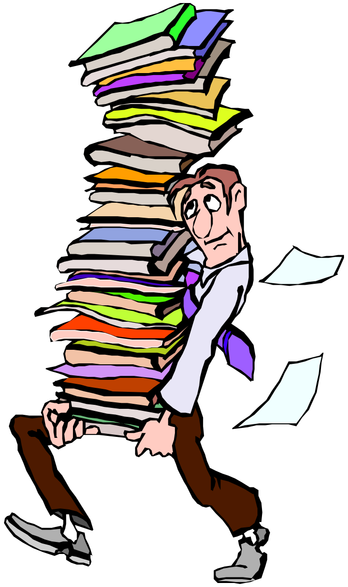
Review

Repetition Structures

Goal

- To learn about the three types of loops:
  - **`while`**
  - **`for`**
  - **`do-while`**
- To avoid common iteration errors
  - infinite loop error
  - off-by-one error
- To understand nested loops



Control Structures

- Decision making is a necessary step in non-trivial programs
- Control structures are programming blocks that allow the programmers to implement decision making and/or repetition operations
- Three major categories of control structures:
  1. Selection structures
  2. Repetition structures
  3. Jump structures

What is a Loop (i.e. Iteration)?

- Something that repeats
  - Each ‘iteration’ may do something different
  - Has a defined condition to stop repeating (Otherwise it will go on for ever! i.e. infinite loop)

Repetition Structure

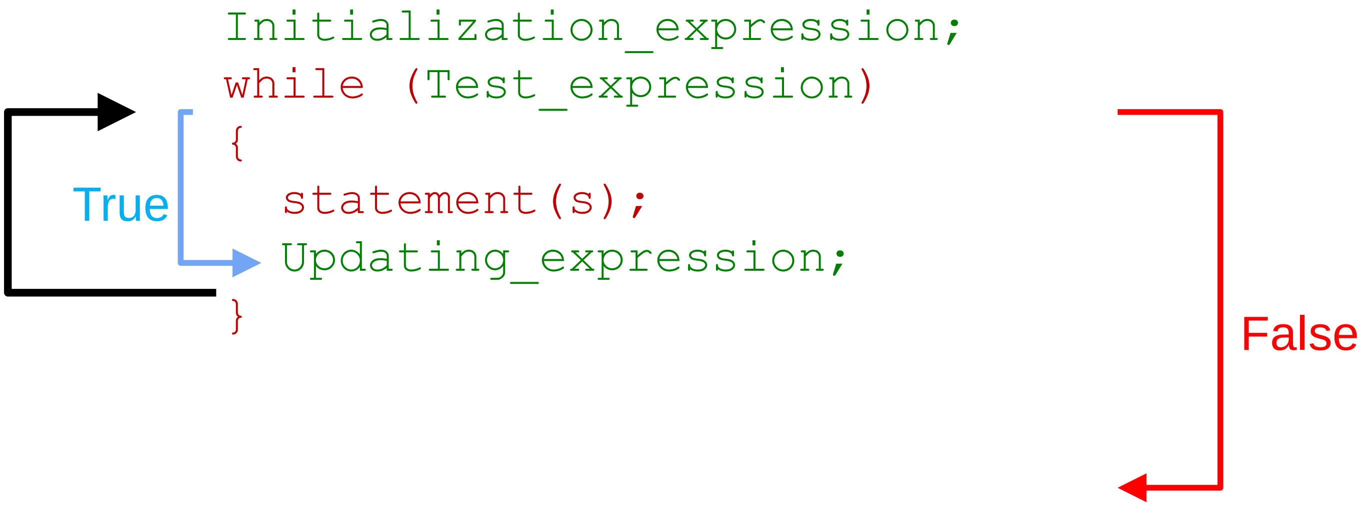
- A repetition structure (loop) is a control structures that allows programs to run a statement or series of statements repetitively, as long as certain test condition is true
- Loop has a ***condition*** and ***loop block***
- The loop block also called the ***body of the loop*** confined between the braces
- The braces are not required if the body is composed of only ONE statement

Different Type of Loops

- There are three different types of repetition (loop) in :
  - **while** Loop
  - **for** Loop
  - **do** … **while** Loop

How Does it Work?

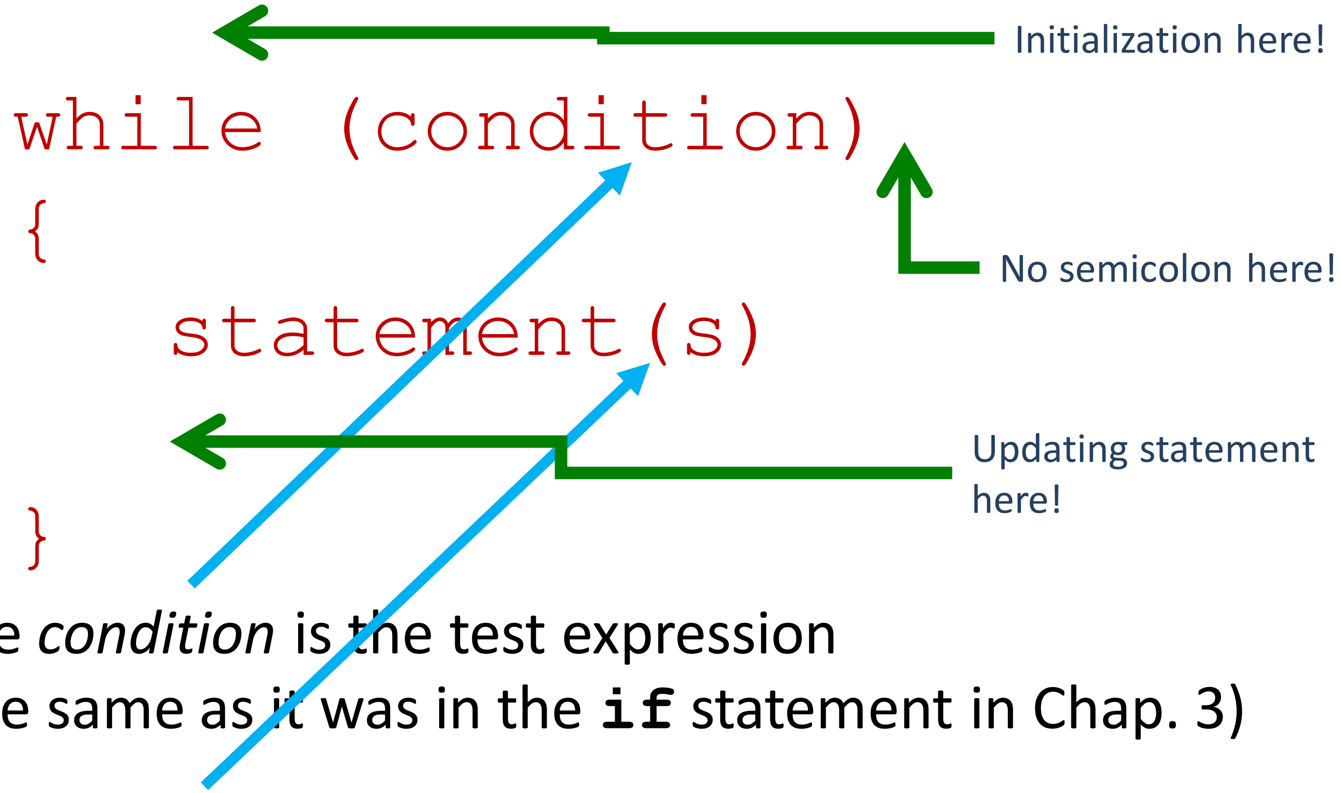
- To create a loop you need three expressions:
  - An *initialization expression*
  - A *test expression* to control and finally stop the loop
  - An *updating expression* that progresses towards the value of the test condition
- **Example:**



```
Initialization_expression;
while (Test_expression)
{
   statement(s);
   Updating_expression;
}
```

while Loop

What is while Loop?



```
while (condition)
{
    statement(s)

}
```

Initialization here!

No semicolon here!

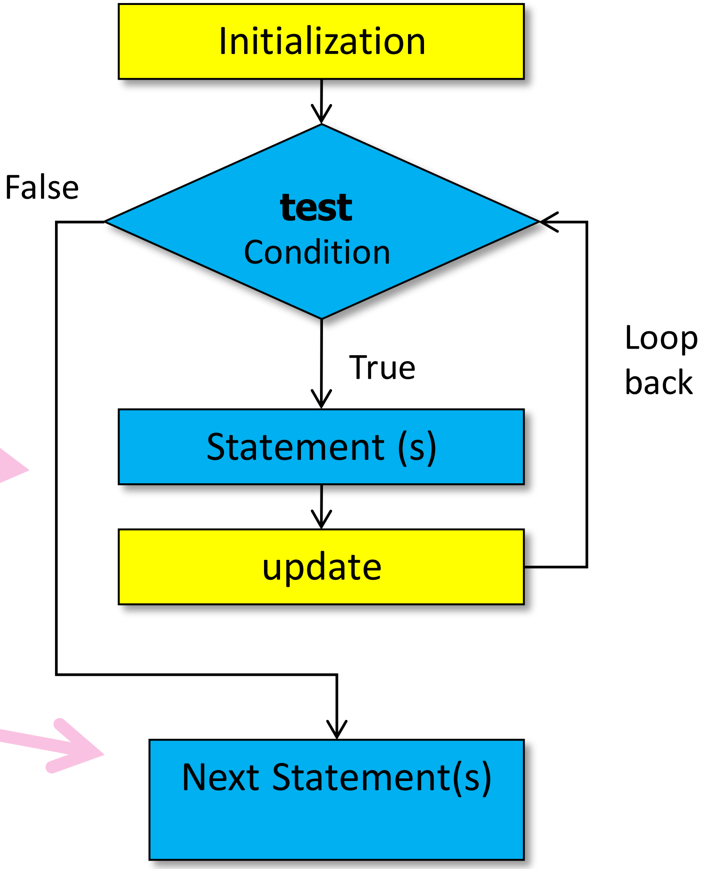
Updating statement here!

The *condition* is the test expression
(the same as it was in the the **`if`** statement in Chap. 3)

The statements are repeatedly executed
until the condition is **`false`**

while Loop: Flowchart

```
while (condition)
{
   statement(s)
}

 Next statement(s)
```



Example 1

This can be value of a variable set by the user externally

```
void main ()
{
    int i = 0;            // Initialization expression

    while ( i < 5 )       // Test expression
    {
        println(i);

        i++;              // Updating expression
    }

    println();
}
```

True

False

Output:

```
0
1
2
3
4
```

Example 2

Output

```
E
N
G
G

2
3
3
```

```
void main ()
{
  String name = "ENGG 233";
  int i = 0;                      // Initialization

  while ( i< name.length ( ) )     // Test expression
  {
    println(name.charAt(i));          // Print a character
    i++;                         // Updating expression
  }
  println("There are", i, "characters.");
}
```

Example 3

```
void main (){
     float x;
     x = random(1, 12);            // Initialization expression

      while ((x > 1) && (x  < 10)) {     // Test expression
         println(x, "is in range!");

         x = random (1, 12);            // Updating expression
       }
       println("number ", x, "is out of range");
}
```

This program keeps running… as long as x is between 1 and 10. It stops when the number is out of range.

What is wrong here?

```
void main()
{
   int i ;               // Declaration of i

   while ( i < 5 )       // Test expression
   {
         println(i);
         i++;            // Updating expression
   }
}
```

Variable i is not initialized! It holds an unknown value.

Not Really an Infinite Loop

- Due to what is called “wrap around”, the previous infinite loop ***will*** actually end.
- At some point the value stored in the declared variable **`i`** gets to the largest positive value. When it is incremented, the value stored “wraps around” to become a negative number.

That definitely stops the loop!

What is wrong here?

```
void main ()
{
   int i = 0;      // Initialization expression

   while ( i < 5 ) // Test expression
   {
         println(i);
         i--;     // Updating expression
   }
}
```

variable i does not progress towards 5!

What is wrong here?

i is set to 5

The i++; statement makes i get bigger and bigger

The condition will never become false

an infinite loop?

```
int i = 5;
while (i > 0)
{
    print(i, " ");
    i++;
}
```

The output never ends

```
5 6 7 8 9 10 11...
```

Yes! This is an infinite loop

What is Wrong Here?

That semicolon causes the while loop to have an “empty body” which is executed forever.

;

The `i` in (`i > 0`) never changes!

```
int i = 5;
while (i > 0);
{
    print(i," ");
    i--;
}
```

There is no output!

What is wrong here?

- In the investment program:

```
int year = 1;
while (year <= 20)
{
    balance = balance * (1 + RATE / 100);
}
```

- The variable `year` is not updated in the body

Forgetting to update the variable used in the condition is common

How to Stop/Exit a Loop?

- Using **break** statement:
- Used to abruptly jump out of loops
- break statement terminates a loop.

Not a good way to terminate a loop: indeterminism

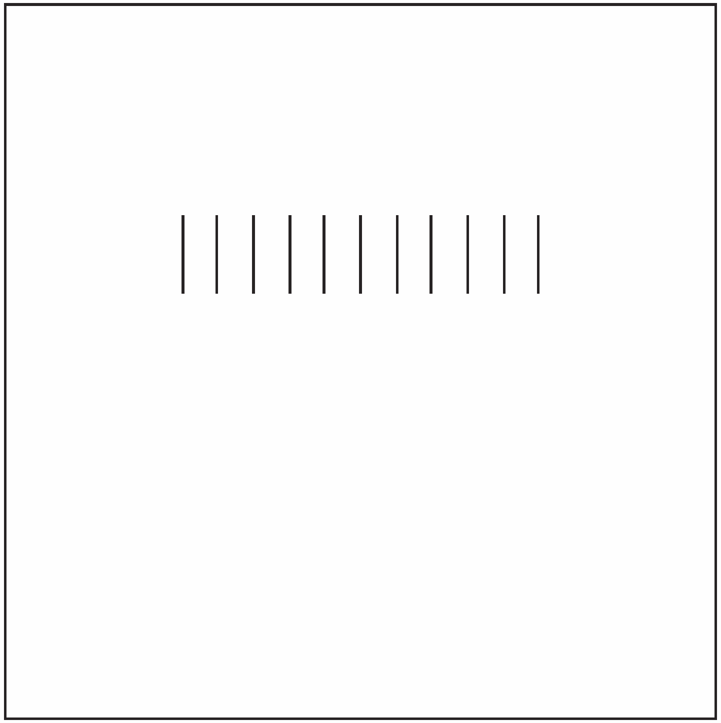
```
int noob = 5;          // Initialization expression
while (true)           // Test expression
  {
      print(noob," ");
      if (noob == 10)
          break;       // jump expression
      noob++;          // Updating expression
  }
```

Output:

```
5 6 7 8 9 10
```

Example: Draw a set of 11 vertical legs:

- All the same height:
  - 20 pixels high
- All the same distance apart
  - 10 pixels apart
- All the same color (black)
- Centered on the screen
  - 200 x 200 ‘grid’
  - First (left) at:
    - x = 50, y = 60;
  - Last (right) at:
    - x = 150, y = 60;

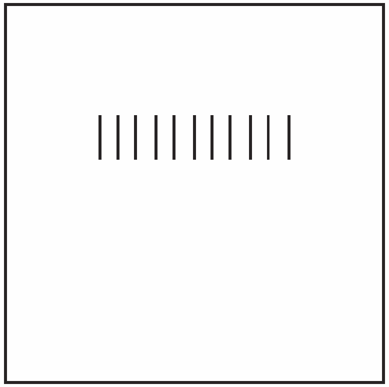


Modified from Slides by Don Smith

Why use Iteration?

- Without Iteration:

```
// No variables
stroke(0);
line( 50,60, 50,80);
line( 60,60, 60,80);
line( 70,60, 70,80);
line( 80,60, 80,80);
line( 90,60, 90,80);
line(100,60,100,80);
line(110,60,110,80);
line(120,60,120,80);
line(130,60,130,80);
line(140,60,140,80);
line(150,60,150,80);
```

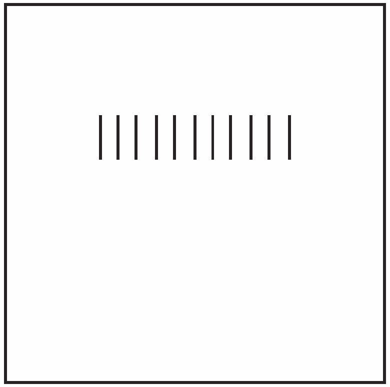


- Study what changes
  - x’s increase each time
- What is the pattern?
  - Add 10 each time
- When does it stop?
  - Last line is at x = 150

Planning Iteration (Step 1)

- Plan variables
  - Set Initial values

```
// No variables
stroke(0);
line( 50,60, 50,80);
line( 60,60, 60,80);
line( 70,60, 70,80);
line( 80,60, 80,80);
line( 90,60, 90,80);
line(100,60,100,80);
line(110,60,110,80);
line(120,60,120,80);
line(130,60,130,80);
line(140,60,140,80);
line(150,60,150,80);
```

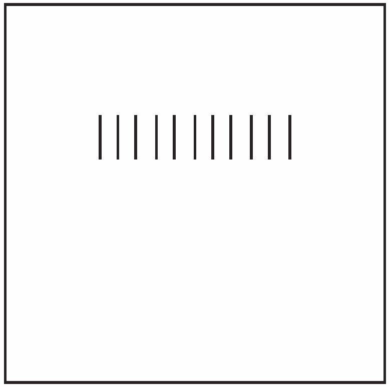


```
// With x variable
int x = 50;
int spacing = 10;

..
```

Planning Iteration (Step 2)

- Rewrite with variables:

```
// No variables
stroke(0);
line( 50,60, 50,80);
line( 60,60, 60,80);
line( 70,60, 70,80);
line( 80,60, 80,80);
line( 90,60, 90,80);
line(100,60,100,80);
line(110,60,110,80);
line(120,60,120,80);
line(130,60,130,80);
line(140,60,140,80);
line(150,60,150,80);
```



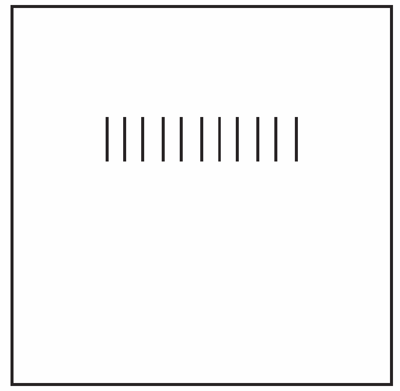
```
// With x variable
int x = 50;
int spacing = 10;

line(x,60,x,80);
x = x + spacing;
line(x,60,x,80);
x = x + spacing;
line(x,60,x,80);
x = x + spacing;
line(x,60,x,80);
x = x + spacing;
line(x,60,x,80);
x = x + spacing;
line(x,60,x,80);
..
```

Modified from Slides by Don Smith

Planning Iteration (3)

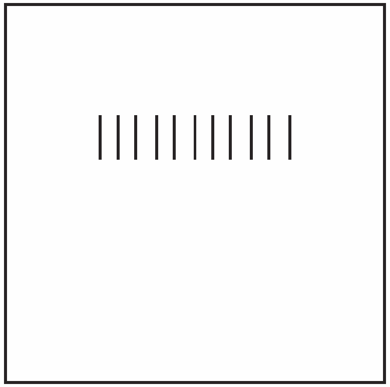
- Find the repetitive code

```
// With x variable
int x = 50;
int spacing = 10;

line(x,60,x,80);
x = x + spacing;
line(x,60,x,80);
x = x + spacing;
line(x,60,x,80);
x = x + spacing;
line(x,60,x,80);
x = x + spacing;
line(x,60,x,80);
x = x + spacing;
line(x,60,x,80);
x = x + spacing;
line(x,60,x,80);
..
```



Planning Iteration (4)

- Plan the ‘exit’ condition
  - Draw no more lines if…
    - x > 150?
    - x < 150?
    - x <= 150?
    - x >= 150?
- Declare and initialize an ‘end’ variable
  - Same type as x (**int**)
  - Obvious name
    - **endLegs**
  - Set to max value
    - **= 150**

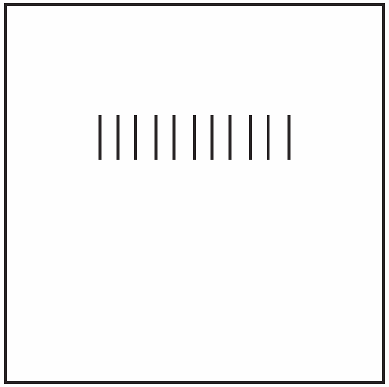


```
// With x variable
int x = 50;
int spacing = 10;
int endLegs = 150;

line(x,60,x,80);
x = x + spacing;
line(x,60,x,80);
x = x + spacing;
line(x,60,x,80);
x = x + spacing;
line(x,60,x,80);
x = x + spacing;
line(x,60,x,80);
x = x + spacing;
line(x,60,x,80);
x = x + spacing;
line(x,60,x,80);
..
```

Draw legs using Iteration

- Use the same ‘initialize’
- Put the repetitive code inside the ‘loop’
- Put your exit condition in the ‘test’

```
// Loop Version
int x = 50;
int spacing = 10;
int endLegs = 150;

while(x <= endLegs) {
   line(x,60,x,80);
   x = x + spacing;
}
```



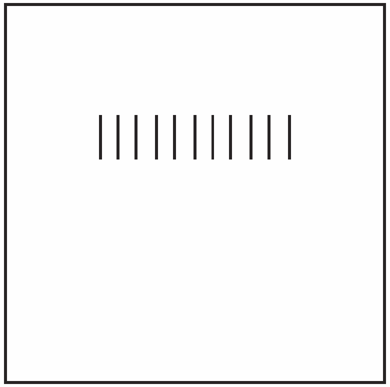
```
// With x variable
int x = 50;
int spacing = 10;
int endLegs = 150;

line(x,60,x,80);
x = x + spacing;
line(x,60,x,80);
x = x + spacing;
line(x,60,x,80);
x = x + spacing;
line(x,60,x,80);
x = x + spacing;
line(x,60,x,80);
x = x + spacing;
line(x,60,x,80);
x = x + spacing;
line(x,60,x,80);
..
```

Three Parts of the Loop

- Find the three parts:
  - Initialize
  - Test
  - Update
    - Inside loop body!

```
// Loop Version
int x = 50;
int spacing = 10;
int endLegs = 150;

while(x <= endLegs) {
   line(x,60,x,80);

   x = x + spacing;
}
```

Loop body



Modified from Slides by Don

for Loop

for Loop Synopsis

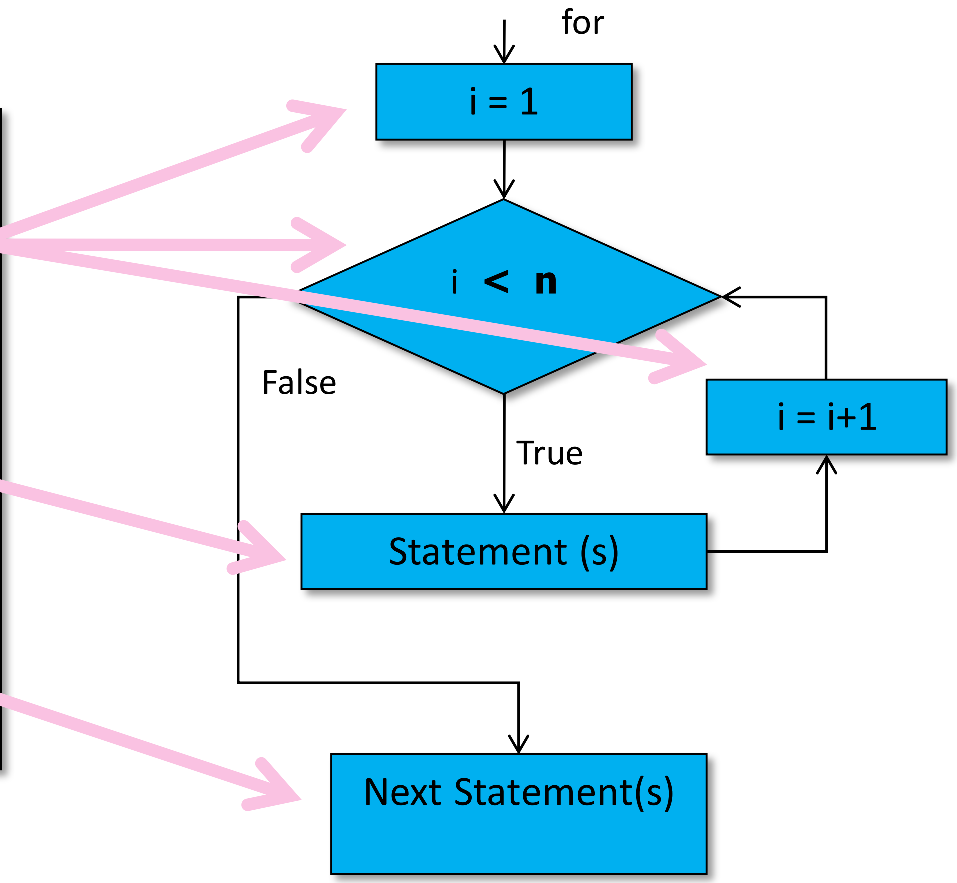
- The **for** loop is also a pre-test loop
- It tests the condition before its block of statement(s) is implemented
- General form of a for loop:

```
for(Initialization_expression; Test_expression;
  Updating_expression)
 {
      statement(s);
 }
```

- It stops if `Test_expression` turns to **false**

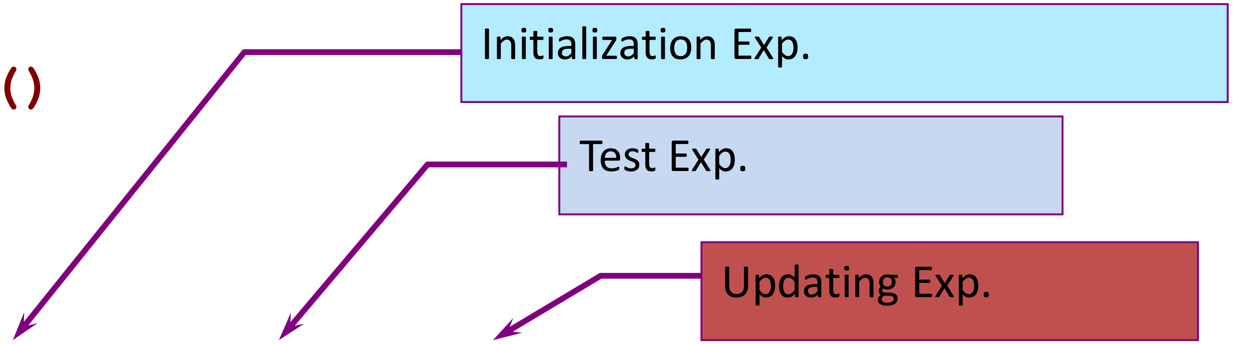
for Loop: Flowchart

```
for(int i=1; i<n; i++)
{
     statement(s)…
}
Next statement(s)…
```



Example

```
void main ()
{
    int i;

    for (i = 0;  i < 5;  i++)
    {
          println(i);

    }

}
```



Initialization Exp.

Test Exp.

Updating Exp.

Output:

```
0
1
2
3
4
```

Scope of Loop Variable

- You can declare a variable at the initialization section of a for loop:

Declaration &
Initialization

```
Void main()
{
    for (int i = 0;  i < 5;  i++)
    {
          println(i);
    }
}
```

- The “loop variable” when defined as part of the **for** statement cannot be used before or after the **for** statement – it only exists as part of the **for** statement and should not be used anywhere else in a program.
- A **for** statement can use variables that are not part of it, but they should not be used as the loop variable.

for Loop: Advantages (1)

- Programmers prefer using for-loop because all three expressions are in the same place, and reduces the chance of missing one of the three expressions.

```
for (initialization; test; update)
{
    statements
}
```

The initialization is code that happens once, before the check is made, in order to set up for counting how many times the statements will happen.

The update is code that causes the check to eventually become false. Usually it's incrementing or decrementing the loop variable.

Example

- Using char type to control a loop

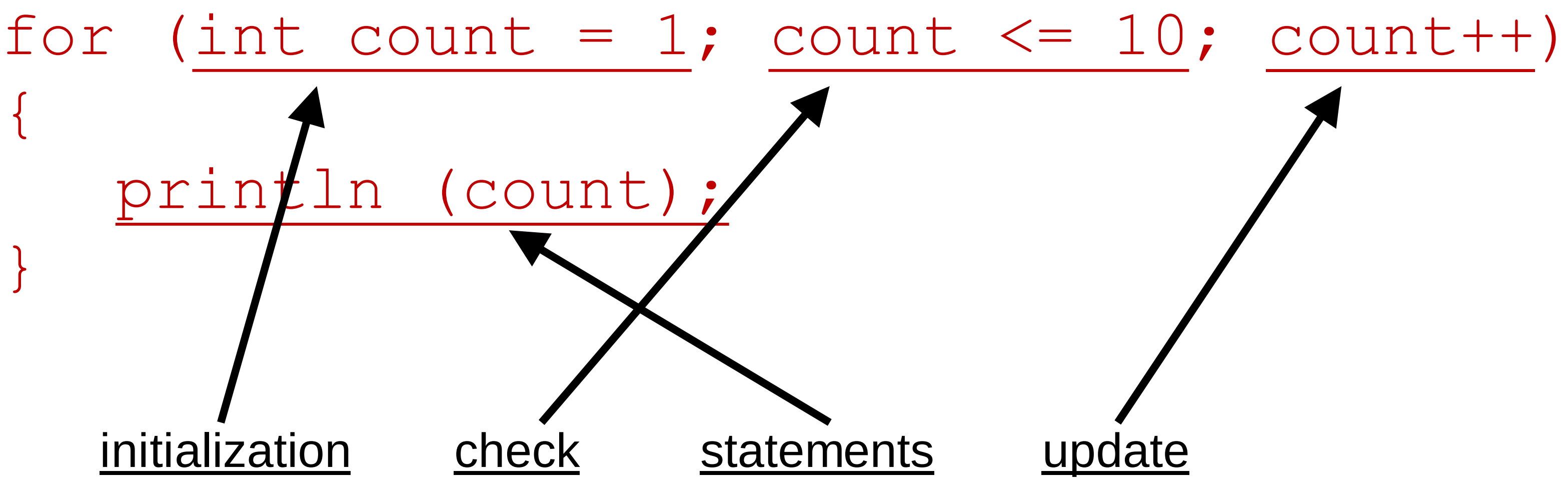
```
void main()
{
   for (char start = 'A'; start < 'G'; start++)
   {
         print(start);
   }
}
```

Output is:

```
ABCDEF
```

Repeat Certain Times

- Ideal for doing something a certain number of times or causing a variable to take on a sequence of values



```
for (int count = 1; count <= 10; count++)
{
    println (count);
}
```

initialization

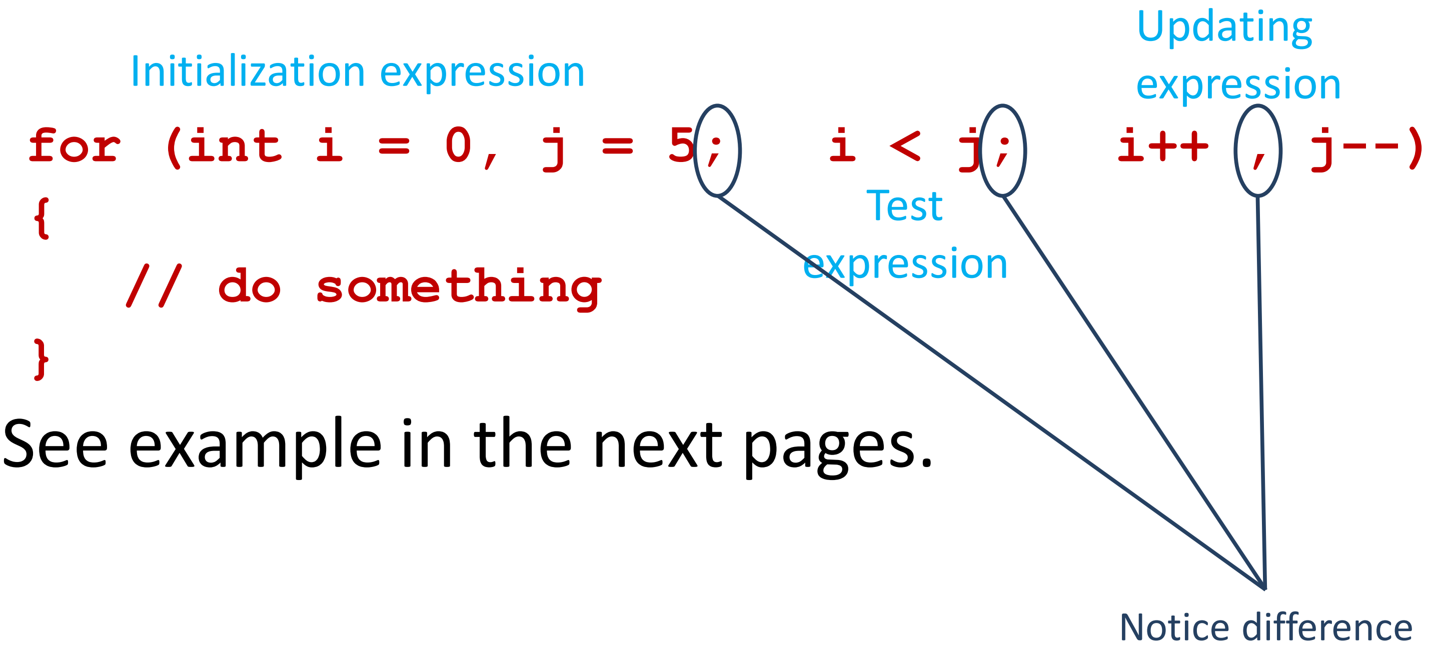
check

statements

update

for Loop: Advantages (2)

- for-loop is also powerful because initialization and updating expressions can do more than one initialization and more than one updating (**separated by a comma**):



```
for (int i = 0, j = 5;   i < j;   i++ , j--)
{
    // do something
}
```

- See example in the next pages.

Example

```
void main ()
{
  for (int i = 0, j = 5; i < j; i++, j--)
  {
    println(i, j);
  }
}
```

Calculate Execution Times

for loop

```
for (int i = 0;i <= 5; i++)
   println( i, " ");
```

output

```
0 1 2 3 4 5
```

Note that the output statement is executed six times, not five

Counting no. of Iterations?

- Symmetric Bound:
  - The range a ≤ n ≤ b is *symmetric*, both end points are included in the `for` loop:

```
for( int n=a; n<=b; n++ )…
```

- The loop with symmetric bounds, is executed `(b – a) + 1` times.

Counting no. of Iterations?

- Asymmetric Bound:

  The range a ≤ n < b is *symmetric*, both end points are included in the **`for`** loop:

```
for( int n=a; n<b; n++ )…
```

- The loop with asymmetric bounds, is executed **`(b – a)`** times or b times when a is 0
- Many coders use *asymmetric* form for problems involving doing something *N* times.

Taking Bigger Steps

for loop

```
for (int i = 0;i < 9;i += 2)
   println( i, " ");
```

Output

```
0 2 4 6 8
```

The “step” value can be other than increment or decrement.

Here the value 2 is added.

There are only 5 iterations, though.

Anything Wrong Here?

for loop

```
for (int i = 0;
       i != 9;
       i += 2)
   println( i, " ");
```

```
0 2 4 6 8 10 12…
```

The output never ends!

= = and != are best avoided
in the check of a `for` statement

Taking Even Bigger Steps

for loop to hand-trace

```
for (int i = 1; i <= 20; i *= 2)
   println( i, " ");
```

Output

```
1 2 4 8 16
```

The update can be any expression

`do … while` Loop

do … while Loop Synopsis

- The **do while** loop is also called a post-test loop:

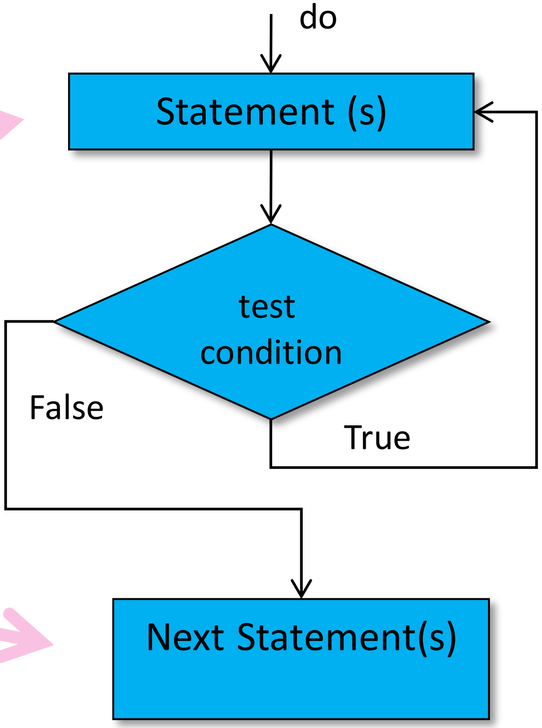
  It tests the condition after at least once its block of statement(s) is executed
- format of a `do … while` loop:

```
do
{
   statement(s)…
} while (test_condition);
```

- It stops if `test_condition` turns to **false**

do … while Loop: Flowchart

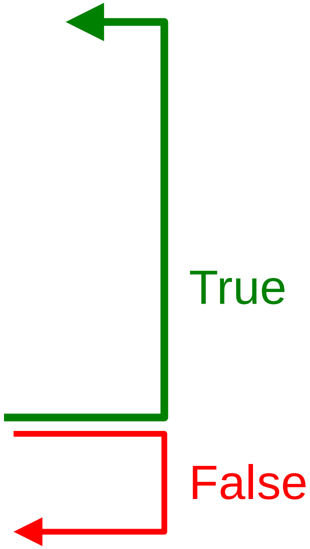
```
do {
    statement(s)…
   } while (test_condition);

Next statement(s)…
```



How Does it Work?

- **do while** also needs at least three expressions:
  - An initialization expression.
  - A test expression to control and finally stop the loop.
  - An updating expression that progress towards the value of the test condition

```
Initialization_expression;
do
 {
     statement(s);

     Updating_expression;

 } while (Test_expression);
```



Example

```
void main()
{
    int i = 0;       // Initialization exp.

    do
    {
        println(i);

        i++;                // Updating exp.
    } while ( i < 5 );  // Test exp.

    println();
}
```

True

False

Output:

```
0
1
2
3
4
```

Nested Loop

Nested Loops

- Nested loops are used mostly for data in tables as rows and columns.
- The processing across the columns is a loop, as you have seen before, “nested” inside a loop for going down the rows.
- Each row is processed similarly so design begins at that level. After writing a loop to process a generalized row, that loop, called the “inner loop,” is placed inside an “outer loop.”

Nested Loop

- A loop can be nested within another loop. For example:

```
Initialization_expression-outer;
while ( Test_expression-outer)
{
 statement(s);

Initialization_expression-inner;
while (Test_expression-inner)
   {
      statement(s);
      Updating_expression-inner;
   }

   Updating_expression-outer;
  }
```

Once the inner loop ended, goes back to the outer loop and the remaining part of the outer loop continues

Nested Loop: Example 1

```
void main()
{
    int row = 0;
    while(row < 5)
    {
        int column = 0;
        while (column <= row)
        {
            print("*");
            column++;
        }
        println();
        row++;
    }
}
```

Output is:

```
*
**
***
****
*****
```

Nested Loop: Example 2

```
void main ()
{
    char start = 'A';
    char end = 'G';
    while (start <= end)
    {
        char p = start;
        while (p <= end)
        {
            print(p);
            p++;
        }
        println();
        start++;
    }
}
```

Using char type to control a loop

Output is:

```
ABCDEFG
BCDEFG
CDEFG
DEFG
EFG
FG
G
```

Example 2

- Using char type to control a loop

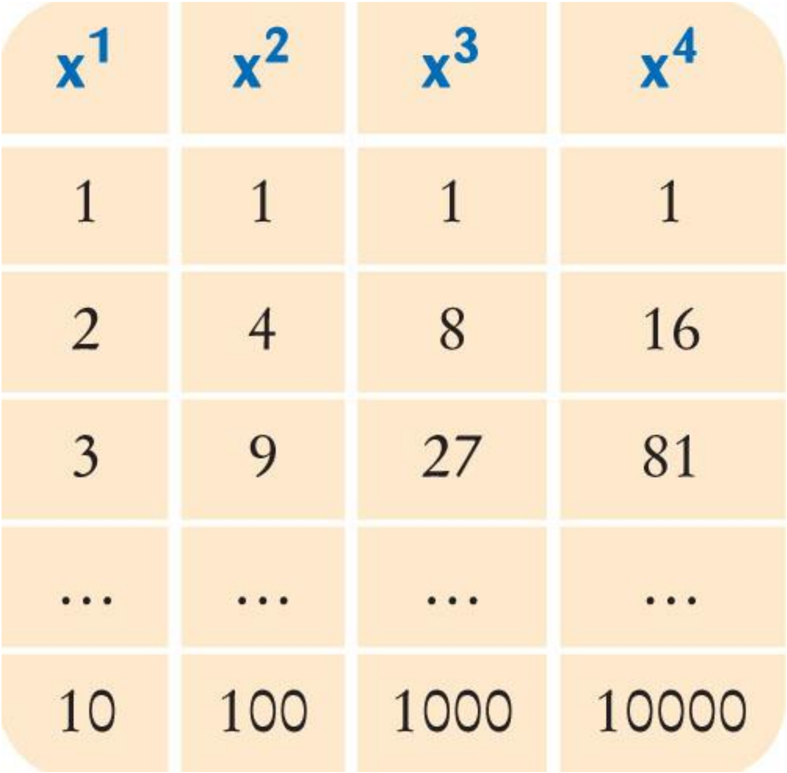
```
void main()
{
  for (char start = 'A', end = 'G'; start < end; start++)
  {
    for (char p = start; p < end; p++ )
    {
      print(p);
    }
    println();
  }
}
```

Output is:

```
ABCDEF
BCDEF
CDEF
DEF
EF
F
```

Nested Loop: Example 3

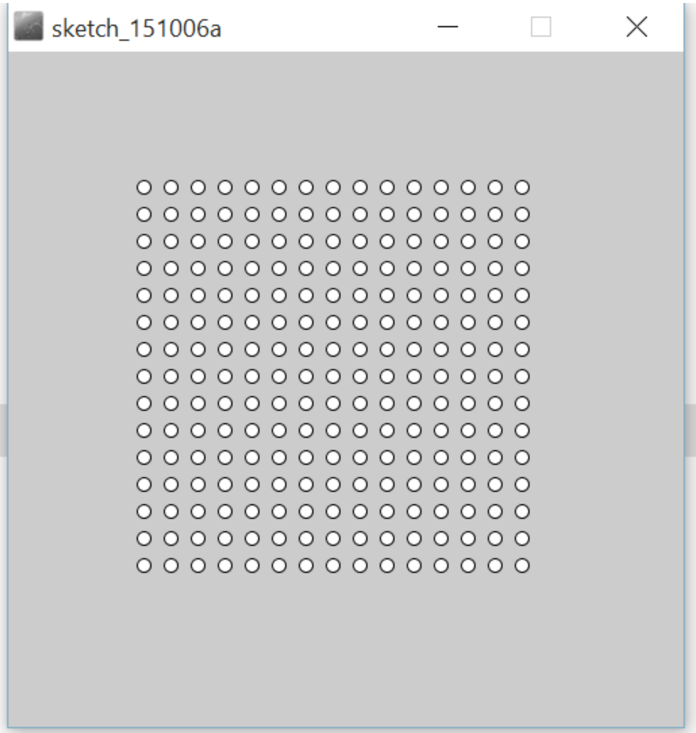
Write a program to produce a table of powers.
The output should be something like this:



x^1	x^2	x^3	x^4
1	1	1	1
2	4	8	16
3	9	27	81
…	…	…	…
10	100	1000	10000

Example 5

```
size (500, 500);


for (int i = 100; i < 400; i += 20)
{
     for (int j = 100; j < 400; j += 20)
     {
        ellipse (i, j, 10, 10);

     }

}
```



Jump Statements

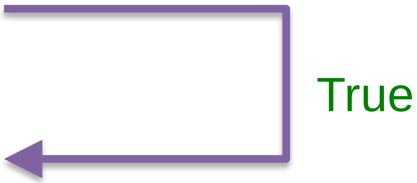
Jump Statements

- The purpose of a jump structure is to move the CPU control from one point of the program to another. There three predefined jump statements :
  - **`break;`**
  - **`continue;`**
  - **`goto`** // We do not use this one

`continue` Statement

- The ***continue*** *statement* is used to skip the rest of statements and proceed immediately to the end of the loop body, but not to exit.
- General form of continue statement

```
while (condition1)
{
    statements1;
    if (condition2)
    continue;
    statements2;

}
```



Example 1

```
for (int noob=0; noob <= 20; noob++)
{

    if ((noob % 4) == 0)
        continue;

    println(noob);
}
```

if the number is divisible by 4, skip this iteration

This program prints all of the numbers from 0 to 20 that aren’t divisible by 4.

Example 2

- Is there anything wrong here?

```
int i=0;
while (i < 10)
{
    if (i==5)
        continue;
    print(i, " ");
    i++;
}
```

This program is supposed to print every number between 0 and 9 except 5. It actually prints:

```
0 1 2 3 4
```

and then goes into an infinite loop. When i is 5, the if statement is true, and the loop returns back to the top. i is never incremented. Therefore, on the next pass, i is still 5, the if statement is still true, and the program continues to loop forever!

Summary

- Loops execute a block of code repeatedly while a condition remains true.
- The for loop is used when a value runs from a starting point to an ending point with a constant increment or decrement.
- The do loop is appropriate when the loop body must be executed at least once.
- Nested loops are commonly used for processing tabular structures.
- A sentinel value denotes the end of a data set, but it is not part of the data.
- You can use a Boolean variable to control a loop. Set the variable to true before entering the loop, then set it to false to leave the loop.