Chuwa - Java Backend 1208 Assignment

Delin "Angelina" Liang

12 Dec 2025

Question 1. What is the difference between JDK, JRE, and JVM?

ANSWER

JDK (Java Development Kit) - is the toolkit for Java-related development. It contains/includes JRE and other tools like javac (compiler) and jdb (debugger)

JRE (Java Runtime Environment) - is the software package that allows Java applications to run. It includes JVM core libraries and supporting files, but does not handle/ deals with compiling

JVM (Java Virtual Machine) - is an abstract execution environment that runs Java bytecode. It allows Java applications to be platform independent and could also be used by other languages

In summary:

- Use **JRE** to run Java applications
- Use **JDK** to develop and run Java applications
- **JVM** is to make Java applications platform-independent

Question 2. What are the main differences between primitive types and reference types?

ANSWER

Data & Storage:

- **Primitives:** The variable is the actual data value itself, which are directly stored on the stack.
- **References:** Points to a value, which the memory address/pointer is stored on the stack, and the actual data it is pointing to is stored on the heap

Comparing:

- **Primitives:** == could be used to do direct value compare
- **References:** Needs Objects' own `equals()` function to compare values, otherwise == is only comparing the memory address, not the actual value

Types:

- **Primitives:** Limited types, only have `byte, short, int, long, float, double, char, boolean`
- **References:** Includes a variety of types, including wrapper classes of primitives, collections, and user-defined classes/objects

Collections:

- **Primitives:** Cannot be directly stored in `Collections`. Requires the use of **wrapper classes** or **autoboxing**
- **References:** Can be stored directly in `Collections`

Question 3. Is Java pass-by-value or pass-by-reference? Explain.

ANSWER

Java is **pass by value**.

First, let's see the definition of "pass by value": "makes a copy in memory of the parameter's value, or a copy of the contents of the parameter."
Though in Java's nature, since most variables are reference types, it would make the assignment of the variable acts more similar to the definition of "pass by reference". But we also need to keep in mind that, in Java, the actual "value"of any variable on the stack is the actual value for primitive types or the reference for reference types. That is, for a reference variable, the value on the stack is the address on the heap at which the real object resides. When any variable is passed to a method in Java, the value of the variable on the stack is copied into a new variable inside the new method.

In summary, we say Java is *always* pass by value because for reference type variables, their "value"is the reference they are storing on the stack, which is also what is passed on for assignment operations

Question 4. Why are Strings immutable in Java?

ANSWER

Security:

- **Sensitive Data:** `String`s are widely used in storing sensitive information like network host details, environment data path value, device IDs, etc. Mutable variable in these cases could lead to serious vulnerabilities, since anyone with any code could change this information after passing a security check
- **Thread-Safety:** Immutability of `String`s allows them to be safely shared across multiple threads without the need for explicit synchronization, which simplifies concurrent programming
- String Constant Pool: The String Constant Pool could not work if strings are mutable, since it's common that multiple references would point to the same string in the SCP, and if a string could be changed, it would affect all the references pointing to this specific string, leading to unpredictable behavior

Performance:
Since `String`s are frequently used as keys in hash-based collections like `HashMap` and `HashSet`, its immutability guarantees that the `hashCode` of a string remains constant throughout its lifetime, thus significantly increasing the efficiency of Collections

Question 5. What is the String Constant Pool and how does it work?

ANSWER

The String Constant Pool (SCP) is **a dedicated memory region in the Heap where the Java Virtual Machine (JVM) stores string literals**. This allows commonly used string literals to be reused, which saves memory space and at the same time increases the efficiency of comparing

How it works?
In most cases, for creating string literals, SCP works in the following steps:

1 The user creates a string literal
2 JVM check for it in the constant pool
2.1 If it does not exist, create a new literal and add it to the pool
2.2 If it already exists, reuse the existing literal

For example, in the following case:

```
String s1 = "Cat";
String s2 = "Cat";
```

A string literal `"Cat"` would be created and added to the pool for `s1`, and for `s2`, it would be directed to point to the existing string `"Cat"` in the pool

But there are also some special cases:
When the user uses the `new` keyword, besides managing the constant pool, a new string object will also be added to the Heap, separated from the SCP. In this case, the reference is pointing to the value stored on the Heap, not SCP.

Similarly, for the example above:

```
String s1 = "Cat";
String s2 = "Cat";
String s3 = new String("Cat");
```

We would get the following result for reference comparison:

```
s1 == s2  // true
s1 == s3  // false
```

Question 6. What is the purpose of the final keyword in Java?

ANSWER

The `final` keyword is mostly used in cases where one wants to define something to be "immutable"in Java:

- **Variables:** When used in declarations, it defines constants
- **Methods:** Used to prevent child classes from overriding the methods
- **Classes:** Prevent inheritance, and make the class immutable

Question 7. What does the static keyword mean for variables or methods?

ANSWER

The `static` keyword is generally used to declare that a **variable/method belongs to the outer class**, instead of a single object.

In this case, all objects created based on that class share the same variable data/method data. Saving significant memory space and allowing the user to access the variable directly through the class name

Some Notes:

- Static variables are the **one and only** copy for that class
- Static methods can be called **without creating an object**
- Static variables exist as long as the class is loaded, persisting across all function calls and object instantiations. Any changes made will *always* affect *all* related objects
- Static methods cannot directly access non-static (instance) variables or methods because they don't know which object's data to use

Question 8. What is a static block and when does it run?

ANSWER

A static block is **a block of code declared within** `static...|`. Similar to static variables and methods, they belong to the class, not the objects. A simple example would be:

```
class Sample {
...
// Static block
static {
    staticVariable = 10;
    System.out.println("Static block executed.");
}
...
}
```

When does it run? Static blocks run **only once** during the **Class Loading Phase**

Question 9. Can a static method access non-static variables? Why or why not?

ANSWER

A static method CANNOT access non-static variables

Reasoning:
The main conflict comes from how static methods and non-static variables are tied to classes and objects:

- **Non-static variables** belong to a specific instance of a class (or to say a specific object). Every time you create a new object, that object gets its own unique copy of the non-static variables
- **Static methods** belong to the class itself. They can be called even if no objects of the class are created

Therefore, we can see that a running static method does not guarantee that an instance of the class exists, and the system also doesn't know which object variable you would want to access, since there might exists multiple instance for a class

Question 10. Describe the JVM loading order: static block, static variables, and constructor.

ANSWER

When a class is first **loaded**:

- **static variables** are initialized based on code order

- **static blocks** are executed based on code order

* Both these 2 steps run **only once** for each class

Then, when an **object is created**

- the **constructor** runs to initialize instance-level data

Question 11. What is the difference between a static variable and a constant defined as public static final?

ANSWER

The differences between a `static variable` and a `public static final` constant includes:

- **Mutability:** `static variable`s are mutable, while `public static final` constants are not
- **Assignment:** `static variable`s could be reassigned after initialization during program execution while `public static final` constants cannot
- **Access:** `static variable`s could be accessed through an instance (besides directly through class name), while `public static final` constants could only be accessed through class names

Question 12. Explain why immutable objects (like String) are thread-safe by design. Provide an example scenario.

ANSWER

In general, immutable objects are thread-safe because once they are created, they cannot have their internal state modified. Therefore, when in a multi-threaded environment, there are no issues such as data races that are caused by concurrent modifications.

Example:
Suppose we have a variable

```
public static final String BASE_URL = "https://api.example.com/v1/";
```

which is a variable that stores the base URL for a web service.
If we make it mutable here, and we have the following threads running at the same time:

```
t1: change the value to BASE_URL = "https://api.example.com/v2/";
t2: access URL
t3: read data from the URL
```

Since here t1 changes the value of `BASE_URL`, t2 and t3 would have unpredictable behavior such as accessing the old address `".../v1/"` or throwing error when reading data

Question 13. What problem does making a class final solve in terms of immutability and inheritance security?

ANSWER

By making a class `final`, it ensures **immutability** by:

- Prevents others from inheriting the class
- Prevents methods from being overridden by subclasses
- Keeps its fields to remain read-only

- Keeps the object's behavior to remain consistent at the language level

and keeps **inheritance security** by preventing:

- Subclasses overriding key methods that lead to security vulnerabilities
- Subclasses modifying internal logic can break the correctness of the parent class
- Subclasses introducing mutable fields can compromise thread safety
- Subclasses making the parent class unpredictable
- Fake polymorphism