CCBP TECH 4.0 INTENSIVE

With Industry-Ready Certification (IRC)

Java Summary Cheat Sheet - 5

https://www.ccbp.in/

Lambda Expressions

Functional Interface: A functional interface is an `interface` that has a single `abstract` method (SAM).

```
interface Runner {
    void run();
}
```

Lambda Expressions: They provide a way to represent a function as an `object`. It is an anonymous function that can be passed around as a value.

```
interface Runner {
    void run();
}
class BaseballPlayer {
    public static void main(String[] args) {
        Runner ref = () -> System.out.println("Running");
        ref.run();
    }
}
```

```
// Output is:
Running
```

Lambda Expression with Parameters: The `lambda` expressions can have parameters similar to methods.

```
interface Sum {
    int add(int a, int b);
}
class Main {
    public static void main(String[] args) {
        Sum ref = (a, b) -> a + b;
        System.out.println(ref.add(2, 3)); // 5
    }
}
```

Lambda body for a block of code: The lambda expression consisting a block of code is surrounded by curly braces.

```
interface Sum {
    int add(int a, int b);
}
class Main {
    public static void main(String[] args) {
        Sum ref = (x, y) -> {
            int numSum = x + y;
            if (numSum > 0)
              return numSum;
            return 0;
        };
        System.out.println(ref.add(2, 3)); // 5
    }
}
```

Streams

Stream: A `stream` is a sequence of elements that supports various operations for processing the elements. We can perform operations on a stream without modifying the original data.

Creating an Empty Stream: The `Stream` interface consists of a `static` and default method `empty()` that can be used to create an empty stream.

```
Stream<String> stream = Stream.empty();
```

Creating a Stream with Collections: All the classes that implement `Collection` interface consists of a `stream()` method that can be used for creating a `stream` of its corresponding type.

```
List<Integer> numbers = Arrays.asList(1, 2, 3, 4, 5);
Stream<Integer> numStream = numbers.stream();
```

We can also create a stream directly using the `of()` method of the `Stream` interface.

```
Stream<Integer> stream = Stream.of(1, 2, 3, 4, 5);
```

Chaining Streams: We can combine multiple `stream` operations together to form a single expression.

```
streamSource.intermediateOperation().terminalOperation();
```

Intermediate Operations: An intermediate operation is a stream operation that produces a new stream as output after performing the specified operations. A stream can have zero or multiple intermediate operation.

Method	Syntax	Usage
filter()	stream.filter(Predicate)	used to filter the elements of a stream based on a condition.
map()	stream.map(Function)	used to perform an operation on each element of the stream.
sorted()	stream.sorted(Comparator)	used to sort the elements of the stream.
distinct()	stream.distinct()	used to remove duplicate elements from the stream.

Terminal Operations: A terminal operation is a stream operation that consumes the elements of a stream and produces a result. After a terminal operation, no other operation can be done.

Method	Syntax	Usage
forEach()	stream.forEach(Consumer);	used to iterate and perform the specified operation on each element of the stream.
count()	stream.count();	used to count the number of elements in a stream.
collect()	stream.collect(Collector);	commonly used to perform a reduction operation on the elements of a stream and produce a result.
anyMatch()	stream.anyMatch(Predicate);	used to check if any of the elements in the stream has matched a specified condition.
allMatch()	stream.allMatch(Predicate);	used to check if all the elements in the stream match a specified condition.
noneMatch()	stream.noneMatch(Predicate);	used to check if no element in the stream matches a specified condition.
findFirst()	stream.findFirst();	used to find the first element in a stream.

Method	Syntax	Usage
findAny()	stream.findAny();	used to find any element in a stream.
reduce()	stream.reduce(BinaryOperator);	used to combine a stream of elements into a single result, using a specified operation.
min()	stream.min(Comparator);	used to find the minimum element of a stream.
max()	stream.max(Comparator);	used to find the maximum element of a stream.

Example 1:

```
List<String> names = Arrays.asList("Alice", "Bob", "Charlie", "Dave");
names.stream()
    .filter(eachName -> eachName.length() > 4)
    .forEach(name -> System.out.println(name));
```

```
// Output is:
Alice
Charlie
```

Example 2:

```
List<Integer> numbers = Arrays.asList(2, 5);
numbers.stream()
    .map(eachNumber -> eachNumber * 2)
    .forEach(number -> System.out.println(number));
```

```
// Output is:
4
10
```

Optionals

Optional Class: The `Optional` class is designed to be used as a return type for methods that may or may not return a value. It provides several methods to help prevent the `NullPointerException`.

Creating Optionals: The Optional class provides different methods to create optionals.

empty()
of()
ofNullable()

empty(): The `empty()` is a static method that creates an instance of `Optional` with no values.

```
Optional<Integer> optionalInt = Optional.empty();
System.out.println(optionalInt);  // Optional.empty
```

of(): The `of()` is a static method that accepts a non-null value as an argument and returns an instance of the `Optional` with the specified value.

```
Optional<Integer> optionalInt = Optional.of(324);
System.out.println(optionalInt); // Optional[324]
```

ofNullable(): The `ofNullable()` is a static method that accepts a value and returns the instance of `Optional` with the specified value. If null is provided as a value, it returns an `empty` optional.

```
Optional<Integer> optionalInt = Optional.ofNullable(324);
System.out.println(optionalInt); // Optional[324]
```

```
Optional<Integer> optionalInt = Optional.ofNullable(null);
System.out.println(optionalInt); // Optional.empty
```

Optional Methods:

Method	Syntax	Usage
isPresent()	optional.isPresent();	used to check if an element is present in the optional.
get()	optional.get();	used to get the element in the optional. It throws an exception if no element is present.
orElse()	optional.orElse();	used to get a default value (constant value) if the optional is empty.
orElseGet()	optional.orElseGet(Supplier);	used to get a default value (dynamic value) if the optional is empty.
ifPresent()	optional.ifPresent(Consumer);	used to perform specified operation on the elements in the optional.
map()	optional.map(Function);	used to perform a specified operation on elements in an optional and return a new optional.
filter()	optional.filter(Predicate);	used to filter the elements in an optional based on a specified condition and return a new optional.

I/O Streams

Paths: In Java, the `Path` interface from `java.nio.file` package represents a path to a file or directory in the file system. The paths are of two types:

Relative path
Absolute path

Relative Path: A relative path is a path that specifies the location of a file or directory relative to the current working directory.
Example: documents\file.txt
Absolute Path: An absolute path is a path that specifies the exact location of a file or directory in the file system, starting from the root directory.
Example: /home/rahul/documents/file.txt

Finding Absolute Path using Java

```
File file = new File("file.txt");
System.out.println(file.getAbsolutePath());
```

```
// Output is:
/home/rahul/documents/file.txt
```

BufferedWriter: The `BufferedWriter` buffers characters to provide for the efficient writing of single characters, arrays, and strings. The `Writer` writes text to a character-output stream.
Writing Data to a File

```
try {
    BufferedWriter buffer = new BufferedWriter(new FileWriter("destination.txt"));
    buffer.write("Writing to a file using BufferedWriter");
    buffer.close();
} catch (IOException e) {
    System.out.println(e.getMessage());
}
```

```
// Text in the destination.txt file
Writing to a file using BufferedWriter
```

BufferedReader: The `BufferedReader` is a class from the `java.io` package extends the abstract class `Reader`. It uses the `Reader` object to read data from a file and it stores the data in an internal buffer.
Reading Data from a File: The `BufferedReader` provides various methods to read data from a file:

read()
readLine()

read(): The `read()` method reads a single character from a file. It returns `int` data type.

```
// Text in the source.txt file
Hello World!
```

```
try {
    BufferedReader br = new BufferedReader(new FileReader("source.txt"));
    char[] arr = new char[100];
    br.read(arr);
    System.out.println(arr); // Hello World!
    br.close();
} catch (Exception e) {
    System.out.println(e.getMessage());
}
```

readLine(): The `readLine()` method is used to read a single line of text in the file. It returns `String` data type.

```
try {
    BufferedReader br = new BufferedReader(new FileReader("source.txt"));
    BufferedWriter bw = new BufferedWriter(new FileWriter("destination.txt"));
    String line = br.readLine();
    bw.write(line);
    br.close();
    bw.close();
} catch (Exception e) {
    System.out.println(e.getMessage());
}
```

```
// Text in the source.txt file
Hello World!
This is the second line.
```

```
// Text in the destination.txt file
Hello World!
```

Generics

Type Parameters: In Java, `type parameter` names are typically written in uppercase letters to distinguish them from regular class or interface names. The most commonly used type parameter names are:

E - Element
K - Key
N - Number
T - Type
V - Value

Generics: Generics allows us to write a single piece of code that can work with multiple types. Generics do not work with primitive types.

Generic Classes: In Java, a `generic class` is a class that can work with multiple data types. This allows for flexibility and reusability in programming.

```java
class SampleClass<T> {
    private T data;
    public SampleClass(T data) {
        this.data = data;
    }
    public void printDataType() {
        System.out.println("Type: " + this.data.getClass().getSimpleName());
    }
}
class Main {
    public static void main(String[] args) {
        SampleClass<Integer> intObj = new SampleClass<>(3);
        intObj.printDataType(); // Type: Integer
        SampleClass<String> stringObj = new SampleClass<>("Java");
        stringObj.printDataType(); // Type: String
   }
}
```

Generic Methods: We can create a method in Java that can be used with any type of data by using `generics`. This type of method is known as a `generics` method.

```java
class SampleClass<T> {
    private T data;
    public SampleClass(T data) {
        this.data = data;
    }
    public T getData() {
        return this.data;
    }
}
class Main {
    public static void main(String[] args) {
        SampleClass<Integer> intObj = new SampleClass<>(3);
        System.out.println(intObj.getData()); // 3
        SampleClass<String> stringObj = new SampleClass<>("Java");
        System.out.println(stringObj.getData()); // Java
   }
}
```

Generic Interfaces: We can create `interfaces` in Java that can be used with any type of data by using `generics`. This type of method is known as a `generics` `interface`.

```java
interface Processor<T> {
    void process(T t);
}
class Main<T> implements Processor<T> {
    public void process(T obj) {
        System.out.println("The process() method is called");
    }
    public static void main(String[] args) {
        Main<String> obj = new Main<>();
        obj.process("Java");
    }
}
```

```
// Output is:
The process() method is called
```

Bounded Types: We can use bounded `types` by specifying the upper bound type parameter with the `extends` keyword.

```
class Main<T extends Number> {
    T data;
    Main(T data) {
        this.data = data;
    }
    void display() {
        System.out.println(data);
    }
    public static void main(String[] args) {
        Main<Integer> intObj = new Main<>(3);
        intObj.display(); // 3
        Main<Double> doubleObj = new Main<>(3.14);
        doubleObj.display(); // 3.14
    }
}
```

Java Behind the Scenes

The `JRE` (Java Runtime Environment), `JVM` (Java Virtual Machine), and `JDK` (Java Development Kit) are the components of the `Java ecosystem`, and they are often used together when developing and running Java programs.

Java Development Kit (JDK): `JDK` provides the environment to develop and execute the Java program. It includes `JRE` and other development tools like compilers, debugger, etc.

Java Runtime Environment (JRE): `JRE` is an installation package that provides an environment to only run(not develop) the Java program onto our machine. It consists of many components like a Java class library, tools, and a separate `JVM`.

Java Virtual Machine (JVM): `JVM` is a software program that allows us to run Java programs on a computer. It is designed to work on any type of hardware or operating system, so we can run Java programs on any device as long as it has a `JVM` installed. This makes Java programs portable, meaning they can be run on any device with a `JVM`.

Execution of Java Programs: The execution of a Java code series of steps that convert the source code into a form that the computer can understand and execute.

Source code: All the Java codes we write consists of a `.java` as the file extension. These Java files act as the source of our Java codes.
Compilation: We use a command-line tool called `javac` provided by the JDK, to compile Java programs. The Java compiler takes the source file as input and produces a compiled file with the `.class` file extension. This file contains the `Java bytecode`.
Interpretation: To execute the bytecode, we use the Java interpreter, which is a command-line tool called `java`. The Java interpreter takes the `class` file as input and runs it on the `JVM`.
Executing Machine Code: `Machine code` is a low-level code that is understandable by the machine. It typically consists of 0's and 1's. The `machine code` is platform-dependent. The `JVM` is responsible for creating the `machine code` that is understandable for the specific processor or OS.

Java Archive (JAR): `JAR` is a file format used to package and share Java programs and libraries. It is similar to a ZIP file but it also includes metadata about the contents of the `JAR` file, such as the main class of a program, the version of the Java runtime required to run the program, and any dependencies on other libraries.

Classpath: The `classpath` specifies the locations where the `JVM` should look for class files when it is asked to load a class.