```
display method in anonymous inner class
Enclosing class x : 10
Enclosing class constant : Hello
Overloaded main method
dummy
Enclosing block y : 20
Enclosing block z : 30
w variable shadowing : 50
```

Compile time error in case of using static variable which is not final:

```java
package com.demo.inner;

class Parent {
    public void display() {
        System.out.println("display method in parent class");
    }
}
```

```java
public class TestAnonymousInner {

    public static void main(String[] args) {

        Parent p = new Parent() {
            static final int a = 10;
            static int b = 50;
            public void display() {
                System.out.println("display method in anonymous inner class");
            }
        };

        p.display();
    }
}
```

Question 48: What is Constructor Chaining in java?

Answer: when one constructor calls another constructor, it is known as constructor chaining. This can be done in two ways:

- this() : it is used to call the same class constructor
- super() : it is used to call the parent class constructor

this() Example:

```java
public class Employee {

    private String name;
    private int age;
    private int salary;

    public Employee() {
        this("Mike");
    }
    public Employee(String name) {
        this(name, 20);
    }
    public Employee(String name, int age) {
        this(name, age, 20000);
    }
    public Employee(String name, int age, int salary) {
        this.name = name;
        this.age = age;
        this.salary = salary;
    }

    void print() {
        System.out.println("Employee name : " + name);
        System.out.println("Employee age : " + age);
        System.out.println("Employee salary : " + salary);
    }

    public static void main(String[] args) {
        Employee obj = new Employee();
        obj.print();
    }

}
```

Output:

```
Employee name : Mike
Employee age : 20
Employee salary : 20000
```

super() example:

```java
package com.test;

class Company {
    private String name;
    private int age;

    public Company(String name) {
        System.out.println("ABC Company");
        System.out.println("Details : ");
    }
    public Company(String name, int age) {
        this(name);
        this.name = name;
        this.age = age;
    }
    void display() {
        System.out.println("Name : " + name);
        System.out.println("Age : " + age);
    }
}
```

```java
class Employee extends Company {
    public Employee() {
        super("Mike", 20);
    }
    public Employee(String name) {
        this();
    }
}

public class ConstructorChaining {

    public static void main(String[] args) {
        Employee obj = new Employee();
        obj.display();
    }

}
```

Output:

```
ABC Company
Details :
Name : Mike
Age : 20
```

Some points to remember:

- this() can call same class constructor only
- super() can call immediate super class constructor only
- this() and super() call must be the first statement, because of this reason both cannot be used at the same time
- you must use super() to call the parent class constructor but if you do not provide a super() call, JVM will put it automatically

Question 49: What is init block?

Answer: init block is called instance initializer block

- init block is used to initialize the instance data members
- init block runs each time when object of the class is created
- init block runs in the order they appear in the program
- compiler replaces the init block in each constructor after the super() statement
- init block is different from static block which runs at the time of class loading

Example 1:

```
class TestInit {
    {
        System.out.println("init block called");
    }
    TestInit() {
        System.out.println("default constructor called");
    }
}

public class InitBlockTest {

    public static void main(String[] args) {
        new TestInit();
    }

}
```

Output:

```
init block called
default constructor called
```

Example 2:

```
ABC Company
Details :
Name : Mike
Age : 20
```

Some points to remember:

- this() can call same class constructor only
- super() can call immediate super class constructor only
- this() and super() call must be the first statement, because of this reason both cannot be used at the same time
- you must use super() to call the parent class constructor but if you do not provide a super() call, JVM will put it automatically

Question 49: What is init block?

Answer: init block is called instance initializer block

- init block is used to initialize the instance data members
- init block runs each time when object of the class is created
- init block runs in the order they appear in the program
- compiler replaces the init block in each constructor after the super() statement
- init block is different from static block which runs at the time of class loading

Example 1:

```
class TestInit {
    {
        System.out.println("init block called");
    }
    TestInit() {
        System.out.println("default constructor called");
    }
}

public class InitBlockTest {

    public static void main(String[] args) {
        new TestInit();
    }

}
```

Output:

```
init block called
default constructor called
```

Example 2:

```java
package com.test;

class Parent {
    {
        System.out.println("Parent class init block 1");
    }

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

    {
        System.out.println("Parent class init block 2");
    }
}
class Child extends Parent {
    {
        System.out.println("Child class init block 1");
    }

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

    {
        System.out.println("Child class init block 2");
    }
}
```

```java
public class InitBlockTest {

    public static void main(String[] args) {
        new Child();
    }

}
```

Output:

```
Parent class init block 1
Parent class init block 2
Parent constructor called
Child class init block 1
Child class init block 2
Child constructor called
```

Example 3:

```java
package com.test;

class Parent {
    {
        System.out.println("Parent class init block 1");
    }

    static {
        System.out.println("Parent class static block 1");
    }

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

    {
        System.out.println("Parent class init block 2");
    }

    static {
        System.out.println("Parent class static block 2");
    }
}

class Child extends Parent {
    {
        System.out.println("Child class init block 1");
    }

    static {
        System.out.println("Child class static block 1");
    }

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

    {
        System.out.println("Child class init block 2");
    }

    static {
        System.out.println("Child class static block 2");
    }
}

public class InitBlockTest {

    public static void main(String[] args) {
        new Child();
    }

}
```

Output:

```
Parent class static block 1
Parent class static block 2
Child class static block 1
Child class static block 2
Parent class init block 1
Parent class init block 2
Parent constructor called
Child class init block 1
Child class init block 2
Child constructor called
```

Order of execution:

- static blocks of super classes
- static blocks of the class
- init blocks of super classes
- constructors of super classes
- init blocks of the class
- constructors of the class

1. The code in static initialization block will be executed at class load time (and yes, that means only once per class load), before any instances of the class are constructed and before any static methods are called.
2. The instance initialization block is actually copied by the Java compiler into every constructor the class has. So, the code in instance initialization block is executed **exactly** before the code in constructor.

Question 50: What is called first, constructor or init block?

Answer: Constructor is invoked first. Compiler copies all the code of instance initializer block into the constructor after first statement super().

Question 51: What is Variable shadowing and Variable hiding in Java?

Answer: **Variable shadowing**: When a local variable inside a method has the same name as one of the instance variables, the local variable shadows the instance variable inside the method block.

Example 1:

```java
public class Test {
    //instance variables
    String name = "Mike";
    int age = 15;

    public void display() {
        //local variables
        String name = "John";
        int age = 20;

        System.out.println("Name : " + name);
        System.out.println("Age : " + age);
    }

    public static void main(String[] args) {
        Test obj = new Test();
        obj.display();
    }
}
```

Output:

```
Name : John
Age : 20
```

If you want to access instance variables then you can do so using 'this' keyword like below:

Example 2:

```java
public class Test {
    //instance variables
    String name = "Mike";
    int age = 15;

    public void display() {
        //local variables
        String name = "John";
        int age = 20;

        System.out.println("Name : " + name);
        System.out.println("Age : " + age);
        System.out.println("Name : " + this.name);
        System.out.println("Age : " + this.age);
    }
    public static void main(String[] args) {
        Test obj = new Test();
        obj.display();
    }
}
```

Output:

```
Name : John
Age : 20
Name : Mike
Age : 15
```

Variable Hiding: When the child and parent classes both have a variable with the same name, the child class variable hides the parent class variable.

Example 1:

```
package com.test;

class ParentClass {
    String x = "Parent's x";

    public void print() {
        System.out.println(x);
    }
}

class ChildClass extends ParentClass {
    String x = "Child's x";

    public void print() {
        System.out.println(x);
    }
}
public class Test {
    public static void main(String[] args) {
        ParentClass obj = new ChildClass();
        obj.print();
    }
}
```

Output:

```
Child's x
```

If you want to access parent's class variable then you can do this using super keyword:

Example 2:

```
package com.test;

class ParentClass {
    String x = "Parent's x";

    public void print() {
        System.out.println(x);
    }
}
class ChildClass extends ParentClass {
    String x = "Child's x";

    public void print() {
        System.out.println(x);
        System.out.println(super.x);
    }
}
```

```
public class Test {
    public static void main(String[] args) {
        ParentClass obj = new ChildClass();
        obj.print();
    }
}
```

Output:

```
Child's x
Parent's x
```

Variable hiding is not same as Method Overriding:

While variable hiding looks like overriding a variable (similar to method overriding), it is not. Overriding is applicable only to methods while hiding is applicable to variables.

In the case of method overriding, overridden methods completely replace the inherited methods, so when we try to access the method from a parent's reference by holding a child's object, the method from the child class gets called.

But in variable hiding, the child class hides the inherited variables instead of replacing them, so when we try to access the variable from the parent's reference by holding the child's object, it will be accessed from the parent class.

When an instance variable in a subclass has the same name as an instance variable in a super class, then the instance variable is chosen from the reference type.

```java
package com.test;

class ParentClass {
    String x = "Parent's x";

    public void print() {
        System.out.println(x);
    }
}

class ChildClass extends ParentClass {
    String x = "Child's x";

    public void print() {
        System.out.println(x);
    }
}
```

```java
public class Test {
    public static void main(String[] args) {
        ParentClass obj = new ChildClass();
        System.out.println(obj.x);
        obj.print();
    }
}
```

Output:

```
Parent's x
Child's x
```

Question 52: What is a constant and how we create constants in Java?

Answer: A constant is a variable whose value cannot change once it has been assigned. To make any variable a constant, we can use 'static' and 'final' modifier like below:

public static final TYPE NAME_OF_CONSTANT_VARIABLE = VALUE;

We can also use "enum" to define constants.

Question 53: Explain enum

Answer: enum in java is a data type which contains a fixed set of constants. In enum, we can also add variables, methods and constructors. Some common examples of enums are: days of week, colors, excel report columns etc.

Some points to remember:

- enum constants are static and final implicitly
- enum improves type safety
- enum can be declared inside or outside of a class
- enum can have fields, constructors (private) and methods
- enum cannot extend any class because it already extends Enum class implicitly but it can implement many interfaces
- We can use enum in switch statement
- We can have main() method inside an enum
- enum has values(), ordinal() and valueOf() methods. values() return an array containing all values present inside enum, ordinal() method returns the index of given enum value and valueOf() method returns the value of given constant enum
- enum can be traversed
- enum can have abstract methods
- enum cannot be instantiated because it contains private constructor only
- The constructor is executed for each enum constant at the time of enum class loading
- While defining enum, constants should be declared first, prior to any fields or methods, or else compile time error will come

Example 1:

main() method in enum, iterating over enum:

```
public enum Color {

    RED, GREEN, BLUE, BLACK;

    private Color() {
        System.out.println("Constructor called for : " + this.toString());
    }

    public static void main(String[]  args) {
        for(Color c : Color.values()) {
            System.out.println(c + " at index : " + c.ordinal());
        }
    }

}
```

Output:

```
Constructor called for : RED
Constructor called for : GREEN
Constructor called for : BLUE
Constructor called for : BLACK
RED at index : 0
GREEN at index : 1
BLUE at index : 2
BLACK at index : 3
```

Example 2:

```
package com.enumeration.demo;

enum Employee {
    Mike(15), John(20), Lisa(12), Dave(25);
    private int age;
    int getAge() {
        return age;
    }
    private Employee(int age) {
        this.age = age;
        System.out.println("Constructor called for : " + this.toString());
    }
}

public class Test {
    public static void main(String[] args) {
        System.out.println("Age of Lisa is : " + Employee.Lisa.getAge());
    }
}
```

Output:

```
Constructor called for : Mike
Constructor called for : John
Constructor called for : Lisa
Constructor called for : Dave
Age of Lisa is : 12
```

Question 54: What is Cloneable?

Answer: Cloneable is an interface in Java which needs to be implemented by a class to allow its objects to be cloned.

A class implements the Cloneable interface to indicate to the Object.clone() method that it is legal for that method to make a field-for-field copy of instances of that class.

If you try to Clone an object which doesn't implement the Cloneable interface, it will throw CloneNotSupportedException.

Example without implementing Cloneable interface:

Program 1:

```
public class Employee {

    private String name;
    private int age;

    public Employee(String name, int age) {
        this.name = name;
        this.age = age;
    }

    public String getName() {   return name;    }
    public int getAge() {   return age;     }
    public void setName(String name) {  this.name = name;   }
    public void setAge(int age) {   this.age = age;     }
```

```
    public static void main(String[] args) {
        Employee e1 = new Employee("Mike", 10);
        System.out.println("Employee 1, name : " + e1.getName());

        Employee e2 = e1;
        e2.setName("John");

        System.out.println("Employee 1, name : " + e1.getName());
        System.out.println("Employee 2, name : " + e2.getName());
    }

}
```

Output:

```
Employee 1, name : Mike
Employee 1, name : John
Employee 2, name : John
```

Here, we have created Employee object e1 with new keyword but then we created another object Employee e2 which has the same reference as of e1. So, any change in e2 object will reflect in e1 object and vice-versa.

Now, let's implement Cloneable interface in our Employee class and invoke the clone() method on e1 object to make its clone:

Program 2:

```
public class Employee implements Cloneable{
    private String name;
    private int age;

    public Employee(String name, int age) {
        this.name = name;
        this.age = age;
    }

    public String getName() {   return name;    }
    public int getAge() {   return age;     }
    public void setName(String name) {  this.name = name;   }
    public void setAge(int age) {   this.age = age;     }

    public static void main(String[] args)
            throws CloneNotSupportedException {

        Employee e1 = new Employee("Mike", 10);
        System.out.println("Employee 1, name : " + e1.getName());

        Employee e2 = (Employee) e1.clone();
        e2.setName("John");

        System.out.println("Employee 1, name : " + e1.getName());
        System.out.println("Employee 2, name : " + e2.getName());
    }
}
```

Output:

```
Employee 1, name : Mike
Employee 1, name : Mike
Employee 2, name : John
```

In the above example, we have only primitive types in our Employee class, what if we have an object type i.e. another class object reference, see the example below:

Program 3:

```
package com.clonealbe.demo;

class Company {
    private String name;
    public Company(String name) {
        this.name = name;
    }
    public String getName() {   return name;    }
    public void setName(String name) {  this.name = name;   }

}
```

```
public class Employee implements Cloneable{
    private String name;
    private int age;
    private Company company;

    public Employee(String name, int age, Company company) {
        this.name = name;
        this.age = age;
        this.company = company;
    }
    public String getName() {   return name;    }
    public int getAge() {   return age;     }
    public void setName(String name) {  this.name = name;   }
    public void setAge(int age) {   this.age = age;     }
    public Company getCompany() {   return company;     }
    public void setCompany(Company company) {   this.company = company;     }

    public static void main(String[] args) throws CloneNotSupportedException {
        Company c1 = new Company("Company_ABC");
        Employee e1 = new Employee("Mike", 10, c1);
        System.out.println("Employee 1, company name : " + e1.getCompany().getName());

        Employee e2 = (Employee) e1.clone();
        System.out.println("Employee 2, company name : " + e2.getCompany().getName());
        e2.getCompany().setName("XYZ");
        System.out.println("----------------------------");
        System.out.println("Employee 1, company name : " + e1.getCompany().getName());
        System.out.println("Employee 2, company name : " + e2.getCompany().getName());
    }
}
```

Can you guess the output of the last 2 sysout? Here is the output:

```
Employee 1, company name : Company_ABC
Employee 2, company name : Company_ABC
----------------------------
Employee 1, company name : XYZ
Employee 2, company name : XYZ
```

If you are surprised with the above output, then let me make it clear by saying that, by default Object's clone() method provide Shallow copy. This brings us to the next interview question: What is shallow copy and deep copy

Question 55: What is Shallow Copy and Deep Copy?

Answer: **Shallow Copy**: When we use the default implementation of clone() method, a shallow copy of object is returned, meaning if the object that we are trying to clone contains both primitive variables and non-primitive or reference type variable, then only the object's reference is copied not the entire object itself.

Consider this with the example:

Employee object is having Company object as a reference, now when we perform cloning on Employee object, then for primitive type variables, cloning will be done i.e. new instances will be created and copied to the cloned object but for non-primitive i.e. Company object, only the object's reference will be copied to the cloned object. It simply means Company object will be same in both original and cloned object, changing the value in one will change the value in other and vice-versa.

Now, if you want to clone the Company object also, so that your original and cloned Employee object will be independent of each other, then you have to perform Deep Copy.

Deep Copy: in Deep copy, the non-primitive types are also cloned to make the original and cloned object fully independent of each other.

Program 1:

```
package com.clonealbe.demo;

class Company implements Cloneable {
    private String name;
    public Company(String name) {
        this.name = name;
    }
    public String getName() {   return name;    }
    public void setName(String name) {  this.name = name;   }

    @Override
    protected Object clone() throws CloneNotSupportedException {
        return super.clone();
    }
}

public class Employee implements Cloneable {
    private String name;
    private int age;
    private Company company;

    public Employee(String name, int age, Company company) {
        this.name = name;
        this.age = age;
        this.company = company;
    }

    public String getName() {   return name;    }
    public int getAge() {   return age;     }
    public void setName(String name) {  this.name = name;   }
    public void setAge(int age) {   this.age = age;     }
    public Company getCompany() {   return company;     }
    public void setCompany(Company company) {   this.company = company;     }
```

```java
    @Override
    protected Object clone() throws CloneNotSupportedException {
        Employee employee = (Employee) super.clone();
        employee.company = (Company) company.clone();
        return employee;
    }
    public static void main(String[] args) throws CloneNotSupportedException {
        Company c1 = new Company("Company_ABC");
        Employee e1 = new Employee("Mike", 10, c1);
        System.out.println("Employee 1, company name : " + e1.getCompany().getName());

        Employee e2 = (Employee) e1.clone();
        System.out.println("Employee 2, company name : " + e2.getCompany().getName());
        e2.getCompany().setName("XYZ");
        System.out.println("---------------------------");
        System.out.println("Employee 1, company name : " + e1.getCompany().getName());
        System.out.println("Employee 2, company name : " + e2.getCompany().getName());
    }
}
```

Output:

```
Employee 1, company name : Company_ABC
Employee 2, company name : Company_ABC
---------------------------
Employee 1, company name : Company_ABC
Employee 2, company name : XYZ
```

In above example, we have overridden the clone method in our employee class and we called the clone method on mutable company object.

We can also use ***Copy constructor*** to perform deep copy:

Program 2:

```java
package com.clonealbe.demo;

class Company {
    private String name;
    public Company(String name) {
        this.name = name;
    }
    public String getName() {   return name;    }
    public void setName(String name) {  this.name = name;   }

}

public class Employee {
    private String name;
    private int age;
    private Company company;
    public Employee(String name, int age, Company company) {
        this.name = name;
        this.age = age;
        this.company = company;
    }

    //Copy constructor
    public Employee(Employee emp) {
        this.name = emp.getName();
        this.age = emp.getAge();
        Company company = new Company(emp.getCompany().getName());
        this.company = company;
    }
```

```
    public String getName() {    return name;     }
    public int getAge() {    return age;      }
    public void setName(String name) {  this.name = name;    }
    public void setAge(int age) {    this.age = age;      }
    public Company getCompany() {    return company;      }
    public void setCompany(Company company) {    this.company = company;      }

    public static void main(String[] args) {
        Company c1 = new Company("Company_ABC");
        Employee e1 = new Employee("Mike", 10, c1);
        System.out.println("Employee 1, company name : " + e1.getCompany().getName());

        //Invoking copy constructor
        Employee e2 = new Employee(e1);
        System.out.println("Employee 2, company name : " + e2.getCompany().getName());
        e2.getCompany().setName("XYZ");
        System.out.println("---------------------------");
        System.out.println("Employee 1, company name : " + e1.getCompany().getName());
        System.out.println("Employee 2, company name : " + e2.getCompany().getName());
    }
}
```

Output:

```
Employee 1, company name : Company_ABC
Employee 2, company name : Company_ABC
---------------------------
Employee 1, company name : Company_ABC
Employee 2, company name : XYZ
```

There are 2 other methods by which you can perform deep copy:

- By using Serialization, where you serialize the original object and returns the deserialized object as a clone
- By using external library of Apache Commons Lang. Apache Common Lang comes with SerializationUtils.clone() method for performing deep copy on an object. It expects all classes in the hierarchy to implement Serializable interfaces else SerializableException is thrown by the system

Question 56: What is Serialization and De-serialization?

Answer: Serialization is a mechanism to convert the state of an object into a byte stream while De-serialization is the reverse process where the byte stream is used to recreate the actual object in memory. The byte stream created is platform independent that means objects serialized on one platform can be deserialized on another platform.

To make a Java Object serializable, the class must implement Serializable interface. Serializable is a Marker interface. ObjectOutputStream and ObjectInputStream classes are used for Serialization and Deserialization in java.

We will serialize the below Employee class:

```
import java.io.Serializable;

public class Employee implements Serializable{
    private String name;
    private int age;
    private transient int salary;

    public Employee(String name, int age, int salary) {
        this.name = name;
        this.age = age;
        this.salary = salary;
    }

    @Override
    public String toString() {
        return "Employee [name=" + name + ", age=" + age
                + ", salary=" + salary + "]";
    }
}
```

SerializationDemo.java:

```
public class SerializationDemo {
    public static void main(String[] args) {
        Employee emp = new Employee("Mike", 15, 20000);
        String file = "C:\\temp\\byteStream.txt";
        try {
            FileOutputStream fos = new FileOutputStream(file);
            ObjectOutputStream oos = new ObjectOutputStream(fos);
            oos.writeObject(emp);

            fos.close();
            oos.close();
            System.out.println("Employee object is serialized : " + emp);
        } catch (IOException e1) {
            System.out.println("IOException is caught");
        }

        try {
            FileInputStream fis = new FileInputStream(file);
            ObjectInputStream ois = new ObjectInputStream(fis);
            Employee emp1 = (Employee) ois.readObject();

            fis.close();
            ois.close();
            System.out.println("Employee object is de-serialized : " + emp1);
        } catch (IOException e) {
            System.out.println("IOException is caught");
        } catch (ClassNotFoundException e) {
            System.out.println("ClassNotFoundException is caught");
        }
    }
}
```

Output:

```
Employee object is serialized : Employee [name=Mike, age=15, salary=20000]
Employee object is de-serialized : Employee [name=Mike, age=15, salary=0]
```

Here, while de-serializing the employee object, salary is 0, that is because we have made salary variable to be 'transient'. 'static' and 'transient' variables do not take part in Serialization process. During de-serialization, transient variables will be initialized with their default values i.e. if objects, it will be null and if "int", it will be 0 and static variables will be having the current value.

And if you look at the file present in C:/temp/bytestream.txt, you can see how the object is serialized into this file,

Question 57: What is SerialVersionUID?

Answer: **SerialVersionUID**: The serialization process at runtime associates an id with each Serializable class which is known as SerialVersionUID. It is used to verify the sender and receiver of the serialized object. The sender and receiver must have the same SerialVersionUID, otherwise, *InvalidClassException* will be thrown when you deserialize the object. A Serializable class can declare its own UID explicitly by declaring a field. It must be static, final and of type long. *Remember, there is an exception for SerialVersionUID that although it is static, it gets serialized too, so that at the object deserialization the sender and receiver can be verified.*

If a serializable class doesn't explicitly declare a serialVersionUID, then the serialization runtime will calculate a default one for that class based on various aspects of class. This default serialVersionUID gets changed, when you add a new field or remove the transient keyword from a variable or convert the static variable to non-static variable. And if you are modifying the class