Java Exam Preparation Questions with Answers

Below is a comprehensive set of university-level questions and answers for 10 key topics from your Java exam outline. Each topic includes 5 conceptual questions and 5 code-based questions, designed to be logical, challenging, and time-consuming, with detailed answers to aid your preparation.

1. JDK, JRE, and JVM

Conceptual Questions

1. **What is the role of the JDK in Java development, and how does it differ from the JRE?**
   Answer: The JDK (Java Development Kit) is a software development environment used for developing Java applications. It includes tools like javac (compiler), jar (archiver), and javadoc (documentation generator), along with the JRE. The JRE (Java Runtime Environment) provides the runtime environment to execute Java programs, including the JVM (Java Virtual Machine) and core libraries. The key difference is that the JDK is for development (compiling and debugging), while the JRE is for running Java applications.
2. **How does the JVM ensure platform independence?**
   Answer: The JVM ensures platform independence by executing Java bytecode, which is platform-neutral. The javac compiler translates Java source code into bytecode, which the JVM interprets or compiles into machine code specific to the host platform. This abstraction allows the same bytecode to run on any platform with a compatible JVM, without needing source code modifications.
3. **What is the purpose of the ClassLoader in the JVM?**
   Answer: The ClassLoader is a JVM subsystem responsible for loading class files into memory during program execution. It follows a delegation model, where it first checks parent class loaders (Bootstrap, Extension, System) before loading a class

itself. It ensures security, modularity, and dynamic loading of classes.

4. **How does Just-In-Time (JIT) compilation improve JVM performance?**
   Answer: JIT compilation improves performance by compiling bytecode into native machine code at runtime, rather than interpreting it. The JIT compiler optimizes frequently executed code paths (hot spots) using techniques like inlining and loop unrolling, reducing execution time compared to interpretation.
5. **What are the implications of garbage collection in the JVM?**
   Answer: Garbage collection (GC) in the JVM automatically reclaims memory from objects no longer referenced, preventing memory leaks. It improves developer productivity but can introduce performance overhead due to pause times during GC cycles. Modern GC algorithms (e.g., G1) minimize pauses but may still affect real-time applications.

Code-Based Questions

1. **Write a program to display the JDK version using** System.getProperty()**.**
   Code:

```
public class JDKVersion {
    public static void main(String[] args) {
        String jdkVersion = System.getProperty("java.version");
        System.out.println("JDK Version: " + jdkVersion);
    }
}
```

Answer: This program uses System.getProperty("java.version") to retrieve the JDK version. When run, it outputs the version (e.g., "17.0.2"). It demonstrates accessing JVM system properties.

7. **Create a program that triggers an** OutOfMemoryError **to show JVM heap limitations.**

   Code:

```
import java.util.ArrayList;
public class HeapOverflow {
        public static void main(String[] args) {
                ArrayList<byte[]> list = new ArrayList<>();
                try {
                    while (true) {
                        list.add(new byte[1024 * 1024]); // Allocate 1MB chunks
                    }
                } catch (OutOfMemoryError e) {
System.out.println("OutOfMemoryError occurred after allocating " + list.size() + " MB");
                }
        }
}
```

   Answer: This program continuously allocates 1MB byte arrays until the JVM's heap is exhausted, triggering an OutOfMemoryError. The try-catch block catches the error and displays the amount of memory allocated. It illustrates JVM heap management.

20. **Write a program to display JVM memory statistics using** Runtime**.**

   Code:

```
    public class MemoryStats {
        public static void main(String[] args) {
```

```
23.             Runtime runtime =
  Runtime.getRuntime();
24.             long maxMemory =
  runtime.maxMemory() / (1024 * 1024);
25.             long freeMemory =
  runtime.freeMemory() / (1024 * 1024);
26.             long totalMemory =
  runtime.totalMemory() / (1024 * 1024);
27.             System.out.println("Max Memory:
  " + maxMemory + " MB");
28.             System.out.println("Free
  Memory: " + freeMemory + " MB");
29.             System.out.println("Total
  Memory: " + totalMemory + " MB");
30.         }
  }
```

Answer: This program uses Runtime.getRuntime() to access JVM memory details. It converts bytes to MB for readability and displays max, free, and total memory, demonstrating JVM memory management.

31. **Implement a program to load a class dynamically using** Class.forName()**.**
Code:

```
32.     public class DynamicClassLoader {
33.         public static void main(String[]
  args) {
34.             try {
35.                 Class<?> cls =
  Class.forName("java.util.ArrayList");
36.                 System.out.println("Class
  loaded: " + cls.getName());
37.             } catch (ClassNotFoundException
  e) {
```

```
            System.out.println("Class not found: " + e.getMessage());
        }
    }
}
```

Answer: This program uses Class.forName() to dynamically load the ArrayList class. If successful, it prints the class name; otherwise, it catches ClassNotFoundException. It shows the ClassLoader's role in dynamic loading.

41. **Write a program that simulates garbage collection by creating and dereferencing objects.**

Code:

```
public class GarbageCollectionDemo {
    public static void main(String[] args) {
        for (int i = 0; i < 100000; i++) {
            Object obj = new Object(); // Create object
            obj = null; // Dereference for GC eligibility
        }
        System.out.println("Requesting GC...");
        System.gc(); // Suggest garbage collection
        System.out.println("GC suggested. Check JVM logs for details.");
    }
}
```

Answer: This program creates and dereferences Object instances in a loop, making them eligible for garbage collection. System.gc()

suggests GC, though it’s not guaranteed to run immediately. It demonstrates GC eligibility and JVM behavior.

2. Encapsulation

Conceptual Questions

1. **How does encapsulation enhance data security in Java?**
   Answer: Encapsulation hides an object’s internal state by making fields private and providing public methods (getters/setters) for controlled access. This prevents direct modification of data, ensuring integrity and validation, thus enhancing security.
2. **Why are getter and setter methods critical for encapsulation?**
   Answer: Getters and setters allow controlled access to private fields. Getters retrieve values, while setters can include validation logic to ensure only valid data is assigned, maintaining encapsulation and data integrity.
3. **How can encapsulation be violated?**
   Answer: Encapsulation can be violated by exposing mutable objects through getters (e.g., returning a reference to a List), allowing external code to modify the internal state directly, bypassing validation.
4. **What is the role of the** final **keyword in encapsulation?**
   Answer: The final keyword prevents fields from being reassigned after initialization, ensuring immutability. It enhances encapsulation by protecting critical data from unintended changes.
5. **How does encapsulation support thread safety?**
   Answer: Encapsulation can ensure thread safety by controlling access to shared data via synchronized methods or immutable objects, preventing race conditions in multithreaded environments.

Code-Based Questions

1. **Write a class** BankAccount **with encapsulated fields for balance and account number.**
   Code:

```
2. public class BankAccount {
3.     private double balance;
4.     private String accountNumber;
5.     public BankAccount(String
  accountNumber, double initialBalance) {
6.         this.accountNumber = accountNumber;
7.         this.balance = initialBalance >= 0
  ? initialBalance : 0;
8.     }
9.     public double getBalance() { return
  balance; }
10.         public String getAccountNumber() {
  return accountNumber; }
11.         public void deposit(double amount)
  {
12.             if (amount > 0) balance +=
  amount;
13.         }
14.         public boolean withdraw(double
  amount) {
15.             if (amount > 0 && balance >=
  amount) {
16.                 balance -= amount;
17.                 return true;
18.             }
19.             return false;
20.         }
  }
```

Answer: This class encapsulates balance and accountNumber as private fields. Getters provide read-only access, and deposit/withdraw methods validate inputs, ensuring data integrity.

21. **Create a** Student **class with encapsulated grades.**
Code:

```
22.     public class Student {
```

```
23.         private String name;
24.         private double[] grades;
25.         public Student(String name, double[] grades) {
26.             this.name = name;
27.             this.grades = new double[grades.length];
28.             for (int i = 0; i < grades.length; i++) {
29.                 this.grades[i] = grades[i] >= 0 ? grades[i] : 0;
30.             }
31.         }
32.         public String getName() { return name; }
33.         public double[] getGrades() { return grades.clone(); }
34.         public void updateGrade(int index, double grade) {
35.             if (index >= 0 && index < grades.length && grade >= 0) {
36.                 grades[index] = grade;
37.             }
38.         }
  }
```

Answer: The Student class encapsulates name and grades. The getGrades method returns a copy of the array to prevent external modification, and updateGrade validates inputs.

39. **Implement a** Car **class with encapsulated speed.**

Code:

```
40.     public class Car {
41.         private double speed;
42.         public Car(double speed) {
```

```
43.             this.speed = speed >= 0 ? speed
  : 0;
44.         }
45.         public double getSpeed() { return
  speed; }
46.         public void setSpeed(double speed)
  {
47.             if (speed >= 0 && speed <= 200)
  this.speed = speed;
48.         }
49.         public void accelerate(double
  increment) {
50.             if (increment > 0)
  setSpeed(speed + increment);
51.         }
  }
```

Answer: The Car class encapsulates speed with validation in the constructor and setSpeed. The accelerate method uses the setter to ensure valid speed updates.

52. **Write a program demonstrating encapsulation violation.**

Code:

```
53.     import java.util.ArrayList;
54.     public class BadEncapsulation {
55.         private ArrayList<String> items =
  new ArrayList<>();
56.         public ArrayList<String> getItems()
  { return items; } // Violation
57.         public void addItem(String item) {
  items.add(item); }
58.         public static void main(String[]
  args) {
59.             BadEncapsulation be = new
  BadEncapsulation();
60.             be.addItem("Book");
```

```
            ArrayList<String> list = be.getItems();
            list.clear(); // Modifies internal state
            System.out.println("Items: " + be.getItems().size()); // Outputs 0
        }
}
```

Answer: This program returns a direct reference to items, allowing external code to modify it (e.g., clear()), violating encapsulation. A fix would return items.clone().

65. **Create a thread-safe encapsulated** Counter **class.**

Code:

```
    public class Counter {
        private int count;
        public synchronized int getCount() { return count; }
        public synchronized void increment() { count++; }
        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();
            System.out.println("Final count: " + counter.getCount()); // 2000
        }
}
```

Answer: The Counter class uses synchronized methods to ensure thread-safe access to count. The main method tests concurrent increments, ensuring accurate results.

3. Inheritance

Conceptual Questions

1. **What is the purpose of inheritance in Java?**
   Answer: Inheritance allows a class (subclass) to inherit fields and methods from another class (superclass), promoting code reuse, modularity, and extensibility. It models "is-a" relationships (e.g., a Dog is an Animal).
2. **How does the** super **keyword work in inheritance?**
   Answer: The super keyword refers to the superclass. It can call superclass constructors (super()), access superclass methods/fields, or invoke overridden methods, ensuring proper initialization and behavior extension.
3. **What is the "is-a" relationship in inheritance?**
   Answer: The "is-a" relationship indicates that a subclass is a specialized type of its superclass (e.g., a Car is a Vehicle). It ensures logical hierarchy and supports polymorphism.
4. **Why doesn't Java support multiple inheritance?**
   Answer: Java avoids multiple inheritance to prevent ambiguity (e.g., the "diamond problem" where two superclasses define the same method). Instead, it supports multiple interfaces to achieve similar flexibility.
5. **How does inheritance affect constructor calls?**
   Answer: When a subclass object is created, the superclass constructor is called first (implicitly or via super()), ensuring the superclass is initialized before the subclass adds its own initialization.

Code-Based Questions

1. **Write a program with a** Vehicle **base class and** Car **subclass.**
   Code:

```
class Vehicle {
    protected String brand;
    public Vehicle(String brand) { this.brand = brand; }
    public String getBrand() { return brand; }
}
class Car extends Vehicle {
    private int doors;
    public Car(String brand, int doors) {
            super(brand);
            this.doors = doors;
        }
        public String getDetails() { return brand + " with " + doors + " doors"; }
        public static void main(String[] args) {
            Car car = new Car("Toyota", 4);
            System.out.println(car.getDetails()); // Toyota with 4 doors
        }
}
```

   Answer: The Vehicle class defines a brand, and Car extends it, adding doors. The super keyword calls the superclass constructor, and getDetails demonstrates inherited and new behavior.

18. **Create a** Person **and** Student **class hierarchy.**
   Code:

```
    class Person {
        protected String name;
        public Person(String name) { this.name = name; }
```

```
22.         public String getName() { return name; }
23.     }
24.     class Student extends Person {
25.         private int id;
26.         public Student(String name, int id) {
27.             super(name);
28.             this.id = id;
29.         }
30.         public String getDetails() { return name + ", ID: " + id; }
31.         public static void main(String[] args) {
32.             Student student = new Student("Alice", 123);
33. System.out.println(student.getDetails()); // Alice, ID: 123
34.         }
}
```

Answer: The Person class is extended by Student, which adds an id. The super keyword initializes the name, and getDetails combines inherited and subclass fields.

35. **Implement a program using** protected **fields in inheritance.**
Code:

```
36.     class Animal {
37.         protected String species;
38.         public Animal(String species) { this.species = species; }
39.         public String getSpecies() { return species; }
40.     }
```

```
class Dog extends Animal {
    private String breed;
    public Dog(String species, String breed) {
        super(species);
        this.breed = breed;
    }
    public String getDetails() { return species + ", Breed: " + breed; }
    public static void main(String[] args) {
        Dog dog = new Dog("Canine", "Labrador");
        System.out.println(dog.getDetails()); // Canine, Breed: Labrador
    }
}
```

Answer: The protected species field is accessible in the Dog subclass. The program demonstrates inheritance with protected access and proper constructor chaining.

52. **Write a program preventing inheritance with** final**.**

Code:

```
final class Immutable {
    private int value;
    public Immutable(int value) { this.value = value; }
    public int getValue() { return value; }
}
// class Test extends Immutable {} // Compilation error
public class FinalClassTest {
```

```
60.         public static void main(String[]
  args) {
61.             Immutable obj = new
  Immutable(42);
62.             System.out.println("Value: " +
  obj.getValue()); // Value: 42
63.         }
  }
```

Answer: The final keyword prevents Immutable from being extended. The commented-out line would cause a compilation error, demonstrating the effect of final.

64. **Create a banking system hierarchy with** Account **and** SavingsAccount**.**
Code:

```
65.     class Account {
66.         protected double balance;
67.         public Account(double balance) {
  this.balance = balance; }
68.         public double getBalance() { return
  balance; }
69.     }
70.     class SavingsAccount extends Account {
71.         private double interestRate;
72.         public SavingsAccount(double
  balance, double interestRate) {
73.             super(balance);
74.             this.interestRate =
  interestRate;
75.         }
76.         public double calculateInterest() {
  return balance * interestRate; }
77.         public static void main(String[]
  args) {
```

```
            SavingsAccount sa = new SavingsAccount(1000, 0.05);
            System.out.println("Interest: " + sa.calculateInterest()); // Interest: 50.0
        }
}
```

Answer: The SavingsAccount extends Account, inheriting balance and adding interestRate. The calculateInterest method uses inherited fields, showing inheritance in action.

4. Polymorphism

Conceptual Questions

1. **What are the two types of polymorphism in Java?**
   Answer: Java supports **compile-time (static) polymorphism** via method overloading, where multiple methods with the same name differ by parameters, resolved at compile time. **Runtime (dynamic) polymorphism** is achieved through method overriding, where a subclass provides a specific implementation, resolved at runtime via dynamic method dispatch.
2. **How does method overriding enable runtime polymorphism?**
   Answer: Method overriding allows a subclass to redefine a superclass method with the same signature. The JVM uses dynamic method dispatch to call the appropriate method based on the object's actual type at runtime, enabling polymorphic behavior.
3. **What is the role of upcasting in polymorphism?**
   Answer: Upcasting involves treating a subclass object as its superclass type. It enables polymorphic behavior by allowing superclass references to call overridden methods in subclasses, leveraging dynamic method dispatch.
4. **Why can't static methods be overridden?**
   Answer: Static methods belong to the class, not instances, so they are resolved at compile time based on the reference type, not the

object's actual type. Overriding applies to instance methods for runtime polymorphism.

5. **How does polymorphism support the Open-Closed Principle?**
   Answer: Polymorphism allows new functionality to be added via subclasses or interfaces without modifying existing code. This adheres to the Open-Closed Principle, as classes are open for extension (new subclasses) but closed for modification.

Code-Based Questions

1. **Write a program demonstrating method overriding in a** Shape **hierarchy.**
   Code:

```
abstract class Shape {
    abstract double getArea();
}
class Circle extends Shape {
    private double radius;
    public Circle(double radius) { this.radius = radius; }
    @Override
    double getArea() { return Math.PI * radius * radius; }
}
class Rectangle extends Shape {
    private double width, height;
    public Rectangle(double width, double height) { this.width = width; this.height = height; }
    @Override
    double getArea() { return width * height; }
    public static void main(String[] args) {
        Shape circle = new Circle(5);
18.             Shape rectangle = new Rectangle(4, 6);
19.             System.out.println("Circle Area: " + circle.getArea()); // Circle Area: 78.54...
20.             System.out.println("Rectangle Area: " + rectangle.getArea()); // Rectangle Area: 24.0
21.         }
    }
```

Answer: This program uses an abstract Shape class with an overridden getArea method in Circle and Rectangle. Upcasting to Shape demonstrates runtime polymorphism.

22. **Create a program with method overloading for calculating areas.**
Code:

```
23.     class AreaCalculator {
24.         public double calculateArea(double radius) {
25.             return Math.PI * radius * radius;
26.         }
27.         public double calculateArea(double width, double height) {
28.             return width * height;
29.         }
30.         public static void main(String[] args) {
31.             AreaCalculator calc = new AreaCalculator();
32.             System.out.println("Circle Area: " + calc.calculateArea(5)); // Circle Area: 78.54...
```

```
        System.out.println("Rectangle Area: " + calc.calculateArea(4, 6)); // Rectangle Area: 24.0
    }
}
```

Answer: The calculateArea method is overloaded for circles (one parameter) and rectangles (two parameters). The compiler resolves the correct method at compile time, showing static polymorphism.

35. **Implement a program using an interface for polymorphism.**

Code:

```
interface Printable {
    void print();
}
class Book implements Printable {
    private String title;
    public Book(String title) { this.title = title; }
    public void print() { System.out.println("Book: " + title); }
}
class Magazine implements Printable {
    private String name;
    public Magazine(String name) { this.name = name; }
    public void print() { System.out.println("Magazine: " + name); }
    public static void main(String[] args) {
        Printable book = new Book("Java Guide");
        Printable magazine = new Magazine("Tech Weekly");
51.             book.print(); // Book: Java
  Guide
52.             magazine.print(); // Magazine:
  Tech Weekly
53.         }
  }
```

Answer: The Printable interface defines a print method, implemented differently by Book and Magazine. Upcasting to Printable demonstrates polymorphic behavior.

54. **Write a program using** instanceof **for safe polymorphic handling.**
Code:

```
55.     class Animal {
56.         void makeSound() {
  System.out.println("Generic sound"); }
57.     }
58.     class Dog extends Animal {
59.         void makeSound() {
  System.out.println("Woof"); }
60.     }
61.     class Cat extends Animal {
62.         void makeSound() {
  System.out.println("Meow"); }
63.         public static void main(String[]
  args) {
64.             Animal[] animals = {new Dog(),
  new Cat(), new Animal()};
65.             for (Animal animal : animals) {
66.                 animal.makeSound();
67.                 if (animal instanceof Dog)
  {
68.
  System.out.println("This is a Dog");
```

```
                } else if (animal instanceof Cat) {
                    System.out.println("This is a Cat");
                }
            }
        }
}
```

Answer: This program uses instanceof to check object types in a polymorphic array. Each makeSound call uses the overridden method, and instanceof identifies specific types.

74. **Create a program demonstrating covariant return types.**

Code:

```
    class Vehicle {
        Vehicle getInstance() { return this; }
    }
    class Car extends Vehicle {
        @Override
        Car getInstance() { return this; }
        public static void main(String[] args) {
            Vehicle vehicle = new Car();
            Car car = (Car) vehicle.getInstance();
            System.out.println("Instance is: " + car.getClass().getSimpleName()); // Instance is: Car
        }
}
```

Answer: The Car class overrides getInstance to return a Car type instead of Vehicle, demonstrating covariant return types. The program confirms the returned object’s type.

5. Exception Handling

Conceptual Questions

1. **What is the difference between checked and unchecked exceptions?**
   Answer: Checked exceptions (e.g., IOException) are checked at compile time, requiring try-catch or throws. Unchecked exceptions (e.g., NullPointerException) are runtime exceptions and don't require explicit handling, allowing more flexible coding but riskier error management.
2. **How does the** try-with-resources **statement work?**
   Answer: The try-with-resources statement ensures that resources (e.g., files, sockets) implementing AutoCloseable are automatically closed after the try block, even if an exception occurs, reducing boilerplate code for resource cleanup.
3. **What is the purpose of the** finally **block?**
   Answer: The finally block executes code (e.g., resource cleanup) regardless of whether an exception is thrown or caught. It ensures critical cleanup operations are performed, unless the JVM exits abruptly.
4. **How do** throw **and** throws **differ?**
   Answer: throw is used to explicitly throw an exception (e.g., throw new IOException()). throws is used in a method signature to declare exceptions that might be thrown, alerting callers to handle them.
5. **What are chained exceptions?**
   Answer: Chained exceptions allow an exception to wrap another as its cause (e.g., new Exception("Message", cause)). This improves debugging by preserving the root cause of an error across method calls.

Code-Based Questions

1. **Write a program handling an** ArithmeticException**.**
   Code:

```
2. public class ArithmeticTest {
3.     public static void main(String[] args) {
4.         try {
5.             int result = 10 / 0;
6.             System.out.println("Result: " + result);
7.         } catch (ArithmeticException e) {
8.             System.out.println("Error: Division by zero");
9.         }
10.         }
}
```

Answer: This program attempts to divide by zero, triggering an ArithmeticException. The try-catch block catches it and prints an error message, preventing program termination.

11. **Create a program using** try-with-resources **for file handling.**

Code:

```
12.     import java.io.BufferedReader;
13.     import java.io.FileReader;
14.     public class FileReaderTest {
15.         public static void main(String[] args) {
16.             try (BufferedReader br = new BufferedReader(new FileReader("test.txt"))) {
17.                 String line = br.readLine();
18.                 System.out.println("First line: " + line);
19.             } catch (IOException e) {
20.                 System.out.println("Error reading file: " + e.getMessage());
```

```
        }
    }
}
```

Answer: The try-with-resources ensures the BufferedReader is closed automatically. If the file doesn’t exist, an IOException is caught, demonstrating safe resource management.

23. **Implement a custom exception class.**

Code:

```
class InvalidAgeException extends Exception {
    public InvalidAgeException(String message) { super(message); }
}
public class CustomExceptionTest {
    public static void setAge(int age) throws InvalidAgeException {
        if (age < 0 || age > 150) throw new InvalidAgeException("Invalid age: " + age);
        System.out.println("Age set: " + age);
    }
    public static void main(String[] args) {
        try {
            setAge(-5);
        } catch (InvalidAgeException e) {
            System.out.println(e.getMessage());
        }
    }
}
```

Answer: The InvalidAgeException is a custom exception thrown when an invalid age is provided. The try-catch block handles it, showing custom exception usage.

39. **Write a program demonstrating exception propagation.**

Code:

```
public class ExceptionPropagation {
    public static void method2() throws IOException {
        throw new IOException("Error in method2");
    }
    public static void method1() throws IOException {
        method2();
    }
    public static void main(String[] args) {
        try {
            method1();
        } catch (IOException e) {
            System.out.println("Caught: " + e.getMessage());
        }
    }
}
```

Answer: The IOException thrown in method2 propagates through method1 to main, where it’s caught. This demonstrates how exceptions travel up the call stack.

54. **Create a program using chained exceptions.**

Code:

```
public class ChainedExceptionTest {
    public static void main(String[] args) {
        try {
            try {
                int[] arr = new int[1];
                arr[2] = 10; // Triggers ArrayIndexOutOfBoundsException
            } catch (ArrayIndexOutOfBoundsException e) {
                throw new RuntimeException("Processing error", e);
            }
        } catch (RuntimeException e) {
            System.out.println("Error: " + e.getMessage());
            System.out.println("Cause: " + e.getCause());
        }
    }
}
```

Answer: The inner try block triggers an ArrayIndexOutOfBoundsException, which is wrapped in a RuntimeException as its cause. The outer catch prints both the message and cause, showing chained exceptions.

6. Constructors

Conceptual Questions

1. **What is the purpose of a constructor in Java?**
   Answer: A constructor initializes an object's state when it's created. It has the same name as the class, no return type, and is called automatically via the new operator to set initial values.
2. **How does the** this **keyword work in constructors?**
   Answer: In constructors, this refers to the current object. It's used to access instance variables (e.g., to avoid shadowing) or call another constructor in the same class (constructor chaining).

3. **What is a parameterized constructor?**
   Answer: A parameterized constructor accepts parameters to initialize an object’s fields with specific values, allowing customized object creation compared to a default constructor.
4. **What happens if a class has no constructor defined?**
   Answer: If no constructor is defined, the compiler provides a default no-argument constructor that initializes fields to their default values (e.g., null for objects, 0 for numbers).
5. **How does constructor overloading work?**
   Answer: Constructor overloading allows multiple constructors in a class with different parameter lists. The compiler selects the appropriate constructor based on the arguments provided during object creation.

Code-Based Questions

1. **Write a class with a default and parameterized constructor.**
   Code:

```
public class Person {
    private String name;
    private int age;
    public Person() {
        this.name = "Unknown";
        this.age = 0;
    }
    public Person(String name, int age) {
            this.name = name;
            this.age = age;
        }
        public String getDetails() { return name + ", " + age; }
        public static void main(String[] args) {
            Person p1 = new Person();
16.             Person p2 = new Person("Alice", 25);
17. System.out.println(p1.getDetails()); // Unknown, 0
18. System.out.println(p2.getDetails()); // Alice, 25
19.         }
    }
```

Answer: The Person class has a default constructor and a parameterized constructor. The main method demonstrates both, showing different initialization behaviors.

20. **Create a class using** this **for constructor chaining.**

Code:

```
21.     public class Rectangle {
22.         private double width, height;
23.         public Rectangle() {
24.             this(1.0, 1.0); // Chain to parameterized constructor
25.         }
26.         public Rectangle(double width, double height) {
27.             this.width = width;
28.             this.height = height;
29.         }
30.         public double getArea() { return width * height; }
31.         public static void main(String[] args) {
32.             Rectangle r1 = new Rectangle();
33.             Rectangle r2 = new Rectangle(5, 3);
34.             System.out.println("Default Area: " + r1.getArea()); // Default Area: 1.0
35.             System.out.println("Custom Area: " + r2.getArea()); // Custom Area: 15.0
36.         }
    }
```

Answer: The default constructor chains to the parameterized constructor using this, ensuring consistent initialization. The main method tests both constructors.

37. **Implement a class with constructor handling invalid input.**
Code:

```
38.     public class Circle {
39.         private double radius;
40.         public Circle(double radius) {
41.             if (radius < 0) throw new IllegalArgumentException("Radius cannot be negative");
42.             this.radius = radius;
43.         }
44.         public double getArea() { return Math.PI * radius * radius; }
45.         public static void main(String[] args) {
46.             try {
47.                 Circle c = new Circle(-1);
48.             } catch (IllegalArgumentException e) {
49.                 System.out.println("Error: " + e.getMessage());
50.             }
51.             Circle c2 = new Circle(2);
52.             System.out.println("Area: " +
  c2.getArea()); // Area: 12.56...
53.         }
  }
```

Answer: The constructor validates the radius and throws an exception for negative values. The main method demonstrates error handling and valid initialization.

54. **Write a program with constructor overloading.**

Code:

```
55.     public class Book {
56.         private String title;
57.         private String author;
58.         public Book() {
59.             this("Unknown", "Unknown");
60.         }
61.         public Book(String title) {
62.             this(title, "Unknown");
63.         }
64.         public Book(String title, String
  author) {
65.             this.title = title;
66.             this.author = author;
67.         }
68.         public String getDetails() { return
  title + " by " + author; }
69.         public static void main(String[]
  args) {
70.             Book b1 = new Book();
71.             Book b2 = new Book("Java
  Guide");
72.             Book b3 = new Book("Java
  Guide", "Alice");
73.
  System.out.println(b1.getDetails()); //
  Unknown by Unknown
74.
  System.out.println(b2.getDetails()); // Java
  Guide by Unknown
75.
  System.out.println(b3.getDetails()); // Java
  Guide by Alice
76.         }
  }
```

Answer: The Book class has three overloaded constructors, chaining to the most specific one. The main method tests all constructors, showing flexible initialization.

77. **Create a class passing an object to its constructor.**

Code:

```
78.     public class Order {
79.         private String item;
80.         private int quantity;
81.         public Order(String item, int
  quantity) {
82.             this.item = item;
83.             this.quantity = quantity;
84.         }
85.         public Order(Order other) {
86.             this.item = other.item;
87.             this.quantity = other.quantity;
88.         }
89.         public String getDetails() { return
  item + ", Quantity: " + quantity; }
90.         public static void main(String[]
  args) {
91.             Order o1 = new Order("Laptop",
  2);
```

```
                Order o2 = new Order(o1);
        System.out.println(o1.getDetails()); // Laptop, Quantity: 2
        System.out.println(o2.getDetails()); // Laptop, Quantity: 2
    }
}
```

Answer: The Order class has a copy constructor that initializes a new object based on another Order. The main method demonstrates copying an object’s state.

7. Method Overloading

Conceptual Questions

1. **What is method overloading in Java?**
   Answer: Method overloading allows multiple methods in the same class with the same name but different parameter lists (number, type, or order). The compiler resolves the correct method at compile time based on the arguments.
2. **How does method overloading differ from overriding?**
   Answer: Overloading occurs in the same class with different parameter lists and is resolved at compile time (static polymorphism). Overriding occurs in a subclass with the same method signature and is resolved at runtime (dynamic polymorphism).
3. **What are the benefits of method overloading?**
   Answer: Overloading improves code readability and flexibility by allowing methods with related functionality to share the same name, reducing the need for distinct method names.
4. **Can overloaded methods have different return types?**
   Answer: Yes, but the return type alone cannot distinguish overloaded methods. The parameter list must differ, as the

compiler uses the method signature (name and parameters) for resolution.

5. **What happens if an overloaded method call is ambiguous?**
   Answer: If the compiler cannot determine the best match for an overloaded method (e.g., due to compatible parameter types), it results in a compilation error, requiring explicit casting to resolve ambiguity.

Code-Based Questions

1. **Write a program with overloaded methods for addition.**
   Code:

```
public class Adder {
    public int add(int a, int b) { return a + b; }
    public double add(double a, double b) { return a + b; }
    public int add(int a, int b, int c) { return a + b + c; }
    public static void main(String[] args) {
        Adder adder = new Adder();
        System.out.println("Int sum: " + adder.add(1, 2)); // Int sum: 3
        System.out.println("Double sum: " + adder.add(1.5, 2.5)); // Double sum: 4.0
        System.out.println("Three int sum: " + adder.add(1, 2, 3)); // Three int sum: 6
    }
}
```

Answer: The Adder class overloads the add method for different parameter types and counts. The main method demonstrates calling each version based on arguments.

12. **Create a program overloading a method to print different data types.**
Code:

```
13.    public class Printer {
14.        public void print(String s) { System.out.println("String: " + s); }
15.        public void print(int i) { System.out.println("Integer: " + i); }
16.        public void print(double d) { System.out.println("Double: " + d); }
17.        public static void main(String[] args) {
18.            Printer printer = new Printer();
19.            printer.print("Hello"); // String: Hello
20.            printer.print(42); // Integer: 42
21.            printer.print(3.14); // Double: 3.14
22.        }
       }
```

Answer: The Printer class overloads print for String, int, and double. The main method shows how the compiler selects the appropriate method based on the argument type.

23. **Implement a program with overloaded methods handling arrays.**
Code:

```
24.    public class ArrayProcessor {
25.        public int sum(int[] arr) {
26.            int total = 0;
27.            for (int x : arr) total += x;
28.            return total;
29.        }
30.         public double sum(double[] arr) {
31.             double total = 0;
32.             for (double x : arr) total += x;
33.             return total;
34.         }
35.         public static void main(String[] args) {
36.             ArrayProcessor ap = new ArrayProcessor();
37.             int[] intArr = {1, 2, 3};
38.             double[] doubleArr = {1.5, 2.5, 3.5};
39.             System.out.println("Int sum: " + ap.sum(intArr)); // Int sum: 6
40.             System.out.println("Double sum: " + ap.sum(doubleArr)); // Double sum: 7.5
41.         }
  }
```

Answer: The sum method is overloaded for int[] and double[]. The main method tests both, showing how overloading handles different array types.

42. **Write a program demonstrating method overloading ambiguity.**

Code:

```
43.     public class AmbiguousOverload {
44.         public void process(long a, int b) { System.out.println("Long, Int: " + a + ", " + b); }
45.         public void process(int a, long b) { System.out.println("Int, Long: " + a + ", " + b); }
46.         public static void main(String[] args) {
```

```
47.             AmbiguousOverload ao = new AmbiguousOverload();
48.             ao.process(1, 2); // Compilation error: ambiguous call
49.             ao.process((long) 1, 2); // Long, Int: 1, 2
50.             ao.process(1, (long) 2); // Int, Long: 1, 2
51.         }
    }
```

Answer: The process method is overloaded, but ao.process(1, 2) is ambiguous because both int values can be cast to long. Explicit casting resolves the ambiguity, as shown.

52. **Create a program overloading methods for string manipulation.**

Code:

```
53.     public class StringUtils {
54.         public String repeat(String s) { return s + s; }
55.         public String repeat(String s, int times) {
56.             StringBuilder sb = new StringBuilder();
57.             for (int i = 0; i < times; i++) sb.append(s);
58.             return sb.toString();
59.         }
60.         public static void main(String[] args) {
61.             StringUtils su = new StringUtils();
62.             System.out.println("Double: " + su.repeat("Hi")); // Double: HiHi
```

```
            System.out.println("Triple: " + su.repeat("Hi", 3)); // Triple: HiHiHi
        }
}
```

Answer: The repeat method is overloaded to repeat a string twice or a specified number of times. The main method demonstrates both versions.

8. Abstract Classes

Conceptual Questions

1. **What is an abstract class in Java?**
   Answer: An abstract class is a class declared with the abstract keyword that cannot be instantiated. It may contain abstract methods (without implementation) and concrete methods, serving as a blueprint for subclasses.
2. **How do abstract classes differ from interfaces?**
   Answer: Abstract classes can have state (fields), concrete methods, and constructors, while interfaces have only abstract methods (or default/static methods since Java 8) and no state. A class can extend one abstract class but implement multiple interfaces.
3. **When should you use an abstract class over a concrete class?**
   Answer: Use an abstract class when you want to provide common functionality and state for subclasses but prevent instantiation of the base class itself (e.g., a Shape class with a common color field).
4. **Can an abstract class have a constructor?**
   Answer: Yes, an abstract class can have constructors to initialize fields. Subclasses call these constructors (via super) when instantiated, ensuring proper initialization.
5. **What is the purpose of abstract methods?**
   Answer: Abstract methods declare a method signature without implementation, forcing subclasses to provide their own implementation. They ensure subclasses adhere to a contract while allowing customization.

Code-Based Questions

1. **Write an abstract class** Animal **with an abstract method.**
   Code:

```
abstract class Animal {
    protected String name;
    public Animal(String name) { this.name = name; }
    abstract void makeSound();
}
class Dog extends Animal {
    public Dog(String name) { super(name); }
    void makeSound() { System.out.println(name + " says Woof"); }
        public static void main(String[] args) {
            Animal dog = new Dog("Rex");
            dog.makeSound(); // Rex says Woof
        }
}
```

   Answer: The Animal class is abstract with an abstract makeSound method. Dog provides the implementation, and the main method demonstrates polymorphic behavior.

14. **Create an abstract class for a** Vehicle **hierarchy.**
   Code:

```
abstract class Vehicle {
    protected String model;
    public Vehicle(String model) { this.model = model; }
    abstract double getSpeed();
}
class Car extends Vehicle {
```

```
21.         public Car(String model) { super(model); }
22.         double getSpeed() { return 120.5; }
23.         public static void main(String[] args) {
24.             Vehicle car = new Car("Sedan");
25.             System.out.println(car.model + " speed: " + car.getSpeed()); // Sedan speed: 120.5
26.         }
    }
```

Answer: The Vehicle abstract class defines a model and an abstract getSpeed method. Car implements it, and the main method tests the hierarchy.

27. **Implement an abstract class with a concrete method.**

Code:

```
28.     abstract class Shape {
29.         protected String color;
30.         public Shape(String color) { this.color = color; }
31.         public String getColor() { return color; }
32.         abstract double getArea();
33.     }
34.     class Circle extends Shape {
35.         private double radius;
36.         public Circle(String color, double radius) {
37.             super(color);
38.             this.radius = radius;
39.         }
40.         double getArea() { return Math.PI * radius * radius; }
41.         public static void main(String[] args) {
42.             Shape circle = new Circle("Red", 5);
43.             System.out.println("Color: " + circle.getColor() + ", Area: " + circle.getArea());
44.         }
    }
```

Answer: The Shape class has a concrete getColor method and an abstract getArea method. Circle implements getArea, and the main method uses both methods.

45. **Write a program with multiple subclasses of an abstract class.**
Code:

```
46.     abstract class Employee {
47.         protected String name;
48.         public Employee(String name) { this.name = name; }
49.         abstract double calculateSalary();
50.     }
51.     class Manager extends Employee {
52.         public Manager(String name) { super(name); }
53.         double calculateSalary() { return 50000; }
54.     }
55.     class Developer extends Employee {
56.         public Developer(String name) { super(name); }
57.         double calculateSalary() { return 40000; }
58.         public static void main(String[] args) {
```

```
            Employee[] employees = {new Manager("Alice"), new Developer("Bob")};
            for (Employee e : employees) {
                System.out.println(e.name + ": $" + e.calculateSalary());
            }
        }
}
```

Answer: The Employee abstract class has an abstract calculateSalary method. Manager and Developer implement it differently, and the main method uses polymorphism.

64. **Create an abstract class preventing instantiation.**

Code:

```
    abstract class Printer {
        abstract void print();
    }
    class LaserPrinter extends Printer {
        void print() { System.out.println("Laser printing"); }
        public static void main(String[] args) {
            // Printer p = new Printer(); // Compilation error
            Printer lp = new LaserPrinter();
            lp.print(); // Laser printing
        }
}
```

Answer: The Printer class is abstract, preventing instantiation. LaserPrinter provides the print implementation, and the main method demonstrates its use.

9. Packages and Interfaces

Conceptual Questions

1. **What is the purpose of packages in Java?**
   Answer: Packages organize classes and interfaces into namespaces, preventing name conflicts, improving modularity, and enabling access control (e.g., package-private visibility). They also facilitate code reuse and maintenance.
2. **How does an interface differ from an abstract class?**
   Answer: An interface defines a contract with abstract methods (and default/static methods since Java 8) but no state or constructors. An abstract class can have state, constructors, and concrete methods. A class can implement multiple interfaces but extend only one abstract class.
3. **What is the role of the** default **keyword in interfaces?**
   Answer: Since Java 8, the default keyword allows interfaces to provide method implementations, enabling backward-compatible evolution of interfaces without breaking existing implementations.
4. **How do packages support access control?**
   Answer: Packages define a namespace where classes with no access modifier (package-private) are accessible only within the same package, enhancing encapsulation and controlling visibility.
5. **Why use interfaces for polymorphism?**
   Answer: Interfaces define a contract that multiple unrelated classes can implement, enabling polymorphic behavior. They allow objects of different types to be treated uniformly if they implement the same interface.

Code-Based Questions

1. **Write a program using a package and interface.**
   Code:

```
package com.example;
interface Drawable {
    void draw();
}
public class Circle implements Drawable {
```

```
7.     public void draw() {
  System.out.println("Drawing a circle"); }
8.     public static void main(String[] args)
  {
9.         Drawable circle = new Circle();
10.             circle.draw(); // Drawing a
  circle
11.         }
  }
```

Answer: The Circle class in the com.example package implements the Drawable interface. The main method demonstrates polymorphic use of the interface.

12. **Create an interface with a default method.**

Code:

```
13.     interface Loggable {
14.         void performAction();
15.         default void log() {
  System.out.println("Action logged"); }
16.     }
17.     public class Task implements Loggable {
18.         public void performAction() {
  System.out.println("Task performed"); }
19.         public static void main(String[]
  args) {
20.             Loggable task = new Task();
21.             task.performAction(); // Task
  performed
22.             task.log(); // Action logged
23.         }
  }
```

Answer: The Loggable interface has a default method log. Task implements the interface and uses the default method, showing interface evolution.

24. **Implement a program with multiple classes implementing an interface.**

Code:

```
package com.example;
interface Vehicle {
    void start();
}
class Car implements Vehicle {
    public void start() { System.out.println("Car started"); }
}
class Bike implements Vehicle {
    public void start() { System.out.println("Bike started"); }
    public static void main(String[] args) {
        Vehicle[] vehicles = {new Car(), new Bike()};
        for (Vehicle v : vehicles) {
            v.start();
        }
    }
}
```

Answer: The Vehicle interface is implemented by Car and Bike in the com.example package. The main method uses polymorphism to call start on both.

40. **Write a program using package-private access.**

Code:

```
package com.example;
class Helper {
    void help() { System.out.println("Helper assisting"); }
}
public class Main {
    public static void main(String[] args) {
        Helper helper = new Helper();
        helper.help(); // Helper assisting
    }
}
```

Answer: The Helper class is package-private (no modifier), accessible only within com.example. The Main class in the same package uses it, demonstrating access control.

50. **Create an interface with a static method.**

Code:

```
interface Calculator {
    void compute(int x, int y);
    static int add(int x, int y) { return x + y; }
}
public class BasicCalculator implements Calculator {
    public void compute(int x, int y) { System.out.println("Result: " + (x * y)); }
    public static void main(String[] args) {
        BasicCalculator calc = new BasicCalculator();
        calc.compute(5, 3); // Result: 15
        System.out.println("Static add: " + Calculator.add(5, 3)); // Static add: 8
    }
}
```

Answer: The Calculator interface has a static add method and an abstract compute method. BasicCalculator implements compute, and the main method uses both.

10. File I/O

Conceptual Questions

1. **What is the purpose of the** java.io **package?**
   Answer: The java.io package provides classes for input and output operations, such as reading/writing files, streams, and handling data in various formats, enabling interaction with external resources.
2. **How does** BufferedReader **improve file reading performance?**
   Answer: BufferedReader reduces direct access to the underlying file system by reading data in chunks into a buffer, minimizing I/O operations and improving performance for large files.
3. **What is the difference between byte streams and character streams?**
   Answer: Byte streams (e.g., FileInputStream) handle raw binary data (bytes), suitable for any file type. Character streams (e.g., FileReader) handle text data, automatically managing character encodings.
4. **Why use** try-with-resources **for file I/O?**
   Answer: try-with-resources ensures that file resources are closed automatically after use, preventing resource leaks and simplifying code compared to manual close calls in a finally block.
5. **What are the risks of not closing file resources?**
   Answer: Not closing file resources can lead to resource leaks, locked files, or exhausted file handles, causing program errors or system instability, especially in long-running applications.

Code-Based Questions

1. **Write a program to read a text file using** BufferedReader**.**
   Code:

```
2. import java.io.BufferedReader;
3. import java.io.FileReader;
4. import java.io.IOException;
5. public class FileRead {
6.     public static void main(String[] args)
  {
7.         try (BufferedReader br = new
  BufferedReader(new FileReader("input.txt")))
  {
8.             String line;
9.             while ((line = br.readLine())
  != null) {
10.
  System.out.println(line);
11.                 }
12.             } catch (IOException e) {
13.                 System.out.println("Error:
  " + e.getMessage());
14.             }
15.         }
  }
```

Answer: This program uses BufferedReader with try-with-resources to read and print lines from input.txt. It handles IOException gracefully, ensuring the file is closed.

16. **Create a program to write to a file using** BufferedWriter**.**

Code:

```
17.     import java.io.BufferedWriter;
18.     import java.io.FileWriter;
19.     import java.io.IOException;
20.     public class FileWrite {
21.         public static void main(String[]
  args) {
```

```
22.             try (BufferedWriter bw = new BufferedWriter(new FileWriter("output.txt"))) {
23.                 bw.write("Hello, Java!");
24.                 bw.newLine();
25.                 bw.write("Writing to file.");
26.             } catch (IOException e) {
27.                 System.out.println("Error: " + e.getMessage());
28.             }
29.         }
}
```

Answer: The program uses BufferedWriter to write text to output.txt. try-with-resources ensures the file is closed, and IOException is handled.

30. **Implement a program copying a file using byte streams.**

Code:

```
31.     import java.io.FileInputStream;
32.     import java.io.FileOutputStream;
33.     import java.io.IOException;
34.     public class FileCopy {
35.         public static void main(String[] args) {
36.             try (FileInputStream fis = new FileInputStream("source.txt");
37.                  FileOutputStream fos = new FileOutputStream("dest.txt")) {
38.                 byte[] buffer = new byte[1024];
39.                 int bytesRead;
40.                 while ((bytesRead = fis.read(buffer)) != -1) {
```

```
41.                      fos.write(buffer, 0,
  bytesRead);
42.                  }
43.              } catch (IOException e) {
44.                  System.out.println("Error:
  " + e.getMessage());
45.              }
46.          }
  }
```

Answer: This program copies source.txt to dest.txt using byte streams. try-with-resources ensures streams are closed, and a buffer improves performance.

47. **Write a program handling a missing file gracefully.**
Code:

```
48.     import java.io.BufferedReader;
49.     import java.io.FileReader;
50.     import java.io.FileNotFoundException;
51.     public class FileNotFoundTest {
52.         public static void main(String[]
  args) {
53.             try (BufferedReader br = new
  BufferedReader(new
  FileReader("missing.txt"))) {
54.
  System.out.println(br.readLine());
55.             } catch (FileNotFoundException
  e) {
56.                 System.out.println("File
  not found: " + e.getMessage());
57.             } catch (IOException e) {
58.                 System.out.println("IO
  Error: " + e.getMessage());
59.             }
60.         }
}
```

Answer: The program attempts to read missing.txt. The specific FileNotFoundException catch block handles missing files, while a general IOException catch handles other errors.

61. **Create a program appending to a file.**

Code:

```
import java.io.BufferedWriter;
import java.io.FileWriter;
import java.io.IOException;
public class FileAppend {
    public static void main(String[] args) {
        try (BufferedWriter bw = new BufferedWriter(new FileWriter("log.txt", true))) {
            bw.write("New log entry");
            bw.newLine();
        } catch (IOException e) {
            System.out.println("Error: " + e.getMessage());
        }
    }
}
```

Answer: The program appends text to log.txt using FileWriter with the true append flag. try-with-resources ensures proper resource closure.