Simplicity *itself*

Introducing Closures

Imagine you want to filter a list of strings by their length

In Java (pre version 8)

```
List<String> result = new ArrayList<>();
for (String str : strings) {
    if (str.length() < 10) result.add(str);
}
return result;
```


What if you want to filter by starting characters?

In Java (pre version 8)

```
List<String> result = new ArrayList<>();
for (String str : strings) {
    if (str.startsWith("P")) result.add(str);
}
return result;
```

The only difference is this predicate
(function that returns a boolean)

What if we could re-use the filter algorithm by making the predicate an argument?

Now imagine you want to sort
a list of strings by their lengths

In Java (pre version 8)

```
Collections.sort(strings, new Comparator() {
    int compare(String s1, String s2) {
        return s1.length().compareTo(s2.length());
    }
});
```

This is the only relevant bit: a
function that returns an integer value.
The rest is noise/boilerplate.

Closures are function objects that you can pass as arguments, or assign to variables

In Groovy

```
return strings.findAll { it.length() < 10 }
```

```
return strings.findAll { it.startsWith("P") }
```

```
return strings.sort { it.length() }
```

Returns a value that can be sorted on (the length of each string)

Breaking it down

findAll() is a method that *calls* the given function for each element in the list to determine whether that element should be included in the result

```
return strings.findAll { it.length() < 10 }
```

This is a single-argument function object that returns a boolean. In Groovy, this object is a *closure*.

Breaking it down

```
List findAll(Closure predicate) {
    List<String> result = []
    for (str in strings) {
        if (predicate.call(str)) result << str
    }
    return result
}
```

We can call the closure to get a result

As objects, closures can be assigned to variables and reused:

Think of this as a closure literal, like a list literal, [], or a string literal, "". It's an *instance*.

```
def predicate = { it.length() < 10 }

println strings.findAll(predicate)
println names.findAll(predicate)
println cities.findAll(predicate)
```


Closure forms

A closure that has zero arguments:

```
def fn = {-> true }
```

A function that always returns `true` when called

A closure that has typed arguments:

```
def fn = { int i, String name ->
    println "${name} (at ${i})"
}
```

The `->` terminates the argument list and starts the function body

A closure that has untyped arguments:

```
def fn = { i, name -> return name[i] }
```

The `return` is optional

Closure forms

A closure that has an implicit argument:

```
def fn = { it.length() }
```

Note the lack of ->

The implicit argument is always called `it`

As soon as you use the arrow, ->, the implicit argument disappears. In other words, you will get a `MissingPropertyException` if you try to reference `it`.

A closure…

- can have arguments
  - typed or untyped
- can return a value
  - explicitly or implicitly

A closure is like a method in that it has an argument list, can be called, and can return a value. The difference is that it has no name and doesn't belong to a class/type.

Closures in method calls

Valid if the closure is the last argument:

```
return strings.findAll { it.length() < 10 }
return strings.findAll() { it.length() < 10 }
return numbers.inject(0) { sum, i -> sum + i }
```

Always valid:

```
return strings.findAll({ it.length() < 10 })
return numbers.inject(0, { sum, i -> sum + i })
```

Again, think of a closure literal, { . . . }, like any other literal

Examples - filtering

Filter for all collection/sequence items matching given criteria:

```
return strings.findAll { it.length() < 10 }
```

Return the first element matching the given criteria:

```
return strings.find { it.length() < 10 }
```

Examples - mapping

Map the elements of a list to a new list by applying a given function. This example creates a list containing the lengths of the strings:

```
strings.collect { it.length() }
```

The mapping function

Fetch the names of people (assumes each person object has a `name` property):

```
people.collect { it.name }
```

Create a list of the first ten squares, i.e. x^2:

```
(1..10).collect { it ** 2 }
```


Examples - iterating

Apply a function to every item:

```
strings.each {
    println "Item: ${it}"
}
```

Basically like a 'for' loop, but doesn't support `break` and `continue`. Also, `return` doesn't work as many people expect.

Examples - sorting

Sort by any sortable value (something that implements Comparable):

```
strings.sort { it.length() }
```

Sort by string lengths

Sort via a custom comparator:

```
strings.sort { a, b -> a.compareToIgnoreCase(b) }
```

Case insensitive sort

Examples - filesystem

Sort by any sortable value (something that implements Comparable):

```
strings.sort { it.length() }
```

Sort by string lengths

Sort via a custom comparator:

```
strings.sort { a, b -> a.compareToIgnoreCase(b) }
```

Case insensitive sort

Context

Variable references

```
def n = 3
def adder = { int a ->
    return a + n
}

println adder.call(5)

n = 1
println adder.call(5)
```

Captures a *reference* to n

Prints 8 (5 + 3)

Prints 6 (5 +1)

The closure 'sees' changes in the value of n, even though it's defined outside of the closure

Resources

Java 8 Lambdas tutorial (lambdas are similar to closures):

https://docs.oracle.com/javase/tutorial/java/javaOO/lambdaexpressions.html

Syntax reference:

http://www.groovy-lang.org/closures.html