Day 6

Lambda Example in JavaScript

```
const process = (req, resp) => {
   resp.status(200)
      .type(‘text/html’)
      .send(`<h1>Current time is ${new Date()}`)
}
...
// app is an Express application
app.get(‘/time’, process)
```

A function

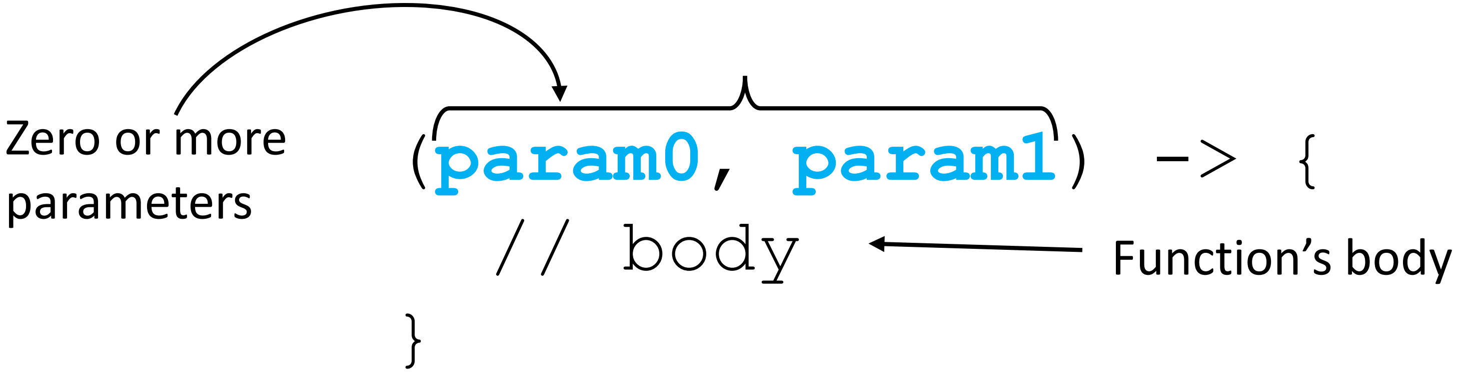
or

```
// app is an Express application
app.get(‘/time’, (req, resp) => {
   resp.status(200)
      .type(‘text/html’)
      .send(‘<h1>Current time is ${new Date()}`)
})
```

A function

What is Lambda Expression?

- Is an expression that represents an anonymous function
  - Can be assigned to a variable
  - Passed as arguments into methods
  - Returned values from methods
- Allow developers to write more declarative and concise program
- Lambda expressions are not classes
  - `this` refers to the enclosing object not the lambda expression



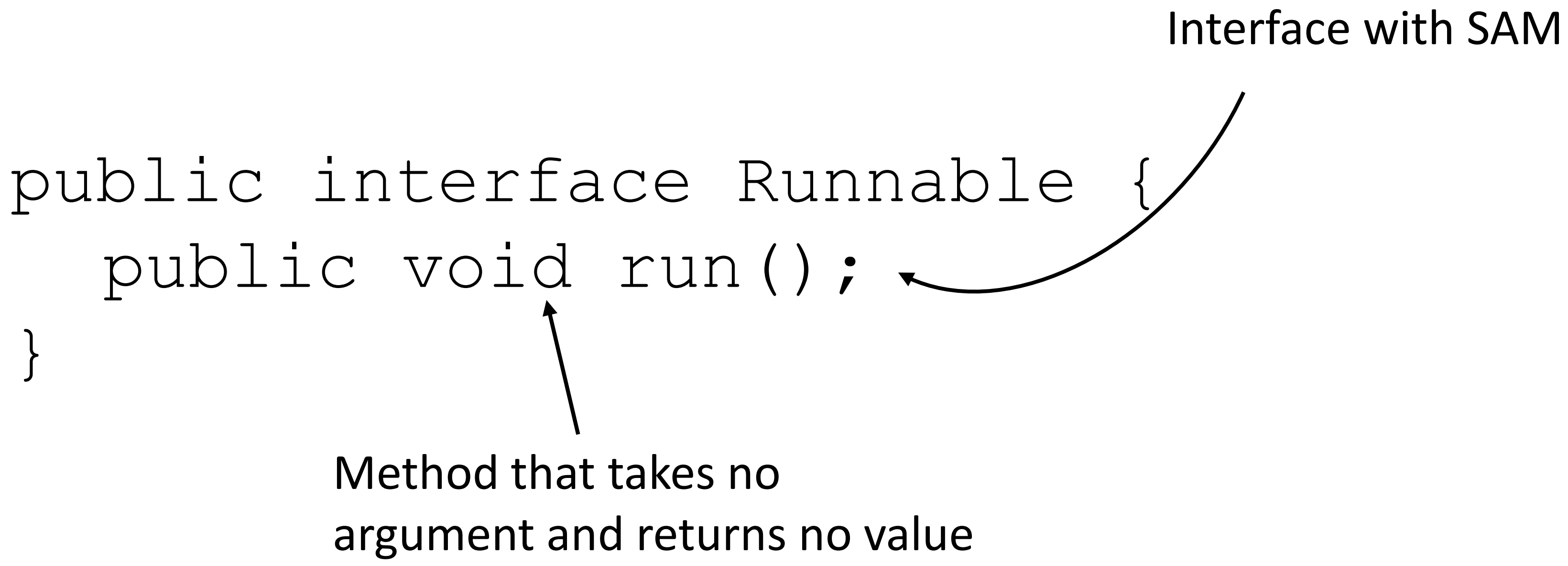
Zero or more parameters

```
(param0, param1) -> {
  // body
}
```

Function's body

Function Type

- Lambda function type is any interface with a single abstract method
  - SAM - Single Abstract Method
- Examples of SAM
  - Runnable, Callable, Function, Predicate



Interface with SAM

```
public interface Runnable {
   public void run();
}
```

Method that takes no
argument and returns no value

Example - `Runnable` Lambda Expression

```
public interface Runnable {
  public void run();
}
```

No parameters

IMPORTANT

Lambda expression will match a method when the signature of the lambda matches the signature of the method

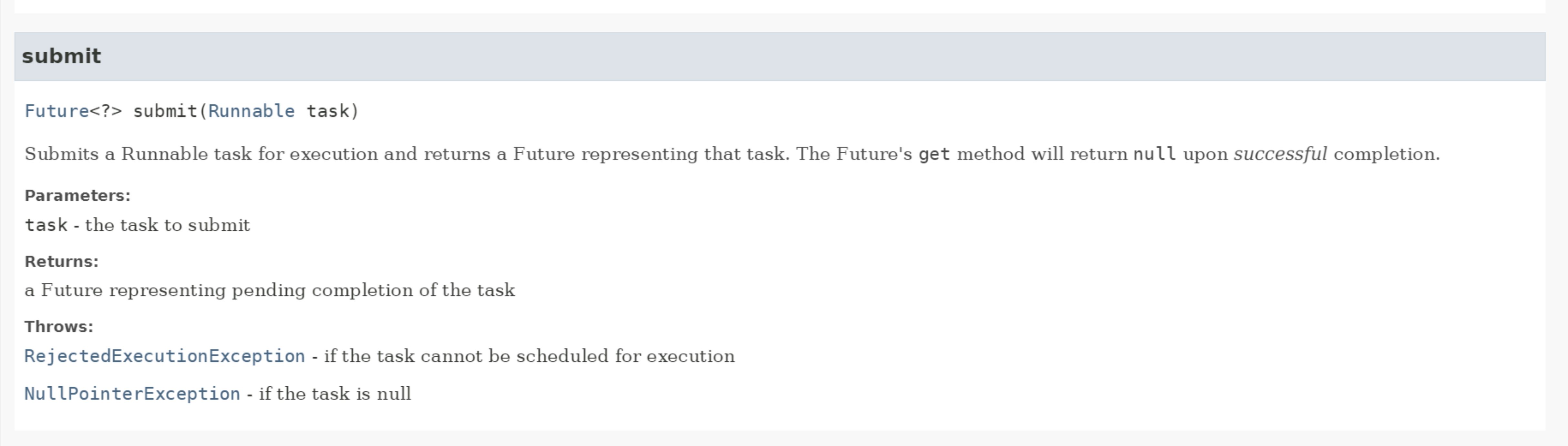
Method name is not taken into consideration

```
() -> {
  System.out.println(“hello, world”);
}
```

No return value, matches the `void` in `run()`

Example - Lambda Expression

`ExecutorService`



submit

`Future<?> submit(Runnable task)`

Submits a Runnable task for execution and returns a Future representing that task. The Future's `get` method will return `null` upon *successful* completion.

Parameters:

`task` - the task to submit

Returns:

a Future representing pending completion of the task

Throws:

`RejectedExecutionException` - if the task cannot be scheduled for execution

`NullPointerException` - if the task is null

From https://docs.oracle.com/en/java/javase/17/docs/api/java.base/java/util/concurrent/ExecutorService.html#submit(java.lang.Runnable)

Example - Using Lambda Expression

```
ExecutorService threadPool = Executors.newFixedThreadPool(2);

threadPool.submit(
  () -> {
    System.out.println(“hello, world”);
  }
);
```

This is the body of the
Runnable interface

Example - List

```
 List<Customer> customers = // Get a list of customers
```

‘External’ loop - the program iterates the collection

```
for (Customer c: customers)
   System.out.printf(“Id: %d, Name: %s, Email: %s\n”,
         c.getCustomerId(), c.getName(), c.getEmail());
```

‘Internal’ loop - the collection iterates over itself. Pass value to the logic

```
customers.forEach(
   c -> {
      System.out.printf(“Id: %d, Name: %s, Email: %s\n”,
            c.getCustomerId(), c.getName(), c.getEmail());
   }
)
```

Method Reference

- Method reference allows us to reuse defined methods as lambda expression
  - Static methods, instance methods

```
List<String> words = new LinkedList<>();
// populate words List
...
words.forEach(w -> System.out.println(w));
```

```
words.forEach(System.out::println);
```

Example - Method Reference - Instance

```
public class LineItem {
   public void print() {
      // Print line item details
      System.out.println(...);
   }
}

List<LineItem> lineItems = new LinkedList<>();
// populate lineItems
...
lineItems.forEach(li -> li.print());
```

References the method
in the instance

```
lineItems.forEach(LineItem::print)
```

Example - Reference Method - Constructor

```
List<Customer> custList = new LinkedList<>();
int count = 5;
for (int i = 0; i < count; i++)
    custList.add(new Customer());
```

```
public <T> List<T> create(int count, Supplier<T> supplier) {
    List<T> list = new LinkedList<T>();
    for (int i = 0; i < count; i++)
        list.add(supplier.get());
    return list;
}
```

Supplier

```
List<Customer> custList = create(5, ()-> new Customer());
```

```
List<Customer> custList = create(5, Customer::new);
```

Collection vs Streams

Cars in carpark

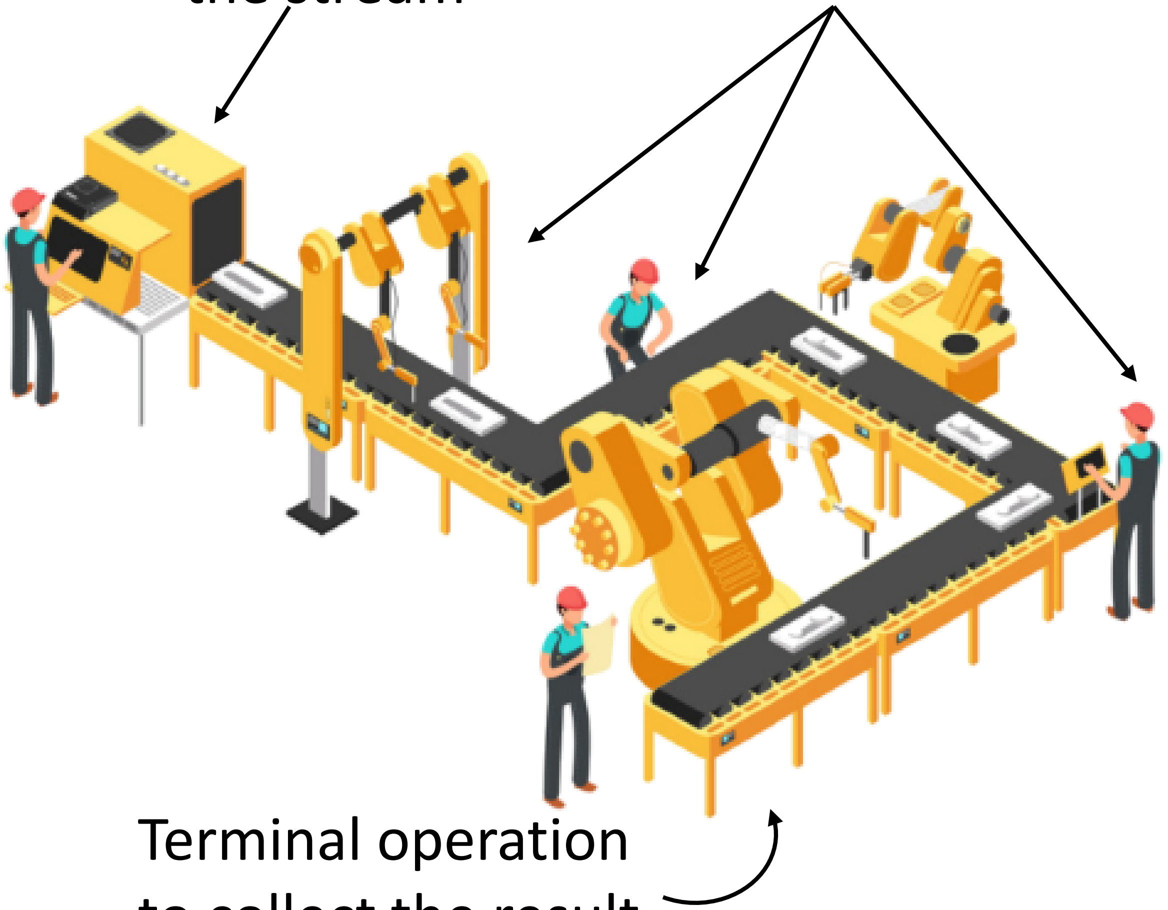
Cars on highway

What are Streams?

- A sequence of data elements like water flowing through a pipeline
  - Unlike collection which are in-memory data structure, stream consume very little memory
- Aggregation and transformation operations performed on these data element as they traverse the pipeline
- Streams can be formed
  - From collections eg. list, set, values or keys from maps,
  - Programmatically generated eg. 20 random numbers
- Streams can be infinite
  - Eg. metrics from IoT sensor

Stream Operations

- A stream consist of 3 types of objects
  - A source
  - Zero or more intermediate operations to be performed on each data element
  - Terminal operation to collect the data



Source of the stream

Intermediate operations

Terminal operation to collect the result

Different Stages

- Stream source - different ways to create data stream
  - Collections with `.stream()` method
  - Generators eg `IntStream.range()`
- Intermediate Operations are operations to be performed on the data element
  - map
  - filter
  - limit, skip
  - max, min
  - count
  - distinct
- Each of the intermediate operations produces a stream
- Terminal operations are operations that collects the result ending the pipeline
  - convert stream to list, set, etc.
  - reduce

Example - Streams

```
// listOfWords is populated with words
List<String> listOfWords = ...
List<String> evenLengthWords = new LinkedList<>();

for (String w: listOfWords) {
   if 0 == (w.length() & 1)
      eventLengthWords.add(w.toUpperCase());
}
```

```
// listOfWords is populated with words
List<String> listOfWords = ...

List<String> eventLengthWords = listOfWords.stream()
      .filter(w -> 0 == (w.length() & 1))
      .map(String::toUpperCase)
      .collect(Collectors.toList());
```

Turn a collection into a stream

Apply a series of operations on each data element in the stream

Aggregate the result

Example - Streams

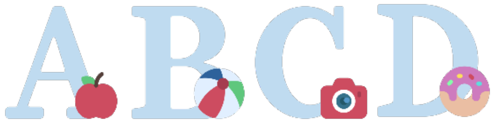
```
BufferedReader reader = new BufferedReader(...);
int numOfLines = reader.lines().count();
```

Returns the lines from the reader as stream

```
BufferedReader reader = new BufferedReader(...);
List<String> unique = reader.lines()
    .flatMap(line -> Stream.of(line.split(“\\s+”)))
    .map(String::trim)
    .map(String::toLowerCase)
    .distinct()
    .sorted((w0, w1) -> w0.length() - w1.length())
    .collect(Collectors.toList())
```

Stream Operations

ABCD map



ABCD filter AC

ABCD reduce