



## Chapter Three: Decisions

# Chapter Goals

---

- ใช้ **if statement** ให้เป็น
- รู้ว่าเปรียบเทียบ **integers, floating-point numbers,** และ **strings** อย่างไร
- เข้าใจข้อมูลชนิด **Boolean**

# The `if` Statement

---

การตัดสินใจ

(สิ่งที่จำเป็นในโปรแกรมใหญ่)

# The `if` Statement



We aren't lost!

We just haven't decided which way to go ... yet.

# The `if` Statement

---

## The `if` statement

allows a program to carry out different actions depending on the nature of the data being processed

ช่วยให้โปรแกรมทำการกระทำต่าง ๆ ให้สำเร็จ โดยดูที่ธรรมชาติของข้อมูลที่กำลังอยู่ในกระบวนการ

# The `if` Statement

The `if` statement is used to implement a decision.

- When a condition is fulfilled, one set of statements is executed.
- Otherwise, another set of statements is executed.

ใช้ `if` statement ในการที่เราจะให้โปรแกรมตัดสินใจ

- เมื่อเงื่อนไขหนึ่งเป็นจริง  
เซตของ **statements** ชุดหนึ่งจะถูก **executed**.
- ถ้าไม่เป็นจริง,  
อีกเซตของ **statements** จะถูก **executed**.

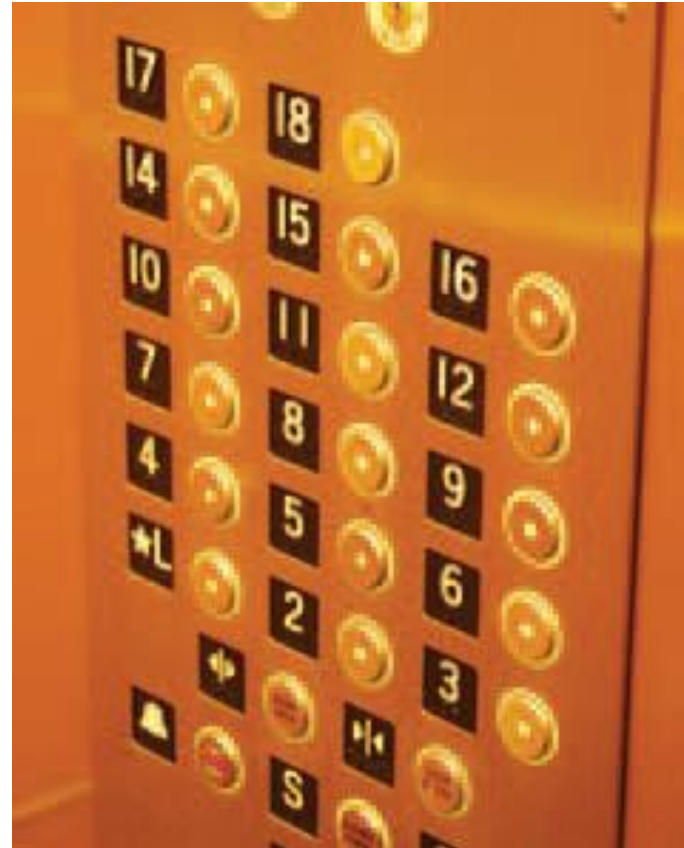
# The `if` Statement



if it's quicker to the candy mountain,  
we'll go that way  
else  
we go that way

# The `if` Statement

*The thirteenth floor!*





# The `if` Statement

*The thirteenth floor!*

*It's missing!*



# The `if` Statement

*The thirteenth floor!*  
*It's missing!*

OH NO !!!

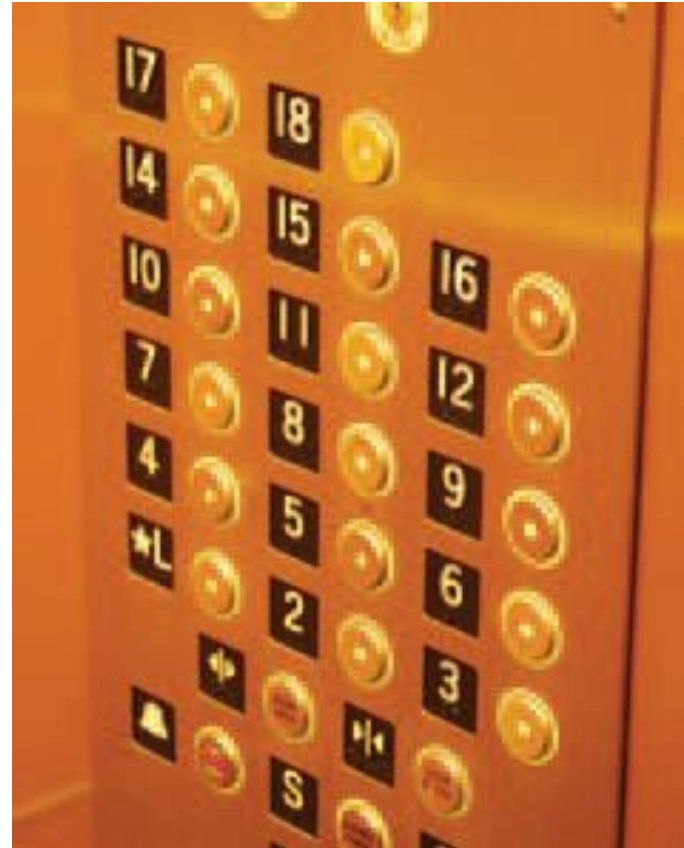


# The `if` Statement

We must write the code to control the elevator.

How can we skip the 13<sup>th</sup> floor?

เราต้องเขียนโค้ดเพื่อ  
ควบคุมลิฟท์  
แต่ผู้ใช้ไม่มีปุ่ม 13 ให้กด?



# The `if` Statement

---

We will model a person choosing a floor by getting input from the user:

เราจะจำลองการเลือกชั้นของบุคคลหนึ่ง โดยการรับข้อมูลจากผู้ใช้

```
int floor;  
Scanner sc= new Scanner(System.in) ;  
System.out.print("Floor: ");  
floor = sc.nextInt();
```

# The if Statement

ถ้าผู้ใช้ป้อนชั้นมากกว่า 13 เข้ามา เช่นชั้น 20,  
โปรแกรมต้องตั้งค่าชั้นจริงๆ (*actual floor*) ให้เท่ากับ 19.

ถ้าเป็นกรณีอื่น,  
โปรแกรมใช้ตัวเลขที่ป้อนเข้ามาได้เลย

เราจำเป็นต้องลดค่า input ลงหนึ่งค่าภายใต้เงื่อนไขหนึ่ง:

```
int actual_floor;  
if (floor > 13) {  
    actual_floor = floor - 1;  
}  
else {  
    actual_floor = floor;  
}
```

# The if Statement

## SYNTAX 3.1 if Statement

A condition that is true or false.  
Often uses relational operators:  
== != < <= > >=

```
if (floor > 13)
{
    actual_floor = floor - 1;
}
else
{
    actual_floor = floor;
}
```

Braces are not required if the branch contains a single statement, but it's good to always use them.

Omit the else branch if there is nothing to do.

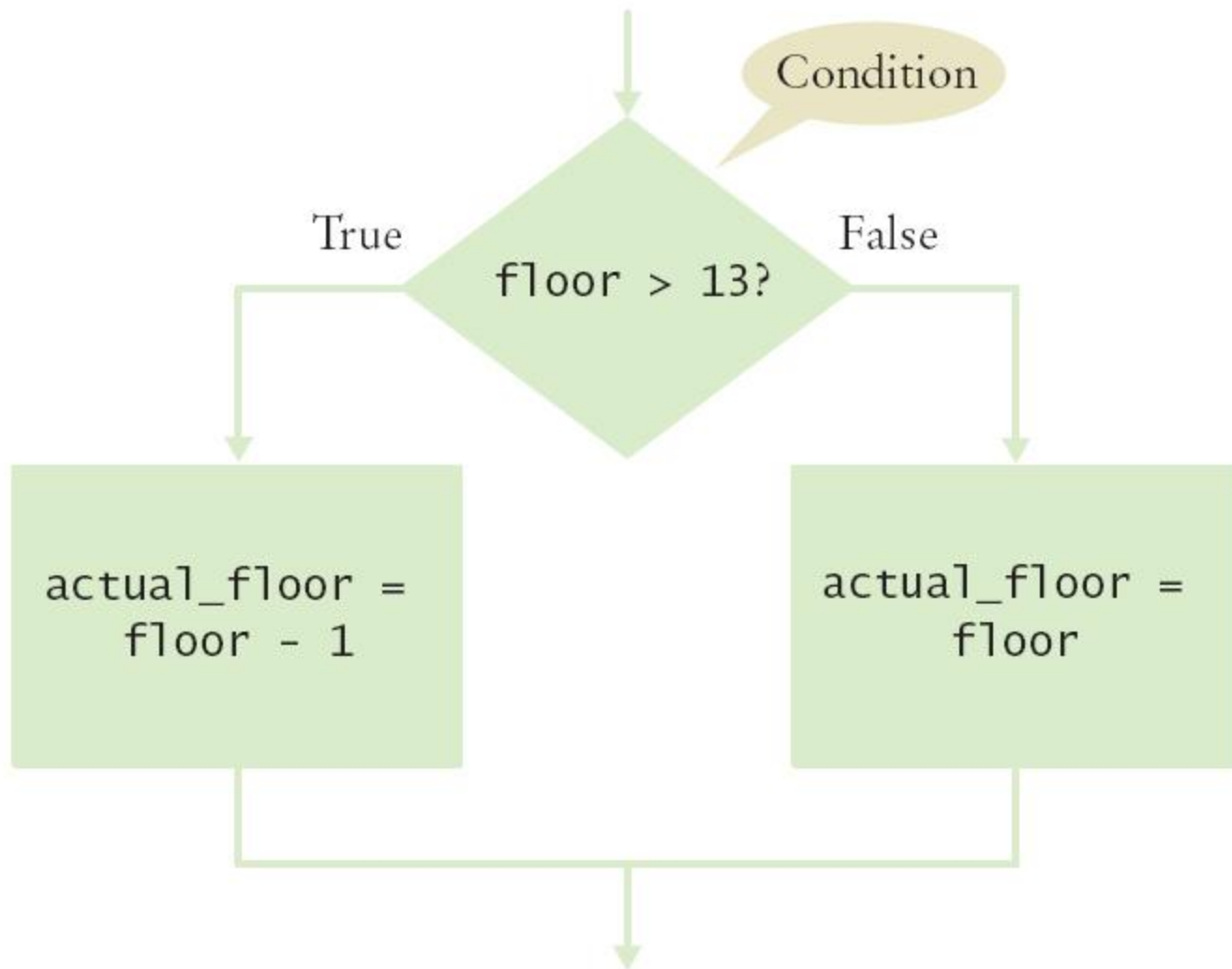
Lining up braces is a good idea.

Don't put a semicolon here!

If the condition is true, the statement(s) in this branch are executed in sequence; if the condition is false, they are skipped.

If the condition is false, the statement(s) in this branch are executed in sequence; if the condition is true, they are skipped.

# The `if` Statement – The Flowchart



# The `if` Statement

---

Sometimes, it happens that there is nothing to do in the `else` branch of the statement.

So don't write it.

บางครั้ง เราก็ไม่อยากให้โปรแกรมทำอะไรใน block ของ `else`

ดังนั้นก็ไม่ต้องเขียนมัน



# The if Statement

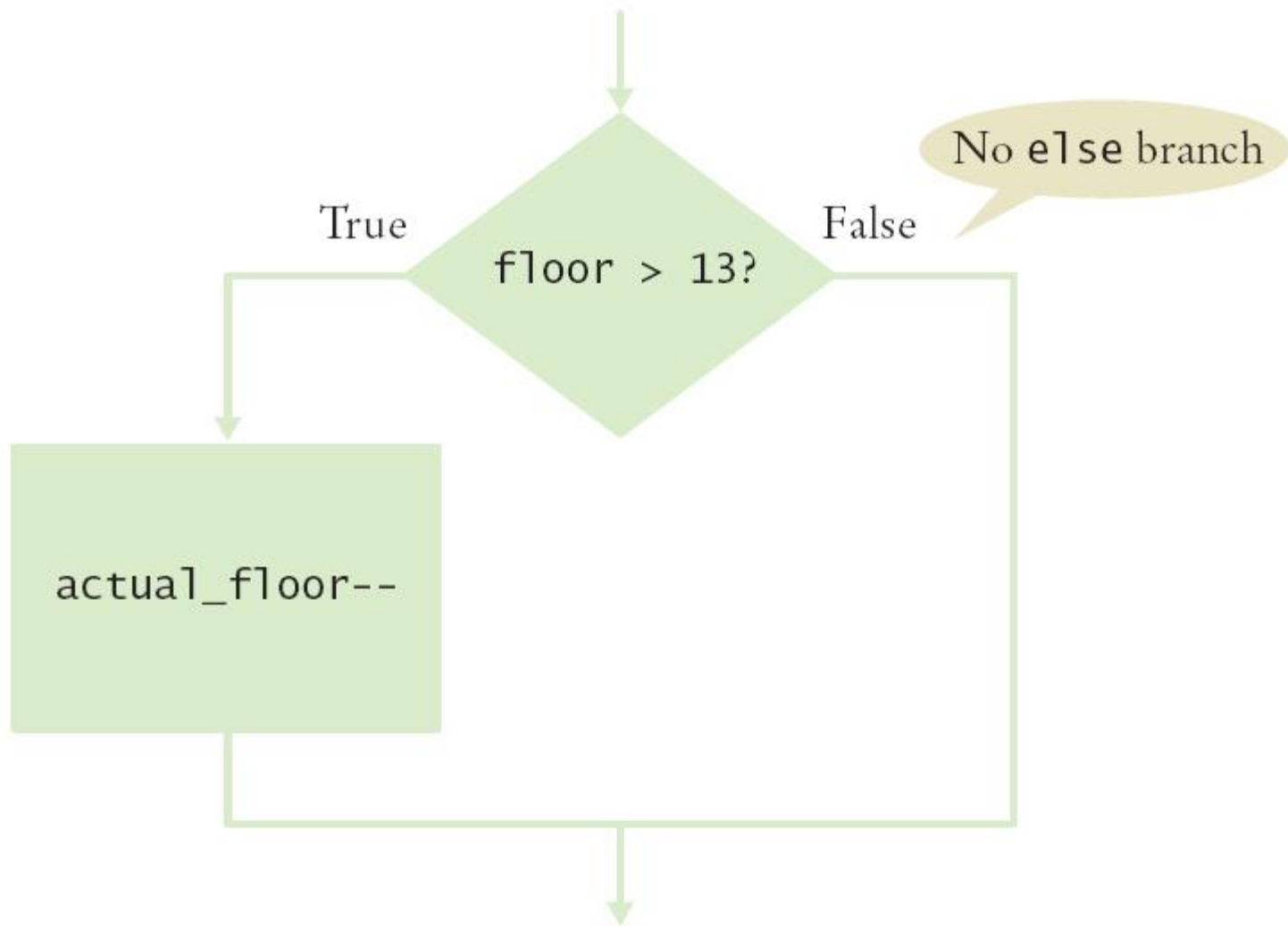
วิธีที่จะเขียนโค้ดอีกแบบ:

เราจะ *decrement* ก็ต่อเมื่อ  $floor > 13$ .

เรา set `actual_floor` ได้เลยก่อนจะ test เงื่อนไข:

```
int actual_floor = floor;  
if (floor > 13) {  
    actual_floor--;  
} // ไม่ต้องใส่ else ตรงนี้ก็ได้อ
```

# The `if` Statement – The Flowchart



# The `if` Statement – A Complete Elevator Program

```
public static void main(String [] args) {  
  
    int floor;  
    Scanner sc_obj = new Scanner(System.in);  
    System.out.println("Floor: ");  
    floor = sc_obj.nextInt();  
    int actual_floor;  
    if (floor > 13) {  
        actual_floor = floor - 1;  
    } else {  
        actual_floor = floor;  
    }  
  
    System.out.println("The elevator will travel to the actual  
                        floor " + actual_floor);  
}
```

# The `if` Statement – Brace Layout

- การทำโค้ดให้อ่านง่ายเป็นการปฏิบัติที่ดี
- การจัดย่อหน้าช่วยเราได้มาก

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

## The `if` Statement – Brace Layout

---

- As long as the ending brace clearly shows what it is closing, there is no confusion.

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

Some programmers prefer this style  
—it saves a physical line in the code.

## The `if` Statement – Always Use Braces

---

When the body of an `if` statement consists of a single statement, you need not use braces:

ในกรณีที่ `if` statement มีแค่ statement เดียวใน block เราไม่ต้องใช้ปีกกา

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

## The `if` Statement – Always Use Braces

---

However, it is a good idea to always include the braces:

- the braces makes your code easier to read, and
- you are less likely to make errors such as ...

แต่เราก็มีปีกกาไว้เถอะ เพราะจะอ่านโค้ดได้ง่ายกว่า และ มีโอกาสที่ error จะเกิดน้อยกว่า

# The `if` Statement – Common Error – The Do-nothing Statement

---

Can you see the error?

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



# The if Statement – Common Error – The Do-nothing Statement

```
if (floor > 13) ; // ERROR ?  
{  
  floor-- ;  
}
```

แบบนี้ไม่ใช่ compiler error.  
compiler จะไม่แจ้งเตือนเราเลย  
มันแปลง if statement นี้ ดังด้านล่าง:

ถ้า  $\text{floor} > 13$ , execute *statement* ที่ไม่ทำอะไรเลย  
(semicolon อย่างเดียวเป็น statement ได้ แต่เป็น statement ที่ไม่ทำอะไร)

หลังจากนั้น โปรแกรม execute โค้ดในปีกกา โดยไม่สนใจเงื่อนไข ( $\text{floor} > 13$ ) อีก  
นั่นคือ *statement* ในปีกกาไม่ได้เป็นส่วนหนึ่งของ if statement อีก

# The `if` Statement – Common Error – The Do-nothing Statement

Can you see the error?

This one should be easy now!

```
if (floor > 13)
{
    actual_floor = floor - 1;
}
else ; ERROR
{
    actual_floor = floor;
}
```

And it really *is* an error this time.

# The `if` Statement – ย่อหน้า เมื่อทำ Nesting (if ซ้อน if)

Block-structured code has the property that *nested* statements are indented by one or more levels.

```
public static void main(String [] args)
{
    int floor;
    ..
    if (floor > 13)
    {
        floor--;
    }
    ..
    return 0;
}
```

0 1 2  
Indentation level

## The `if` Statement – ย่อหน้า เมื่อทำ Nesting

ให้ใช้ `tab key` ในการย่อหน้า

แต่ ...

ไม่ใช่ทุก editor จะมีความกว้าง `tab` เท่ากัน

Luckily most development environments have settings to automatically convert all tabs to spaces.

โชคดีที่ โปรแกรม editor ส่วนใหญ่มีการตั้งค่าให้แปลง `tab` เป็น `space` ได้

# The Conditional Operator

## The Conditional Operator

บางที เราก็คืออยากให้ภาษา **Java** มันทำอย่างนี้ได้:

```
actual_floor = if (floor > 13) {  
    floor - 1;  
}  
else {  
    floor;  
}
```

**Statements** ไม่ได้ **return** ค่าอะไร ดังนั้นเราให้พิมพ์ออกมาเป็น **output** ไม่ได้  
แต่มันก็เป็นความคิดที่ดี

# The Conditional Operator

---

C มี conditional operator ในรูปแบบ

```
condition ? value1 : value2
```

expression ข้างต้นจะ return ค่า **value1** ถ้าเงื่อนไขเป็นจริง หรือ return ค่า **value2** ถ้าเงื่อนไขเป็นเท็จ

# The Conditional Operator

---

For example, we can compute the actual floor number as

```
actual_floor = (floor > 13) ? floor - 1 : floor;
```

which is equivalent to

```
if (floor > 13)
{
    actual_floor = floor - 1;
}
else
{
    actual_floor = floor;
}
```

# The Conditional Operator

---

You can use the conditional operator anywhere that a value is expected, for example:

```
System.out.println("Actual fl: " +  
                    (floor > 13) ? floor - 1 : floor);
```

We don't use the conditional operator in this book, but it is a convenient construct that you will find in many programs.



# The `if` Statement – ลดบรรทัดที่ซ้ำ

```
if (floor > 13)
{
    actual_floor = floor - 1;
    System.out.println("Actual floor:" + actual_floor);
}
else
{
    actual_floor = floor;
    System.out.println("Actual floor: " + actual_floor);
}
```

Do you find anything curious in this code?

# The `if` Statement – ลดบรรทัดที่ซ้ำ

```
if (floor > 13)
{
    actual_floor = floor - 1;
    System.out.println("Actual floor: " + actual_floor);
}
else
{
    actual_floor = floor;
    System.out.println("Actual floor: " + actual_floor);
}
```

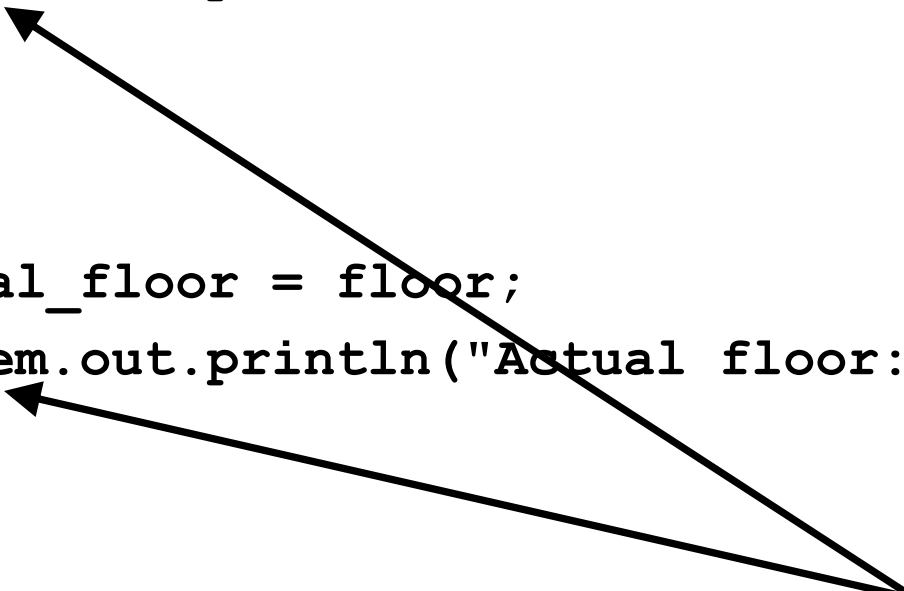
Hmmm...



# The `if` Statement – Removing Duplication

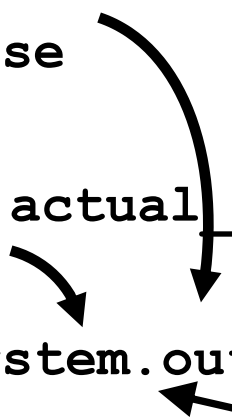
```
if (floor > 13)
{
    actual_floor = floor - 1;
    System.out.println("Actual floor: " + actual_floor);
}
else
{
    actual_floor = floor;
    System.out.println("Actual floor: " + actual_floor);
}
```

Do these depend  
on the test?



# The `if` Statement – Removing Duplication

```
if (floor > 13)
{
    actual_floor = floor - 1;
}
else
{
    actual_floor = floor;
}
System.out.println("Actual floor: " + actual_floor);
```



You should remove  
this duplication.

# Relational Operators



Which way *is* quicker to the candy mountain?

# Relational Operators



Let's compare the distances.

# Relational Operators

*Relational operators*

<      >=  
>      <=  
==      !=

are used to compare numbers and char.

ถูกใช้เพื่อเปรียบเทียบตัวเลขและ char

# Relational Operators

Table 1 Relational Operators

C++	Math Notation	Description
>	>	Greater than
>=	$\geq$	Greater than or equal
<	<	Less than
<=	$\leq$	Less than or equal
==	=	Equal
!=	$\neq$	Not equal



# Relational Operators

## SYNTAX 3.2 Comparisons

These quantities are compared.

```
floor > 13
```

One of: == != < <= > >=

Check that you have the right direction: > (greater) or < (less)

Check the boundary condition: Do you want to include (>=) or exclude (>)?

```
floor == 13
```

Checks for equality.

Use ==, not =.

```
string input;  
if (input == "Y")
```




~~Ok to compare strings.~~

```
double x; double y; const double EPSILON = 1E-14;  
if (fabs(x - y) < EPSILON)
```

Checks that these floating-point numbers are very close.

# Relational Operators

**Table 2 Relational Operator Examples**

Expression	Value	Comment
$3 \leq 4$	true	3 is less than 4; $\leq$ tests for “less than or equal”.
 $3 \leq 4$	Error	The “less than or equal” operator is $\leq$ , not $\leq$ , with the “less than” symbol first.
$3 > 4$	false	$>$ is the opposite of $\leq$ .
$4 < 4$	false	The left-hand side must be strictly smaller than the right-hand side.
$4 \leq 4$	true	Both sides are equal; $\leq$ 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 $5 - 1$ .
 $3 = 6 / 2$	Error	Use $==$ to test for equality.
$1.0 / 3.0 == 0.3333333333$	false	Although the values are very close to one another, they are not exactly equal.
 $"10" > 5$	Error	You cannot compare strings and numbers.

# Relational Operators – Some Notes

Computer keyboards ไม่มี keys ต่อไปนี้:

$\geq$

$\leq$

$\neq$

แต่ใช้แบบนี้แทนได้:

$>=$

$<=$

$!=$

# Relational Operators – Some Notes

เราอาจจะสับสนกับ `== operator` ในตอนแรก ๆ

ใน Java, `=` มีความหมายแล้ว ไม่ได้แปลว่าเปรียบเทียบ, แปลว่าการกำหนดค่าให้ตัวแปรซ้ายมือ

The `== operator` หมายถึงการเปรียบเทียบว่าเท่ากันมั้ย:

```
floor = 13; // Assign 13 to floor
// ทดสอบว่า floor เท่ากับ 13 มั้ย
if (floor == 13)
```

You can compare char as well:

```
if (input == 'Q') ...
```

# Common Error – Confusing = and ==

---

ภาษา Java ไม่ยอมให้ใช้ = ในเงื่อนไขของ if.

Java จะแจ้ง error ถ้าใช้ = ในเงื่อนไขของ if

# Common Error – Confusing = and ==

```
floor == floor - 1; // ERROR
```

statement นี้ test ว่า **floor** เท่ากับ **floor - 1** หรือไม่

มันไม่ได้มีความหมายอะไรในทางโปรแกรมเพราะมันให้ค่า **false** ตลอดเวลา,  
แต่มันก็ไม่ใช่ **compiler error, compiler** จะไม่แจ้งเรา

# Common Error – Confusing = and ==

จำว่า:

ใช่ == กับเงื่อนไขใน *if*.

ใช่ = กับที่ไม่ใช่เงื่อนไขใน *if*.

# Kinds of Error Messages

---

There are two kinds of errors:

Warnings

Errors



# Kinds of Error Messages

---

- Error messages are fatal.
  - The compiler will not translate a program with one or more errors.
- Warning messages are advisory.
  - The compiler will translate the program, but there is a good chance that the program will not do what you expect it to do.

# Kinds of Error Messages

---

It is a good idea to learn how to activate warnings in your compiler.

It as a great idea to write code that emits no warnings at all.

# Kinds of Error Messages

---

We stated there are two kinds of errors.

Actually there's only one kind:

The ones you must read  
(that's all of them!)

# Kinds of Error Messages

---

Read all comments and deal with them.

If you understand a warning, and understand why it is happening, and you don't care about that reason

- Then, and only then, should you ignore a warning.

and, of course,  
you can't ignore an error message!

# Common Error – Exact Comparison of Floating-Point Numbers

---

## *Round off errors*

Floating-point numbers have only a limited precision.  
Calculations can introduce roundoff errors.

# Common Error – Exact Comparison of Floating-Point Numbers

---

*Roundoff errors*

Does  $(\sqrt{r})^2 == 2$  ?

Let's see (by writing code, of course) ...

# Common Error – Exact Comparison of Floating-Point Numbers

```
double r = Math.sqrt(2.0);  
if (r * r == 2) {  
    System.out.println("sqrt(2) squared is 2");  
} else {  
    System.out.print("sqrt(2) squared isnt 2 but ");  
    System.out.format("%.18f\n", r * r);  
}
```

roundoff error



This program displays:

```
sqrt(2) squared is not 2 but 2.000000000000000044
```

# Common Error – Exact Comparison of Floating-Point Numbers

---

*Roundoff errors – a solution*

Close enough will do.

$$|x - y| < \varepsilon$$



# Common Error – Exact Comparison of Floating-Point Numbers

---

Mathematically, we would write that  $x$  and  $y$  are close enough if for a very small number,  $\varepsilon$ :

$$|x - y| < \varepsilon$$

$\varepsilon$  is the Greek letter epsilon, a letter used to denote a very small quantity.

# Common Error – Exact Comparison of Floating-Point Numbers

It is common to set  $\epsilon$  to  $10^{-14}$  when comparing `double` numbers:

```
final double EPSILON = 1E-14;
double r = sqrt(2.0);
if (fabs(r * r - 2) < EPSILON)
{
    System.out.println("sqrt(2) squared is
                        approximately 2");
}
```

# หลายทางเลือก (Multiple Alternatives)



if it's quicker to the candy mountain,  
we'll go that way  
else  
we go that way  
*but what about that way?*

# Multiple Alternatives

---

**if** statements หลาย ๆ statement สามารถถูกเอามา  
รวมกันเพื่อการตัดสินใจที่ซับซ้อน

# Multiple Alternatives

---

ตัวอย่าง: เราจะเขียนโค้ดจัดการกับค่า รีกเตอร์ อย่างไร

# Multiple Alternatives

**Table 3 Richter Scale**

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



# Multiple Alternatives

ในกรณีนี้ เราแตกใน 5 branch (5 สาขา หรือ 5 กรณี):

แต่ละกรณี ก็มีคำอธิบายความเสียหายต่าง ๆ กัน

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

และก็มีกรณีที่ไม่มี ความเสียหาย

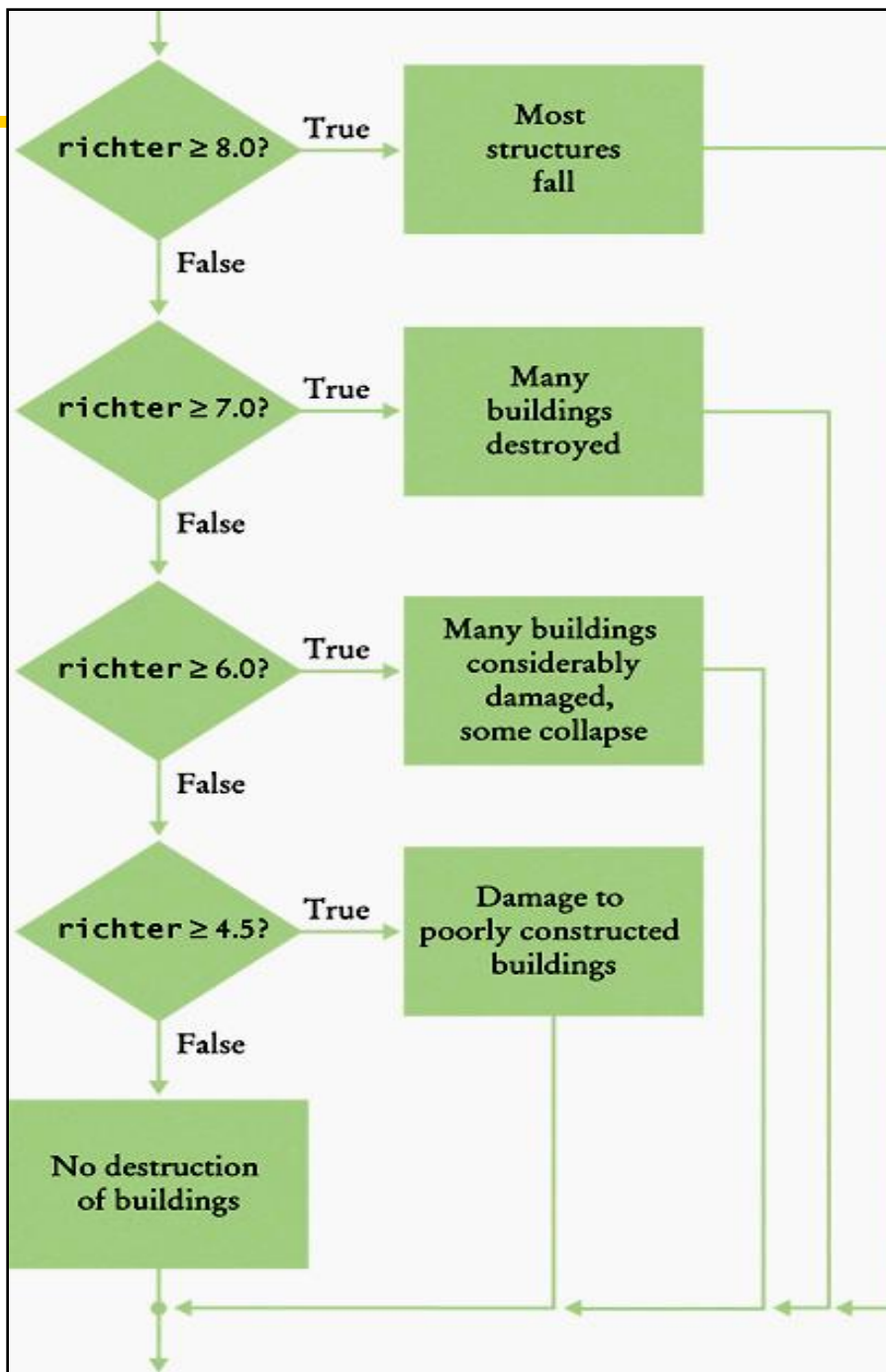
# Multiple Alternatives

---

เราอาจใช้ **if statements** > หลาย ๆ **statement** เพื่อเขียนโปรแกรมที่ต้องการหลายทางเลือก



# Richter flowchart



# Multiple Alternatives

```
if (richter >= 8.0) {
    System.out.println("Most structures fall");
} else if (richter >= 7.0) {
    System.out.println("Many buildings destroyed");
} else if (richter >= 6.0) {
    System.out.println("Many buildings considerably damaged, some
                        collapse");
} else if (richter >= 4.5) {
    System.out.println("Damage to poorly constructed buildings");
} else {
    System.out.println("No destruction of buildings");
}
. . .
```

# Multiple Alternatives

```
if (richter >= 8.0) ← If a test is false,
{
    System.out.println("Most structures fall");
}
else if (richter >= 7.0)
{
    System.out.println("Many buildings destroyed");
}
else if (richter >= 6.0) {
    System.out.println("Many buildings considerably damaged, some
                        collapse");
}
else if (richter >= 4.5)
{
    System.out.println("Damage to poorly constructed buildings");
}
else
{
    System.out.println("No destruction of buildings");
}
. . .
```

# Multiple Alternatives

```
if ( false ) ← If a test is false,
{
    System.out.println("Most structures fall");
}
else if (richter >= 7.0)
{
    System.out.println("Many buildings destroyed");
}
else if (richter >= 6.0) {
    System.out.println("Many buildings considerably damaged, some
                        collapse");
}
else if (richter >= 4.5)
{
    System.out.println("Damage to poorly constructed buildings");
}
else
{
    System.out.println("No destruction of buildings");
}
. . .
```

# Multiple Alternatives

```
if (richter >= 8.0)
```

```
{  
    System.out.println("Most structures fall");  
}
```

If a test is false,  
that block is skipped



```
else if (richter >= 7.0)
```

```
{  
    System.out.println("Many buildings destroyed");  
}
```

```
else if (richter >= 6.0) {
```

```
    System.out.println("Many buildings considerably damaged, some  
                        collapse");  
}
```

```
else if (richter >= 4.5)
```

```
{  
    System.out.println("Damage to poorly constructed buildings");  
}
```

```
else
```

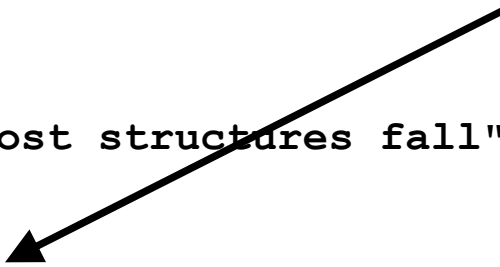
```
{  
    System.out.println("No destruction of buildings");  
}
```

```
...
```

# Multiple Alternatives

If a test is false,  
that block is skipped and  
the next test is made.

```
if (richter >= 8.0)
{
    System.out.println("Most structures fall");
}
else if (richter >= 7.0)
{
    System.out.println("Many buildings destroyed");
}
else if (richter >= 6.0) {
    System.out.println("Many buildings considerably damaged, some
        collapse");
}
else if (richter >= 4.5)
{
    System.out.println("Damage to poorly constructed buildings");
}
else
{
    System.out.println("No destruction of buildings");
}
. . .
```



# Multiple Alternatives

```
if (richter >= 8.0)
{
    System.out.println("Most structures fall");
}
else if (richter >= 7.0)
{
    System.out.println("Many buildings destroyed");
}
else if (richter >= 6.0) {
    System.out.println("Many buildings considerably damaged, some
                        collapse");
}
else if (richter >= 4.5)
{
    System.out.println("Damage to poorly constructed buildings");
}
else
{
    System.out.println("No destruction of buildings");
}
. . .
```

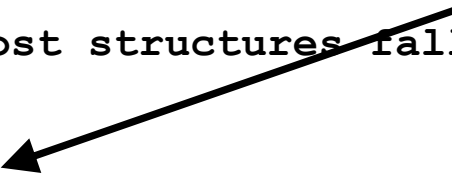
**As soon as one of the  
four tests succeeds,**



# Multiple Alternatives

```
if (richter >= 8.0)
{
    System.out.println("Most structures fall");
}
else if ( true )
{
    System.out.println("Many buildings destroyed");
}
else if (richter >= 6.0) {
    System.out.println("Many buildings considerably damaged, some
        collapse");
}
else if (richter >= 4.5)
{
    System.out.println("Damage to poorly constructed buildings");
}
else
{
    System.out.println("No destruction of buildings");
}
. . .
```

As soon as one of the four tests succeeds,





# Multiple Alternatives

```
if (richter >= 8.0)
```

```
{
```

```
    System.out.println("Most structures fall");
```

```
}
```

```
else if (richter >= 7.0)
```

```
{
```

```
    System.out.println("Many buildings destroyed");
```

```
}
```

```
else if (richter >= 6.0) {
```

```
    System.out.println("Many buildings considerably damaged, some  
collapse");
```

```
}
```

```
else if (richter >= 4.5)
```

```
{
```

```
    System.out.println("Damage to poorly constructed buildings");
```

```
}
```

```
else
```

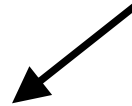
```
{
```

```
    System.out.println("No destruction of buildings");
```

```
}
```

```
...
```

As soon as one of the  
four tests succeeds,  
that block is executed,  
displaying the result,



# Multiple Alternatives

```
if (richter >= 8.0)
{
    System.out.println("Most structures fall");
}
else if (richter >= 7.0)
{
    System.out.println("Many buildings destroyed");
}
else if (richter >= 6.0) {
    System.out.println("Many buildings considerably damaged, some
                        collapse");
}
else if (richter >= 4.5)
{
    System.out.println("Damage to poorly constructed buildings");
}
else
{
    System.out.println("No destruction of buildings");
}
...

```

As soon as one of the four tests succeeds, that block is executed, displaying the result,

and no further tests are attempted.



## Multiple Alternatives – Wrong Order of Tests

---

Because of this execution order,  
when using multiple `if` statements,  
pay attention to the order of the conditions.

เพราะลำดับการ `execute` แบบนี้, เมื่อเราใช้ `if statement`  
หลาย ๆ `statement` ให้ระวังเรื่องการจัดลำดับของเงื่อนไข

# Multiple Alternatives – Wrong Order of Tests

```
if (richter >= 4.5)    // Tests in wrong order
{
    System.out.println("Damage to poorly constructed buildings");
}
else if (richter >= 6.0)
{
    System.out.println("Many buildings considerably damaged, some
                        collapse");
}
else if (richter >= 7.0)
{
    System.out.println("Many buildings destroyed");
}
else if (richter >= 8.0)
{
    System.out.println("Most structures fall");
}
. . .
```

# Multiple Alternatives – Wrong Order of Tests

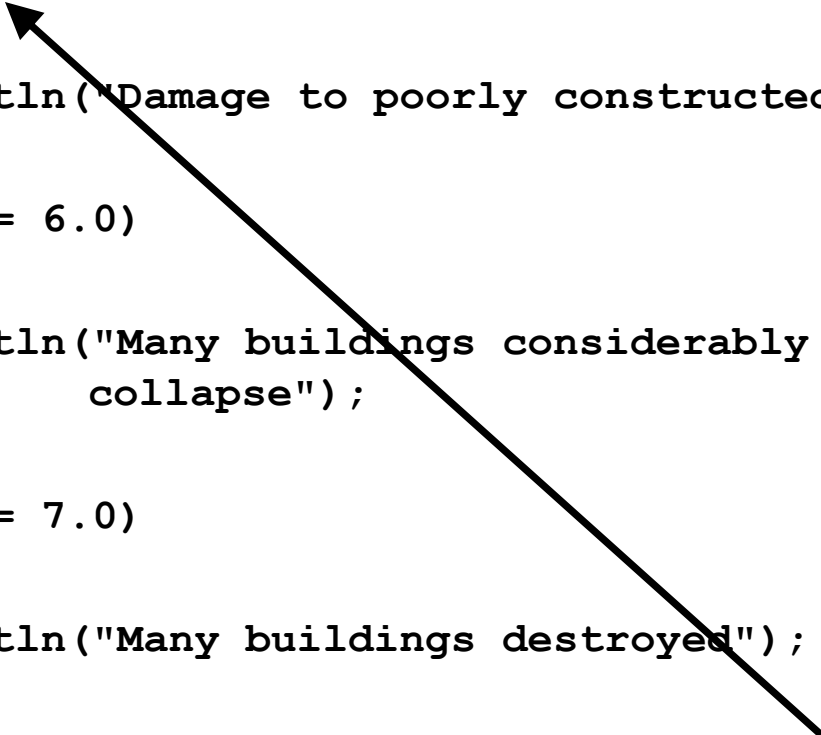
```
if (richter >= 4.5)    // Tests in wrong order
{
    System.out.println("Damage to poorly constructed buildings");
}
else if (richter >= 6.0)
{
    System.out.println("Many buildings considerably damaged, some
                        collapse");
}
else if (richter >= 7.0)
{
    System.out.println("Many buildings destroyed");
}
else if (richter >= 8.0)
{
    System.out.println("Most structures fall");
}
. . .
```

**Suppose the value  
of richter is 7.1,**

# Multiple Alternatives – Wrong Order of Tests

```
if (richter >= 4.5)    // Tests in wrong order
{
    System.out.println("Damage to poorly constructed buildings");
}
else if (richter >= 6.0)
{
    System.out.println("Many buildings considerably damaged, some
                        collapse");
}
else if (richter >= 7.0)
{
    System.out.println("Many buildings destroyed");
}
else if (richter >= 8.0)
{
    System.out.println("Most structures fall");
}
. . .
```

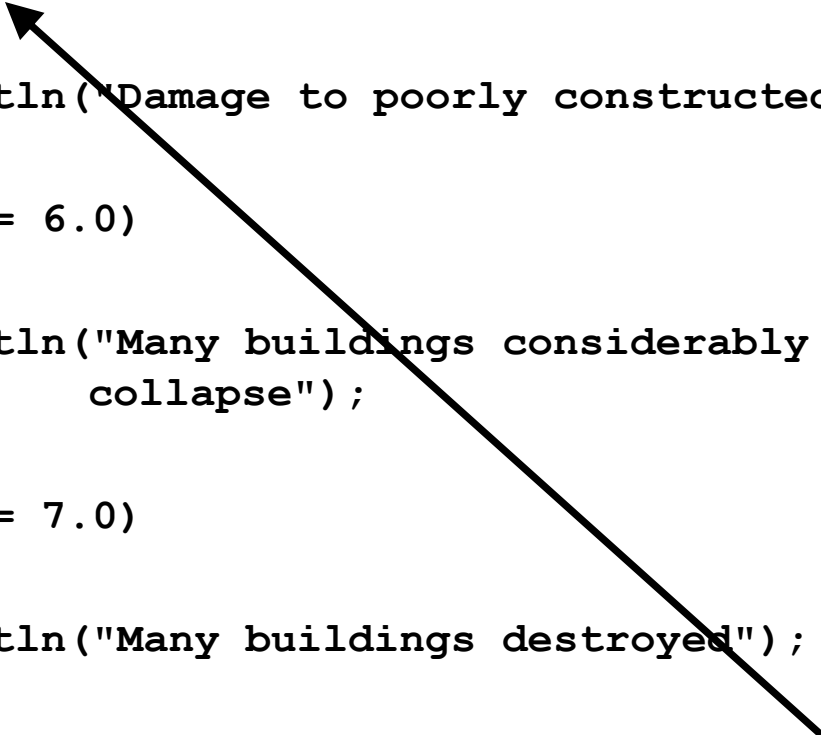
**Suppose the value  
of richter is 7.1,  
this test is true!**



# Multiple Alternatives – Wrong Order of Tests

```
if ( true ) // Tests in wrong order
{
    System.out.println("Damage to poorly constructed buildings");
}
else if (richter >= 6.0)
{
    System.out.println("Many buildings considerably damaged, some
        collapse");
}
else if (richter >= 7.0)
{
    System.out.println("Many buildings destroyed");
}
else if (richter >= 8.0)
{
    System.out.println("Most structures fall");
}
. . .
```

**Suppose the value  
of richter is 7.1,  
this test is true!**



# Multiple Alternatives – Wrong Order of Tests

```
if (richter >= 4.5)    // Tests in wrong order
```

```
{  
    System.out.println("Damage to poorly constructed buildings");  
}
```

```
else if (richter >= 6.0)
```

```
{  
    System.out.println("Many buildings considerably damaged, some  
                        collapse");  
}
```

```
else if (richter >= 7.0)
```

```
{  
    System.out.println("Many buildings destroyed");  
}
```

```
else if (richter >= 8.0)
```

```
{  
    System.out.println("Most structures fall");  
}
```

```
...
```

**Suppose the value  
of richter is 7.1,  
this test is true!**

**and that block is  
executed (Oh no!),**



# Multiple Alternatives – Wrong Order of Tests

```
if (richter >= 4.5)    // Tests in wrong order
{
    System.out.println("Damage to poorly constructed buildings");
}
else if (richter >= 6.0)
{
    System.out.println("Many buildings considerably damaged, some
                        collapse");
}
else if (richter >= 7.0)
{
    System.out.println("Many buildings destroyed");
}
else if (richter >= 8.0)
{
    System.out.println("Most structures fall");
}
....
```

**Suppose the value  
of richter is 7.1,  
this test is true!**

**and that block is  
executed (Oh no!),**

**and we go...**



# The switch Statement

โค้ดข้างล่างอ่านยากมาก

```
int digit;
...
if (digit == 1) { digit_name = "one"; }
else if (digit == 2) { digit_name = "two"; }
else if (digit == 3) { digit_name = "three"; }
else if (digit == 4) { digit_name = "four"; }
else if (digit == 5) { digit_name = "five"; }
else if (digit == 6) { digit_name = "six"; }
else if (digit == 7) { digit_name = "seven"; }
else if (digit == 8) { digit_name = "eight"; }
else if (digit == 9) { digit_name = "nine"; }
else { digit_name = ""; }
```

# The `switch` Statement

Java มี `statement` ที่ช่วยให้โค้ดแบบตะกั่วอ่านง่ายขึ้น:

The `switch` statement.

ONLY a sequence of `if` statements that compares a single integer value against several constant alternatives can be implemented as a `switch` statement.

ลำดับของ `if statement` หลาย `statement` ที่เปรียบเทียบตัวแปรหนึ่งกับค่าหลาย ๆ กรณี  
เท่านั้นที่จะสามารถใช้ `switch statement` ได้

# The switch Statement

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

# Nested Branches

---

เป็นไปได้ที่จะมีหลายๆ กรณี ที่โปรแกรมจะทำงานแบบเดียวกัน (หรือที่จะไป **branch** เดียวกัน)

```
case 1: case 3: case 5: case 7: case 9: odd = true; break;
```

The **default**: branch ที่ถูกเลือก ในกรณีที่ไม่มีกรณีใดถูกเลือกแล้ว

# Nested Branches

ทุก ๆ กรณีของ **switch** ต้องมี **break statement**.

ถ้า **break** หายไป, โปรแกรมจะทำงานลงไปที่กรณีอื่นด้วย, จนกว่าจะเจอ **break** หรือ  
สิ้นสุด `block` ของ **switch**.

ในทางปฏิบัติ, การที่ภาษาเป็นแบบนี้ไม่ค่อยมีประโยชน์ และอาจเกิด **error** ขึ้นได้ง่าย

เพราะถ้าเราลืม **break statement**, โปรแกรมของเราก็จะไปทำงานในส่วนที่เราไม่ต้องการ

# Nested Branches

---

โปรแกรมเมอร์ส่วนใหญ่มองว่า **switch statement** ค่อนข้างอันตราย และชอบใช้ **if statement** มากกว่า

# Nested Branches – ภาษี (Taxes)

---



Taxes...



# Nested Branches – Taxes

มีอะไรอีกหลังจาก บรรทัดที่ 37?



Taxes...

# Nested Branches – Taxes

มีอะไรอีกหลังจาก บรรทัดที่ 37?

... ถ้าจำนวนที่คิดภาษีได้จากบรรทัดที่  
22 มากกว่า บรรทัดที่ 83 ...



Taxes...

# Nested Branches – Taxes



มีอะไรอีกหลังจาก บรรทัดที่ 37?

... ถ้าจำนวนที่คิดภาษีได้จากบรรทัดที่  
22 มากกว่า บรรทัดที่ 83 ...

... และชั้นมีลูก 3 คน ต่ำกว่า 13  
ขวบ ...

Taxes...

# Nested Branches – Taxes



มีอะไรอีกหลังจาก บรรทัดที่ 37?

... ถ้าจำนวนที่คิดภาษีได้จากบรรทัดที่ 22 มากกว่า บรรทัดที่ 83 ...

... และชั้นมีลูก 3 คน ต่ำกว่า 13

... ยกเว้นถ้าฉันได้แต่งงานแล้ว ...

Taxes...

# Nested Branches – Taxes



มีอะไรอีกหลังจาก บรรทัดที่ 37?

... ถ้าจำนวนที่คิดภาษีได้จากบรรทัดที่ 22 มากกว่า บรรทัดที่ 83 ...

... และชั้นมีลูก 3 คน ต่ำกว่า 13

... ยกเว้นถ้าฉันได้แต่งงานแล้ว ...

**เอ แล้วฉันแต่งงานไปหรือยังเนีย?!**

Taxes...

# Nested Branches – Taxes

- ในสหรัฐ เปอร์เซนต์การจ่ายภาษีขึ้นกับสถานะการแต่งงาน
- มีตารางภาษีแบบต่าง ๆ สำหรับผู้เสียภาษีที่ยังไม่ได้แต่งงานแล้ว
- ผู้เสียภาษีที่แต่งงานแล้วจะรวมรายได้เข้าไว้ด้วยกันและคิดภาษีรวมกัน  
ไป

# Nested Branches – Taxes

---

เรามาเริ่มเขียนโค้ดกัน

เหมือนเคย เราเริ่มที่วิเคราะห์ปัญหาก่อน

## Nested Branches – Taxes

---

Nested branching analysis is aided by drawing tables showing the different criteria.

Thankfully, the I.R.S. has done this for us.

การวิเคราะห์ “เงื่อนไขซ้อนกัน (Nested branch)” จะง่ายขึ้น  
ถ้าเราเขียนตารางต่าง ๆ ขึ้นมาก่อน

โชคดีที่ **I.R.S** ทำให้แล้ว



# Nested Branches – Taxes

**Table 4 Federal Tax Rate Schedule**

<b>If your status is Single and if the taxable income is over</b>	<b>but not over</b>	<b>the tax is</b>	<b>of the amount over</b>
\$0	\$32,000	10%	\$0
\$32,000		\$3,200 + 25%	\$32,000
<b>If your status is Married and if the taxable income is over</b>	<b>but not over</b>	<b>the tax is</b>	<b>of the amount over</b>
\$0	\$64,000	10%	\$0
\$64,000		\$6,400 + 25%	\$64,000

Tax brackets for single filers:  
 from \$0 to \$32,000  
 above \$32,000  
 then tax depends on income

Tax brackets for married filers:  
 from \$0 to \$64,000  
 above \$64,000  
 then tax depends on income

## Nested Branches – Taxes

---

ตอนนี้เราก็เข้าใจการคิดภาษีของสหรัฐแล้ว,  
กำหนดสถานะการแต่งงานและตัวเลขรายได้มาให้,  
ลองคำนวณภาษี

# Nested Branches – Taxes



**ARGHHHH!!!!**

# Nested Branches – Taxes

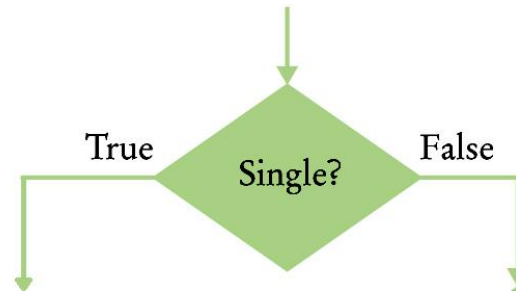
---

- หลักสำคัญตรงนี้คือ มีระดับการตัดสินใจสองระดับ.

Really, only two (at this level).

# Nested Branches – Taxes

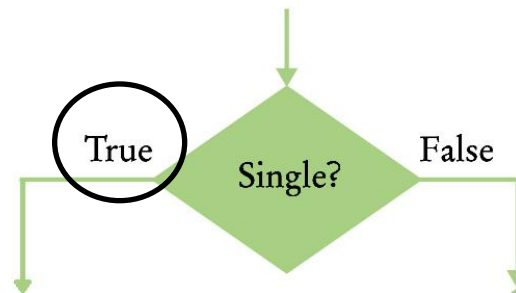
ตอนแรก, เราต้องตัดสินใจสถานะการแต่งงาน



# Nested Branches – Taxes

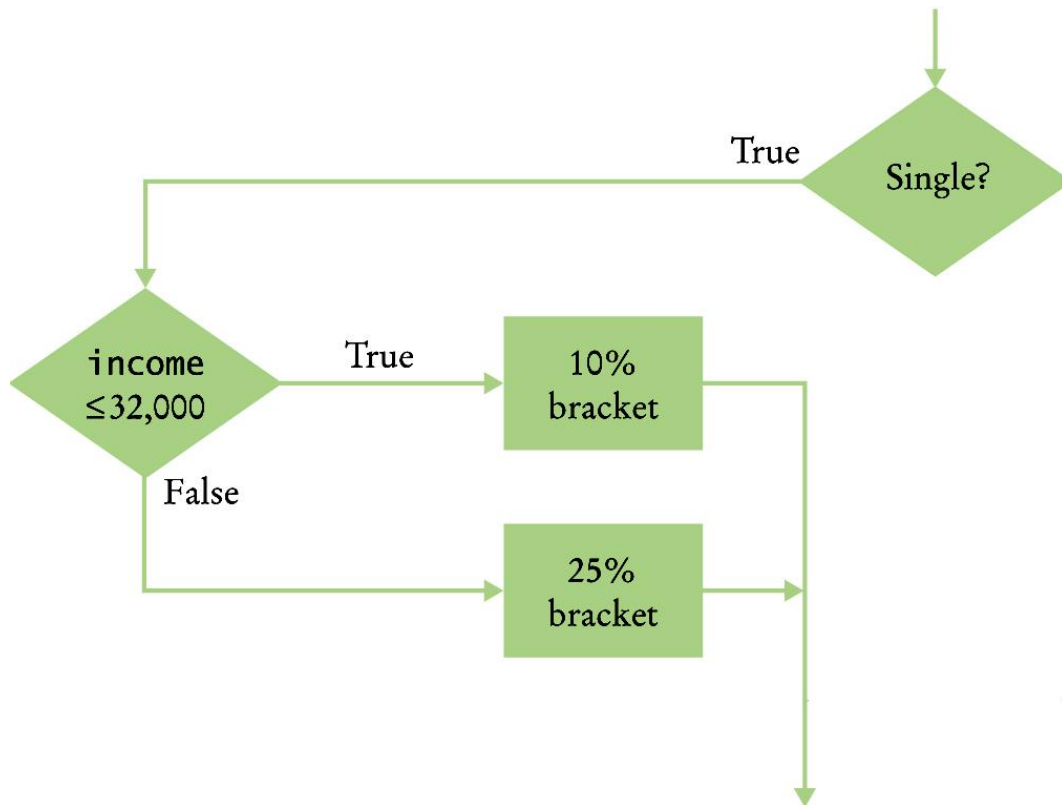
จากนั้น, สำหรับแต่ละสถานะการแต่งงาน,  
เราต้องตัดสินใจเกี่ยวกับกรณีของรายได้

ถ้าเป็นคนโสด ...



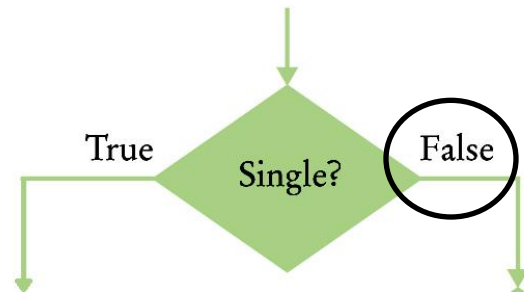
# Nested Branches – Taxes

คนโสดจะมี *nested if* statement ของตัวเอง  
เพื่อตัดสินใจเรื่องรายได้



# Nested Branches – Taxes

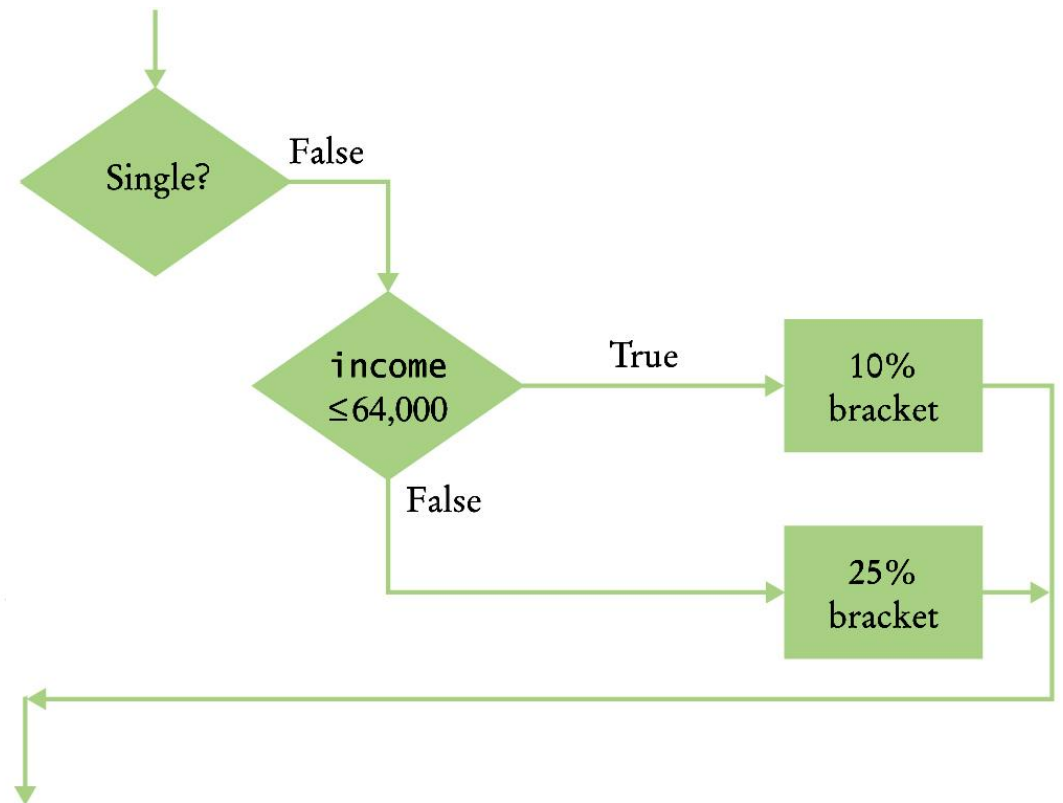
ถ้าแต่งงานแล้ว ...





# Nested Branches – Taxes

จะใช้ *nested if* ที่ต่างออกไปสำหรับตรวจสอบเงื่อนไขรายได้



# Nested Branches – Taxes

ในทางทฤษฎี เราสามารถซ้อนกันกี่ระดับก็ได้

เช่น:

ตอนแรก เปรียบเทียบ รัฐ

จากนั้น เปรียบเทียบ สถานะการแต่งงาน

จากนั้น เปรียบเทียบ ระดับรายได้

จะเห็นได้ว่า เราทำซ้อนกัน 3 ระดับ

# Nested Branches – Taxes

```
public static void main(String [] args) {
    final double RATE1 = 0.10;
    final double RATE2 = 0.25;
    final double RATE1_SINGLE_LIMIT = 32000;
    final double RATE1_MARRIED_LIMIT = 64000;

    Scanner sc_obj = new Scanner(System.in);

    double tax1 = 0, tax2 = 0;

    double income;
    char marital_status;

    System.out.print("Please enter your income: ");
    income = sc_obj.nextDouble();

    System.out.print("Enter s for single, m for married: ");
    marital_status = sc_obj.next().charAt(0);
```

# Nested Branches – Taxes

```
if (marital_status == 's')
{
    if (income <= RATE1_SINGLE_LIMIT)
    {
        tax1 = RATE1 * income;
    }
    else
    {
        tax1 = RATE1 * RATE1_SINGLE_LIMIT;
        tax2 = RATE2 * (income - RATE1_SINGLE_LIMIT);
    }
}
else
```

# Nested Branches – Taxes

```
{
    if (income <= RATE1_MARRIED_LIMIT)
    {
        tax1 = RATE1 * income;
    }
    else
    {
        tax1 = RATE1 * RATE1_MARRIED_LIMIT;
        tax2 = RATE2 * (income - RATE1_MARRIED_LIMIT);
    }
}

double total_tax = tax1 + tax2;

System.out.println("The tax is " + total_tax);
}
```

# Nested Branches – Taxes

In practice two levels of nesting should be enough.  
Beyond that you should be calling your own functions.

– But, you don't know to write functions...

...yet

ในทางปฏิบัติ ซ้อนกัน 2 ระดับก็พอแล้ว ถ้าซ้อนกันมากกว่านั้น เราควรเขียน **function** ขึ้นมา

- แต่เรายังไม่ได้เรียนวิธีเขียน **function** กัน

# Hand-Tracing (การไล่โปรแกรมด้วยมือ)

A very useful technique for understanding whether a program works correctly is called *hand-tracing*.

You simulate the program's activity on a sheet of paper.

You can use this method with pseudocode or Java code.

ในการที่เราจะเข้าใจว่า โปรแกรมทำงานถูกมั้ย เราอาจไล่โปรแกรมของเราด้วยมือได้ โดยเราไล่การทำงานของโปรแกรมลงในกระดาษ (อาจจะใช้ **pseudocode** หรือ ใช้ **code Java** จริง เลยก็ได้)

# Hand-Tracing (การไล่โปรแกรมด้วยมือ)

---

- Depending on where you normally work, get:



# Hand-Tracing

---

- Depending on where you normally work, get:
  - an index card

# Hand-Tracing

---

- Depending on where you normally work, get:
  - an index card
  - an envelope

# Hand-Tracing

---

- Depending on where you normally work, get:
  - an index card
  - an envelope (use the back)

# Hand-Tracing

---

- Depending on where you normally work, get:
  - an index card
  - an envelope (use the back)
  - a cocktail napkin

# Hand-Tracing

---

- กระดาษจะเป็นอะไรก็ได้
  - การ์ดนามบัตร
  - ด้านหลังของซองจดหมาย
  - กระดาษทิชชู

(!)

# Hand-Tracing

คู่มือที่ pseudocode หรือ Java code,

- ใช้ marker, เช่น a paper clip, (หรือ ไม้จิ้มฟัน) เพื่อ mark statement ขณะนั้นไว้
- “Execute หรือ ทำ” statement ทีละ statement
- ทุก ๆ ครั้งที่ค่าในตัวแปรเปลี่ยนแปลง, ฆ่าค่าเก่าทิ้ง และ เขียนค่าใหม่ไว้ด้านล่างค่าเก่า

# Hand-Tracing

---

Let's do this with the tax program.

(take those cocktail napkins out of your pockets and get started!)

# Hand-Tracing

```
public static void main(String [] args) {  
    final double RATE1 = 0.10;  
    final double RATE2 = 0.25;  
    final double RATE1_SINGLE_LIMIT = 32000;  
    final double RATE1_MARRIED_LIMIT = 64000;  
  
    Scanner sc_obj = new Scanner(System.in);
```

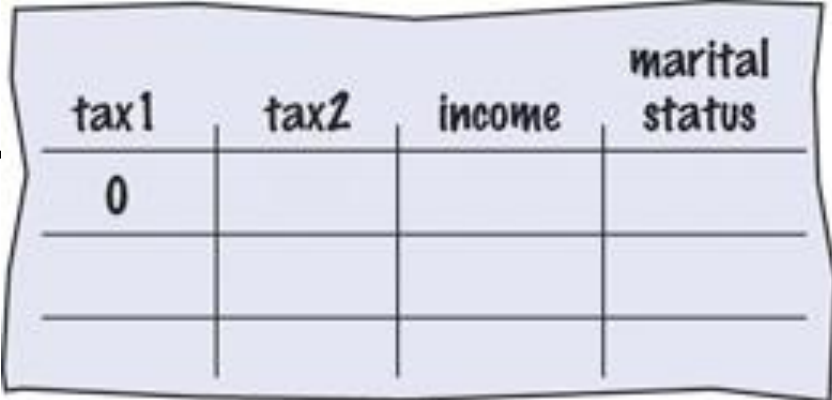
ค่าคงที่ จะไม่เปลี่ยนไปตอนโปรแกรมทำงาน

ดังนั้นเราไม่ต้องเขียนมันตอนไล่ code ก็ได้



# Hand-Tracing

```
public static void main(String [] args) {  
    final double RATE1 = 0.10;  
    final double RATE2 = 0.25;  
    final double RATE1_SINGLE_LIMIT = 32000;  
    final double RATE1_MARRIED_LIMIT = 64000;  
    Scanner sc_obj = new Scanner(System.in);  
  
    double tax1 = 0;  
    double tax2 = 0;
```



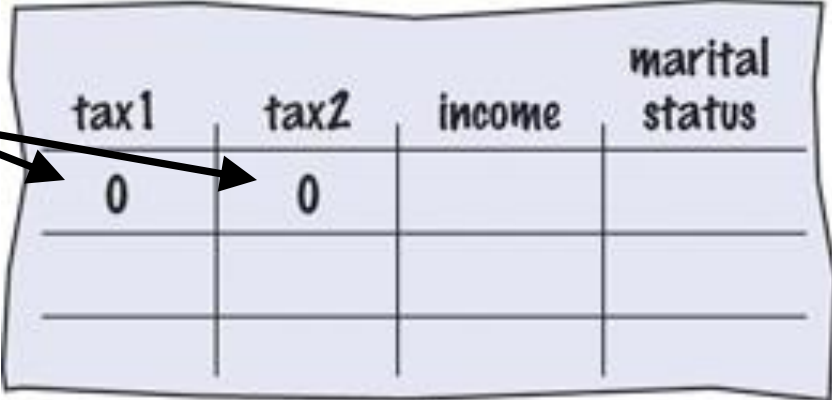
tax1	tax2	income	marital status
0			

# Hand-Tracing

```
public static void main(String [] args) {  
    final double RATE1 = 0.10;  
    final double RATE2 = 0.25;  
    final double RATE1_SINGLE_LIMIT = 32000;  
    final double RATE1_MARRIED_LIMIT = 64000;  
    Scanner sc_obj = new Scanner(System.in);
```

```
double tax1 = 0;
```

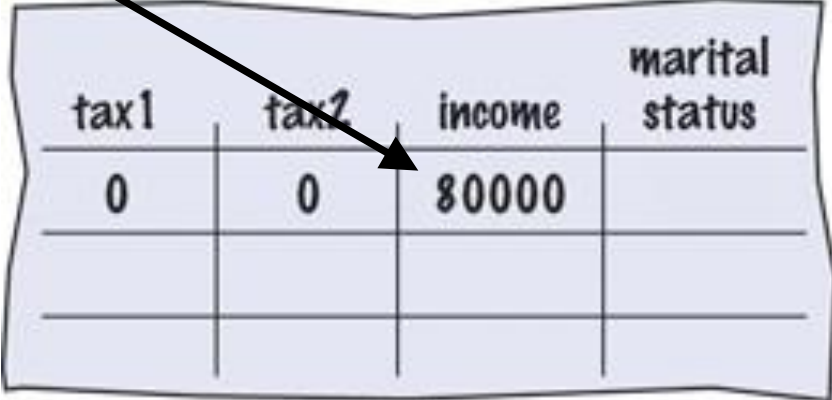
```
double tax2 = 0;
```



tax1	tax2	income	marital status
0	0		

# Hand-Tracing

```
double income;  
System.out.println("Please enter your income: ");  
income = sc_obj.nextInt();
```

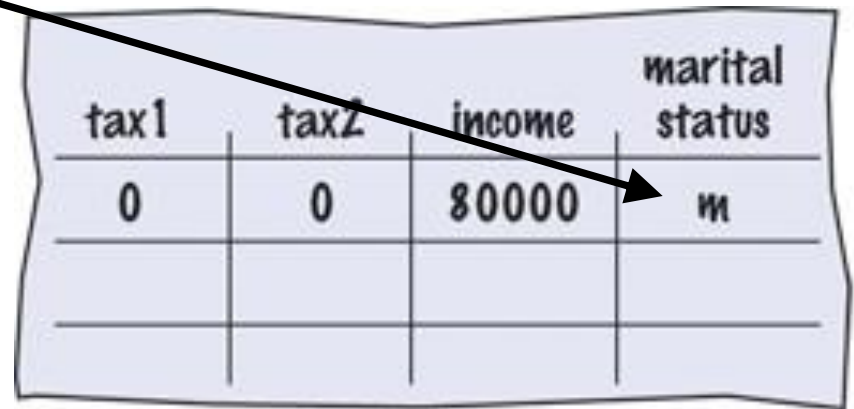


tax1	tax2	income	marital status
0	0	80000	

The user typed 80000.

# Hand-Tracing

```
double income;  
System.out.print("Please enter your income: ");  
income = sc_obj.nextInt();  
  
System.out.print("Enter s for single, m for married: ");  
char marital_status;  
marital_status = sc_obj.next().charAt(0);
```



tax1	tax2	income	marital status
0	0	80000	m

The user typed **m**

# Hand-Tracing

tax1	tax2	income	marital status
0	0	80000	m

```
if (marital_status == 's')
```

```
{  
    if (income <= RATE1_SINGLE_LIMIT)  
    {  
        tax1 = RATE1 * income;  
    }  
    else  
    {  
        tax1 = RATE1 * RATE1_SINGLE_LIMIT;  
        tax2 = RATE2 * (income - RATE1_SINGLE_LIMIT);  
    }  
}
```

```
else
```

# Hand-Tracing

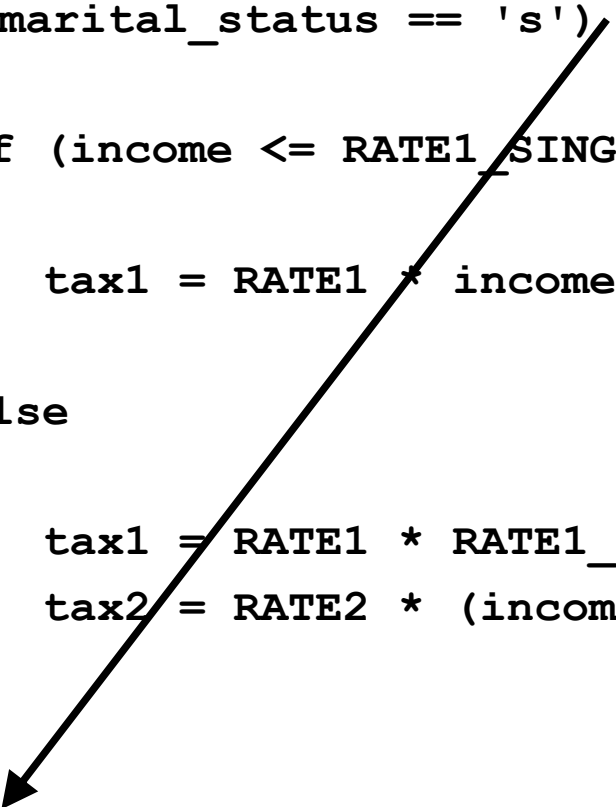
tax1	tax2	income	marital status
0	0	80000	m

```
if ( false )
{
    if (income <= RATE1_SINGLE_LIMIT)
    {
        tax1 = RATE1 * income;
    }
    else
    {
        tax1 = RATE1 * RATE1_SINGLE_LIMIT;
        tax2 = RATE2 * (income - RATE1_SINGLE_LIMIT);
    }
}
else
```

# Hand-Tracing

tax1	tax2	income	marital status
0	0	80000	m

```
if (marital_status == 's')
{
    if (income <= RATE1_SINGLE_LIMIT)
    {
        tax1 = RATE1 * income;
    }
    else
    {
        tax1 = RATE1 * RATE1_SINGLE_LIMIT;
        tax2 = RATE2 * (income - RATE1_SINGLE_LIMIT);
    }
}
else
```



# Hand-Tracing

tax1	tax2	income	marital status
0	0	80000	m

else

```
{
    if (income <= RATE1_MARRIED_LIMIT)
    {
        tax1 = RATE1 * income;
    }
    else
    {
        tax1 = RATE1 * RATE1_MARRIED_LIMIT;
        tax2 = RATE2 * (income - RATE1_MARRIED_LIMIT);
    }
}
```

```
double total_tax = tax1 + tax2;
```



# Hand-Tracing

tax1	tax2	income	marital status
0	0	80000	m

```
else
{
    if (income <= 64000 )
    {
        tax1 = RATE1 * income;
    }
    else
    {
        tax1 = RATE1 * RATE1_MARRIED_LIMIT;
        tax2 = RATE2 * (income - RATE1_MARRIED_LIMIT);
    }
}
double total_tax = tax1 + tax2;
```

# Hand-Tracing

tax1	tax2	income	marital status
0	0	80000	m

```
else
{
    if ( false )
    {
        tax1 = RATE1 * income;
    }
    else
    {
        tax1 = RATE1 * RATE1_MARRIED_LIMIT;
        tax2 = RATE2 * (income - RATE1_MARRIED_LIMIT);
    }
}
double total_tax = tax1 + tax2;
```

# Hand-Tracing

tax1	tax2	income	marital status
0	0	80000	m

```
else
```

```
{
```

```
    if (income <= RATE1_MARRIED_LIMIT)
```

```
    {
```

```
        tax1 = RATE1 * income;
```

```
    }
```

```
    else
```

```
    {
```

```
        tax1 = RATE1 * RATE1_MARRIED_LIMIT;
```

```
        tax2 = RATE2 * (income - RATE1_MARRIED_LIMIT);
```

```
    }
```

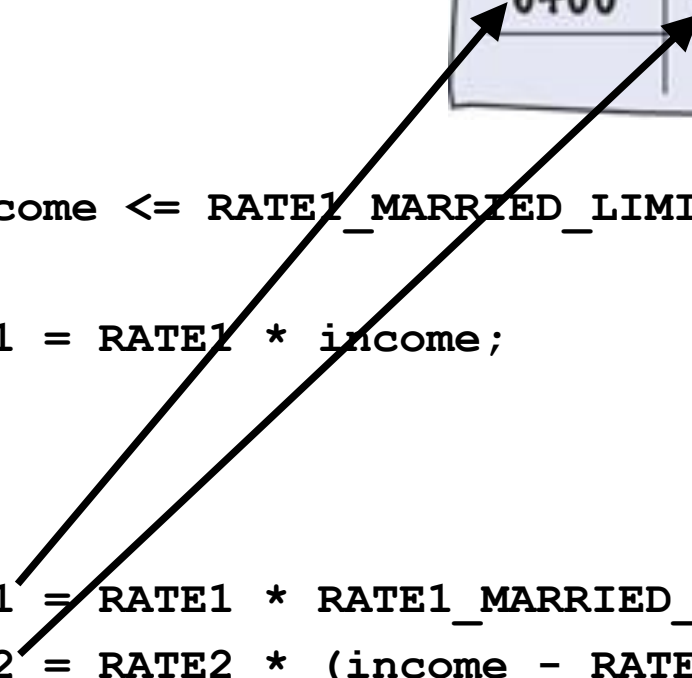
```
}
```

```
double total_tax = tax1 + tax2;
```

# Hand-Tracing

tax1	tax2	income	marital status
<del>0</del>	<del>0</del>	80000	m
6400	4000		

```
else
{
    if (income <= RATE1_MARRIED_LIMIT)
    {
        tax1 = RATE1 * income;
    }
    else
    {
        tax1 = RATE1 * RATE1_MARRIED_LIMIT;
        tax2 = RATE2 * (income - RATE1_MARRIED_LIMIT);
    }
}
double total_tax = tax1 + tax2;
```



# Hand-Tracing

tax1	tax2	income	marital status
<del>0</del>	<del>0</del>	80000	m
6400	4000		

```
else
{
    if (income <= RATE1_MARRIED_LIMIT)
    {
        tax1 = RATE1 * income;
    }
    else
    {
        tax1 = RATE1 * RATE1_MARRIED_LIMIT;
        tax2 = RATE2 * (income - RATE1_MARRIED_LIMIT);
    }
}
double total_tax = tax1 + tax2;
```

# Hand-Tracing

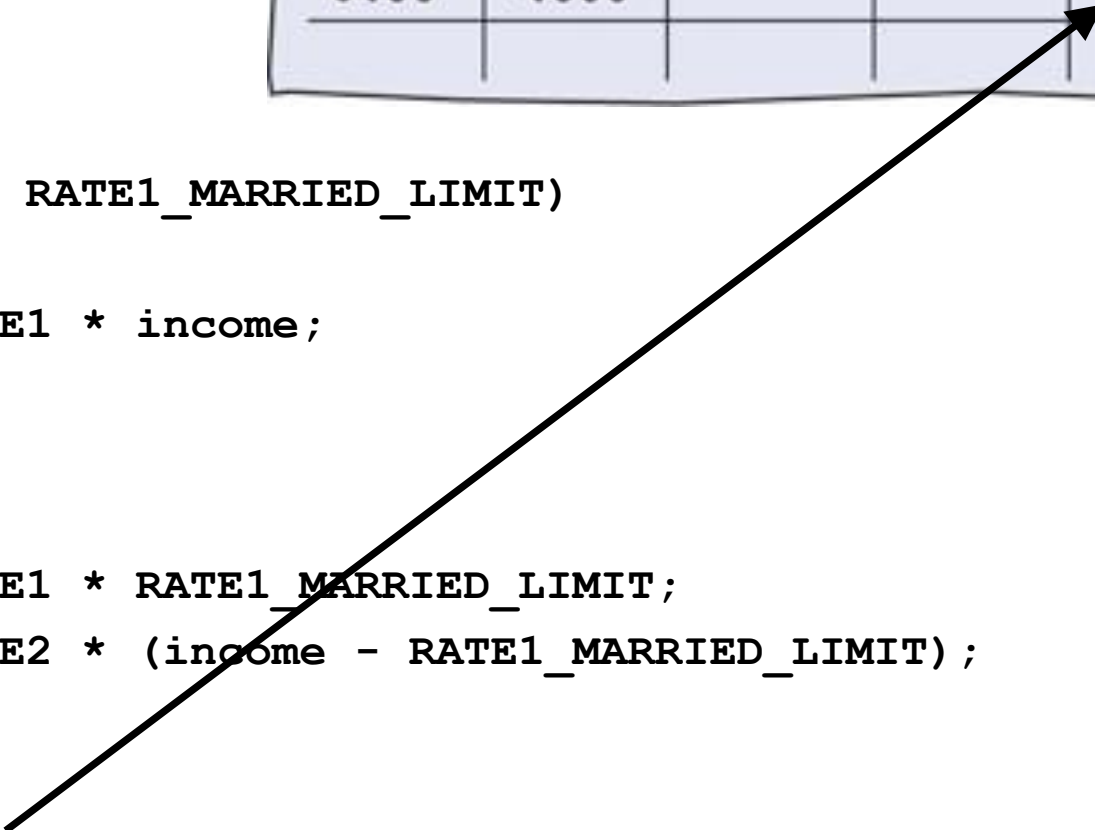
tax1	tax2	income	marital status
<del>0</del>	<del>0</del>	80000	m
6400	4000		

```
else
{
    if (income <= RATE1_MARRIED_LIMIT)
    {
        tax1 = RATE1 * income;
    }
    else
    {
        tax1 = RATE1 * RATE1_MARRIED_LIMIT;
        tax2 = RATE2 * (income - RATE1_MARRIED_LIMIT);
    }
}
↓
double total_tax = tax1 + tax2;
```

# Hand-Tracing

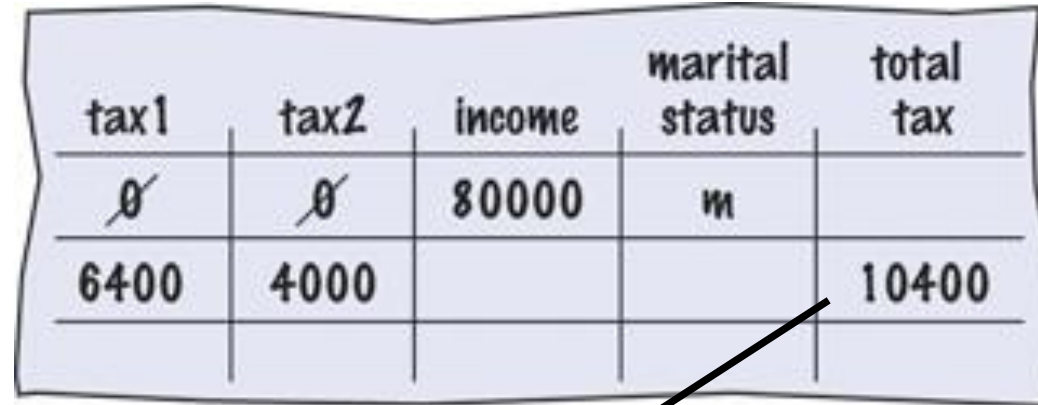
tax1	tax2	income	marital status	total tax
<del>0</del>	<del>0</del>	80000	m	
6400	4000			10400

```
else
{
    if (income <= RATE1_MARRIED_LIMIT)
    {
        tax1 = RATE1 * income;
    }
    else
    {
        tax1 = RATE1 * RATE1_MARRIED_LIMIT;
        tax2 = RATE2 * (income - RATE1_MARRIED_LIMIT);
    }
}
double total_tax = tax1 + tax2;
```



# Hand-Tracing

tax1	tax2	income	marital status	total tax
<del>0</del>	<del>0</del>	80000	m	
6400	4000			10400



```
double total_tax = tax1 + tax2;
```

```
System.out.println("The tax is " + total_tax);
```

```
return 0;
```

```
}
```



## Prepare Test Cases Ahead of Time

---

Consider how to *test* the tax computation program.

Of course, you cannot try out all possible inputs of filing status and income level.

Even if you could, there would be no point in trying them all.

## Prepare Test Cases Ahead of Time

---

If the program correctly computes one or two tax amounts in a given bracket, then we have a good reason to believe that all amounts will be correct.

You should also test on the *boundary conditions*, at the endpoints of each bracket

this tests the  $<$  vs.  $\leq$  situations.

## Prepare Test Cases Ahead of Time

---

There are two possibilities for the filing status and two tax brackets for each status, yielding four test cases.

- Test a handful of boundary conditions, such as an income that is at the boundary between two brackets, and a zero income.
- If you are responsible for error checking, also test an invalid input, such as a negative income.

# Prepare Test Cases Ahead of Time

---

Here are some possible test cases for the tax program:

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

# Prepare Test Cases Ahead of Time

---

It is always a good idea to design test cases *before* starting to code.

Working through the test cases gives you a better understanding of the algorithm that you are about to implement.











# The Dangling else Problem – The Solution

---

แล้วเราจะแก้ปัญหาดangling else อย่างไร

จำได้มั๊ยว่า เราเอา statement ใส่ไว้ใน block ได้



# Boolean Operators



Will we remember next time?  
I wish I could put the way to go in my pocket!

# Boolean Operators



At this geyser in Iceland, you can see ice, liquid water, and steam.

# Boolean Operators

- สมมติว่า เราจะเขียนโปรแกรมเพื่อประมวลผลค่าอุณหภูมิ, และเราต้องการ test ว่าค่านี้เกี่ยวกับน้ำอย่างไร
  - ที่ระดับน้ำทะเล, น้ำจะแข็งที่ 0 degrees Celsius and และเดือดที่ 100 degrees.
- น้ำจะเป็นของเหลว ถ้าอุณหภูมิมากกว่า 0 และ น้อยกว่า 100

# Boolean Operators

- เมื่อเงื่อนไขการตัดสินใจซับซ้อนขึ้น, บ่อยครั้งเราต้องรวมผลการเปรียบเทียบ (ที่เป็นจริงหรือเป็นเท็จ) เข้าไว้ด้วยกัน
- operator ที่รวมเงื่อนไขจริงเท็จ (เงื่อนไข Boolean) เรียกว่า Boolean operator.
- Boolean operators จะนำค่า Boolean หนึ่ง หรือสองค่า มาทำการประมวลผลเพื่อหาค่า Boolean ใหม่

## The Boolean Operator && (and)

ในภาษา Java, && operator (เรียกว่า *and*) ให้ค่าจริง **true** ก็ต่อเมื่อ เงื่อนไข *boolean* ทั้งสองเป็นจริง

```
if (temp > 0 && temp < 100)
{
    System.out.println("Liquid");
}
```

ถ้า **temp** อยู่ในช่วง 0 - 100, ค่า *boolean* ทั้งซ้าย และขวาของ && จะเป็นจริง, ทำให้ *expression* ทั้งหมดเป็นจริง (นั่นคือ && return ค่า *boolean* ที่เป็น **true**)

ถ้า **temp** ไม่ได้อยู่ในช่วง 0-100, *expression* ทั้งหมดจะให้ค่าเท็จ **false**.



# The Boolean Operator || (or)

|| operator (เรียกว่า *or*) ให้ผลลัพธ์เป็น **true** ถ้าเงื่อนไขอันใดอันหนึ่งเป็นจริง

- เขียนโดยใช้สัญลักษณ์ขีดสองตัวติดกัน

```
if (temp <= 0 || temp >= 100)
{
    System.out.println("Not liquid");
}
```

ถ้า *expression* อันใดอันหนึ่งไม่ว่าจะอันซ้ายหรือขวาเป็นจริง, *expression* ทั้งหมดจะเป็นจริง

“Not liquid” จะไม่ถูกพิมพ์ถ้าเงื่อนไขทั้งซ้ายและขวาเป็น **false**.

# The Boolean Operator ! (not)

บางครั้งเราต้องการจะกลับค่าเท็จเป็นจริง หรือ จริงเป็นเท็จ เราใช้ *not* operator.

! operator ใช้ค่า boolean ค่าเดียว และจะให้ค่าเป็น true ถ้าค่า boolean ข้างๆ มันเป็น false จะให้ค่าเป็น false ถ้าค่า boolean ข้างๆ มันเป็น true

```
if (!frozen) {  
    System.out.println("Not frozen");  
}
```

“Not frozen” จะถูกพิมพ์ออกมาเมื่อ frozen มีค่าเป็น 0

**! false is true.**

# Boolean Operators

สรุป *truth table*:

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


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

A	!A
true	false
false	true

where A and B เป็นค่าจาก Boolean expressions.

# Boolean Operators – Some Examples

Table 6 Boolean Operators

Expression	Value	Comment
<code>0 &lt; 200 &amp;&amp; 200 &lt; 100</code>	false	Only the first condition is true. Note that the < operator has a higher precedence than the && operator.
<code>0 &lt; 200    200 &lt; 100</code>	true	The first condition is true.
<code>0 &lt; 200    100 &lt; 200</code>	true	The    is not a test for “either-or”. If both conditions are true, the result is true.
 <code>0 &lt; 200 &lt; 100</code>	true	<b>Error:</b> The expression <code>0 &lt; 200</code> is true, which is converted to 1. The expression <code>1 &lt; 100</code> is true. You never want to write such an expression; see Common Error 3.5 on page 107.

# Boolean Operators – Some Examples



`-10 && 10 > 0`

`true`

**Error:** `-10` is not zero. It is converted to `true`. You never want to write such an expression; see Common Error 3.5 on page 107.

`0 < x && x < 100 || x == -1`

`(0 < x && x < 100)  
|| x == -1`

The `&&` operator has a higher precedence than the `||` operator.

`!(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 Error – Combining Multiple Relational Operators

พิจารณา expression ต่อไปนี้

```
if (0 <= temp <= 100)...
```

มันเหมือนกับอสมการทางคณิตศาสตร์:

$$0 \leq \text{temp} \leq 100$$

แต่ใน Java แล้ว Compiler จะแจ้ง error ออกมา

# Common Error – Combining Multiple Relational Operators

ความผิดพลาดที่พบบ่อยอีกอันคือ

```
if (x && y > 0) ... // Compile Error
```

แทนที่จะเป็น

```
if (x > 0 && y > 0) ...
```

(**x** and **y** are **ints**)

# Common Error – สับสนระหว่าง && และ | |

พิจารณาสถานะเหล่านี้ เพื่อจ่ายภาษี

สถานะเราจะเป็น โสดถ้าสิ่งเหล่านี้ข้อใดอย่างหนึ่งเป็นจริง:

- ไม่เคยแต่งงานมาก่อน
- แยกกันอยู่หรือหย่าอย่างถูกต้องตามกฎหมาย
- เป็นแม่หม้ายและไม่เคยแต่งงานใหม่

ควรจะให้ && หรือ | | ค่ะ

เนื่องจาก จะเป็น โสดเมื่อเงื่อนไขข้อใดอย่างหนึ่งเป็นจริง,  
เราต้องใช้ **OR** ในการรวมเงื่อนไขเหล่านี้



# Common Error – สับสนระหว่าง && และ | |

พิจารณาสถานะเพื่อจ่ายภาษีอีกตัวอย่างหนึ่ง:

สถานะเราจะเป็นแต่งงานแล้ว ถ้าทุกอย่างต่อไปนี้เป็นจริง:

- คู่ของคุณตายน้อยกว่า 2 ปีและคุณไม่ได้แต่งงานใหม่
- คุณมีลูกที่ไม่ได้อ้างว่าอยู่โดยไม่มีคุณเป็นผู้ปกครอง
- ลูกของคุณอาศัยในบ้านของคุณในปีนั้น
- คุณจ่ายเงินมากกว่า 50 % เพื่อเลี้ยงดูเด็กคนนี้

ใช้ && หรือ | | ดี?

เพราะว่าทุกเงื่อนไขต้องเป็นจริง, เราต้องใช้ && รวมทุกเงื่อนไข.

# Nested Branches – Taxes



**Wait, I *am* still married**

Taxes...

# Short Circuit Evaluation

เมื่อนิ่ง **expression** กลายเป็น **true or false** แล้ว  
เราไม่จำเป็นต้องทำมันแล้ว

**expression && expression && expression && ...**

จะเห็นว่า ถ้าเราเจอ **expression** ใดข้างต้นเป็น **false** เราก็ไม่ต้องทำ **expression** ถัด ๆ  
มาแล้ว เพราะ ถ้ามี **false** อย่างน้อยหนึ่ง ก็ทำให้ได้ค่า **false** แล้ว

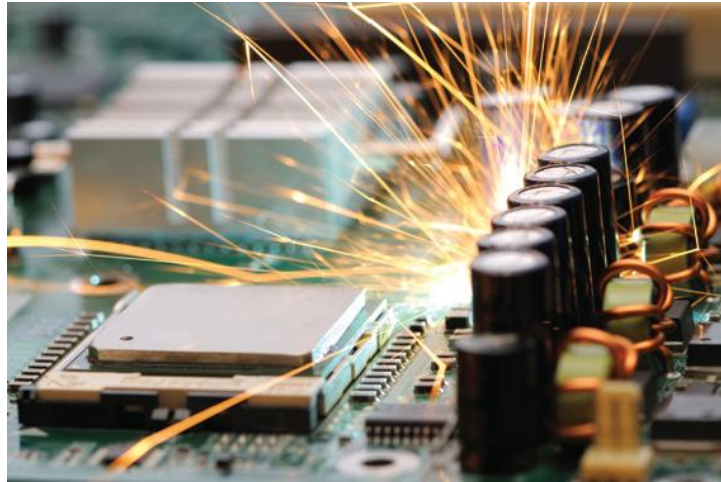
**expression || expression || expression || ...**

ทำนองเดียวกัน ถ้าเราเจอ **expression** ใดข้างต้นเป็น **true** เราก็ไม่ต้องทำ **expression**  
ถัด ๆ มาแล้ว เพราะ ถ้ามี **true** อย่างน้อยหนึ่ง ก็ทำให้ได้ค่า **true** แล้ว

# Short Circuit Evaluation

ภาษา Java หยุดหาค่า **expression** ตัวอื่น ๆ ถ้ามันรู้แล้วว่า จะ **return** ค่าอะไรแน่ ๆ

ความสามารถนี้เรียกว่า การหาผลลัพธ์ทาง **boolean** แบบ *short circuit*



But not the shocking kind.

# DeMorgan's Law

---

Suppose we want to charge a higher shipping rate if we don't ship within the continental United States.

```
shipping_charge = 10.00;  
if (!(country == "USA"  
      && state != "AK"  
      && state != "HI"))  
    shipping_charge = 20.00;
```

This test is a little bit complicated.

DeMorgan's Law to the rescue!

# DeMorgan's Law

---

DeMorgan's Law allows us to rewrite complicated *not/and/or* messes so that they are more clearly read.

```
shipping_charge = 10.00;  
if (country != "USA"  
    || state == "AK"  
    || state == "HI")  
    shipping_charge = 20.00;
```

Ah, much nicer.

But how did they do that?

# DeMorgan's Law

---

DeMorgan's Law:

**$!(A \ \&\& \ B)$  is the same as  $!A \ || \ !B$**

(change the  $\&\&$  to  $||$  and negate all the terms)

**$!(A \ || \ B)$  is the same as  $!A \ \&\& \ !B$**

(change the  $||$  to  $\&\&$  and negate all the terms)

# DeMorgan's Law

So

!(country == "USA" && state != "AK" && state != "HI")

becomes:

!(country == "USA") || !(state != "AK") || !(state != "HI")

and then we make those silly !( ... == ... )'s and !( ... != ... )'s better by making !( == ) be just != and !( != ) be just ==.

country != "USA" || state == "AK" || state == "HI"



# Input Validation with `if` Statements



**You, the programmer, doing Quality Assurance**

***(by hand!)***

# Input Validation with `if` Statements

Let's return to the elevator program and consider input validation.



# Input Validation with `if` Statements

---

- Assume that the elevator panel has buttons labeled 1 through 20 (*but not 13!*).
- The following are illegal inputs:
  - The number 13
  - Zero or a negative number
  - A number larger than 20
  - A value that is not a sequence of digits, such as five
- In each of these cases, we will want to give an error message and exit the program.

# Input Validation with `if` Statements

---

It is simple to guard against an input of 13:

```
if (floor == 13)
{
    System.out.println("Error: " +
        " There is no thirteenth floor.");
    return 1;
}
```

# Input Validation with `if` Statements

---

The statement:

```
return 1;
```

immediately exits the `main` function and therefore terminates the program.

It is a convention to return with the value 0 if the program completes normally, and with a non-zero value when an error is encountered.

# Input Validation with `if` Statements

To ensure that the user doesn't enter a number outside the valid range:

```
if (floor <= 0 || floor > 20)
{
    System.out.println("Error: " +
        " The floor must be between 1 and 20.");
    return 1;
}
```

## Input Validation with `if` Statements

---

Later you will learn more robust ways to deal with bad input, but for now just exiting main with an error report is enough.

Here's the whole program with validity testing:

# Input Validation with `if` Statements – Elevator Program

```
public static void main(String [] args)
{
    int floor;
    Scanner sc_obj = new Scanner(System.in);
    System.out.println("Floor: ");
    floor = sc_obj.nextInt();

    // The following statements check various input errors
    if (floor == 13)
    {
        System.out.println("Error: There is no thirteenth floor.");
        return 1;
    }
    if (floor <= 0 || floor > 20)
    {
        System.out.println("Error: floor must be between 1 and 20.");
        return 1;
    }
}
```



# Input Validation with `if` Statements – Elevator Program

```
// Now we know that the input is valid
int actual_floor;
if (floor > 13)
{
    actual_floor = floor - 1;
}
else
{
    actual_floor = floor;
}

System.out.println(Elevator will travel to the actual floor " +
                    actual_floor);

return 0;
}
```

# Chapter Summary

---

## **Use the `if` statement to implement a decision.**

- The `if` statement allows a program to carry out different actions depending on the nature of the data to be processed.

## **Implement comparisons of numbers and objects.**

- Relational operators (`<` `<=` `>` `>=` `==` `!=`) are used to compare numbers and strings.
- Lexicographic order is used to compare strings.

## **Implement complex decisions that require multiple `if` statements.**

- Multiple alternatives are required for decisions that have more than two cases.
- When using multiple `if` statements, pay attention to the order of the conditions.

# Chapter Summary

---

## **Implement decisions whose branches require further decisions.**

- When a decision statement is contained inside the branch of another decision statement, the statements are *nested*.
- Nested decisions are required for problems that have two levels of decision making.

## **Draw flowcharts for visualizing the control flow of a program.**

- Flow charts are made up of elements for tasks, input/ outputs, and decisions.
- Each branch of a decision can contain tasks and further decisions.
- Never point an arrow inside another branch.

## Chapter Summary

---

### **Design test cases for your programs.**

- Each branch of your program should be tested.
- It is a good idea to design test cases before implementing a program.

### **Use the `bool` data type to store and combine conditions that can be `true` or `false`.**

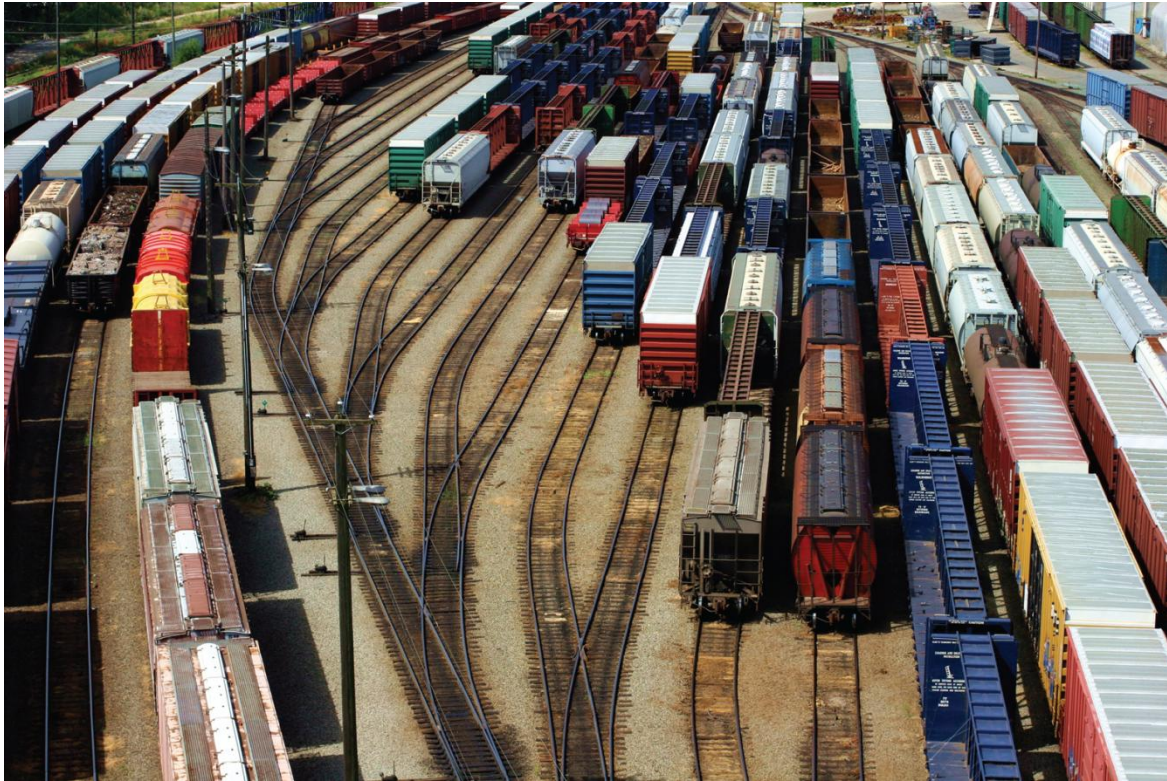
- The `bool` type `bool` has two values, `false` and `true`.
- Java has two Boolean operators that combine conditions: `&&` (*and*) and `||` (*or*).
- To invert a condition, use the `!` (*not*) operator.
- The `&&` and `||` operators use *short-circuit evaluation*: As soon as the truth value is determined, no further conditions are evaluated.
- De Morgan's law tells you how to negate `&&` and `||` conditions.

## Chapter Summary

---

**Apply `if` statements to detect whether user input is valid.**

- When reading a value, check that it is within the required range.
- Use the `fail` function to test whether the input stream has failed.



## End Chapter Three