Midtem-2

A comprehensive cheat sheet covering essential Java data structures, algorithms, and object-oriented concepts for exam preparation. Includes recursion, generics, sorting, Big-O notation, and more.

Recursion and Generics

Recursive Methods

Definition: A method that calls itself to solve a smaller subproblem. Must have a base case to stop recursion.

Example (Factorial):

```java
public int factorial(int n) {
  if (n == 0) {
    return 1; // Base case
  } else {
    return n * factorial(n - 1); // Recursive call
  }
}
```

Key Components:

- **Base Case:** Condition to terminate recursion.
- **Recursive Step:** Reduces the problem to a smaller instance.

Important considerations:

- Ensure that each recursive call moves closer to the base case.
- Avoid infinite recursion by carefully defining the base case.

Common Pitfalls: StackOverflowError if the recursion goes too deep (no base case or base case not reached).

Generic Classes

Definition: Classes that can work with different data types without being rewritten for each type. Use `<T>` (or other capital letters) to represent the type parameter.

Example (Generic LinkedList):

```java
public class LinkedList<T> {
  private Node<T> head;

  private static class Node<T> {
    T data;
    Node<T> next;

    Node(T data) {
      this.data = data;
      this.next = null;
    }
  }

  public void add(T data) { /* ... */ }
  public T get(int index) { /* ... */ }
}
```

Usage: `LinkedList<Integer> intList = new LinkedList<>();` `LinkedList<String> stringList = new LinkedList<>();`

Benefits: Type safety, code reusability, and reduced code duplication.

Sorting Algorithms

Recursive Sorting Algorithms

Recursive Insertion Sort:

```
public static void
recursiveInsertionSort(int arr[], int n)
{
    if (n <= 1)
        return;

    recursiveInsertionSort(arr, n-1);

    int last = arr[n-1];
    int j = n-2;

    while (j >= 0 && arr[j] > last)
    {
        arr[j+1] = arr[j];
        j--;
    }
    arr[j+1] = last;
}
```

Merge Sort: Divides the array into halves, recursively sorts them, and then merges the sorted halves.

Quick Sort: Selects a 'pivot' element and partitions the array around it, then recursively sorts the two partitions.

Time Complexity (Merge Sort): O(n log n) in all cases.

Time Complexity (Quick Sort): O(n log n) on average, O(n^2) in the worst case.

Tracing Recursive Calls

Understanding Call Stacks: Each recursive call adds a new frame to the call stack. Track the values of variables and the return addresses.

Example: Tracing `factorial(3)`:

1. `factorial(3)` calls `factorial(2)`
2. `factorial(2)` calls `factorial(1)`
3. `factorial(1)` calls `factorial(0)`
4. `factorial(0)` returns 1
5. `factorial(1)` returns 1 * 1 = 1
6. `factorial(2)` returns 2 * 1 = 2
7. `factorial(3)` returns 3 * 2 = 6

Debugging Tip: Use print statements or a debugger to step through the recursive calls and inspect the values of variables at each step.

Binary Search

Definition: Efficient search algorithm for sorted arrays. Repeatedly divides the search interval in half.

Algorithm:

1. Find the middle element.
2. If the middle element is the target, return its index.
3. If the target is less than the middle element, search the left half.
4. If the target is greater than the middle element, search the right half.

Time Complexity: O(log n)

Example:

```
public int binarySearch(int[] arr, int
target) {
    int left = 0;
    int right = arr.length - 1;

    while (left <= right) {
        int mid = left + (right - left)
/ 2; // Prevents overflow

        if (arr[mid] == target) {
            return mid;
        } else if (arr[mid] < target) {
            left = mid + 1;
        } else {
            right = mid - 1;
        }
    }

    return -1; // Target not found
}
```

Big-O and Object-Oriented Concepts

Big-O Notation

Definition: Describes the upper bound of an algorithm's time or space complexity. Represents the worst-case scenario.

Common Big-O Values:

- O(1) - Constant time
- O(log n) - Logarithmic time
- O(n) - Linear time
- O(n log n) - Linearithmic time
- O(n^2) - Quadratic time
- O(2^n) - Exponential time
- O(n!) - Factorial time

Key Considerations: Focuses on how the runtime or space requirements grow as the input size increases.

Examples:

- Accessing an element in an array by index: O(1)
- Searching for an element in a linked list: O(n)
- Sorting an array using merge sort: O(n log n)

Constructors

Default Constructor: A constructor with no parameters. If no constructor is defined, Java provides a default constructor.

Parameterized Constructor: A constructor with parameters to initialize object attributes.

`this` **Keyword:** Refers to the current object. Used to differentiate between instance variables and method parameters with the same name.

`super` **Keyword:** Refers to the parent class. Used to call the parent class's constructor or access parent class members.

Copy Constructor: Creates a new object that is a copy of an existing object.
* **Shallow Copy:** Copies the values of the object's fields. If the fields are references to other objects, only the references are copied.
* **Deep Copy:** Copies the values of the object's fields and recursively copies the objects referenced by those fields.

Example (Copy Constructor - Deep Copy):

```
public class MyClass {
  private int[] data;

  public MyClass(MyClass other) {
    this.data = new int[other.data.length];
    for (int i = 0; i < other.data.length; i++) {
      this.data[i] = other.data[i];
    }
  }
}
```

Inheritance and Polymorphism

Inheritance: A mechanism where a new class (child class) inherits properties and behaviors from an existing class (parent class). Use the `extends` keyword.

Reference Diagrams: Visual representations of object relationships and memory allocation.

Reference Semantics: Variables hold references to objects, not the objects themselves. Assigning one variable to another copies the reference, not the object.

Polymorphism: The ability of an object to take on many forms. Achieved through inheritance and interfaces.
* **Overriding:** Providing a specific implementation of a method in a subclass that is already defined in its superclass.
* **Overloading:** Defining multiple methods in the same class with the same name but different parameters.

Example (Inheritance):

```
class Animal {
  public void makeSound() {
    System.out.println("Generic animal sound");
  }
}

class Dog extends Animal {
  @Override
  public void makeSound() {
    System.out.println("Woof!");
  }
}
```

ArrayLists, Generics, and Expressions

ArrayList and Generics

ArrayList: A dynamic array that can grow or shrink in size. Part of the `java.util` package.

Adding Elements: `add(element)` appends to the end, `add(index, element)` inserts at a specific index.

Removing Elements: `remove(index)` removes the element at the specified index, `remove(Object o)` removes the first occurrence of the specified element.

Printing Elements: Iterate through the ArrayList and print each element.

Example:

```java
ArrayList<String> names = new ArrayList<>();
names.add("Alice");
names.add("Bob");
System.out.println(names); // Output: [Alice, Bob]
names.remove("Alice");
System.out.println(names); // Output: [Bob]
```

Benefits of Generics with ArrayList: Type safety, prevents runtime errors related to incorrect types.

Boxing and Unboxing

Boxing: Automatic conversion of a primitive type to its corresponding wrapper class object (e.g., `int` to `Integer`).

Unboxing: Automatic conversion of a wrapper class object to its corresponding primitive type (e.g., `Integer` to `int`).

Example:

```java
Integer intObj = 5; // Boxing
int num = intObj;    // Unboxing
```

Potential Issues: NullPointerException if unboxing a null wrapper object.

Instantiating Concrete Classes vs Interfaces

Concrete Class: A class that provides implementations for all its methods. Can be directly instantiated using `new`.

Interface: A blueprint of a class. Contains only abstract methods (methods without implementation) and constants. Cannot be directly instantiated, but can be implemented by classes.

Example:

```java
interface MyInterface {
  void doSomething();
}

class MyClass implements MyInterface {
  @Override
  public void doSomething() {
    System.out.println("Doing something");
  }
}

MyClass obj = new MyClass(); // Valid
// MyInterface iface = new MyInterface(); // Invalid - cannot instantiate an interface
MyInterface iface = new MyClass(); // Valid - instantiating a class that *implements* the interface
```