

برنامه نویسی پیشرفته

رفع اشکال: جلسه ۴



Stack vs. Heap memory

Primitive vs. reference types, autoboxing/unboxing

Equals vs. ==

wrapper classes

Basics of testing (JUnit) and debugging (IDE tools)

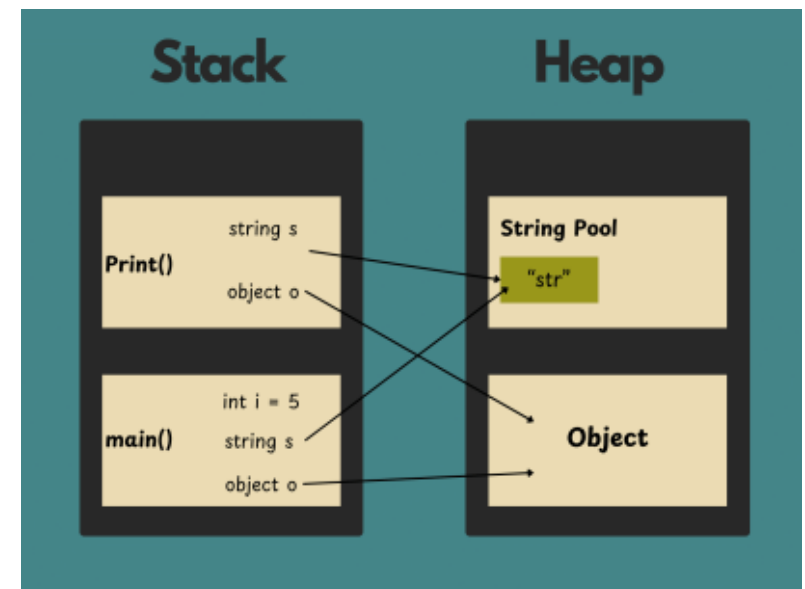
Stack vs. Heap memory

1. Stack Memory

- Used for local variables and method calls.
- Follows LIFO (Last In, First Out) order.
- Memory is automatically allocated and deallocated.
- Faster access but has limited size.
- Stores primitive variables and references to objects (not the objects themselves).

2. Heap Memory

- Used for storing objects (instances of classes).
- Managed by Garbage Collector.
- Slower access compared to Stack but has more space.
- All objects are stored in Heap, while their references are in Stack.



5 KEY POINTS

- **1. Global (Instance) Variables Are Not Stored in Stack**

In Java, instance variables (fields of a class) are stored in the Heap, while local variables are stored in the Stack.

- **Example:**

```
class Example {  
    int instanceVariable = 42; // Stored in Heap  
    void method() {  
        int localVariable = 10; // Stored in Stack  
    }  
}
```

■ 2. Stack Variables Do NOT Get Default Values, But Heap Variables Do

Local (Stack) variables must be initialized before use.

Heap variables (instance fields) get default values automatically.

- **Example:**

```
class Example {  
    int num; // Default: 0 (Stored in Heap)  
    void method() {  
        int localNum; // Not initialized (Stored in Stack)  
        System.out.println(localNum); // ERROR!  
    }  
}
```

■ 3. Static variables are stored in heap

Static variables are not stored in stack

- **Example:**

```
class Example {  
    static int staticVar = 100; // Stored in  
}
```

4. Difference between == and .equals()

- **Example:**

```
String s1 = new String("Java");  
String s2 = new String("Java");  
System.out.println(s1 == s2); // false (Different references)  
System.out.println(s1.equals(s2)); // true (Same value in Heap)
```

■ 5. Changing Objects in Heap Through Stack References

Since Stack only stores references, modifying an object through one reference affects all references pointing to the same object.

- **Example:**

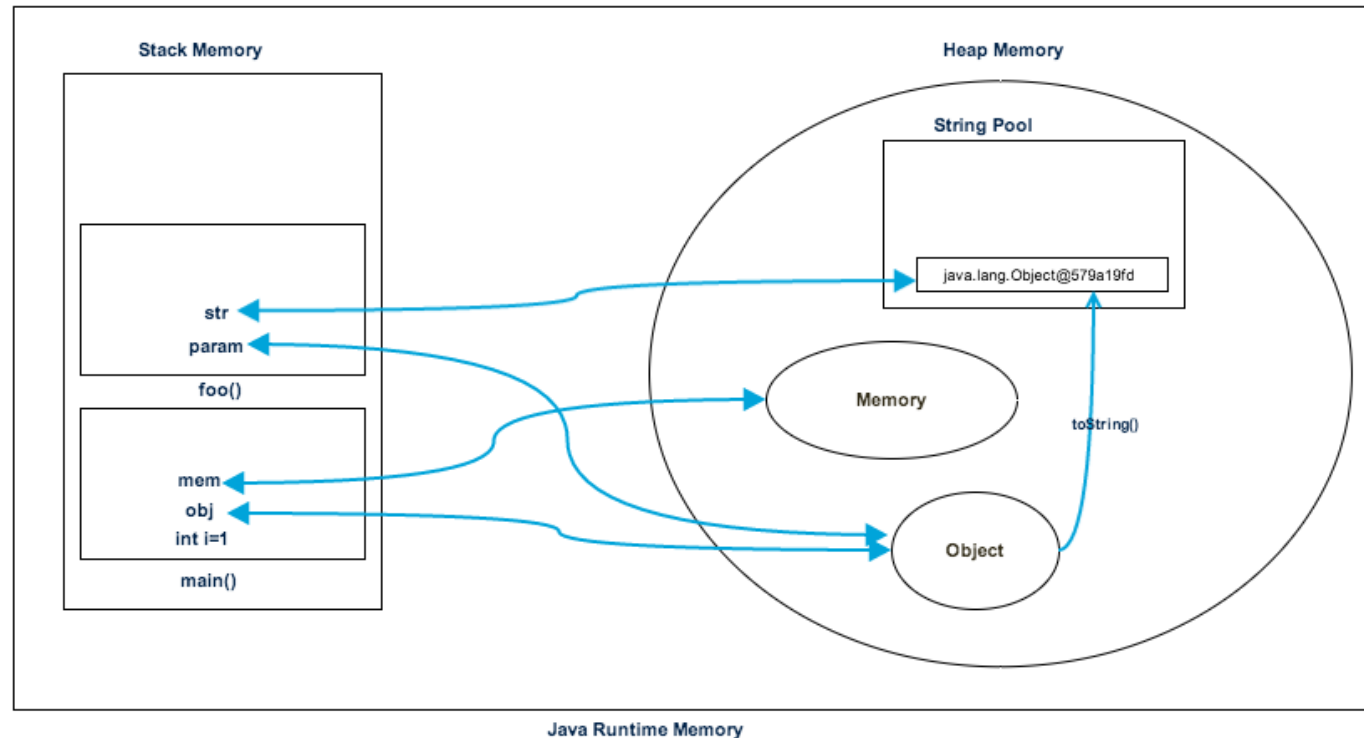
```
public class Test {  
    public static void main(String[] args) {  
        Data d1 = new Data(); // Object is in Heap, d1 is in Stack  
        d1.value = 10;  
        Data d2 = d1; // Copies the reference, NOT the object  
        d2.value = 20;  
        System.out.println(d1.value); // 20 (same object in Heap)  
    }  
}
```

QUESTIONS

- What will happen if a reference to an object in the Heap is lost in Stack, but the object is still referenced somewhere else in the Heap?

ANSWERS

The object will not be **garbage collected** immediately, as it still has an active reference elsewhere. It will remain in the Heap until all references to it are lost, and the **Garbage Collector** identifies it as unreachable.



MEMORY ALLOCATION EXAMPLES

❖ Arrays

■ `int[] arr = new int[2];`

• Stack:

A reference to the memory address in the heap is stored. This reference points to an array in the heap that contains 5 int values.

• Heap:

5 memory cells of type int are allocated consecutively.
The value of each cell is initialized to 0 by default.

■ `int[][] matrix = new int[2][1];`



• Heap:

First, an array of 3 references (for the 3 rows) is allocated in the heap. This array only stores references to arrays of 4 integers (representing the rows).

Then, for each of these rows, a 4-element array is allocated in the heap.

matrix	100C	FFF4
arr	1000	FFF8
		FFFC

	1000
0	1004
0	1008
	100C
1018	1010
1020	1014
	1018
0	101C
	1020
0	1024

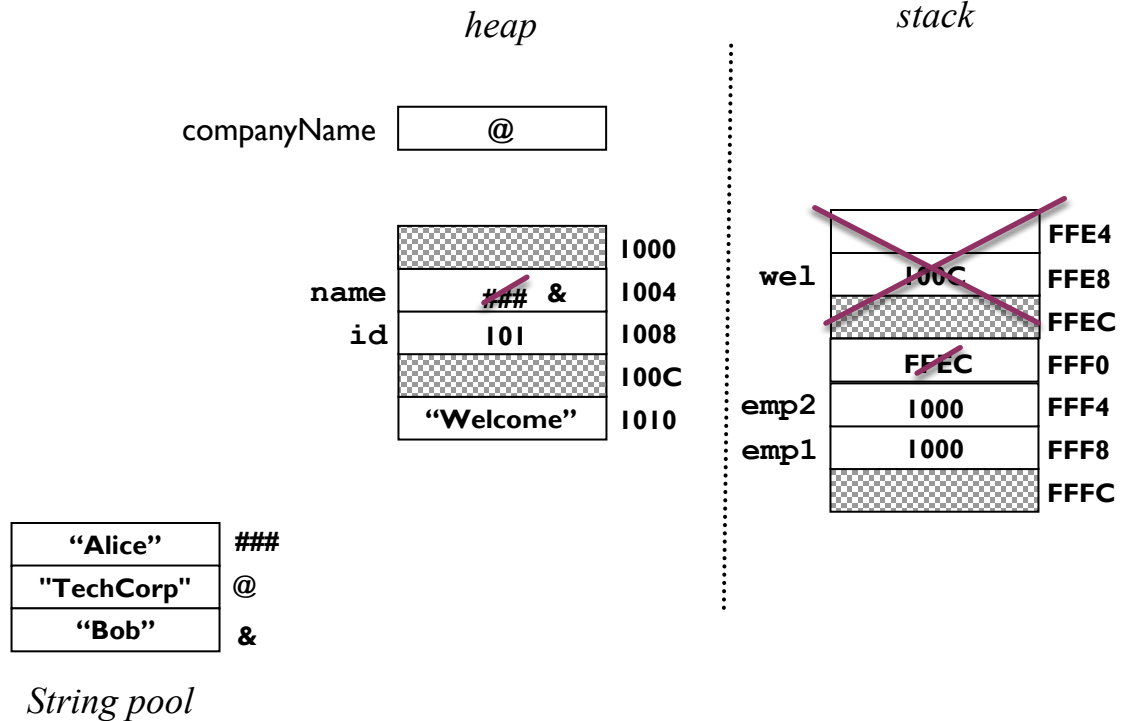


```
class Employee {  
    String name;  
    private int id;  
    static String companyName = "TechCorp"; // Static variable  
    public Employee(String name, int id) {  
        this.name = name;  
        this.id = id;  
    }  
    public void sayWelcome() {  
        String wel = new String(" Welcome " );  
        System.out.println(wel + name);  
    }  
}
```

```

public class MemoryAllocationDemo {
    public static void main(String[] args) {
        // Creating Employee object (Heap)
        Employee emp1 = new Employee("Alice", 101);
        // Copying the object (Shallow copy)
        Employee emp2 = emp1;
        // Changing emp1's details
        emp1.name = "Bob"; // emp2's name will also be
        "Bob" because it's a shallow copy
        emp1.sayWelcome();
    }
}

```



What if we had an array of employees?

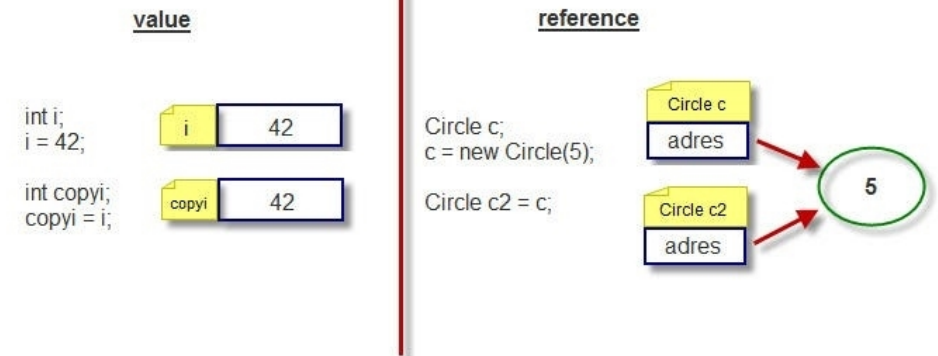
PRIMITIVE VS. REFERENCE TYPES

❖ Primitive types

- Primitive types in Java store the actual value of the data.
- These types includes: int, float, double, Boolean, char, byte, short, long.
- They have a fixed value and are not dependent on any object or structure.
- They are stored in the Stack and consume less memory.

❖ Reference types

- Reference types in Java store the address of an object in memory, not the object itself.
- These types can include classes (objects), arrays, and interfaces.
- The object is stored in the Heap, and its address is stored in the Stack.



AUTOBOXING AND UNBOXING

❖ Autoboxing

- Autoboxing happens when a primitive type is automatically converted to its corresponding reference type.
- This automatic conversion is handled by the Java compiler, so you don't need to do it manually.

```
int num = 10;
```

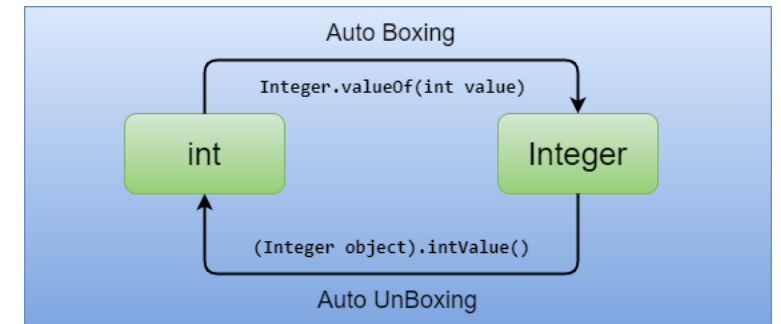
```
Integer obj = num; // Autoboxing: automatic conversion from int to Integer
```

❖ Unboxing

- Unboxing happens when a reference type is automatically converted to its corresponding primitive type.
- This conversion is also done automatically by the compiler.

```
Integer obj = 10;
```

```
int num = obj; // Unboxing: automatic conversion from Integer to int
```



QUESTIONS

- What will be the output of the following code, and why?

```
Integer num1 = 200;
```

```
Integer num2 = 200;
```

```
System.out.println(num1 == num2);
```

- Break down the code below and explain the process it follows

```
Integer a = 5;
```

```
Integer b = 10;
```

```
Integer sum = a + b; // Autoboxing and unboxing occur here.
```

ANSWERS

A) The output will be false.

Explanation: In Java, Integer objects are cached for values between -128 and 127. For values outside this range, new Integer objects are created, and therefore, they will not refer to the same memory address.

`num1 == num2` compares the memory addresses (references) of the two Integer objects, and since they are different objects (for the value 200), the result is false.

To compare the actual values of the two Integer objects, you should use the `equals()` method instead.

B)

Unboxing: `a` and `b` are unboxed from Integer objects to primitive `int` values (5 and 10).

Addition: The operation `a + b` is performed as a regular `int` operation (`5 + 10`), resulting in 15.

Re-boxing: The result 15 is boxed back into a new Integer object (`Integer sum = 15`).

EQUALS() VS ==

❖ == operator

- The == operator is used to compare primitive types by their values and reference types by their memory addresses (i.e., whether they refer to the same object in memory).
- For primitive types, == compares the actual values.
- For reference types (objects), == checks if two references point to the exact same object in memory (reference comparison).

◆ Example with primitives:

```
int a = 5;
```

```
int b = 5;
```

```
System.out.println(a == b); // true, compares values
```

◆ Example with reference types:

```
String str1 = new String("Hello");
```

```
String str2 = new String("Hello");
```

```
System.out.println(str1 == str2); // false, compares memory addresses (different objects)
```


EQUALS() METHOD

❖ Equals() Method

- The Equals() method is defined in the Object class, and it is used to compare the contents or values of objects, not their memory addresses.
- By default, equals() behaves like == (i.e., comparing references), but it can be overridden in a class to compare the actual data inside the objects.

◆ Example with reference types:

```
String str1 = new String("Hello");
```

```
String str2 = new String("Hello");
```

```
System.out.println(str1.equals(str2)); // true, compares values (contents of the strings)
```

WRAPPER CLASSES

- Wrapper classes are used to wrap primitive types into objects. Java provides a wrapper class for each primitive type.

Why use wrapper classes?

1. Object storage: Wrapper classes are useful when you need to store primitive values in collections like ArrayList, which can only store objects.
2. Utility methods: Wrapper classes come with useful methods, such as:
 - `parseInt()`, `parseDouble()` for parsing strings into primitive types.
 - `toString()`, `equals()`, and `compareTo()` for object manipulation.
3. Nullability: A wrapper class can be null, while a primitive type cannot. This is useful in some situations like database operations where a field might not have a value.

primitive	reference
int	Integer
char	Character
double	Double
Boolean	Boolean
byte	Byte

JUNIT AND DEBUGGING

❖ JUNIT

- **What it is:** JUnit is a popular testing framework for Java applications.
- **Purpose:** It helps developers write and run unit tests to ensure that individual parts of the application (like methods or functions) work correctly.
- **Annotations:** Key annotations in JUnit include `@Test` (marks a method as a test case), `@Before` (setup before tests), and `@After` (cleanup after tests).
- **Assertions:** Methods like `assertEquals()` and `assertTrue()` are used to check the correctness of test results.
- **JUnit Versions :** JUnit 4 and 5 are the most commonly used versions. JUnit 5 is the latest, offering more features and improvements over JUnit 4.

JUNIT AND DEBUGGING

❖ Debugging

- **What it is:** Debugging is the process of identifying and fixing issues (bugs) in a program.
- **Tools:** IDEs like IntelliJ IDEA or Eclipse offer built-in debuggers for step-by-step execution and variable inspection.
- **Common techniques:**
 - **Breakpoints:** Pause the execution of code at certain points to inspect the state.
 - **Watch variables:** Track changes to specific variables during runtime.
 - **Step Over/Into/Out:** Control the execution flow to analyze how the code is working or where it fails.
- **Logging:** Adding log statements can also help trace the flow and identify errors.

TIME TO CODE

Student Management System with classes for **Student**, **Course**, **Professor**, **Enrollment**

Student:

Represents a student with attributes like studentID, name, major, grades.

Course:

Represents a course with attributes like courseID, courseName, credits.

Professor:

Represents a professor with professorID, name, and a list of courses they teach.

Enrollment:

Tracks which students are enrolled in which courses.

EXERCISE FOR MEMORY ALLOCATION

```
public class MemoryAllocationExample {  
    static class Person {  
        String name;  
        int age;  
        public Person(String name, int age) {  
            this.name = name;  
            this.age = age;  
        }  
        public void displayInfo() {  
            System.out.println("Name: " + name + ",Age: " + age);  
        }  
    }  
}
```

```
public static void main(String[] args) {  
    int[] numbers = new int[5];  
    Person person1 = new Person("Alice", 25);  
    Person person2 = new Person("Bob", 30);  
    Person[] people = new Person[2];  
    people[0] = person1;  
    people[1] = person2;  
    System.out.println("Using object references:");  
    for (Person person : people) {  
        person.displayInfo();  
    }  
    System.out.println("\nUsing integer array:");  
    for (int i = 0; i < numbers.length; i++) {  
        numbers[i] = i * 10;  
        System.out.println("numbers[" + i + "] = " + numbers[i]);  
    }  
}
```

پایان

