Index:

Jvm,jdk,jre

String builder,buffer

Try catch

Exceptional handling

Garbage collector

Wrapper class

files

Core java :

Of course! Here's a **Core Java Interview Questions and Answers** guide — covering topics like String, Builder, Buffer, immutability, == vs equals(), JVM, JDK, and OOP concepts.

* **🔹 1. String, StringBuilder, StringBuffer**

**Q1. What is the difference between String, StringBuilder, and StringBuffer?**

| **Feature** | **String** | **StringBuilder** | **StringBuffer** |
| --- | --- | --- | --- |
| Mutability | Immutable | Mutable | Mutable |
| Thread-safe | No | No | Yes (synchronized) |
| Performance | Slow in concatenation | Fast | Slower than StringBuilder |

**Q2. Why is String immutable in Java?**

* Because once a String object is created, it cannot be changed.
* **Reasons:**
  + Security (used in class loading, file paths, etc.)
  + Thread safety (safe to use across threads)
  + Caching in string pool (string literals are shared)

**Q3. How does StringBuilder differ from StringBuffer?**

* Both are mutable, but:
  + StringBuffer is **synchronized** → thread-safe but slower.
  + StringBuilder is **not synchronized** → faster but not thread-safe.

**Q4. What happens in:**

String s1 = "abc";

String s2 = s1 + "def";

* s2 creates a **new String object** ("abcdef") since String is immutable. s1 remains "abc".

**🔹 2. == vs equals()**

**Q1. What is the difference between == and .equals()?**

* == checks **reference** (memory address).
* .equals() checks **content/value**.

**Q2. Example:**

String s1 = "hello";

String s2 = new String("hello");

System.out.println(s1 == s2); // false (different memory)

System.out.println(s1.equals(s2)); // true (same content)

**Q3. What is String interning?**

* JVM maintains a **string pool**.
* If a string with the same content exists, new variables point to the same object to save memory.

**🔹 3. JVM, JDK, JRE**

**Q1. What is the difference?**

| **Term** | **Description** |
| --- | --- |
| **JVM (Java Virtual Machine)** | Runs the Java bytecode (.class files) |
| **JRE (Java Runtime Environment)** | JVM + libraries to run Java apps |
| **JDK (Java Development Kit)** | JRE + tools for development (compiler, debugger, etc.) |

**Q2. JVM Memory Areas:**

* **Heap**: Objects
* **Stack**: Method calls and local variables
* **Method Area**: Class metadata
* **PC Register**: Address of current instruction
* **Native Method Stack**: Native (C/C++) method calls

**Q3. What is JIT (Just-In-Time) Compiler?**

* A part of JVM that compiles bytecode to native code at runtime to improve performance.

**Q4,🔹== vs .equals() in Java**

| **Feature** | **== Operator** | **.equals() Method** |
| --- | --- | --- |
| **Type** | **Operator** | **Method (defined in Object class)** |
| **Checks** | **Reference equality (memory address)** | **Content equality (data/values)** |
| **Applicable to** | **Primitives & Objects** | **Objects only** |
| **Customizable** | **❌ No** | **✅ Yes (can override in your own classes)** |

Q5,

Absolutely! Here's a **deep dive into Exception Handling in Java** — super important for interviews and real-world development. We'll cover concepts, types, keywords, best practices, and code examples.

**🚨 What is Exception Handling in Java?**

**Exception handling** is a mechanism to handle **runtime errors** so that the normal flow of the application can be maintained.

**🔹 1. Exception Hierarchy**

Throwable

├── Exception (checked)

│ ├── IOException

│ ├── SQLException

│ └── etc.

└── RuntimeException (unchecked)

├── NullPointerException

├── ArithmeticException

└── etc.

* Throwable is the superclass for all errors and exceptions.
* Exception: conditions that applications might want to catch.
* Error: serious problems (like OutOfMemoryError) — not meant to be caught.

**🔹 2. Types of Exceptions**

| **Type** | **Description** | **Examples** |
| --- | --- | --- |
| **Checked Exception** | Caught or declared in the method signature | IOException, SQLException |
| **Unchecked Exception** | Occur during runtime | NullPointerException, ArithmeticException |
| **Error** | Serious issues beyond app control | OutOfMemoryError, StackOverflowError |

**🔹 3. Exception Handling Keywords**

| **Keyword** | **Use** |
| --- | --- |
| try | Wrap code that might throw an exception |
| catch | Handle the exception |
| finally | Executes code regardless of exception |
| throw | Throw an exception manually |
| throws | Declare exceptions in method signature |

**🔸 Example:**

try {

int a = 10 / 0; // ArithmeticException

} catch (ArithmeticException e) {

System.out.println("Cannot divide by zero: " + e);

} finally {

System.out.println("This block always executes.");

}

**🔹 4. throw vs throws**

| **throw** | **throws** |
| --- | --- |
| Used to explicitly throw an exception | Used in method signature |
| Only one exception can be thrown | Can declare multiple exceptions |
| Followed by instance | Followed by class name |

**Example:**

void checkAge(int age) {

if (age < 18)

throw new IllegalArgumentException("Underage not allowed");

}

void readFile() throws IOException {

FileReader fr = new FileReader("file.txt");

}

**🔹 5. Multiple Catch Blocks**

try {

String s = null;

System.out.println(s.length());

} catch (NullPointerException e) {

System.out.println("Null Pointer caught!");

} catch (Exception e) {

System.out.println("General exception caught.");

}

**Note**: Always catch subclasses first. Catching Exception before NullPointerException will cause a compile error.

**🔹 6. finally Block Use Case**

Used for:

* Closing DB connections
* Closing file readers
* Cleanup code

try {

int[] arr = new int[3];

arr[5] = 10;

} catch (ArrayIndexOutOfBoundsException e) {

System.out.println("Index error");

} finally {

System.out.println("Cleanup done");

}

**🔹 7. Custom Exception**

You can define your own exceptions by extending Exception or RuntimeException.

class MyException extends Exception {

MyException(String msg) {

super(msg);

}

}

public void validate(int age) throws MyException {

if (age < 18)

throw new MyException("Not eligible");

}

**🔹 8. Best Practices**

* Don’t catch generic Exception unless needed.
* Close resources using **try-with-resources** (Java 7+).
* Use custom exceptions for business logic.
* Avoid swallowing exceptions silently.
* Log exceptions properly for debugging.

**🔹 9. try-with-resources (Java 7+)**

Auto-close resources like BufferedReader, FileReader.

try (BufferedReader br = new BufferedReader(new FileReader("data.txt"))) {

String line = br.readLine();

} catch (IOException e) {

e.printStackTrace();

}

**🔹 10. Exception Propagation**

If not caught in the current method, the exception **bubbles up** to the caller.

public void a() {

b(); // exception in b() propagates to a()

}

public void b() {

int x = 10 / 0;

}