Coding Bootcamp

Core Java

TECHNOLOGY

Java 11 Features

Learning Objectives

By the end of this lesson, you will be able to:

- Utilize the various methods available in the String API and the File API
- Convert a collection into an array using Java 11
- Apply Predicate.not() method to refines predicate operations
- Implement var lambda in Java 11 and understand its limitations
- Use nest-based access and the new methods available for the Reflection API

TECHNOLOGY

HttpClient

Java 11

Java 11 is the first long-term support feature release of the Java programming language after Java 8.

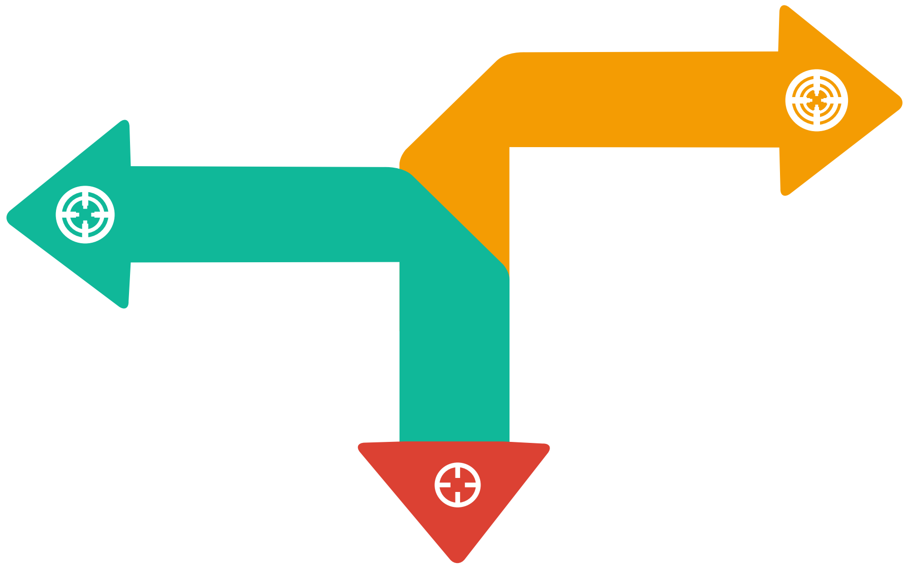
Introduced in 2018
and provides various features

What Is HttpClient?

It is a class released in Java 9 in the incubator module. A new version is introduced in Java 11 and moved to the Java.net.http package.

- It can be built with a builder.
- It is an immutable object used to send multiple requests.
- A bodyHandle needs to be present for every httpRequest.

HttpClient



Used for sending requests and retrieving their responses

Determines how to handle the response received from the HTTP request

Helps to send the HTTP request without any third-party APIs

HttpClient

Syntax to create an HttpClient object:

```
HttpClient client =
HttpClient.newHttpClient();
```

HttpClient

Add properties of the behavior of HttpClient:

```
HttpClient client =
HttpClient.newBuilder()
        .version(Version.xx)

.connectTimeout(Duration.ofSeconds(55))
        .build();
```

HttpClient

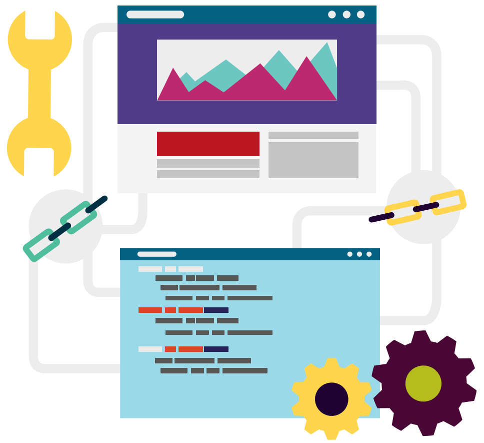
There are three methods to send synchronous and asynchronous requests:

send(HttpRequest, BodyHandler)

sendAsync(HttpRequest, BodyHandler)

Sending Concurrent Request

Blocks request until it has been sent and the response has been received



HttpClient

The send() method takes two parameters: the request and the BodyHandler.

```
HttpRequest request =
HttpRequest.newBuilder(URI.create("http://abc.com/pos
t"))
      .header("Content-Type", "application/json")
      .POST(HttpRequest.BodyPublishers.ofString("{\n"
          + "  \"json\":null,\n"
          + "  \"origin\":\"112.196.145.x\",\n"
          + "  \"url\":\"http://abc.com/patch\"\n"
          + "}"))
    .build();
  client.send(request,
HttpResponse.BodyHandlers.ofString());
```

Note:

There is a request using the HttpRequest call in which the header is specified.

POST specify is a post request and parameter that specifies the body for the post request.

HttpClient

The map operation is applied, and the async HttpRequest is sent.

```
private static List<CompletableFuture<String>>
concurrentCalls(final List<URI> urlList) {

        return urlList.stream()
                .map(url -> client.sendAsync(
                        HttpRequest.newBuilder(url)
                                .GET().build(),

HttpResponse.BodyHandlers.ofString())
                        .thenApply(response ->
response.body()))
                .collect(Collectors.toList());
    }
```

HttpClient

send(HttpRequest, BodyHandler)

sendAsync(HttpRequest, BodyHandler)

Sending Concurrent Request

It helps as a non-blocking call. It sends the request and receives the response asynchronously.

HttpClient

sendAsync will return the CompletableFuture<HttpResponse<String>>.

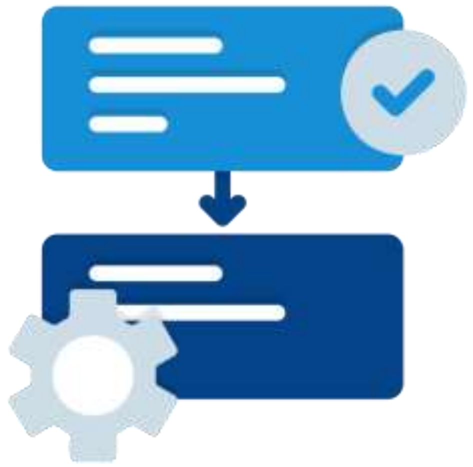
```
HttpRequest request =
HttpRequest.newBuilder(URI.create("http://abc.com/pos
t"))
      .header("Content-Type", "application/json")
      .POST(HttpRequest.BodyPublishers.ofString("{\n"
          + "  \"json\":null,\n"
          + "  \"origin\":\"112.196.145.x\",\n"
          + "  \"url\":\"abc.com/patch\"\n"
          + "}"))
    .build();
 CompletableFuture<String> stringCompletableFuture =
client
        .sendAsync(request,
HttpResponse.BodyHandlers.ofString());
```

HttpClient

send(HttpRequest, BodyHandler)

sendAsync(HttpRequest, BodyHandler)

Sending Concurrent Request



Shows how to send concurrent requests

Utilizing File Methods in Java

Problem Statement:

You have been asked to develop a Java 11 program that effectively utilizes file methods for efficient file handling and manipulation.

Outcome:

By developing a Java 11 program that utilizes file methods, you will learn efficient techniques for handling and manipulating files. This will enhance your ability to perform tasks such as reading, writing, and managing files effectively within your applications.

Note: Refer to the demo document for detailed steps: 01_Utlizing_File_Methods_in_Java

Assisted Practice: Guidelines

Steps to be followed are:

1. Open the Eclipse IDE and create a new Java project
2. Create a temporary file in a directory
3. Use the try and the catch block
4. Create one more path for the file in the temporary directory path
5. Write the string content
6. Use this static method to write the string and execute the code directly

Implementing HTTP Client in Java

Problem Statement:

You have been asked to develop a Java 11 application using the HTTP Client API.

Outcome:

By developing a Java 11 application using the HTTP Client API, you will learn to send HTTP requests and handle responses efficiently. This skill is essential for building applications that interact with web services and APIs.

Note: Refer to the demo document for detailed steps: 02_Implementing_HTTP_Client_in_Java

Assisted Practice: Guidelines

Steps to be followed are:

1. Open the Eclipse IDE and create a new Java project
2. Open the web browser and search for newsapi.org
3. Login with your account to get an API key
4. Hit the URL to get the data in the form of a response
5. Record and concatenate the API key as a separate variable in Eclipse IDE
6. Create the HTTP client object, HTTP request object, and then the response objects
7. Execute the code

TECHNOLOGY

String APIs

simplilearn

String APIs

The String class in Java provides a set of methods, known as the String API, for manipulating and operating on String objects.

String API Methods

- **repeat(int)**
- **isBlank()**
- **strip()**
- **stripLeading()**
- **stripTrailing()**

String APIs

The **repeat(int)** method repeats a string a given number of times.

Example:

```
import Java.util.ArrayList;
import Java.util.List;

public class Example {
   public static void main(String[] args) {
      String text = " Hello ";
      System.out.println(text.repeat(3)); // "
hello  text. Repeat  hello ‟
        }
}
```

String APIs

The **isBlank()** method checks if a string is empty or has any spaces.

Example:

```
import Java.util.ArrayList;
import Java.util.List;

public class Example {
   public static void main(String[] args) {
      String text = " Hello ";
      System.out.println(text.isBlank()); //
false
      System.out.println("".isBlank()); //
true
        } }
```

String APIs

The **strip()** method removes the trailing and leading whitespaces.

Example:

```
import Java.util.ArrayList;
import Java.util.List;

public class Example {
   public static void main(String[] args) {
      String sample = " hello ";
      System.out.println(sample.strip()); //
"hello"
        }
}
```

String APIs

The **stripLeading()** method removes the leading whitespaces.

Example:

```
import Java.util.ArrayList;
import Java.util.List;

public class Example {
   public static void main(String[] args) {
      String sample = " hello ";

System.out.println(sample.stripLeading()); //
"hello "
        }
}
```

String APIs

The **stripTrailing()** method removes the trailing whitespaces.

Example:

```
import Java.util.ArrayList;
import Java.util.List;

public class Example {
   public static void main(String[] args) {
      String sample = " hello ";

System.out.println(sample.stripTrailing()); //
" hello”
        }
}
```

TECHNOLOGY

Collections to Array

Collections to Array

Java 11 gives an easy way to convert a collection into an array.

Syntax:

```
arrayName =
listName.toArray(String[]::new);
```

Collections to Array

Example:

```
public class Example {
   public static void main(String[] args) {

      List<String> list =
Arrays.asList("John", "Julie");
      // Old way
      String[] names = list.toArray(new
String[list.size()]);
      System.out.println(names.length);
      // New way
      names = list.toArray(String[]::new);
      System.out.println(names.length);
   }
}
```

Converting Collections to Array Java

Problem Statement:

You have been asked to implement a Java program that converts collections to arrays efficiently.

Outcome:

By implementing a Java program that converts collections to arrays efficiently, you will learn techniques for transforming data structures, enhancing your ability to manage and manipulate data within your applications.

Note: Refer to the demo document for detailed steps: 03_Converting_Collections_to_Array_in_Java

Assisted Practice: Guidelines

Steps to be followed are:

1. Open the Eclipse IDE and create a new Java project
2. Write a few emails as a new array list
3. Convert the array list back to an array
4. Print emails and the data coming in
5. Create an array of objects

TECHNOLOGY

File APIs

File APIs

Java 11 facilitates easy reading and writing of files by using the overload method.

Two new static methods in Java.nio.file are added in Java 11:

of(String, String[])

Returns a path by converting a path string

of(net.URL)

Returns a URI by converting a URL

File APIs

Example to implement APIs:

```
import Java.io.File;
import Java.io.IOException;
import Java.net.URI;
import Java.nio.file.Files;
import Java.nio.file.Path;

public class Example {
    public static void main(String[] args) throws IOException {
        String folderTemp = System.getProperty("Java.io.tmpdir");
        // Create Path from a sequence of Strings
        Path path1 = Path.of(folderTemp, "hello.txt");
        System.out.println(path1);
        System.out.println(Files.exists(path1));
        File Files. Exists = new File(path1.toString());
        //Create the file
        if (file.createNewFile()) {
            System.out.println("File is created!");
        } else {
            System.out.println("File already exists.");
        }
```

File APIs

```
String uriPath = "file:///" + folderTemp.replace("\\", "/") +
"hello.txt";
        URI uri = URI.create(uriPath);
        System.out.println(uri);
        // Create Path from a URI
        Path path2 = Path.of(uri);
        System.out.println(path2);
        System.out.println(Files.exists(path2));
    }
}
```

File APIs

The **Java.nio.file.Files** class has been enhanced with the addition of four new methods that make it easier to read strings from files and write strings to files directly. These new methods comprise:

readString(Path)

This reads all the content from a file into a string, decoding from bytes to characters using the UTF-8 charset.

readString(Path, Charset)

This method also reads all the content from a file into a string and decodes the bytes into characters with the help of Charset.

writeString(Path, CharSequence, Java.nio.file. OpenOption[])

This method writes a CharSequence to a file. Characters are encoded into bytes with UTF-8 charSet.

writeString(Path, CharSequence, Java.nio.file. Charset, OpenOption[])

This method also writes a CharSequence to a file, and Characters are encoded into bytes with the help of Charset.

File APIs

Example for writeString:

```
import Java.io.File;
import Java.io.IOException;
import Java.nio.charset.Charset;
import Java.nio.file.Files;
import Java.nio.file.Path;
public class Example {
    static Path createTempPath() throws IOException {
        Path pathTemp = Files.createTempFile("hello", ".txt");
        File fileTemp = pathTemp.toFile();
        fileTemp.deleteOnExit();

        return pathTemp;
    }
    public static void main(String[] args) throws IOException {
        String s = "New Iphone has a price of $1500 or about
$1600";
```

File APIs

```
Path path1 = Files.writeString(createTempPath(), s);
        System.out.println(path1);
        System.out.println(Files.readString(path1));
        Charset Files. ReadString = Charset.forName("ISO-8859-3");
        Path path2 = Files.writeString(createTempPath(), s,
charset);
        System.out.println(path2);
        String Files. ReadString = Files.readString(path2,
charset);
        System.out.println(string);
    }
}
```

Not Predicate

Not Predicate

Java 11 added a new static method predicate.not().

The predicate class is available in the Java.util.function package.

Syntax:

```
Negate =
Predicate.not(positivePredicate);
```

Here, a positivePredicate is a predicate whose negation is required, and the return value of this method is a predicate.

Not Predicate

Create one predicate and initialize the required conditions to it

```
import Java.util.Arrays;
import Java.util.List;
import Java.util.function.Predicate;
import Java.util.stream.Collectors;

public class Example {
    public static void main(String[] args)
    {
        List<Integer> numbers
            = Arrays.asList(1, 2, 3, 4, 5, 6, 7, 8, 9, 10);
        // creating a predicate for negation
        Predicate<Integer> even = i -> i % 2 == 0;
        // creating a predicate object which
        // is negation os supplied predicate
        Predicate<Integer> odd = Predicate.not(even);
        // filtering the even number using even predicate
        List<Integer> evenNumbers
            = numbers.stream().filter(even).collect(
                Collectors.toList());
```

Not Predicate

```
// filtering the odd number using odd predicate
        List<Integer> oddNumbers
            = numbers.stream().filter(odd).collect(
                Collectors.toList());
        // Print the Lists
        System.out.println(evenNumbers);
        System.out.println(oddNumbers);
    }
}
```

TECHNOLOGY

Var in Lambda

Var in Lambda

Java 11 permits the use of var in a lambda expression, and can be used for applying modifiers to the local variables.

Syntax:

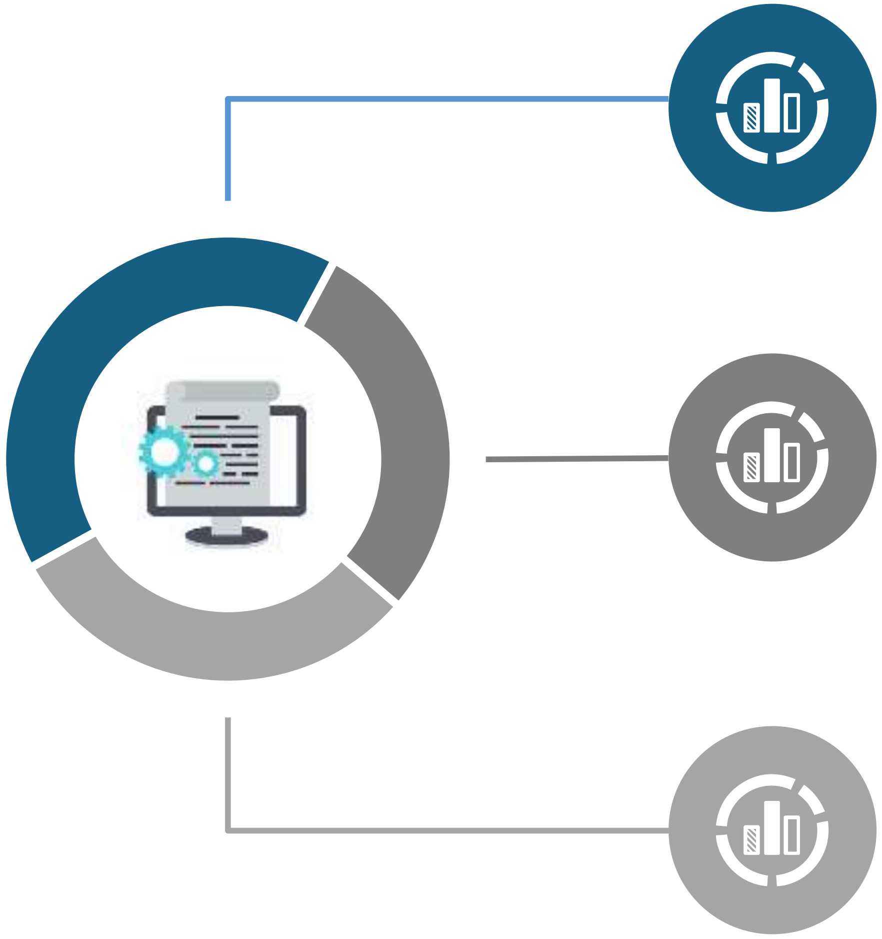
```
(@NonNull var value1, @Nullable var value2) ->
value1 + value2
```

Var in Lambda

Example:

```
import Java.util.Arrays;
import Java.util.List;
import Java.util.stream.Collectors;

@interface NonNull {}

public class Example {
   public static void main(String[] args) {
      List<String> list = Arrays.asList("hello", "world");

      String list2 = list.stream()
         .map((@NonNull var listItem) -> listItem.toUpperCase())
         .collect(Collectors.joining(", "));

      System.out.println(list2);
   }
}
```

Var in Lambda: Limitations



Cannot be combined with other parameters
(var v1, v2) -> v1 + v2

Can only be used within the parentheses
var v1 -> v1.toLowerCase()

Cannot be combined with other typed parameters
(var v1, String v2) -> v1 + v2

Implementing Lambda and Local Var in Java

Problem Statement:

You have been asked to design a Java 11 program that leverages lambda expressions and local variable type inference.

Outcome:

By designing a Java 11 program that leverages lambda expressions and local variable type inference, you will learn to write concise and readable code, enhancing the flexibility and efficiency of your programming practices.

Note: Refer to the demo document for detailed steps: 04_Implementing_Lambda_and_Local_Var_in_Java

Assisted Practice: Guidelines

Steps to be followed are:

1. Open the Eclipse IDE and create a new Java project
2. Create a list which goes as the list of type string and then the emails as arrays dot as a list and executing the code
3. Create the data as comma separated values and execute the code
4. Write the variable or the var keywords, inside the lambdas

Utilizing Lambda Expressions in Java

Problem Statement:

You have been asked to create a program that utilizes lambda expressions in Java.

Outcome:

By creating a program that utilizes lambda expressions in Java, you will learn to write more concise and readable code, making your applications more efficient and easier to maintain.

Note: Refer to the demo document for detailed steps: 05_Utilizing_Lambda_Expressions_in_Java

Assisted Practice: Guidelines

Steps to be followed are:

1. Open the Eclipse IDE and create a new Java project
2. Create a functional interface
3. Implement the interface
4. Use anonymous class for running an interface
5. Create a function, which is the implementation for the Lambda expression
6. Write another interface as login and executing the code with sample data
7. Execute the Lambda expression with example data

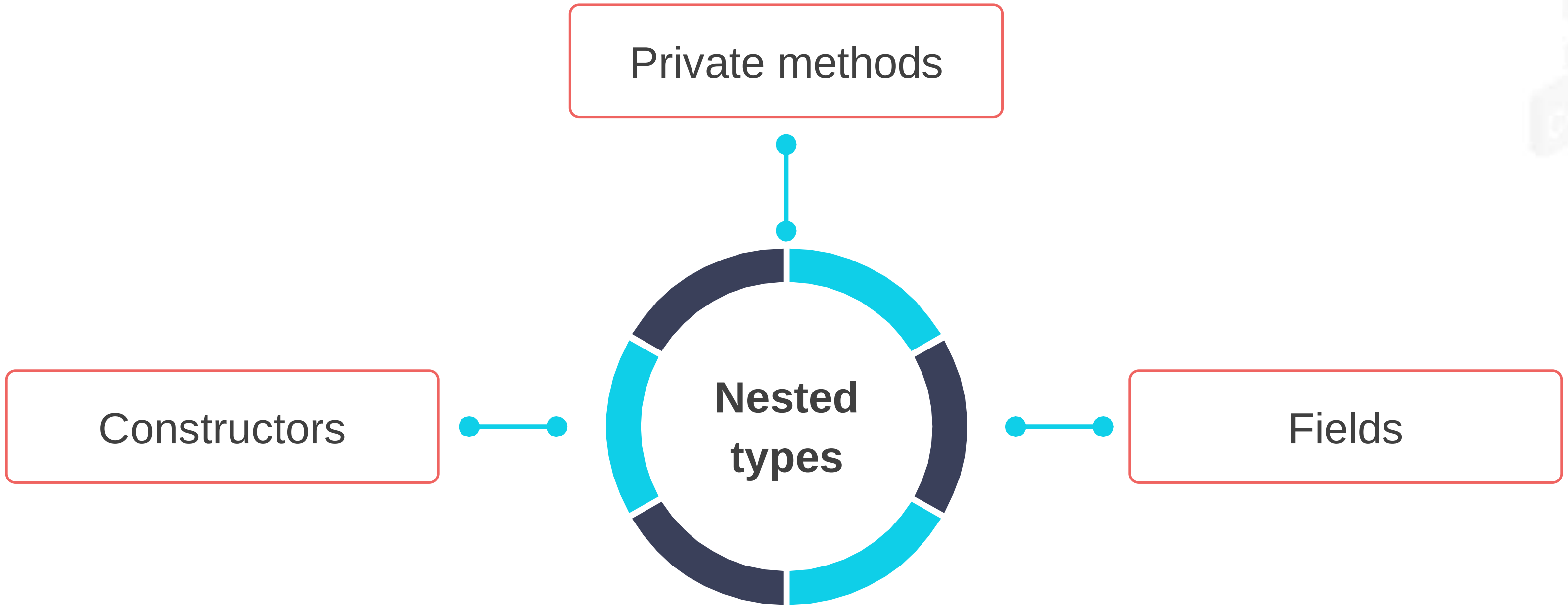
TECHNOLOGY

Nest-Based Access

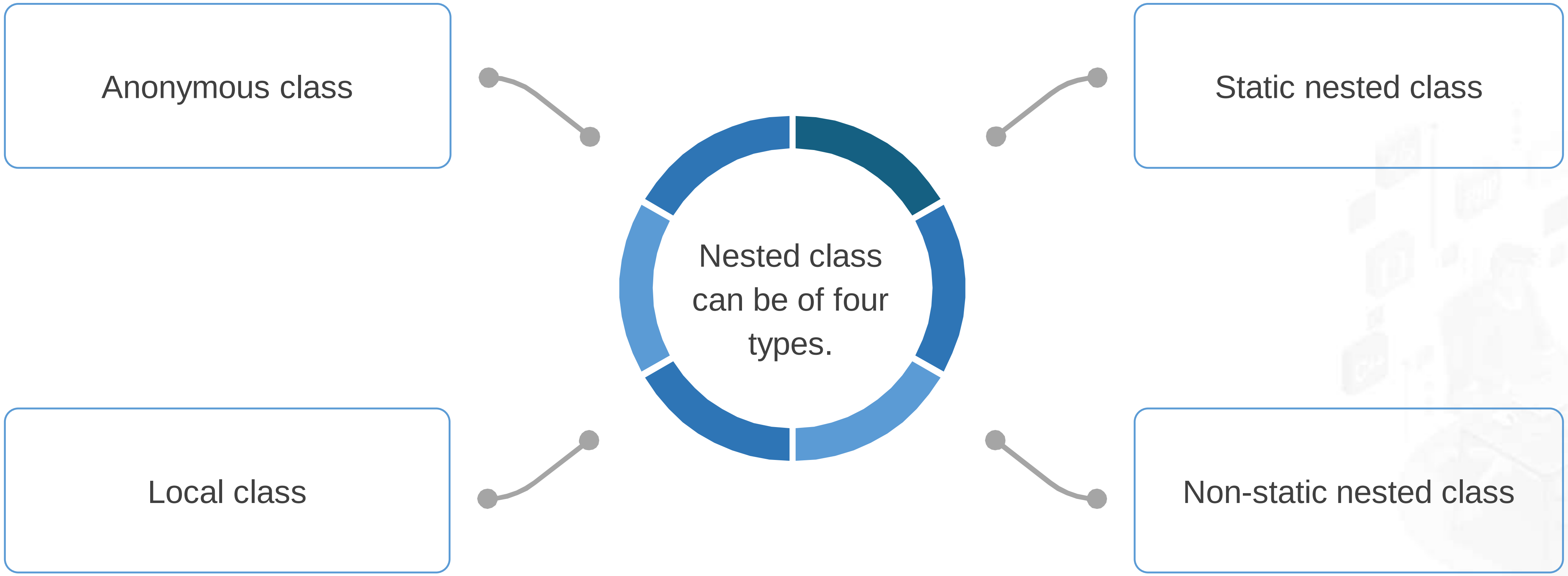
Nest-Based Access

A nest-based access control permits classes to access each other's private members. Java 11 allows interfaces and classes to be nested within each other.

The compiler inserts these into the code at the time of program execution.



Nest-Based Access



Nestmate allows verification and communication of nested classes.

Nest-Based Access

Private methods are present inside a **Nested class.**

```
public class Example {

         private void display() {
                  System.out.println("private method");
         }

         class NestedMain{
                  void msg() {
                           display();
                  }
         }
         public static void main(String[] args){

                  Example obj = new Example();
                  Example.NestedMain nObj = obj.new NestedMain();
                  nObj.msg();
         }   }
```

Nest-Based Access

Methods are used to fetch information about nest-based access control.

- ✓ **getNestHoost()**
  Used to get the name of the nest host
- ✓ **isNestmateOf()**
  Employed for checking whether a class is nestmate
- ✓ **getNestMembers()**
  Returns an array of nest members that includes interfaces and classes

Nest-Based Access

Example: The nest host of the nest can be fetched with the getNestHost() method.

```
import Java.util.Arrays;
public class Example {
        private void display() {
                System.out.println("Hello world");
        }
        class NestedMain{
                void msg() {
                        display();
                }       }
        public static void main(String[] args){
                Example obj = new Example();
                Example.NestedMain n = obj.new NestedMain();
                n.msg();
                // Get Nest Host Name
        System.out.println(Example.class.getNestHost());
                // Get Nest Members
        System.out.println(Arrays.toString(Example.class.getNestMembers())
);
                // Check whether a class is nestmateg
        System.out.println(Example.class.isNestmateOf(NestedMain.class));
        }   }
```

Implementing Method References in Java

Problem Statement:

You have been asked to develop a program that employs method references in Java.

Outcome:

By developing a program that employs method references in Java, you will learn to simplify your code by referencing existing methods directly, making your applications more concise and readable.

Note: Refer to the demo document for detailed steps: 06_Implementing_Method_References_in_Java

ASSISTED PRACTICE

Assisted Practice: Guidelines

Steps to be followed are:

1. Open the Eclipse IDE and create a new Java project
2. Create a functional interface and a class for a static void registered user
3. Create Lambda expressions
4. Create a reference to the interface and execute the code
5. Execute the log-in reference
6. Create methods that can do a return and execute the code with example data
7. Write the reference variable notification and execute the code

Implementing Optional Class in Java

Problem Statement:

You have been asked to implement an optional class in Java.

Outcome:

By implementing the Optional class in Java, you will learn to handle potential null values safely and effectively. This will help you write more robust and error-resistant code by avoiding null pointer exceptions.

Note: Refer to the demo document for detailed steps: 07_Implementing_Optional_Class_in_Java

Assisted Practice: Guidelines

Steps to be followed are:

1. Open the Eclipse IDE and create a new Java project, followed by a class
2. Create another constructor which will be parameterized with the values initialized to the inputs
3. Create users with example data and execute the code
4. Implement a safe execution through the optional class
5. Create another optional object and rerun the code

Implementing Comparator with Lambda Expressions

Problem Statement:

You have been asked to design an efficient comparator utilizing lambda expressions for object comparison.

Outcome:

By designing an efficient comparator using lambda expressions in Java, you will learn to streamline object comparison, resulting in more concise and readable code. This approach enhances your ability to sort and manage collections effectively.

Note: Refer to the demo document for detailed steps: 08_Implementing_Comparator_With_Lambda_Expressions

Assisted Practice: Guidelines

Steps to be followed are:

1. Open the Eclipse IDE and create a new Java project, followed by a class
2. Generate constructors with the fields and create the default constructor
3. Create a class and a method that returns an array list
4. Add product objects with sample data
5. Use the sort method that takes a list as input
6. Create a comparator object and execute the code
7. Write an anonymous class or a Lambda expression

Implementing Streams in Java

Problem Statement:

You have been asked to optimize Java streams for seamless processing and manipulation of large datasets.

Outcome:

By optimizing Java streams for seamless processing and manipulation of large datasets, you will learn to leverage parallel processing and efficient data operations. This will enhance your ability to handle large volumes of data quickly and effectively in your applications.

Note: Refer to the demo document for detailed steps: 09_Implementing_Streams_in_Java

Assisted Practice: Guidelines

Steps to be followed are:

1. Create a new Java project
2. Create one of the stream objects
3. Create a stream using various approaches
4. Iterate the streams and execute the code
5. Create a stream of integer numbers and execute the code
6. Generate another stream and re-execute the code

Implementing Streams with User-Defined Objects

Problem Statement:

You have been asked to facilitate efficient stream-based operations on user-defined objects in Java.

Outcome:

By facilitating efficient stream-based operations on user-defined objects in Java, you will learn to process and manipulate custom data types using the power of streams. This enhances your ability to perform complex data operations with greater ease and efficiency.

Note: Refer to the demo document for detailed steps: 10_Implementing_Streams_with_User_Defined_Objects

ASSISTED PRACTICE

Assisted Practice: Guidelines

Steps to be followed are:

1. Create a new Java project, followed by a class
2. Create constructors and define certain attributes for the class
3. Execute the code
4. Work with streams and re-execute the code
5. Create some mapped products with sample data
6. Write a new user object and run the code

Key Takeaways

- Java 11 introduces a modern HttpClient for synchronous and asynchronous requests.
- New methods like isBlank(), strip(), and stripLeading() improve whitespace management.
- The Predicate.not() method refines predicate operations, making the code more expressive.
- The toArray() method returns an array of collection elements, enhancing integration and utility.
- Nest-based access control allows classes in the same package to access each other's private members.
- The use of var in lambda expressions enables clearer and more concise code.

TECHNOLOGY

Thank You