KEEP IN MIND BEFORE READING:

There is little to no code in this study guide, this is because the document was made for studying for the oral exam ***only***.

If you're having trouble with a concept, check out websites like GeeksforGeeks and W3Schools to learn more about it. If you prefer video tutorials, Coding with John has some great videos on Java concepts. You'll get a better grasp of these topics as you work on your project, but don't wait until you start the project to study them.

1. Classes, objects, and data types:

a. For loop types (3):

The three types of loops in Java are:

- "for" loop: It is used for iterating a specific number of times. It consists of an initialization statement, a condition, and an iteration statement.
- "while" loop: It is used when the number of iterations is unknown beforehand. The loop continues until the specified condition becomes false.
- "do-while" loop: It is similar to the "while" loop, but it guarantees that the loop body executes at least once before checking the condition.

b. Constructors:

Constructors are special methods in a class used for initializing objects. They have the same name as the class and do not have a return type, not even "void." Constructors are called automatically when an object is created using the "new" keyword. All classes have constructors by default: if you do not create a class constructor yourself, Java creates one for you.

c. Iterable Interface:

It provides a method called **iterator()** that returns an Iterator object used to traverse the elements in the collection. By implementing the Iterable interface, a class allows objects of that class to be used with the enhanced "for" loop.

In general, an object Implementing Iterable allows it to be iterated. An iterable interface allows an object to be the target of enhanced for loop(for-each loop).

d. Encapsulation:

Encapsulation is an OOP principle that binds data (attributes) and methods (behavior) together within a class. It aims to hide the internal workings of an object and expose only the necessary information or functionality through well-defined interfaces. Encapsulation helps achieve data abstraction, data hiding, and code modularity, enhancing the maintainability and reusability of code.

e. Static, non-static methods:

Static methods belong to the class itself rather than to individual instances (objects) of the class. They can be called using the class name without creating an object. Non-static methods, on the other hand, are associated with specific objects of the class and can only be invoked on those objects. Non-static methods can access both static and non-static members of a class, whereas static methods can only access static members.

2. Extending Classes, inheritance, abstract classes, and interfaces

a. Super Fathers objects and fathers constructor:

In Java, the keyword "super" is used to refer to the immediate parent class of a subclass. When a subclass extends a superclass, it inherits all the accessible members (fields and methods) of the superclass. The **super()** keyword is used to invoke the constructor of the superclass. It must be the first statement in the constructor of the subclass, and if not explicitly called, the default constructor of the superclass is implicitly invoked.

b. Overloading and overriding:

Overloading is a feature in Java that allows multiple methods in a class to have the same name but different parameters. Overloaded methods are differentiated based on the number, type, or order of parameters. Overriding, on the other hand, occurs when a subclass provides its implementation of a method that is already defined in its superclass. The overridden method in the subclass should have the same name, return type, and parameters as the method in the superclass.

c. Function and interface may have default methods:

Starting from Java 8, interfaces can have default methods. A default method is a method defined in an interface with an implementation. It provides a default behavior that can be overridden by implementing classes.

Similarly, functional interfaces are interfaces that have only one abstract method, but they can also contain default methods. Functional interfaces are typically used for lambda expressions and method references.

d. Anonymous classes - 3 types of how to make:

Anonymous classes are classes that are declared and instantiated at the same time without explicitly giving them a name. There are three ways to create anonymous classes in Java:

Anonymous Inner Class: Declared & instantiated as part of a method/constructor parameter.

Anonymous Subclass: Created by extending a class and overriding its methods.

Anonymous Interface Implementation: Created by implementing an interface & defining its methods.

e. Local class is an inner class declared inside a block:

A local class is a class declared inside a block, such as a method, constructor, or even another block. Local classes are accessible only within the block where they are defined. They are useful when you need to create a class that is closely related to the block's functionality and not used elsewhere in the code.

f. Outer objects:

In Java, an outer object refers to an instance of the enclosing class in which an inner class is defined. Inner classes have an implicit reference to their outer class instance. This reference allows the inner class to access the members (fields and methods) of the outer class. Outer objects are created separately from the inner objects and are necessary for the inner class to function properly.

g. Interface may have colliding default methods:

If two or more interfaces that are implemented by a class define a default method with the same signature, it results in a collision. In such cases, the class implementing these interfaces must explicitly override the default method and provide its implementation. This resolves the conflict and specifies which implementation should be used.

h. Access modifiers:

Access modifiers in Java determine the visibility and accessibility of classes, methods, and variables. There are four access modifiers in Java:

public: Accessible from anywhere.
private: Accessible only within the same class.
protected: Accessible within the same class, subclasses, and same package.
default (no modifier): Accessible within the same package.

i. Abstract classes:

An abstract class in Java is a class that cannot be instantiated but can be subclassed. It serves as a blueprint for other classes and can contain abstract methods, concrete methods, and instance variables. Abstract methods are declared without implementation and must be overridden by the subclasses. Abstract classes provide a way to achieve abstraction and create common behavior for related classes.

3. Exception handling:

a. Exception vs. error:

In Java, exceptions and errors are both subclasses of the Throwable class but serve different purposes.

Exceptions: Exceptions are used to indicate exceptional conditions that can be handled programmatically. They are recoverable and allow the program to continue its execution by catching and handling them appropriately.
Errors: Errors, on the other hand, represent severe problems that usually cannot be recovered from. They are often caused by system failures or resource limitations and are typically not caught or handled by applications. Examples include OutOfMemoryError and StackOverflowError.

b. How to point where an exception occurred, print a msg when the exception raised:

To identify where an exception occurred and print a message when an exception is raised, you can use the following steps:

1. Catch the exception using a try-catch block.
2. In the catch block, use the **printStackTrace()** method of the caught exception object to print the stack trace, which includes the exception message and the line numbers where the exception occurred.
3. You can also use the **getMessage()** method of the exception object to retrieve the exception message and print it along with a custom message.

java Copy code

```java
try {
  // Code that may throw an exception
} catch (Exception e) {
  System.out.println("An exception occurred: " + e.getMessage());
  e.printStackTrace();
}
```

Image: Try-catch block in Java

4. Threads

a. Life cycle of a thread:

A thread in Java goes through several stages in its life cycle:

New: When a thread is created, it is in a new state. The thread is not yet scheduled for execution.
Runnable: After calling the **start()** method, the thread enters the runnable state. It is ready to run, but the scheduler has not selected it yet.
Running: The thread moves from the runnable state to the running state when the scheduler selects it for execution.
Blocked/Waiting: A running thread can become blocked or wait for various reasons, such as waiting for I/O operations or acquiring a lock.
Timed Waiting: Similar to the blocked state, but with a specified time period.
Terminated: The thread completes its execution or is explicitly terminated by calling the **stop()** method. Once terminated, the thread cannot be restarted.

b. How to start and stop a thread:

To start a thread, you need to create an instance of the thread class and call its **start()** method. The **start()** method internally calls the **run()** method, which contains the code to be executed in the new thread.

To stop a thread, you can use the **interrupt()** method, which sets the interrupt status of the thread to true. It is up to the thread's code to check the interrupt status using the **isInterrupted()** method and gracefully exit the thread's execution loop.

c. How to create a thread with overriding and Runnable:

There are two ways to create a thread in Java:

1. By extending the Thread class:
    - Create a class that extends the Thread class.
    - Override the **run()** method in the subclass with the code to be executed in the thread.
    - Create an instance of the subclass and call its **start()** method to start the thread.

2. By implementing the Runnable interface:
    - Create a class that implements the Runnable interface.
    - Implement the **run()** method from the Runnable interface with the code to be executed in the thread.
    - Create an instance of the class.

- Create a Thread object, passing the instance of the class as a parameter to the Thread constructor.
- Call the **start()** method on the Thread object to start the thread.

Example (implementing Runnable):

```java
class MyRunnable implements Runnable {
  public void run() {
    // Code to be executed in the thread
  }
}

// Creating and starting the thread
Thread thread = new Thread(new MyRunnable());
thread.start();
```

d. Default method in Runnable thread:

The Runnable interface in Java is a functional interface, meaning it has only one abstract method called **run()**. It does not provide any default methods by default. However, since Java 8, interfaces can have default methods, so it is possible to define additional default methods in an interface that extends the Runnable interface.

e. Does Thread implement Runnable? Yes, we only need a run method to implement Runnable:

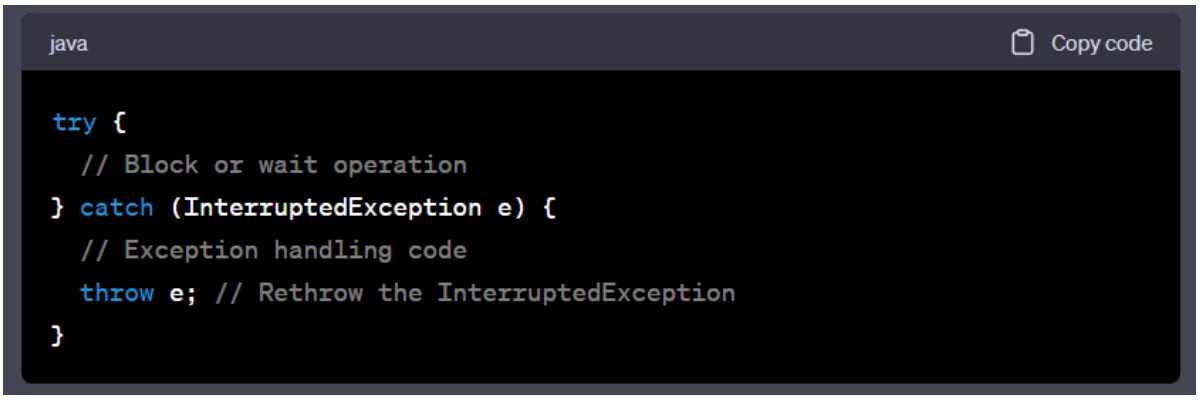
Yes, the Thread class in Java implements the Runnable interface. The Thread class itself provides an implementation of the **run()** method, which is the entry point for the thread's execution. When a Thread object is started, it internally calls the **run()** method. By implementing the Runnable interface, you can provide your own implementation of the **run()** method and pass it to a Thread object to execute in a separate thread.

f. How to stop a thread? Interrupt:

To stop a thread gracefully in Java, you can use the **interrupt()** method provided by the Thread class. When you call **interrupt()** on a thread, it sets the interrupt status of the thread to true. It does not forcefully stop the thread but acts as a request for the thread to stop. It is up to the thread's code to check the interrupt status periodically using the **isInterrupted()** method and terminate its execution accordingly.

g. How to throw an InterruptedException:

The InterruptedException is a checked exception that is thrown when a thread is interrupted while it is in a blocking or waiting state, such as when waiting for I/O operations or acquiring locks. To throw an InterruptedException, you can use the throw keyword followed by an instance of the InterruptedException class.



i. NotifyAll:

The **notifyAll()** method is similar to the **notify()** method but wakes up all the threads that are waiting on the same object's monitor. It is called by a thread that holds the object's monitor and wants to notify all waiting threads. All the notified threads move from the waiting state to the notified state and can compete for the object's monitor to continue their execution.

j. IllegalMonitorStateException:

The IllegalMonitorStateException is a runtime exception that occurs when a thread tries to invoke the **wait()**, **notify()**, or **notifyAll()** methods on an object without owning the object's monitor. It typically happens when the calling thread does not synchronize on the object using the synchronized keyword before calling these methods.

k. How does a thread notify another thread? Thread.notify(), and a random thread is chosen:

When a thread calls the **notify()** method on an object, it wakes up a single thread that is waiting on the same object's monitor. However, it is important to note that the specific thread that is awakened is not deterministic. The Java runtime system chooses the thread from the waiting set of threads, and the selection is generally not in any specific order or based on priority.

l. Deadlocks:

A deadlock is a situation in concurrent programming where two or more threads are blocked forever, waiting for each other to release resources. It occurs when each thread holds a resource that the other thread needs to proceed, resulting in a circular dependency. Deadlocks can cause a significant impact on the program's execution, leading to a system freeze or unresponsiveness.

m. Thread starvation:

Thread starvation occurs when a thread is unable to make progress and is continuously delayed or preempted by other threads. It happens when a higher-priority thread or multiple threads with higher priority constantly occupy shared resources, leaving a lower-priority thread with limited or no opportunity to execute. Thread starvation can result in reduced performance or even deadlock situations.

n. Volatile:

The volatile keyword in Java is used to indicate that a variable may be modified by multiple threads. It ensures that the variable's value is always read from and written to the main memory, avoiding any caching or optimization by individual threads. The use of volatile ensures that changes made by one thread are visible to all other threads, preventing inconsistencies due to thread interleaving and ensuring thread-safe communication.

5. Lambda expression, generics, and reflection:

a. Functional Interface:

A functional interface in Java is an interface that contains only one abstract method. It serves as a target for lambda expressions or method references. Functional interfaces enable the use of functional programming concepts in Java and provide a concise way to represent behavior as data.

b. How to write a lambda function:

A lambda function in Java is a concise way to define an anonymous function. It allows you to write code that can be treated as a value and passed around. A lambda function is defined using the following syntax:

```
(parameter_list) -> { lambda_body }
```

where the parameter_list specifies the parameters of the function, and the lambda_body contains the code to be executed.

Example:

```
// Lambda function that adds two numbers
(int a, int b) -> { return a + b; }
```

c. Implement interface on the fly using lambda expressions:

Lambda expressions in Java can be used to implement functional interfaces on the fly without the need for explicit class definitions. By providing a concise syntax, lambda expressions allow you to define behavior inline, making the code more expressive and readable.

Example:

```java
// Functional interface with one abstract method
interface Calculator {
  int calculate(int a, int b);
}

public class Main {
  public static void main(String[] args) {
    // Implementing the interface using a lambda expression
    Calculator addition = (a, b) -> a + b;
    int result = addition.calculate(2, 3);
    System.out.println(result);  // Output: 5
  }
}
```

d. Stack using reflections:

Reflection in Java allows you to inspect and manipulate classes, interfaces, fields, methods, and constructors at runtime. While reflection can be used to access and modify stack frames, it is important to note that it is generally not recommended to use reflection for stack manipulation, as it can lead to unsafe and error-prone code.

e. Classes in the reflection package:

The reflection package in Java (java.lang.reflect) provides classes and interfaces that enable runtime reflection. Some important classes in the reflection package include:

Class: Represents a class or interface at runtime and provides methods to inspect its members, annotations, and generic type information.
Constructor: Represents a constructor of a class and provides methods to create new instances of the class.
Method: Represents a method of a class and provides methods to invoke the method dynamically.
Field: Represents a field of a class and provides methods to get or set the value of the field.

f. Object of a generic type without instantiating the generic variable:

In Java, it is not possible to directly create an object of a generic type without instantiating the generic variable. The generic type parameter provides compile-time type safety and is erased at runtime. If you want to create an object of a generic type, you need to instantiate the generic variable with an actual type argument.

```
class MyClass<T> {
  private T value;

  public MyClass(T value) {
    this.value = value;
  }

  public T getValue() {
    return value;
  }
}

// Creating an object of generic type by instantiating the generic variable
MyClass<Integer> myObject = new MyClass<>(42);
int value = myObject.getValue();  // Getting the value (42)
```

g. Generic type:

A generic type in Java allows you to create classes, interfaces, or methods that can operate on different types while maintaining type safety. It enables code reusability and flexibility by providing a way to define classes or methods that work with different types specified at compile time.

In this example, the Box class is generic, and the type parameter T can be replaced with any specific type.

```
// Generic class
class Box<T> {
  private T content;

  public void setContent(T content) {
    this.content = content;
  }

  public T getContent() {
    return content;
  }
}

// Creating an instance of the generic class with a specific type argument
Box<String> stringBox = new Box<>();
stringBox.setContent("Hello");
String content = stringBox.getContent();  // Getting the content ("Hello")
```