Advanced Computer Programming

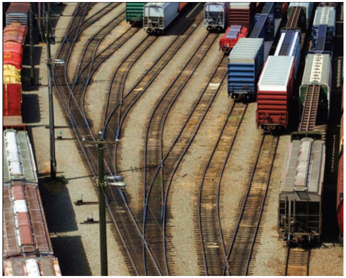
[Lecture 03]

Saeed Reza Kheradpisheh

kheradpisheh@ut.ac.ir

Department of Computer Science
Shahid Beheshti University
Spring 1397-98

DECISIONS



One of the essential features of computer programs is their ability to **make decisions**. Like a train that changes tracks depending on how the switches are set, a program can take different actions depending on inputs and other circumstances.

The `if` Statement

Usage

The **`if` statement** allows a program to carry out different actions depending on the nature of the data to be processed.
It is used to implement a decision.

The `if` Statement

Usage

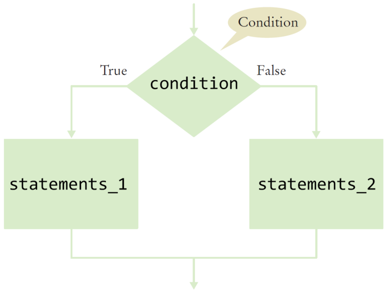
The **`if` statement** allows a program to carry out different actions depending on the nature of the data to be processed.
It is used to implement a decision.

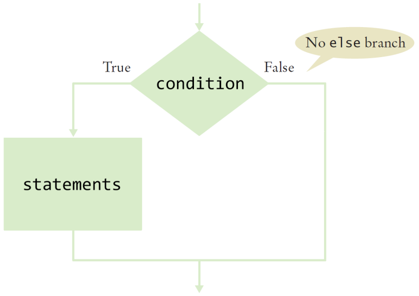
Syntax

```
if (condition)
{
    //body
    statements
}
```

```
if (condition)
{
    //body
    statements_1
}
else
{
    //body
    statements_2
}
```

The `if` Statement





The `if` Statement: Example

This elevator panel "skips" the thirteenth floor. The floor is not actually missing—the computer that controls the elevator adjusts the floor numbers above 13.

The `if` Statement: Example

This elevator panel "skips" the thirteenth floor. The floor is not actually missing—the computer that controls the elevator adjusts the floor numbers above 13.

```
int floor = in.nextInt();
int actualFloor = 0;
if (floor > 13)
{
    actualFloor = floor - 1;
}
else
{
    actualFloor = floor;
}
```

The `if` Statement: Example

This elevator panel "skips" the thirteenth floor. The floor is not actually missing—the computer that controls the elevator adjusts the floor numbers above 13.

```
int floor = in.nextInt();
int actualFloor = floor;
if (floor > 13)
{
    actualFloor--;
}
```

There are Two Types of People

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

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

Always Use Braces

- Braces can be omitted when there is only a single statement in the body of the `if` statement

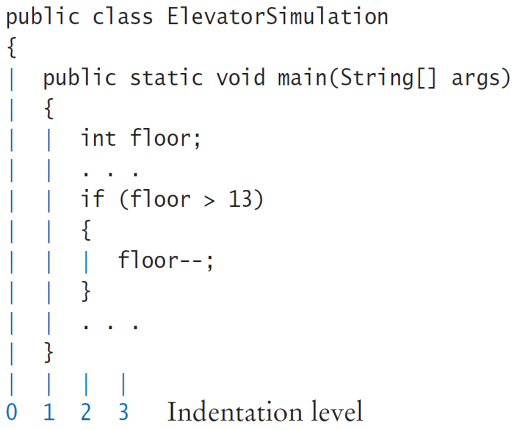
```
if (floor > 13)
    floor--;
```

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

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

Tabs

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



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

Use the ’**Tab**’ key on your keyboard to make indentations.

Avoid Duplication in Branches

Look to see whether you duplicate code in each branch. If so, move it out of the if statement.

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

Avoid Duplication in Branches

Look to see whether you duplicate code in each branch. If so, move it out of the if statement.

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

Relational Operators

Usage

In Java, you use a **relational operator** to compare two values.

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

Relational Operators

Usage

In Java, you use a **relational operator** to compare two values.

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

Double check the equality operator!

Relational Operators: Examples

Expression	Value	Comment
`3 <= 4`	`true`	3 is less than 4; <= tests for "less than or equal".
🚫 `3 =< 4`	**Error**	The "less than or equal" operator is <=, not =<. The "less than" symbol comes first.
`3 > 4`	`false`	> is the opposite of <=.
`4 < 4`	`false`	The left-hand side must be strictly smaller than the right-hand side.
`4 <= 4`	`true`	Both sides are equal; <= tests for "less than or equal".
`3 == 5 - 2`	`true`	== tests for equality.
`3 != 5 - 1`	`true`	!= tests for inequality. It is true that 3 is not 5 – 1.
🚫 `3 = 6 / 2`	**Error**	Use == to test for equality.
`1.0 / 3.0 == 0.333333333`	`false`	Although the values are very close to one another, they are not exactly equal. See Common Error 3.2 on page 91.
🚫 `"10" > 5`	**Error**	You cannot compare a string to a number.

Think Twice, Code Once

Problem

Consider you have a watermelon of weight w and you want to divide it into two parts, each weights an even number.
Write a program that determines whether the division is possible or not, for a given w.

Comparing Strings

- **Equality**: by use of the `equal` method

```
String str = "Tomato";
if (str.equals("Tom"))
{
    System.out.println("Hi");
}
if (str.substring(0, 3).equals("Tom"))
{
    System.out.println("Bye");
}
```

Output:

Comparing Strings

- **Equality**: by use of the `equal` method

```
String str = "Tomato";
if (str.equals("Tom"))
{
    System.out.println("Hi");
}
if (str.substring(0, 3).equals("Tom"))
{
    System.out.println("Bye");
}
```

Output: `Bye`

Comparing Strings

- **Ordering**: by use of the `compareTo` method.
  This ordering is very similar to the way in which words are **sorted in a dictionary**.

  Assume that we want to compare values of two string variables `string1` and `string2`, if
  - `string1.compareTo(string2)` **<** **0**, then
    then the string `string1` comes **before** the string `string2` in the dictionary.
  - `string1.compareTo(string2)` **>** **0**, then
    then the string `string1` comes **after** the string `string2` in the dictionary.
  - `string1.compareTo(string2)` **==** **0**, then
    `string1` and `string2` are **equal**.

Multiple Alternatives

In many situations, there are more than two cases for a decision.

Multiple Alternatives

In many situations, there are more than two cases for a decision.

For example, consider a program that displays the effect of an earthquake, as measured by the Richter scale

Table 3 Richter Scale

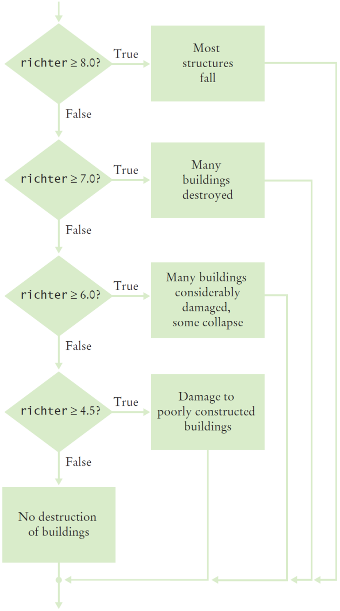
Value	Effect
8	Most structures fall
7	Many buildings destroyed
6	Many buildings considerably damaged, some collapse
4.5	Damage to poorly constructed buildings

Multiple Alternatives

Make use of `if-else if` structure:

```
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



Nested Branches

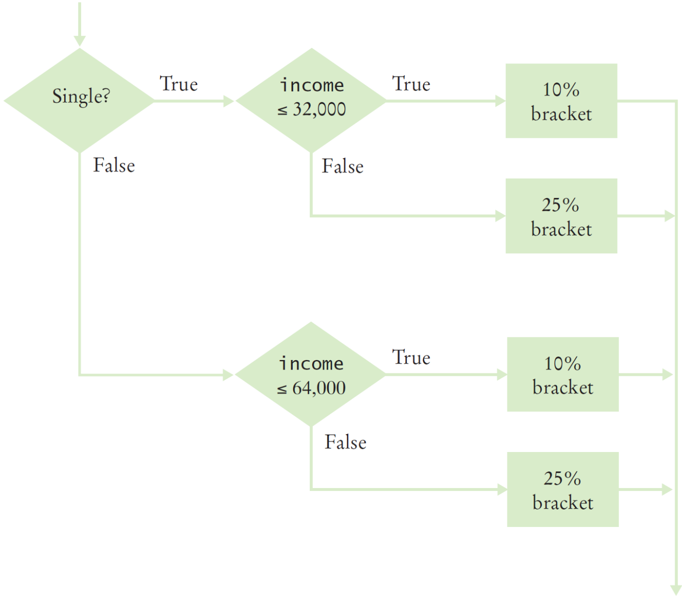
Definition

When a decision statement is contained inside the branch of another decision statement, the statements are **nested**.

Example: In the United States, different tax rates are used depending on the taxpayers marital status.

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

Nested Branches



Nested Branches

```
if (maritalStatus.equals("s"))
{
   if (income <= RATE1_SINGLE_LIMIT)
   {
      tax1 = RATE1 * income;
   }
   else
   {
      tax1 = RATE1 * RATE1_SINGLE_LIMIT;
      tax2 = RATE2 * (income - RATE1_SINGLE_LIMIT);
   }
}
else
{
   if (income <= RATE1_MARRIED_LIMIT)
   {
      tax1 = RATE1 * income;
   }
   else
   {
      tax1 = RATE1 * RATE1_MARRIED_LIMIT;
      tax2 = RATE2 * (income - RATE1_MARRIED_LIMIT);
   }
}
```

The Dangling `else` Problem

Definition

The ambiguous else is called a **dangling `else`**.

You can avoid this pitfall if you always use braces.

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

The Dangling `else` Problem

Definition

The ambiguous else is called a **dangling `else`**.

You can avoid this pitfall if you always use braces.

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

The compiler ignores all indentation and matches the else with the preceding `if`.

The Dangling `else` Problem

Definition

The ambiguous else is called a **dangling `else`**.

You can avoid this pitfall if you always use braces.

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

Who is the First?

Problem

Write a program that takes information of two students including their last name and GPA. The program should then print the name of the student with higher GPA. In case of equality, the program should print the name with lower rank in alphabetic order.

Boolean Variables

Usage

To store a condition that can be true or false, you use a **Boolean variable**.

- In Java, the `boolean` data type has exactly two values, denoted `false` and `true`.
- You can use `boolean` variables later in your program to make a decision.

```
boolean failed = true;
if (failed)
{// Only executed if failed has been set to true
     .   .   .
}
```

Boolean Operators

Definition and Usage

When you make complex decisions, you often need to combine Boolean values. An operator that combines Boolean conditions is called a **Boolean operator**.

Boolean Operators

Definition and Usage

When you make complex decisions, you often need to combine Boolean values. An operator that combines Boolean conditions is called a **Boolean operator**.

- **`&&`** (called and) operator: yields `true` only when both conditions are true.

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

Boolean Operators

Definition and Usage

When you make complex decisions, you often need to combine Boolean values. An operator that combines Boolean conditions is called a **Boolean operator**.

- **`&&`** (called and) operator: yields `true` only when both conditions are true.

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

- **`||`** (called or) operator: yields the result `true` if at least one of the conditions is true.

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

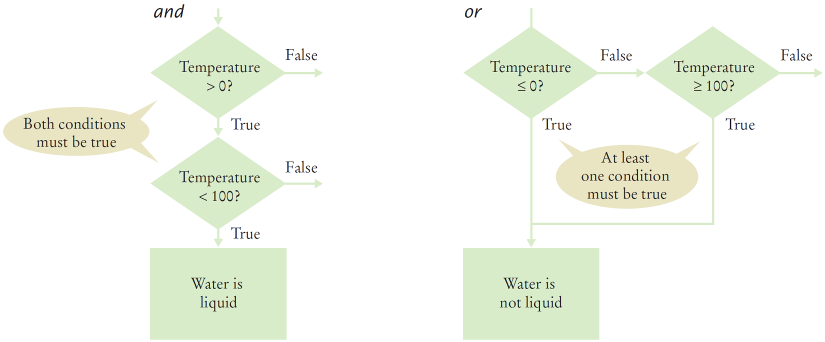
Boolean Operators

Definition and Usage

When you make complex decisions, you often need to combine Boolean values. An operator that combines Boolean conditions is called a **Boolean operator**.

- **&&** (called and) operator: yields `true` only when both conditions are true.
```
if (temp > 0 && temp < 100) {
System.out.println("Liquid"); }
```
- **||** (called or) operator: yields the result `true` if at least one of the conditions is true.
```
if (temp <= 0 || temp >= 100) {
System.out.println("Not liquid"); }
```
- **!** (called not) operator: takes a single condition and evaluates to `true` if that condition is `false`.
```
if (!frozen) { System.out.println("Not frozen"); }
```

Boolean Operators



Boolean Operators

Expression	Value	Comment
`0 < 200 && 200 < 100`	`false`	Only the first condition is true.
`0 < 200 || 200 < 100`	`true`	The first condition is true.
`0 < 200 || 100 < 200`	`true`	The `||` is not a test for “either-or”. If both conditions are true, the result is true.
`0 < x && x < 100 || x == -1`	`(0 < x && x < 100) || x == -1`	The `&&` operator has a higher precedence than the `||` operator (see Appendix B).
🚫 `0 < x < 100`	**Error**	**Error:** This expression does not test whether `x` is between `0` and `100`. The expression `0 < x` is a Boolean value. You cannot compare a Boolean value with the integer `100`.
🚫 `x && y > 0`	**Error**	**Error:** This expression does not test whether `x` and `y` are positive. The left-hand side of `&&` is an integer, `x`, and the right-hand side, `y > 0`, is a Boolean value. You cannot use `&&` with an integer argument.
`!(0 < 200)`	`false`	`0 < 200` is `true`, therefore its negation is `false`.
`frozen == true`	`frozen`	There is no need to compare a Boolean variable with `true`.
`frozen == false`	`!frozen`	It is clearer to use `!` than to compare with `false`.

Boolean Algebra

Problem

Write a program that evaluates boolean *and*, *or*, and *xor*.
The input is a 3-character string in one of the following forms:

- "`PaQ`", for `P` *and* `Q`,
- "`PoQ`", for `P` *or* `Q`,
- "`PxQ`", for `P` *xor* `Q`,

where `P` and `Q` can be either '`0`' or '`1`'.

Boolean Algebra

Problem

Write a program that evaluates boolean *and*, *or*, and *xor*.
The input is a 3-character string in one of the following forms:

- ”`PaQ`”, for `P` *and* `Q`,
- ”`PoQ`”, for `P` *or* `Q`,
- ”`PxQ`”, for `P` *xor* `Q`,

where `P` and `Q` can be either ’`0`’ or ’`1`’.
Add ”`PtQ`”, for *if* `P` *then* `Q`.

Some Handy Tools

Conditional Operator

Usage

Sometimes you just need to switch between two values according to a condition. In this case, the conditional operator facilitates your job.

```
condition ?  value1 :  value2;
```

The value of that expression is either `value1` if the test passes or `value2` if it fails.

Some Handy Tools

Conditional Operator

Example

```
actualFloor = floor > 13 ?  floor - 1 :  floor;
```

equals to

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

Some Handy Tools

The `switch` Statement

Usage

An `if/else if/else` sequence that compares a value against several alternatives can be implemented as a `switch` statement.

```
switch (variableName)
{
        case value_1:
             statements_1
             break;
        case value_2:
             statements_2
             break;
        ...
        case value_n:
             statements_n
             break;
        default:
             statements
             break;
}
```

Some Handy Tools

The `switch` Statement

Example

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

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

Enumeration Types

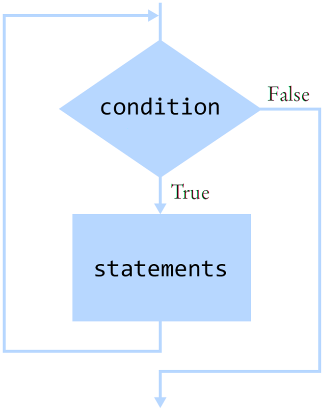
An enumeration type is a type that has a finite set of named values.

```
public enum TypeName {NAME_1, NAME_2, ..., NAME_n}
```

Example

```
public class TaxReturn
{
    public enum FilingStatus {SINGLE, MARRIED}
    public static void main(String[] args)
    {
        FilingStatus status = FilingStatus.SINGLE;
        ...
    }
}
```

The `while` Loop



```
while (condition)
{
      statements
}
```