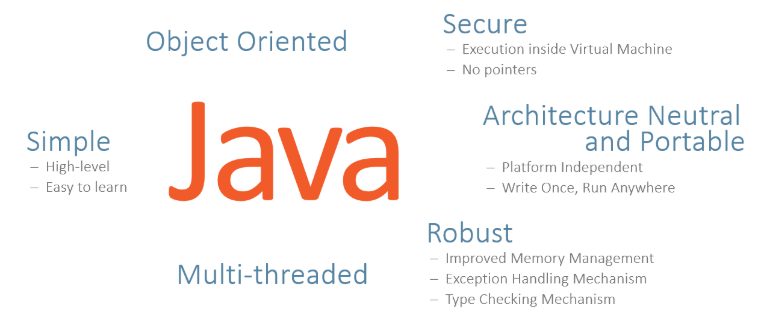
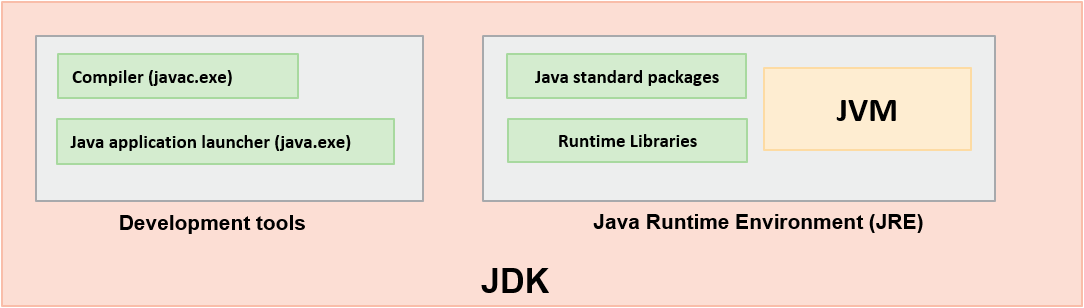
**Java Basics**



* **Java Architecture**

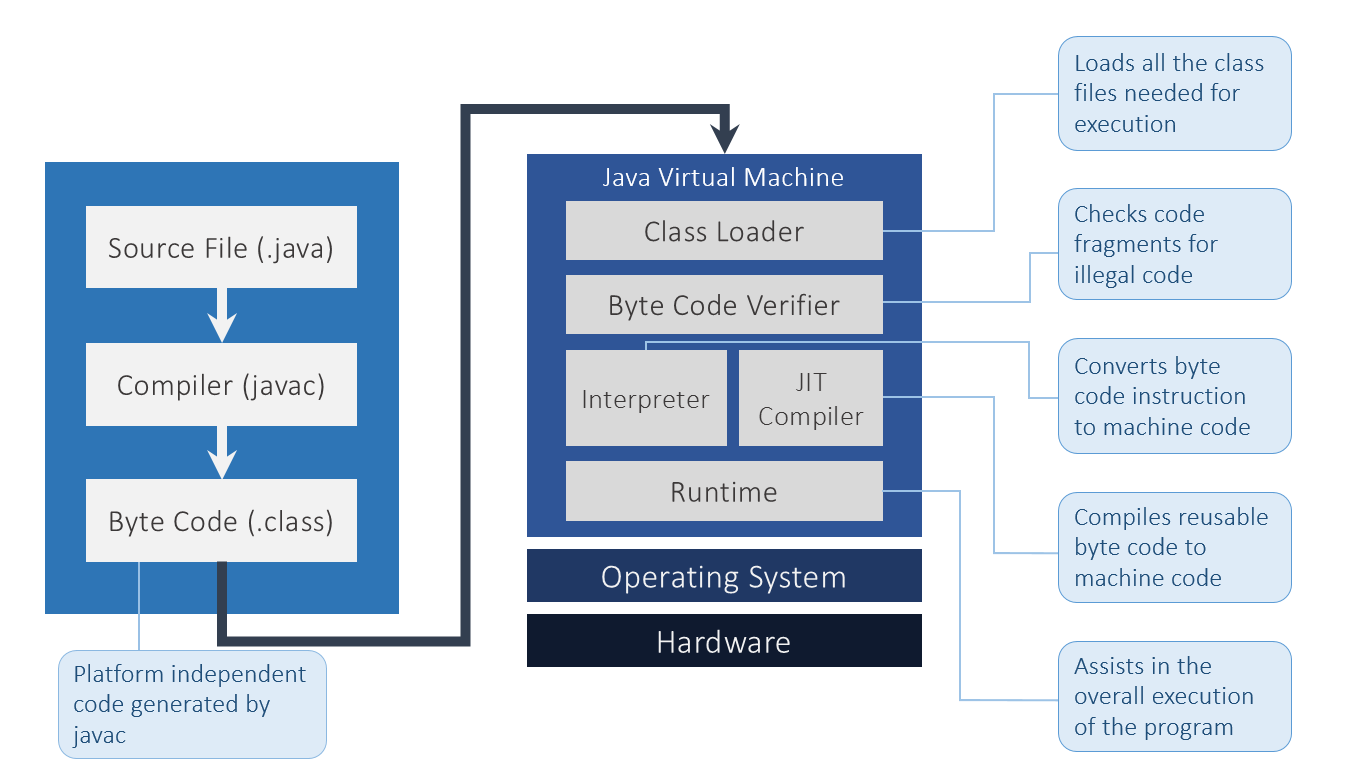


Java SE is actually a Java Development Kit (JDK) that comprises of:

* Development Tools
* Java Runtime Environment (JRE)

Development tools include Java compiler (javac.exe) and Java interpreter (java.exe). These tools are required to prepare a Java application for execution.

Java Virtual Machine (JVM) executes a Java application using certain specific libraries and files provided by JRE. JVM is the actual runtime in which a Java application executes.



* **Class and Object**

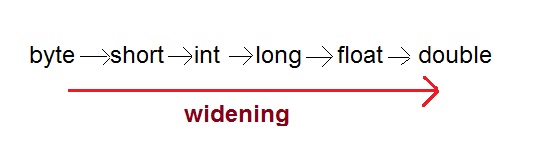
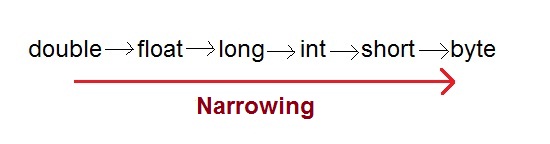
All real-world entities in a problem domain are objects. We define a class to model objects and provide a blueprint to their state and behavior. Through object oriented approach we derive class, state and behavior.

* An instance of a class contains its own copy of variables (attributes) and the methods (behavior) with all the other objects of that class.
* Objects collaborate by exchanging messages with one another via method invocations (i.e., invoking each other's behaviors).
* **Constructors/ getters & setters**

Constructor is a block of code that initializes the newly created object .

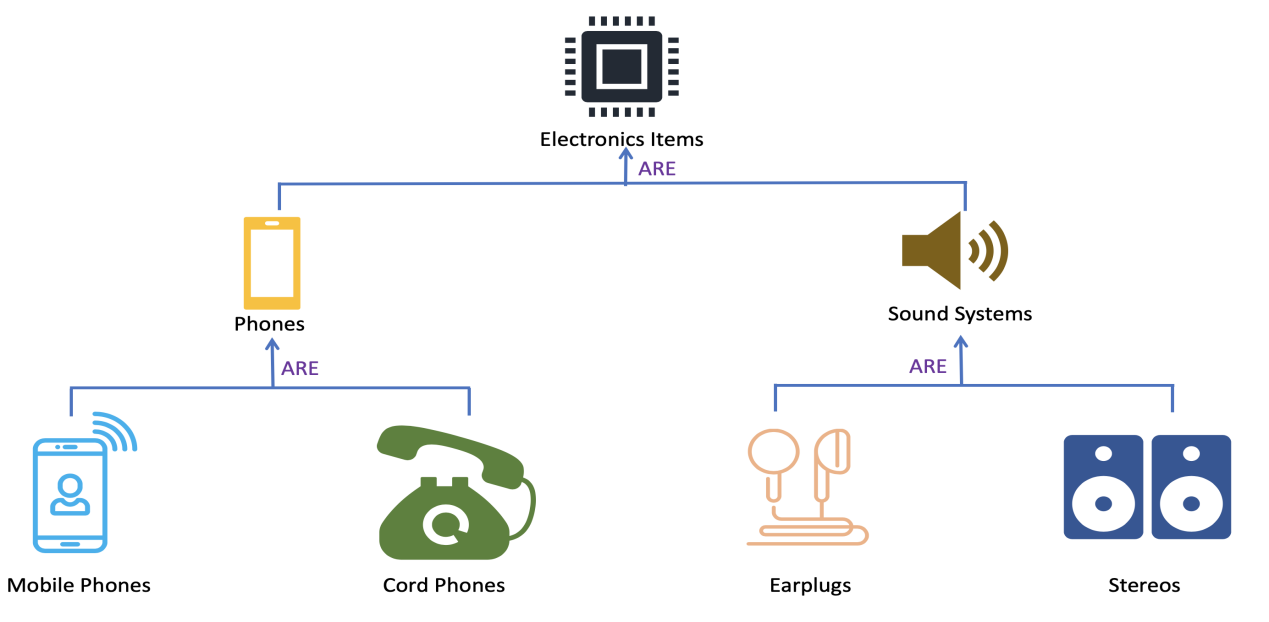
Getter and setter are two methods that are used for retrieving and updating value of a variable. . Getter and setter are also known as accessor and mutator in Java.

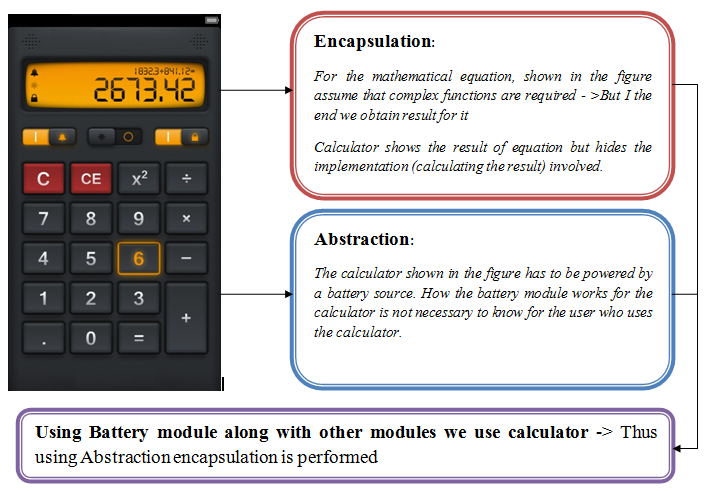
* **Data Types and Type Casting**
* **Primitive data types**: The primitive data types include boolean, char, byte, short, int, long, float and double.
* **Non-primitive data types**: The non-primitive data types include Classes, Interfaces, and Arrays.





* **Arrays** - Example
* **Conditional Statements & Enhanced For Loop** - Example
* **OOPS Concept**
  + Inheritance
  + Polymorphism
  + Abstraction
  + Encapsulation





Data Hiding vs Encapsulation

Abstraction vs Encapsulation

Abstraction vs Data Hiding

Encapsulation = Abstraction + Data Hiding

* **Package and Access modifiers**

 A package is a grouping mechanism in Java which contains all related classes and interfaces.  
The package is a folder that contains all these related classes and interfaces.  
They also restrict access to certain classes and interfaces

|  |  |  |
| --- | --- | --- |
| ****Keyword**** | ****Applicable To**** | ****Who can Access**** |
| private | Data members, methods, and inner class | All members from the same class only |
| (No keyword, usually we call it default) | Data members, methods, classes, and interfaces | All classes from within the same package |
| protected | Data members, methods, and inner class | All classes from within the same package as well as all subclasses i.e. even subclasses residing in a different package |
| public | Data members, methods, classes, and interfaces | Any class |

* **Keywords -** this/super/static/final

Keyword **'THIS'**in Java is a reference variable that refers to the current object.It can be used to refer current class instance variable.

**Super**Keyword in Java. The super keyword in Java is a reference variable which is used to refer immediate parent class object.

The **static variable** can be used to refer to the common property of all objects.The static variable gets memory only once in the class area at the time of class loading.

A **static method** belongs to the class rather than the object of a class.

A static method can be invoked without the need for creating an instance of a class.

The **final** keyword in java is used to restrict the user. The java final keyword can be used in many context. Final can be:

1. variable
2. method
3. Class

* **Interface/Abstract/Inner/Enums**

An **interface** is a reference type in Java. It is similar to class. It is a collection of abstract methods.

A class which is declared with the abstract keyword is known as an **abstract** class in Java.

Java **inner** class or nested class is a class which is declared inside the class or interface

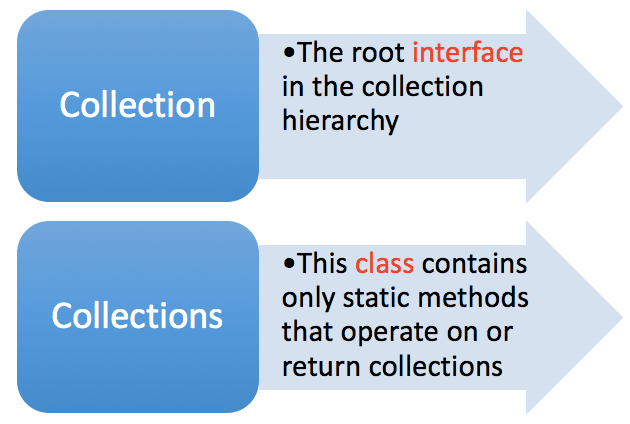
An **enum**is a special "class" that represents a group of constants (unchangeable variables, like final variables).

To create an enum, use the enum keyword (instead of class or interface), and separate the constants with a comma. Note that they should be in uppercase letters:

* **Collections**

The Collection in Java is a framework that provides an architecture to store and manipulate the group of objects.Java Collections can achieve all the operations that you perform on a data such as searching, sorting, insertion, manipulation, and deletion.

The java.util package contains all the [classes](https://www.javatpoint.com/object-and-class-in-java) and [interfaces](https://www.javatpoint.com/interface-in-java) for the Collection framework.





Iterator/Comparable/Comparator

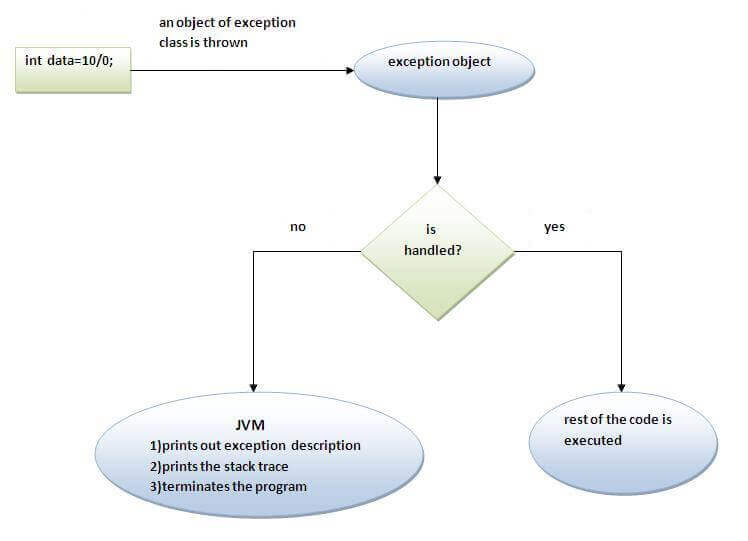
* **Exception Handling**

**Exception** is an abnormal condition.The **Exception Handling** in Java is one of the powerful mechanism to handle the runtime errors so that normal flow of the application can be maintained.

The **java.lang.Throwable** class is the root class of Java Exception hierarchy which is inherited by two subclasses.The Throwable class is the superclass of all errors and exceptions in the Java language. Only objects that are instances of this class (or one of its subclasses) are thrown by the Java Virtual Machine or can be thrown by the Java throw statement.









* **IO**

A stream can be defined as a sequence of data. There are two kinds of Streams

InPutStream − The InputStream is used to read data from a source.

OutPutStream − The OutputStream is used for writing data to a destination.

****

* **Garbage Collection**

**Java 8**

OOP is about encapsulating data in objects behind interfaces and using inheritance to build objects in re-usable pieces.

FP, however, is about not changing states or having side-effects.

****Java Language****

Default and Static Methods in Interfaces

Repeating Annotations

Functional Interfaces

Lambda Expressions

****Java Compiler****

Named Parameters

****Java Libraries****

Optional class

Date/Time API

Stream API

* **Default and Static method**

It is common to add new functionalities to the existing APIs in the real world. But that should happen with out spending much effort.

Prior to Java SE 8, interfaces were expected to have abstract methods only. And, the classes implementing the interfaces had to override all those abstract methods. However, Java SE 8 has made it possible to hold ****method definitions**** in an interface using **default methods** which will make sure avoiding the need for breaking the existing implementations unnecessarily.

The major reason for presenting default methods was to improve the Collections API to have a support for lambda expressions.

****Rule 1****: Classes always win. A method definition in the class or its super class takes precedence over the default method definition that is available in the interface.

****Rule 2****:Otherwise, sub-interfaces win: the method with the exact signature in the most specific default-functionality providing interface will be selected.

****Rule 3****: If the choice continues to be ambiguous, the class that inherits multiple interfaces should explicitly select the default method implementation to be used just by overriding it and the overridden method should have an explicit call to the desired default behavior.

When we have common behaviors for all the implementations of an interface, making them ****static****inside the interface will make them part of the interface itself. No external utility class would be required in such a case.

Just like the **static methods** of a class, the static methods of an interface belong to the interface, and not specific to any instance of its implementing classes.

These methods can only be invoked using the interface name.

* **Lambda Expressions**

Lambda expressions add the essence of functional programming in Java. They are functional constructs without classes, which can be passed like objects and executed as required. They also make the modifiers, return type and parameter types completely optional.

(arguments) -> (body)

* An ****argument list****: Parameter list should be the same (in terms of number, type and order of arguments) as that of the abstract method of the interface.

Argument types can be eliminated, making them inferred types. i.e. (int argument) and (argument) are same.  
 Also, parenthesis () can be eliminated if there is only one argument.

* The ****arrow**** (->) token
* The ****body****: Single statement or a block.  
  Presence of curly braces is not mandatory when the body contains not more than one statement. In addition, the return type of any lambda expression/anonymous function will be the type of the expression that the body evaluates to.

If the body contains a block of statements, curly braces should enclose them and return statement becomes mandatory when the block returns something.

* **Functional Interfaces and Method References**

The target type of any lambda expression is of ****Functional Interfaces****. Functional interfaces strictly have abstract methods of count one. However, they are allowed to have any number of static or default methods.

****@FunctionalInterface**** annotation helps designing an interface as functional interface. Doing so, will make the compiler raise an error if the count of abstract methods in the interface exceeds one.

@FunctionalInterface

interface Shape {

void draw(); // The only abstract method

}

Collections.sort(empList , (employee1, employee2) -> employee1.getCountry().compareTo(employee2.getCountry()));

The compiler will be able to refer the above lambda expression based on the following:

Based on the current context, compiler infers, second parameter should be of type java.util.Comparator Interface. This inference has happened based on the definition of Collections.sort() method.

java.util.Comparator has exactly one abstract method compare() and can be used as the second parameter of sort().

The argument list of lambda expression (employee1, employee2) matches the compare(Object objt1, Object objt2) method present in the Comparator interface.

The return type of lambda's body is int, that exactly matches the compare() method's return type.

The body of the lambda here, throws no checked exception, and hence matches compare() method's complete signature.

* **Repeating Annotations**
* **Optional Class**

Using the ****if**** construct has become the most common way of performing null checks. We know this is the construct most widely used for writing business validations. However, null checks do not directly contribute to business functionality.

The **Optional** class, present in the java.util package, represents a container that may hold null or non-null values.  
It provides a number of methods to perform null check without polluting the code.

The intention was to provide a limited mechanism for library method return types where there needed to be a clear way to represent "no result", and using null for such was overwhelmingly likely to cause errors.

For example, you probably should never use it for something that returns an array of results, or a list of results; instead return an empty array or list. You should almost never use it as a field of something or a method parameter. Routinely using it as a return value for getters would definitely be over-use.

Optional.of(employeeService) // definitely have the service

.map(EmployeeService::getEmployee) // getEmployee() might return null

.map(Employee::getId) // get ID from employee if there is one

.ifPresent(System.out::println); // and if there is an ID, print it

* **Predefined Functional Interfaces**

|  |  |  |  |
| --- | --- | --- | --- |
| ****Type**** | ****Functional Interface**** | ****Abstract Method**** | ****Description**** |
| Function | Function<T, R> | R apply(T t) | Function that accepts single argument and produces result |
| Predicate | Predicate<T> | boolean test(T t) | Boolean-valued function that takes single argument |
| Consumer | Consumer<T> | void accept(T t) | Function that accepts single argument but returns no result |
| Supplier | Supplier<T> | T get() | Function that denotes a supplier of results |

* **Traversing Collections**

****How**** - creating the iterator, looping through it, and checking for more elements

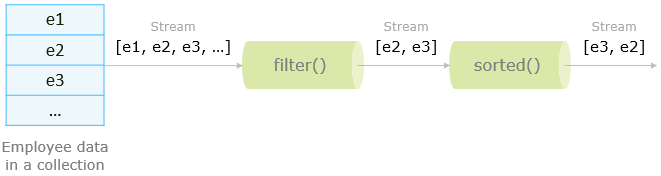
****What**** - the business functionality

* **Streams**

A Stream denotes the flow of a ****group of elements in sequence****from a specific ****source****, and supports different ****data processing operations****.

The group of elements in sequence belongs to a specific type, and can have sources like collections, I/O resources or arrays.

The data processing operations like filter, map, sort, count, etc. can be easily used to manipulate the data in a stream.



Since most of the stream operations return a stream back, they can be pipelined in order to make the code clear and concise. Stream operations can be pipelined without having to maintain intermediate results.

Different Operations On Streams-

**Intermediate Operations:**

1. **map:**The map method is used to returns a stream consisting of the results of applying the given function to the elements of this stream.

List number = Arrays.asList(2,3,4,5);  
List square = number.stream().map(x->x\*x).collect(Collectors.toList());

1. **filter:** The filter method is used to select elements as per the Predicate passed as argument.

List names = Arrays.asList("Reflection","Collection","Stream");  
List result = names.stream().filter(s->s.startsWith("S")).collect(Collectors.toList());

1. **sorted:** The sorted method is used to sort the stream.

List names = Arrays.asList("Reflection","Collection","Stream");  
List result = names.stream().sorted().collect(Collectors.toList());

**Terminal Operations:**

1. **collect:** The collect method is used to return the result of the intermediate operations performed on the stream.

List number = Arrays.asList(2,3,4,5,3);  
Set square = number.stream().map(x->x\*x).collect(Collectors.toSet());

1. **forEach:** The forEach method is used to iterate through every element of the stream.

List number = Arrays.asList(2,3,4,5);  
number.stream().map(x->x\*x).forEach(y->System.out.println(y));

1. **reduce:** The reduce method is used to reduce the elements of a stream to a single value.  
   The reduce method takes a BinaryOperator as a parameter.

# List number = Arrays.asList(2,3,4,5); int even = number.stream().filter(x->x%2==0).reduce(0,(ans,i)-> ans+i);

# Here ans variable is assigned 0 as the initial value and i is added to it .

**Java 11**

* **var keyword in Java**

**var**keyword is added to the Java language since Java 10 (JDK 10), supporting a new feature called local variable type inference, in which Java compiler guesses the type of the variables based on the surround context – allowing programmers to not declare the type explicitly.

List<String> list = **new** ArrayList<String>();

var list = **new** ArrayList<String>();

**public** Map<String, List<Student>> foo() {

    // return a map

}

Map<String, List<Student>> map = foo(); // or

var map = foo();

//From Java 11

Predicate<String> predicate = (@Nullable var a) -> true;

* **HTTP Client**

The new HttpClient can be used either synchronously or asynchronously. A synchronous request blocks the current thread until the reponse is available. BodyHandlers define the expected type of response body (e.g. as string, byte-array or file). Similiar to BodyHandlers you use BodyPublishers to define the type of data you want to send as body of the request such as strings, byte-arrays, files or input-streams.

//GET

var request = HttpRequest.newBuilder()

.uri(URI.create("https://winterbe.com"))

.GET()

.build();var client = HttpClient.newHttpClient();

HttpResponse<String> response =

client.send(request, HttpResponse.BodyHandlers.ofString());

System.out.println(response.body());

//POST

var request = HttpRequest.newBuilder()

.uri(URI.create("https://postman-echo.com/post"))

.header("Content-Type", "text/plain")

.POST(HttpRequest.BodyPublishers.ofString("Hi there!"))

.build();

var client = HttpClient.newHttpClient();

var response = client.send(request, HttpResponse.BodyHandlers.ofString());

System.out.println(response.statusCode()); // 200

* **Streams**

Streams were introduced in Java 8 and now receive three new methods. Stream.ofNullable constructs a stream from a single element.

Stream.ofNullable(null).count() // 0

The methods dropWhile and takeWhile both accept a predicate to determine which elements to abandon from the stream.

Stream.of(1, 2, 3, 2, 1)

.dropWhile(n -> n < 3)

.collect(Collectors.toList()); // [3, 2, 1]

Stream.of(1, 2, 3, 2, 1)

.takeWhile(n -> n < 3)

.collect(Collectors.toList()); // [1, 2]

* **Strings**

One of the most basic classes String gets a few helper methods for trimming or checking whitespace and for streaming the lines of a string.

" ".isBlank(); // true

" Foo Bar ".strip(); // "Foo Bar"

“ Foo Bar ".stripTrailing(); // " Foo Bar"

“ Foo Bar ".stripLeading(); // "Foo Bar "

"Java".repeat(3); // "JavaJavaJava"

"A\nB\nC".lines().count(); // 3