ICS 111
Introduction to Computer Science I

Nikki Manuel
Leeward Community College

Loops : `while`, `do-while`, `for`

Week 7
Fall 2019

Loops

Loops

Computers are great at doing things over and over again!

- Perform calculations over and over again
  - Often the same calculation
  - Simple to complex calculations
- Print the same thing many times or with a slight variation

To accomplish this, we use **loops** to repeat lines of code.

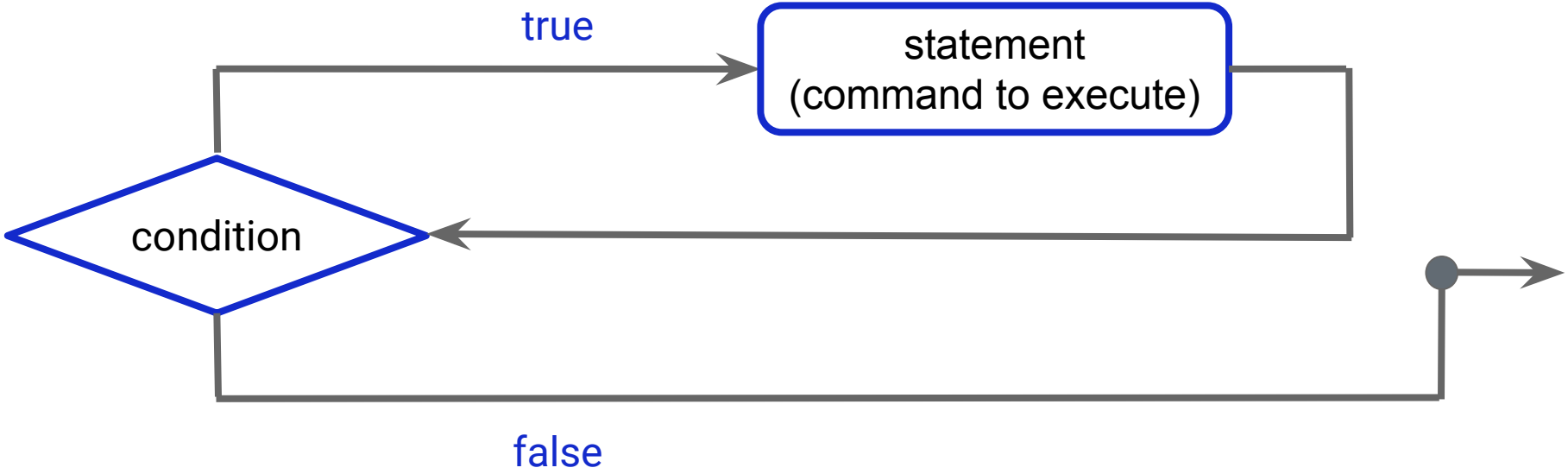
Types of Loops

- `while` loop
- `do-while` loop
- `for` loop

`while` Loops

`while` Loops

`while` loops repeat code as long as a condition is true



while Syntax

```
while (condition) {
    // Loop body
    Statement(s);
}
```

Loop body: the part of the loop that contains the statements to be repeated.

A one-time execution of a loop body is referred to as an **iteration** (or repetition) of the loop.

`while` Syntax

```
while (condition) {
    // Loop body
    Statement(s);
}
```

The **condition** is a Boolean expression that controls whether the loop body will be executed.

It is evaluated each time to determine if the loop body will be executed. If the evaluation is **`true`**, the loop body is executed; if **`false`**, the entire loop terminates.

`while` Loops - What You Need

- `while` keyword

- 3 things:
  1. The condition
     - Will evaluate to true or false
  2. Initialization expression
     - A starting point for the loop
  3. Modification expression
     - A statement that modifies the initialization expression so the condition will eventually become false

while Syntax

```
// initialization expression is before the loop

while (condition) {
   // Code in the loop body should be indented.
   // Modification expression is inside.
   // Statements to repeat as long as the
   // condition is true.
}
```

while Example

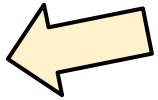
```
// initialization expression
int counter = 1;
int limit = 3;
String hello = "Hello!";


while (counter < limit) {
   System.out.println(hello);
}
```

while Example

```
// initialization expression
int counter = 1;
int limit = 3;
String hello = "Hello!";


while (counter < limit) {
    System.out.println(hello);
}
```



These variables are the **initialization expression**.

They act as the starting point for the loop.

while Example

```
// initialization expression
int counter = 1;
int limit = 3;
String hello = "Hello!";


while (counter < limit) {
    System.out.println(hello);
}
```

This is the **condition**.

It uses comparison operators to evaluate true/false.

while Example

```
// initialization expression
int counter = 1;
int limit = 3;
String hello = "Hello!";


while (counter < limit) {
    System.out.println(hello);
}
```

This is the code that will be repeated.

`while` Example – Something's wrong!

```
// initialization expression
int counter = 1;
int limit = 3;
String hello = "Hello!";


while (counter < limit) {
   System.out.println(hello);
}
```

There is no **modification expression**.

Therefore, the condition will always evaluate to the same thing -- <u>meaning the loop will either never execute or will infinitely repeat</u>.

In this code, the condition will always be true and the loop will repeat with no end.

`while` Example – Let's try again!

```
// initialization expression
int counter = 1;
int limit = 3;
String hello = "Hello!";

while (counter < limit) {
    System.out.println(hello);
    counter = counter + 1;
}
```

limit = 3	
counter	condition
1	true
2	true
3	false

How many times will this loop?
What is the value of counter at the end?

`while` Example #2

```
// initialization expression
int counter = 5;
int limit = 10;
char chk = '!';

while (counter < limit) {
   System.out.println(chk);
   counter++; // modification expression
}
```

`while` Example #2 – What is the output?

```
// initialization expression
int counter = 5;
int limit = 9;
char chk = '!';

while (counter < limit) {
   System.out.println(chk);
   counter++; // modification expression
}
```

`limit` = 9	
`counter`	condition
5	true
6	true
7	true
8	true
9	false

Test Yourself : `while` Loops #1

How many times is the loop body repeated?

What is the output of the loop?

```
int x = 1;
while (x < 10) {
    if (x % 2 == 0) {
        System.out.println(x);
    }
}
```

Test Yourself : `while` Loops #2

How many times is the loop body repeated?

What is the output of the loop?

```
int x = 1;
while (x < 10) {
    if (x % 2 == 0) {
        System.out.println(x++);
    }
}
```

Test Yourself : `while` Loops #3

How many times is the loop body repeated?

What is the output of the loop?

```
int x = 1;
while (x < 10) {
    if ((x++) % 2 == 0) {
        System.out.println(x);
    }
}
```

Test Yourself : `while` Loops #4

What is the output of the following loop? Explain why.

```
int x = 112263;
while (x > 0) {
   x++;
}
System.out.println("x is: " + x);
```

`do-while` Loops

`do-while` Loops

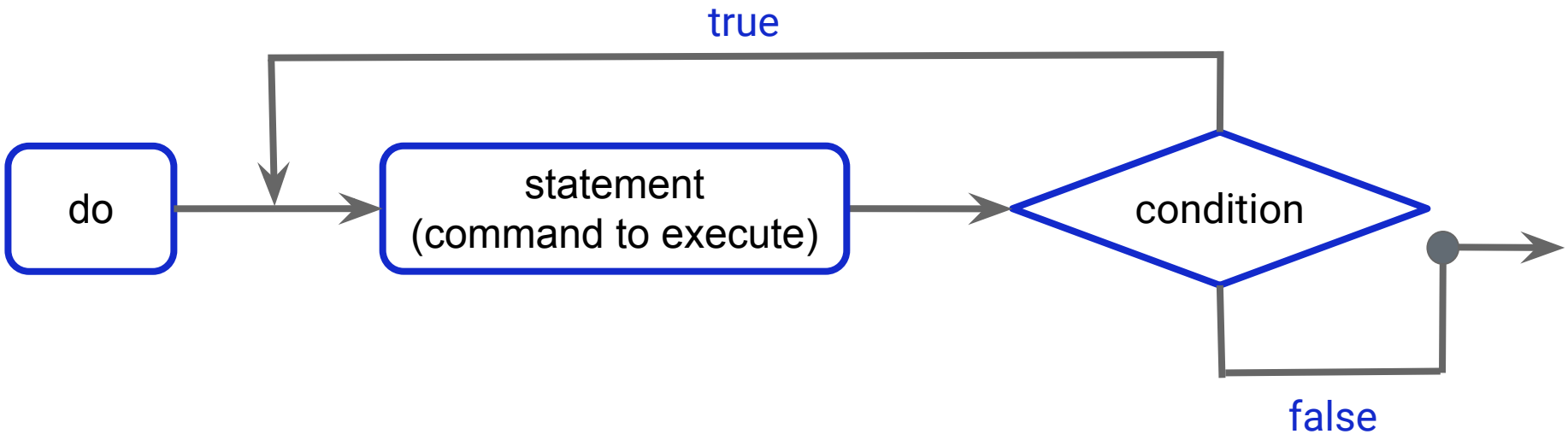
A **`do-while`** loop is the same as a **`while`** loop, except that it *first* executes the code in the loop body, *and then* checks the loop condition.

The loop body will be repeatedly executed for as long as the loop condition is **`true`**.

Note: a **`do-while`** loop will always execute at least once.

`do-while` Loops

`do-while` loops iterate once then repeat as long as the condition is true



do-while Syntax

```
do {
    // Loop body: code to repeat
} while (condition);
```

Remember that program flow is from top to bottom!

This is why the loop will repeat at least once.

Test Yourself : `do-while` Syntax

```
do {
    // Loop body: code to repeat
} while (condition);
```

Where do the following loop components go?

- Initialization expression
- Modification expression

`do-while` Loops vs. `while` Loops

The difference between a **`while`** loop and a **`do-while`** loop is the order in which the condition is evaluated and the loop body executed.

Use a **`do-while`** loop if you have statements inside the loop that must be executed at least once, as in the case of the do-while loop.

`for` Loops

`for` Loops

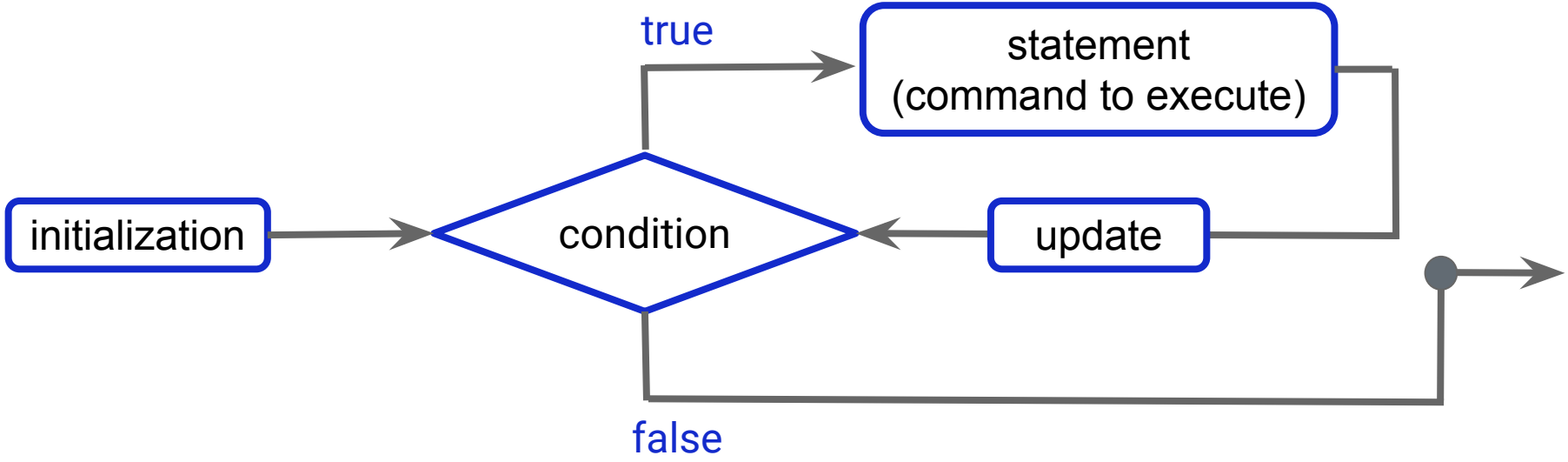
`for` loops are a more compact version of `while` loops

- The loop components are all on the same line

Any `for` loop can be converted into a `while` loop

- And vice versa

`for` Loops


true
statement
(command to execute)
initialization
condition
update
false

`for` Syntax

```
for (initialize; condition; modification) {
    // Code in the for loop should be indented.
    // Statements to repeat as long as the
    // condition is true.
}
```

- Contains the same 3 components as a `while` loop
- Except it uses the `for` keyword

`for` Syntax - Alternative Terms

You can also think of the loop control parts like this:

```
for (initial-action; loop-continuation-condition;
      action-after-each-iteration) {
   // Loop body
}
```

A `for` loop generally uses a variable to control how many times the loop body is executed and when the loop terminates. This variable is referred to as a **control variable**.

`for` Syntax - Alternative Terms

```
for (initial-action; loop-continuation-condition;
      action-after-each-iteration) {
    // Loop body
}
```

`initial-action` : often initializes the control variable

`loop-continuation-condition` : tests whether the control variable has reached a termination value

`action-after-each-iteration` : increments or decrements the control variable

for Example

```
int counter = 1;
int limit = 3;
String hello = "Hello!";


for (counter = 1; counter < limit; counter++) {
   System.out.println(hello);
}
```

`for` Example - Infinite Loop

```
int counter = 1;
int limit = 3;
String hello = "Hello!";


for (counter = 1; counter < limit; counter--) {
   System.out.println(hello);
}
```

`for` Example - Code Will Never Run

```
int counter = 1;
int limit = 3;
String hello = "Hello!";


for (counter = 100; counter < limit; counter--) {
   System.out.println(hello);
}
```

`for` Example – Declaring the Increment Variable

It is possible to declare and initialize an increment loop in the for:

```
for (int counter = 1; counter < 3; counter++) {
    System.out.println("Hello!");
}
```

Important: The variable `counter` will only be viewable within the loop. It cannot be used elsewhere.

Test Yourself : `for` Loops #1

Do the following loops result in the same value of `total`?

```
for (int i = 0; i < 10; ++i) {
    total += i;
}
```

```
for (int i = 0; i < 10; i++) {
    total += i;
}
```

Test Yourself : `for` Loops #2

What are the three parts of a `for` loop control?

If a variable is declared in a `for` loop, can it be used after the loop exits?

Write a `for` loop that prints the numbers from 1 to 100.

Test Yourself : `for` Loops #3

Convert the following for loop into a while loop and a do-while loop:

```
int num = 0;
for (int i = 0; i <= 1000; i++) {
    num = num + i;
}
```

Test Yourself : `for` Loops #4

In terms of n, how many iterations are in each of the following loops?

```
int x = 0;
while (x < n) {
    x++;
}
```

```
for (int x = 0; x <= n; x++) {
}
```

```
int x = 5;
while (x < n) {
    x++;
}
```

```
int x = 5;
while (x < n) {
    x = x +3;
}
```

`for` vs. `do-while` vs. `while` Loops

Use the loop statement that is most intuitive & comfortable for you.

In general, a **`for`** loop may be used if the number of repetitions is known in advance - for example, print “hello” a hundred times.

A **`while`** loop may be used if the number of repetitions is not fixed.

A **`do-while`** loop can be used to replace a **`while`** loop if the loop body has to be executed before the continuation condition is tested.

Loop Concepts

Evaluating the Condition

`for` and **`while`** loops

- Evaluate the condition *before* running code
- May or may not run code at all

`do-while` loops

- Evaluate the condition *after* running code
- Runs the code at least once

Counters & Accumulators

Counters

- A variable that is used keep track of counting
  - Tracks what point in the iteration you are in
- Typically `i++`

Accumulators

- A variable that is used to keep track of a sum
- Uses same syntax as a counter, but typically a number other than 1, usually a variable amount:

```
total = total + itemPrice;
```

Sentinel Value

A **sentinel value** is a special input value that signifies the end of the input and is used to terminate a loop

- Use a value that is outside the range of valid input
- The presence of a sentinel value guarantees the termination of a loop
- Also known as a flag value or signal value

Nested Loops

Just as with `if` statements, a loop can be nested inside another loop.

Each time the outer loop is repeated, the inner loops are reentered.

For example:

```
for (int i = 0; i < 10; i++)
   for (int j = 0; j < i; j++)
      System.out.println(i*j);
```

Nested Loops - Warning!

Be aware that a nested loop can take a long time to run!

For example:

```
for (int i = 0; i < 10000; i++)
    for (int j = 0; j < 10000; j++)
        for (int k = 0; k < 10000; k++)
            // code here performs some action
```

The action is performed *one trillion times*. If it takes 1 microsecond to perform the action, the loop would run for 277 hours. 😊

Variable Scope

Variable scope is where a variable can be used in your program after it has been declared.

It depends where a variable is declared to determine where it can be used

Variable Scope – Example

```
int numLoops = 5;

for (int i = 1; i < numLoops; i++) {

   System.out.println(i);

}

System.out.println(i);
```

Variable Scope – Example

```
int numLoops = 5;

for (int i = 1; i < numLoops; i++) {

    System.out.println(i);

}
```

Because the variable `i` is declared as the initialization expression for this `for` loop, its use is restricted to the for loop.

You can use the variable `i` anywhere in the for loop, inside the curly braces { and }

Variable Scope – Example

```java
int numLoops = 5;

for (int i = 1; i < numLoops; i++) {

    System.out.println(i);

}

System.out.println(i);
```

```
 ---- jGrasp exec: vsExample
vsExample.java:14: error: cannot find symbol
System.out.println(i);
                   ^
  symbol:   variable i
  cocation: class vsExample
1 error

 ---- jGrasp wedge2: exit code for process is 1
 ---- jGrasp: operation complete
```

Variable Scope – We could do this!

```
int numLoops = 5;
int i = 0;   // declare the variable outside of the loop

for (i = 1; i < numLoops; i++) {

   System.out.println(i);

}

System.out.println(i);    // after the loop, i will be 5
```

Variable Scope – Example #2

```
1  int i = 0;
2  try {
3      if (i < 5) {
4          int num = 3;
5          while (num > 0) {
6              double d = 0.8;
7          }
8      }
9  }
10 catch (SomeException e) {
11     double f = 4.5;
12 }
```

What is the scope of `i`?

> Lines 1 - 12

For this program, `i` is at the top level. The scope is anywhere after it's declared.

Variable Scope – Example #2

```
1  int i = 0;
2  try {
3      if (i < 5) {
4          int num = 3;
5          while (num > 0) {
6              double x = 0.8;
7          }
8      }
9  }
10 catch (SomeException e) {
11     double y = 4.5;
12 }
```

What is the scope of **`num`**?

> Lines 4 - 7

Anywhere in the `if` statement.

Variable Scope – Example #2

```
1  int i = 0;
2  try {
3      if (i < 5) {
4          int num = 3;
5          while (num > 0) {
6              double x = 0.8;
7          }
8      }
9  }
10 catch (SomeException e) {
11     double y = 4.5;
12 }
```

What is the scope of **x**?

> Line 6

Only in the **while** loop.

Variable Scope – Example #2

```
1  int i = 0;
2  try {
3      if (i < 5) {
4          int num = 3;
5          while (num > 0) {
6              double x = 0.8;
7          }
8      }
9  }
10 catch (SomeException e) {
11     double y = 4.5;
12 }
```

What is the scope of `y`?

> Line 11

Only in the `catch` block.

Avoid these problems!

Declare all your variables at the top

- They will be easy to locate and to document

Follow the coding standard

- Your code will be organized
- Scoping will become more obvious

When you indent your code properly it's clear where code is nested