



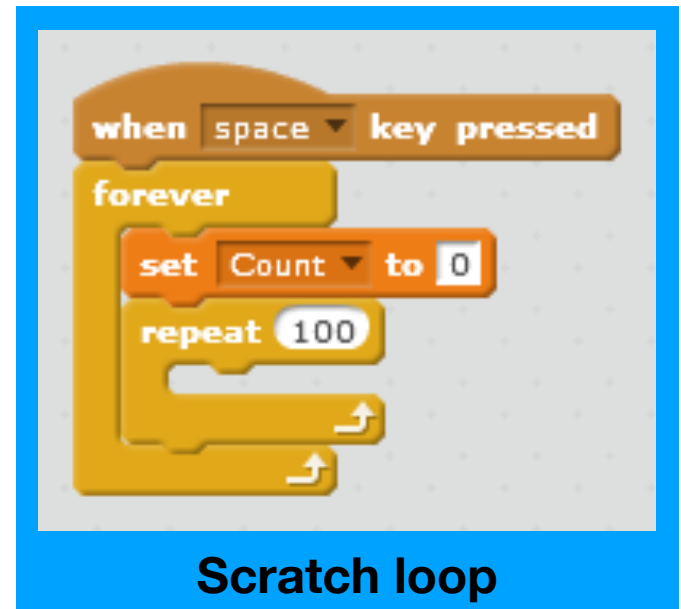
Iterative Statements

David Greenstein
Monta Vista High School

Iteration

1 of 16

- Repeating the same code fragment several times is called *iterating*.
- Iterating allows for repetitive tasks to be done efficiently, and computers are perfect for the task.
- Most programming languages offer control statements that iterate based on:
 - a condition to be satisfied (Java *while*)
 - a set number of repetitions (Java *for*)



```
def forExamples():
    print('Example 1')
    for counter in range(1,11):
        print(counter)
    print()
    print('Example 2')
    for counter in range(2,11,2):
        print(counter)
    print()
    print('Example 3')
    for counter in range(1,11):
        print(counter, end='')
```

Python loop

Java *while* Statement

2 of 16

```
while ( condition )
{
    statement1 ;
    statement2 ;
    ...
    statementN ;
}
```

condition is any logical expression, as in **if**

The **body** of the loop

Don't loop using *true*

```
while ( true )
{
    ...
    if ( condition ) break ;
    ...
}
```

Put *!condition* into the while

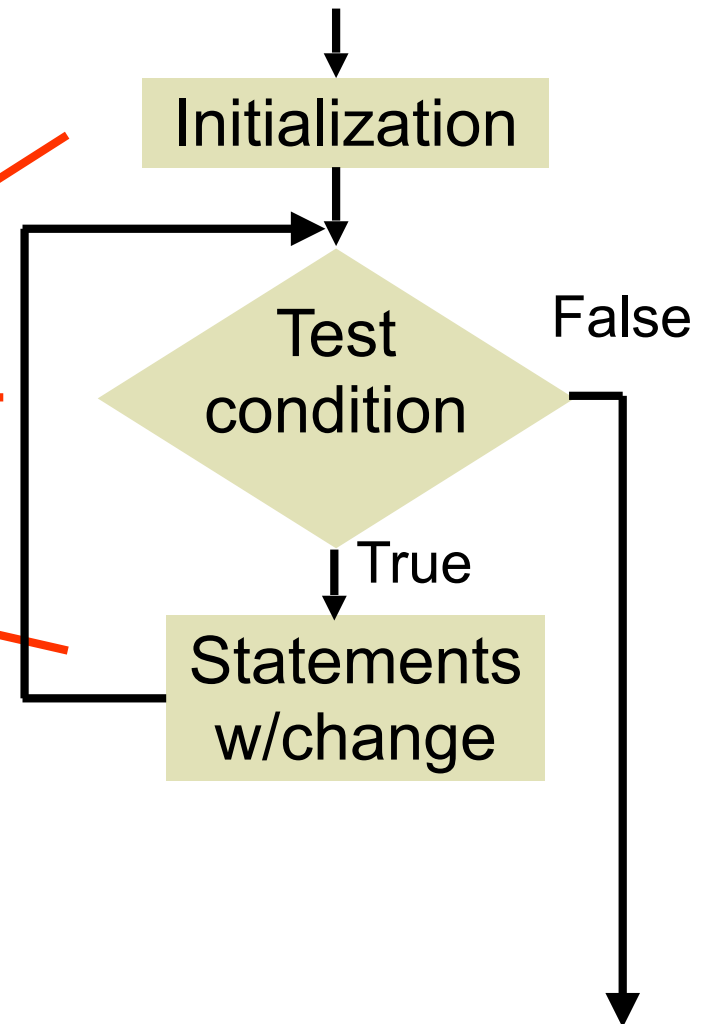
Java *while*

3 of 16

- Example:

```
// Returns the smallest n
// such that 2^n >= x
public static int intLog2 (int x)
{
    int n = 0, p = 1;
    while (p < x)
    {
        p *= 2;
        n++;
    }
    return n;
}
```

Flowchart



- Every *while* statement needs:
 - an **initialization** statement
 - a **condition** to test
 - a **change** within the block that affects the condition

Loop Invariant

4 of 16

- A loop invariant is an assertion that is true before the loop and at the end of each iteration.
- Invariants help us reason about the code.

```
int x = 5, n = 0, p = 1;
while (p < x)
{
    p *= 2;
    n++;
}
...
```

Loop invariant:
 $p = 2^n$

Java *for* Statement 5 of 16

- ***for*** is a shorthand that combines in one statement initialization, condition, and change:

```
for ( initialization; condition; change)
{
    statement1;
    statement2;
    . . .
    statementN;
}
```

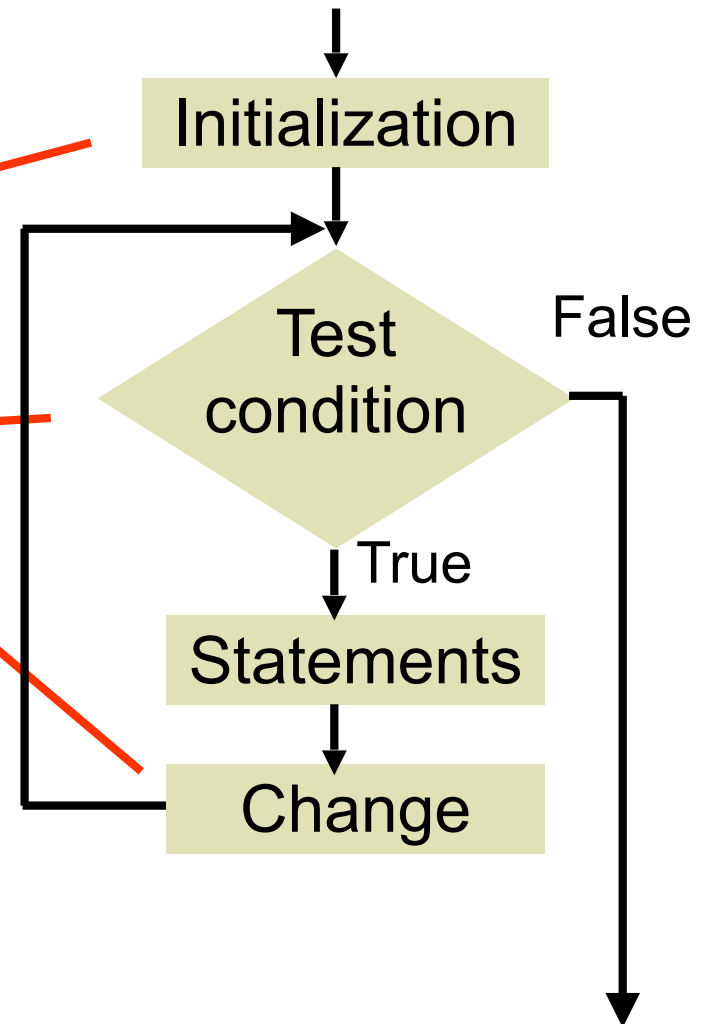
Java for

6 of 16

- Example:

```
// Returns the smallest n
// such that 2^n >= x
public static int intLog2 (int x)
{
    int n = 0, p;
    for (p = 1; p < x; p *= 2 )
    {
        n++;
    }
    return n;
}
```

Flowchart



- Every *for* statement explicitly states:

- an **initialization** statement
- a **condition** to test
- a **change** that affects the condition

Java *for* Statement 7 of 16

- Regular *for loop*

```
char[] chars = {'a', 'b', 'c', 'd'};
String str = "";
for ( int a = 0; a < chars.length; a++)
    str += chars[a];
```

- Irregular *for loop* - compiles and executes, **but**

```
char[] chars = {'a', 'b', 'c', 'd'};
String str = "";
int a = 0;
for ( ; a < chars.length ; ) {
    str += chars[a];
    a++;
}
```

Use **while**
loop instead

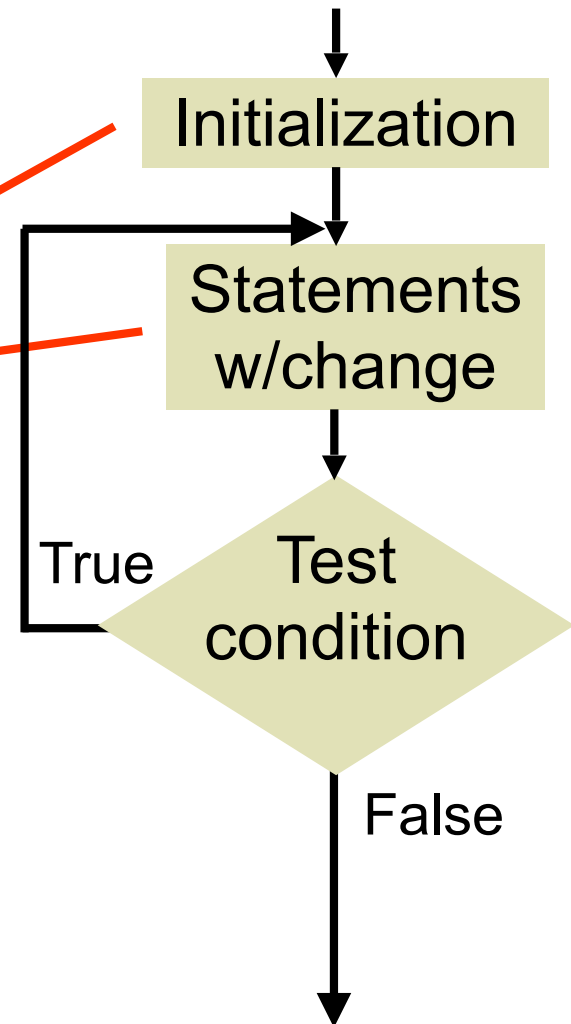
Java *do-while*

8 of 16

- Example:

```
// Returns 2 to the n power
// Precondition: n > 0
public static int intPow2 (int n)
{
    int cnt = 1, p = 1;
    do {
        p *= 2;
        cnt++;
    } while (cnt <= n);
    return p;
}
```

Flowchart



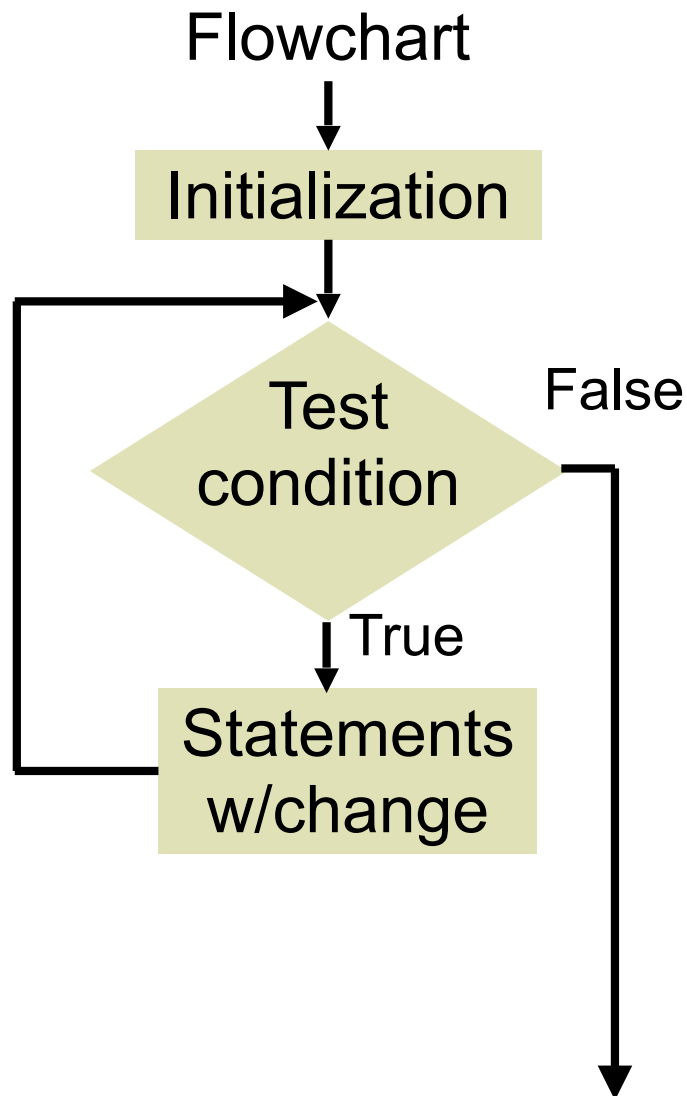
- Every *do-while* statement needs:

- an **initialization** statement
- a **change** within the block that affects the condition
- a **condition** to test

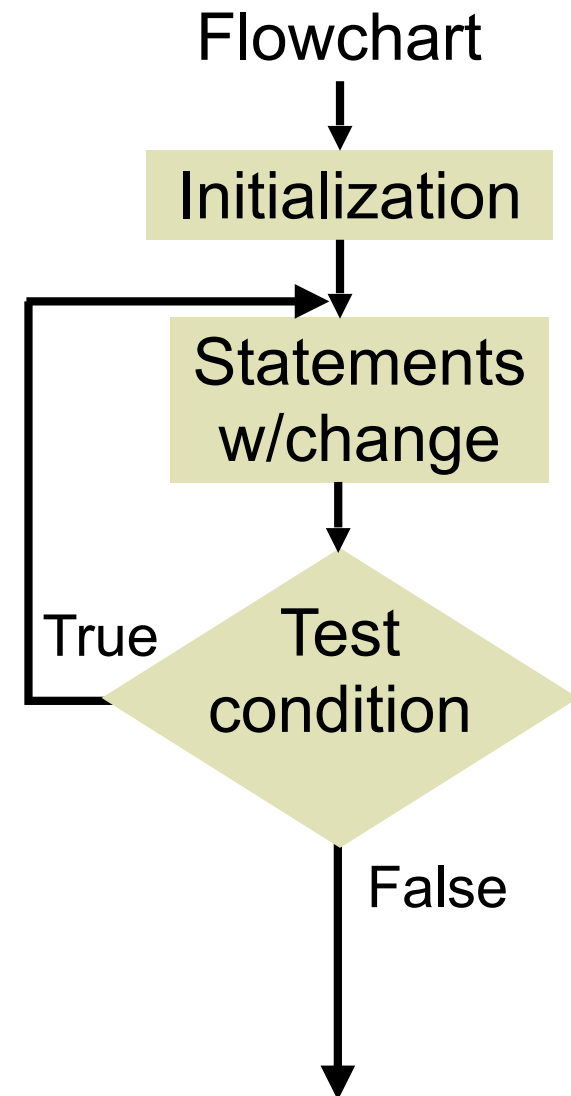
Java *while* vs *do-while*

9 of 16

while loop flow



do-while loop flow



Java *while* vs *do-while*

10 of 16

- *do-while*'s can be avoided by using a *while* statement

do-while

```
public static int intPow2 (int n)
{
    int cnt = 1, p = 1;
    do {
        p *= 2;
        cnt++;
    } while (cnt <= n);
    return p;
}
```

while

```
public static int intPow2 (int n)
{
    int cnt = 0, p = 1;
    while (cnt <= n) {
        p *= 2;
        cnt++;
    }
    return p;
}
```

break in Loops

11 of 16

- **break** in a loop instructs the program to immediately quit the current iteration and go to the first statement following the loop. **Use sparingly!**
- **return** in a loop instructs the program to immediately quit the current method and return to the calling method.
- A **break** or **return** must be inside an **if** or an **else**, otherwise the code after it in the body of the loop will be unreachable.

break in Loops (cont)

12 of 16

- Examples:

Better without break

```
int d = n - 1;

while (d > 1)
{
    if (n % d == 0)
        break;
    d--;
}

if ( d > 1 ) // if found a divisor
    ...
```

```
int d = n - 1;

while (d > 1 && n % d != 0)
    d--;

if ( d > 1 ) // if found a divisor
    ...
```

return in Loops

13 of 16

- Recommended to use *return* for “break-like” result

```
public int search(String[] list, String word)
{
    for (int k = 0; k < list.length; k++)
    {
        if (list[k].equals(word))
            return k;
    }

    return -1;
}
```

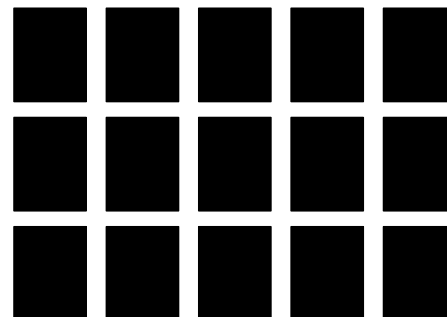
Nested Loops

14 of 16

- A loop within a loop is called **nested**.

```
// Draw a 5 by 3 grid:  
  
for (int x = 0; x < 50; x += 10)  
{  
    for (int y = 0; y < 30; y += 10)  
    {  
        g.fillRect(x, y, 8, 8);  
    }  
}
```

Result:

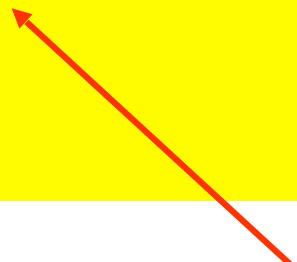


break Inside Nested Loops

15 of 16

- **Danger!!!**

```
for (int r = 0; r < m.length; r++)
{
    for (int c = 0; c < m[0].length; c++)
    {
        if (m [ r ] [ c ] == 'X' )
            break;
    }
}
...
```



Breaks out of the inner loop but continues with the outer loop

Triangular Nested Loops

16 of 16

- The inner loop variable starts at the outer loops next value.

```
for (int i = 0; i < a.length; i++)
{
    for (int j = i + 1; j < a.length; j++)
    {
        if (a [ i ].equals(a [ j ])
            System.out.println ("Duplicate " + a [ j ] );
        }
    }
}
```

