Chapter 3

Program Control Flow

Lecture slides for:

Java Actually: A Comprehensive Primer in Programming

Khalid Azim Mughal, Torill Hamre, Rolf W. Rasmussen

Cengage Learning, 2008.

ISBN: 978-1-844480-933-2

`http://www.ii.uib.no/~khalid/jac/`

Permission is hereby granted to use these lecture slides in conjunction with the book.

Modified: 14/11/08

Overview

- Boolean expressions
- Control flow: selection statements
- Control flow: loops
- Assertions

Boolean expressions

- Conditions can be expressed by means of *boolean expressions*.
- A boolean expression has the primitive data type `boolean`, and always evaluates to one of two values: `true` or `false`.
- A boolean expression can be formulated using *relational operators* and / or *logical operators*.

```
// Boolean variables are like other variables
boolean flag, OK;
flag = true;
OK = flag;
```


Relational operators

Operator	**Meaning** (`a` *and* `b` *are arithmetic expressions)*
`a == b`	`a` is equal to `b`?
`a != b`	`a` is not equal to `b`?
`a < b`	`a` is less than `b`?
`a <= b`	`a` is less than or equal to `b`?
`a > b`	`a` is greater than `b`?
`a >= b`	`a` is greater than or equal to `b`?

- All relational operators are *binary.*
- *Precedence*: (high to low)

  1. *Arithmetic operators* 2. *Relational operators* 3. *Assignment operators*

```
int i = 1999;
boolean isEven = i%2 == 0;           // false
float numHours = 56.5;
boolean overtime = numHours > 37.5;  // true
```

Logical operators

Operator	Meaning
`!`	Negation, results in inverting the `truth` value of the operand, i.e. `!true` evaluates to `false`, and `!false` evaluates to `true`.
`&&`	Conditional And, evaluates to `true` if both operands have the value `true`, and `false` otherwise.
`||`	Conditional Or, evaluates to `true` if one or both operands have the value `true`, and `false` otherwise.

- Precedence rules:
  1. Negation
  2. Conditional And
  3. Conditional Or
- The binary operators Conditional Or (`||`) and And (`&&`) associate from *left to right*.
- The unary operator Negation (`!`) associate from *right to left*.

Logical operators

- *Short-Circuit evaluation* of logical operators.
  – Evaluation stops as soon as the truth value of the expression can be determined.
- Example:

```
boolean b1 = (4 == 2) && (1 < 4) // false - parentheses can be left out.
boolean b2 = (!b1) || (2.5 > 8)  // true - parentheses can be left out.
boolean b3 = !(b1 && b2)         // true
boolean b4 = b1 || !b3 && b2     // false
```

Evaluation sequence:

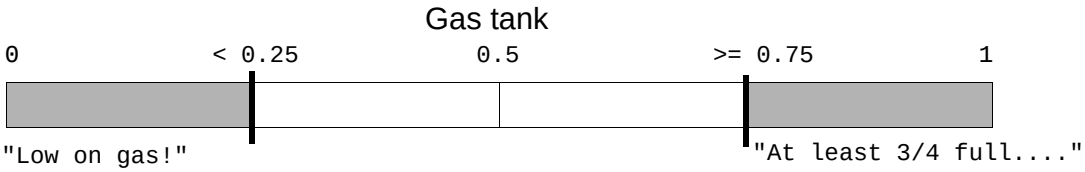
```
(b1 || ((!b3) && b2)) => (false || ((false) && b2))
=> (false || (false)) => (false)

(!b3) && b2) => ((false) && b2) => (false)
                                   (*)
```

Note: Value of the expression must be `false` *since one operand of Conditional And has the value* `false`*. Thus, evaluation stops without considering the value of b2.*

De Morgans laws

b1 *og* b2 *are boolean expressions*
`!(b1 && b2) <=> (!b1 || !b2)`
`!(b1 || b2) <=> (!b1 && !b2)`

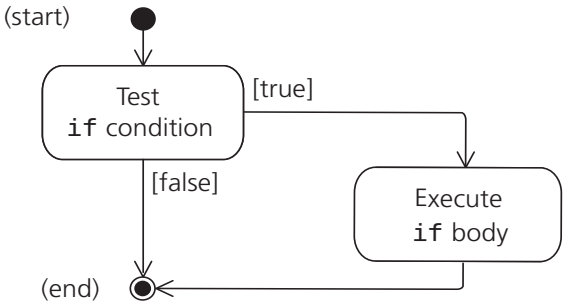


`!(gas_tank < 0.75 && gas_tank < 0.25) <=> !(gas_tank < 0.75) || !(gas_tank < 0.25)`
`!(gas_tank < 0.75) <=> (gas_tank >= 0.75)`
`!(gas_tank < 0.25) <=> (gas_tank >= 0.25)`
`!(gas_tank < 0.75 && gas_tank < 0.25) <=> (gas_tank >= 0.75) || (gas_tank >= 0.25)`

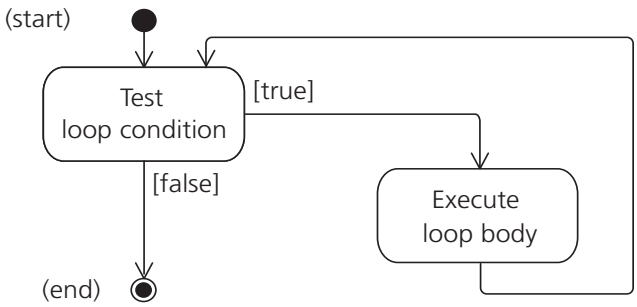
Boolean expression like these can be used to control the program's flow of execution, e.g. to turn on a red lamp if the gas tank level falls below 25%.

Controlling flow of execution

- Two types of statements can be used to control the execution of actions:
  - Selection statement: action is *executed only* if a certain condition is true.
  - Loop statement: action is *repeated* as long as a certain condition is true.



(a) Control flow: `if` statement

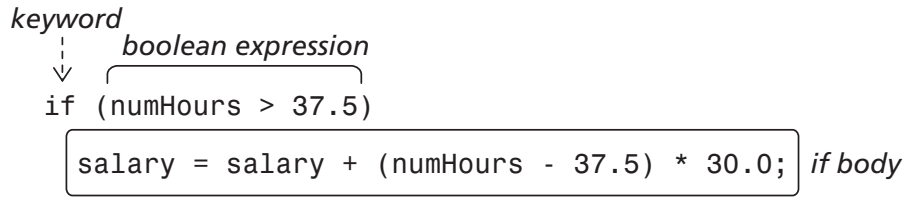


(b) Control flow: loop

  - The condition is specified as a boolean expression.

Control flow: selection statements

- Simple selection statement: `if`
  - Performs an action, the `if` body, if a given condition is true.



Problem: We want to calculate the weekly salary of an employee that has a fixed salary of 750 USD, but receives additional payment if she works overtime. The overtime payment is based on a fixed salary of 30 USD per hour for every hour past the normal work week of 37.5 hours.

- An employee who hasn't worked overtime, has a salary of:
  `750.0`
- An employee who has worked overtime, has a salary of:
  `750.0 + (numHours - 37.5) * 30.0`

Simple selection statement: `if`

- A simple `if`-statement can be used to decide whether an action (comprised of a set of statements) will be executed.

Syntax:

```
if (<boolean expression>) {
    /* statements for if body */
}
```

Semantics:

- Evaluate the boolean expression:
  - If it is `true`, execute the statements in the *if body*, and continue with the rest of the program.
  - If it is `false`, skip the *if body*, and continue with the rest of the program.

Example:

```
salary = 750.0;
if (numHours > 37.5) {
    salary = salary + (numHours - 37.5) * 30.0;
}
```

Simple selection statement: `if`

```
// Calculating weekly salary, version 1.
import java.util.Scanner;
public class Salary1 {
  public static void main(String[] args) {
    final double NORMAL_WORKWEEK = 37.5;

    // Read the number of hours worked this week.
    Scanner keyboard = new Scanner(System.in);
    System.out.print("Enter the number of hours worked [decimal number]: ");
    double numHours = keyboard.nextDouble();

    // Calculate the weekly salary and print it to the terminal window.
    double salary = 750.0;                          // (1) weekly salary
    if (numHours > NORMAL_WORKWEEK)                                    // (2)
      salary = salary + (numHours - NORMAL_WORKWEEK) * 30.0;           // (3)
    System.out.printf("Salary for %.1f hours is %.2f USD%n",
                      numHours, salary);                               // (4)
  }
```



```
}
```

Running the program:

```
Enter the number of hours worked [decimal number]: 46.5
Salary for 46.5 hours is 1020.00 USD
```

Statements and blocks

- A statement in Java can be *simple* (i.e. consist of only one action) or *compound* (i.e. consist of multiple statements grouped in a *block*).

A "local" block

- Block notation: `{`<-- *initiates a block* `. . .` *ends a block* --`>}`
- A block is used to group a *sequence of statements*, and may contain variable declarations.
- A block (of statements) is considered one statement (called a *compound statement*), and can be used wherever a simple statement can be used.
- If a block contains only a single statement, block notation is not needed.

Statement terminator: `;`

- Semicolon (`;`) is used to *terminate* a statement in Java.
- A compound statement does *not* need a semicolon to terminate it.
- A semicolon alone denotes the *empty statement* that does nothing.

More on selection

Problem: We want to calculate weekly salary based on hourly wages and the number of hours worked during the current week.

Assume 50% extra salary is paid for all hours above `37.5` hours a week.

1. For someone working overtime, the salary can be calculated by:
   ```
   hourlyRate * 37.5 + 1.5 * hourlyRate * (numHours - 37.5)
   ```
2. For someone not working overtime, the salary can be calculated by:
   ```
   hourlyRate * numHours
   ```

A program for calculation of weekly salaries has to make a choice between these formulas, depending on the number of hours worked during the current week.

If the condition for overtime is true

```
salary = hourlyRate*37.5 + 1.5*hourlyRate*(numHours-37.5)
```

else (i.e. the condition is false)

```
salary = hourlyRate*numHours
```

Selection statement with two choices: `if-else`

- An `if-else`-statement can be used to select between *two* alternative actions.

Syntax:

```
if (<boolean expression>) {
    /* statements for if body */
}
else {
    /* statements for else body */
}
```

Semantics:

- Evaluate the boolean expression:
  - If it is `true`, execute the statements in the *if body*, and continue with the rest of the program.
  - If it is `false`, execute the statements in the *else body*, and continue with the rest of the program.

Selection statement with two choices: `if-else`

```
// Calculating weekly salary, version 2.
import java.util.Scanner;
public class Salary2 {
  public static void main(String[] args) {
    final double NORMAL_WORKWEEK = 37.5;
    final double FIXED_SALARY = 750.0;

    // Read the number of hours worked this week.
    Scanner keyboard = new Scanner(System.in);
    System.out.print("Enter the number of hours worked [decimal number]: ");
    double numHours = keyboard.nextDouble();

    //  Calculate the weekly salary and print it to the terminal window.
    double salary = 0.0;                                  // weekly salary
    if (numHours <= NORMAL_WORKWEEK) {                    // (1)
      salary = FIXED_SALARY;                              // (2) if body
    } else {                                              // (3)
      salary = FIXED_SALARY +
```

```
                (numHours - NORMAL_WORKWEEK) * 30.0;       // (4) else body
      }
      System.out.printf("Salary for %.1f hours is %.2f USD%n",
                        numHours, salary);
    }
}
```

Running the program 1:

```
Enter the number of hours worked [decimal number]: 39.5
Salary for 39.5 hours is 810.00 USD
```

Running the program 2:

```
Enter the number of hours worked [decimal number]: 42.5
Salary for 42.5 hours is 900.00 USD
```


Nested selection statements

Desired program behaviour:

If *gas tank is less than 3/4 full:*

Check if it is less than 1/4 full and output message "Low on gas!"

else:

Gi melding "At least 3/4 tank. Go on!"

Program code (containing a logical error):

```
if (gas_tank < 0.75)         // 1st if-statement
    if (gas_tank < 0.25)     // 2nd if-statement
        System.out.println("Low on gas!");
else
    System.out.println("At least 3/4 tank. Go on!");
```

Q: Which `if`-statement is the `else`-body associated with?

A: The `else`-body is *always* associated to the *nearest* `if`-statement.

- The program code above contains a *logical error*!

Nested selection statements

Correct program code:

```
if (gas_tank < 0.75) {   // Necessary to use block notation
    if (gas_tank < 0.25) // if-body of 1st if-statement
        System.out.println("Low on gas!");
} else {
    System.out.println("At least 3/4 tank. Go on!");
}
```

Best practice:

- *Enclose if-body and else-body using block notation,* `{}`.

Nested selection statements

Can we find a simpler solution to the problem?

What is wrong with the following code?

```
if (gas_tank < 0.75 && gas_tank < 0.25) {
    System.out.println("Low on gas!");
} else {
    System.out.println("At least 3/4 tank. Go on!");
}
```

- The 2nd message can be printed even if the tank is *not* 3/4 full!
- Be careful when combining nested conditions.

`if (`b_1`)`
`  if (`b_2`)`
`  if (`b_n`)`
`  ...`
`else`
`  ...`
	is not necessarily equal to
	(equal if there is no `else`-body)
		`if (`b_1 `&&` b_2 `...` `&&` b_n`)`
		`  ...`
		`else`
		`  ...`

Nested selection statements

A better solution:

```
if (gas_tank < 0.25) {
    System.out.println("Low on gas!");
} else if (gas_tank >= 0.75) { // if-statement in else-body
    System.out.println("At least 3/4 tank. Go on!");
}
```

Common errors when using `if`-statements

```
if (a = b)    // Syntax error: condition is not a boolean expression!
    System.out.println("Syntax error!");

if (a == b) ; // Empty statement
    System.out.println("a equals b!"); // Logical error: always executed!

if (a == b) ; // No error: Empty statement means "do nothing"
else System.out.println("a is not equal to b!");
```


Chaining `if-else` statements

Desired program behaviour:

Print "Too high!" if guess > answer

Print "Too low!" if guess < answer

Print "Correct answer!" if guess == answer

Program code:

```
if (guess > answer) {
    System.out.println("Too high!");
} else if (guess < answer) {
    System.out.println("Too low!");
} else { // if (guess == answer)
    System.out.println("Correct answer!");
}
```

Control flow: loops

- Loops allow repeated execution of a part of the program.
  - The boolean expression (`true` or `false`) that controls loop termination, is called the *loop condition*.
  - The part of the program that is repeatedly executed, is called the *loop body*.
  - The loop body can be comprised of a single statement or a compound statement.
- Java offers 3 types of loops:
  1. `while`-loop.
  2. `do-while`-loop
  3. `for`-loop
  - The loops differ in *when the loop body is executed* and *when the loop condition is tested*.
  - Some loops execute the loop body before testing the loop condition, while others test the condition before (possibly) executing the loop body.

`while`-loop

Syntax:

```
while (<loop condition>) {
    /* loop body */
}
```

Semantics:

- Execute the *loop body* while the *loop condition* is `true`.
  - Tests the loop condition before executing the loop body.
  - If the loop condition is `false` at the entry point of the loop, the loop body is skipped.
- The loop condition is a *boolean expression*.

```
number == 0.0
amount <= balance
carryOn   // boolean variable
(amount <= balance) && carryOn
(age >= 18) && (age <= 67) || (age == 80)
```

Example: Sentinel-controlled repetition

Problem: We want to calculate the average of a variable number of floating-point values, given by the user.

```
// Reading n decimal numbers from the keyboard and calculate the average.
import java.util.Scanner;
public class Average {
  public static void main(String[] args) {
    Scanner keyboard = new Scanner(System.in);
    double sentinel = 0.0;
    System.out.println("Input a sequence of numbers, one per line.");
    System.out.printf("Terminate with %.1f.%n", sentinel);
    int count = 0;
    double sum = 0.0;
    double number = keyboard.nextDouble();
    while (number != sentinel) {
      sum = sum + number;
      count = count + 1;
      number = keyboard.nextDouble();
    }
```



```
    if (count == 0) {
      System.out.println("You didn't input any numbers!");
    } else { // count is different from 0
      System.out.printf("The average value is %.1f%n", (sum/count));
    }
  }
}
```

Running the program:

```
Input a sequence of numbers, one per line.
Terminate with 0.0.
0.0
You didn't input any numbers!
```

Exercise: Run the program and check that it calculates the average of values 1.25, 3.33, 4.56 and -2.34 correctly. The last line of the of the program should be as follows.

```
The average value is 1.7
```

Example: counter-controlled repetition

Problem: For a class of 8 students, we want to register how many passed and failed the exam, and what the overall percentage of failure was.

```
// Calculating exam statistics. Version 1.
import java.util.Scanner;
public class ExamResult {
  public static void main(String[] args) {
    Scanner keyboard = new Scanner(System.in);
    int numStudents = 0, passed = 0, failed = 0;
    while (numStudents < 8) {
      System.out.print("Input exam result [1=passed, 2=failed]: ");
      int result = keyboard.nextInt();
      if (result == 1) {
        passed = passed + 1;
      } else {
        failed = failed + 1;
      }

      numStudents = numStudents + 1;
    }
```



```
    System.out.println("Number of students: " + numStudents);
    System.out.println("Number passed: " + passed);
    System.out.println("Number failed: " + failed);
    System.out.printf("Percentage of failure: %.2f%%",
                      (failed/numStudents)*100.0);
  }
}
```

Running the program:

```
Input exam result [1=passed, 2=failed]: 1
Input exam result [1=passed, 2=failed]: 2
Input exam result [1=passed, 2=failed]: 3
Input exam result [1=passed, 2=failed]: 43
Input exam result [1=passed, 2=failed]: 1
Input exam result [1=passed, 2=failed]: 2
Input exam result [1=passed, 2=failed]: 3
Input exam result [1=passed, 2=failed]: 45
Number of students: 8
Number passed: 2
Number failed: 6
Percentage of failure: 0.00%
```

- *The program contains logical errors!*

- Validity of input is not controlled, i.e. the program is not *robust*!
  - We need to verify that the input is valid.
- The calculation of failure percentage is wrong due to use of integer division!
  - Explicit conversion (cast) of values used in the expression is necessary.
  - The cast operator creates a value of the given type using the operand value, without modifying the operand itself.

```
int i = 20;
double d = (double)i / 16; // d is assigned the value 1.25D
```

- We need a more robust program!

Example: counter-controlled repetition — robustness

```
// Calculating exam statistics. Version 2.
import java.util.Scanner;
public class ExamResult2 {
  public static void main(String[] args) {
    Scanner keyboard = new Scanner(System.in);
    int numStudents = 0, passed = 0, failed = 0;
    while (numStudents < 8) {
      System.out.print("Input exam result [1=passed, 2=failed]: ");
      int result = keyboard.nextInt();
      if (result == 1) {
        passed = passed + 1;
        numStudents = numStudents + 1;
      } else if (result == 2) {
        failed = failed + 1;
        numStudents = numStudents + 1;
      } else {
        System.out.println("Invalid input data!");
      }
```

```
    }
    System.out.println("Number of students: " + numStudents);
    System.out.println("Number passed: " + passed);
    System.out.println("Number failed: " + failed);
    System.out.printf("Percentage of failure: %.2f%%",
                      ((double)failed/numStudents)*100.0);
  }
}
```

Running the program:

```
Input exam result [1=passed, 2=failed]: 1
Input exam result [1=passed, 2=failed]: 2
Input exam result [1=passed, 2=failed]: 3
Invalid input data!
Input exam result [1=passed, 2=failed]: 1
Input exam result [1=passed, 2=failed]: 2
Input exam result [1=passed, 2=failed]: 2
Input exam result [1=passed, 2=failed]: 1
Input exam result [1=passed, 2=failed]: 1
Input exam result [1=passed, 2=failed]: 1
Number of students: 8
Number passed: 5
```



```
Number failed: 3
Percentage of failure: 37.50%
```

do-while-loop

Syntax:

```
do {
    /* loop body */
} while (<loop condition>);
```

Semantics:

- Execute the *loop body* while the *loop condition* is `true`.
  - Test the loop condition after executing the loop body.
  - The `do-while`-loop always executes the loop body at least once.

Example: do-while-loop

```
// Calculating exam statistics. Version 3.
import java.util.Scanner;
public class ExamResult3 {
  public static void main(String[] args) {
    Scanner keyboard = new Scanner(System.in);
    int numStudents = 0, passed = 0, failed = 0;
```



```
    do {
      System.out.print("Input exam result [1=passed, 2=failed]: ");
      int resultat = keyboard.nextInt();
      if (resultat == 1) {
        passed = passed + 1;
        numStudents = numStudents + 1;
      } else if (resultat == 2) {
        failed = failed + 1;
        numStudents = numStudents + 1;
      } else {
        System.out.println("Invalid input data!");
      }
    } while (numStudents < 8);
    System.out.println("Number of students: " + numStudents);
    System.out.println("Number passed: " + passed);
    System.out.println("Number failed: " + failed);
    System.out.printf("Percentage of failure: %.2f%%",
                      ((double)failed/numStudents)*100.0);
  }
}
```

Comparing `while` og `do-while`

- In both types of loops:
  - the execution of the loop body stops if the loop condition becomes `false`.
  - the loop condition must have a valid value before it is tested.
  - make sure the loop condition is affected by an action in the loop body, so that it evaluates to `true`, otherwise you risk an *infinite loop*.
  - *Counter-controlled* loops, where the number of repetitions is known in advance, is most commonly used.
- `do-while`-loop
  - The loop body is executed at least once.
  - The loop condition is tested at the end of the loop, after the loop body.
- `while`-loop
  - The loop body may not be executed at all.
  - The loop condition is tested at the start of the loop, before the loop body.
  - Loop variables must be initialised before the entry point of the loop.
  - Useful to control errors in data that can cause the program to terminate, e.g.

```
while (objRef != null) { ... objRef.message() ... }
```


Comparing `while` and `do-while`

- What is the difference between the following 2 loops?

```
...
while (catIsAway) {
    mouse.play();
}
...
```

```
...
do {
    mouse.play();
} while (catIsAway);
...
```

Nested loops

- A loop can contain any statement in its loop body, also another loop.
- Example: Printing a multiplication table

```
int number = 1, limit = 10;
while (number <= limit) {           // Outer loop
  int times = 1;
  while (times <= limit) {          // Inner loop
    int product = number * times;
    System.out.println(number + " x " + times + " = " + product);
    times = times + 1;
  }
  number = number + 1;
}
```


Assertions

How can we guarantee that the program behaves as expected?

- This requires thorough *testing*.
- One way to build trust in the program is using assertions.
- An *assertion* is a boolean expression that we claim is `true` during program execution.
  - The expression concerns results computed during execution.
- Such assertions can be placed in the source code, and will be executed when the program is run.
  - If an assertion holds, we can assume that the part of the program the assertion is about, is working correctly.

Assertions: `assert`-statement

Syntax:

```
assert <BooleanExpression>;                  // simple form
assert <BooleanExpression> : <Expression> ; // general form
```

The semantics of the simple form:

If the truth value of `<BooleanExpression>` *is* `false`*:*

print an error message on the terminal;

abort the program;

else: // truth value of `<BooleanExpression>` *is* `true`

execution continues with the first statement after the `assert`*-statement;*

- The general form leads to the value of `<Expression>` being printed together with the error message, if assertion is `false`.

Enabling assertions

- Assertions are only executed if they are *turned on*:

```
>java -ea Speed
```

- The program can also be run with assertions *turned off*, the assertions are not executed — which is the default behaviour:

```
>java Speed
```

- Compilation is no different than for a program without assertions:

```
>javac Speed.java
```

Example: Using the simple form of assertions

```
import java.util.Scanner;
/**
 *  Assertions -- simple form
 */
public class Speed {
  public static void main(String[] args) {
    Scanner keyboard = new Scanner(System.in);
    System.out.print("Input distance (km): ");
    double distance = keyboard.nextDouble();
    System.out.print("Input time (hours): ");
    double time = keyboard.nextDouble();

    assert distance >= 0.0 && time > 0.0;

    double speed = distance / time;
    System.out.printf("Speed (km/hour): %.1f%n", speed);
  }
}
```


- Compiling and running the `Speed` class:

```
>javac Speed.java
>java -ea Speed
Input distance (km): 12.0
Input time (hours): 3.0
Speed (km/hour): 4.0

>java -ea Speed
Input distance (km): 15.0
Input time (hours): 0.0
Exception in thread "main" java.lang.AssertionError
        at Speed.main(Speed.java:13)
```

- Running the `Speed` class *without assertions*:

```
>java Speed
Input distance (km): 16.0
Input time (hours): 0.0
Speed (km/hour): Infinity
```

```
>java Speed
Input distance (km): 18.0
Input time (hours): 5.0
Speed (km/hour): 3.6
```


Example: Using assertions with additional information

```
import java.util.Scanner;
/**
 *  Assertions -- with additional information
 */
public class Speed2 {
  public static void main(String[] args) {
    Scanner keyboard = new Scanner(System.in);
    System.out.print("Input distance (km): ");
    double distance = keyboard.nextDouble();
    System.out.print("Input time (hours): ");
    double time = keyboard.nextDouble();

    assert distance >= 0.0 && time >0.0 : "Either distance is negative or " +
                                          "time is less than or equal to 0.0";
    double speed = distance / time;
    System.out.printf("Speed (km/hour): %.1f%n", speed);
  }
}
```

- Compiling and running the `Speed2` class:

```
>javac Speed2.java
>java -ea Speed2
Input distance (km): 12.0
Input time (hours): -3.0
Exception in thread "main" java.lang.AssertionError: Either
distance is negative or time is less than or equal to 0.0
        at Speed2.main(Speed2.java:13)
```


How to use assertions

- The previous examples have used assertions to *validate user input*.
  - Use assertions to check that variable do not contain unexpected values, before they are used.
- Do not use assertions to control parameter values in a method if other clients use this method.
  - This control is only performed if assertions is turned on during program execution.
  - Using assertions to control values will print a standard error message (`AssertionError`) that provides little information about the cause of the problem.
- Do not use assertions to perform calculations needed by the program.
  - These calculations will not be performed if assertions are turned off when the program is run.