CS 106
Introduction to Computer Science I

06 / 04 / 2015

Instructor: Michael Eckmann
Today’s Topics

• Questions / comments?
• Calling methods (noting parameter(s) and their types, as well as the return type)
• Logical operators (on booleans)
• switch statement
• Loops
  – while
  – for
  – do while
Review

• Method call
• Parameter
• What a method returns
  – How to “capture” what a method returns
• Packages / classes / methods
method terminology

• void – another “type” in Java which is used when no value is needed.
method terminology

• **Parameter** – methods have 0 or more parameters. These specify what types of values are used when a method call is made.

• **Calling a method** – invoking a method by giving its name in a statement in your program:
  
  – e.g.
  – System.out.println(“Hey”); // method call for println method
  – height = Integer.parseInt(height_str); // method call for parseInt method

• **Note:** A String is being passed in as a parameter for println. Same for parseInt.
method terminology

• **Return type of a method** – This is what type the result of a method call gives.

  e.g.
  – `System.out.println(“Hey”); // nothing is returned (void)`
  – `height = Integer.parseInt(height_str); // an int is returned`

• The return type and number of parameters and types are all specified in the definition of a method. For the Java API methods, we can look this stuff up online.
Logical operators

• Not !
• And & or &&
• Or | or ||

• The double ones (&& and ||) are “short circuit” operators --- if left operand is sufficient to determine truth or falsity, then the right operand is not evaluated.
• The single ones (& and |) always evaluate both operands.
• This really only matters if you have side-effects,
• e.g. Like a ++ or --
Logical operators

- Precedence first !, then && then ||.
- Parentheses are still performed first. Should be used to force the order that the programmer desires.

- Not operator, !, takes one operand to its right.
- And operator, &&, works on two operands
- Or operator, ||, works on two operands
Logical operators

NOT:
!true = false,  !false = true

AND:
true && true = true
true && false = false
false && true = false
false && false = false

OR:
true || true = true
true || false = true
false || true = true
false || false = false
Logical operators

• Examples:

    boolean end_of_file = false;
    //...
    if (! end_of_file)
        // ...

    if (!(age >=21))
        // ...

Logical operators

if (!(age >=21))
    // ...

    // how else might we write the above?

if ((age <= 12) || (age >= 65))
    // get a discount at the movies...

if (you_are_rich && you_are_good_looking)
    System.out.println(“You got it made.”);
    // what types must you_are_rich and you_are_good_looking, be?
Logical operators

if ((age >= 21) && (++count > 10))
   // ...

here if age is not >= 21 then regardless of what's on the right hand side of the &&, the whole thing will be false (because false AND anything = false), because && is the “short circuit” format of the AND operator, (++count > 10) will not be evaluated.

if ((age >= 21) & (++count > 10))
   // ...

here, even if age is not >= 21 the right hand side WILL be evaluated
What's the difference in the result?
Logical operators

&& (short circuit AND) vs. & (non-short circuit AND)

if ((age >=21) && (++count > 10))
    // ...
if ((age >=21) & (++count > 10))
    // ...

The difference is, that 1 will be added to count (and compared to 10) regardless of whether or not age is >= 21 in the second if, but in the first if, 1 will be added to count (and compared to 10) only if age is indeed >=21.
Logical operators

if ((height >= 72) || (countPeople() <= 100))
   // ...

here if height is >= 72, then the whole condition is guaranteed to be true (because true OR anything = true) and the || causes the right hand side NOT to be evaluated. Therefore, the method countPeople() would not be called

if ((height >= 72) | (countPeople() <= 100))
   // ...

here even if height is >=72, the righthand side WILL be evaluated because of the use of the non-short circuit OR |.
Logical operators

|| (short circuit OR) vs. | (non-short circuit OR)

if ((height >= 72) || (countPeople() <= 100))
   // ...

if ((height >= 72) | (countPeople() <= 100))
   // ...

The difference is, that countPeople() method will ALWAYS be called in the second if but won't be called in the first if if the left hand side evaluates to true
Logical operators

if ((height >= 72) || (countPeople() <= 100))
   // ...

Would countPeople() be called (and it's returned value compared to 100) if height happens to be < 72?
Why or why not?
Logical operators

if ((height >= 72) || (countPeople() <= 100))
    // ...

Would countPeople() be called (and its returned value compared to 100) if height happens to be < 72?

Why or why not?

Yes it would because the LHS being false doesn't determine the outcome of the whole condition, even though we are using the short-circuit version of OR.
Logical operators

if ((age >= 21) && (++count > 10))
  // ...

Would ++count > 10 occur if age is >= 21?
Why or why not?
Logical operators

if ((age >= 21) && (++count > 10))
   // ...

Would ++count > 10 occur if age is >= 21?
Why or why not?

Yes it would because the LHS being true doesn't determine the outcome of the whole condition, even though we are using the && short-circuit version of AND.
Switch

switch (some_var)
{
    case 1:
        // do stuff if some_var's value is 1
        break;
    case 2:
        // do stuff if some_var's value is 2
        break;
    default:
        // do stuff if none of the cases executed.
}
Switch

break; // this statement exits out of the curly braces in which is it enclosed

and program continues after the }

let's write some code showing an example use of switch to emulate a menu like,

1. print *'s
2. print $'s
3. print @'s
4. print !'s
Switch

• Important comments about switch.

• The value in the switch is compared in order to the cases. If it is unequal to a case, it gets compared to the case below it and so on. When/if it gets to an equal case, then the code within that case is executed and if there is no break, the *code in lower cases continues to be executed until a break statement or the* `}`.

• **BE CAREFUL** about putting break; statements in. Unless you're trying to do something fancy, they **should be used at the end of each case.**

• If the value in the switch doesn't equal any of the cases, the optional default case executes.
while loops

- While loops are used to repeat lines of code until some condition is met.

  e.g.

  ```
  while (condition) {
    // statements to do while condition is true.
  }
  ```

- condition is evaluated and if true the code within the curly braces executes. Then condition is tested again and if true the code within the curly braces executes. This continues until the condition is false (or a break; statement is hit.)
while loops

• It's extremely important for you to know exactly what code is executed in what order for you to be able to follow what a program is doing. Let's follow the code below line by line in order. (assume that it is inside a main method of a class.)

```java
int x = 1;
System.out.println("Just before the loop");
while (x < 4)
{
    System.out.print("*");
    x++;
}
System.out.println("\nJust got out of the loop");
```
while loops

int total = 0;
while (total < 100)
{
    total = total + 10;
    System.out.println("total = "+ total);
}
System.out.println("finished printing totals");

// What would this code do?
while loops

• Sentinel controlled loops vs. counter controlled loops.
• Counter controlled loops are those loops that usually execute a set number of times every time.
• A sentinel is a value that is used to terminate a loop. For instance, if you wanted a user to enter in some numbers (but you didn't know how many s/he was going to enter) to be averaged (and assuming that they are all supposed to be positive) we can ask the user to enter in -1 after s/he enters all the numbers so we know s/he's finished.
• Let's see an example.
while loops

int inputNum=0, count = 0;
String inputStr;
double total = 0.0;
while (inputNum != -1)
{
    inputStr = JOptionPane.showInputDialog("Enter a number to be averaged");
    inputNum = Integer.parseInt(inputStr);
    total += inputNum;
    count++;
}
System.out.println("average = \(\frac{\text{total}}{\text{count}}\));

// any problems with this code???? Hint: yes there's a logic error.
while loops

• Sentinel controlled loops vs. counter controlled loops.
• Counter controlled loops are those loops that usually execute a set number of times every time.
• As a programmer what value makes sense to choose as the sentinel is specific to the situation.
• Could anyone characterize what would be a bad value to choose as the sentinel?
while loops

• Order of execution of a while loop.
  • First the condition is tested.
    – If it is true the statements within the curly braces execute
    – Then condition is tested again
    – If it is true the statements within the curly braces execute again
    – Then condition is tested again
    – ...
    – This continues until the condition is false when tested. At that time, program control continues with the code after the right curly brace.

• Note: It IS possible that even on the first test of the condition, that it may be false. If it is, the statements within the loop don't execute at all.
while loops

• What happens to loops that do not alter the value of a variable in the condition?
while loops

• What happens to loops that do not alter the value of variable(s) in the condition?

• If that condition is true the first time, it will stay true forever and we will have an infinite loop.

• e.g.

```java
int x = 2;
while (x > 0)
{
    System.out.println("x = " + x);
}
```
while loops

• Any other ways to get an infinite loop?
while loops

• Would this be infinite?

```java
int x = 2;
while (x > 0)
{
    System.out.println("x = " + x);
    x++;  
}
```
Loop terminology

• We say that a loop *iterates* some number of times.
• One *iteration* of a loop is one execution of the block of code inside it.
The `for` loop is a counter controlled repetition structure that compactly lets a programmer specify the control variable, its initial value, the condition to be checked each time through the loop and the amount to increment (or decrement) the control variable.

```
for ( expression1; expression2; expression3 )
    statement or compound statement
```
for loops

for ( expression1; expression2; expression3 )
statement or compound statement

where expression1 contains the declaration and initialization of the control variable

expression2 contains the continuation condition and the ending value of the control variable

expression3 contains the increment or decrement of the control variable (and the amount of this change in value each time through the loop)
for loops

for ( expression1; expression2; expression3 )
{
    statements
}

Order of operations:
1. expression1 executes first (and only once)
2. expression2 executes and if the result is false, program control goes to the code after the right curly brace
3. if the result is true, then the statements within the curly braces of the for loop execute in order then expression3 executes (which usually alters the value of the loop variable.) Then “goto” 2 above.
for loop example

for ( int x = 10; x >= 1; x-- )
{
    do some statements here ....
}

• note that we used x-- to subtract 1 from the counter each time through the loop, we could have used:
  
x = x - 1 or x -= 1 or --x
for loop

• Each of the three expressions in the for loop structure are optional.
• don’t need expression1 if control variable declared and initialized before the loop
• don’t need expression2 if you desire having an infinite loop
• don’t need expression3 if you change the value of the control variable within the loop
for loop

for ( int cntr = 1;  cntr <= 20;  cntr = cntr + 1 )
{
    do some statements here ....
}

• this is essentially the same thing as the following while loop

int cntr = 1;
while ( cntr <= 20 )
{
    do some statements here ....
    cntr = cntr + 1;
}

• except, the variable cntr is not available to the program after the for loop, but it is available to be referenced after the while loop

Michael Eckmann - Skidmore
College - CS 106 - Summer 2015
exercise

• Write an application that calculates the product of the odd integers from 1 to 15 and then displays the results in a message dialog.

• Let's write it using a while loop
• Then let's write it using a for loop
do while loops

• While loops and for loops all have the possibility of having the statements inside them execute zero or more times. *Zero* being the key word here.

• When would it be possible for those loops to execute zero times?

• The big difference between those kinds of loops and do-while loops are that do-while loops execute their statements at least once (that is, one or more times.)
do while loops

do
{
    statements to do
} while (condition);

– In this kind of loop, the condition is tested after the loop executes the first time. If the condition is true it does the statements again, and so on until the condition is false.
do while loops

do
{
    statements to do 1, 3 (if 2 was true)
}
while (condition);

In this kind of loop, the condition is tested after the loop executes the first time. If the condition is true it does the statements again, and so on until the condition is false.
do while loops

```c
int cntr = 1;
while ( cntr <= 20 )
{
    do some statements here ....
    cntr = cntr + 1;
}
```

// the above is basically equivalent to:

```c
int cntr = 1;
do
{
    do some statements here ... 
    cntr = cntr + 1;
} while (cntr <= 20);
```
break;

• We've seen how break; reacts in switch statements.

• break; acts similarly within the curly braces of a while loop, for loop or do-while loop.

• It terminates the loop and program control continues after the loop as if it ended normally.
int x = 0;
while (x <= 10)
{
    if (x == 5)
        break;
    break;
    System.out.println("x = " + x);
    x++;
}

// what's this code going to do?