HST 952

Computing for Biomedical Scientists

Lecture 3

Recap

Algorithm:

A series of unambiguous steps for solving a problem

Object-oriented programming:

Approach to programming that relies on objects and interactions among objects

Recap

Object:

- Has attributes (properties)
- Has methods (methods define the way the object interacts with "the world")

Each method associated with an object **implements an algorithm** that solves a particular problem that is specific to the object

Recap

Class definition:

- Provides a template of the attributes and methods associated with a kind of object
- An object is a **specific instantiation of a class**

Recap

Example class: Vehicle

Attributes might include:

- Number of wheels
- Type
- Color
- Manufacturer
- Mileage

Methods might include:

- getMileage()
- setMileage(int miles)
- setMileage(Vehicle otherv)

Recap

Objects (instances) of class Vehicle

VolvoS60

Number of wheels : 4
Type: Car
Color: Blue
Manufacturer: Volvo
Mileage: 5,000 miles

DodgeNeon

Number of wheels : 4
Type: Car
Color: Green
Manufacturer: Dodge
Mileage: 10,000 miles

Recap

Objects (instances) of class Vehicle

HarleyDynaGlide

Number of wheels : 2
Type: Motorcycle
Color: Black
Manufacturer: Harley-Davidson
Mileage: 1,000 miles

MackVision

Number of wheels : 18
Type: Truck
Color: Red
Manufacturer: Mack
Mileage: 50,000 miles

Recap

Imagine that we want to keep track of how many tons a truck can haul, but are not interested in this for cars or motorcycles

One approach to doing this could involve making the Vehicle class a **parent** class (superclass) of the Car, Motorcycle, and Truck (sub)classes

Recap

- Attributes and methods common to all vehicles would be described in the Vehicle class
- Attributes that are specific to a Car, Truck, or Motorcycle would be specified in the appropriate subclass
- For our example, the Truck class could have the attribute *tonnage* and methods *getTonnage()* and *setTonnage()* which the Car and Motorcycle classes would not have

This Lecture

We will examine some Java constructs that serve as building blocks/tools for implementing an algorithm in Java

Java constructs include:

- Expressions (boolean, arithmetic, logical, etc.)
- Operators (comparison, logical, arithmetic)
- Statements (assignment, branch, loop, etc.)

Our focus will be on constructs that are important for determining the flow of control in a program

Boolean Expressions

- Evaluate to *true* or *false*
- May be used with other language constructs to determine the *flow of control* in a program
- Involve comparison operators and/or logical operators

Comparison Operators

Symbol	Description
==	is equal to
!=	is not equal to
<	is less than
<=	is less than or equal to
>	is greater than
>=	is greater than or equal to

Comparison Operators

- The last four comparison operators may only be used to compare values from ordered sequences (numbers, characters)
- Examples of boolean expressions:

‘y’ < ‘z’	(evaluates to true)
5.9 >= 23	(evaluates to false)
true == false	(evaluates to false)
25 != 25	(evaluates to false)

Logical Operators

- Binary operator (requires two operands)
- Unary operator (requires just one operand)

Symbol	Description
&&	and (binary operator)
||	or (binary operator)
!	not (unary operator)

Logical Operators

- How the && operator works

  true && true evaluates to true

  true && false evaluates to false

  false && true evaluates to false

  false && false evaluates to false

- Once one of its operands is false, a boolean expression involving && evaluates to false

Logical Operators

- How the || operator works

  true || true evaluates to true

  true || false evaluates to true

  false || true evaluates to true

  false || false evaluates to false

- Once one of its operands is true, a boolean expression involving || evaluates to true

Logical Operators

- How the ! operator works
  - ! true evaluates to false
  - ! false evaluates to true

- a boolean expression involving ! evaluates to the opposite of the operand's truth value

Logical Operators

- ! has the highest precedence
- && has the next highest precedence
- || has the lowest precedence
- may use parentheses to group parts of an expression to force a particular order of evaluation

Logical Operators

What do the boolean expressions in these assignment statements evaluate to?

```
boolean firstBool = true || false && false;
boolean secondBool = (true || false) && false;
boolean thirdBool = firstBool || secondBool;
boolean fourthBool = !true || false;
boolean fifthBool = !(true && false);
```

Flow of control

- May need to write methods that have to choose one path out of several possible paths

  (programs for an ATM machine have to choose how much money to dispense based on your input and balance)
- May need to repeat an action several times to obtain a desired result

  (e.g., solution to the GCD problem)

Branching statements

- Allow us to make a choice of an action given two or more options
- Use implicit or explicit boolean expressions in making the choice

Examples of Java branching statements:

- if /if-else statement (uses explicit boolean expr.)
- switch statement (uses implicit boolean expr.)

if/if-else statement

Syntax:

```
if (boolean expression)
    {
        // perform action1
    }
else
    {
        // perform action2
    }
```

if/if-else statement

- The actions following the if part of the statement are performed only when the boolean expression evaluates to true
- If the boolean expression evaluates to false, the actions following the else part of the statement are executed (when an else is present)
- The curly braces group together all the actions to be performed
- If only one action is to be performed, the curly braces may be omitted

if/if-else statement

Example:

```
if (score < 100)
    System.out.println("Score is less than 100);
if (score < 90)
    System.out.println("Score is less than 90);
if (score < 80)
    System.out.println("Score is less than 80);
```

what happens when score is 50?

if/if-else statement

```
if (score < 80)
    System.out.println(“Score is less than 80);
else {
    if (score < 90)
        System.out.println(“Score is less than 90);
   else {
       if (score < 100)
    System.out.println(“Score is less than 100);
   }
}
```

what happens when score is 50?

if/if-else statement

Easier way to write the same sequence of statements:

```
if (score < 80)
    System.out.println("Score is less than 80);
else if (score < 90)
        System.out.println("Score is less than 90);
else if (score < 100)
          System.out.println("Score is less than 100);
```

what happens when score is 120?

Switch statement

- Multi-way branching statement
- Makes a decision on which branch to take based on the value of an integer or character expression (called the *controlling expression*)
- Can be mapped to an equivalent if-else sequence (but not always the other way around)
- Syntax next

Switch statement

```
switch(int or char expression) {
    case int or char constant:
        // perform action1
        break;
    case int or char constant:
        // perform action2
        break;
    default:
        // perform action3
        break;
}
```

Switch statement

Note:

- there is an implicit equality comparison between the int or char expression in the switch and the constant in a case
- default case is optional
- break statement ends each case and is necessary to prevent *fall-through*

Switch statement

Example (firstInitial is a variable of type char):

```
switch(firstInitial) {
    case ‘A’:
        System.out.println(“Instructor is Aziz”);
        break;
    case ‘O’:
        System.out.println(“Instructor is Omolola”);
        break;
```

Example continued on next slide

Switch statement

```
    case ‘Q’:
        System.out.println(“Instructor is Qing”);
        break;
    default:
        System.out.println(“Unknown instructor”);
        break;
}
```

Switch statement

Another example (myNum is a variable of type int):

```
switch(myNum) {
    case 1:
        System.out.println("The number is one");
    case 2:
        System.out.println("The number is two");
    default:
        System.out.print("The number is neither");
        System.out.println(" one nor two");
}
```

What happens when myNum is 1?

Switch statement

Another example (myNum is a variable of type int):

```
switch(myNum) {
    case 1:
        System.out.println("The number is one");
    case 2:
        System.out.println("The number is two");
    default:
        System.out.print("The number is neither");
        System.out.println(" one nor two");
}
```

What happens when myNum is 3?

Switch statement

Example: Cases with no breaks (firstInitial is a variable of type char):

```
switch(firstInitial) {
    case ‘A’:
    case ‘a’:
        System.out.println(“Instructor is Aziz”);
        break;
    case ‘O’:
    case ‘o’:
        System.out.println(“Instructor is Omolola”);
        break;
```

Example continued on next slide

Switch statement

```
    case ‘Q’:
    case ‘q’:
        System.out.println(“Instructor is Qing”);
        break;
    default:
        System.out.println(“Unknown instructor”);
        break;
}
```

Fall-through is desired in this example

Loop Statements

- Allow us to repeat an action/several actions until a particular condition is met

Examples of Java loop statements:

- while
- do-while
- for

While Loop

Syntax:

while(*boolean expression*)
{
   // perform action(s)
}

While Loop

- The actions in the loop body are performed only when the boolean expression evaluates to true
- If the boolean expression is true, the actions are performed until it is false
- If the boolean expression is never false, we may have an *infinite loop* (actions performed until program runs out of memory resources, etc.)
- This implies that there should be a statement in the body of the loop that alters the loop's course

While Loop

Example:

```
int iterator = 0;
while(iterator < 10)
{
    System.out.println("Iterator is " + iterator);
    iterator = iterator + 1;
    // another way of writing the line above is iterator += 1
}
```

Do-While Loop

Syntax:

```
do
{
   // perform action(s)
} while(boolean expression);
```

Do-While Loop

- The actions in the loop body are performed until the boolean expression evaluates to false
- If the boolean expression is never false, we may also have an *infinite loop*
- This implies that there should be a statement in the body of the loop that alters the loop's course

  (ensures that the boolean expression is eventually false)

Do-While Loop

Example:

```
int iterator = 0;
do
{
    System.out.println("Iterator is " + iterator);
    iterator = iterator + 1;
   // another way of writing the line above is iterator++
} while(iterator < 10);
```

How does this differ from the while loop example in terms of what gets printed?

For Loop

Syntax:

for(*initializer*; *boolean expression*; *update action*)

```
{
   // perform actions
}
```

For Loop

Example:

```
int iterator;
for(iterator = 0; iterator < 10; iterator++)
{
   System.out.println("Iterator is " + iterator);
}
```

Read

Chapter 3