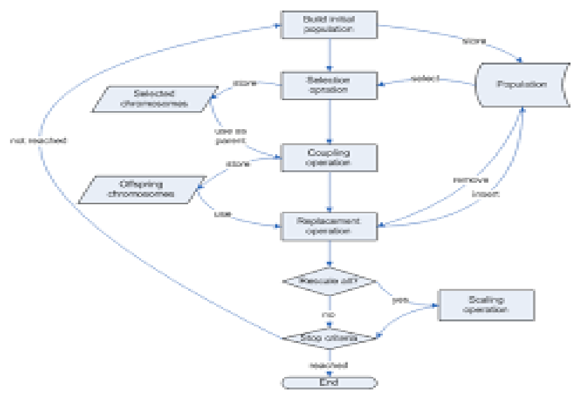
Program control: Sequential Execution

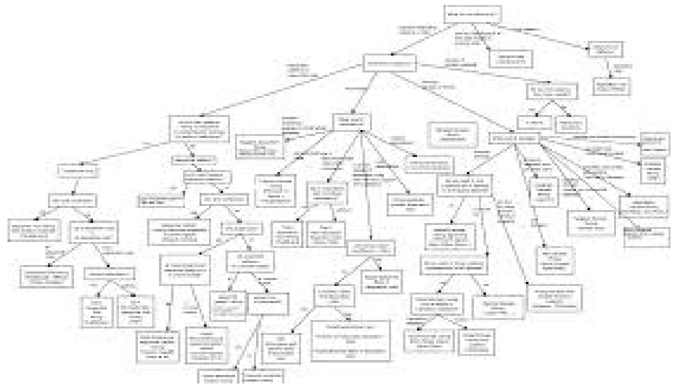
Algorithms and Program Control

- Any computing problem can be solved by executing a series of actions in a specific order.
- An algorithm is a procedure for solving a problem in terms of
  - the actions to execute and
  - the order in which these actions execute
- Specifying the order in which statements (actions) execute in a program is called program control.

Algorithms and Program Control

- Java is a generic programming language suitable for a wide spectrum of applications
- Some applications may need to implement very complex algorithms which may conditionally execute different procedures, or repeat certain operations several times, or skip some stages in certain circumstances





Does Java have enough structural elements to implement algorithms of any complexity?

Structured Program Theorem

• The **Structured Program Theorem** states that algorithms of any complexity can be implemented using only three basic program control structures

1. Sequential execution of statements
2. Conditional execution of statements
3. Repetitive execution of statements

• Java is based on the concept of **Structured Programming** and supports all three program control structures

• Java also supports other flow control statements inherited from other programming languages (continue, break). These flow control structures are redundant and should be avoided as their use indicates poor program structuring.

Sequential execution

- This is the simplest execution control structure: statements are executed one after the other in the order they are written in the program

```
Scanner inp = new Scanner(System.in);
System.out.print("Input two numbers: ");
double d1 = inp.nextDouble();
double d2 = inp.nextDouble();
double average = ( d1 + d2 )/2.0;

double d3 = 3.0*d1*d1 - 4.0*d2;
```

- Sequential execution presumes that statements can refer only variables which have been declared. Forward referencing is not supported.

```
double d1 = 4.0, d2 = 7.5;
double average = ( d1 + d2 + d3)/3.0;

double d3 = d1;
```

d3 has not been declared yet

Compound statements

- A group of sequential statements can be combined into one compound statement
- Compound statements are indicated by curly braces

```
double d1, d2, average;
d1 = 4.0;
{
   d2 = 5.5*d1;
   double sum = d1 + d2;
   average = sum/2.0;
}
average += 2.2;
d1 = average - sum;    // sum doesn't exist here
```

- A variable declared inside a compound statement exists only inside this compound statement

Chained assignments

- A statement can include several assignment operators

```
int a1, a2;
a2 = a1 = 5;   // equivalent to a1=5;  a2=a1;
```

providing that the left operand is always an **`lvalue`**

- Considering that associatively of the assignment operator is right to left the statement can be rewritten as

```
a2 = (a1 = 5);
```

- The expression can be modified further

```
a2 = 4 + (a1 = 5 - 3);  // equivalent to a1 = 5-3;  a2= 4 + a1;
```

- This statement with chained assignments is not valid

```
a2 = 4 + ((a1+4) = 5);  // a1+4 is not an lvalue
```

Statements with chained assignments are supported by Java, but they don't lead to any performance improvement (they are not very common)

Statements with increment/decrement

- Analysis and implementation of statements with increment and decrement operators need special attention

```
int a1=0, a2=0;

a2 = a1++ - 4;   // equivalent to a2 = a1-4;   a1 += 1;
a2 = ++a1 - 4;   // equivalent to a1 += 1;     a2 = a1-4;;
```

- Increment/decrement can be applied only to an lvalue

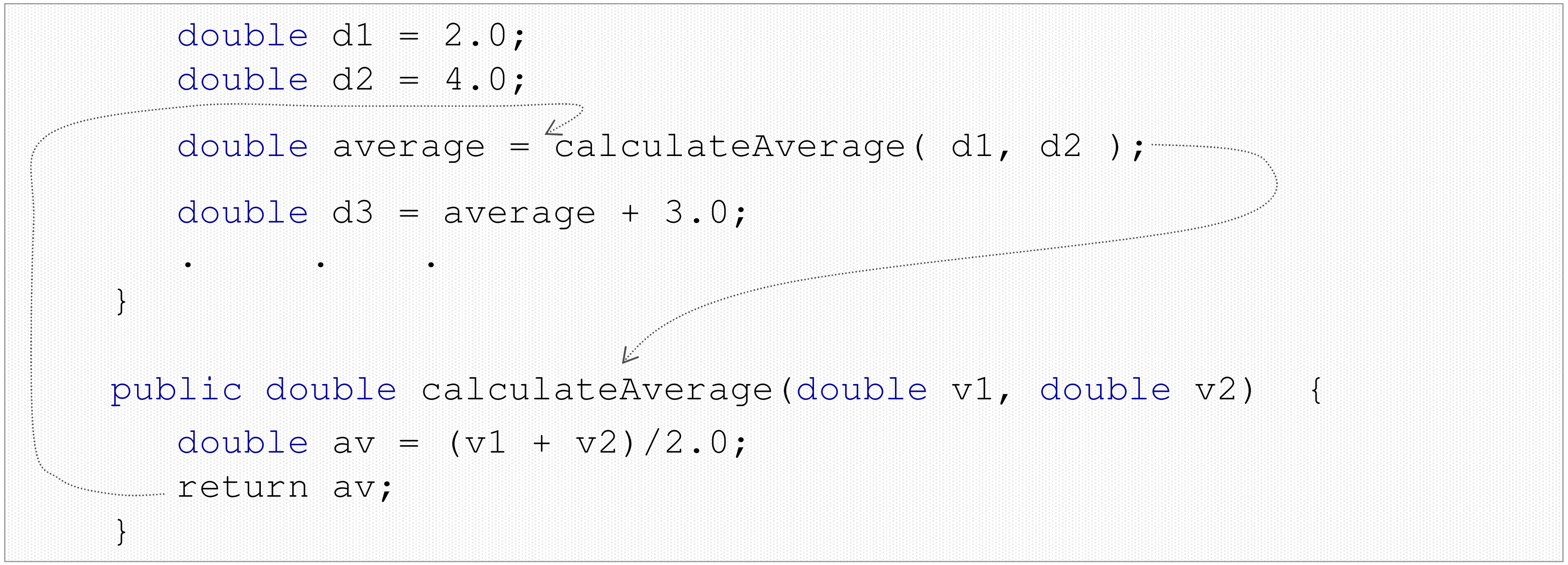
```
int a1=0, a2=0;

a2 = (a1-4)++;  //is not equivalent to a2=a1-4; (a1-4)+=1;-error
```

- You cannot put increment/decrement on the left side of the assignment operator

```
++a2 = a1 - 4;  // compilation error
```

Statements with method calls

- Sequential execution presumes that the next statement cannot be executed before execution of the current one has not been completed



```
        double d1 = 2.0;
        double d2 = 4.0;

        double average = calculateAverage( d1, d2 );

        double d3 = average + 3.0;
        .     .    .
    }

    public double calculateAverage(double v1, double v2)  {
        double av = (v1 + v2)/2.0;
        return av;
    }
```

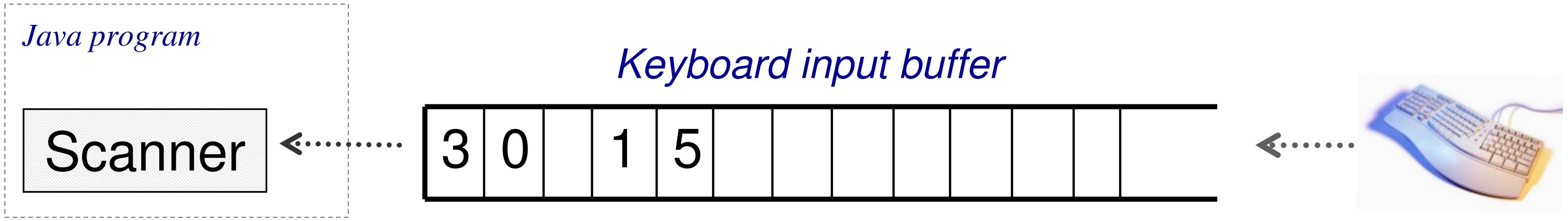
- If a statement includes a method call, it cannot be completed before the method returns the program control back

Statements with user input methods

- Execution of statements with nextDouble() and nextInt() methods may introduce confusion in your program flow

```
Scanner inp = new Scanner(System.in);
System.out.print("Enter an integer number: ");
int num = inp.nextInt();
int result = 2*num*num;
```

- You may expect that a program will pause to get an integer value entered by the user. However, if an integer value is already in the keyboard buffer (left from the previous input), it will be taken without any pause ignoring current user input



Deviation from sequential execution

- Incorrect method calls can break the sequential execution of statements

```
// this method is supposed to display a menu and return an option
public int displayMenu()  {    <
    System.out.println("--- Menu ---");
    System.out.println("1. Calculate area");
    System.out.println("2. Calculate perimeter ");
    System.out.println("Please select an option: ");
    char option = getOption(); // OK
    System.out.println("Press any key to continue: ");
    waitForKeyPress(); // OK
    displayMenu();
    System.out.println("Press Q to quit "); // will never be executed
    waitForKeyPress('Q');                    // will never be executed
    return option;                           // will never be executed
}
```

a trivial call of itself

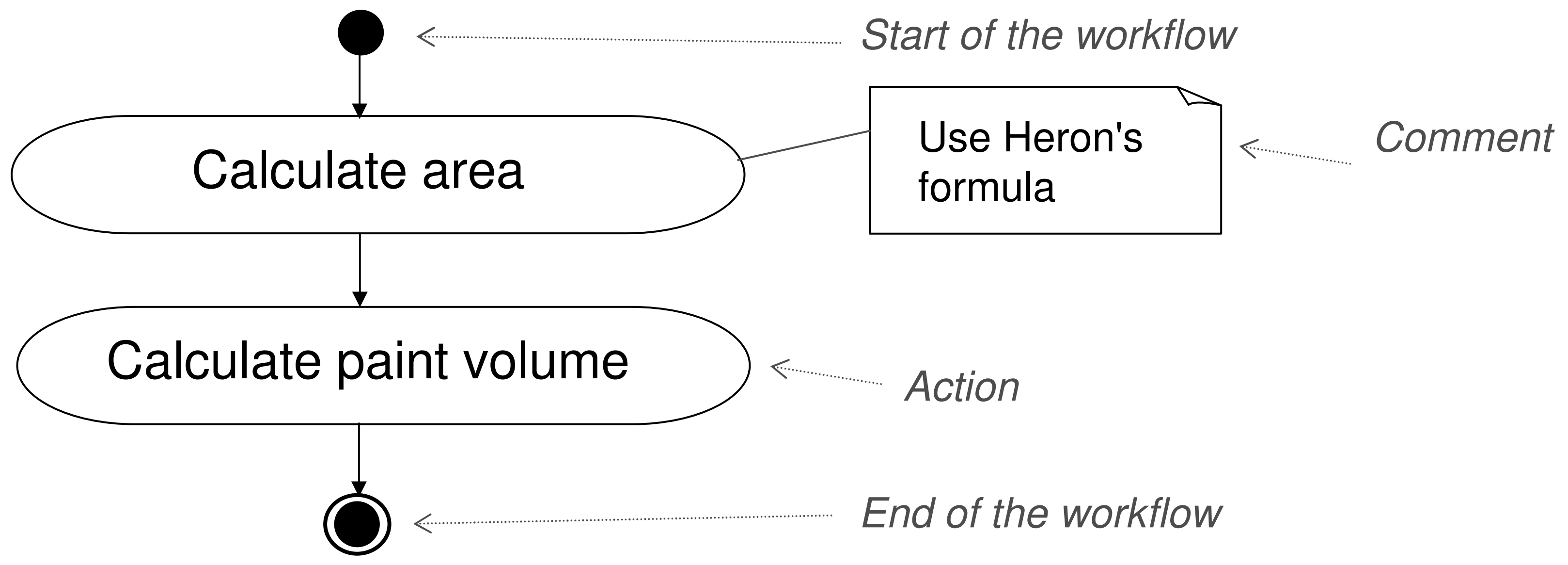
Program writing style

- Sequentially executed statements can be logically subdivided into groups of actions according to their purpose. Blank lines and comments do not affect program execution, but they make the program code easier to follow

```
// this method calculates paint quantity for a triangular area
public double calculatePaintVolume(double a, double b, double c)
{
    // calculate area
    double s = ( a + b + c )/2.0;
    double g = s*(s-a)*(s-b)*(s-c);
    double area = Math.sqrt(g);

    // calculate paint volume
    double paintVolume = DENSITY * area;

    return paintVolume;
}
```

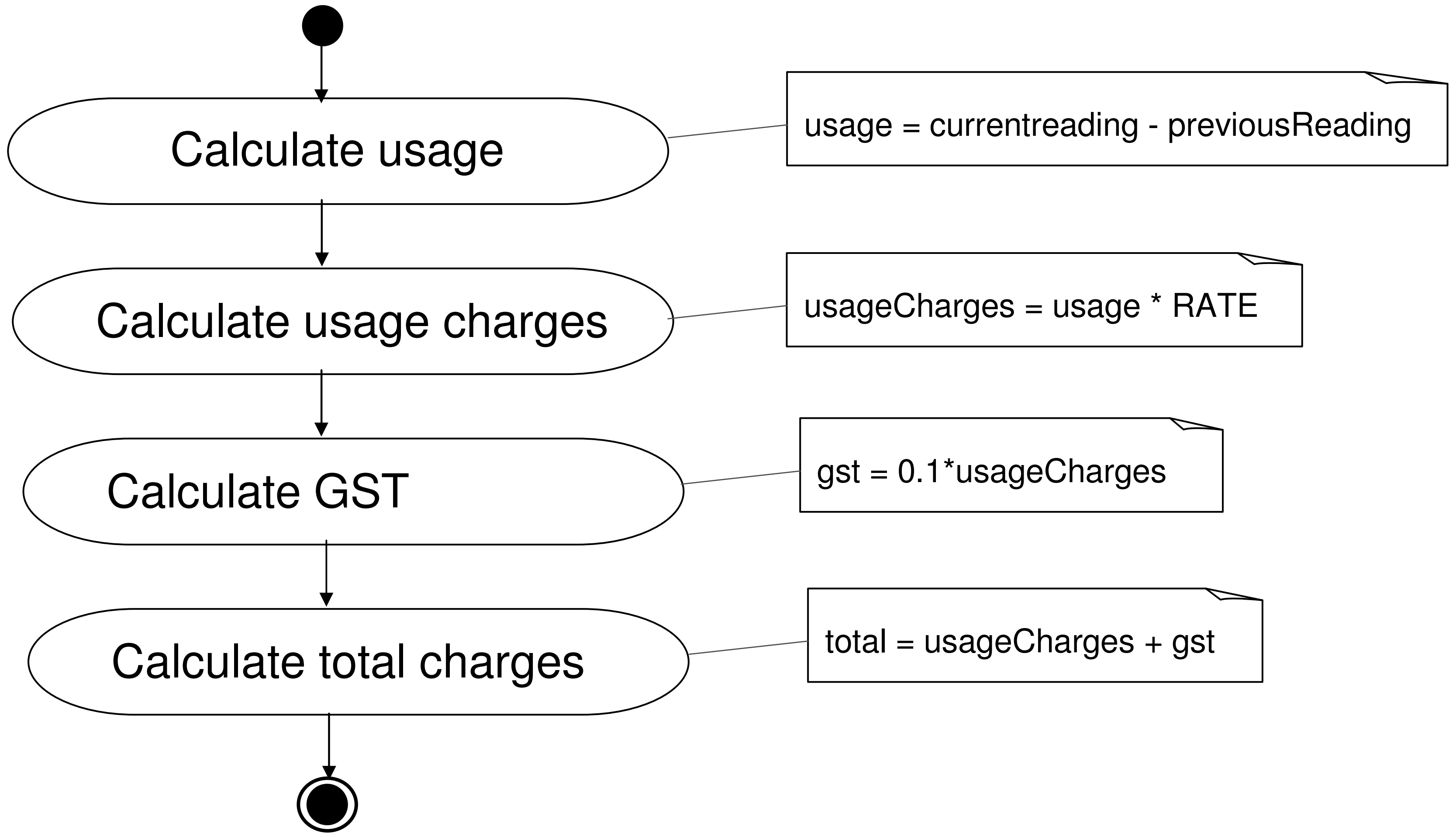
UML activity diagrams

- UML class diagrams describe relationship between classes and required fields and methods inside classes
- How workflow inside methods can be described?
- Activity diagrams can provide graphical representation of workflow of control inside methods



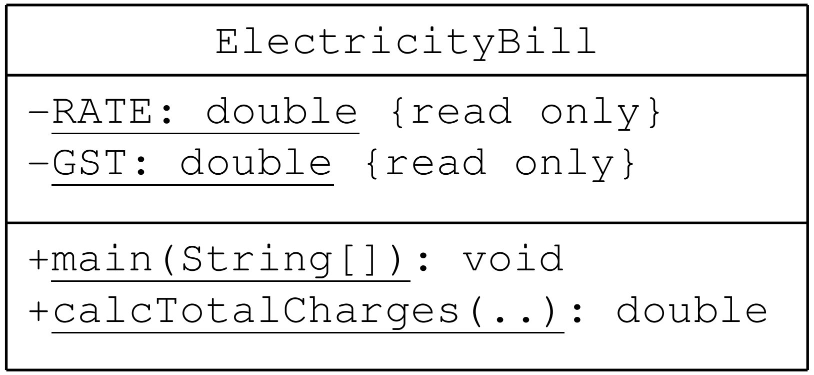
UML activity diagrams

Example: Calculate total charges for electricity usage

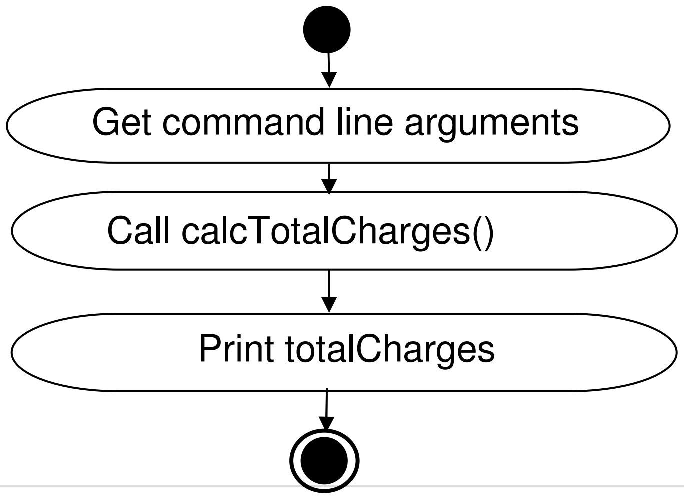


Example: A program that calculates total charges for electricity usage

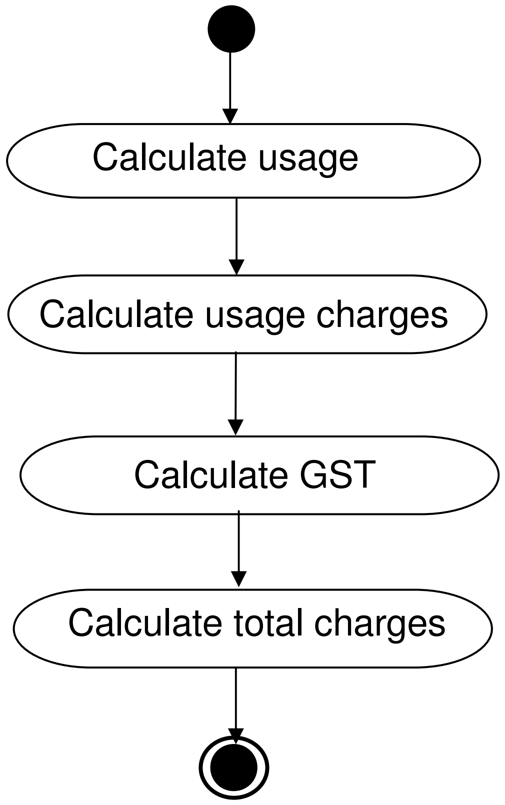
Class diagram



Workflow in main() method



Workflow in calcTotalCharges() method



Example: A program that calculates total charges for electricity usage

ElectricityBill.java

```
class ElectricityBill    {
   private static final double RATE = 0.234;
   private static final double GST = 0.1;

   public static void main(String[] args)
   {
       double prev = Double.parseDouble(args[0]);
       double curr = Double.parseDouble(args[1]);
       double total = calcTotalCharges( prev, curr );
       System.out.println("Total charges: " + total );
   }

   public static double calcTotalCharges(double pr, double cr)
   {
       double usage = cr - pr;
       double usageCharges = usage * RATE;
       double gstCharges = GST * usageCharges;
       return ( usageCharges + gstCharges );
   }
}
```

Relational and logical operators

Selection statements

Introduction

- Before writing a program to solve a problem, have a thorough understanding of the problem and a carefully planned approach to solving it.
- Understand the types of building blocks that are available and employ proven program-construction techniques.
- In this lecture we discuss
  - Java’s relational and logical operators
  - Java’s **`if, if…else`** statements
  - Java’s **`switch`** statement

Equality and Relational Operators

- Equality operators (== and !=)
- Relational operators (>, <, >= and <=)
- Both equality operators have the same level of precedence, which is *lower* than that of the relational operators.
- The equality operators associate from *left to right*.
- The relational operators all have the same level of precedence and also associate from *left to right*.

Algebraic operator	Java equality or relational operator	Sample Java condition	Meaning of Java condition
Equality operators			
=	`==`	`x == y`	x is equal to y
≠	`!=`	`x != y`	x is not equal to y
Relational operators			
>	`>`	`x > y`	x is greater than y
<	`<`	`x < y`	x is less than y
≥	`>=`	`x >= y`	x is greater than or equal to y
≤	`<=`	`x <= y`	x is less than or equal to y

| Equality and relational operators.

Logical Operators

- Java’s **logical operators** enable you to form more complex conditions by combining simple conditions.
- The logical operators are
  - && (conditional AND)
  - || (conditional OR)
  - & (boolean logical AND)
  - | (boolean logical inclusive OR)
  - ^ (boolean logical exclusive OR)
  - ! (logical NOT).

Logical Operators (Cont.)

- The && (**conditional AND**) operator ensures that two conditions are *both true* before choosing a certain path of execution.
- On the next slide are shown all four possible combinations of false and true values for *expression1* and *expression2.*
- Such tables are called **truth tables**. Java evaluates to false or true all expressions that include relational operators, equality operators or logical operators.

expression1	expression2	expression1 && expression2
`false`	`false`	`false`
`false`	`true`	`false`
`true`	`false`	`false`
`true`	`true`	`true`

&& (conditional AND) operator truth table.

Logical Operators (Cont.)

- The || (**conditional OR**) operator ensures that *either or both* of two conditions are true before choosing a certain path of execution.
- Operator && has a higher precedence than operator ||.
- Both operators associate from left to right.

expression1	expression2	expression1 || expression2
`false`	`false`	`false`
`false`	`true`	`true`
`true`	`false`	`true`
`true`	`true`	`true`

\|\| (conditional OR) operator truth table.

Logical Operators (Cont.)

- The parts of an expression containing && or || operators are evaluated only until it's known whether the entire condition is true or false.
- This feature of conditional AND and conditional OR expressions is called **short-circuit evaluation.**
- Example:

```
int a=1, b=0;
boolean flag;
flag = (a==0) && (b==0);
flag = (a==0) || (b==0)
```

this operand is not evaluated

both operands are evaluated

Logical Operators (Cont.)

- The **boolean logical AND** (**&**) and **boolean logical inclusive OR (|)** operators are identical to the && and || operators, except that the & and | operators *always evaluate both of their operands* (i.e., they do not perform short-circuit evaluation).
- This is useful if the right operand has a required **side effect**—a modification of a variable’s value.

Logical Operators (Cont.)

- A simple condition containing the **boolean logical exclusive OR** (^) operator is true *if and only if* one of its operands is true and the other is false.
- If both are true or both are false, the entire condition is false.
- This operator is guaranteed to evaluate both of its operands.

expression1	expression2	expression1 ^ expression2
`false`	`false`	`false`
`false`	`true`	`true`
`true`	`false`	`true`
`true`	`true`	`false`

^ (boolean logical exclusive OR) operator truth table.

UNIVERSITY OF WOLLONGONG

Logical Operators (Cont.)

- The ! (**logical NOT**, also called **logical negation** or **logical complement**) operator "reverses" the meaning of a condition.
- The logical negation operator is a *unary* operator that has only one condition as an operand.
- The logical negation operator is placed *before* a condition to choose a path of execution if the original condition (without the logical negation operator) is false.
- In most cases, you can avoid using logical negation by expressing the condition differently with an appropriate relational or equality operator.

expression	! expression
`false`	`true`
`true`	`false`

`!` (logical NOT) operator truth table.

Control Structures

Selection Statements in Java

- Three types of selection statements.
- **`if`** statement:
  - Performs an action, if a condition is *true*; skips it, if *false*.
  - Single-selection statement—selects or ignores a single action (or group of actions).
- **`if…else`** statement:
  - Performs an action if a condition is *true* and performs a different action if the condition is *false*.
  - Double-selection statement—selects between two different actions (or groups of actions).
- **`switch`** statement
  - Performs one of several actions, based on the value of an expression.
  - Multiple-selection statement—selects among *many different actions* (or *groups of actions*).

`if` Single-Selection Statement

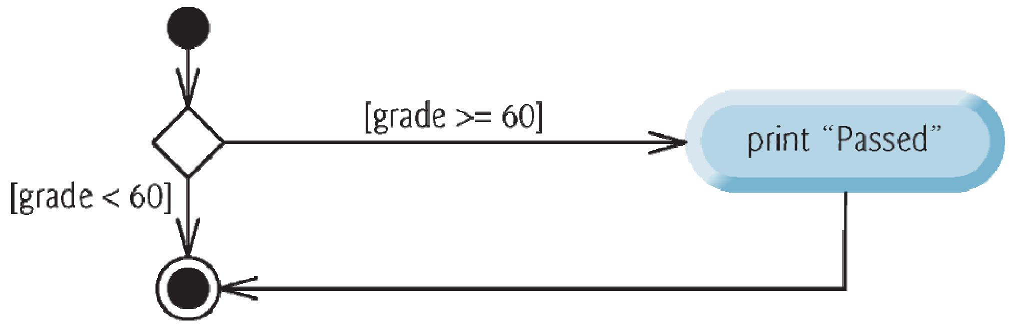
- Pseudocode

```
If student's grade is greater than or equal to 60
  Print "Passed"
```

- If the condition is false, the Print statement is ignored, and the next statement in order is performed.
- Indentation
  - Optional, but recommended
  - Emphasizes the inherent structure of structured programs
- The preceding pseudocode *If* in Java:

```java
if (studentGrade >= 60)
   System.out.println("Passed");
```

- Corresponds closely to the pseudocode.



`if` single-selection statement UML activity diagram.

`if…else` Double-Selection Statement

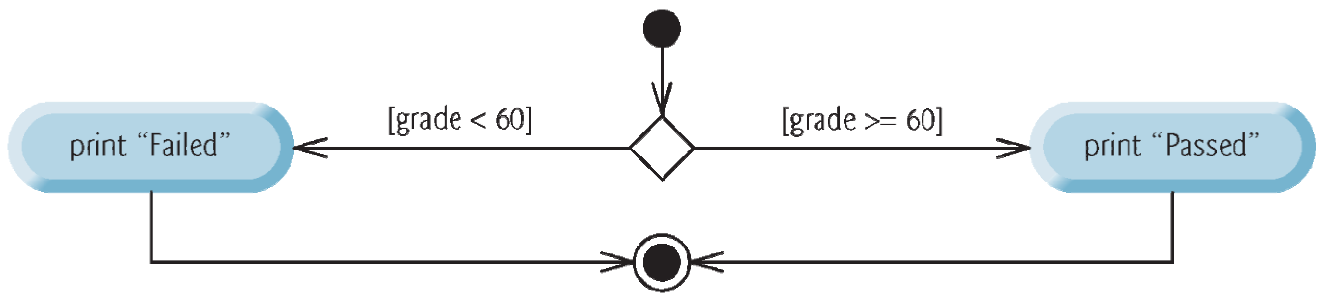
- **`if…else`** double-selection statement—specify an action to perform when the condition is true and a different action when the condition is false.
- Pseudocode

```
If student’s grade is greater than or equal to 60
  Print “Passed”
Else
  Print “Failed”
```

- The preceding *If…Else pseudocode statement in Java:*

```java
if (grade >= 60)
    System.out.println("Passed");
else
    System.out.println("Failed");
```

- Note that the body of the else is also indented.

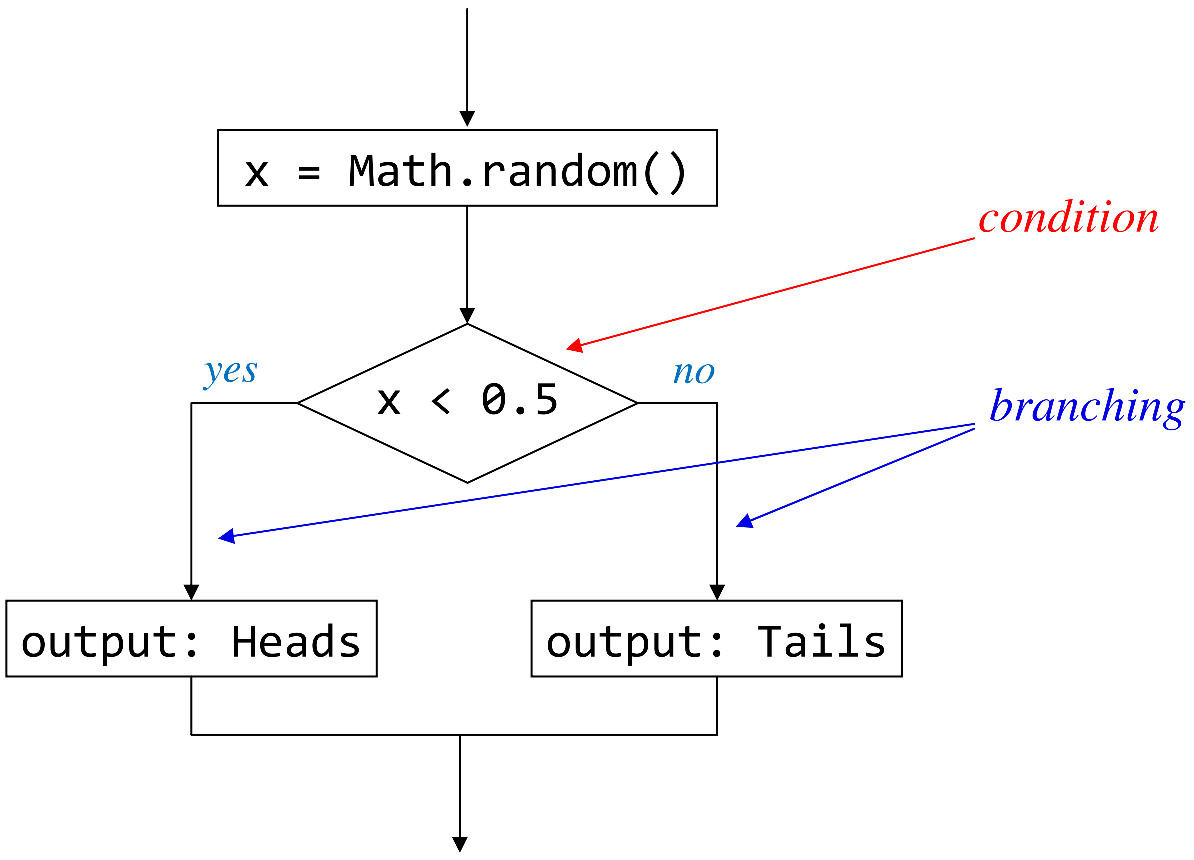


`if...else` double-selection statement UML activity diagram.

A problem: coin slip

- Task
  - Simulate a fair coin flip
- Output
  - Randomly and equiprobably print “Heads” or “Tails”
- Algorithm
  - generate a evenly distributed real number between 0 and 1;
  - if the number is less than 0.5, print “Heads”;
  - else (if the number is greater than or equal to 0.5) print “Tails”.

Flowchart: coin flip



`if` and `if-else` statements

```
if (<boolean expression>) { <statements> }

if (<boolean expression>) { <statements T> }
else                      { <statements F> }
```

```
if ( x < 0.5)
   System.out.println(“Heads”);
else
   System.out.println(“Tails”);
```

when `x >= 0.5`

Example: coin flip

```
public class Flip {

    public static void main(String[] args) {

        // Math.random() returns a value between 0.0 and 1.0
        double x = Math.random();

        // so it is heads or tails 50% of the time
        if ( x < 0.5)
             System.out.println("Heads");
        else
             System.out.println("Tails");
    }
}
```

`if…else` Double-Selection Statement (Cont.)

Nested `if…else` Statements

- A program can test multiple cases by placing if…else statements inside other **`if…else`** statements to create nested ***`if…else`*** *statements.*
- Pseudocode:

```
If student's grade is greater than or equal to 90
  Print "A"
else
  If student's grade is greater than or equal to 80
    Print "B"
  else
    If student's grade is greater than or equal to 70
      Print "C"
    else
      If student's grade is greater than or equal to 60
        Print "D"
      else
        Print "F"
```

`if…else` Double-Selection Statement (Cont.)

- This pseudocode may be written in Java as

```
if (studentGrade >= 90)
   System.out.println("A");
else
   if (studentGrade >= 80)
      System.out.println("B");
   else
      if (studentGrade >= 70)
         System.out.println("C");
      else
         if (studentGrade >= 60)
            System.out.println("D");
         else
            System.out.println("F");
```

- If **`studentGrade >= 90`**, the first four conditions will be true, but only the statement in the **`if`** part of the first **`if…else`** statement will execute. After that, the **`else`** part of the “outermost” **`if…else`** statement is skipped.

if…else Double-Selection Statement (Cont.)

- Most Java programmers prefer to write the preceding nested **if…else** statement as

```
if (studentGrade >= 90)
   System.out.println("A");
else if (studentGrade >= 80)
   System.out.println("B");
else if (studentGrade >= 70)
   System.out.println("C");
else if (studentGrade >= 60)
   System.out.println("D");
else
   System.out.println("F");
```

- The two forms are identical except for the spacing and indentation, which the compiler ignores.

A Problem: leap year

- Task
  - Create a program to test whether a year in the Gregorian calendar is a leap year
  - A leap year is a year containing one additional day (29 February)
- Output
  - Print “true” if a given year is a leap year; otherwise print “false”
- Solution
  - every year that is exactly divisible by 4 is a leap year
  - except for years that are exactly divisible by 100
  - but these centurial years are leap years if they are exactly divisible by 400

A Problem: leap year (revisited)

- Task
  - Create a program to test whether a year in the Gregorian calendar is a leap year
  - A leap year is a year containing one additional day (29 February)
- Output
  - Print “true” if a given year is a leap year; otherwise print “false”
- Solution 2
  - if a year is not divisible by 4, then it is a common year;
  - else if the year is not divisible by 100, then it is a leap year;
  - else if the year is not divisible by 400, then it is a common year;
  - else it is a leap year

Example: leap year (conditional)

```
public class LeapYearConditional {
    public static void main(String[] args) {

        int year = Integer.parseInt(args[0]);
        boolean isLeapYear;

        // not divisible by 4
        if (year % 4 != 0)
            isLeapYear = false;
        // not divisible by 100
        else if (year % 100 != 0)
            isLeapYear = true;
        // not divisible by 400
        else if (year % 400 != 0)
            isLeapYear = false;
        else
            isLeapYear = true;

        System.out.println(isLeapYear);
    }
}
```

`if…else` Double-Selection Statement (Cont.)

Dangling-else Problem

- The Java compiler always associates an **`else`** with the immediately preceding **`if`** unless told to do otherwise by the placement of braces ({ and }).
- Referred to as the dangling-else problem.
- The following code is not what it appears:

```
if (x > 5)
   if (y > 5)
      System.out.println("x and y are > 5");
else
   System.out.println("x is <= 5");
```

- Beware! This nested **`if…else`** statement does *not* execute as it appears. The compiler actually interprets the statement as

```
if (x > 5)
   if (y > 5)
      System.out.println("x and y are > 5");
   else
      System.out.println("x is <= 5");
```

`if…else` Double-Selection Statement (Cont.)

- To force the nested **`if…else`** statement to execute as it was originally intended, we must write it as follows:

```
if (x > 5)
{
   if (y > 5)
      System.out.println("x and y are > 5");
}
else
   System.out.println("x is <= 5");
```

- The braces indicate that the second **`if`** is in the body of the first and that the **`else`** is associated with the *first* ***`if`***.

`Student` Class: Nested **`if…else`** Statement

Class Student

- Class **`Student`** stores a student's name and average and provides methods for manipulating these values.
- The class contains:
  - instance variable **`name`** of type **`String`** to store a Student's name
  - instance variable **`average`** of type **`double`** to store a Student's average in a course
  - a **`constructor`** that initializes the **`name`** and **`average`**
  - methods **`setName`** and **`getName`** to set and get the Student's **`name`**
  - methods **`setAverage`** and **`getAverage`** to *set* and *get* the Student's **`average`**
  - method **`getLetterGrade`**, which uses nested **`if…else`** statements to determine the Student's letter grade based on the Student's **`average`**

Student Class: Nested **`if…else`** Statement

- The constructor and method **`setAverage`** each use *nested* ***`if`*** *statements* to *validate* the value used to set the **`average`**—these statements ensure that the value is greater than 0.0 *and* less than or equal to 100.0; otherwise, **`average`**'s value is left *unchanged*.

- Each if statement contains a *simple* condition. If the condition in line 15 is *true*, only then will the condition in line 16 be tested, and *only* if the conditions in both line 15 *and* line 16 are *true* will the statement in line 17 execute.

```java
//             Student.java
// Student class that stores a student name and average.
public class Student
{
   private String name;
   private double average;

   // constructor initializes instance variables
   public Student(String name, double average)
   {
      this.name = name;

      // validate that average is > 0.0 and <= 100.0; otherwise,
      // keep instance variable average's default value (0.0)
      if (average > 0.0)
         if (average <= 100.0)
            this.average = average; // assign to instance variable
   }

   // sets the Student's name
   public void setName(String name)
   {
      this.name = name;
   }
```

`Student` class that stores a student name and average. (Part 1 of 3.)

```java

   // retrieves the Student's name
   public String getName()
   {
      return name;
   }

   // sets the Student's average
   public void setAverage(double studentAverage)
   {
      // validate that average is > 0.0 and <= 100.0; otherwise,
      // keep instance variable average's current value
      if (average > 0.0)
         if (average <= 100.0)
            this.average = average; // assign to instance variable
   }

   // retrieves the Student's average
   public double getAverage()
   {
      return average;
   }

```

Student class that stores a student name and average. (Part 2 of 3.)

```java
    // determines and returns the Student's letter grade
    public String getLetterGrade()
    {
       String letterGrade = ""; // initialized to empty String

       if (average >= 90.0)
          letterGrade = "A";
       else if (average >= 80.0)
          letterGrade = "B";
       else if (average >= 70.0)
          letterGrade = "C";
       else if (average >= 60.0)
          letterGrade = "D";
       else
          letterGrade = "F";

       return letterGrade;
    }
} // end class Student
```

`Student` class that stores a student name and average. (Part 3 of 3.)

`switch` Multiple-Selection Statement

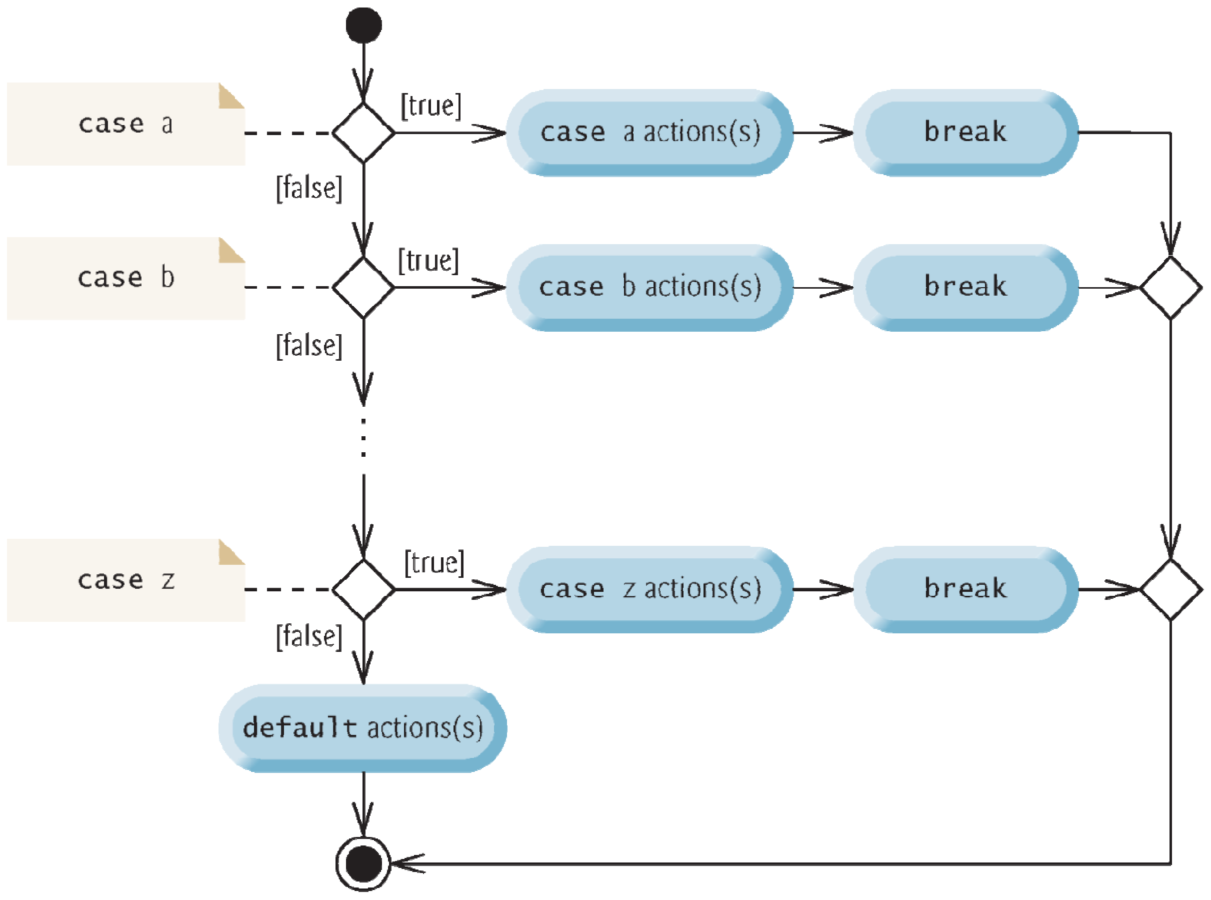
- **`switch` multiple-selection statement** performs different actions based on the possible values of a **constant integral expression** of type **`byte`**, **`short`**, **`int`** or **`char`**.

`switch` Multiple-Selection Statement

- The **`switch`** statement consists of a block that contains a sequence of **`case`** **labels** and an optional default **`case`**.
- The program evaluates the **controlling expression** in the parentheses following keyword **`switch`**.
- The program compares the controlling expression's value (which must evaluate to an integral value of type **`byte`**, **`char`**, **`short`** or **`int`**, or to a **`String`**) with each case label.
- If a match occurs, the program executes that **`case`**'s statements.
- The **`break`** **statement** causes program control to proceed with the first statement after the **`switch`**.

`switch` Multiple-Selection Statement

- Most **`switch`** statements use a **`break`** in each **`case`** to terminate the **`switch`** statement after processing the **`case`**.
- The **`break`** statement is not required for the **`switch`**'s last **`case`** (or the optional default **`case`**, when it appears last), because execution continues with the next statement after the **`switch`**.



| switch multiple-selection statement UML activity diagram with break statements.

`switch` Multiple-Selection Statement

- **`switch`** does *not* provide a mechanism for testing ranges of values—every value must be listed in a separate **`case`** label.
- Note that each **`case`** can have multiple statements.
- **`switch`** differs from other control statements in that it does not require braces around multiple statements in a **`case`**.
- Without break, the statements for a matching **`case`** and subsequent **`case`**s execute until a **`break`** or the end of the **`switch`** is encountered. This is called “falling through.”
- If no match occurs between the controlling expression’s value and a **`case`** label, the default **`case`** executes.
- If no match occurs and there is no default **`case`**, program control simply continues with the first statement after the **`switch`**.

`switch` Multiple-Selection Statement

- When using the `switch` statement, remember that each `case` must contain a constant integral expression.
- An integer constant is simply an integer value.
- In addition, you can use **character constants**—specific characters in single quotes, such as 'A', '7' or '$'—which represent the integer values of characters.
- The expression in each `case` can also be a **constant variable**—a variable that contains a value which does not change for the entire program. Such a variable is declared with keyword `final`.
- Java has a feature called `enum` types—`enum` type constants can also be used in `case` labels.

A problem: rock-paper-scissors

- Task
  - Simulate a rock-paper-scissors hand
- Output
  - Randomly and equiprobably print “rock” or “paper” or “scissors”
- Algorithm
  - generate a evenly distributed real number between 0 and 1;
  - if the number is less than 1/3, print “rock”;
  - else if the number is less than 2/3, print “paper”;
  - else print “scissors”.

A problem: rock-paper-scissors

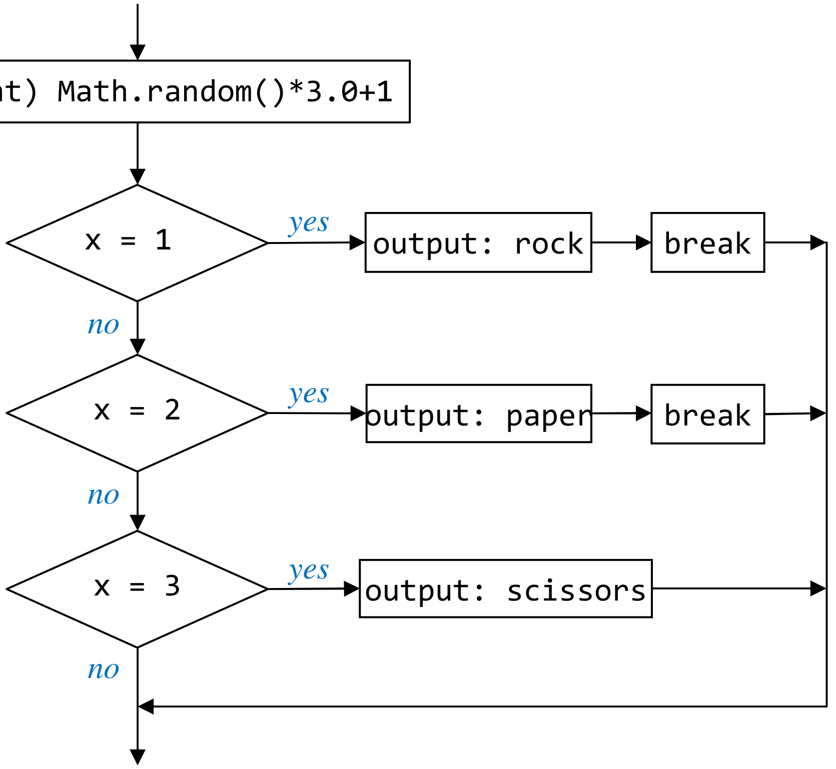
- Task
  - Simulate a rock-paper-scissors hand

- Target
  - Randomly and equiprobably print “rock” or “paper” or “scissors”

- Algorithm 2
  - generate 3 evenly distributed real number between 1 and 3;
  - when the number is 1, print “rock”;
  - when the number is 2, print “paper”;
  - when the number is 3, print “scissors”;

UNIVERSITY OF WOLLONGONG

Flowchart: rock-paper-scissors



Example: rock-paper-scissors (switch)

```
public class RockPaperScissorsSwitch {

  public static void main(String[] args) {

    // Math.random() returns a value between 0.0 and 1.0
    double x = Math.random();
    // create a choice variable to represent rock or paper or scissors
    int hand = (int) (x*3.0)+1;

    switch(hand) {
    case 1:
      System.out.println("rock");
      break;
    case 2:
      System.out.println("paper");
      break;
    case 3:
      System.out.println("scissors");
    }
  }
}
```

Example: what a day

```java
public class TestEnum {
    public static enum Day {SUNDAY, MONDAY, TUESDAY, WEDNESDAY,THURSDAY, FRIDAY,
    SATURDAY}

    public static void main(String[] args) {
        int theDay = Integer.parseInt(args[0]);

        switch (theDay) {
        case 1:
            System.out.println(Day.MONDAY + " is bad.");
            break;
        case 5:
            System.out.println(Day.FRIDAY + " is better.");
            break;
        case 6: case 7:
            System.out.println("Weekends are best.");
            break;
        default:
            if (theDay > 7) {
                System.out.println("Please enter a number < 7.");
            } else {
                System.out.println("Midweek days are so-so.");
            }
            break;
        }
    }
}
```

Example: where is the error

```
public class Test{

    public static void main(String[] args) {
        int number = Integer.parseInt(args[0]);

        switch (number) {
        case 0:
            System.out.println("Number is 0");
            break;
        case 1:
            System.out.println("Number is 1");

        case 2:
            System.out.println("Number is 2");
            break;
        default:
            System.out.println("Number is not in range 0-2");;
        }
    }
}
```

Suggested reading

Java: How to Program (Early Objects), 10th Edition

- Chapter 4: Control statements: Part 1
  - 4.1, 4.2, 4.4, 4.5, 4.6, 4.7
- Chapter 5: Control statements: Part 2
  - 5.6, 5.7