10/12/2022

4.1

The while loop

Iteration

Iteration, in the context of computer programming, is a process wherein a set of instructions are repeated a specified number of times or until a condition is met.

Each time the set of instructions is executed is called an **iteration**.

Another term for iteration is **loop**… the program "loops back" to an earlier step and repeats.

while syntax

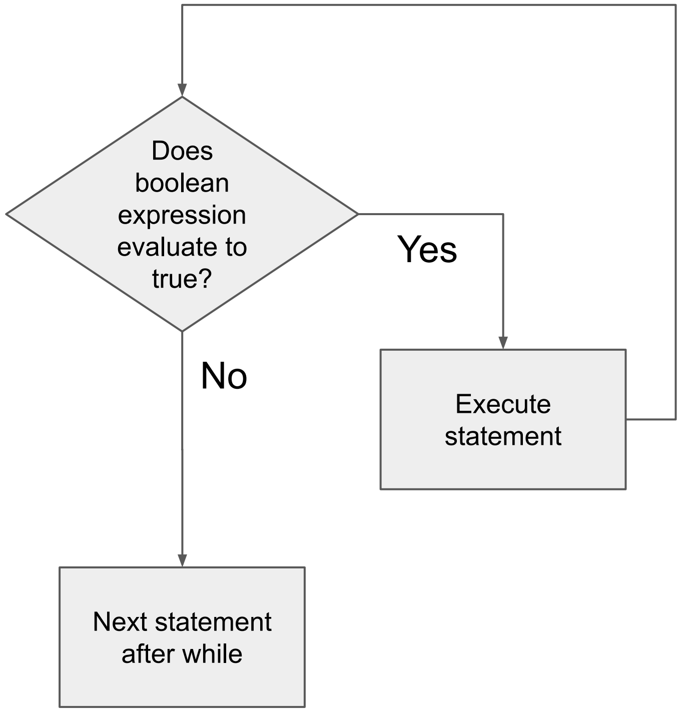
`while` (*boolean expression*)

    statement

Example:

```
int i = 1;
while (i <= 100) {
     System.out.println(i);
     i++;
}
```

Flowchart of while


Does boolean expression evaluate to true?
Yes
No
Execute statement
Next statement after while

++ and -- operators make for compact loops

Both of the loops below do the same thing. Programmers differ on style.

```
int i = 1;
while (i <= 100) {
    System.out.println(i);
    i++;
}
```

```
int i = 1;
while (i <= 100) {
    System.out.println(i++);
}
```

`i++` means post-increment, so it evaluates to the current value of `i`, then increments it.

What would be printed out if `System.out.println(++i)` was used?

while syntax and curly braces

`while (`*boolean expression*`)`

    statement

Just like if, *statement* can be any legal statement… a single statement, or a block. This works:

```
int i = 1;
while (i <= 100)
     System.out.println(i++);
```

Most Java coding conventions require curly braces even for a single statement, just like if.

```
int i = 1;
while (i <= 100) {
     System.out.println(i++);
}
```

++ and -- operators can be used in condition

You'll see this kind of cleverness sometimes.
Usually, people just use a for loop for this kind of thing. (Covered a few slides from now.)

```
void printSomethingManyTimes(String text, int count) {
  while (--count >= 0) {
    System.out.println(text);
  }
}
```

= (assignment) can be used in the condition

Your loop condition may depend on some code that repeats every iteration:

```
String command = getNextCommand();
while (command != null) {
    executeCommand(command);
    command = getNextCommand();
  }
}
```

The assignment operator can be used in the condition to avoid the repetition:

```
String command;
while ((command = getNextCommand()) != null) {
    executeCommand(command);
  }
}
```

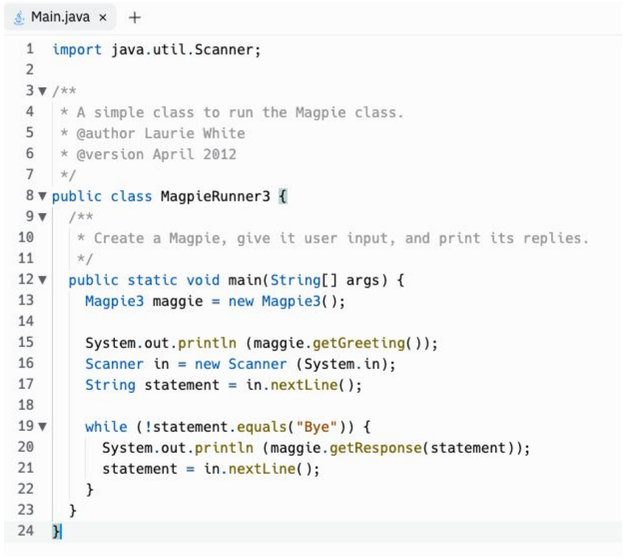
Programmers may differ on the style here. (The first style was used in Magpie.)

Sentinel values

A sentinel value is a special value that tells the loop to terminate.
The code from the previous slide is an example of this:
getNextCommand returns null when there are no more commands to execute.

```
String command;
while ((command = getNextCommand()) != null) {
    executeCommand(command);
  }
}
```

Input-controlled loops



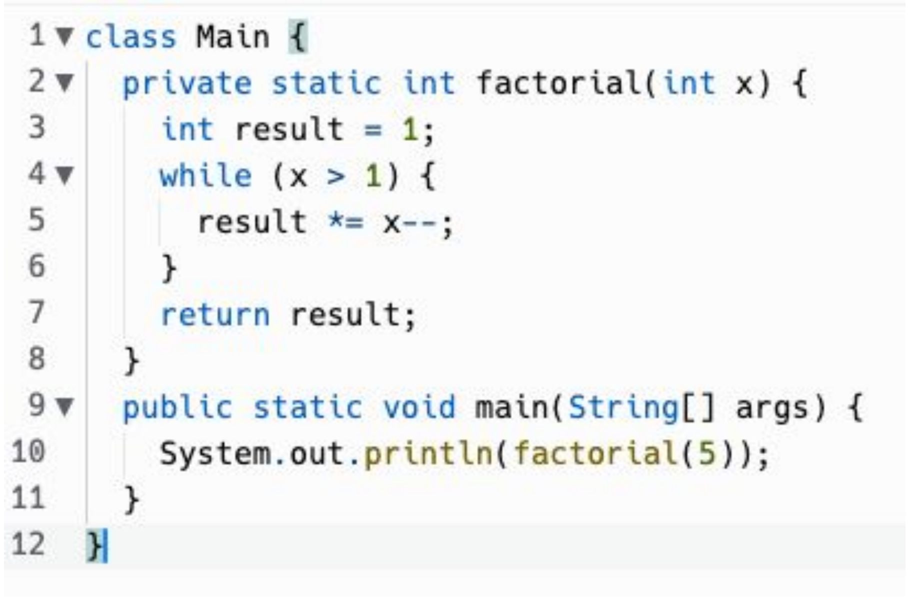
```java
import java.util.Scanner;

/**
 * A simple class to run the Magpie class.
 * @author Laurie White
 * @version April 2012
 */
public class MagpieRunner3 {
  /**
   * Create a Magpie, give it user input, and print its replies.
   */
  public static void main(String[] args) {
    Magpie3 maggie = new Magpie3();

    System.out.println (maggie.getGreeting());
    Scanner in = new Scanner (System.in);
    String statement = in.nextLine();

    while (!statement.equals("Bye")) {
      System.out.println (maggie.getResponse(statement));
      statement = in.nextLine();
    }
  }
}
```

The while loop is often used for **input-controlled loops**, where **input** is being taken from the user, or from a file on disk, or the network.

An example is the MagpieRunner, where the while loop continues until the user enters "Bye", the sentinel value which tells Magpie that no more input is coming and terminates the loop.

Tracing loops

Let's trace through this loop together…



```
class Main {
  private static int factorial(int x) {
    int result = 1;
    while (x > 1) {
      result *= x--;
    }
    return result;
  }
  public static void main(String[] args) {
    System.out.println(factorial(5));
  }
}
```

do…while syntax

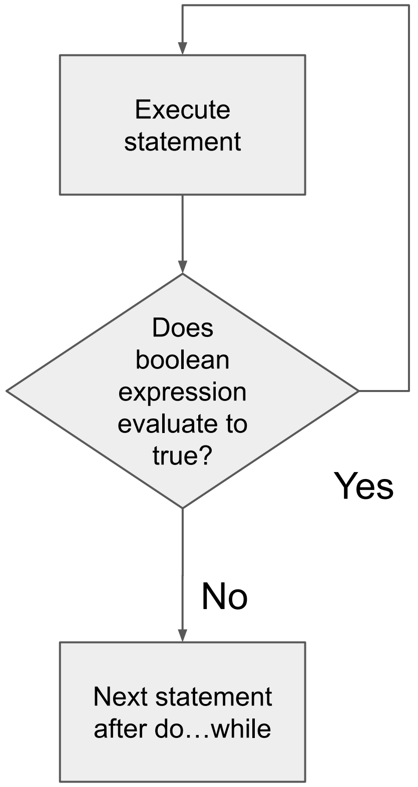
```
do
      statement
while  (boolean expression)
```

Sometimes, you want to check some condition AFTER the body of the loop has run, not before.

Example:

```
String name;
do {
  System.out.println("Enter your name.");
  name = scanner.nextLine();
} while (name.length() == 0);
```

Flowchart of while


Execute statement
Does boolean expression evaluate to true?
Yes
No
Next statement after do…while

Infinite loops

```
void serveRequestsForever() {
  while (true) {
    handleNextRequest();
  }
}
```

This may seem strange, but it has its place.
Sometimes the loop isn't really infinite, but the termination condition of the loop is complicated.
There are ways to break out of a loop, even an infinite one (break, return).

4.2

The for loop

Counter-controlled loops

We looked at input controlled loops, which are often done using while.

For statements are often used to do **counter-controlled loops**, where the loop is repeated a specific number of times, and a numeric counter is used to track which iteration the loop is on.

However, really, any of the loop statements in Java can be used to write any possible program. Which loop to use is a matter of what you think best expresses the intent of the program.

for syntax

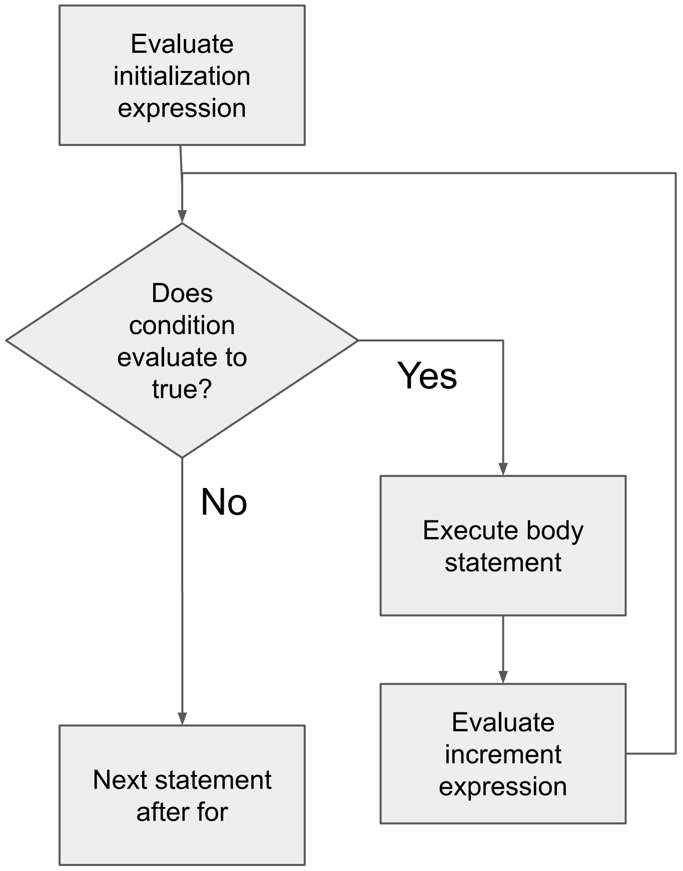
`for` (*initialization; condition; increment*)

statement

Example:

```
int i;
for (i = 1; i <= 100; i++) {
    System.out.println(i);
}
```

Flowchart of for


Evaluate initialization expression
Does condition evaluate to true?
Yes
No
Execute body statement
Evaluate increment expression
Next statement after for

for syntax

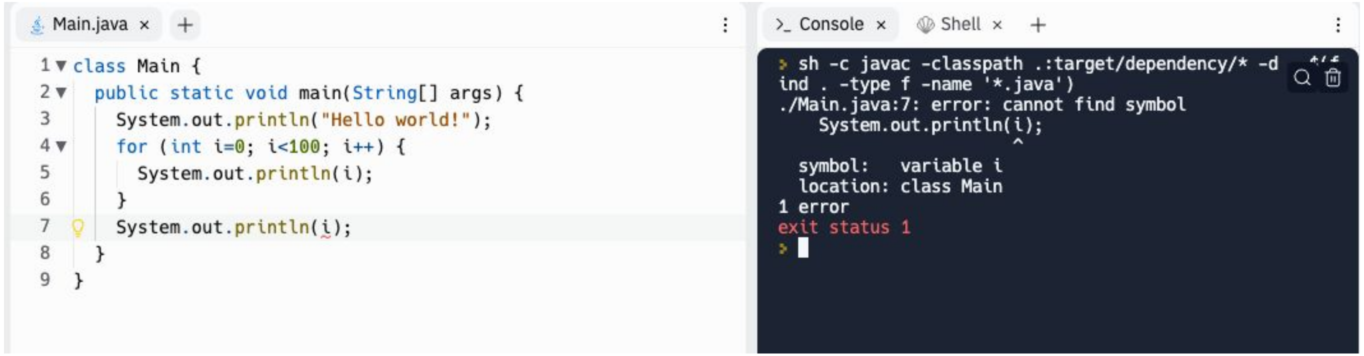
`for` (*initialization; condition; increment*)

    statement

initialization may also declare variables

```
for (int i = 1; i <= 100; i++) {
    System.out.println(i);
}
```

The scope of any variable declarations is purely the for loop itself.



```
1 class Main {
2   public static void main(String[] args) {
3     System.out.println("Hello world!");
4     for (int i=0; i<100; i++) {
5       System.out.println(i);
6     }
7     System.out.println(i);
8   }
9 }
```

```
> sh -c javac -classpath .:target/dependency/* -d
ind . -type f -name '*.java')
./Main.java:7: error: cannot find symbol
    System.out.println(i);
                       ^
  symbol:   variable i
  location: class Main
1 error
exit status 1
>
```

for syntax

`for` (*initialization; condition; increment*)

    statement

The last example started a counter at 1, which can be done. But like most things in Java, and computer science in general, counters usually start at 0 by convention. It makes interfacing with other code and data structures more straightforward.

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

Generally not…

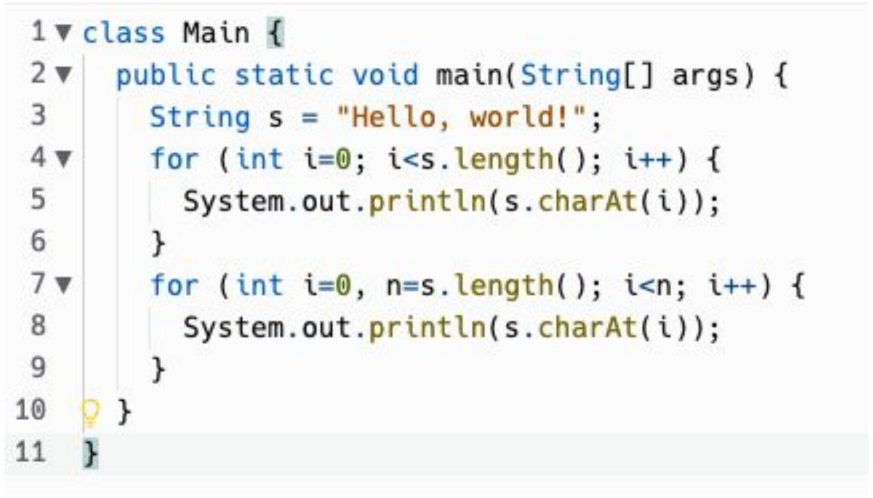
```
for (int i = 1; i <= n; i++) {
    doSomething(i);
}
```

for syntax

`for` (*initialization; condition; increment*)

    statement

initialization could even declare multiple variables. I often declare "int i=0, n=s.length()". Why?

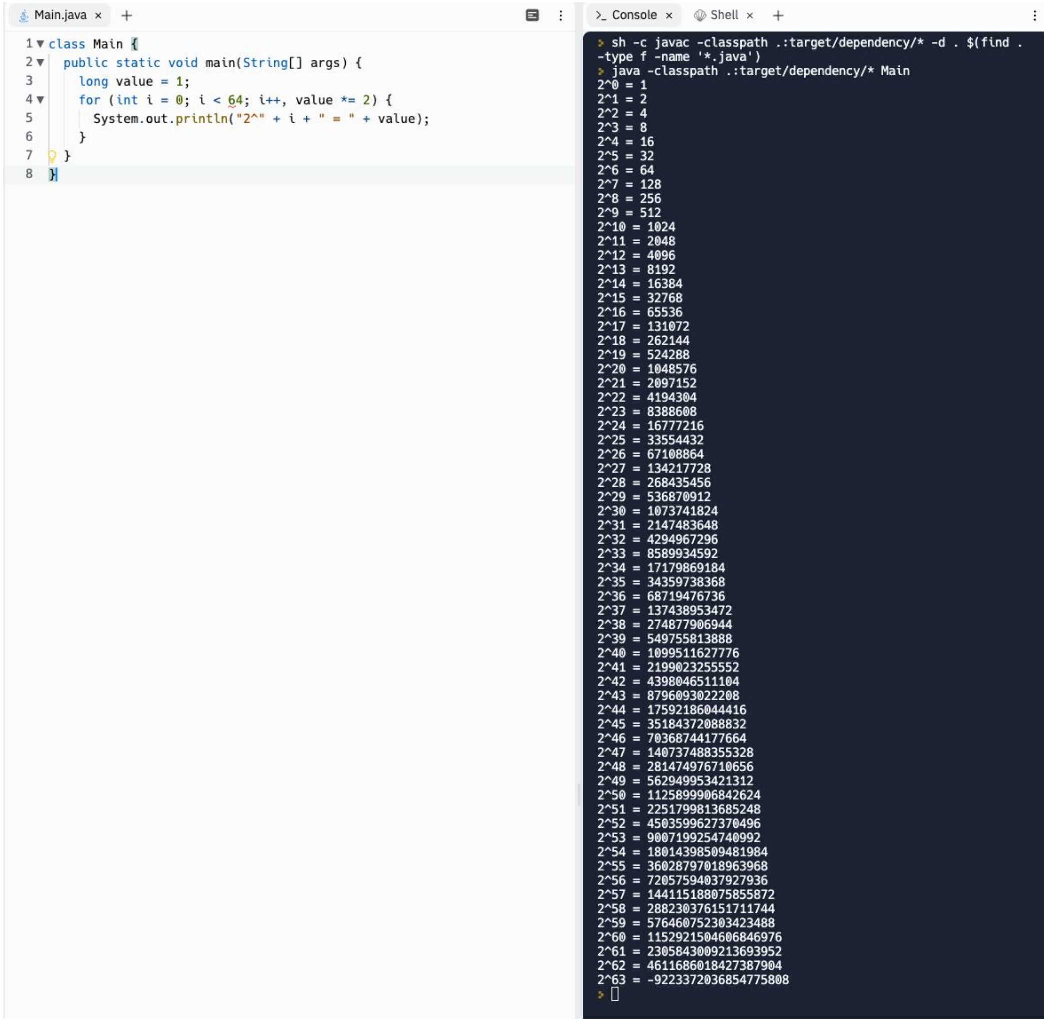


```java
class Main {
  public static void main(String[] args) {
    String s = "Hello, world!";
    for (int i=0; i<s.length(); i++) {
      System.out.println(s.charAt(i));
    }
    for (int i=0, n=s.length(); i<n; i++) {
      System.out.println(s.charAt(i));
    }
  }
}
```

for syntax

The *increment* expression can use "," to update multiple variables.

You can't declare variables of different types in the *initialize* expression, though.



```java
class Main {
  public static void main(String[] args) {
    long value = 1;
    for (int i = 0; i < 64; i++, value *= 2) {
      System.out.println("2^" + i + " = " + value);
    }
  }
}
```

```
> sh -c javac -classpath .:target/dependency/* -d . $(find .
-type f -name '*.java')
> java -classpath .:target/dependency/* Main
2^0 = 1
2^1 = 2
2^2 = 4
2^3 = 8
2^4 = 16
2^5 = 32
2^6 = 64
2^7 = 128
2^8 = 256
2^9 = 512
2^10 = 1024
2^11 = 2048
2^12 = 4096
2^13 = 8192
2^14 = 16384
2^15 = 32768
2^16 = 65536
2^17 = 131072
2^18 = 262144
2^19 = 524288
2^20 = 1048576
2^21 = 2097152
2^22 = 4194304
2^23 = 8388608
2^24 = 16777216
2^25 = 33554432
2^26 = 67108864
2^27 = 134217728
2^28 = 268435456
2^29 = 536870912
2^30 = 1073741824
2^31 = 2147483648
2^32 = 4294967296
2^33 = 8589934592
2^34 = 17179869184
2^35 = 34359738368
2^36 = 68719476736
2^37 = 137438953472
2^38 = 274877906944
2^39 = 549755813888
2^40 = 1099511627776
2^41 = 2199023255552
2^42 = 4398046511104
2^43 = 8796093022208
2^44 = 17592186044416
2^45 = 35184372088832
2^46 = 70368744177664
2^47 = 140737488355328
2^48 = 281474976710656
2^49 = 562949953421312
2^50 = 1125899906842624
2^51 = 2251799813685248
2^52 = 4503599627370496
2^53 = 9007199254740992
2^54 = 18014398509481984
2^55 = 36028797018963968
2^56 = 72057594037927936
2^57 = 144115188075855872
2^58 = 288230376151711744
2^59 = 576460752303423488
2^60 = 1152921504606846976
2^61 = 2305843009213693952
2^62 = 4611686018427387904
2^63 = -9223372036854775808
> 
```

for syntax

`for` (*initialization; condition; increment*)

    statement

initialization, condition, and ***increment*** are all optional

```
for (;;) { … }
```

is an infinite loop, the same as while (true) { … }

Why would you omit initialization?
Sometimes the initialization needed takes multiple statements and can't be easily expressed in a single expression, so you do it before the for loop and omit the initialization.

Why would you omit increment?
Increment logic may similarly get complicated and be better expressed within the body of the loop.

for syntax

`for` (*initialization; condition; increment*)
    statement

for statements are very frequently used with numeric counters, usually integers.
But they don't have to be… the expressions can be most anything.

```
// Cast spell to magically look east as far as possible.
for (Room room = player.getLocation();
     room != null;
     room = room.getEast()) {
  printRoomContents(room);
}
```

`for` is very flexible in this way.
Some languages like BASIC have a FOR statement that only can initialize and increment a numeric counter.

return in loops

You can exit a while or for loop by returning out of the enclosing method.
One student wrote this for the TruthGame pickNext method:

```
public int pickNext() {
  for (int i=1; i<=3; i++) {
    if (isTruth(i)) {
      return i;
    }
  }
  return -1;
}
```

break

You can also exit a while or for loop and just move on to the next statement after the loop, using **break**.

```
while (true) {
  String command = scanner.nextLine();
  if (command.equals("quit")) {
    break;
  }
  ...
}
System.out.println("Well, goodbye, then!");
```

continue

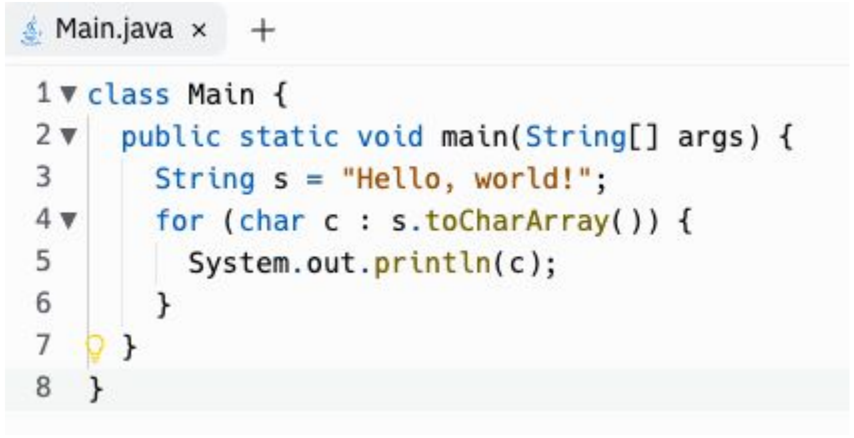
The continue statement jumps back to the top of the loop and re-evaluates the condition. It's useful for skipping the body of the loop, like to ignore blank lines. Otherwise, you'd need a big if/else. It works in while, do/while, or for loops.

```
while (scanner.hasNextLine()) {
  String line = scanner.nextLine();
  if (line.equals("")) {
    continue;
  }
  processNonEmptyLine(line);
}
```

for each loop

In 2004, Java added the "for each loop" or enhanced for loop, which is nice syntactic sugar for iterating over collections. This resembles the `for` statement in Python which iterates over a sequence.

It'll be covered later in Unit 6… for now, let's focus on the **general form** of the for statement that we just learned.



```java
class Main {
  public static void main(String[] args) {
    String s = "Hello, world!";
    for (char c : s.toCharArray()) {
      System.out.println(c);
    }
  }
}
```

Practice!

Replit: ForBuzz

Replit: WordList