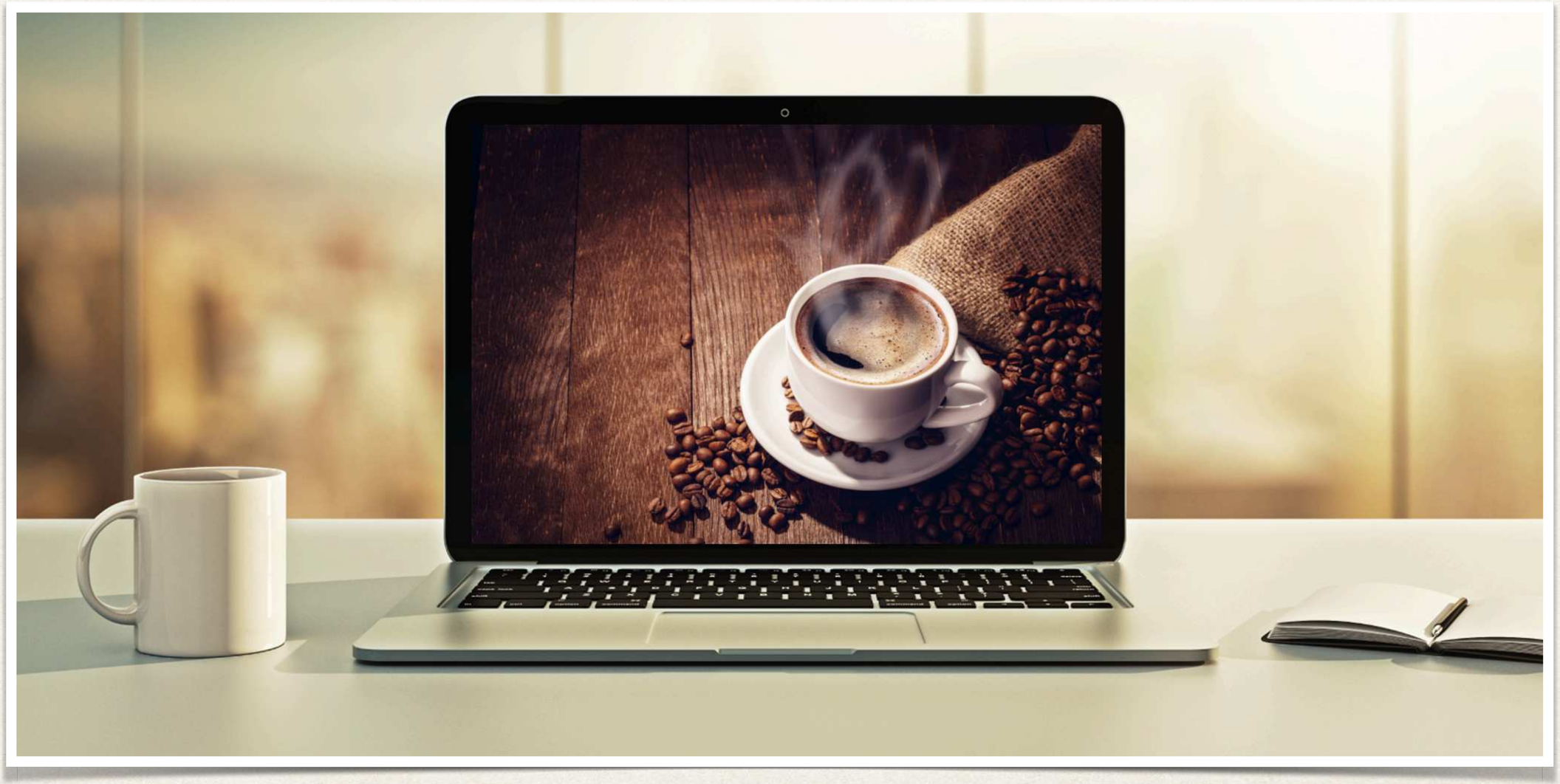
luv2code

Lambda



Lambda

- Allows for expressing instances of a Single-Method Instance
- These “Single-Method Instances” are also known as Functional Instances
- Provides a clear and concise way to represent one method
- Mostly used for iteration, filtering and extracting data

```
(parameters) -> expression
```

Lambda

- Concise Syntax: Functional syntax to express instances
- Functional Programming: This can also facilitate parallel processing
- Stream API: Lambdas work in conjunction with the Stream API (will go over later in course)

```
(parameters) -> expression
```

```
names.ForEach((n) -> System.out.println(n));
```

Wait what…..

Original Loop

List of names:

```
List<String> names = new ArrayList<>();

names.add("Eric");
names.add("Melissa");
names.add("Elijah");
names.add("Milo");
```

Original Loop:

```
for (int i = 0; i < names.size(); i++) {
   System.out.println(names.get(i));
}
```

For Each Loop

List of names:

```
List<String> names = new ArrayList<>();

names.add("Eric");
names.add("Melissa");
names.add("Elijah");
names.add("Milo");
```

For Each Loop:

```
for (String name: names) {
   System.out.println(name);
}
```

Lambda in Action

List of names:

```
List<String> names = new ArrayList<>();

names.add(“Eric”);
names.add(“Melissa”);
names.add(“Elijah”);
names.add(“Milo”);
```

Lambda:

```
names.ForEach((n) -> System.out.println(n));
```

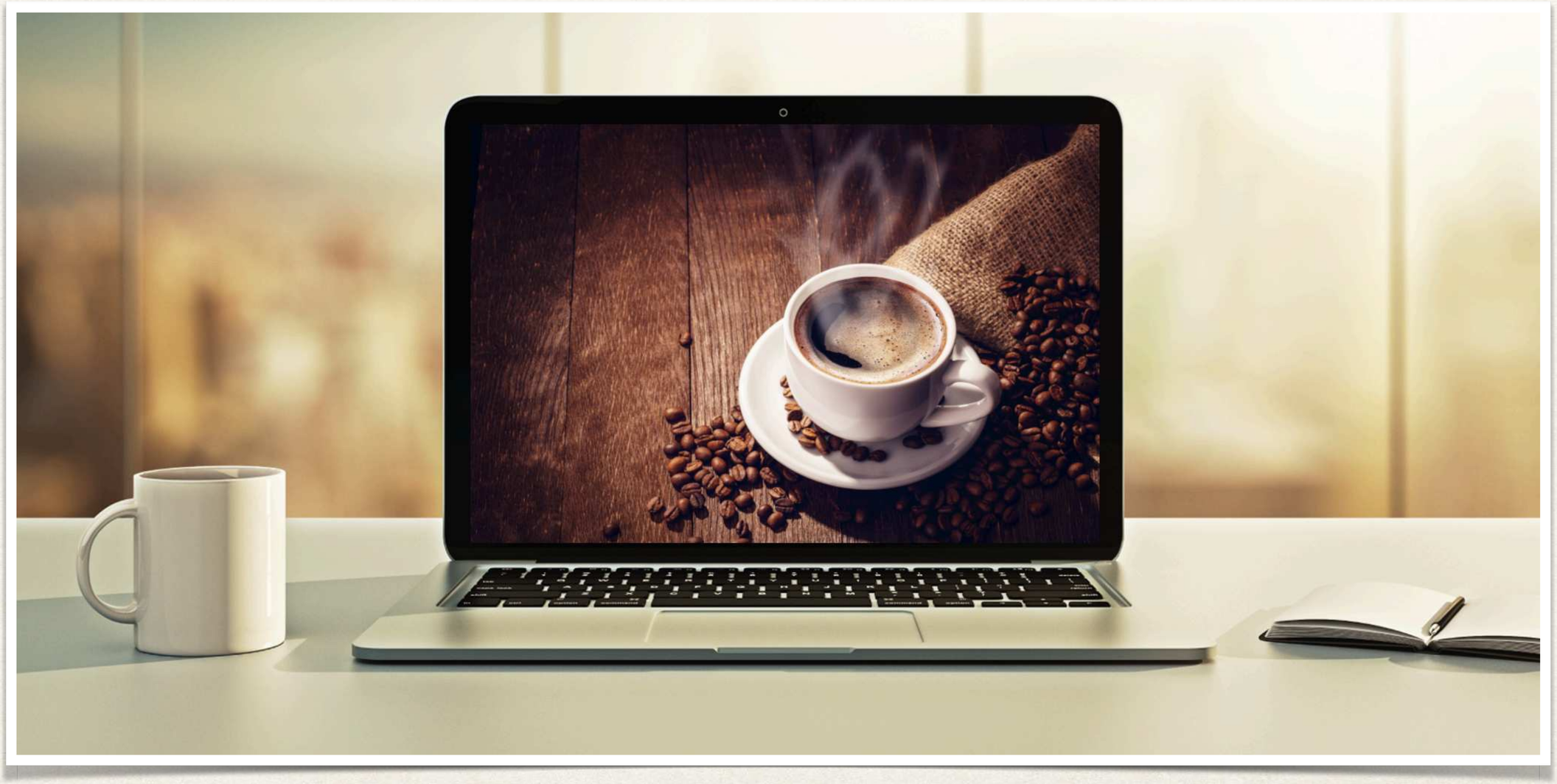
Lambda Limitations

- Can only be used with Functional Interfaces (single abstract method)
- Might be less readable for developers that are unfamiliar with lambda expressions

```
(parameters) -> expression
```

luv2code

Write a Functional Interface



Functional Interfaces

- Normal Java Interface that has only 1 abstract method inside
- We can use a lambda to change the functionality of the method

Let's create one

Functional Interfaces

- Before making a Lambda we need to make a functional interface

```
@FunctionalInterface
public interface Greetings {

    void greetings();
}
```

- Now we can create our lambda

```
Greetings lambda = () -> System.out.println("Hello Java Developers!");
```

```
lambda.greetings(); //Hello Java Developers!
```

Functional Interfaces

- Add functionality to the interface abstract methods

```
@FunctionalInterface
public interface StringEndings {

    String perform(String s);
}
```

- Create and run our new lambda expression

```
StringEndings exclamationMark = (s) -> s + "!";
System.out.println(exclamationMark.perform("Hello"));
```

Functional Interfaces

- Add functionality to the interface abstract methods

```
@FunctionalInterface
public interface StringCompare {

    String perform(String s1, String s2);
}
```

- Now that we have our functional interface we can create the logic for our lambda

Functional Interfaces

```java
@FunctionalInterface
public interface StringCompare {

    String perform(String s1, String s2);
}
```

```java
String a = “Milo”;
String b = “Tester”;

StringCompare value = (s1, s2) -> {
    if (s1.length() > s2.length()) {
        return s1;
}
    return s2;
}

String longerWord = value.perform(a, b);
System.out.println(longerWord); // Tester
```

Predicate

- A functional interface (already created) that returns true or false
- Mainly used for conditional checks
- Has a single abstract method, (test) that takes in a generic type parameter

```
Predicate<Integer> lessThan100 = i -> (i<100);
```

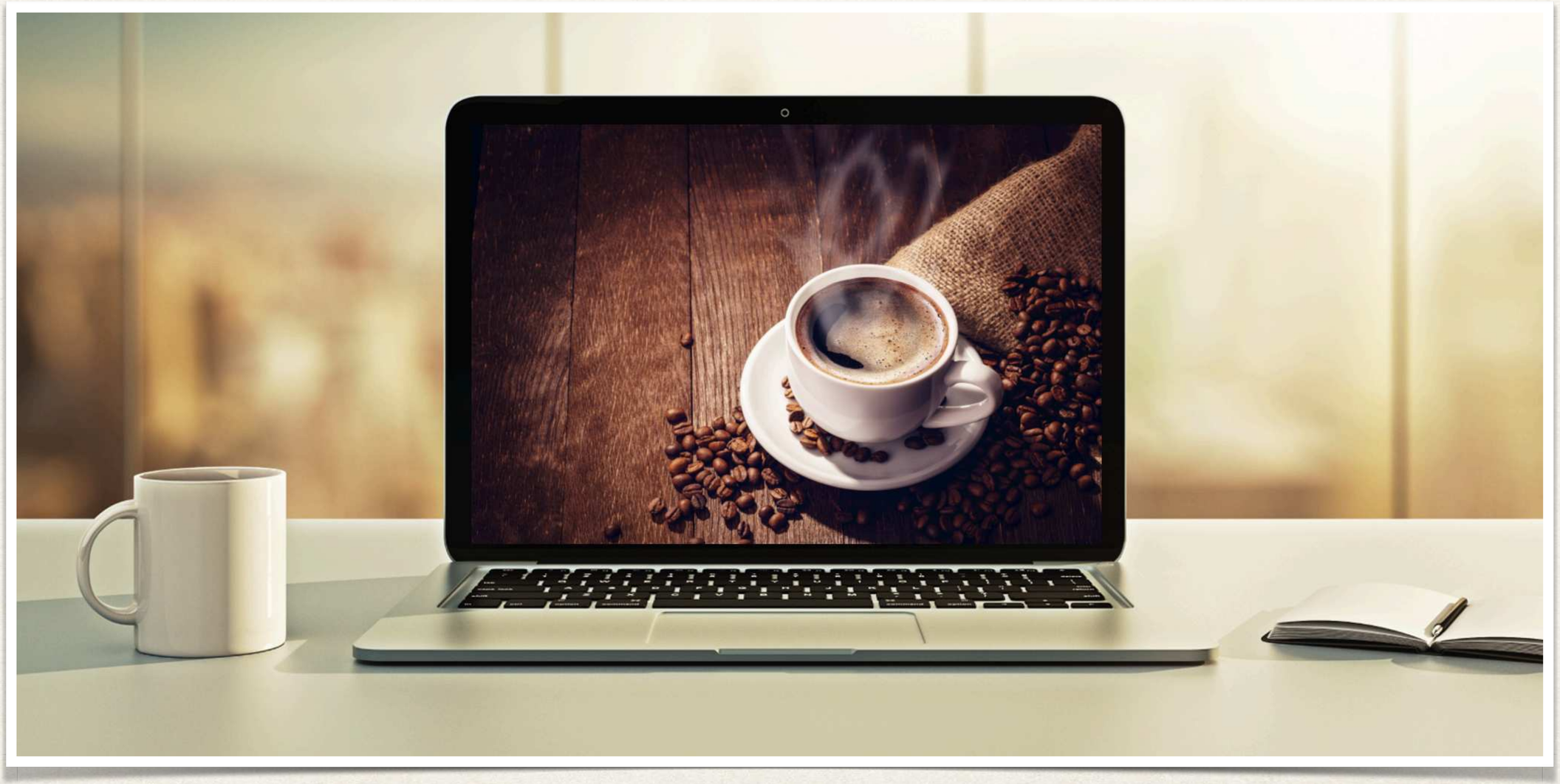
```
boolean result = lessThan100.test(55);
System.out.println(result); //true
```

Predicate

```
Predicate<Integer> lessThan100 = i -> (i<100);

Predicate<Integer> greaterThan50 = i -> (i>50);

// and()
boolean result = lessThan100.and(greaterThan50).test(55);
System.out.println(result); //true

// or()
boolean result = lessThan100.or(greaterThan50).test(3);
System.out.println(result); //true

// negate()
boolean result = lessThan100.negate().test(3);
System.out.println(result); //false
```

luv2code

Lambda Try/Catch



Lambda with Try/Catch

- Exception Handling is used to handle the runtime errors so that the normal flow of an application can remain.
- Exception Handling is an event that occurs during execution of the code.
- This happens when the code / software runs into a condition that it was not expected and does not know how to handle this error

```
(parameters) -> {expression try/catch}
```

Functional Interfaces

- Add functionality to the interface abstract methods

```
@FunctionalInterface
public interface Calculate {

    int perform(int a, int b);
}
```

- Now that we have our functional interface we can create the logic for our lambda try/catch

Functional Interfaces

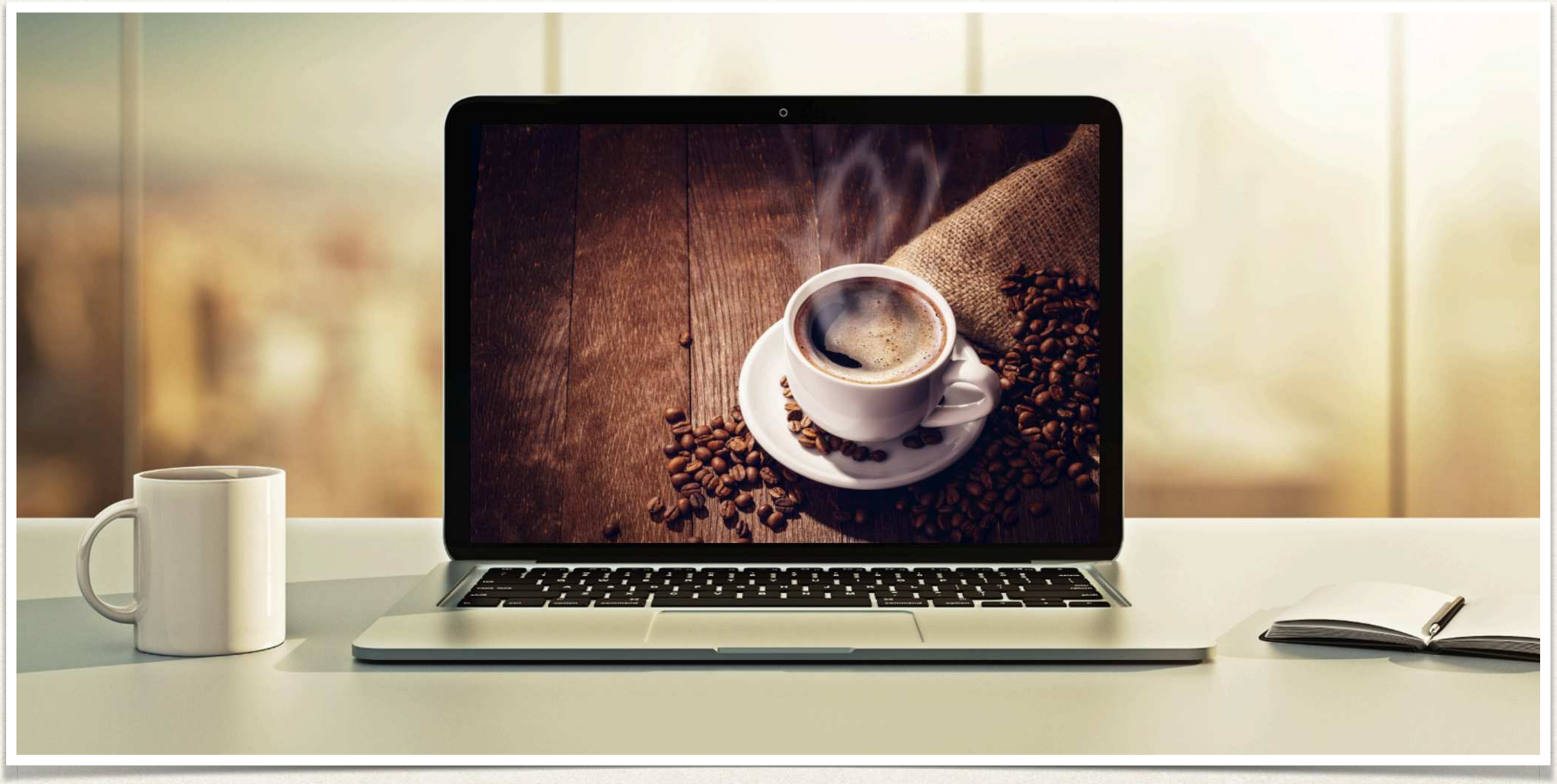
```
@FunctionalInterface
public interface Calculate {

      int perform(int a, int b);
}
```

- Add functionality to the interface abstract methods

```
Calculate divide = (a, b) -> {
  try {
      return a/b;
  } catch(ArithmeticException e) {
      e.printStackTrace();
      return -1
  }

int solution = divide.perform(10, 0);
System.out.println(solution); // Catch Error
```

luv2code

Method Referencing



Method Referencing

- A way to use methods as arguments in higher-order functions.
- Provides clear and concise way to represent methods by name instead of a lambda expression
- Useful when working with functional interfaces

```
Function<String, Integer> methodRef = Integer::parseInt;
```

- Uses the :: (double colon) operator

Function<T, R >

```
Function<T, R> methodRef = Integer::parseInt;
```

- Function is a functional interface from java.util.function
- Takes in a function argument “T” and returns the function result “R”
- Has “apply” method that takes in “T”.

```
Function<String, Integer> methodRef = Integer::parseInt;
methodRef.apply(“5”); // 5
```

Method Referencing

- Here is an example of method referencing:

```
String name = “Eric”;
Predicate<String> methodRef = str::startsWith;
methodRef.test(“E”); // true
```

- Uses the :: (double colon) operator

```
List<String> myList = Arrays.asList(“A”, “B”, “C”);

// Using Lambda
myList.forEach(s -> System.out.println(s));

// Using method reference
myList.forEach(System.out::println);
```

Method Referencing Example

```
public class Vowels {

   public static int countVowels(String s) {

        int count = 0;
        int vowels = “AEIOUaeiou”;

        for (char c : s.toCharArray()) {
            if (vowels.indexOf(c) != -1) {
                count++;
            }
        }
        return count;
   }
```

Method Referencing Example

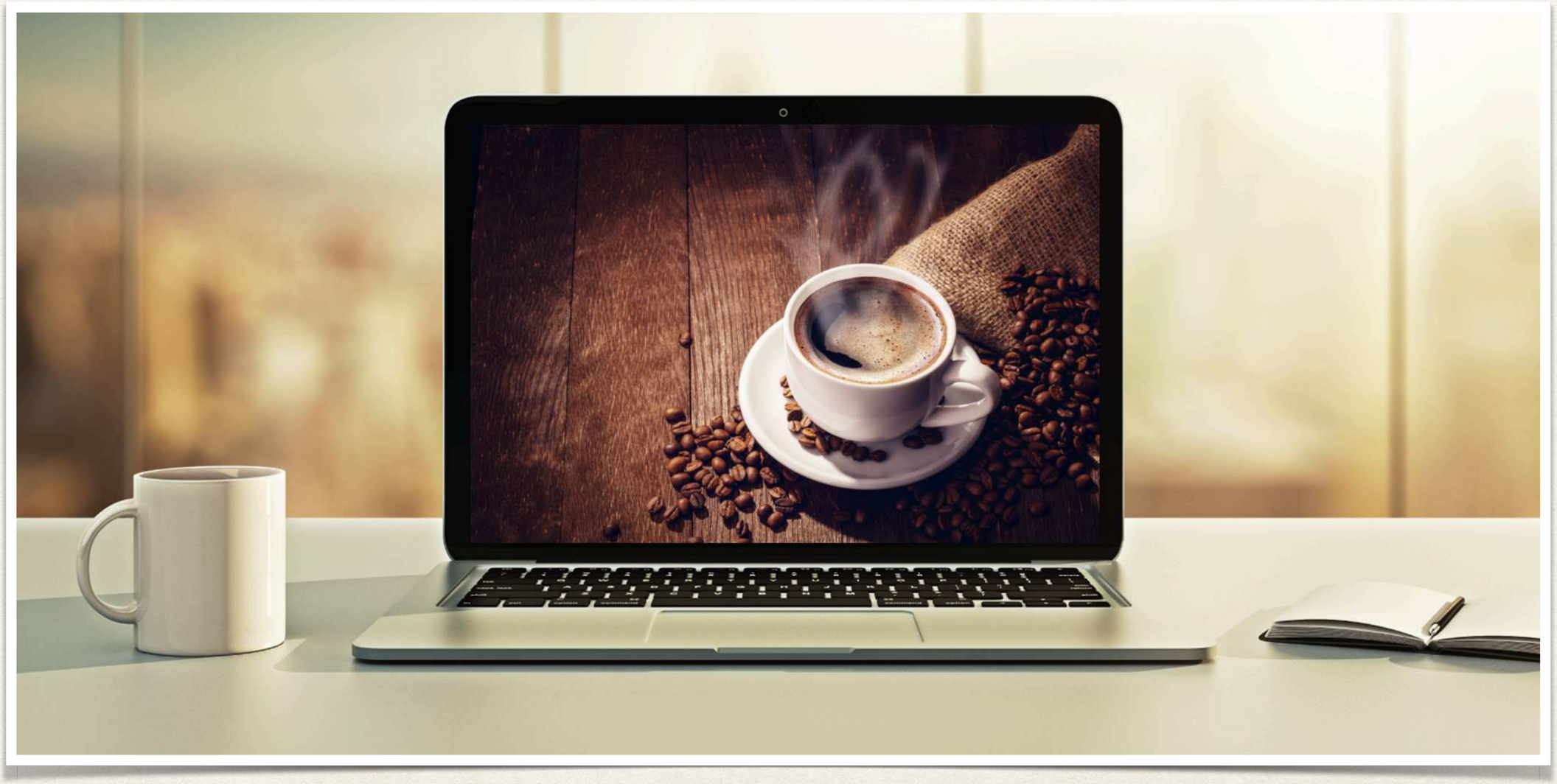
- Provides clear and concise way to represent methods by name instead of a lambda expression

```
public class Vowels {

   public static int countVowels(String s) {

        int count = 0;
        int vowels = "AEIOUaeiou";

        for (char c : s.toCharArray()) {
            if (vowels.indexOf(c) != -1) {
                count++;
            }
        }
        return count;
   }
```

```
public class Main {

   public static int main(String[] args) {
        Function<String, Integer> countVowelsFunction = Vowels::countVowels;
        System.out.println(countVowelsFunction.apply("umbrella")); // 3
   }
```

luv2code

Streams



Streams

- Introduced to provide a new abstraction of data manipulation using a functional approach
- Represent a sequence of elements and support different kinds of operations to process the elements
- The source of elements for a stream come from collections, arrays, Input/Output or generators
- Streams have two types of operations: Intermediate and Terminal

Stream Operations (Intermediate)

- Operations that transform a stream into another stream.
- Lazy: Meaning they do not execute until a result of the processing is actually needed

```
filter(Predicate<T> predicate)
```

- Filters elements based on a condition

```
map(Function<T, R> mapper)
```

- Transforms elements from one form to another

Stream Operations (Terminal)

- Produces a result or side effect
- How most streams end due to needing some type of result

```
forEach(Consumer<T> action)
```

  - Applies an action to each element in the stream

```
collect(Collector)
```

  - Transforms the elements in a stream into a different form, often Lists, Sets or Maps

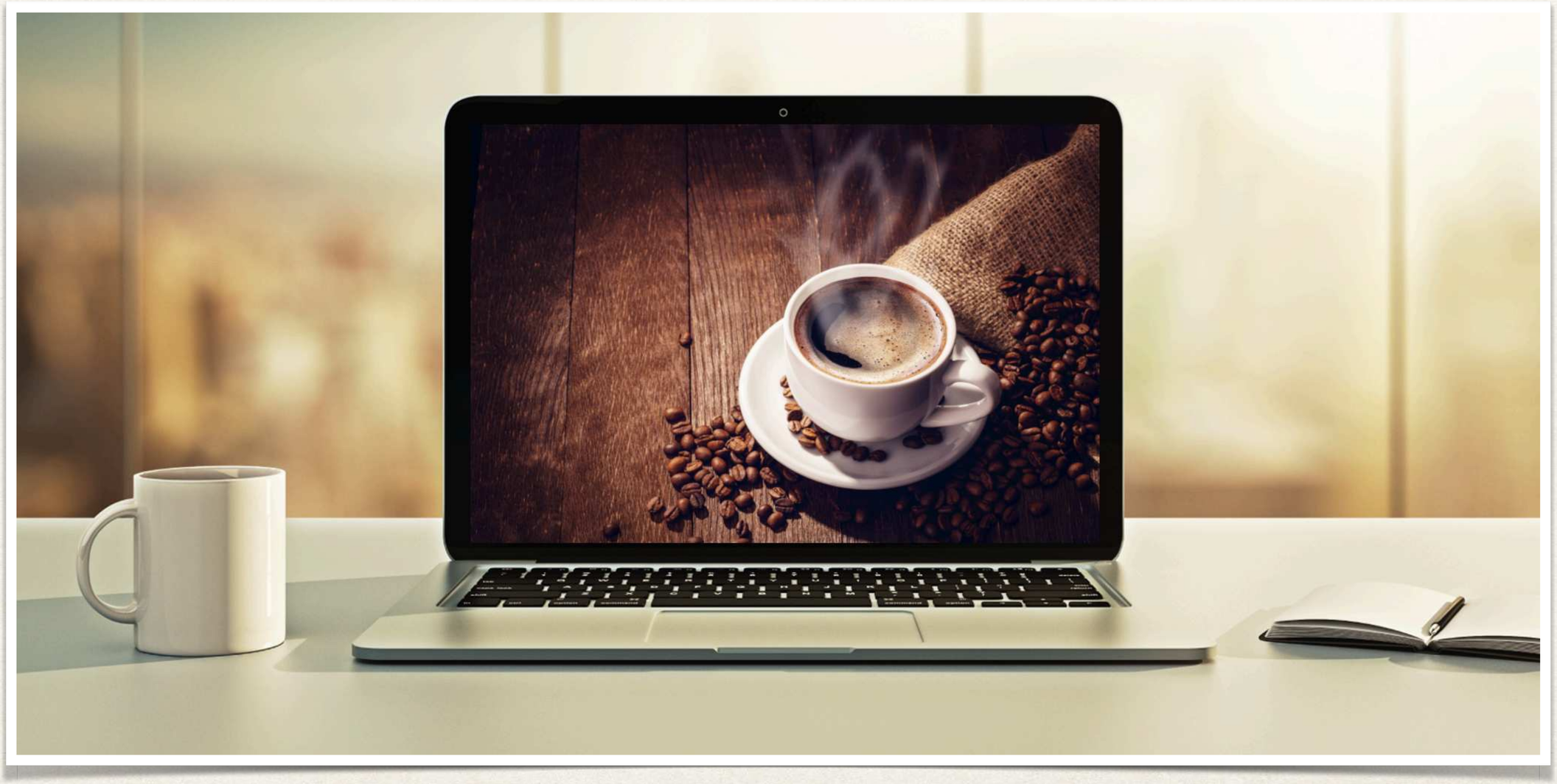
Stream Pipeline

- Combination of source, zero or more intermediate operations and a terminal operation

```
List<String> names = Arrays.asList(“Eric”, “Melissa”, “Elijah”,
“Milo”, “Adil”);

List<String> result = names.stream() // Creating Stream
     .filter(s -> s.length() <= 6) // Filtering by length
     .map(String::toUpperCase) // Converting to uppercase
     .sorted().                    // Sort my letter
     .collect(Collectors.toList()); // Collecting into list

result.forEach(System.out::println); // Printing each element
```

luv2code

Streams with Objects



Streams

- Streams help with what you would need to do loops for
- Helps make the process easier to create and maintain
- Let's go ahead and check out an example of how streams can help with readability of Java Objects

Employee Object

- Employee can have an id, first name and years of service

```
public class Employee {

    private int id;
    private String firstName;
    private int yearsOfService;

    public Employee(int id, String firstName, int yearsOfService) {
           this.id = id;
           this.firstName = firstName;
           this.yearsOfService = yearsOfService;
    }

    ** Getter Methods **
 }
```

Employee Object

```
public class Main {
   public static int main(String[] args) {

      List<Employee> employees = ArrayList<>();

      employees.add(new Employee(1, “Eric”, 8));
      employees.add(new Employee(2, “Milo”, 5));
      employees.add(new Employee(3, “Melissa”, 12));
      employees.add(new Employee(4, “Elijah”, 1));
      employees.add(new Employee(5, “Adil”, 24));
      employees.add(new Employee(6, “Enrique”, 1));
      employees.add(new Employee(7, “Chad”, 18));
   }
}
```

```
public class Employee {

    private int id;
    private String firstName;
    private int yearsOfService;

    public Employee(int id, String firstName, int yearsOfService) {
           this.id = id;
           this.firstName = firstName;
           this.yearsOfService = yearsOfService;
    }

    ** Getter Methods **
}
```

- How many Employee’s have over 10 years of experience?
- Print the name of each employee who has a name that starts with an “E”

Problem 1

- How many Employee's have over 10 years of experience?

Without Streams:

```
int employeeYearsAboveTen = 0;
for (Employee e : employees) {
   if (e.getYearsOfService() > 10) {
      employeeYearsAboveTen++;
   }
}

System.out.println(employeeYearsAboveTen);
```

With Streams:

```
int employeeYearsAboveTen = employees.stream()
   .filter(e -> e.getYearsOfService() > 10)
   .count();

System.out.println(employeeYearsAboveTen);
```

Problem 2

- Print the name of each employee who has a name that starts with an “E”

Without Streams:

```
List<Employee> namesStartWithE = new
   ArrayList<>();

for (Employee e : employees) {
   if (e.getFirstName().startsWith(“E")) {
      namesStartWithE.add(e);
   }
}

for (Employee n : namesStartWithE) {
   System.out.println(n.getFirstName());
}
```

With Streams:

```
List<Employee> namesStartWithE = employees.stream()
   .filter(e -> getFirstName().startsWith(“E"))
   .toList();

namesStartWithE.forEach(n ->
   System.out.println(n.getFirstName()));
```