JAVA BASICS

Is Java 100% object-oriented? Why or why not?

Java is not considered 100% or "purely" object-oriented, despite being a strongly object-oriented language. This classification stems from two primary reasons:

- Primitive Data Types:
  Java incorporates primitive data types such as int, float, boolean, char, etc. These are not objects; they do not possess methods or support object-oriented features like inheritance. In a purely object-oriented language, all data, including basic values, would be represented as objects. While Java provides wrapper classes (e.g., Integer, Float) to treat primitives as objects when needed, their existence as distinct, non-object types prevents Java from achieving 100% object-oriented purity.
- Static Members:
  Java allows for static methods and variables within classes. These static members belong to the class itself rather than to specific instances (objects) of that class. Static methods can be invoked without creating an object, and static variables can be accessed directly through the class name. This capability enables a more procedural style of programming in certain contexts, deviating from the strict object-centric paradigm of a purely object-oriented language where all operations are performed through message passing to objects.

What actually happens when you run `javac` vs when you run `java`?

When you run `javac`:

Compilation:
The `javac` command invokes the Java compiler. Its primary function is to read Java source code files (files with a `.java` extension) and translate them into Java bytecode.
Bytecode Generation:
The output of `javac` is one or more `.class` files. These files contain platform-independent bytecode, which is a low-level representation of your Java program that can be understood and executed by the Java Virtual Machine (JVM).
Error Checking:
During compilation, `javac` performs syntax and semantic checks on your source code. If it finds any errors (e.g., typos, incorrect method calls, type mismatches), it will report them as compilation errors, preventing the creation of `.class` files until the errors are resolved.

When you run `java`:

JVM Invocation:
The `java` command launches the Java Virtual Machine (JVM). The JVM is the runtime environment that executes Java bytecode.
Class Loading:
The JVM loads the specified `.class` file (or files) into memory. This involves verifying the bytecode for security and integrity.
Execution:
The JVM's execution engine then interprets or just-in-time (JIT) compiles the bytecode into machine-specific instructions and executes your program. This is where your Java application actually runs and performs its intended operations.
Runtime Operations:
During execution, the JVM manages memory (garbage collection), handles threads, and interacts with the underlying operating system and hardware as required by your program.

❓ What actually happens when you run `javac` vs when you run `java`?

👉 `javac` (Java Compiler)

- `javac` is the **Java compiler**.
- It takes your **`.java` source code** (human-readable) and **compiles it into `.class` files** (bytecode).

Example:

javac HelloWorld.java

- Produces → `HelloWorld.class`

⚡ At this point, your code is **not machine code yet**; it's **bytecode**, which is platform-independent.

👉 `java` (Java Launcher)

- `java` is **not a compiler**; it's the **Java interpreter/launcher**.

When you run:

```
java HelloWorld
```

- It does **not** look for `HelloWorld.java`.
  Instead, it loads the **compiled `HelloWorld.class` bytecode** into the JVM.

- The JVM uses the **ClassLoader** to load the class, then finds the `public static void main(String[] args)` method as the program entry point.

- The JVM executes the bytecode using the **JIT (Just-In-Time) compiler**, which translates bytecode → native machine code at runtime for efficiency.

✅ Summary:

- `javac` = compile `.java` → `.class` (bytecode).

- `java` = run `.class` (bytecode) using JVM + JIT → machine code.

Can we run a Java program without a `main` method in Java 8+?

👉 **Answer:**
No, in Java 8 and later, you cannot run a standalone Java program without a `main` method.

Before Java 7, there were **tricks/loopholes** (like using a static initializer block) to run code without a `main`. Example:

```
class Test {
   static {
       System.out.println("Hello without main!");
       System.exit(0);
   }
}
```

- ✅ This worked in Java 6.

But from **Java 7 onward (including Java 8+)**, the JVM strictly requires a `main` method as the **entry point**.
If you try the above in Java 8+, you'll get:

```
Error: Main method not found in class Test
```

-

⚡ **Exceptions:**

- In **Java EE / Jakarta EE, Android, Applets, Servlets, or JavaFX**, you don't always write a `main` method yourself, because the **container/framework provides its own entry point**.
- But for a **normal standalone Java application**, `main` is mandatory in Java 8+.

Why is main method public static void main(String[] args) and not something else? What if we change the order of keywords?

👉 1. `public`

- It must be `public` so the JVM can access it from outside the class.
- If it were `private`, `protected`, or default (package-private), JVM wouldn't be able to call it.
- JVM looks for a public main as the entry point.

👉 2. `static`

- JVM does not create an object of your class to start execution.
- So `main` must be `static`, meaning it can be called without creating an instance.
- Otherwise, JVM would have to construct an object (which needs a constructor), and we'd have a chicken–egg problem. This is crucial as it provides a clear and immediate entry point for program execution, saving memory and execution time by avoiding unnecessary object creation.

👉 3. `void`

- The JVM doesn't expect a return value from `main`.
- If it returned an `int` (like C/C++), there would be ambiguity about what to do with the return code.
- Instead, Java uses `System.exit(status)` to return exit codes.

👉 4. `main`

- This is just the name convention that JVM looks for as the entry point.
- You can define another method like `public static void start(String[] args)`, but JVM won’t call it unless you explicitly do so.

👉 5. `String[] args`

- This allows the program to receive command-line arguments. This parameter allows the `main` method to accept command-line arguments as an array of strings. These arguments can be used to provide input or configure the program's behavior at runtime. The name `args` is a convention, and it can be changed to any valid identifier, but the type `String[]` is mandatory.

JVM passes any arguments after the class name to this array.
Example:

java Hello a b c

- → `args = {"a", "b", "c"}`

❓ What if we change the order of keywords?

✅ Java allows keyword order variation as long as it makes sense to the compiler.
For example, all of these are valid:

public static void main(String[] args)

static public void main(String[] args)

final strictfp public static void main(String[] args)

⚠️ But something like:

void static public main(String[] args) // ❌ INVALID fails, because return type (`void`) must come before the method name.

✅ Final Answer:

- `main` is public (JVM needs access), static (no object needed), void (no return to JVM), and takes a `String[]` for command-line args.
- Keyword order can vary a bit (`public static` vs `static public`), but `void` must stay before `main`.

If Java is platform independent, then why do we still need JVMs specific to OS (Windows JVM, Linux JVM, etc.)?
Java's platform independence stems from its "write once, run anywhere" philosophy, achieved through the use of bytecode and the Java Virtual Machine (JVM). While Java source code is compiled into platform-independent bytecode, the JVM itself is platform-dependent.

Here's why:

Bytecode is Universal, Execution is Not:

Java source code is compiled into bytecode, which is a universal, platform-neutral intermediate language. This bytecode can be understood by any JVM, regardless of the underlying operating system or hardware.

JVM Translates Bytecode to Native Code:

To execute the bytecode, the JVM must translate it into native machine code specific to the underlying operating system and hardware architecture. Since operating systems (Windows, Linux, macOS) have different system calls, memory management, and hardware interactions, a single JVM cannot directly execute bytecode on all platforms.

Platform-Specific Implementations of JVM:

Therefore, different implementations of the JVM are required for each platform. For example, a Windows JVM is specifically designed to interact with the Windows operating system and its hardware, while a Linux JVM is designed for Linux systems.

These platform-specific JVMs ensure that the same Java bytecode can be executed seamlessly on various systems.

In essence, Java achieves platform independence at the bytecode level, allowing developers to write code once without worrying about the target platform. The platform-dependent JVM then acts as the crucial intermediary, translating this universal bytecode into the specific instructions required by the underlying system.

👉 **Key idea:**

- **Java source code → compiled into bytecode (`.class` files).**
- That bytecode is **platform independent** — the same `.class` can run on Windows, Linux, Mac, etc.
- But the JVM itself is a **native program** written in C/C++ and compiled for a particular operating system and CPU architecture.

⚡ Why different JVMs are needed?

1. **Bytecode → Native Code:**
    - The JVM must ultimately convert bytecode into **machine instructions** that the **underlying CPU + OS** understand.
    - Since Windows, Linux, and Mac have different system calls, file structures, memory management, etc., the JVM implementation must adapt to each environment.
2. **Platform Abstraction Layer:**
    - Java is “write once, run anywhere” only at the **bytecode level**.
    - At runtime, the JVM acts as a **bridge** between your portable bytecode and the OS-specific details.
3. **JIT Compiler & Libraries:**

    - The **JIT compiler** (inside JVM) generates machine code optimized for the CPU it's running on.
    - The JVM also interacts with **native libraries** (for I/O, threading, graphics), which differ across OSs.

✅ **Summary Answer:**

- **Java is platform-independent at the source code / bytecode level.**
- **JVM is platform-dependent,** because it has to translate bytecode into the specific machine instructions of the OS and CPU where it runs.
- That's why you download different JVMs for Windows, Linux, or Mac — but your `.class` files remain unchanged.

Is Java compiled or interpreted? Or both? Explain how.

Java is both a compiled and interpreted language. This hybrid approach enables Java's "Write Once, Run Anywhere" (WORA) capability.

How it works:

Compilation to Bytecode:

- Java source code, written by a developer in `.java` files, is first compiled into an intermediate format called bytecode.
- This compilation is performed by the Java Compiler (javac).
- The resulting bytecode files have a `.class` extension.
- Bytecode is platform-independent, meaning it is not specific to any particular machine architecture or operating system.

Interpretation and Execution by JVM:

- The compiled bytecode is then executed by the Java Virtual Machine (JVM).

- The JVM acts as an interpreter, translating the bytecode instructions into machine-specific code at runtime, which the underlying hardware and operating system can understand and execute.
- This interpretation allows the same bytecode to run on any system that has a compatible JVM installed, regardless of the underlying platform.

Just-In-Time (JIT) Compilation for Performance:

- To improve performance, modern JVMs incorporate a Just-In-Time (JIT) compiler.
- The JIT compiler analyzes frequently executed sections of bytecode (hotspots) during runtime and compiles them into native machine code.
- This compiled native code can then be executed directly by the CPU, bypassing the interpretation step for those specific sections, leading to significant performance gains.

Java is **both compiled and interpreted (and even JIT-compiled).**

⚡ Step-by-step process:

1. **Compilation (javac):**
    - Java source (`.java`) → compiled by `javac` → **bytecode (`.class`)**.
    - This is a **one-time compilation**.
    - Bytecode is **platform-independent**, unlike C/C++ machine code.
2. **Interpretation (JVM):**
    - The JVM reads bytecode and **interprets** it into machine instructions, instruction by instruction.
    - This ensures portability — same bytecode runs everywhere.
    - Downside: interpretation is slower than native execution.
3. **JIT Compilation (Just-In-Time Compiler):**

    - To improve performance, the JVM includes a **JIT compiler**.
    - JIT compiles frequently used bytecode sections into **native machine code at runtime**, caching them for reuse.
    - So “hotspot” code runs almost as fast as C/C++.

✅ **Summary in one line:**
Java is **compiled into bytecode** (platform-independent), then **interpreted by the JVM**, and frequently **JIT-compiled into native machine code** for speed.

Suppose you only have JRE installed, not JDK. Can you run Java code? Can you compile Java code?

If only the Java Runtime Environment (JRE) is installed, not the Java Development Kit (JDK), the following applies:

Running Java Code:

- Yes, you can run pre-compiled Java code (bytecode in `.class` files or packaged in `.jar` files) with only the JRE installed. The JRE includes the Java Virtual Machine (JVM) and the necessary libraries to execute Java applications.
-

Compiling Java Code:

- No, you cannot compile Java source code (`.java` files) into bytecode with only the JRE installed. The JRE does not include the Java compiler (`javac`), which is a component of the JDK. Therefore, to compile Java code, you need the JDK.

1. **Running Java code (✅ Possible with JRE):**
    - JRE (Java Runtime Environment) contains:
        - JVM (to execute `.class` files)
        - Core libraries (`rt.jar` etc.)
        - Other runtime support files

So, if you already have compiled bytecode (`.class` files), you **can run it** with:

```
java HelloWorld
```

---

2. **Compiling Java code (❌ Not possible with only JRE):**
    - The **compiler (`javac`) is part of the JDK**, not the JRE.
    - Without JDK, you cannot compile `.java` → `.class`.

If you try:

```
javac HelloWorld.java
```
You'll get:

```
'javac' is not recognized as an internal or external command...
```

✅ Final Word:

- **JRE only → Can run code, cannot compile code.**
- **JDK (which includes JRE + compiler + tools) → Can compile and run.**

⚡Trick twist: In **Java 9+**, Oracle stopped providing a separate JRE download — the JDK ships with both compiler and runtime. But conceptually, the distinction still matters.

Why does Java say “Write once, run anywhere” but not truly “compile once, run anywhere”?

- Java’s slogan is **“Write Once, Run Anywhere (WORA)”** because:
    - You write your source code **once** (`.java`).
    - Compile it into **bytecode** (`.class`).
    - That bytecode can run **anywhere**, on any platform, **as long as a compatible JVM is available**.

⚡ **Why not *“Compile once, run anywhere”*?**

Because compilation is **not 100% portable**:

1. **JDK / Compiler Differences**
    - Different JDK versions have different compilers (`javac`).
    - Example: A program using **Java 17 features** won’t compile with JDK 8.
    - So compilation depends on which JDK you use.

2. **Bytecode Versioning**
    - Compiled bytecode is tied to a **specific class file version**.
    - Example:
        - Java 8 → `.class` files with major version 52.0
        - Java 11 → `.class` files with major version 55.0

If you compile with a newer JDK but try to run on an **older JVM**, you'll see:

UnsupportedClassVersionError

    -
3. **Native Dependencies**
    - If your code uses **JNI (Java Native Interface)** or platform-specific libraries, the compiled `.class` may not behave identically across systems.

✅ **Final Word:**

- **WORA (Write Once, Run Anywhere):** source code + bytecode can run on any JVM.
- **Not "Compile Once, Run Anywhere":** because compiled bytecode may depend on the **Java version and JVM compatibility**.

Can a `.class` file run without a `.java` file? Under what condition?

Yes, a `.class` file can be executed without the corresponding `.java` source file being present.

Condition:

The primary condition for running a `.class` file without its `.java` source is the presence of a Java Runtime Environment (JRE) or a Java Development Kit (JDK) installed on the system, which includes the Java Virtual Machine (JVM). The JVM is responsible for interpreting and executing the bytecode contained within the `.class` file.

Explanation:

Compilation:
The `.java` file, which contains the human-readable source code, is first compiled by the Java compiler (`javac`) into bytecode. This bytecode is then stored in a `.class` file.
Execution:
The JVM directly executes the bytecode within the `.class` file. It does not require the original `.java` source file at runtime. The `.class` file contains all the necessary instructions for the JVM to run the program.
Main Method:
For a standalone application, the `.class` file must contain a `main` method, which serves as the entry point for the program's execution.

Limitations:

- If the program uses multiple classes, all required `.class` files must be present at runtime.
- If some are missing, you'll get `ClassNotFoundException` or `NoClassDefFoundError`.
- But the `.java` source is never needed for execution.

✅ **Final Word:**

- `.java` file is **needed only for compilation**.
- `.class` file (bytecode) is **needed for execution**.

- So yes, you can delete the `.java` file and still run the program, as long as the `.class` file exists.

If you remove `public` from `public class Main { ... }`, will the program still compile and run? Why/why not?

If the `public` modifier is removed from `public class Main { ... }` in a Java program, the program will still compile, but it will not run as an executable application if `Main` is intended to be the entry point class.

Explanation:

> **Compilation:**
> Java compilation focuses on syntax and type checking. Removing `public` from the class declaration does not introduce a syntax error, so the Java compiler (`javac`) will successfully compile the `.java` file into a `.class` file.
> **Execution (Running):**
> The Java Virtual Machine (JVM) requires the entry point class (the one containing the `public static void main(String[] args)` method) to be `public`. When you attempt to run the compiled `.class` file using the `java` command, the JVM searches for a `public` class with the specified name that contains the `main` method. If the class is not `public`, the JVM cannot access it from outside its package, and therefore it cannot find and execute the `main` method. This will typically result in an error message indicating that the main class could not be found or loaded.

❓ **Without `public`, what will be the default class type?**

If you don't specify any modifier before `class`, the class gets **package-private (a.k.a. default access)**.

⚡ Meaning of package-private (default):

- The class is **accessible only within the same package**.
- It cannot be accessed directly from classes in **other packages**.
- Example:

```
// File: pkg1/ClassA.java
package pkg1;

class ClassA {   // 👈 no modifier = package-private
    void show() {
        System.out.println("Hello from ClassA");
    }
}

// File: pkg2/Test.java
package pkg2;

public class Test {
    public static void main(String[] args) {
        ClassA obj = new ClassA();  // ❌ Compilation error
    }
}
```

Error:

```
ClassA is not public in pkg1; cannot be accessed from outside package
```

✅ Final Word:

- **No modifier** → **package-private (default access)**.
- Meaning: visible only inside the **same package**.
- Other valid modifiers for top-level classes are only **public** or **default (package-private)**.

- `protected` and `private` **cannot** be used for top-level classes (only for inner classes, methods, variables).
- This default access level is also sometimes referred to as "no modifier" access.

🚀 Let's unpack **why pointers were removed (or heavily restricted)** from many modern programming languages, even though they were core to languages like C and C++.

🔹 What are pointers, briefly?

A **pointer** is a variable that stores the memory address of another variable.
Example in C:

```
int x = 10;
int *p = &x;  // p stores the address of x
```

This gives **low-level control**: you can access, modify, or even reinterpret memory directly.

🔹 Why many languages moved away from raw pointers

1. **Safety & Security Risks**
   - **Dangling pointers** → accessing memory after it's freed → crashes or undefined behavior.
   - **Buffer overflows** → writing outside allocated memory → security vulnerabilities (e.g., many C-based exploits).
   - **Pointer arithmetic mistakes** → subtle, hard-to-detect bugs.

Example:

```
 int arr[5];
int *p = arr;
```

*(p + 10) = 50; // writes outside array → corruption

2. Languages like Java, Python, C#, Rust (safe mode) deliberately **hide raw pointers** to prevent such errors.

---

2. **Memory Management Complexity**
    - In C/C++, programmers must manage `malloc`/`free` (or `new`/`delete`).
    - Forgetting to free → **memory leaks**.
    - Freeing too early → **dangling pointers**.
    - Freeing twice → **double free error**.
3. Modern languages rely on **garbage collection** (Java, Python, Go) or **ownership/borrowing systems** (Rust), making explicit pointers unnecessary.

---

3. **Simpler Syntax & Developer Productivity**
    - Many developers, especially beginners, found pointer syntax confusing (`*p`, `&x`, `p++`).
    - Removing raw pointers makes language easier to learn and reduces accidental misuse.
    - Instead, languages use **references** or **objects**, which conceptually do the same thing (they point), but without exposing raw addresses.

---

4. **Portability & Abstraction**
    - Pointers expose **machine-level details** (actual memory addresses, word size).
    - This ties code to a specific architecture.

- High-level languages wanted **platform independence** (e.g., Java's *"Write once, run anywhere"*), so they abstracted away direct memory access.

---

5. **Encapsulation & Modern Programming Paradigms**
   - Object-oriented and managed languages emphasize **encapsulation**—you interact with objects, not raw memory.
   - Exposing pointers would allow bypassing protections (e.g., modifying private fields directly).
   - Instead, controlled references (like Java references or C# managed references) maintain safety.

---

6. **Alternative Mechanisms Replace Pointers**
   - **References** (C++, Java, C#) → behave like safe pointers without arithmetic.
   - **Handles** (Windows APIs, databases) → indirect references to resources.
   - **Smart pointers** (C++) → automatically manage memory (RAII).
   - **Borrow checker** (Rust) → ensures safe memory usage at compile time.
7. So the *usefulness* of pointers is preserved, but the *risks* are reduced.

🔹 Summary

Pointers weren't removed because they're useless—they're extremely powerful.
They were removed (or hidden) in many modern languages because they are:
✅ error-prone,
✅ unsafe,

✅ hard for beginners,
✅ unnecessary when higher-level abstractions exist.

Languages like **C and C++ keep them** for system programming and fine-grained control.
Languages like **Java, Python, Go, C#, Swift** hide or replace them with safer constructs.
Languages like **Rust** reinvented them with *ownership and borrowing* for both **safety + performance**.

Both **double free error** and **memory leak** are classic issues tied to manual memory management (like in C or C++). Let me break them down clearly:

🔹 Memory Leak

A **memory leak** happens when:

- You allocate memory on the heap (using `malloc`, `calloc`, or `new`),
- But you **forget to free** it after use.

The memory remains “reserved” but **unreachable** because no variable points to it anymore.
Over time, leaks pile up, reducing available memory and slowing down the system.

Example (C):

```
#include <stdlib.h>



int main() {

    int *arr = (int*) malloc(100 * sizeof(int));

    // allocated 100 integers on heap



    arr = NULL;  // Lost the reference, memory still reserved, but unreachable

    return 0;    // memory not freed → MEMORY LEAK
}
```

⚠️ Effect: The OS may not immediately crash, but long-running programs (like servers) can **slowly consume all RAM**.

🔹 Double Free Error

A **double free error** happens when:

- You free the same memory **twice**.
- After the first `free`, the memory is already returned to the system.
- Calling `free` again on the same pointer corrupts the memory allocator and can cause crashes or security vulnerabilities (attackers can exploit it to inject malicious code).

Example (C):

```
#include <stdlib.h>


int main() {
    int *p = (int*) malloc(sizeof(int));
    *p = 42;


    free(p);  // first free → OK
    free(p);  // second free → ERROR (undefined behavior)
    return 0;
}
```

⚠️ Effect: Could cause a **segmentation fault**, **heap corruption**, or be exploited for **arbitrary code execution**.

🔹 Key Difference

Aspect	Memory Leak	Double Free Error
What happens	Allocated memory is never released.	Freed memory is freed again.
Consequence	Wasted memory → program hogs RAM.	Crash, undefined behavior, or security exploit.
Severity	Performance issue (slows system).	Stability/security issue (can crash/exploit).

✅ Modern languages (Java, Python, Go, C#, etc.) avoid these problems by using **garbage collection** or **automatic memory management**.
✅ In C++, smart pointers (`unique_ptr`, `shared_ptr`) help reduce mistakes.
✅ In Rust, the **ownership system** guarantees at compile time that neither leaks nor double frees happen (without garbage collection).

Let's take a **simple task**: dynamically creating an array of integers, filling it, and printing the elements.

I'll show you how this looks in **C with raw pointers** vs. **Java/Python with safe references**.

🔹 C (with raw pointers, manual memory management)

```
#include <stdio.h>

#include <stdlib.h>
```

```
int main() {
    int n = 5;
    // Allocate memory dynamically for an array of n integers
    int *arr = (int*) malloc(n * sizeof(int));

    if (arr == NULL) {
        printf("Memory allocation failed\n");
        return 1;
    }

    // Fill array
    for (int i = 0; i < n; i++) {
        arr[i] = (i + 1) * 10;
    }

    // Print array
    for (int i = 0; i < n; i++) {
        printf("%d ", arr[i]);
    }

    // Free memory (manual step required)
    free(arr);

    return 0;
}
```

✅ Pros: Full control over memory.
⚠️ Cons: If you forget `free(arr);` → memory leak. If you call `free(arr);` twice → double free error.

🔹 Java (with references & garbage collection)

```
public class DynamicArrayExample {

    public static void main(String[] args) {

        int n = 5;

        // Allocate array (heap-managed, but no raw pointers)

        int[] arr = new int[n];


        // Fill array

        for (int i = 0; i < n; i++) {

            arr[i] = (i + 1) * 10;

        }


        // Print array

        for (int num : arr) {

            System.out.print(num + " ");

        }

        // No need to free memory → Garbage Collector handles it

    }

}
```

✅ No manual memory management.
⚠️ No direct pointer arithmetic (safer but less control).

🔹 Python (with list references)

```
n = 5

arr = [ (i+1) * 10 for i in range(n) ]  # dynamic list, heap-managed


for num in arr:

    print(num, end=" ")


# No free() → Python garbage collector handles memory
```

✅ Simpler and safer, everything is a reference under the hood.
⚠️ Lower control over raw memory compared to C.

✨ Key Difference

- **C:** You deal with **raw pointers** (`malloc`, `free`) → flexible but error-prone.
- **Java/Python:** You deal with **references** (safe handles to objects) → no pointer arithmetic, memory is automatic.

👉 Do you want me to **extend this comparison** to a case where we **resize the array** (like `realloc` in C vs. list expansion in Python vs. `ArrayList` in Java)? That's where the difference becomes even clearer.

More precisely, the "Write Once, Run Anywhere" principle assumes the presence of a compatible JVM on the target machine. However, differences in JVM implementation, OS-specific bugs, or hardware architecture issues (like endianness or missing features) can break this ideal behavior.

👉 Bonus point:
Even differences in the underlying file system (case sensitivity, file path formats) or available libraries can cause unexpected runtime issues despite using the same bytecode.

❓ **Why is a Java Applet considered obsolete, and what were the core security issues that led to its deprecation, despite running inside the browser with a JVM?**

Java Applets are considered obsolete primarily due to security concerns, declining browser support, and the emergence of more modern and secure web technologies.

Core Security Issues Leading to Deprecation:

- **Exploitable Vulnerabilities in the Java Plugin:** The Java browser plugin, which was necessary to run applets, became a frequent target for attackers. Numerous critical vulnerabilities were discovered over time, allowing malicious applets to bypass the Java security sandbox and execute arbitrary code on the client machine.
- **Lack of User Understanding and Misuse of Permissions:** Applets often required users to grant elevated permissions to perform certain actions (e.g., accessing local files). Many users, lacking a full understanding of the security implications, would readily grant these permissions, making their systems vulnerable to compromise by untrusted or malicious applets.
- **Complexity of the Security Model:** While Java's security sandbox was designed to isolate applets, the complexity of the model and the interactions with the browser environment created opportunities for security flaws and exploits. Maintaining a secure environment for applets proved challenging.
- **Rise of Alternative Technologies:** As web technologies like JavaScript, HTML5, and CSS evolved, they offered increasingly powerful and secure ways to create interactive web content, reducing the need for applets. These

technologies run directly within the browser's native security model, which is generally more robust and less prone to the types of vulnerabilities seen in external plugins.

Declining Browser Support:

Major web browsers like Chrome, Firefox, Edge, and Safari progressively phased out support for NPAPI (Netscape Plugin Application Programming Interface), the technology that allowed browser plugins like the Java plugin to function. This effectively rendered applets unusable in modern browsers.

❓ But… Java has both **primitive types (like int, char, boolean)** and **reference types (like objects)**.

👉 Why does having primitive types in Java technically break the pure object-oriented principle, and why didn't Java designers make everything an object like in some other OOP languages (e.g., Smalltalk)?

✅ **Why does it break pure OOP?**

- In pure OOP languages (like Smalltalk), everything is an object, and even numbers are objects.
- But in Java, primitives like `int`, `boolean`, `char`, and `double` are not objects—they are stored directly in memory for speed.

✅ **Why didn't Java designers make everything an object?**

- Performance:
    - Objects have extra memory overhead (object headers, references), and method calls on objects are slower than working with primitives directly.
    - For tasks requiring a huge number of number operations (e.g., numeric computations), primitives are way faster and use less memory.

👉 That’s why Java uses **primitive types for performance and simplicity**, and provides **Wrapper Classes** (like `Integer`, `Double`) when object behavior is needed (e.g., in Collections).

Reasons for not making everything an object:

- **Performance:** Primitive types are stored directly in memory, making operations on them significantly faster than operations on objects, which involve overhead like heap allocation, object headers, and potential garbage collection. For frequent arithmetic or logical operations, using primitives drastically improves performance.
- **Memory Efficiency:** Objects require more memory than primitives. Each object carries overhead (e.g., object header, padding), and storing many small objects would lead to increased memory consumption and potential cache misses. Primitives, being lightweight, reduce memory footprint.
- **Simplicity and Directness:** Primitives represent fundamental values directly, which can be simpler and more intuitive for certain operations compared to interacting with objects that encapsulate these values.

While Java provides wrapper classes (e.g., `Integer`, `Float`) to allow primitive values to be treated as objects when necessary (e.g., for collections or generic types), the core design choice to retain primitives was a pragmatic decision to balance the benefits of object-oriented programming with the need for efficient low-level operations.

Java objects inherently carry memory overhead beyond the space required for their declared fields. This overhead is primarily due to:

- **Object Header:** Every Java object, regardless of its content, includes a header. This header typically contains:
    - **Mark Word:** Stores metadata such as hash code, garbage collection flags, and synchronization information (e.g., lock status for `synchronized` blocks).
    - **Class Pointer (Klass Pointer):** A pointer to the object's class, which describes its type and methods.

    - **Array Length (for arrays):** If the object is an array, the header also includes an `int` field to store the array's length.
- **Padding/Alignment:** JVMs often align objects in memory to optimize performance, typically to multiples of 8 bytes. If an object's size (including its header and fields) is not a multiple of this alignment boundary, padding bytes are added to round up its size. This ensures efficient memory access and processing.

Factors Influencing Overhead:

- **JVM Implementation:** The exact size and structure of the object header can vary slightly between different JVM implementations (e.g., HotSpot, OpenJ9).
- **System Architecture (32-bit vs. 64-bit JVM):** The size of pointers (like the class pointer) differs between 32-bit and 64-bit JVMs, impacting the header size. 64-bit JVMs generally have larger pointers, leading to increased overhead. However, techniques like Compressed Ordinary Object Pointers (OOPs) can mitigate this on 64-bit systems by using 32-bit pointers for objects within a certain heap size.

Example:

A `java.lang.Long` object, which stores a single primitive `long` (8 bytes), will consume more than 8 bytes in memory due to its object header and potential padding. On a 64-bit system with typical JVM settings, it might consume around 24 bytes in total.

Implications:

This overhead is particularly significant when dealing with a large number of small objects, as the cumulative overhead can lead to substantial memory consumption. This is why using primitive types directly or specialized collections that store primitives can be more memory-efficient than using wrapper objects or generic collections when memory is a critical concern.

Look for more details-
https://dzone.com/articles/whats-wrong-with-small-objects-in-java