Decisions

The `if` Statement

- The `if` statement lets a program carry out different actions depending on a condition

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

The `if` Statement

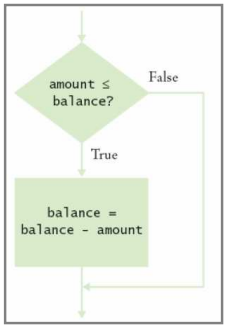

Figure 1:
Flowchart for an `if` statement

The `if/else` Statement

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

The `if/else` Statement

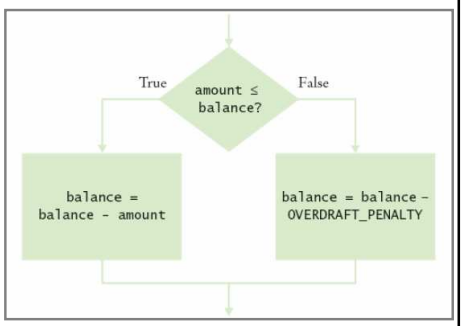

Figure 2:
Flowchart for an `if/else` statement

Statement Types

- **Simple statement**

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

- **Compound statement**

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

Also `while`, `for`, etc. (loop statements–Chapter 7)

Statement Types

- **Block statement**

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

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**

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

Comparing Floating-Point Numbers

- 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
```

Comparing Floating-Point Numbers

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

```
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!!!
```

- Use equals method:

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

`==` tests identity, equals tests equal contents

- Case insensitive test ("Y" or "y")

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

Continued...

Comparing Strings

- s.compareTo(t) < 0 means s comes before t in the dictionary
- "car" comes before "cargo"
- All uppercase letters come before lowercase: "Hello" comes before "car"

Lexicographic Comparison

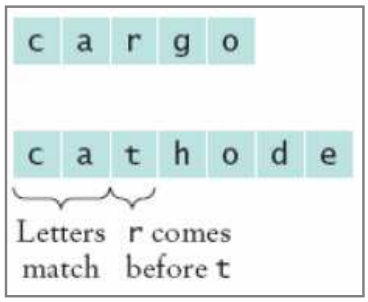

Figure 3:
Lexicographic Comparison

Comparing Objects

- == tests for identity, equals for identical content
-

```
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`
- Caveat: `equals` must be defined for the class

Object Comparison

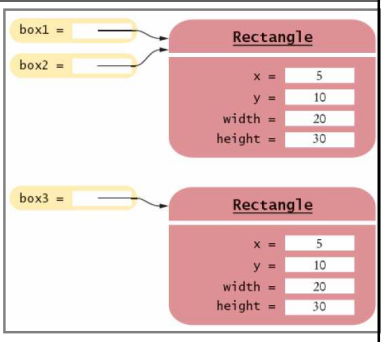

Figure 4:
Comparing Object References

Testing for `null`

- **`null` reference refers to no object**

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

- **Can be used in tests:**

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

Testing for `null`

- **Use ==, not `equals`, to test for `null`**
- **`null` is not the same as the empty string `""`**

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`?

Answers

3. **(a) 0; (b) 1; (c) an exception is thrown**

Self-Check

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: 5, 7, 8. Logically questionable: 1, 4, 6

Multiple Alternatives: Sequences of 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";
. . .
```

Multiple Alternatives: Sequences of Comparisons

- Don't omit `else`

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

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
                 collapse";
28:       else if (richter >= 4.5)
29:          r = "Damage to poorly constructed buildings";
30:       else if (richter >= 3.5)
31:          r = "Felt by many people, no destruction";
32:       else if (richter >= 0)
33:          r = "Generally not felt by people";
34:       else
35:          r = "Negative numbers are not valid";
36:       return r;
37:    }
```

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
             scale: ");
13:       double magnitude = in.nextDouble();
14:       Earthquake quake = new Earthquake(magnitude);
15:       System.out.println(quake.getDescription());
16:    }
17: }
```

Multiple Alternatives: Nested Branches

- **Branch inside another branch**

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

Tax Schedule (Updated for 2006 rates)

If your filing status is single		If your filing status is married	
Tax Bracket	**Percentage**	**Tax Bracket**	**Percentage**
$0 … $7,550	10%	$0 … $15,100	10%
Amount over $7,550, up to $30,650	15%	Amount over $15,100, up to $61,300	15%
Amount over $30,650, up to $74,200	25%	Amount over $61,300, up to $123,700	25%
Amount over $74,200, up to $154,800	28%	Amount over $123,700, up to $188,450	28%
Amount over $154,800, up to $336,550	33%	Amount over $188,450, up to $336,550	33%
Amount over $336,550	35%	Amount over $336,550	35%

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**

Continued…

Nested Branches

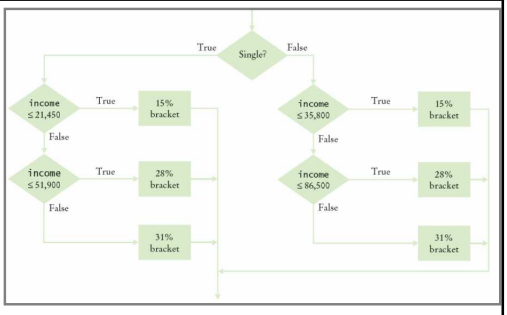

Figure 5:
Income Tax Computation Using 1992 Schedule

File `TaxReturn.java`

Note: This example is not up to date for 2005 rates… old example.

```
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
               (married): ");
16:       String input = in.next();
17:       int status = 0;18:
```

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: }
```

File TaxReturnTester.java

Output

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

Using Boolean Expressions: The `boolean` Type

- George Boole (1815-1864): pioneer in the study of logic
- value of expression `amount < 1000` is `true` or `false`.
- `boolean` type: one of these 2 truth values

Using Boolean Expressions: The `boolean` Type

Using Boolean Expressions: Predicate Method

- A predicate method returns a `boolean` value

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

- Use in conditions

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

Continued...

Using Boolean Expressions: Predicate Method

- Useful predicate methods in `Character` class:

```
isDigit
isLetter
isUpperCase
isLowerCase
```

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

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

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

Using Boolean Expressions: The Boolean Operators

- `&&` and
- `||` or
- `!` Not
-
```
if (0 < amount && amount < 1000) . . .
```
-
```
if (input.equals("M") || input.equals("S")) . . .
```
-
```
if (!input.equals("M")) . . .
```

&& and || Operators

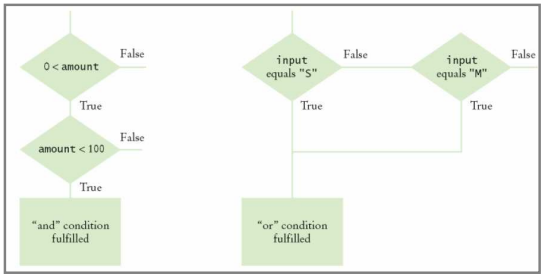

Figure 6:
Flowcharts for && and || Combinations

Truth Tables

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

A	B	A||B
True	*Any*	True
False	True	True
False	False	False

A	!A
True	False
False	True

Using Boolean Variables

- `private boolean married;`
- **Set to truth value:**

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

- **Use in conditions:**

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

Using Boolean Variables

- **Also called *flag***
- **It is considered unpolished to write a test such as**

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

Just use the simpler test

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