Tricky Java Output Questions for 10+ Years Experience

Question 1: String Pool and Garbage Collection

java

```java
public class StringTest {
    public static void main(String[] args) {
        String s1 = new String("Hello");
        String s2 = s1.intern();
        String s3 = "Hello";

        System.out.println(s1 == s2);
        System.out.println(s2 == s3);
        System.out.println(s1 == s3);
        System.out.println(s1.equals(s3));
    }
}
```

Output:

```
false
true
false
true
```

Explanation:

- `s1` creates a new String object in heap
- `s2` gets reference to string pool entry (intern())
- `s3` directly references string pool
- `==` compares references, `equals()` compares content

Question 2: Integer Cache and Autoboxing

java

```
public class IntegerTest {
    public static void main(String[] args) {
        Integer a = 127;
        Integer b = 127;
        Integer c = 128;
        Integer d = 128;
        Integer e = new Integer(127);

        System.out.println(a == b);
        System.out.println(c == d);
        System.out.println(a == e);
        System.out.println(a.equals(e));
    }
}
```

Output:

```
true
false
false
true
```

Explanation:

- Integer cache works for values -128 to 127
- Beyond this range, new objects are created
- `new Integer()` always creates new object
- `equals()` compares values, not references

Question 3: Method Overloading and Varargs

```
java
```

```java
public class OverloadTest {
    static void test(Object o) {
        System.out.println("Object");
    }

    static void test(String s) {
        System.out.println("String");
    }

    static void test(String... s) {
        System.out.println("Varargs");
    }

    public static void main(String[] args) {
        test("Hello");
        test(null);
        test();
    }
}
```

Output:

```
String
String
Varargs
```

Explanation:

- Most specific method is chosen first
- `null` matches String (more specific than Object)
- Empty call matches varargs
- Varargs has lowest priority in overload resolution

Question 4: Exception Handling and Finally

java

```
public class ExceptionTest {
    static int test() {
        try {
            return 1;
        } catch (Exception e) {
            return 2;
        } finally {
            return 3;
        }
    }

    static int test2() {
        int x = 1;
        try {
            return x;
        } finally {
            x = 2;
        }
    }

    public static void main(String[] args) {
        System.out.println(test());
        System.out.println(test2());
    }
}
```

Output:

```
3
1
```

Explanation:

- `finally` return overrides try/catch return
- In test2(), return value is determined before finally executes
- Finally block cannot change already determined return values (primitives)

Question 5: Static Initialization Order

```
java
```

```java
class Parent {
    static {
        System.out.println("Parent static");
    }

    {
        System.out.println("Parent instance");
    }

    Parent() {
        System.out.println("Parent constructor");
    }
}

class Child extends Parent {
    static {
        System.out.println("Child static");
    }

    {
        System.out.println("Child instance");
    }

    Child() {
        System.out.println("Child constructor");
    }
}

public class InitTest {
    public static void main(String[] args) {
        Child c = new Child();
    }
}
```

Output:

```
Parent static
Child static
Parent instance
Parent constructor
Child instance
Child constructor
```

Explanation:

- Static blocks execute in inheritance hierarchy order
- Instance blocks and constructors execute parent first
- Static initialization happens only once per class

Question 6: Generics and Type Erasure

```
java

import java.util.*;

public class GenericsTest {
    public static void main(String[] args) {
        List<String> list1 = new ArrayList<>();
        List<Integer> list2 = new ArrayList<>();

        System.out.println(list1.getClass() == list2.getClass());
        System.out.println(list1.getClass().getGenericSuperclass());

        String[] arr = {"a", "b"};
        List<String> list3 = Arrays.asList(arr);
        arr[0] = "modified";
        System.out.println(list3.get(0));
    }
}
```

Output:

```
true
class java.util.AbstractList
modified
```

Explanation:

- Type erasure removes generic info at runtime
- Both lists have same runtime class
- Arrays.asList() returns view of original array
- Modifications to array reflect in list

Question 7: Anonymous Inner Class Capture

```
java
```

```
public class InnerTest {
    private int x = 10;

    public void test() {
        int y = 20;

        Runnable r = new Runnable() {
            public void run() {
                System.out.println(x);
                System.out.println(y);
            }
        };

        x = 30;
        // y = 40; // This would cause compilation error
        r.run();
    }

    public static void main(String[] args) {
        new InnerTest().test();
    }
}
```

Output:

```
30
20
```

Explanation:

- Instance variables can be modified after capture
- Local variables must be effectively final
- Anonymous class captures variable values at creation time for locals

---

Question 8: HashMap and Custom Objects

```
java
```

```
import java.util.*;

class Key {
    private String name;

    Key(String name) {
        this.name = name;
    }

    @Override
    public boolean equals(Object obj) {
        if (obj instanceof Key) {
            return name.equals(((Key) obj).name);
        }
        return false;
    }

    // hashCode() not overridden
}

public class HashMapTest {
    public static void main(String[] args) {
        Map<Key, String> map = new HashMap<>();
        Key k1 = new Key("test");
        Key k2 = new Key("test");

        map.put(k1, "value1");
        map.put(k2, "value2");

        System.out.println(map.size());
        System.out.println(map.get(k1));
        System.out.println(map.get(k2));
        System.out.println(k1.equals(k2));
    }
}
```

Output:

```
2
value1
value2
true
```

Explanation:

- Without overriding hashCode(), objects have different hash codes
- HashMap treats them as different keys despite equals() returning true
- Violates hashCode contract: equal objects must have equal hash codes

---

Question 9: Stream Operations and Laziness

```java
import java.util.*;
import java.util.stream.*;

public class StreamTest {
    public static void main(String[] args) {
        List<String> list = Arrays.asList("a", "b", "c", "d");

        Stream<String> stream = list.stream()
            .filter(s -> {
                System.out.println("Filter: " + s);
                return s.length() > 0;
            })
            .map(s -> {
                System.out.println("Map: " + s);
                return s.toUpperCase();
            });

        System.out.println("Stream created");

        List<String> result = stream.limit(2).collect(Collectors.toList());
        System.out.println(result);
    }
}
```

Output:

```
Stream created
Filter: a
Map: a
Filter: b
Map: b
[A, B]
```

Explanation:

- Stream operations are lazy until terminal operation

- Operations are applied element by element
- limit(2) stops processing after 2 elements
- No processing happens until collect() is called

Question 10: Multi-threading and Volatile

```java
public class VolatileTest {
    private static boolean flag = false;
    private static int count = 0;

    public static void main(String[] args) throws InterruptedException {
        Thread t1 = new Thread(() -> {
            while (!flag) {
                count++;
            }
            System.out.println("Count: " + count);
        });

        Thread t2 = new Thread(() -> {
            try {
                Thread.sleep(1000);
                flag = true;
                System.out.println("Flag set to true");
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
        });

        t1.start();
        t2.start();

        t1.join(5000);
        t2.join();

        if (t1.isAlive()) {
            System.out.println("Thread 1 still running - infinite loop!");
            System.exit(0);
        }
    }
}
```

Possible Output:

```
Flag set to true
Thread 1 still running - infinite loop!
```

Explanation:

- Without `volatile`, flag changes may not be visible across threads
- JIT compiler might optimize the loop assuming flag never changes
- This demonstrates the need for proper synchronization
- With `volatile boolean flag`, the program would terminate correctly

Key Concepts Tested:

1. **String Pool and Interning**
2. **Integer Caching (-128 to 127)**
3. **Method Overloading Resolution**
4. **Exception Handling Flow**
5. **Class Initialization Order**
6. **Type Erasure in Generics**
7. **Anonymous Class Variable Capture**
8. **HashMap Contract Violations**
9. **Stream Lazy Evaluation**
10. **Memory Visibility in Multithreading**

These questions test deep understanding of JVM behavior, memory management, concurrency, and subtle language features that experienced Java developers should master.