INFO1113 Object-Oriented Programming

Week 2A: Control Flow, Loops and Static Methods

Slides developed by Tyson Thomas

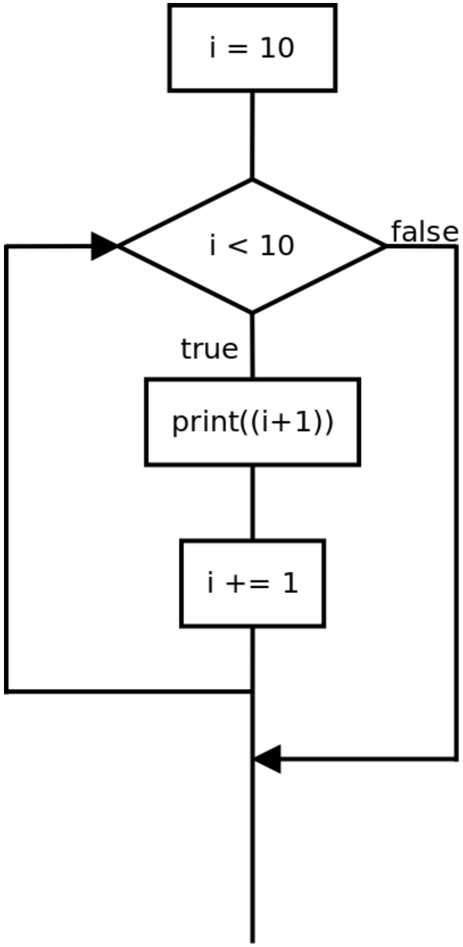
Copyright Warning

Topics

- Control flow (s. 3)
- While loop (s. 16)
- For loop (s. 21)
- Static methods (s. 30)

Loops

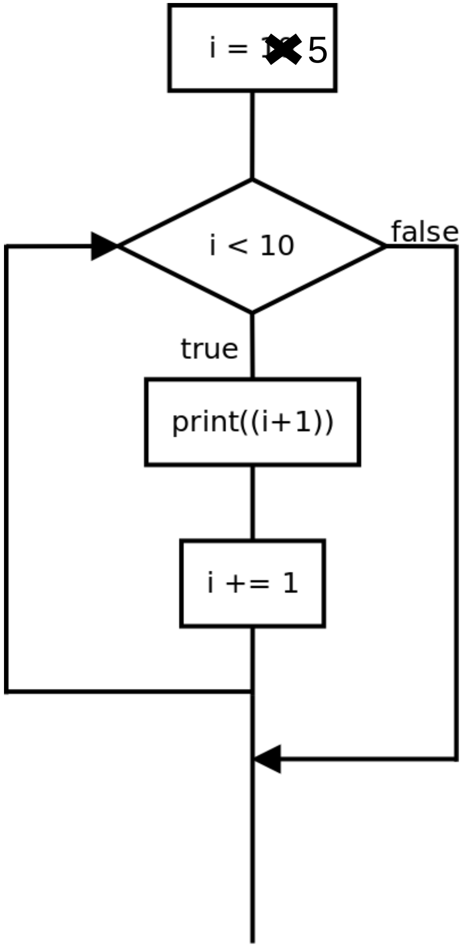
Remember flow control diagrams?



Refer to Chapter 4.1, page 237, (Java, An Introduction to Problem Solving & Programming, Savitch & Mock)

Loops

Remember flow control diagrams?



What if we changed i to **5**?

What would be the output of this program?

Refer to Chapter 4.1, page 237, (Java, An Introduction to Problem Solving & Programming, Savitch & Mock)

Loops

4 types of loops we can write within Java.

- while
- do-while
- for
- for-each

The constructs are part of the language’s syntax and typically follow a similar pattern.

Loops

Syntax: `while (`*condition*`)` statement

As with if statements, for this branch to start and *continue* execution the *condition* must be **true**.

```
while(condition) {


}
```

Refer to Chapter 4.1, pages 238-251, (Java, An Introduction to Problem Solving & Programming, Savitch & Mock)

Loops

Syntax: `while (`*`condition`*`)` statement

As with if statements, for this branch to start and *continue* execution the *condition* must be **true**.

```
while(condition) {


}
```

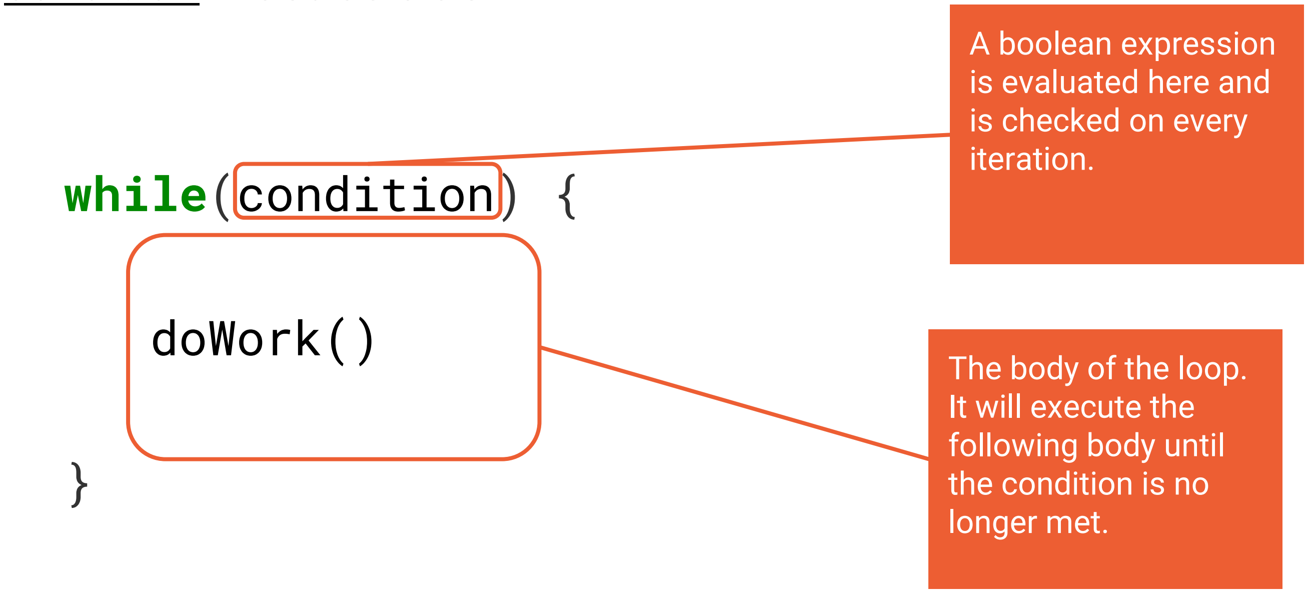
A boolean expression is evaluated here and is checked on every iteration.

Refer to Chapter 4.1, pages 238-251, (Java, An Introduction to Problem Solving & Programming, Savitch & Mock)

Loops

Syntax: while (*condition*) statement

As with if statements, for this branch to start and *continue* execution the *condition* must be **true**.



```
while(condition) {
    doWork()
}
```

A boolean expression is evaluated here and is checked on every iteration.

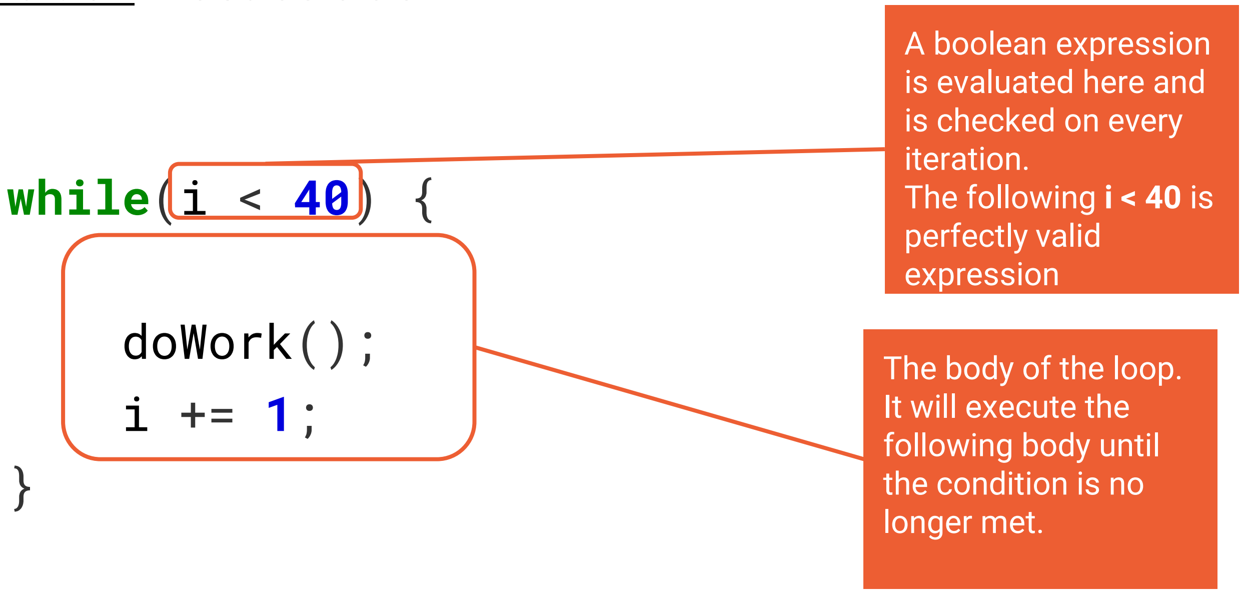
The body of the loop. It will execute the following body until the condition is no longer met.

Refer to Chapter 4.1, pages 238-251, (Java, An Introduction to Problem Solving & Programming, Savitch & Mock)

Loops

Syntax: while (*condition*) statement

As with if statements, for this branch to start and *continue* execution the *condition* must be **true**.



```
while(i < 40) {
    doWork();
    i += 1;
}
```

A boolean expression is evaluated here and is checked on every iteration.
The following **i < 40** is perfectly valid expression

The body of the loop. It will execute the following body until the condition is no longer met.

Refer to Chapter 4.1, pages 238-251, (Java, An Introduction to Problem Solving & Programming, Savitch & Mock)

Loops

Syntax: do {} while(*condition*) statement

Similar to the while loop but it will always execute the block at-least ***once*** and *continue* execution if *condition* is **true**.

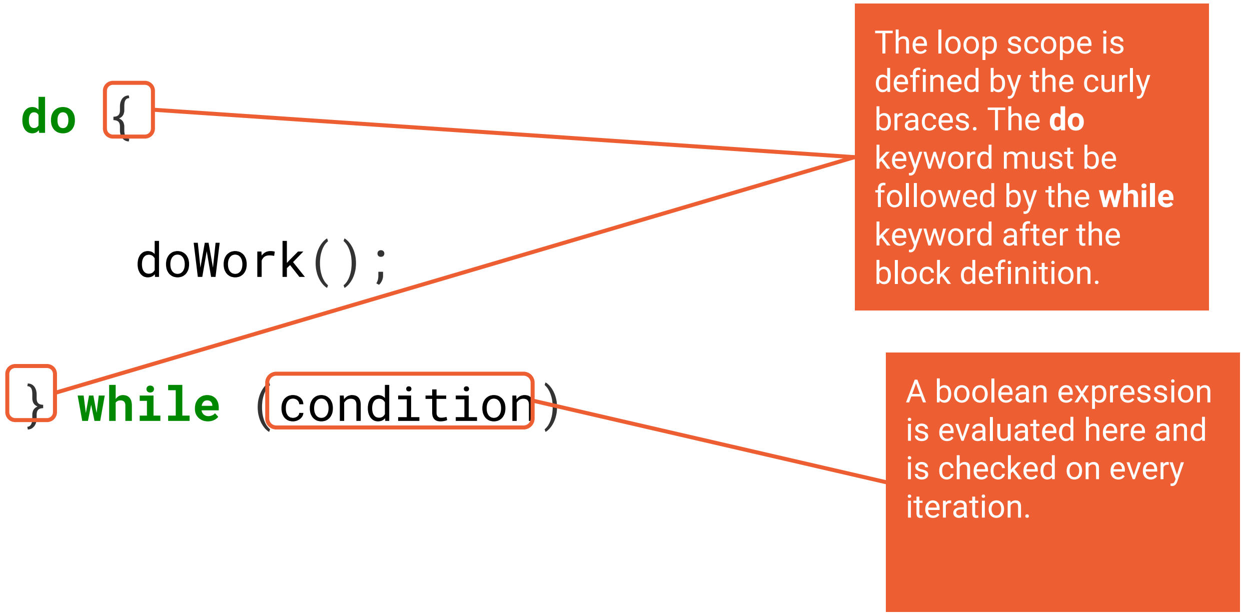
```
do {

    doWork();

} while (condition)
```

Refer to Chapter 4.1, pages 238-251, (Java, An Introduction to Problem Solving & Programming, Savitch & Mock)

Loops

Syntax: `do {} while(`*`condition`*`)` statement

Similar to the while loop but it will always execute the block at-least ***once*** and *continue* execution if *condition* is **true**.



```
do {

     doWork();

} while (condition)
```

The loop scope is defined by the curly braces. The **do** keyword must be followed by the **while** keyword after the block definition.

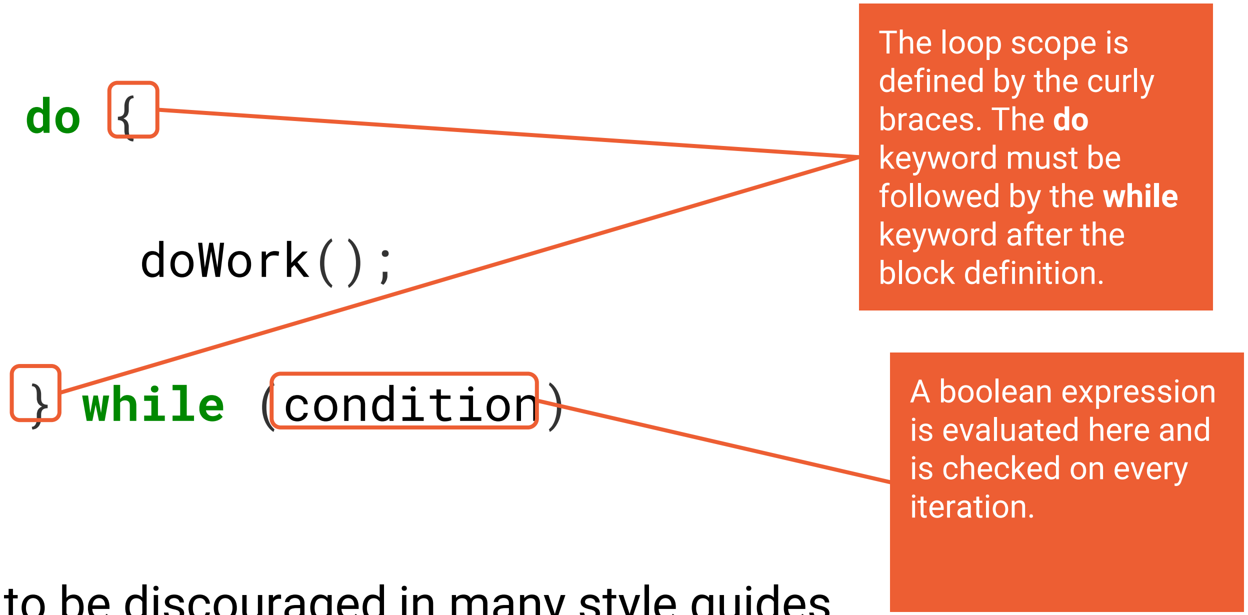
A boolean expression is evaluated here and is checked on every iteration.

Refer to Chapter 4.1, pages 238-251, (Java, An Introduction to Problem Solving & Programming, Savitch & Mock)

Loops

Syntax: `do {} while(`*__`condition`__*`)` statement

Similar to the while loop but it will always execute the block at-least ***once*** and *continue* execution if *__condition__* is **true**.



```
do {

    doWork();

} while (condition)
```

The loop scope is defined by the curly braces. The **do** keyword must be followed by the **while** keyword after the block definition.

A boolean expression is evaluated here and is checked on every iteration.

Note: do-while is known to be discouraged in many style guides

Refer to Chapter 4.1, pages 238-251, (Java, An Introduction to Problem Solving & Programming, Savitch & Mock)

Let’s write some loops!

Loops

Syntax: for(*[variable]*; *[condition]*; *[update]*) statement

for loops are broken up into 3 separate sections. **Variables, Conditions** and **Updates** sections.

```
for( [variable]; [condition]; [update] )
{

   doWork();

}
```

Refer to Chapter 4.1, pages 251-260, (Java, An Introduction to Problem Solving & Programming, Savitch & Mock)

Loops

Syntax: for(*[variable]; [condition]; [update])* statement

for loops are broken up into 3 separate sections. **Variables, Conditions** and **Updates** sections.

```
for( [variable]; [condition]; [update] ) {

   doWork();

}
```

We are able to create and initialise variables for our loop here. They will be restricted to the loop’s scope.

Refer to Chapter 4.1, pages 251-260, (Java, An Introduction to Problem Solving & Programming, Savitch & Mock)

Loops

Syntax: for(*[variable]*; *[condition]*; *[update]*) statement

for loops are broken up into 3 separate sections. **Variables, Conditions** and **Updates** sections.

```
for( int i = 0; [condition]; [update] ) {

  doWork();

}
```

A common variable is a counter for our for loop.

Refer to Chapter 4.1, pages 251-260, (Java, An Introduction to Problem Solving & Programming, Savitch & Mock)

Loops

Syntax: *`for( [variable]; [condition]; [update])`*
statement

`for` loops are broken up into 3 separate sections. **Variables, Conditions** and **Updates** sections.

```
for( int i = 0; [condition]; [update] ) {

   doWork();

}
```

The boolean expression to the inputted here. No different than a while loop

Refer to Chapter 4.1, pages 251-260, (Java, An Introduction to Problem Solving & Programming, Savitch & Mock)

Loops

Syntax: for(*[variable]*; *[condition]*; *[update]*) statement

for loops are broken up into 3 separate sections. **Variables, Conditions** and **Updates** sections.

```
for( int i = 0; i < 10; [update] ) {

   doWork();

}
```

Let's say we wanted to loop 10 times

Refer to Chapter 4.1, pages 251-260, (Java, An Introduction to Problem Solving & Programming, Savitch & Mock)

Loops

Syntax: `for( [variable]; [condition]; [update])` statement

`for` loops are broken up into 3 separate sections. **Variables, Conditions** and **Updates** sections.

```
for( int i = 0; i < 10; [update] ) {

  doWork();

}
```

This is the update component. Were we update any variables defined within the ***variable*** section (**or** variables defined in the outer scope_

Refer to Chapter 4.1, pages 251-260, (Java, An Introduction to Problem Solving & Programming, Savitch & Mock)

Loops

Syntax: `for(` *[variable]*`;` *[condition]*`;` *[update]*`)` statement

`for` loops are broken up into 3 separate sections. **Variables, Conditions** and **Updates** sections.

```
for( int i = 0; i < 10; i += 1) {

  doWork();

}
```

So we can increment by 1, similar to the while loop.

Refer to Chapter 4.1, pages 251-260, (Java, An Introduction to Problem Solving & Programming, Savitch & Mock)

Loops

Syntax: for(*[variable]*; *[condition]*; *[update]*) statement

for loops are broken up into 3 separate sections. **Variables, Conditions** and **Updates** sections.

```java
for( int i = 0; i < 10; i += 1) {

   doWork();

}
```

So we can increment by 1, similar to the while loop.

Regardless you can always rewrite a **for** loop as a **while** loop.

Refer to Chapter 4.1, pages 251-260, (Java, An Introduction to Problem Solving & Programming, Savitch & Mock)

Using a for-loop!

Okay, what about for-each?

Loops

Syntax: for(*binding* : *collection*) statement

for-each loops involve the use of **iterators** (exception being arrays).

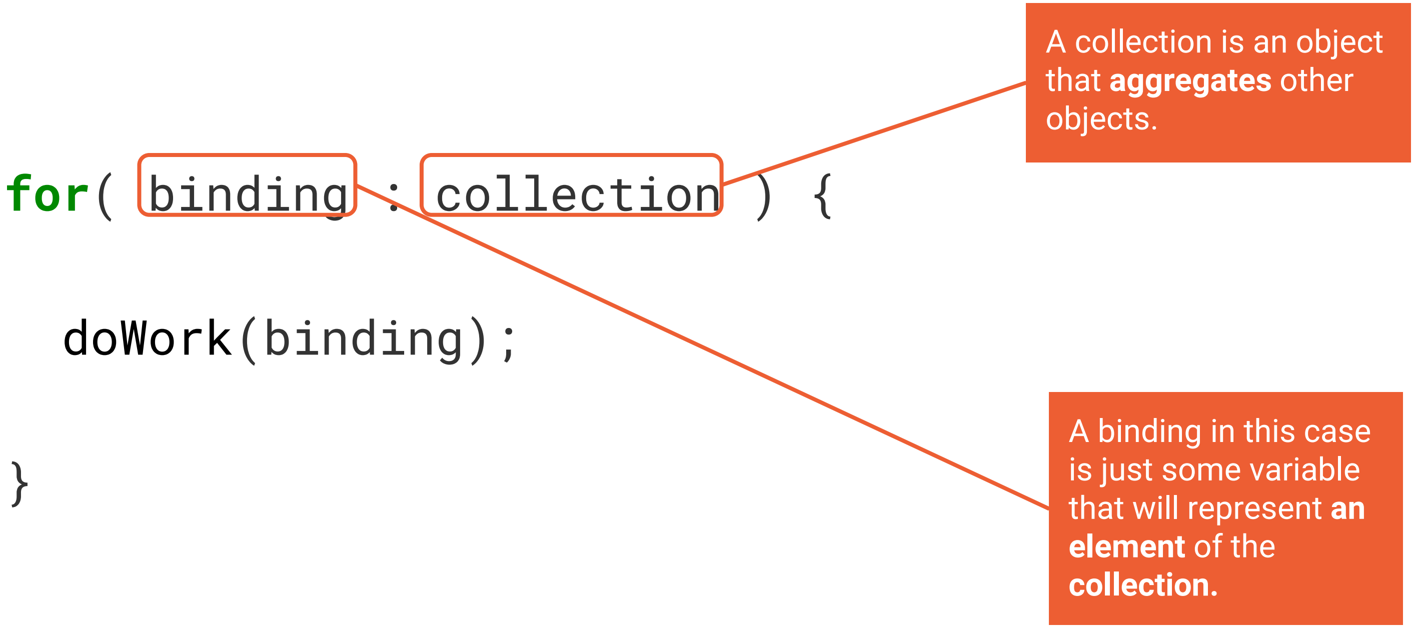
```
for( binding : collection ) {
   doWork(binding);
}
```

Refer to Chapter 4.1, pages 260, (Java, An Introduction to Problem Solving & Programming, Savitch & Mock)

Loops

Syntax: `for(` *binding* `:` *collection* `)` statement

`for`-each loops involve the use of **iterators** (exception being arrays).



```
for( binding : collection ) {

   doWork(binding);

}
```

A collection is an object that **aggregates** other objects.

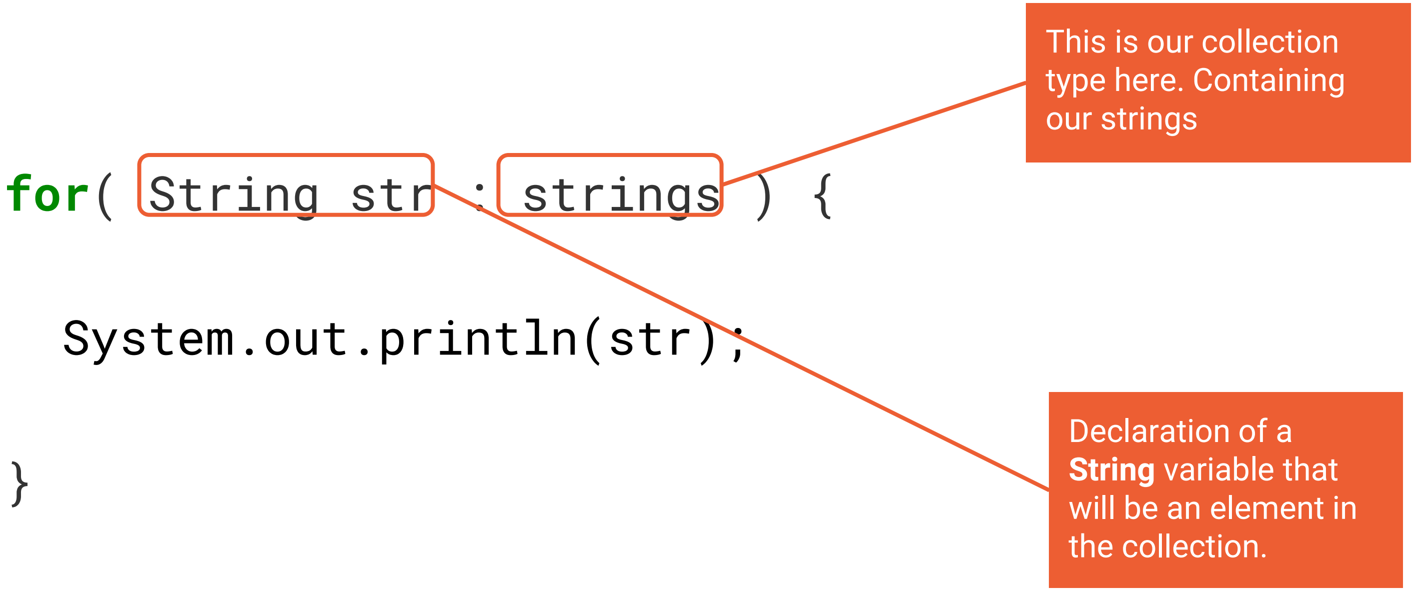
A binding in this case is just some variable that will represent **an element** of the **collection.**

Refer to Chapter 4.1, pages 260, (Java, An Introduction to Problem Solving & Programming, Savitch & Mock)

Loops

Syntax: `for(` *`binding`* `:` *`collection`* `)` statement

`for`-each loops involve the use of **iterators** (exception being arrays).



```
for( String str : strings ) {

  System.out.println(str);

}
```

This is our collection type here. Containing our strings

Declaration of a **String** variable that will be an element in the collection.

Refer to Chapter 4.1, pages 260, (Java, An Introduction to Problem Solving & Programming, Savitch & Mock)

Loops

Syntax: `for(` *binding* `:` *collection* `)` statement

`for`-each loops involve the use of **iterators** (exception being arrays).

```
for( String str : strings ) {
  System.out.println(str);
}
```

String[],
ArrayList<String>,
List<String>,
Set<String>,
Deque<String>
…

What kind of object **aggregates** other objects?
Anything that implements **hasNext(), next(),** and optionally **remove()**

Refer to Chapter 4.1, pages 260, (Java, An Introduction to Problem Solving & Programming, Savitch & Mock)

Loops

Syntax: for(*binding* : *collection*) statement

for-each loops involve the use of **iterators** (exception being arrays).

```
for( String str : strings ) {
  System.out.println(str);
}
```

What information are we missing by using a **for**-each loop?

Refer to Chapter 4.1, pages 260, (Java, An Introduction to Problem Solving & Programming, Savitch & Mock)

Loops

Syntax: `for(` *binding* `:` *collection* `)` statement

`for`-each loops involve the use of **iterators** (exception being arrays).

```
for( String str : strings ) {
  System.out.println(str);
}
```

What information are we missing by using a **for-**each loop? **an array index**

Refer to Chapter 4.1, pages 260, (Java, An Introduction to Problem Solving & Programming, Savitch & Mock)

Using a for-each loop

Static Methods

Syntax:

```
static [final] return_type name ([parameters])
```

A method is a stored set of instructions bound to an object. In the case of a static method, the object is the class which it is defined in.

Example:

```
public static int addThree(int a, int b, int c) {
      return a+b+c;
}
```

Refer to Chapter 6.2, pages 433-436, (Java, An Introduction to Problem Solving & Programming, Savitch & Mock)

Static Methods

Binds the method to the class. Without **static** it is an instance method.

Syntax:

```
static [final] return_type name ([parameters])
```

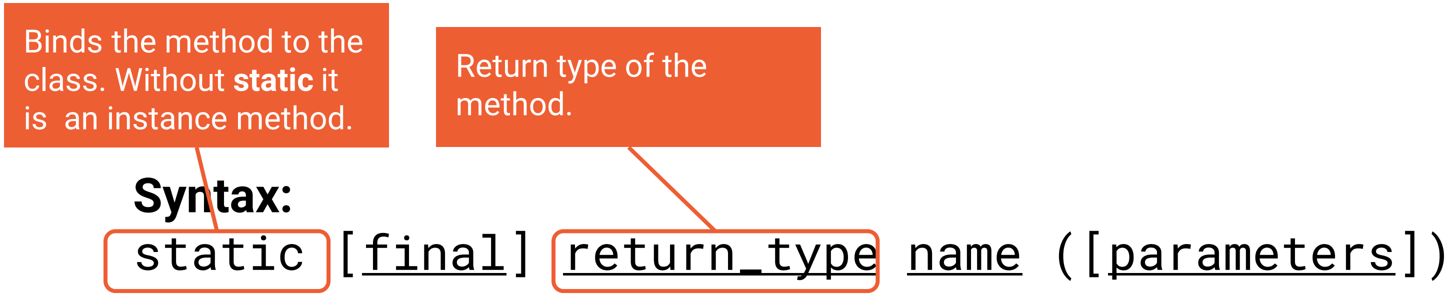
A method is a stored set of instructions bound to an object. In the case of a static method, the object is the class which it is defined in.

Example:

```
public static int addThree(int a, int b, int c) {
      return a+b+c;
}
```

Refer to Chapter 6.2, pages 433-436, (Java, An Introduction to Problem Solving & Programming, Savitch & Mock)

Static Methods



Binds the method to the class. Without **static** it is an instance method.

Return type of the method.

Syntax:

```
static [final] return_type name ([parameters])
```

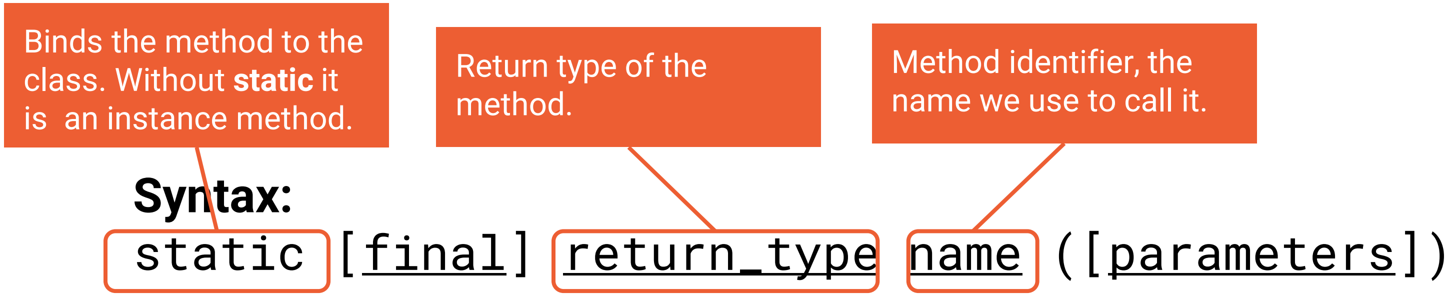
A method is a stored set of instructions bound to an object. In the case of a static method, the object is the class which it is defined in.

Example:

```
public static int addThree(int a, int b, int c) {
        return a+b+c;
}
```

Refer to Chapter 6.2, pages 433-436, (Java, An Introduction to Problem Solving & Programming, Savitch & Mock)

Static Methods



Binds the method to the class. Without **static** it is an instance method.

Return type of the method.

Method identifier, the name we use to call it.

Syntax:

```
static [final] return_type name ([parameters])
```

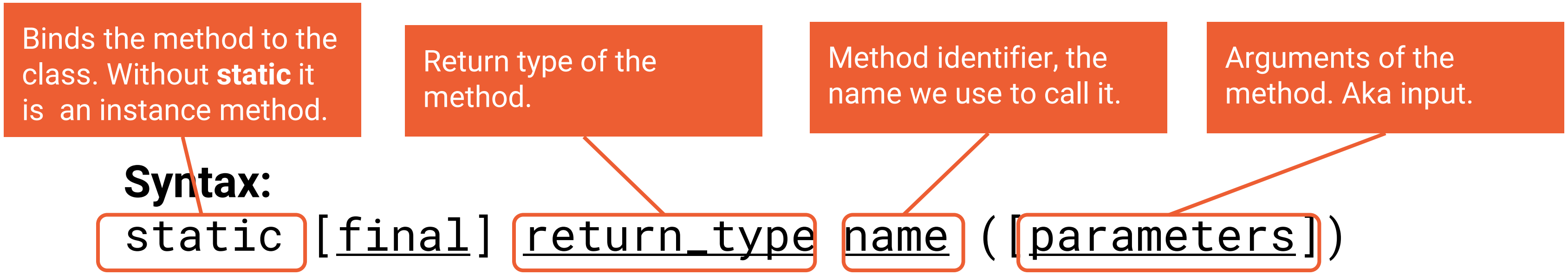
A method is a stored set of instructions bound to an object. In the case of a static method, the object is the class which it is defined in.

Example:

```java
public static int addThree(int a, int b, int c) {
       return a+b+c;
}
```

Refer to Chapter 6.2, pages 433-436, (Java, An Introduction to Problem Solving & Programming, Savitch & Mock)

Static Methods



Binds the method to the class. Without **static** it is an instance method.

Return type of the method.

Method identifier, the name we use to call it.

Arguments of the method. Aka input.

Syntax:

```
static [final] return_type name ([parameters])
```

A method is a stored set of instructions bound to an object. In the case of a static method, the object is the class which it is defined in.

Example:

```
public static int addThree(int a, int b, int c) {
      return a+b+c;
}
```

Refer to Chapter 6.2, pages 433-436, (Java, An Introduction to Problem Solving & Programming, Savitch & Mock)

Static Methods

Syntax:

```
static [final] return_type name ([parameters])
```

A method is a stored set of instructions bound to an object. In the case of a static method, the object is the class which it is defined in.

Example:

```
public static return_type addThree(int a, int b, int c)
{
        return a+b+c;
}
```

The return type can be any **primitive type**, **reference type** or **void**

Refer to Chapter 6.2, pages 433-436, (Java, An Introduction to Problem Solving & Programming, Savitch & Mock)

Return types

Java can use any primitive or reference type as a **return** type. The compiler will check and ensure that any assignment to the return value of a method is correct.

There is a special return type that you **may** or **may not** have encountered: **void.**

void does not return any value and any void method is typically used for manipulating passed data or output.

Returning data from a method is generally used for **querying** or object **creation.**

Refer to Chapter 6.2, pages 433-436, (Java, An Introduction to Problem Solving & Programming, Savitch & Mock)

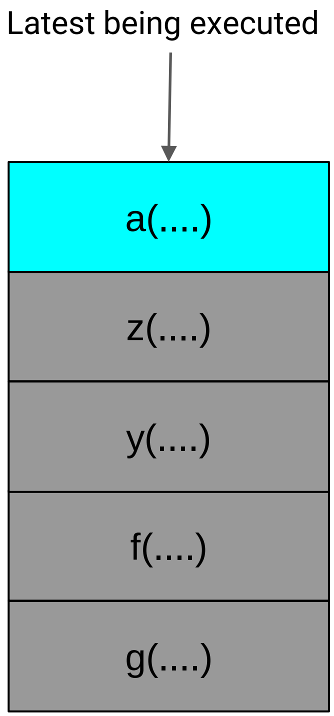
Static methods

Call Stack

Java is a stack-based language so when a method is executed it is put onto a ***call-stack***.

The method being executed at the top of the stack is the most recently called method.

A method finishes executing once it has reached a return state or for **void** method, once it has reached the end of method scope.

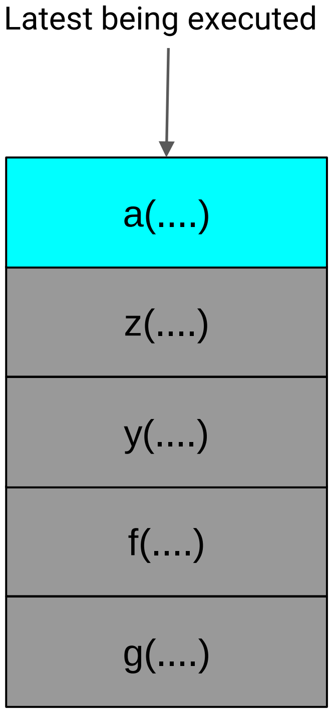


Refer to JVM Specification, Java Virtual Machine Stacks, (https://docs.oracle.com/javase/specs/jvms/se8/html/jvms-2.html#jvms-2.5.2)

Call Stack

Each method executed gets a **Frame** allocated and a **frame** will hold data, partial results, return values and **dynamic linking**.

A **Frame** is created when a method is invoked at runtime by the java virtual machine.



Refer to JVM Specification, Java Virtual Machine Stacks, (https://docs.oracle.com/javase/specs/jvms/se8/html/jvms-2.html#jvms-2.6)

Let’s break down the call stack

See you next time!