Java Interview Questions — Part 4

Multithreading, Collections, OOP & Design Patterns, Exception Handling, System Design, Pattern Programs (Q99 – Q122)

Prepared for Shammi Bharti

Contents

Section A — Multithreading & Concurrency (Q99 to Q106)

A **Thread** is a separate path of execution that can run alongside other threads in the same program. This topic is asked a lot in SDET interviews because of parallel test execution.

Q99. Producer Consumer Problem

What is being asked: A classic concurrency problem — one thread (Producer) creates data and puts it into a shared list/queue, while another thread (Consumer) removes and uses that data. They must not step on each other (e.g., consumer should not try to remove from an empty list).

Logic:

- Use a shared queue with a maximum capacity.
- The Producer waits (`wait()`) if the queue is full; otherwise it adds an item and notifies (`notify()`) any waiting consumer.
- The Consumer waits if the queue is empty; otherwise it removes an item and notifies any waiting producer.
- `synchronized` blocks are used so only one thread touches the shared queue at a time.

```
import java.util.LinkedList;
import java.util.Queue;

public class ProducerConsumer {
    static final int CAPACITY = 5;
    static Queue<Integer> queue = new LinkedList<>();

    public static void main(String[] args) {
        Thread producer = new Thread(ProducerConsumer::produce);
        Thread consumer = new Thread(ProducerConsumer::consume);
        producer.start();
        consumer.start();
    }

    static void produce() {
        int value = 0;
        while (true) {
            synchronized (queue) {
                while (queue.size() == CAPACITY) {
                    try { queue.wait(); } catch (InterruptedException e) {}
                }
                queue.add(value);
```

```
                System.out.println("Produced: " + value);
                value++;
                queue.notify();
            }
        }
    }

    static void consume() {
        while (true) {
            synchronized (queue) {
                while (queue.isEmpty()) {
                    try { queue.wait(); } catch (InterruptedException e) {}
                }
                int value = queue.poll();
                System.out.println("Consumed: " + value);
                queue.notify();
            }
        }
    }
}
```

Q100. Create Thread Using Thread Class

Logic: Make a class that `extends Thread` and override its `run()` method with the code you want to run in parallel. Call `.start()` (never call `run()` directly — that would just run it normally on the current thread, not a new one).

```
class MyThread extends Thread {
    public void run() {
        System.out.println("Thread running using Thread class");
    }
}

public class CreateThreadExample {
    public static void main(String[] args) {
        MyThread t1 = new MyThread();
        t1.start();
    }
}
```

Q101. Create Thread Using Runnable

Logic: Implement the `Runnable` interface and define your code inside `run()`. Then pass that `Runnable` object into a `new Thread(...)` and call `.start()`. This approach is usually preferred over extending `Thread`, because Java does not support multiple

inheritance — implementing an interface keeps your class free to extend something else if needed.

```
class MyRunnable implements Runnable {
    public void run() {
        System.out.println("Thread running using Runnable interface");
    }
}

public class CreateRunnableExample {
    public static void main(String[] args) {
        Thread t1 = new Thread(new MyRunnable());
        t1.start();
    }
}
```

Q102. Synchronization Example

What is being asked: Show how to prevent two threads from accessing shared data (like a bank balance) at the same time, which could cause incorrect results.

Logic: Mark the critical method with the `synchronized` keyword. Java will then make sure only one thread can be executing that method on a given object at any time — other threads must wait their turn.

```
class Counter {
    int count = 0;

    synchronized void increment() {
        count++;
    }
}

public class SynchronizationExample {
    public static void main(String[] args) throws InterruptedException {
        Counter counter = new Counter();

        Runnable task = () -> {
            for (int i = 0; i < 1000; i++) {
                counter.increment();
            }
        };

        Thread t1 = new Thread(task);
        Thread t2 = new Thread(task);
        t1.start();
        t2.start();
        t1.join();
```

```
        t2.join();

        // result is always 2000, thanks to synchronized
        System.out.println("Final count: " + counter.count);
    }
}
```

Q103. Callable and Future Example

What is being asked: `Runnable` cannot return a value or throw a checked exception. `Callable` solves this — it can return a result, which you collect later using a `Future`.

Logic: Submit a `Callable` task to an `ExecutorService`. This immediately returns a `Future` object (a placeholder for the result that will be ready later). Call `future.get()` when you actually need the result — this will wait if the task is not finished yet.

```
import java.util.concurrent.*;

public class CallableFutureExample {
    public static void main(String[] args) throws Exception {
        ExecutorService executor = Executors.newSingleThreadExecutor();

        Callable<Integer> task = () -> {
            Thread.sleep(1000);
            return 42;
        };

        Future<Integer> future = executor.submit(task);
        System.out.println("Doing other work while task runs...");

      Integer result = future.get();  // waits here until the result is ready
        System.out.println("Result: " + result);

        executor.shutdown();
    }
}
```

Q104. ExecutorService Example

What is being asked: Instead of manually creating and managing individual `Thread` objects, use a thread pool to efficiently reuse a fixed number of threads for many tasks.

Logic: Create a thread pool with `Executors.newFixedThreadPool(n)`. Submit tasks using `execute()` or `submit()`. The pool automatically assigns each task to an available thread. Always call `shutdown()` when done, so the program can exit cleanly.

```
import java.util.concurrent.ExecutorService;
import java.util.concurrent.Executors;

public class ExecutorServiceExample {
    public static void main(String[] args) {
        ExecutorService executor = Executors.newFixedThreadPool(3);

        for (int i = 1; i <= 5; i++) {
            int taskId = i;
            executor.execute(() -> {
                String threadName = Thread.currentThread().getName();
               System.out.println("Task " + taskId + " executed by " + threadName);
            });
        }

        executor.shutdown();
    }
}
```

Q105. ConcurrentHashMap Example

What is being asked: Show how to use a thread-safe HashMap that multiple threads can read/write at the same time without corrupting data (unlike a normal `HashMap`, which is not thread-safe).

Logic: `ConcurrentHashMap` internally divides the map into segments and only locks the small portion being modified, rather than locking the entire map (unlike the older `Hashtable`, which locks everything). This makes it both safe and fast for multi-threaded use.

```
import java.util.concurrent.ConcurrentHashMap;

public class ConcurrentHashMapExample {
    public static void main(String[] args) {
        ConcurrentHashMap<String, Integer> map = new ConcurrentHashMap<>();

        Runnable task = () -> {
            for (int i = 0; i < 1000; i++) {
                map.merge("count", 1, Integer::sum);
            }
        };

        Thread t1 = new Thread(task);
        Thread t2 = new Thread(task);
        t1.start();
        t2.start();
```

```
        try {
            t1.join();
            t2.join();
        } catch (InterruptedException e) {}

        // safely reaches 2000
        System.out.println("Final count: " + map.get("count"));
    }
}
```

Q106. Avoid Deadlocks

What is being asked: A deadlock happens when two or more threads are stuck forever, each waiting for a resource (lock) held by the other. Explain how to prevent this.

Logic / How to avoid it:

- **Lock ordering:** Always acquire multiple locks in the same, consistent order across all threads, so no two threads can be holding one lock while waiting for another in a circular way.
- **Lock timeout:** Use `tryLock()` with a timeout instead of waiting forever for a lock.
- **Avoid nested locks** where possible, or keep the locked sections as small as possible.

```
public class DeadlockExample {
    static final Object lockA = new Object();
    static final Object lockB = new Object();

    public static void main(String[] args) {
        // Both threads acquire locks in the SAME order to avoid deadlock
        Thread t1 = new Thread(() -> {
            synchronized (lockA) {
                System.out.println("Thread 1 holds Lock A");
                synchronized (lockB) {
                    System.out.println("Thread 1 holds Lock A and Lock B");
                }
            }
        });

        Thread t2 = new Thread(() -> {
            synchronized (lockA) {     // same order as t1: A then B
                System.out.println("Thread 2 holds Lock A");
                synchronized (lockB) {
                    System.out.println("Thread 2 holds Lock A and Lock B");
                }
            }
```

```
        });

        t1.start();
        t2.start();
    }
}
```

Interview tip: If both threads had locked `lockA` and `lockB` in *different* orders, a deadlock could occur. Keeping the order consistent is the simplest fix.

Section B — Collections Framework (Q107 to Q111)

Q107. HashMap Internal Working

What is being asked: Explain how `HashMap` stores and retrieves data internally — a very common theory question.

Explanation (in simple words):

- A `HashMap` internally uses an array of "buckets."
- When you call `put(key, value)`, Java calculates a `hashCode()` for the key, and uses that to decide which bucket it goes into.
- If two different keys land in the same bucket (called a "collision"), they are stored together as a small linked list inside that bucket (in modern Java, if a bucket gets too many collisions, it converts to a balanced tree for better performance).
- When you call `get(key)`, Java again calculates the hash, jumps straight to the right bucket, then compares keys using `equals()` to find the exact match.
- This is why `HashMap` lookups are very fast on average — close to `O(1)` — instead of having to check every single entry.

```
import java.util.HashMap;

public class HashMapDemo {
    public static void main(String[] args) {
        HashMap<String, Integer> map = new HashMap<>();
        map.put("apple", 1);
        map.put("banana", 2);

        // internally: hashCode("apple") decides the bucket index
        System.out.println(map.get("apple"));
    }
}
```

Q108. HashSet Internal Working

What is being asked: Explain how `HashSet` works internally.

Explanation: A `HashSet` is internally built on top of a `HashMap`. Every value you add to a `HashSet` is actually stored as a **key** in a hidden internal `HashMap`, with a fixed dummy value (a constant `Object`) attached to every key. This is exactly why a `HashSet` does not allow duplicates and does not guarantee order — it inherits these properties directly from `HashMap`.

```
import java.util.HashSet;

public class HashSetDemo {
    public static void main(String[] args) {
        HashSet<String> set = new HashSet<>();
        set.add("apple");
```

```
        set.add("apple");   // ignored, already exists as a "key"

        System.out.println(set.size());   // prints 1
    }
}
```

Q109. Remove Repeated Elements from ArrayList

Logic: Convert the list into a `LinkedHashSet` (removes duplicates while keeping order), then convert it back into a list.

```
import java.util.*;

public class RemoveDuplicatesFromArrayList {
    public static void main(String[] args) {
     List<Integer> list = new ArrayList<>(Arrays.asList(1, 2, 2, 3, 4, 4, 5));

        List<Integer> result = new ArrayList<>(new LinkedHashSet<>(list));

        System.out.println(result);
    }
}
```

Output: `[1, 2, 3, 4, 5]`

Q110. Difference Between ArrayList and LinkedList

Explanation (theory, in simple words):

Point	ArrayList	LinkedList
Internal structure	Backed by a resizable array	Backed by a doubly-linked chain of nodes
Access by index `get(i)`	Fast — `O(1)`, jumps straight there	Slow — `O(n)`, must walk from the start
Insert/Delete in the middle	Slow — `O(n)`, must shift elements	Fast — `O(1)` once you have the node, just relink pointers
Memory usage	Less overhead per element	More overhead (stores extra `next`/`prev` pointers per node)
Best used when	You read/access elements a lot	You insert/delete a lot, especially at the ends

```
import java.util.ArrayList;
import java.util.LinkedList;
```

```
public class ArrayListVsLinkedList {
    public static void main(String[] args) {
     ArrayList<Integer> arrayList = new ArrayList<>();  // fast random access
     LinkedList<Integer> linkedList = new LinkedList<>(); // fast insert/delete

        arrayList.add(1);
        linkedList.add(1);
    }
}
```

Q111. Difference Between HashMap and ConcurrentHashMap

Explanation (theory, in simple words):

Point	HashMap	ConcurrentHashMap
Thread safety	Not thread-safe — multiple threads can corrupt data	Thread-safe — designed for multi-threaded use
Locking	No locking at all	Locks only a small segment/bucket being modified, not the whole map
Null keys/values	Allows one null key and multiple null values	Does not allow null keys or null values at all
Performance in multi-threading	Will break or behave unpredictably	Safe and reasonably fast under concurrent access
When to use	Single-threaded programs	Multi-threaded programs, like parallel test execution frameworks

```
import java.util.HashMap;
import java.util.concurrent.ConcurrentHashMap;

public class HashMapVsConcurrentHashMap {
    public static void main(String[] args) {
        // use in single-threaded code
        HashMap<String, Integer> normalMap = new HashMap<>();

        // use when multiple threads write to it
      ConcurrentHashMap<String, Integer> safeMap = new ConcurrentHashMap<>();
    }
}
```

Section C — OOP & Design Patterns (Q112 to Q117)

Q112. Singleton Design Pattern

What is being asked: Ensure that a class can have **only one object (instance)** created in the entire application, and provide a single global way to access it. Commonly used for things like a configuration manager, a logger, or a WebDriver instance in a test framework.

Logic:

- Make the constructor `private` so no other class can create a new object directly using `new`.
- Keep a `private static` reference to the single instance.
- Provide a `public static` method (commonly named `getInstance()`) that creates the object only the first time it is called, and returns the same object every time after that.

```
public class Singleton {
    private static Singleton instance;

    private Singleton() {
        // private constructor - prevents "new Singleton()" from outside
    }

    public static Singleton getInstance() {
        if (instance == null) {
            instance = new Singleton();
        }
        return instance;
    }
}
```

Q113. Factory Design Pattern

What is being asked: Create objects without exposing the exact creation logic to the caller — the caller just asks for a "type," and a Factory class decides which actual object to build and return.

Logic: Define a common interface/abstract class for the product family. Create a `Factory` class with a method that takes some input (like a type name) and returns the correct concrete object based on that input.

```
interface Shape {
    void draw();
}

class Circle implements Shape {
    public void draw() { System.out.println("Drawing Circle"); }
```

```
}

class Square implements Shape {
    public void draw() { System.out.println("Drawing Square"); }
}

class ShapeFactory {
    Shape getShape(String type) {
        if (type.equalsIgnoreCase("circle")) return new Circle();
        if (type.equalsIgnoreCase("square")) return new Square();
        return null;
    }
}

public class FactoryPatternDemo {
    public static void main(String[] args) {
        ShapeFactory factory = new ShapeFactory();
        Shape shape = factory.getShape("circle");
        shape.draw();
    }
}
```

Q114. Builder Design Pattern

What is being asked: Construct a complex object step by step, especially useful when an object has many optional fields (avoids having a constructor with 10 parameters, most of which might be empty/default).

Logic: Create a nested `Builder` class inside the main class. Each "set" method on the Builder returns the Builder itself (`this`), which allows chaining calls together. A final `build()` method creates the actual object using the values collected.

```
public class User {
    private String name;
    private int age;
    private String email;

    private User(Builder builder) {
        this.name = builder.name;
        this.age = builder.age;
        this.email = builder.email;
    }

    public static class Builder {
        private String name;
        private int age;
        private String email;
```

```
        public Builder setName(String name) { this.name = name; return this; }
        public Builder setAge(int age) { this.age = age; return this; }
      public Builder setEmail(String email) { this.email = email; return this; }

        public User build() {
            return new User(this);
        }
    }
}

class BuilderPatternDemo {
    public static void main(String[] args) {
        User user = new User.Builder()
                        .setName("Shammi")
                        .setAge(28)
                        .setEmail("shammi@example.com")
                        .build();
    }
}
```

Q115. Observer Design Pattern

What is being asked: Set up a system where one object (the “Subject”) notifies a list of other objects (the “Observers”) automatically whenever something changes — like a YouTube channel notifying all its subscribers when a new video is uploaded.

Logic: The Subject keeps a list of Observers. It provides methods to `addObserver()` and `notifyObservers()`. When `notifyObservers()` is called, it loops through the list and calls an `update()` method on every registered Observer.

```
import java.util.ArrayList;
import java.util.List;

interface Observer {
    void update(String message);
}

class Subscriber implements Observer {
    String name;
    Subscriber(String name) { this.name = name; }
    public void update(String message) {
        System.out.println(name + " received notification: " + message);
    }
}

class YouTubeChannel {
```

```
    List<Observer> subscribers = new ArrayList<>();

    void addObserver(Observer observer) {
        subscribers.add(observer);
    }

    void notifyObservers(String message) {
        for (Observer observer : subscribers) {
            observer.update(message);
        }
    }
}

public class ObserverPatternDemo {
    public static void main(String[] args) {
        YouTubeChannel channel = new YouTubeChannel();
        channel.addObserver(new Subscriber("Shammi"));
        channel.addObserver(new Subscriber("Rahul"));

        channel.notifyObservers("New video uploaded!");
    }
}
```

Q116. Override a Non-Static Method

What is being asked: Demonstrate **method overriding** — when a child class provides its own specific version of a method already defined in its parent class.

Logic: The method in the child class must have the exact same name, return type, and parameters as the parent's method. Use the `@Override` annotation (good practice, helps catch mistakes at compile time). When you call the method on a child object, Java runs the child's version — this is called **runtime polymorphism**.

```
class Animal {
    void makeSound() {
        System.out.println("Animal makes a sound");
    }
}

class Dog extends Animal {
    @Override
    void makeSound() {
        System.out.println("Dog barks");
    }
}

public class MethodOverridingDemo {
```

```
    public static void main(String[] args) {
        Animal animal = new Dog();   // parent reference, child object
        // Dog's version runs - this is runtime polymorphism
        animal.makeSound();
    }
}
```

Output: `Dog barks`

Q117. Change Order of Method Parameters (Method Overloading)

What is being asked: This demonstrates **method overloading** — having multiple methods with the **same name** in the same class, but with different parameter lists (different number, type, or order of parameters). Java decides which one to call based on the arguments you pass.

Logic: Define several methods with identical names but different parameter signatures. Java picks the matching version at compile time based on the arguments used in the call — this is called **compile-time polymorphism**.

```
public class MethodOverloadingDemo {

    void show(int a, String b) {
        System.out.println("int first: " + a + ", " + b);
    }

    void show(String b, int a) {
        System.out.println("String first: " + b + ", " + a);
    }

    public static void main(String[] args) {
        MethodOverloadingDemo obj = new MethodOverloadingDemo();
        obj.show(5, "Hello");     // calls the first version
       obj.show("Hello", 5);    // calls the second version (order differs)
    }
}
```

Section D — Exception Handling (Q118)

Q118. Implement Custom Exception in Java

What is being asked: Create your own exception class for situations specific to your application, instead of always using Java's built-in exceptions. Example: an `InvalidAgeException` for a registration form.

Logic:

- Create a class that `extends Exception` (for a checked exception, which the caller must handle) or `extends RuntimeException` (for an unchecked exception).
- Provide a constructor that accepts a message and passes it to the parent class using `super(message)`.
- `throw` your custom exception wherever the invalid condition occurs, and handle it using `try-catch` where needed.

```
class InvalidAgeException extends Exception {
    public InvalidAgeException(String message) {
        super(message);
    }
}

public class CustomExceptionDemo {
    static void checkAge(int age) throws InvalidAgeException {
        if (age < 18) {
            String msg = "Age must be 18 or above. Given age: " + age;
            throw new InvalidAgeException(msg);
        }
        System.out.println("Age is valid.");
    }

    public static void main(String[] args) {
        try {
            checkAge(15);
        } catch (InvalidAgeException e) {
            System.out.println("Caught exception: " + e.getMessage());
        }
    }
}
```

Output: `Caught exception: Age must be 18 or above. Given age: 15`

Section E — System Design / LLD (Q119)

Q119. Design a Vending Machine

What is being asked: A Low-Level Design (LLD) question. You need to design the classes and states for a simple vending machine that accepts money, lets the user select a product, dispenses it, and gives change.

Logic / Approach (this is the answer interviewers are really looking for):

- **Identify the entities:** `Product` (name, price, quantity), `Inventory` (holds all products), `VendingMachine` (the main controller), and the machine's `State`.
- **Identify the states** the machine moves through: `Idle` -> `HasMoney` (waiting for selection) -> `Dispensing` -> back to `Idle`. This is a great place to mention the **State Design Pattern** if asked about patterns.
- **Identify the main actions:** `insertCoin()`, `selectProduct()`, `dispenseProduct()`, `returnChange()`.
- Keep responsibilities separate: `Inventory` only manages stock, `VendingMachine` only manages flow/state, and `Product` only stores data — this follows the **Single Responsibility Principle**.

```
import java.util.HashMap;
import java.util.Map;

class Product {
    String name;
    double price;
    int quantity;

    Product(String name, double price, int quantity) {
        this.name = name;
        this.price = price;
        this.quantity = quantity;
    }
}

class Inventory {
    Map<String, Product> products = new HashMap<>();

    void addProduct(Product product) {
        products.put(product.name, product);
    }

    Product getProduct(String name) {
        return products.get(name);
    }

    void reduceStock(String name) {
        Product product = products.get(name);
```

```
        if (product != null && product.quantity > 0) {
            product.quantity--;
        }
    }
}

class VendingMachine {
    Inventory inventory = new Inventory();
    double insertedAmount = 0;

    void insertCoin(double amount) {
        insertedAmount += amount;
     System.out.println("Inserted: " + amount + ", Total: " + insertedAmount);
    }

    void selectProduct(String name) {
        Product product = inventory.getProduct(name);

        if (product == null || product.quantity == 0) {
            System.out.println("Product unavailable.");
            return;
        }
        if (insertedAmount < product.price) {
            System.out.println("Insufficient amount. Please insert more.");
            return;
        }

        dispenseProduct(product);
    }

    void dispenseProduct(Product product) {
        inventory.reduceStock(product.name);
        double change = insertedAmount - product.price;
        insertedAmount = 0;

        System.out.println("Dispensing: " + product.name);
        if (change > 0) {
            System.out.println("Returning change: " + change);
        }
    }
}

public class VendingMachineDemo {
    public static void main(String[] args) {
        VendingMachine machine = new VendingMachine();
        machine.inventory.addProduct(new Product("Coke", 25, 5));

        machine.insertCoin(30);
```

```
        machine.selectProduct("Coke");
    }
}
```

Section F — Pattern Printing Programs (Q120 to Q122)

These questions test your control over nested loops, which is a basic but important skill interviewers check early on.

Q120. Printing Star Patterns

What is being asked: Print a simple right-angled triangle made of stars:

```
*
* *
* * *
* * * *
* * * * *
```

Logic: Use two loops — the outer loop controls the row number, and the inner loop prints stars equal to the current row number.

```
public class StarPattern {
    public static void main(String[] args) {
        int rows = 5;
        for (int i = 1; i <= rows; i++) {
            for (int j = 1; j <= i; j++) {
                System.out.print("* ");
            }
            System.out.println();
        }
    }
}
```

Q121. Printing Number Patterns

What is being asked: Print a number triangle where each row repeats its row number:

```
1
2 2
3 3 3
4 4 4 4
5 5 5 5 5
```

Logic: Same structure as Q120, but instead of printing a fixed `"*"`, print the outer loop variable `i` itself.

```
public class NumberPattern {
    public static void main(String[] args) {
        int rows = 5;
        for (int i = 1; i <= rows; i++) {
            for (int j = 1; j <= i; j++) {
```

```
                System.out.print(i + " ");
            }
            System.out.println();
        }
    }
}
```

Q122. Printing Pyramid Patterns

What is being asked: Print a centred pyramid of stars:

```
    *
   * * *
  * * * * *
 * * * * * * *
* * * * * * * * *
```

Logic: For each row, you need two things: leading spaces (which decrease as rows increase) and stars (which increase as `2*row - 1`).

- Spaces needed in row `i` (out of `rows` total) = `rows - i`.
- Stars needed in row `i` = `2*i - 1`.

```
public class PyramidPattern {
    public static void main(String[] args) {
        int rows = 5;

        for (int i = 1; i <= rows; i++) {
            // print leading spaces
            for (int s = 1; s <= rows - i; s++) {
               System.out.print("  ");   // two spaces to keep stars aligned
            }
            // print stars
            for (int j = 1; j <= (2 * i - 1); j++) {
                System.out.print("* ");
            }
            System.out.println();
        }
    }
}
```

Final Notes

That covers all 122 questions from your list — Strings, Numbers, Arrays, Searching & Sorting, Matrix, Linked List, Stack & Queue, Binary Tree, Graph, Multithreading, Collections, OOP & Design Patterns, Exception Handling, System Design, and Pattern Printing.

How to revise effectively before your interview:

1. Don’t memorise the code character by character — memorise the **logic steps** for each question. If you understand the steps, you can rebuild the code from scratch even under pressure.
2. Practice writing 5-10 of these programs daily **on paper or a plain text editor** (no auto-complete), since that is closer to a real whiteboard/online-assessment interview.
3. Pay special attention to the **theory questions** (HashMap internal working, ArrayList vs LinkedList, HashMap vs ConcurrentHashMap, BFS vs DFS) — these often decide if the interviewer feels you “really understand” Java, not just copy-paste code.
4. For Design Patterns and the Vending Machine LLD question, focus on being able to **explain your design out loud** — interviewers care more about your reasoning than perfect code in these rounds.

All the best for your Automation QA / SDET interviews, Shammi!