



D06

PROGRAMMING with JAVA

Ch6 – Decisions

PowerPoint presentation, created by Angel A. Juan - ajuanp@gmail.com,
for accompanying the book "Big Java", by Cay S. Horstmann (2006)

Chapter Goals

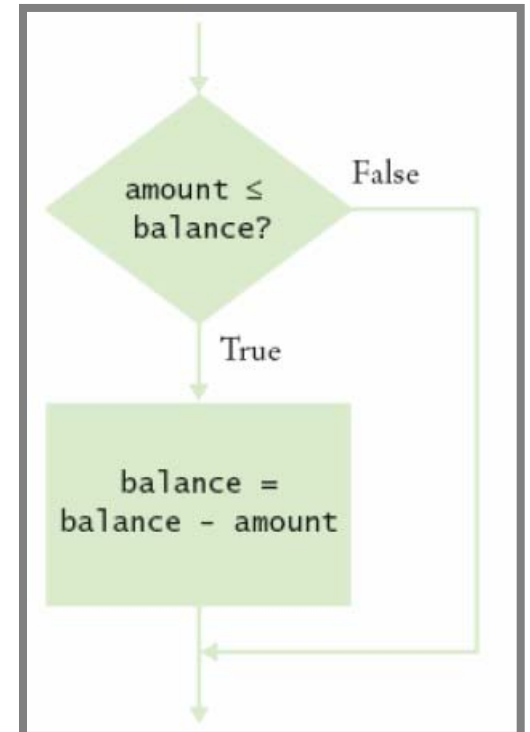
- To be able to implement decisions using **if** statements
- To understand how to group statements into **blocks**
- To learn how to compare integers, floating-point numbers, strings, and objects
- To recognize the correct ordering of decisions in multiple branches
- To program conditions using **Boolean operators** and variables

The `if` Statement

- Computer programs often need to make decisions, taking different actions depending on a condition
- The `if` statement is used to implement a decision. The `if` statement has two parts: a condition and a body. If the condition is true, the body of the statement is executed:

```
if (amount <= balance)
    balance = balance - amount;
```

Figure 1:
Flowchart for an `if` statement

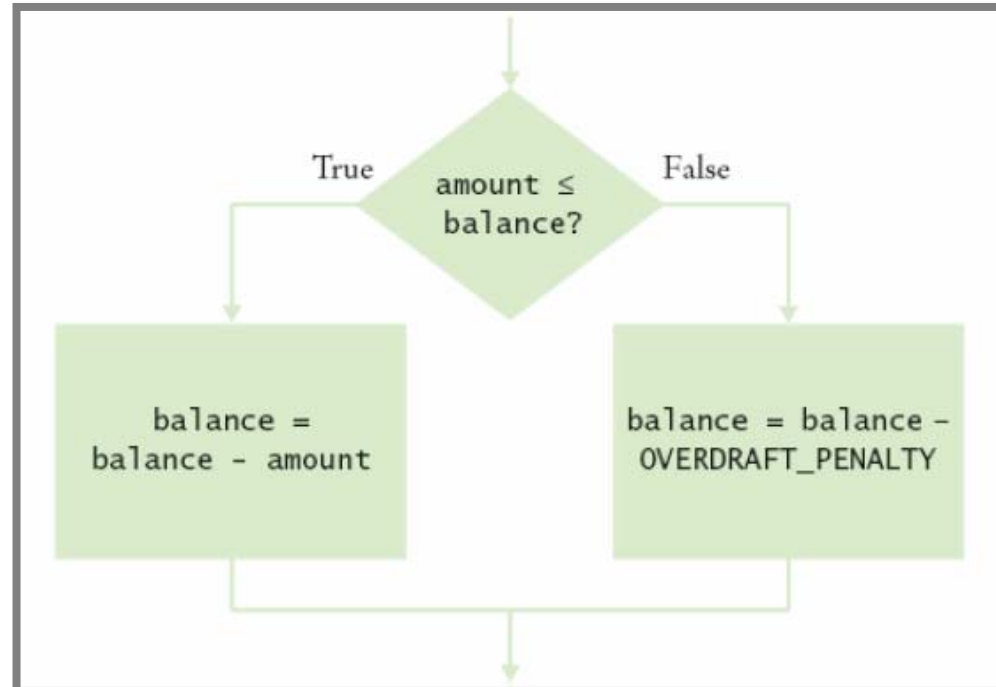


The `if/else` Statement

- To implement a choice between alternatives, use the `if/else` statement:

```
if (amount <= balance)
    balance = balance - amount;
else
    balance = balance - OVERDRAFT_PENALTY;
```

Figure 2:
Flowchart for an
`if/else`
statement



Block statements

- Quite often, the body of the `if` statement consist of multiple statements that must be executed in sequence whenever the condition is true. These statements must be grouped together to form a **block statement** by enclosing them in braces `{}`:

```
{  
    double newBalance = balance - amount;  
    balance = newBalance;  
}
```

Statement Types

- A simple statement:

```
balance = balance - amount;
```

- A compound statement:

```
if (balance >= amount) balance = balance - amount;
```

- Loop statements such as **while**, **for**, etc. (see Chapter 7) are also compound statements

Syntax 6.1: The `if` Statement

```
if(condition)  
    statement
```

```
if (condition)  
    statement1  
else  
    statement2
```

Example:

```
if (amount <= balance)  
    balance = balance - amount;
```

```
if (amount <= balance)  
    balance = balance - amount;  
else  
    balance = balance - OVERDRAFT_PENALTY;
```

Purpose:

To execute a statement when a condition is true or false

Syntax 6.2: Block Statement

```
{  
    statement1  
    statement2  
    . . .  
}
```

Example:

```
{  
    double newBalance = balance - amount;  
    balance = newBalance;  
}
```

Purpose:

To group several statements together to form a single statement

Self-Check

1. Why did we use the condition `amount <= balance` and not `amount < balance` in the example for the `if/else` statement?
2. What is logically wrong with the statement

```
if (amount <= balance)
    newBalance = balance - amount; balance = newBalance;
```

and how do you fix it?

Answers

1. If the withdrawal amount equals the balance, the result should be a zero balance and no penalty
2. Only the first assignment statement is part of the `if` statement. Use braces to group both assignment statements into a block statement

Comparing Values: Relational Operators

- Relational operators compare values:

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



The `==` denotes equality testing while the `=` denotes assignment:

```
a = 5; // Assign 5 to a
if (a == 5) . . . // Test whether a equals 5
```

Comparing Floating-Point Numbers

- Be careful when comparing floating-point numbers in order to cope with **roundoff errors**. Consider this code:

```
double r = Math.sqrt(2);
double d = r * r - 2;
if (d == 0)
    System.out.println("sqrt(2)squared minus 2 is 0");
else
    System.out.println("sqrt(2)squared minus 2 is not 0 but " + d);
```

It prints:

```
sqrt(2)squared minus 2 is not 0 but 4.440892098500626E-16
```



To avoid roundoff errors, **don't use == to compare floating-point numbers**; instead, test whether they are close enough: $|x - y| \leq \epsilon$

```
final double EPSILON = 1E-14;
if (Math.abs(x - y) <= EPSILON)
    // x is approximately equal to y
```

ϵ is a small number such as 10^{-14}

Comparing Strings



Don't use `==` for strings!

```
if (input == "Y") // WRONG!!!
```

To test whether two strings are equal to each other, you must use the method called `equals()`:

```
if (input.equals("Y"))
```

`==` tests **identity**, i.e.: whether the two string variables refer to the same string object (you can have strings with identical contents stored in different objects); on the other hand, `equals()` tests **equal contents**

- In Java, letter case matters. To ignore the letter case use the `equalsIgnoreCase()`:

```
if (input.equalsIgnoreCase("Y"))
```

Continued...

Comparing Strings

- If two strings are not identical to each other, you still may want to know the relationship between them. The `compareTo()` method compares strings in dictionary order: if `s.compareTo(t) < 0` means `s` comes before `t`
- Java is **case sensitive** and sorts characters by putting numbers first, then uppercase characters, then lowercase characters. The space character comes before all other characters:
 - "car" comes before "cargo"
 - "Hello" comes before "car"

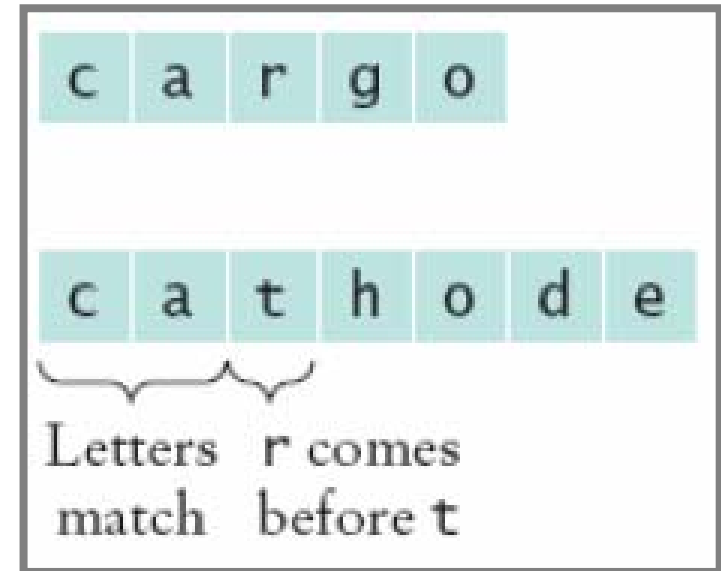


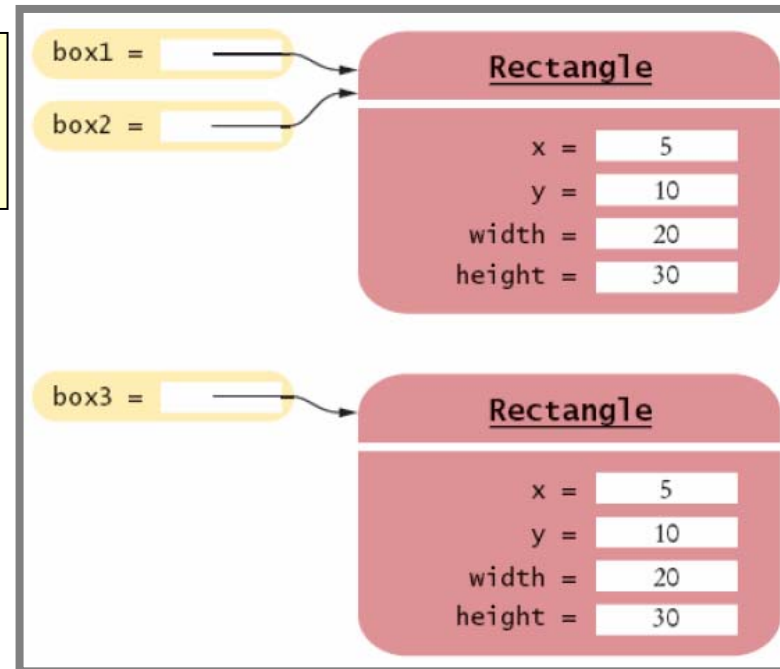
Figure 3:
Lexicographic Comparison

Comparing Objects

⚠ Use the `==` operator to test whether two different references refer to the same object; use `equals()` to test if two different objects have identical contents

```
Rectangle box1 = new Rectangle(5, 10, 20, 30);  
Rectangle box2 = box1;  
Rectangle box3 = new Rectangle(5, 10, 20, 30);
```

- `box1 != box3`,
but `box1.equals(box3)`
- `box1 == box2`



⚠ Caution!: `equals()` works correctly only if it has been implemented in the class

Testing for `null`

- An object reference can have the special value `null` if it refers to no object at all. It is common to use the `null` value to indicate that a value has never been set:

```
String middleInitial = null; // Not set
if ( . . . )
    middleInitial = middleName.substring(0, 1);
```



Use the `==` operator (and not `equals()`) to test whether an object reference is a null reference:

```
if (middleInitial == null)
    System.out.println(firstName + " " + lastName);
else
    System.out.println(firstName + " " + middleInitial + ". "
        + lastName);
```



The `null` reference is not the same as the empty string `""`. The empty string is a valid string of length 0, whereas a `null` indicates that a string variable refers to no string at all

Self Check

3. What is the value of `s.length()` if `s` is:

1. the empty string ""?
2. the string " " containing a space?
3. `null`?

4. Which of the following comparisons are syntactically incorrect? Which of them are syntactically correct, but logically questionable?

```
String a = "1";  
String b = "one";  
double x = 1;  
double y = 3 * (1.0 / 3);
```

1. `a == "1"`
2. `a == null`
3. `a.equals("")`
4. `a == b`
5. `a == x`
6. `x == y`
7. `x - y == null`
8. `x.equals(y)`

Answers

3. (a) 0; (b) 1; (c) an exception is thrown
4. Syntactically incorrect: e, g, h. Logically questionable:
a, d, f

Multiple Alternatives: Sequences of Comparisons

- Many computations require more than a single **if/else** decision. Sometimes, you need to make a series of related comparisons:

```
if (condition1)
    statement1;
else if (condition2)
    statement2;
. . .
else
    statement4;
```

- The first matching condition is executed (order matters)

```
if (richter >= 0) // always passes
    r = "Generally not felt by people";
else if (richter >= 3.5) // not tested
    r = "Felt by many people, no destruction
. . .
```

```
if (richter >= 8.0)
    r = "Most structures fall";
if (richter >= 7.0) // omitted else--ERROR
    r = "Many buildings destroyed
```

- Don't omit **else**:

File Earthquake.java

```
01: /**
02:     A class that describes the effects of an earthquake.
03: */
04: public class Earthquake
05: {
06:     /**
07:         Constructs an Earthquake object.
08:         @param magnitude the magnitude on the Richter scale
09:     */
10:     public Earthquake(double magnitude)
11:     {
12:         richter = magnitude;
13:     }
14:
15:     /**
16:         Gets a description of the effect of the earthquake.
17:         @return the description of the effect
18:     */
```

Continued...

File Earthquake.java

```
19: public String getDescription()  
20: {  
21:     String r;  
22:     if (richter >= 8.0)  
23:         r = "Most structures fall";  
24:     else if (richter >= 7.0)  
25:         r = "Many buildings destroyed";  
26:     else if (richter >= 6.0)  
27:         r = "Many buildings considerably damaged, some  
28:             collapse";  
29:     else if (richter >= 4.5)  
30:         r = "Damage to poorly constructed buildings";  
31:     else if (richter >= 3.5)  
32:         r = "Felt by many people, no destruction";  
33:     else if (richter >= 0)  
34:         r = "Generally not felt by people";  
35:     else  
36:         r = "Negative numbers are not valid";  
37:     return r;  
}
```

Continued...

File Earthquake.java

```
38:  
39:     private double richter;  
40: }
```

File EarthquakeTester.java

```
01: import java.util.Scanner;
02:
03: /**
04:     A class to test the Earthquake class.
05: */
06: public class EarthquakeTester
07: {
08:     public static void main(String[] args)
09:     {
10:         Scanner in = new Scanner(System.in);
11:
12:         System.out.print("Enter a magnitude on the Richter
13:             scale: ");
14:         double magnitude = in.nextDouble();
15:         Earthquake quake = new Earthquake(magnitude);
16:         System.out.println(quake.getDescription());
17:     }
18: }
```

Multiple Alternatives: The `switch` Statement

- A sequence of `if/else/else` that compares a single integer value against several constant alternatives can be implemented as a `switch` statement:

```
int digit;
...
switch (digit)
{
    case 1: System.out.print("one"); break;
    case 2: System.out.print("two"); break;
    case 3: System.out.print("three"); break;
    default: System.out.print("error"); break;
}
```

- The `switch` statement can be applied only in narrow circumstances. The test cases must be constants (integers, characters, or enumerated constants)
- Every branch of the `switch` must be terminated by a `break` instruction. If the `break` is missing, execution falls to the next branch, and so on

Nested Branches

- Some computations have multiple levels of decision making. You first make one decision, and each of the outcomes leads to another decisions

```
if (condition1)
{
    if (condition1a)
        statement1a;
    else
        statement1b;
}
else
    statement2;
```

- Tax Schedule example:

If your filing status is single		If your filing status is married	
Tax Bracket	Percentage	Tax Bracket	Percentage
\$0 ... \$21,450	15%	\$0 ... \$35,800	15%
Amount over \$21,451, up to \$51,900	28%	Amount over \$35,800, up to \$86,500	28%
Amount over \$51,900	31%	Amount over \$86,500	31%

Continued...

Nested Branches

- Compute taxes due, given filing status and income figure: (1) branch on the filing status, (2) for each filing status, **branch** on income level
- The two-level decision process is reflected in two levels of `if` statements. We say that the income test is **nested** inside the test for filing status:

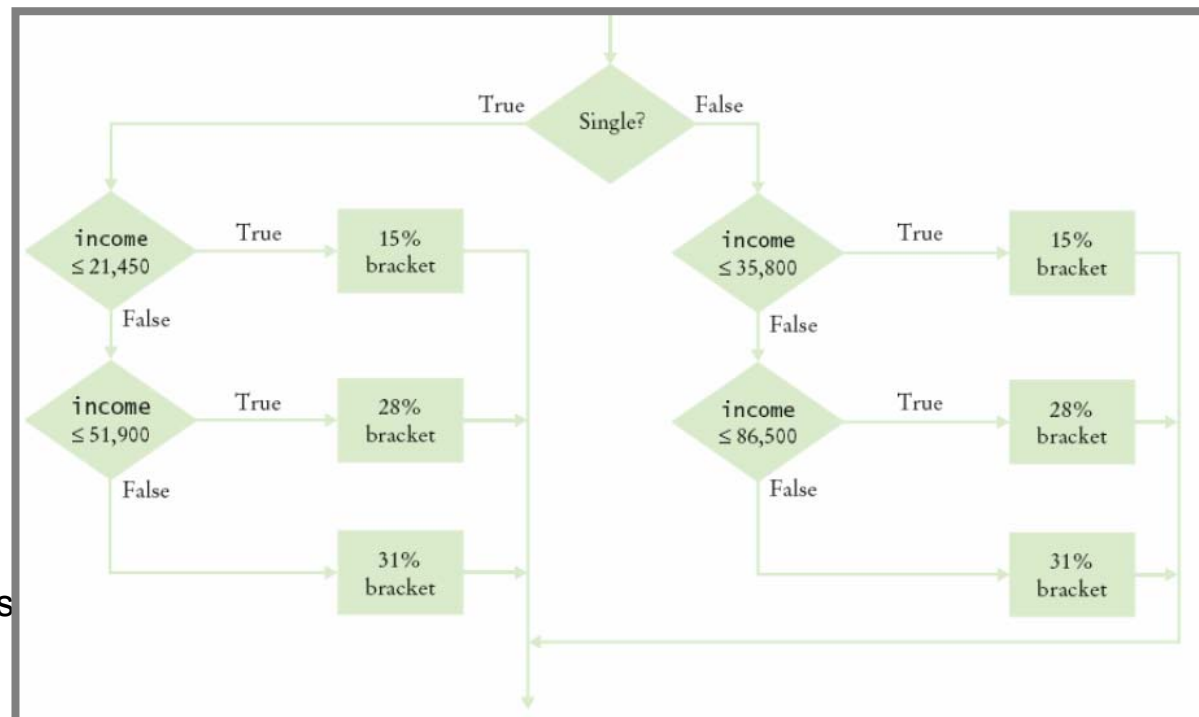


Figure 5:
Income Tax Computation Using
1992 Schedule

File TaxReturn.java

```
01: /**
02:     A tax return of a taxpayer in 1992.
03: */
04: public class TaxReturn
05: {
06:     /**
07:         Constructs a TaxReturn object for a given income and
08:         marital status.
09:         @param anIncome the taxpayer income
10:         @param aStatus either SINGLE or MARRIED
11:     */
12:     public TaxReturn(double anIncome, int aStatus)
13:     {
14:         income = anIncome;
15:         status = aStatus;
16:     }
17:
```

Continued...

File TaxReturn.java

```
18:     public double getTax()
19:     {
20:         double tax = 0;
21:
22:         if (status == SINGLE)
23:         {
24:             if (income <= SINGLE_BRACKET1)
25:                 tax = RATE1 * income;
26:             else if (income <= SINGLE_BRACKET2)
27:                 tax = RATE1 * SINGLE_BRACKET1
28:                     + RATE2 * (income - SINGLE_BRACKET1);
29:             else
30:                 tax = RATE1 * SINGLE_BRACKET1
31:                     + RATE2 * (SINGLE_BRACKET2 - SINGLE_BRACKET1)
32:                     + RATE3 * (income - SINGLE_BRACKET2);
33:         }
```

Continued...

File TaxReturn.java

```
34:         else
35:         {
36:             if (income <= MARRIED_BRACKET1)
37:                 tax = RATE1 * income;
38:             else if (income <= MARRIED_BRACKET2)
39:                 tax = RATE1 * MARRIED_BRACKET1
40:                     + RATE2 * (income - MARRIED_BRACKET1);
41:             else
42:                 tax = RATE1 * MARRIED_BRACKET1
43:                     + RATE2 * (MARRIED_BRACKET2 - MARRIED_BRACKET1)
44:                     + RATE3 * (income - MARRIED_BRACKET2);
45:         }
46:
47:         return tax;
48:     }
49:
50:     public static final int SINGLE = 1;
51:     public static final int MARRIED = 2;
52:
```

Continued...

File TaxReturn.java

```
53:     private static final double RATE1 = 0.15;
54:     private static final double RATE2 = 0.28;
55:     private static final double RATE3 = 0.31;
56:
57:     private static final double SINGLE_BRACKET1 = 21450;
58:     private static final double SINGLE_BRACKET2 = 51900;
59:
60:     private static final double MARRIED_BRACKET1 = 35800;
61:     private static final double MARRIED_BRACKET2 = 86500;
62:
63:     private double income;
64:     private int status;
65: }
```

File TaxReturnTester.java

```
01: import java.util.Scanner;
02:
03: /**
04:     A class to test the TaxReturn class.
05: */
06: public class TaxReturnTester
07: {
08:     public static void main(String[] args)
09:     {
10:         Scanner in = new Scanner(System.in);
11:
12:         System.out.print("Please enter your income: ");
13:         double income = in.nextDouble();
14:
15:         System.out.print("Please enter S (single) or M
16:             (married): ");
17:         String input = in.next();
18:         int status = 0;
```

Continued...

File TaxReturnTester.java

```
19:         if (input.equalsIgnoreCase("S"))
20:             status = TaxReturn.SINGLE;
21:         else if (input.equalsIgnoreCase("M"))
22:             status = TaxReturn.MARRIED;
23:         else
24:         {
25:             System.out.println("Bad input.");
26:             return;
27:         }
28:
29:         TaxReturn aTaxReturn = new TaxReturn(income, status);
30:
31:         System.out.println("The tax is "
32:             + aTaxReturn.getTax());
33:     }
34: }
```

Output:

```
Please enter your income: 50000
Please enter S (single) or M (married): S
The tax is 11211.5
```

Self Check

5. The `if/else/else` statement for the earthquake strength first tested for higher values, then descended to lower values. Can you reverse that order?
6. Some people object to higher tax rates for higher incomes, claiming that you might end up with less money after taxes when you get a raise for working hard. What is the flaw in this argument?

Answers

5. Yes, if you also reverse the comparisons:

```
if (richter < 3.5)
    r = "Generally not felt by people";
else if (richter < 4.5)
    r = "Felt by many people, no destruction";
else if (richter < 6.0)
    r = "Damage to poorly constructed buildings";
. . .
```

6. The higher tax rate is only applied on the income in the higher bracket. Suppose you are single and make \$51,800. Should you try to get a \$200 raise? Absolutely—you get to keep 72% of the first \$100 and 69% of the next \$100

Enumerated Types

- Java 5.0 introduces the **Enumerated Types (ET)**. An ET has a finite set of values:

```
public enum FilingStatus { SINGLE, MARRIED }
```

- You can have any number of values, but you must include them all in the **enum** declaration. You can declare variables of the enumerated type:

```
FilingStatus status = FilingStatus.SINGLE;
```

- Use the **==** operator to compare enumerated values:

```
if (status == FilingStatus.SINGLE) ...
```

Continued...

Enumerated Types



It is common to nest an **enum** declaration inside a class:

```
public class TaxReturn
{
    public TaxReturn(double anIncome, FilingStatus aStatus) {...}
    ...
    public enum FilingStatus { SINGLE, MARRIED }
    private FilingStatus status;
}
```

To access the enumeration outside the class in which it is defined, use the class name as a prefix:

```
TaxReturn return = new TaxReturn(income, TaxReturn.FilingStatus.SINGLE);
```

- An ET variable always can be **null** (in the example, `status` could have three values: `SINGLE`, `MARRIED` and `null`)

Using Boolean Expressions: The `boolean` Type

- The value of a relational expression is either **true** or **false** (e.g.: the value of expression `amount < 1000` is true or false)
- The values `true` and `false` are not numbers, nor are they objects of a class. They belong to a separate type, called **boolean**



Using Boolean Expressions: Predicate Methods

- A **predicate method** returns a **boolean value**

```
public boolean isOverdrawn()  
{  
    return balance < 0;  
}
```

- Use in conditions

```
if (harrysChecking.isOverdrawn()) . . .
```

- Useful predicate methods in **Character** class:

```
if (Character.isUpperCase(ch)) . . .
```

```
isDigit()  
isLetter()  
isUpperCase()  
isLowerCase()
```



- Useful predicate methods in **Scanner** class:
hasNextInt() and **hasNextDouble()**

```
if (in.hasNextInt()) n = in.nextInt();
```

Using Boolean Expressions: The Boolean Operators

▪ **&&** and

```
if (0 < amount && amount < 1000) . . .
```

▪ **||** or

```
if (input.equals("S") || input.equals("M")) . . .
```

▪ **!** Not

```
if (!input.equals("S")) . . .
```

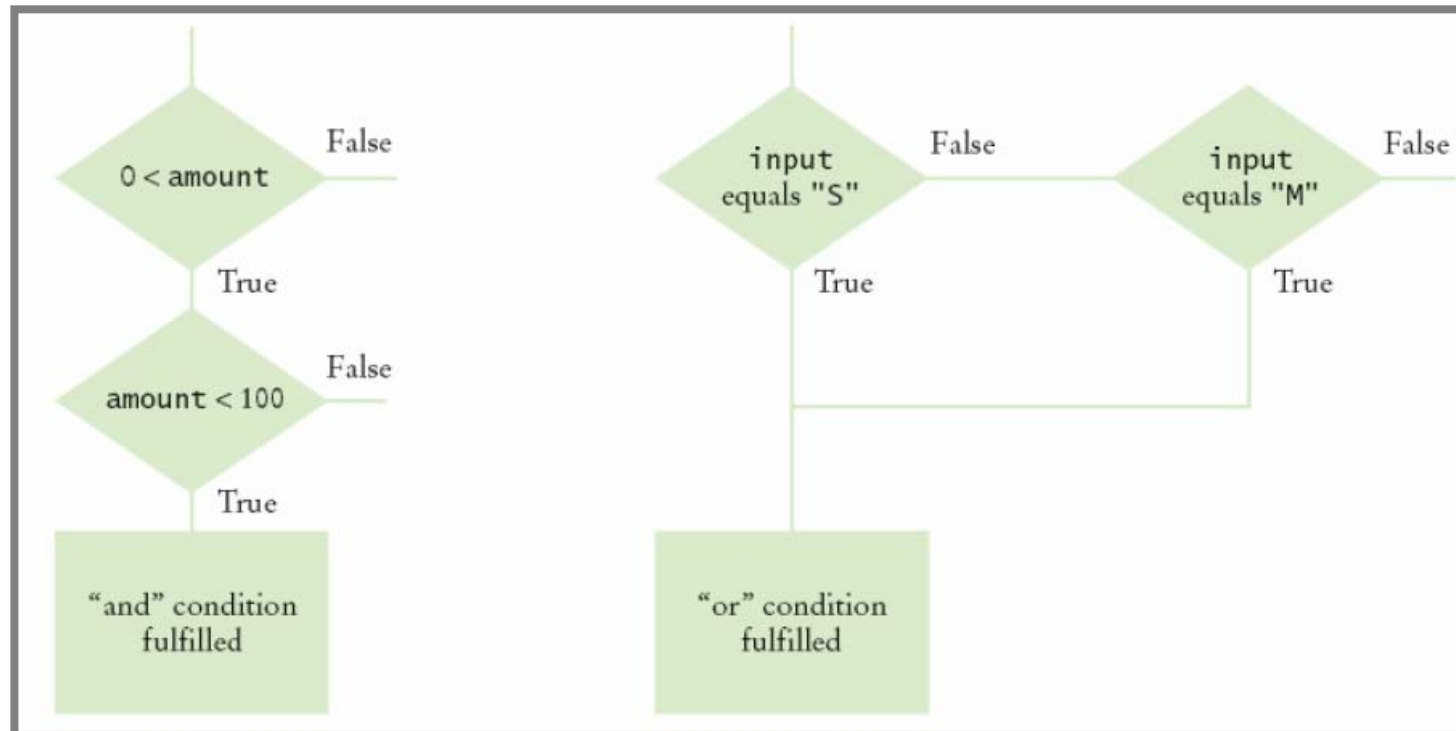


Figure 6:
Flowcharts for && and
|| Combinations

Truth Tables

A	B	A&&B
True	True	True
True	False	False
False	Any	False

A	B	A B
True	<i>Any</i>	True
False	True	True
False	False	False

A	!A
True	False
False	True

Using Boolean Variables

- You can use a **Boolean** variable if you know that there are only two possible values:

```
private boolean married;
```

- Set to truth value:

```
married = input.equals("M");
```

- Use in conditions:

```
if (married) . . . else . . .  
if (!married) . . .
```

- Sometimes Boolean variables are called **flags** because they can have only two states: “up” and “down”

- The second form is preferred (they are equivalent):

```
if (married == true) . . . // Don't
```

```
if (married) . . .
```

Self Check

7. When does the statement

```
system.out.println (x > 0 || x < 0);
```

print false?

8. Rewrite the following expression, avoiding the comparison with false:

```
if (Character.isDigit(ch) == false) . . .
```

Answers

7. When `x` is zero

8. `if (!Character.isDigit(ch)) . . .`

Chapter Summary

- The **if** statement lets a program carry out different actions depending on a condition
- A **block statement** groups several statements together
- **Relational operators** compare values. The **==** operator tests for equality (identity)
- When comparing floating-point numbers, don't test for equality. Instead, check whether they are close enough
- Do not use the **==** operator to compare strings. Use the **equals()** method instead
- The **compareTo()** method compares strings in dictionary order



The **==** operator tests whether two object references are identical. To compare the contents of objects, you need to use the **equals()** method

Continued...

Chapter Summary

- The **null** reference refers to no object
- Multiple conditions can be combined to evaluate complex decisions. The correct arrangement depends on the logic of the problem to be solved
- The **boolean type** has two values: **true** and **false**
- A **predicate method** returns a boolean value
- You can form complex tests with the Boolean operators **&&** (and), **||** (or), and **!** (not)
- You can store the outcome of a condition in a Boolean variable