



Correct? See "Kind of A and B' class activity?

** The String type is only allowed in Java 7 and higher (as rightly mentioned by Austen)*

The `switch` Statement

- Each branch statement in a `switch` statement starts with the reserved word `case`, followed by a *constant* called a *case label*, followed by a colon, and then a sequence of statements
  - Each case label must be of the **same type** as the controlling expression
  - Case labels need not be listed in order or span a complete interval, but each one may **appear only once**
  - Each sequence of statements **may be followed** by a `break` statement (`break;`)

The `switch` Statement

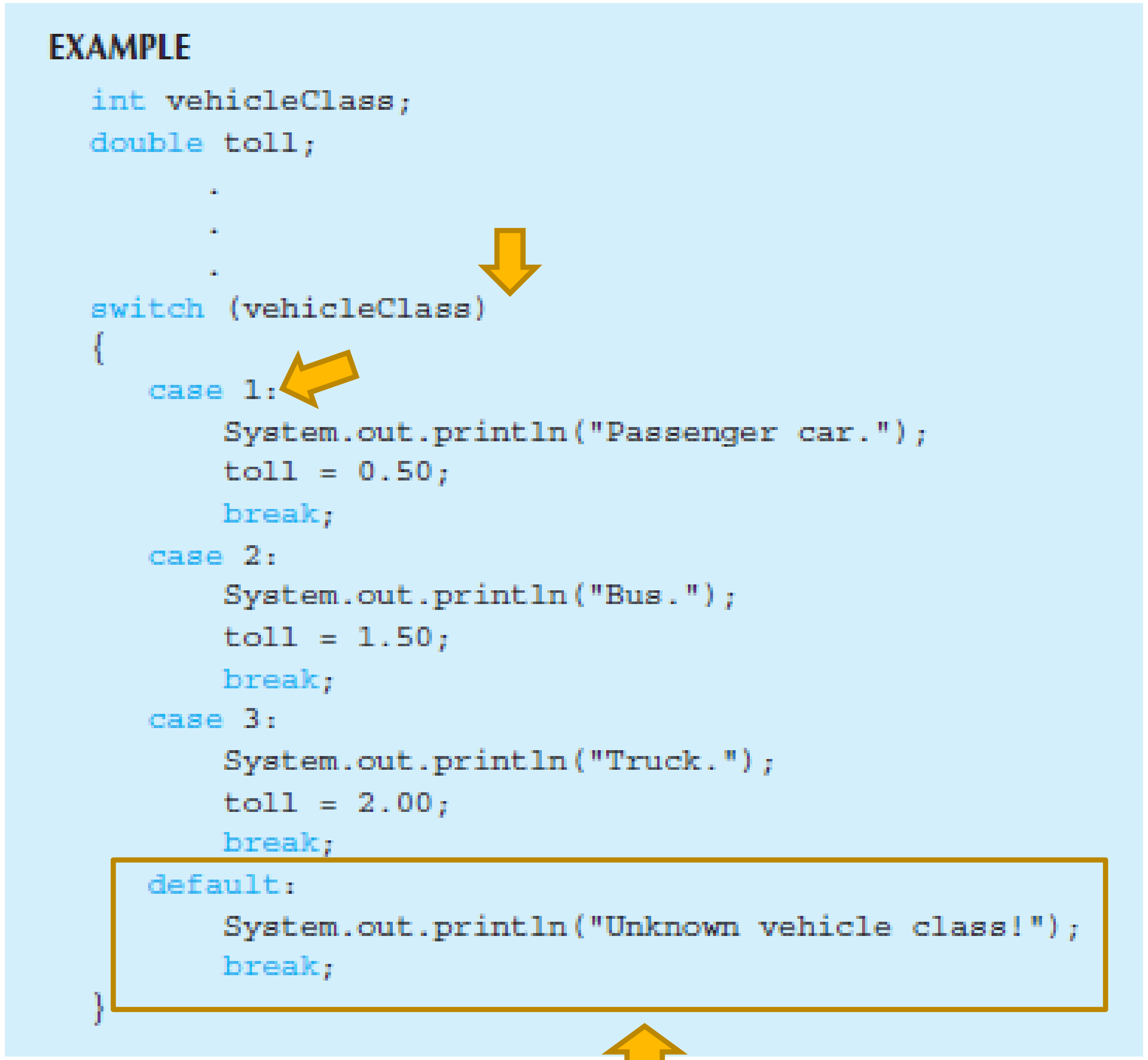
- There can also be a section labeled `default`:
  - The `default` section is **optional,** and is usually last
  - Even if the case labels cover all possible outcomes in a given `switch` statement, it is still a good practice to include a `default` section
    - It can be used to output an error message, for example
- When the controlling expression is evaluated, the code for the case label whose value matches the controlling expression is executed
  - If no case label matches, then the only statements executed are those following the `default` label (if there is one)

The `switch` Statement

- The `switch` statement **ends** when it executes a `break` statement, or when the end of the `switch` statement is reached
  - When the computer executes the statements after a case label, it continues until a `break` statement is reached
  - If the `break` statement is omitted, then after executing the code for one case, the computer will go on to execute the code for the next case
  - If the `break` statement is omitted inadvertently, the compiler will not issue an error message

The `switch` Statement

```
switch (Controlling_Expression)
{
  case Case_Label_1:
           Statement_Sequence_1
           break;
  case Case_Label_2:
           Statement_Sequence_2
           break;

              . . .

  case Case_Label_n:
           Statement_Sequence_n
           break;
  default:
           Default_Statement Sequence
           break;
}
```



EXAMPLE

```
int vehicleClass;
double toll;
      .
      .
      .
switch (vehicleClass)
{
   case 1:
       System.out.println("Passenger car.");
       toll = 0.50;
       break;
   case 2:
       System.out.println("Bus.");
       toll = 1.50;
       break;
   case 3:
       System.out.println("Truck.");
       toll = 2.00;
       break;
   default:
       System.out.println("Unknown vehicle class!");
       break;
}
```

optional

Comparing if/else with switch

Assume that you are writing a simple calculator program that read:

lhs (left hand) integer, operator (string) and rhs (right hand) integer line by line

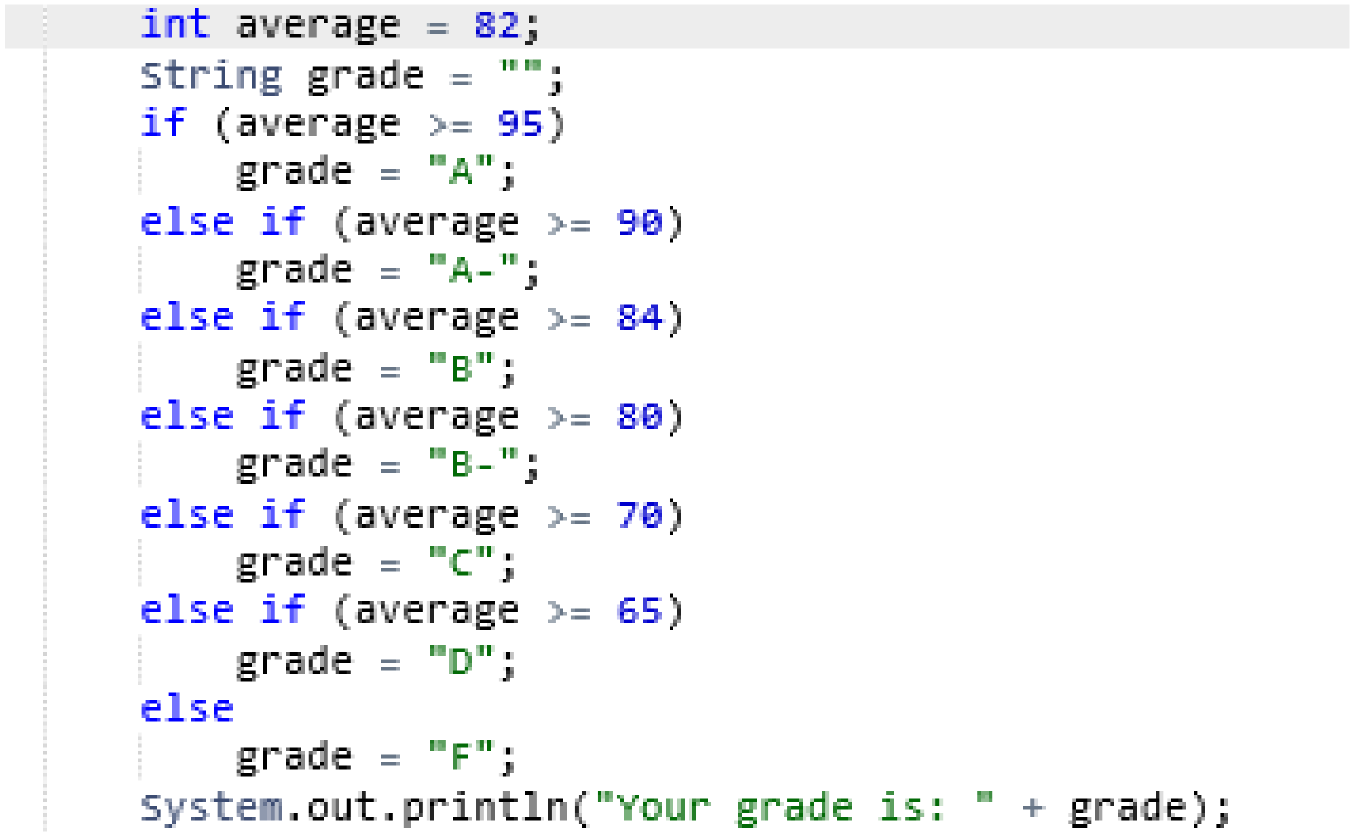
Output:

System.out.println(“Result is “ + lhs);

```
if (operation == '+'){
        lhs = lhs + rhs;
}
else if (operation == '-'){
        lhs = lhs - rhs;
}
else if (operation == '*'){
        lhs = lhs * rhs;
}
else if (operation == '/'){
        lhs = lhs / rhs;
}
```

```
switch(op)
{
        case '+' :  lhs += rhs;
                break;
        case '/' :  lhs /= rhs;
                break;
        case '*' :  lhs *= rhs;
                break;
        case '-' :  lhs *= rhs;
                break;
}
```

Rewrite the multiway if statement using **switch** Statement

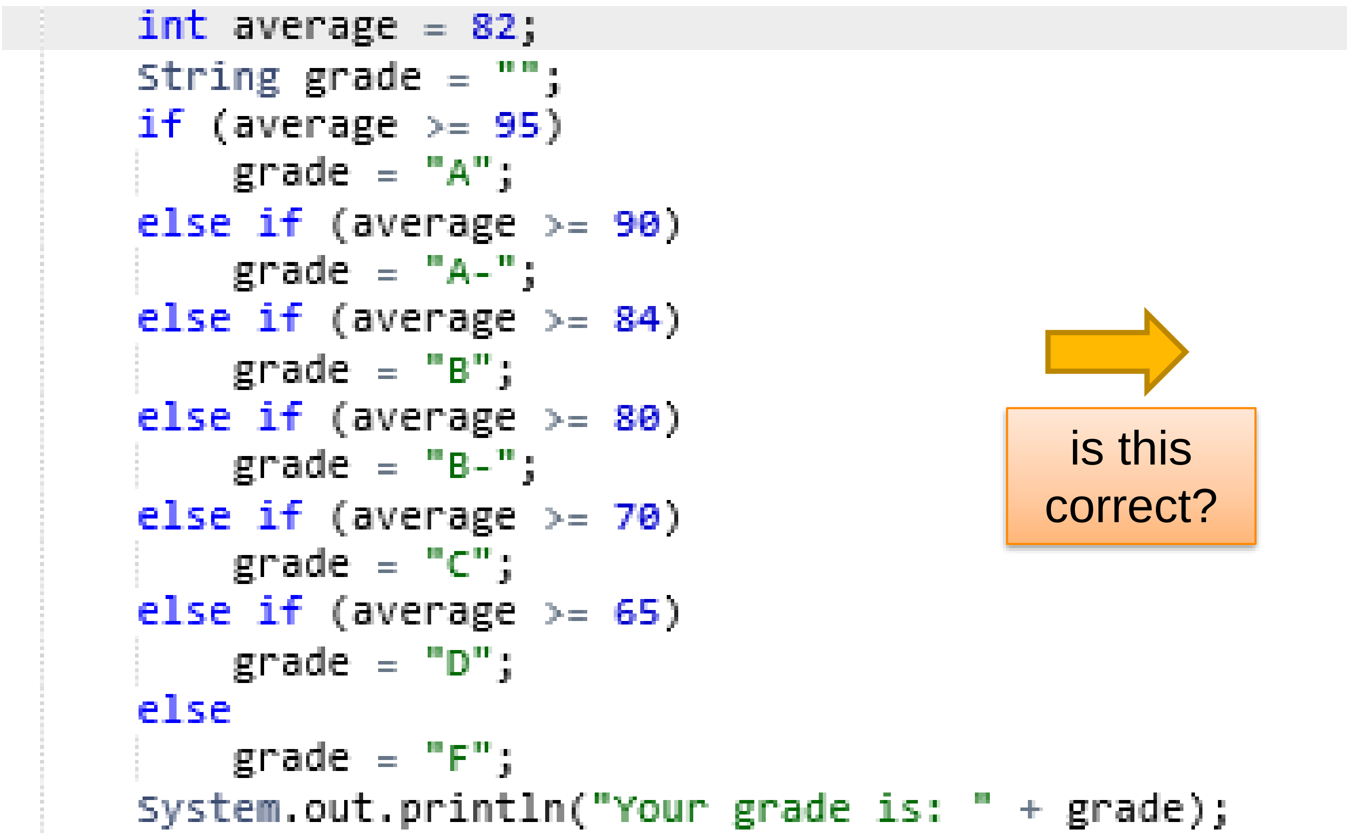


```
int average = 82;
String grade = "";
if (average >= 95)
    grade = "A";
else if (average >= 90)
    grade = "A-";
else if (average >= 84)
    grade = "B";
else if (average >= 80)
    grade = "B-";
else if (average >= 70)
    grade = "C";
else if (average >= 65)
    grade = "D";
else
    grade = "F";
System.out.println("Your grade is: " + grade);
```

Do only A, A- and B,
below this is F

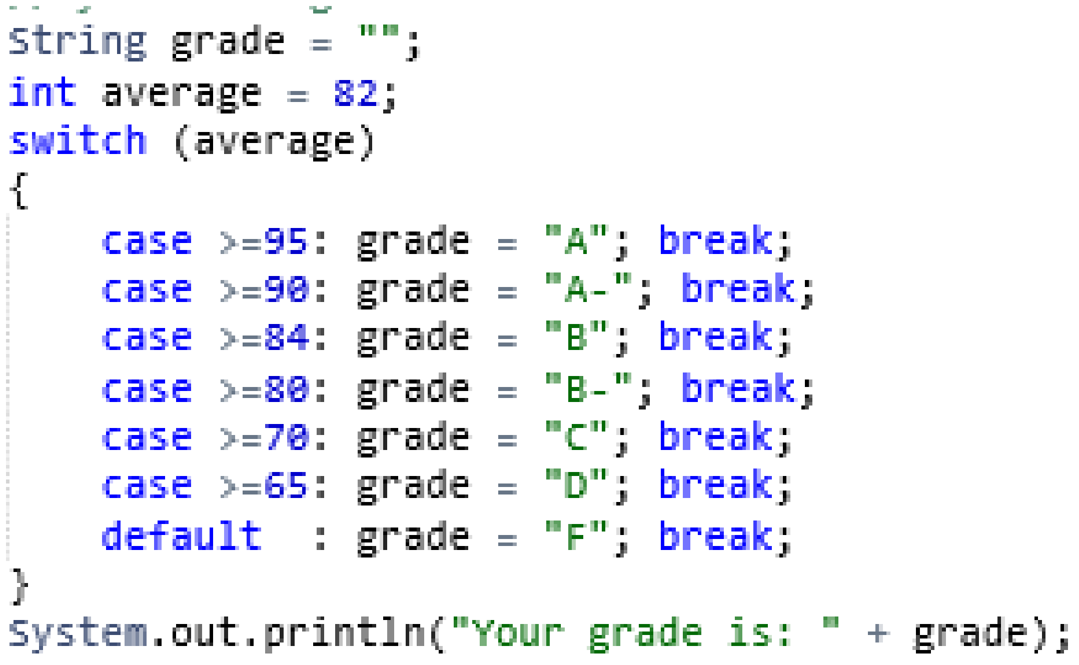
3 minutes

Rewrite the multiway if statement using **switch** Statement?

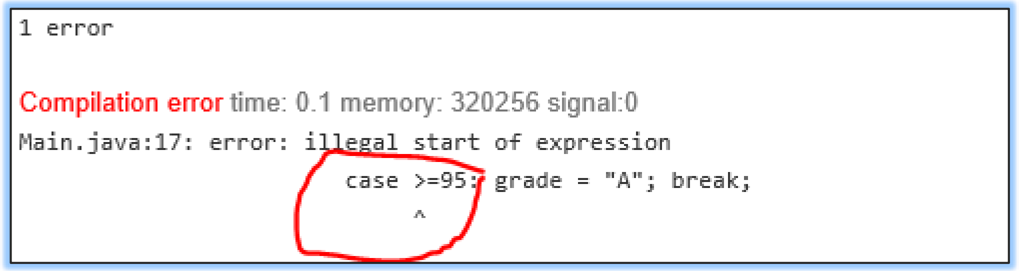


```java
int average = 82;
String grade = "";
if (average >= 95)
    grade = "A";
else if (average >= 90)
    grade = "A-";
else if (average >= 84)
    grade = "B";
else if (average >= 80)
    grade = "B-";
else if (average >= 70)
    grade = "C";
else if (average >= 65)
    grade = "D";
else
    grade = "F";
System.out.println("Your grade is: " + grade);
```

is this correct?



```java
String grade = "";
int average = 82;
switch (average)
{
    case >=95: grade = "A"; break;
    case >=90: grade = "A-"; break;
    case >=84: grade = "B"; break;
    case >=80: grade = "B-"; break;
    case >=70: grade = "C"; break;
    case >=65: grade = "D"; break;
    default  : grade = "F"; break;
}
System.out.println("Your grade is: " + grade);
```



```
1 error

Compilation error time: 0.1 memory: 320256 signal:0
Main.java:17: error: illegal start of expression
        case >=95: grade = "A"; break;
             ^
```

possible solution?

```
//Rewrite the if statement above using a switch statement
switch(grade)
{
    case 100:
    case 99:
    case 98:
    case 97:
    case 96:
    case 95:
    case 94:
    case 93:
    case 92:
    case 91:
    case 90:
        System.out.println("A");
        break;
    case 89:
    case 88:
    case 87:
    case 86:
    case 85:
    case 84:
        System.out.println("B");
        break;
```

Will this work?

Now you can see why we need both if-else and switch statements

Quizz Time

Answers:
Close to your chest

Flow of Control: testing reading

True/False

- An if-else statement chooses between two alternative statements based on the value of a Boolean expression.
    - True
- You may omit the else part of an if-else statement if no alternative action is required.
    - True
- In a switch statement, the choice of which branch to execute is determined by an expression given in parentheses after the keyword switch.
    - True
- In a switch statement, the default case is always executed.
    - False
- Not including the break statements within a switch statement results in a syntax error.
    - False

Flow of Control: testing reading

Multiple Choice

- A compound statement is enclosed between:
    1. [ ]
    2. { }
    3. ( )
    4. < >
        - 2
- A multi-way if-else statement
    1. allows you to choose one course of action.
    2. always executes the else statement.
    3. allows you to choose among alternative courses of action.
    4. executes all Boolean conditions that evaluate to true.
        - 3
- The controlling expression for a switch statement includes all of the following types except:
    1. char
    2. int
    3. byte
    4. double
        - 4

Multiple Choice

- A multi-way if-else statement
  1. allows you to choose one course of action.
  2. always executes the else statement.
  3. allows you to choose among alternative courses of action.
  4. executes all Boolean conditions that evaluate to true.

     - 3

- The controlling expression for a switch statement includes all of the following types except:
  1. char
  2. int
  3. byte
  4. Double
  5. String

     - 4

1. What output will be produced by the following code?

```
public class SelectionStatements
{
        public static void main(String[] args)
        {
                int number = 24;
                if (number % 2 == 0)
                        System.out.print("The condition evaluated to true!");
                else
                        System.out.print("The condition evaluated to false!");
        }
}
```

A : The condition evaluated to true!

2. What would be the output of the code in #1 if number was originally initialized to 25?

A : The condition evaluated to false!

- To compare two strings lexicographically the String method ____________ should be used.
  1. equals
  2. equalsIgnoreCase
  3. compareTo
  4. ==

  A 3

- When using a compound Boolean expression joined by an **&&** in an **if** statement:
  1. Both expressions must evaluate to true for the statement to execute.
  2. The first expression must evaluate to true and the second expression must evaluate to false for the statement to execute.
  3. The first expression must evaluate to false and the second expression must evaluate to true for the statement to execute.
  4. Both expressions must evaluate to false for the statement to execute.

  A 1

- The OR operator in Java is represented by:
  1. !
  2. &&
  3. | |
  4. None of the above

  A 3

- The negation operator in Java is represented by:
    1. !
    2. &&
    3. | |
    4. None of the above

    A 1

- The ____________ operator has the highest precedence.
    1. *
    2. dot
    3. + =
    4. decrement

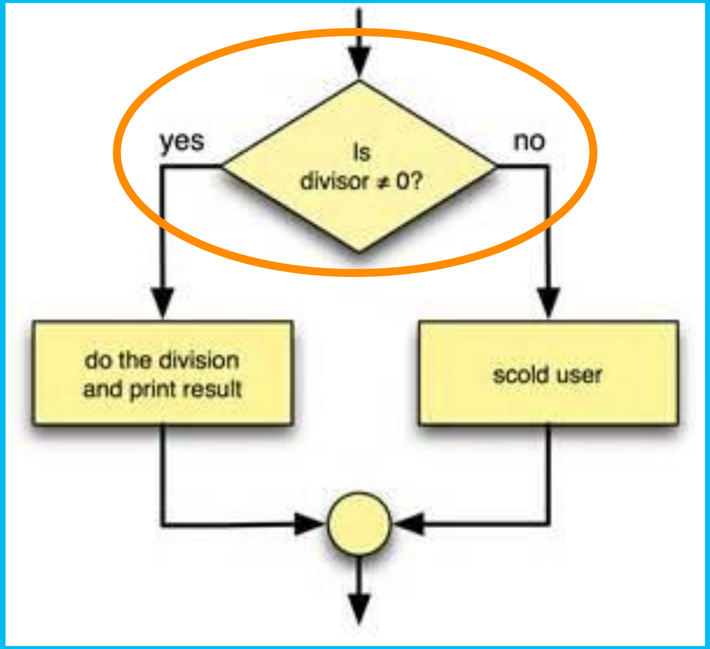
    A 2

- The association of operands with operators is called ______________.
    1. assignment
    2. precedence
    3. binding
    4. lexicographic ordering

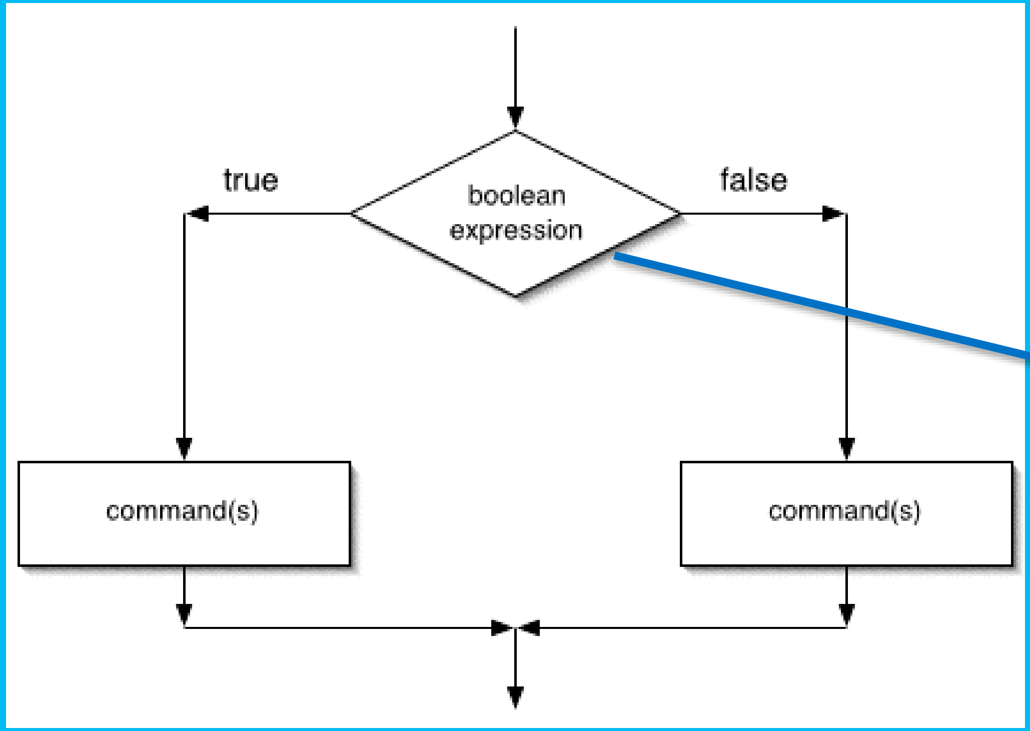
    A 3

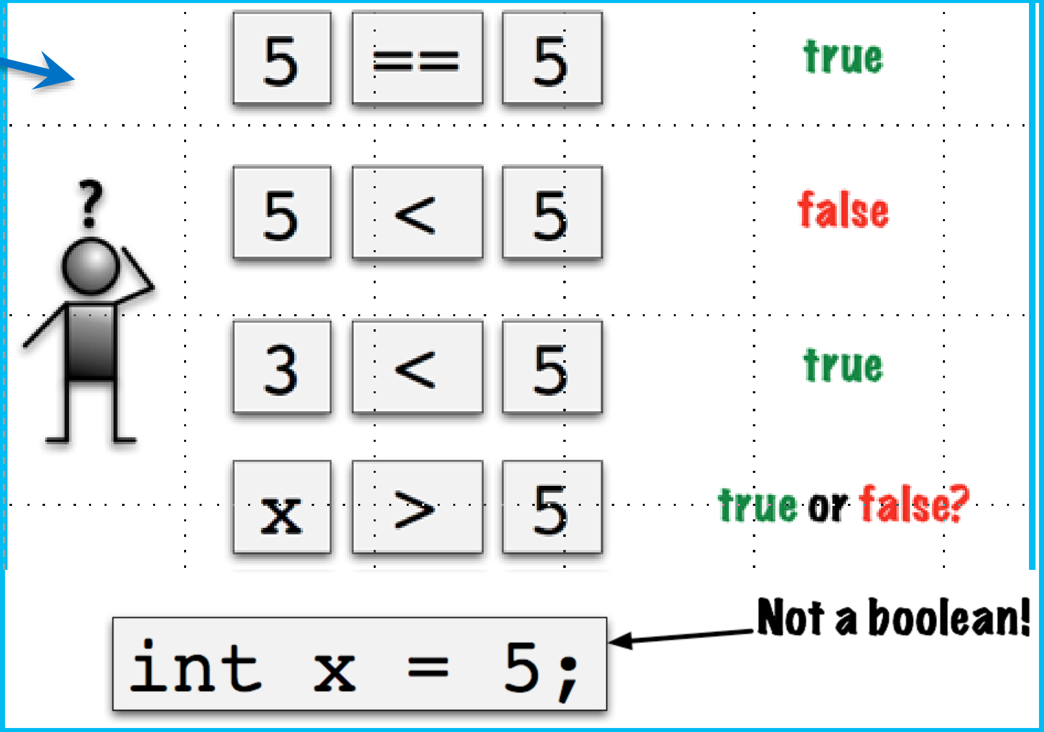
Boolean Expressions

true



false


boolean expression
true
false
command(s)
command(s)


5 == 5 true
5 < 5 false
3 < 5 true
x > 5 true or false?
int x = 5;
Not a boolean!

Boolean Expressions

- A Boolean expression is an expression that is either `true` or `false`
- The simplest Boolean expressions compare the value of two expressions

```
time < limit

yourScore == myScore
```

  - Note that Java uses two equal signs (`==`) to perform equality testing:
    - a single equal sign (`=`) is used only for assignment
  - A Boolean expression does not need to be enclosed in parentheses, unless it is used in an `if-else` statement

Java Comparison Operators

Display 3.3 Java Comparison Operators

MATH NOTATION	NAME	JAVA NOTATION	JAVA EXAMPLES

=	Equal to	`==`	`x + 7 == 2*y`
			`answer == 'y'`

≠	Not equal to	`!=`	`score != 0`
			`answer != 'y'`

>	Greater than	`>`	`time > limit`
≥	Greater than or equal to	`>=`	`age >= 21`
<	Less than	`<`	`pressure < max`
≤	Less than or equal to	`<=`	`time <=limit`

Pitfall: Using == with Strings

- The equality comparison operator (`==`) can correctly test two values of a ***primitive*** type
- However, when applied to two ***objects*** such as objects of the `String` class, `==` tests to see if they are stored in the same memory location (address), not whether or not they have the same value

- what the difference b/w (assume a=2; b=3;)
  1. (a = b)
     - printing a and b will produce
  2. (a==b)
     - print a and b will produce

- Is a.retik == arkady.retik ?
- both live at the same address
- equal? Need to look 'inside'

What the difference?

- HW1.docs == HW1.rtf
- HW1.docs equals HW1.pdf

$\}$ file extension is hidden

Don’t use == with Strings >>> use **equals**

- In order to test two strings to see if they have equal values, use the method **`equals`**, or **`equalsIgnoreCase`**

```
string1.equals(string2)

string1.equalsIgnoreCase(string2)
```

Hello

hello

heLLo

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

Same result? 👍 👎

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

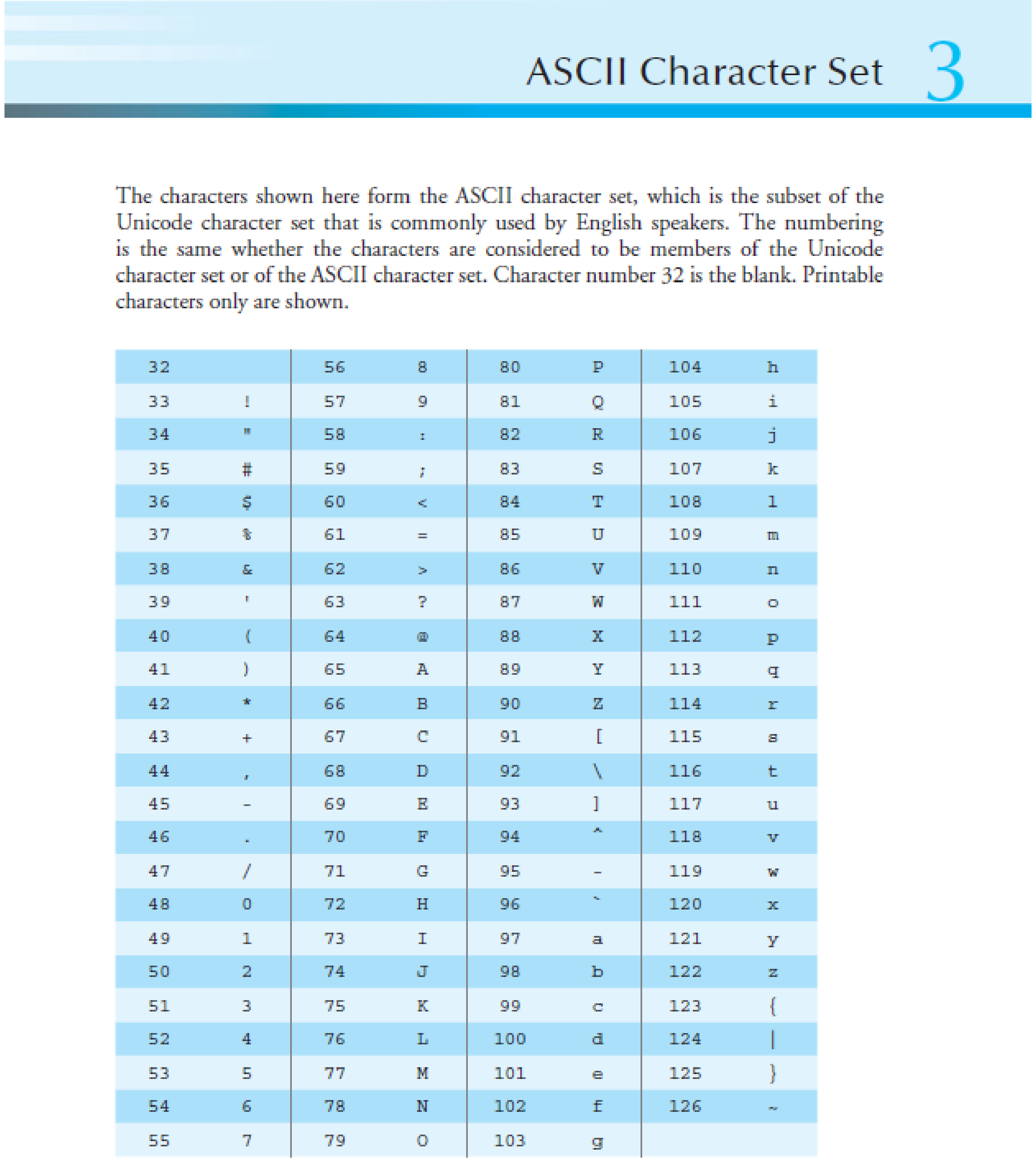
Lexicographic and Alphabetical Order

- ***Lexicographic*** ordering is the same as ***ASCII*** ordering, and includes letters, numbers, and other characters
  - All uppercase letters are in alphabetic order, and all lowercase letters are in alphabetic order, but all **uppercase letters come before lowercase letters**
  - If `s1` and `s2` are two variables of type `String` that have been given `String` values, then
    `s1.compareTo(s2)` **returns a negative number** if `s1` comes before `s2` in lexicographic ordering, **returns zero if the two strings are equal**, and **returns a positive number** if `s2` comes before `s1`

- When performing an **alphabetic comparison** of strings (rather than a lexicographic comparison) that consist of a mix of lowercase and uppercase letters, use the `compareToIgnoreCase` method instead

Why ?

Appendix 3
in *Absolute Java*



ASCII Character Set 3

The characters shown here form the ASCII character set, which is the subset of the Unicode character set that is commonly used by English speakers. The numbering is the same whether the characters are considered to be members of the Unicode character set or of the ASCII character set. Character number 32 is the blank. Printable characters only are shown.

32		56	8	80	P	104	h
33	!	57	9	81	Q	105	i
34	"	58	:	82	R	106	j
35	#	59	;	83	S	107	k
36	$	60	<	84	T	108	l
37	%	61	=	85	U	109	m
38	&	62	>	86	V	110	n
39	'	63	?	87	W	111	o
40	(	64	@	88	X	112	p
41	)	65	A	89	Y	113	q
42	*	66	B	90	Z	114	r
43	+	67	C	91	[	115	s
44	,	68	D	92	\	116	t
45	-	69	E	93	]	117	u
46	.	70	F	94	^	118	v
47	/	71	G	95	_	119	w
48	0	72	H	96	`	120	x
49	1	73	I	97	a	121	y
50	2	74	J	98	b	122	z
51	3	75	K	99	c	123	{
52	4	76	L	100	d	124	|
53	5	77	M	101	e	125	}
54	6	78	N	102	f	126	~
55	7	79	O	103	g		

Building Boolean Expressions: 3 cases

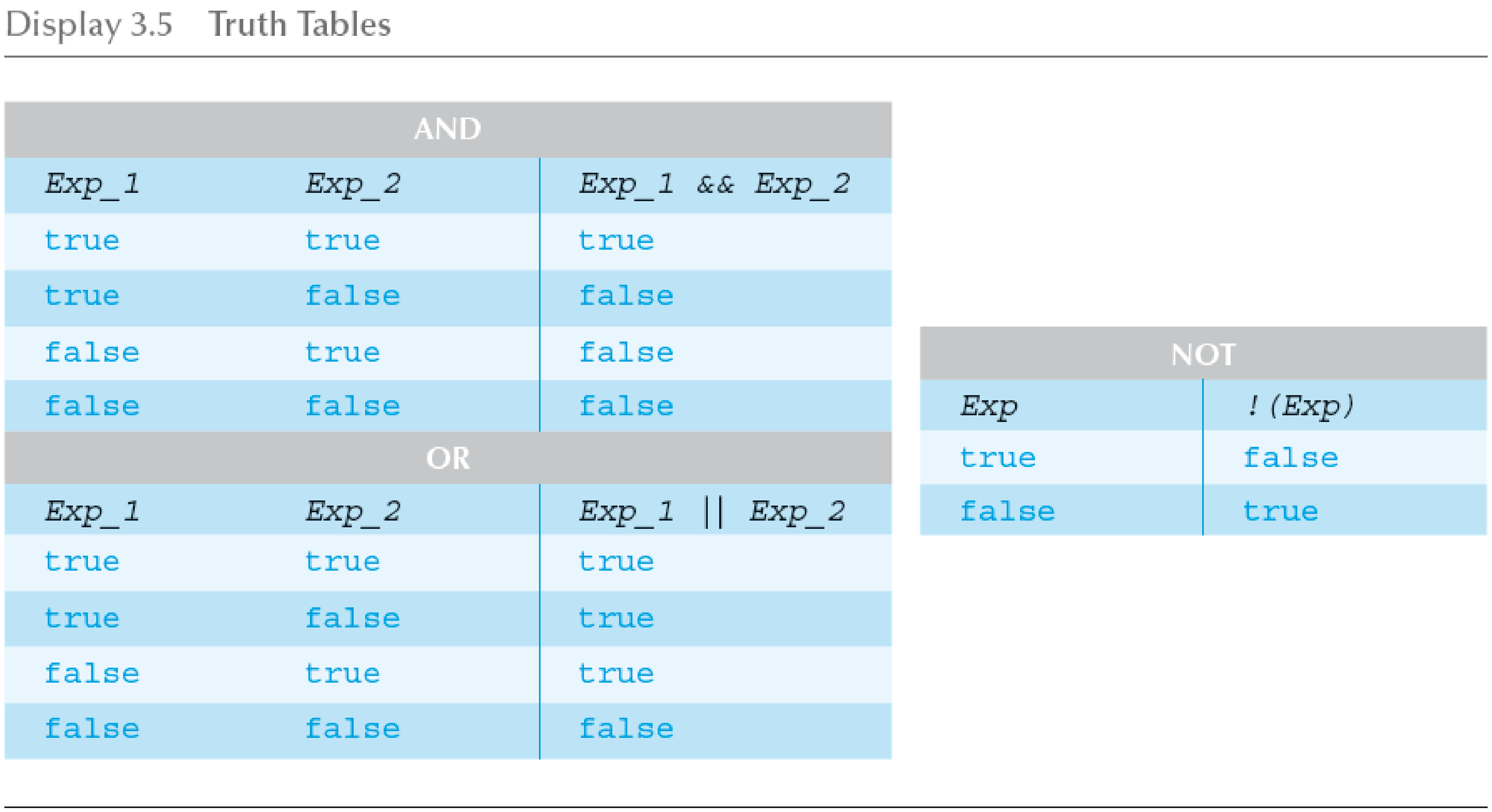
1. When two Boolean expressions are **combined** using the *"and"* (`&&`) operator, the entire expression is true provided **both** expressions are true
   - Otherwise the expression is false
2. When two Boolean expressions are **combined** using the *"or"* (`||`) operator, the entire expression is true as long as **one of the expressions** is true
   - The expression is false only if both expressions are false
3. Any Boolean **expression can be negated** using the `!` operator
   - Place the expression in parentheses and place the `!` operator in front of it

- Unlike mathematical notation, strings of inequalities must be joined by `&&`
  - Use `(min < result) && (result < max)` rather than min < ~~result~~ < max

Evaluating Boolean Expressions

- Even though Boolean expressions are used to control branch and loop statements, Boolean expressions **can exist independently** as well
  - A Boolean variable can be given the value of a Boolean expression by using an assignment statement
- A Boolean expression can be evaluated in the same way that an arithmetic expression is evaluated
  - The only difference is that arithmetic expressions produce a number as a result, while **Boolean expressions produce either `true` or `false` as their result**
  - `boolean madeIt = (time < limit) && (limit < maxLimit);`

Truth Tables



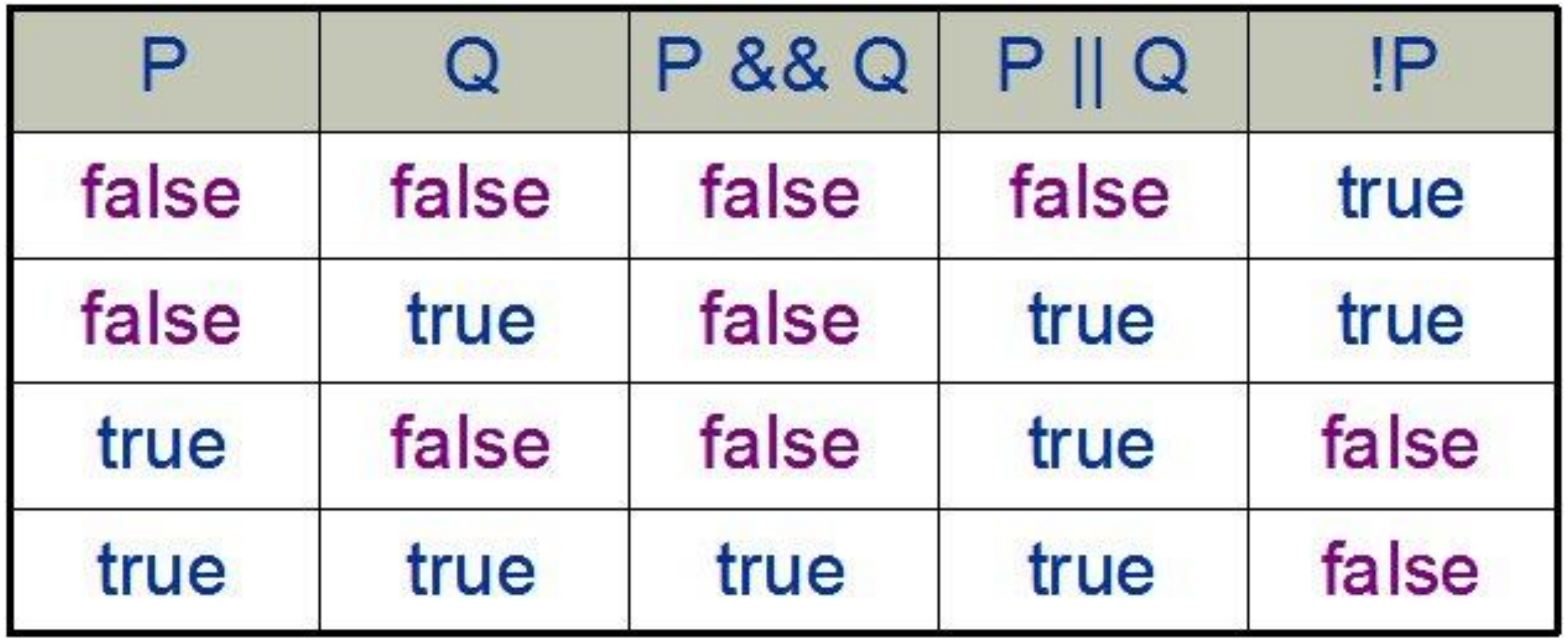
Display 3.5 Truth Tables

AND		
Exp_1	*Exp_2*	*Exp_1 && Exp_2*
`true`	`true`	`true`
`true`	`false`	`false`
`false`	`true`	`false`
`false`	`false`	`false`

OR		
Exp_1	*Exp_2*	*Exp_1 || Exp_2*
`true`	`true`	`true`
`true`	`false`	`true`
`false`	`true`	`true`
`false`	`false`	`false`

NOT	
Exp	*!(Exp)*
`true`	`false`
`false`	`true`

Truth Tables



P	Q	P && Q	P || Q	!P
false	false	false	false	true
false	true	false	true	true
true	false	false	true	false
true	true	true	true	false

Short-Circuit and Complete Evaluation

- Java can take a shortcut when the evaluation of the first part of a Boolean expression produces a result that evaluation of the second part cannot change
- This is called *short-circuit evaluation* or *lazy evaluation*
  - For example, when evaluating two Boolean subexpressions joined by **`&&`**, if the first subexpression evaluates to **`false`**, then the entire expression will evaluate to **`false`**, no matter the value of the second subexpression
  - In like manner, when evaluating two Boolean subexpressions joined by **`||`**, if the first subexpression evaluates to **`true`**, then the entire expression will evaluate to **`true`**

Short-Circuit and Complete Evaluation

- There are times when using short-circuit evaluation can prevent a *runtime error*
  - In the following example, if the number of `kids` is equal to zero, then the second subexpression will not be evaluated, thus preventing a *divide by zero error*
  - Note that reversing the order of the subexpressions will not prevent this

```
if ((kids !=0) && ((toys/kids) >=2)) . . .
```

- Sometimes it is preferable to always evaluate both expressions, i.e., request complete evaluation
  - In this case, use the `&` and `|` operators instead of `&&` and `||`
  - **`in 142 we will only use`** `&& and ||`

Precedence and Associativity Rules

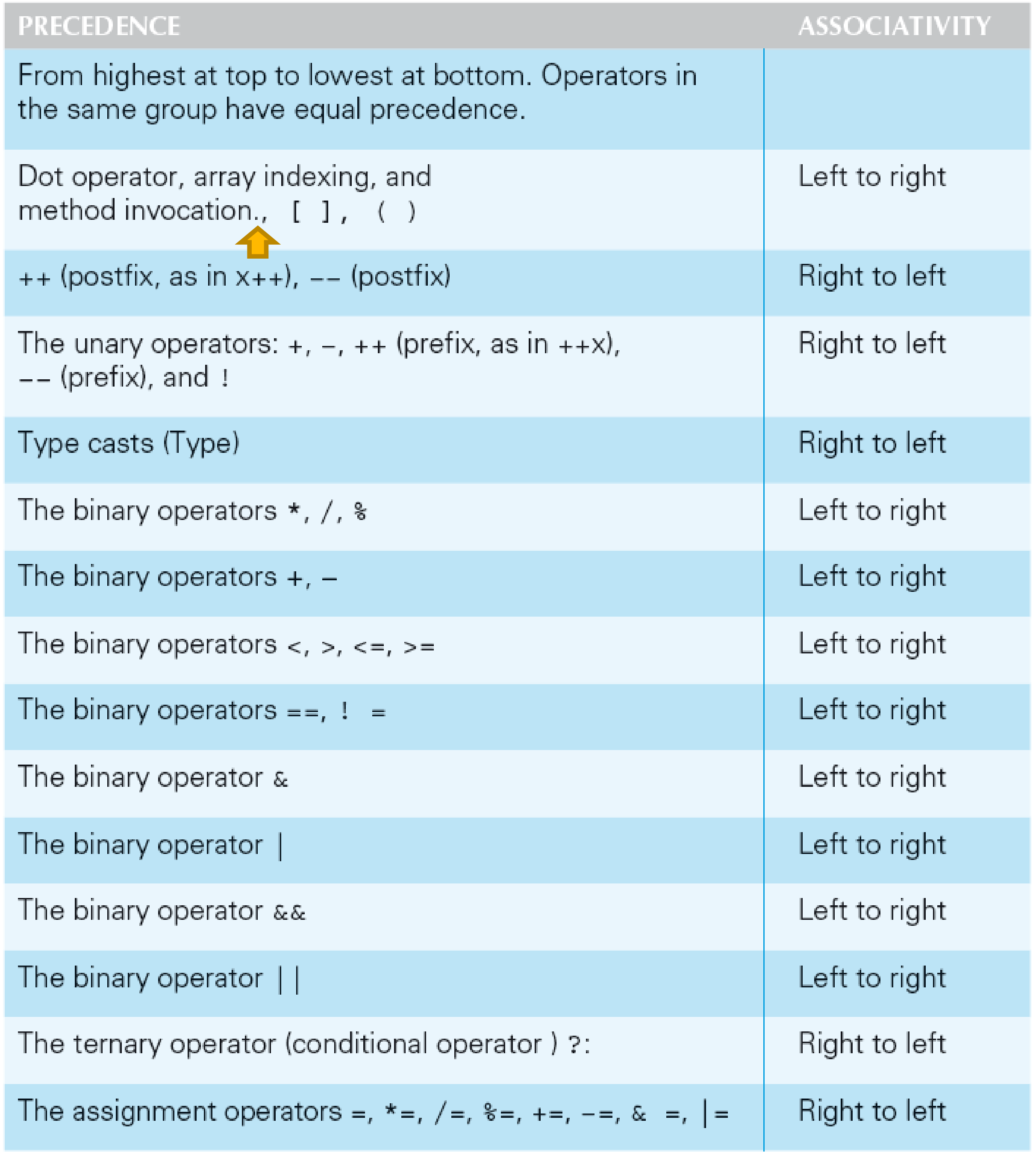
- Boolean and arithmetic expressions need not be fully parenthesized
- If some or all of the parentheses are omitted, Java will follow *precedence* and *associativity* rules (summarized in the following table) to determine the order of operations
  - If one operator occurs **higher** in the table than another, it has ***higher precedence***, and is grouped with its operands before the operator of *lower precedence*
  - If two operators have the **same precedence**, then ***associativity rules*** determine which is grouped first

Precedence and Associativity Rules

Examples?

Savitch p.125-131

Highest Precedence

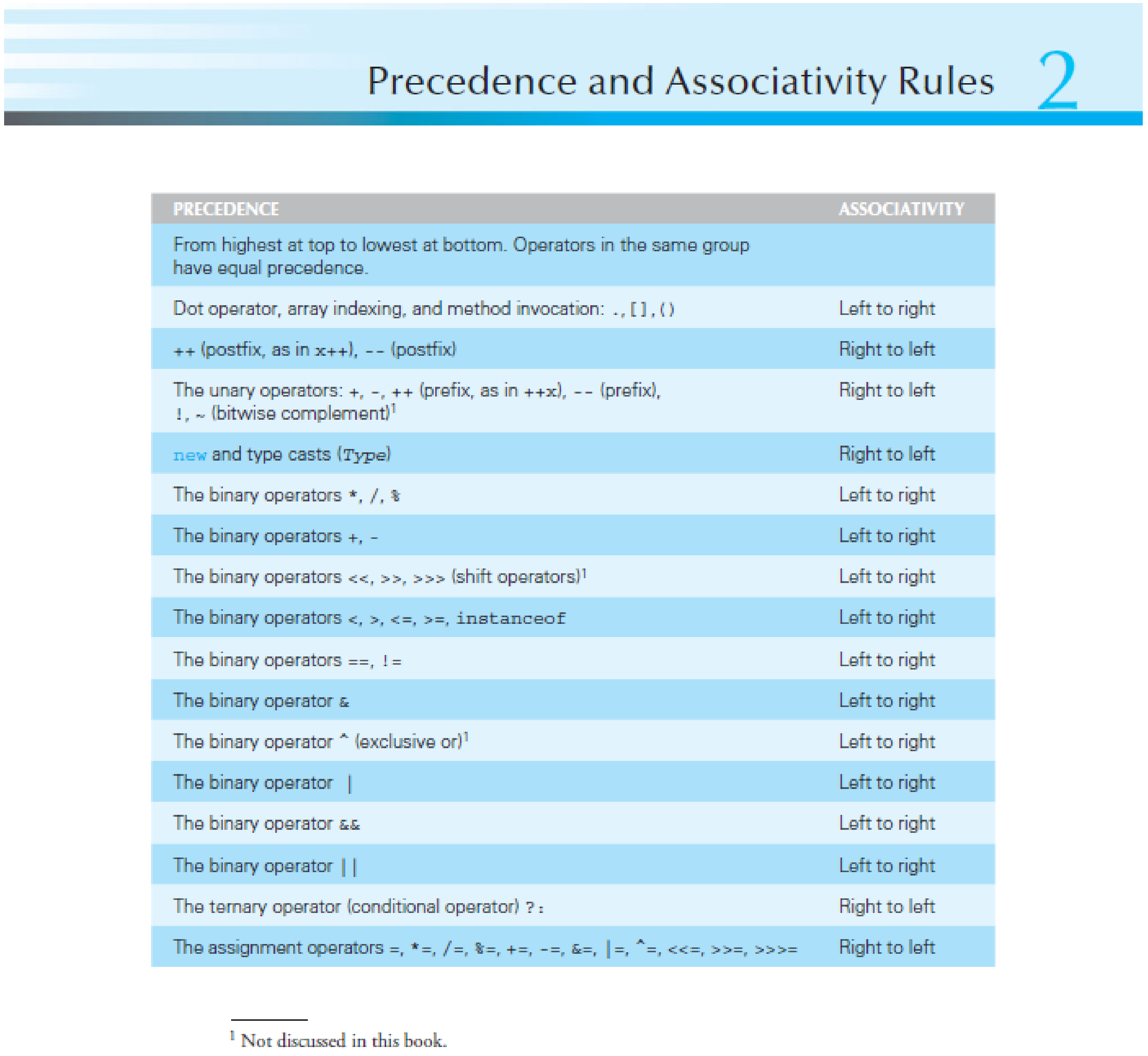


PRECEDENCE	ASSOCIATIVITY
From highest at top to lowest at bottom. Operators in the same group have equal precedence.	
Dot operator, array indexing, and method invocation., [ ], ( )	Left to right
++ (postfix, as in x++), -- (postfix)	Right to left
The unary operators: +, -, ++ (prefix, as in ++x), -- (prefix), and !	Right to left
Type casts (Type)	Right to left
The binary operators *, /, %	Left to right
The binary operators +, -	Left to right
The binary operators <, >, <=, >=	Left to right
The binary operators ==, ! =	Left to right
The binary operator &	Left to right
The binary operator |	Left to right
The binary operator &&	Left to right
The binary operator ||	Left to right
The ternary operator (conditional operator) ?:	Right to left
The assignment operators =, *=, /=, %=, +=, -=, & =, |=	Right to left

Lowest Precedence

Appendix 2 in Absolute Java

TMF



Precedence and Associativity Rules 2

PRECEDENCE	ASSOCIATIVITY
From highest at top to lowest at bottom. Operators in the same group have equal precedence.	
Dot operator, array indexing, and method invocation: `.`, `[ ]`, `( )`	Left to right
`++` (postfix, as in `x++`), `--` (postfix)	Right to left
The unary operators: `+`, `-`, `++` (prefix, as in `++x`), `--` (prefix), `!`, `~` (bitwise complement)[1]	Right to left
`new` and type casts (*Type*)	Right to left
The binary operators `*`, `/`, `%`	Left to right
The binary operators `+`, `-`	Left to right
The binary operators `<<`, `>>`, `>>>` (shift operators)[1]	Left to right
The binary operators `<`, `>`, `<=`, `>=`, `instanceof`	Left to right
The binary operators `==`, `!=`	Left to right
The binary operator `&`	Left to right
The binary operator `^` (exclusive or)[1]	Left to right
The binary operator `|`	Left to right
The binary operator `&&`	Left to right
The binary operator `||`	Left to right
The ternary operator (conditional operator) `?:`	Right to left
The assignment operators `=`, `*=`, `/=`, `%=`, `+=`, `-=`, `&=`, `|=`, `^=`, `<<=`, `>>=`, `>>>=`	Right to left

[1] Not discussed in this book.

Evaluating Expressions

- In general, parentheses in an expression help to document the programmer's intent
  - Instead of relying on precedence and associativity rules, **it is best to include most parentheses, except where the intended meaning is obvious**
- ***Binding***: The association of operands with their operators
  - A fully parenthesized expression accomplishes binding for all the operators in an expression
- ***Side Effects***: When, in addition to returning a value, an expression changes something, such as the value of a variable
  - The *assignment*, *increment*, and *decrement* operators all produce side effects

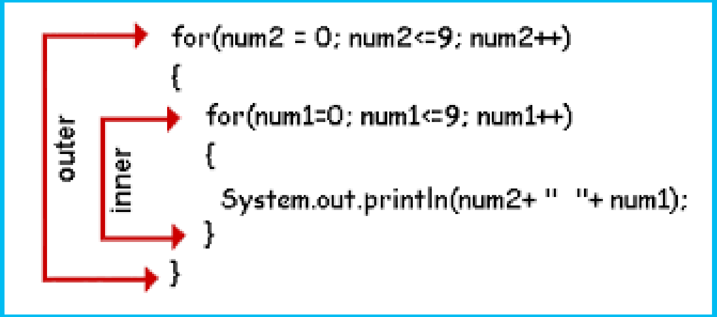
Examples?

Savitch p129-130

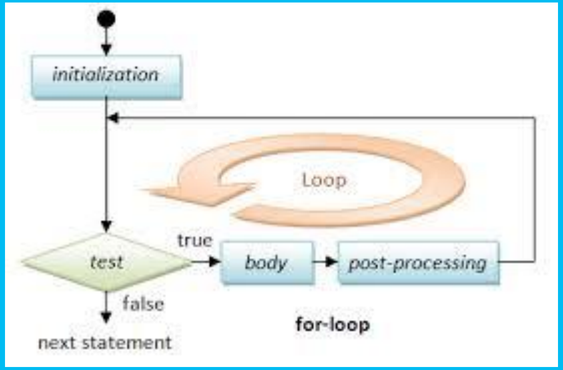
Rules for Evaluating Expressions

- Perform binding
  - Determine the equivalent fully parenthesized expression using the precedence and associativity rules
- Proceeding left to right, evaluate whatever subexpressions can be immediately evaluated
  - These subexpressions will be operands or method arguments, e.g., numeric constants or variables
- Evaluate each outer operation and method invocation as soon as all of its operands (i.e., arguments) have been evaluated

LOOPS



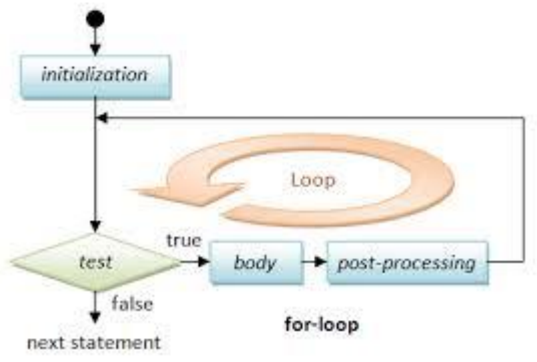
```
for(num2 = 0; num2<=9; num2++)
{
  for(num1=0; num1<=9; num1++)
  {
    System.out.println(num2+ "   "+ num1);
  }
}
```

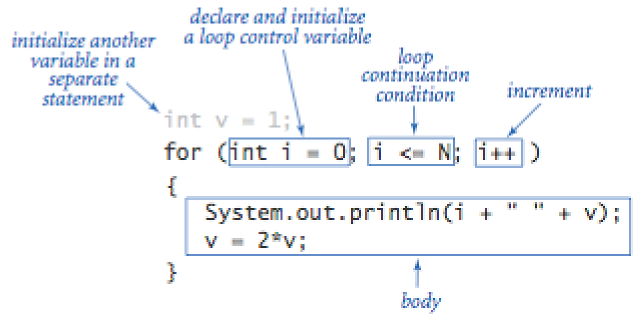


Preview for Mon

Loops

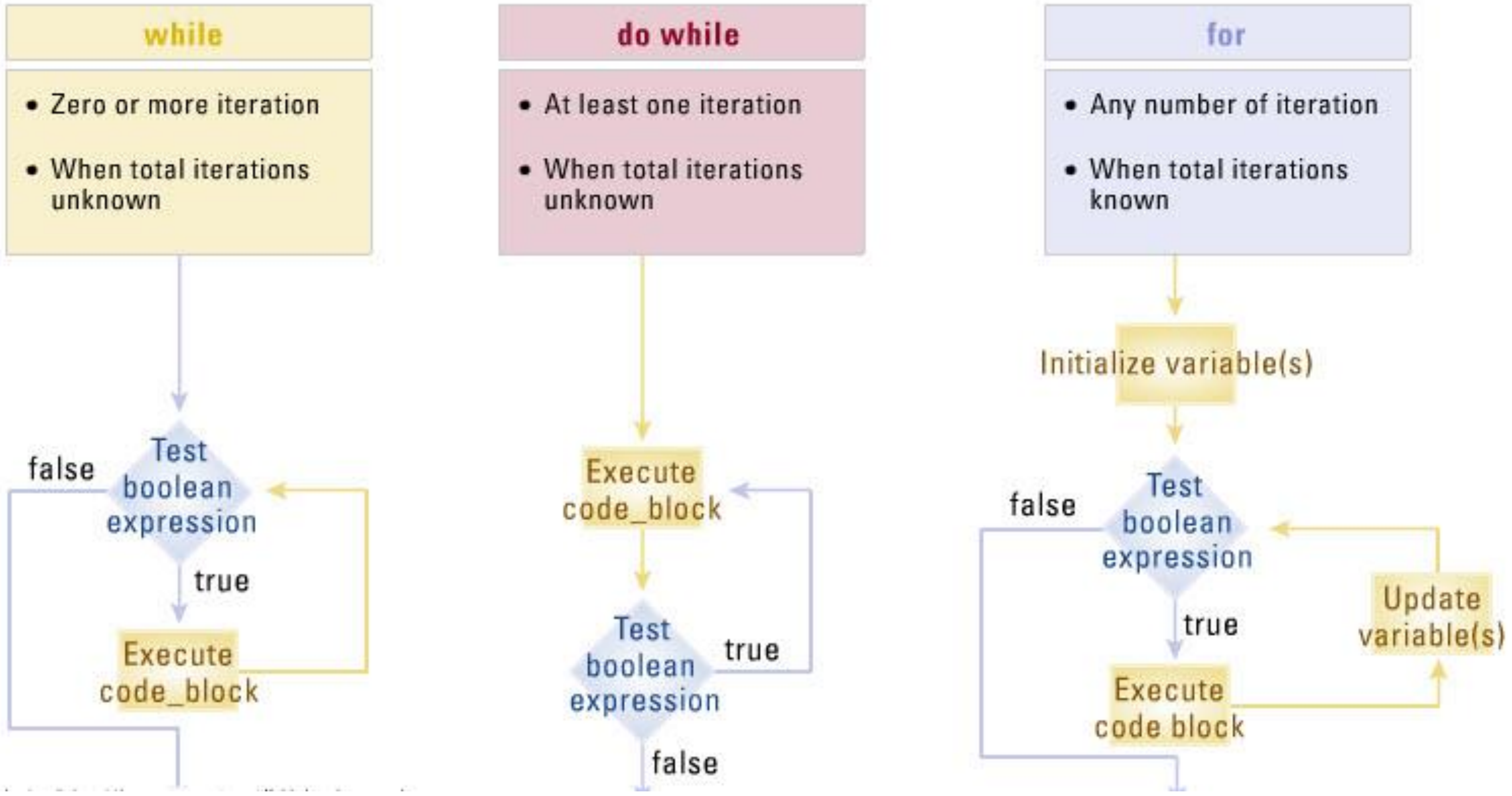
- *Loops* in Java are similar to those in other high-level languages
- Java has three types of loop statements: the **`while`**, the **`do-while`**, and the **`for`** statements
  - The code that is repeated in a loop is called the *body* of the loop
  - Each repetition of the loop body is called an *iteration* of the loop





declare and initialize a loop control variable
initialize another variable in a separate statement
loop continuation condition
increment
body

```
int v = 1;
for (int i = 0; i <= N; i++)
{
   System.out.println(i + " " + v);
   v = 2*v;
}
```


while
Zero or more iteration
When total iterations unknown
false
Test boolean expression
true
Execute code_block
do while
At least one iteration
When total iterations unknown
Execute code_block
Test boolean expression
true
false
for
Any number of iteration
When total iterations known
Initialize variable(s)
false
Test boolean expression
true
Update variable(s)
Execute code block

print largest power of two less than or equal to N	`int v = 1;`
	`while (v <= N/2)`
	`   v = 2*v;`
	`System.out.println(v);`

compute a finite sum (1 + 2 + . . . + N)	`int sum = 0;`
	`for (int i = 1; i <= N; i++)`
	`   sum += i;`
	`System.out.println(sum);`

compute a finite product (N! = 1 × 2 × . . . × N)	`int product = 1;`
	`for (int i = 1; i <= N; i++)`
	`   product *= i;`
	`System.out.println(product);`

print a table of function values	`for (int i = 0; i <= N; i++)`
	`   System.out.println(i + " " + 2*Math.PI*i/N);`

Table: EXAMPLES

`while` statement

- A `while` statement is used to repeat a portion of code (i.e., the loop body) based on the evaluation of a Boolean expression
  - The Boolean expression is checked ***before*** the loop body is executed
    - When **false**, the loop body is **not executed at all**
  - Before the execution of each following iteration of the loop body, the Boolean expression is checked again
    - If true, the loop body is executed again
    - If false, the loop statement ends
  - The loop body can consist of a single statement, or multiple statements enclosed in a pair of braces (`{  }`)

`while` Syntax

```
while (Boolean_Expression)
    Statement
```

Or

```
while (Boolean_Expression)
{
    Statement_1
    Statement_2
          :
          :
    Statement_Last
}
```

`while` statement: examples

Write down an example

- A `while` statement example

`do-while` Statement

- A `do-while` statement is used to execute a portion of code (i.e., the loop body), and then repeat it based on the evaluation of a Boolean expression
  - **The loop body is executed at least once**
    - The Boolean expression is checked ***after*** the loop body is executed
  - The Boolean expression is checked after each iteration of the loop body
    - If true, the loop body is executed again
    - If false, the loop statement ends
    - Don't forget to put a semicolon after the Boolean expression
  - Like the while statement, the loop body can consist of a single statement, or multiple statements enclosed in a pair of braces (`{ }`)

do-while Syntax

```
do
    Statement
while (Boolean_Expression);
```

or

```
do
{
    Statement_1
    Statement_2
          .
          .
          .
    Statement_Last
 } while (Boolean_Expression);
```

do-while statement: examples

Write down an example

- A do-while statement example

The `for` Statement

- The `for` statement is most commonly used to step through an integer variable in equal increments
- It begins with the keyword `for`, followed by **three expressions** in parentheses that describe what to do with one or more *controlling variables*
  - The first expression tells how the control variable or variables are *initialized* or *declared* and *initialized* before the first iteration
  - The second expression determines when the loop should *end*, based on the evaluation of a Boolean expression *before* each iteration
  - The third expression tells how the control variable or variables are *updated after* each iteration of the loop body

The `for` Statement Syntax

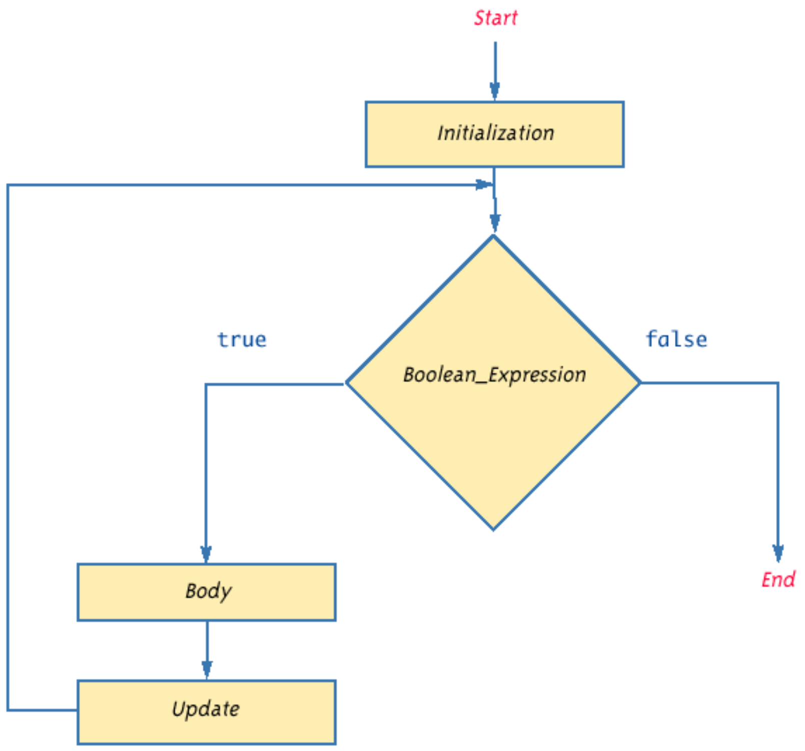
```
for (Initializing; Boolean_Expression; Update)
   Body
```

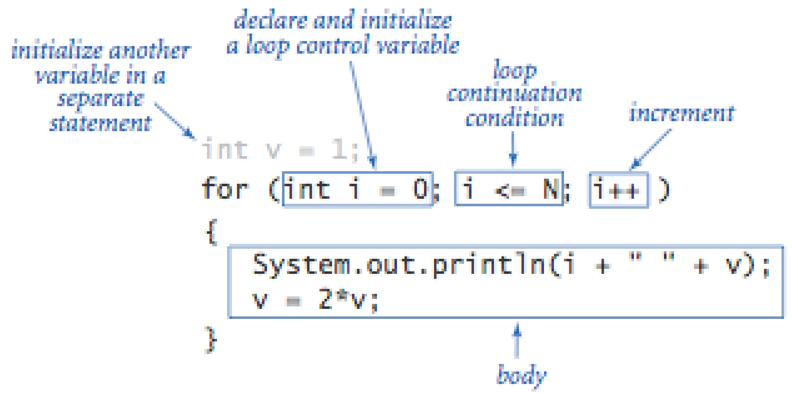
- The `Body` may consist of a single statement or a list of statements enclosed in a pair of braces (`{ }`)
- Note that the three control expressions are separated by **two semicolons**
- Note that there is **no semicolon after the closing** parenthesis at the beginning of the loop

Semantics of the `for` Statement

Display 3.9 Semantics of the for Statement

```
for (Initialization; Boolean_Expression; Update)
        Body
```




initialize another variable in a separate statement
declare and initialize a loop control variable
loop continuation condition
increment
body

```
int v = 1;
for (int i = 0; i <= N; i++)
{
   System.out.println(i + " " + v);
   v = 2*v;
}
```

The `for` Statement: examples

Write down an example

- The `for` statement example

The Comma in `for` Statements

- A `for` loop can contain multiple initialization actions separated with commas
  - Caution must be used when combining a declaration with multiple actions
  - It is illegal to combine multiple type declarations with multiple actions, for example
  - To avoid possible problems, it is best to declare all variables outside the `for` statement
- A `for` loop can contain multiple update actions, separated with commas, also
  - It is even possible to eliminate the loop body in this way
- However, a `for` loop can contain only one Boolean expression to test for ending the loop

Next

- **Read:** Savitch Chapter 3 (3.1, 3.2, **3.3,** 3.4)
  - We will also have Activity in class
- **Homework 2 is due soon**
  - Read it and ask questions

Hands-on: Class Activity (HoA)