Question 25: How to make custom checked / unchecked exception?

Answer: If you want to make a custom unchecked exception class then extend *RuntimeException* and for creating a custom checked exception class, extend *Exception* class.

Question 26: What happens when you throw an exception from finally block?

Answer: When exception is thrown from finally block, then it takes precedence over the exceptions that are thrown from try/catch block

Question 27: What will be Output of below program related to try-catch-finally?

```
class MyException1 extends Exception { }
class MyException2 extends Exception { }

public class DemoException {
    public static void method1() throws Exception {
        try{
            System.out.println("5");
            throw new MyException1();
        } catch (Exception e) {
            System.out.println("6");
            throw new MyException2();
        } finally {
            System.out.println("7");
            throw new Exception();
        }
    }
```

```
    public static void main(String[] args) throws Exception {
        try {
            System.out.println("1");
            method1();
            System.out.println("2");
        } catch (Exception e) {
            System.out.println("3");
            throw new MyException2();
        } finally {
            System.out.println("4");
            throw new MyException1();
        }
    }
}
```

With everything you have read so far, you should be able to answer this easily. I will leave this one for you.

Question 28: Explain try-with-resources

Answer: try-with-resources concept was introduced in Java 7. It allows us to declare resources which will be used inside the try block and it assures us that the resources will be closed after execution of this block. A resource is an object that must be closed after finishing the program. The resources declared must implement *AutoCloseable* interface.

Syntax:

```
public class Demo {
    public static void main(String[] args) {
        try (FileReader reader = new FileReader("C:\\temp\\dummy.txt");){

            //program statements

        } catch (Exception e) {
            System.out.println(e.getMessage());
        }
    }
}
```

Question 29: Why String is Immutable?

Answer: String is immutable for below reasons:

Question 28: Explain try-with-resources

Answer: try-with-resources concept was introduced in Java 7. It allows us to declare resources which will be used inside the try block and it assures us that the resources will be closed after execution of this block. A resource is an object that must be closed after finishing the program. The resources declared must implement *AutoCloseable* interface.

Syntax:

```
public class Demo {
    public static void main(String[] args) {
        try (FileReader reader = new FileReader("C:\\temp\\dummy.txt");){

            //program statements

        } catch (Exception e) {
            System.out.println(e.getMessage());
        }
    }
}
```

Question 29: Why String is Immutable?

Answer: String is immutable for below reasons:

1. **String Pool**: String Pool is possible only because String is Immutable in Java. String pool is a special storage area in Java heap. If the string is already present in the pool, then instead of creating a new object, old object's reference is returned. This way different String variables can refer to the same reference in the pool, thus saving a lot of heap space also. If String is not immutable then changing the string with one reference will lead to the wrong values to other string variables having the same reference.
2. **Security**: String parameters are used in network connections, database URL's, username and passwords etc. Because String is immutable, these values can't be changed. Otherwise any hacker could change the referenced values which will cause severe security issues in the application.
3. **Multi-threading**: Since String is immutable, it is safe for multithreading. A single String instance can be shared across different threads. This avoids the use of synchronization for thread safety. Strings are implicitly thread-safe.
4. **Caching**: The hashcode of string is frequently used in Java. Since string is immutable, the hashcode will remain the same, so it can be cached without worrying about the changes. This makes it a great candidate for using it as a Key in Map.
5. **Class Loaders**: Strings are used in Java ClassLoaders and since String is made immutable, it provides security that correct class is being loaded.

Question 30: What does the equals() method of String class do?

Answer: As we know, Object class is the parent of all classes, and Object class has a equals() method that compares the reference of two objects, but String class has overridden this method, and String class's equals() method compares the contents of two strings.

Program:

```
public class Test {
    public static void main(String[] args) {
        String s1 = "Mark";
        String s2 = "John";
        String s3 = "Mark";

        System.out.println("s1.equals(s2): " + s1.equals(s2));
        System.out.println("s1.equals(s3): " + s1.equals(s3));
    }
}
```

Output:

```
s1.equals(s2): false
s1.equals(s3): true
```

Question 31: Explain StringBuffer and StringBuilder

Answer: Both StringBuffer and StringBuilder classes are used for String manipulation. These are mutable objects. But StringBuffer provides thread-safety as all its methods are synchronized, this makes performance of StringBuffer slower as compared to StringBuilder.

StringBuffer class is present from Java 1.0, but due to its slower performance, StringBuilder class was introduced in Java 1.5

If you are in a single-threaded environment or don't care about thread safety, you should use StringBuilder. Otherwise, use StringBuffer for thread-safe operations.

Question 32: Explain the output of below program related to equals() method of StringBuilder

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

        StringBuilder sb1 = new StringBuilder("hello");
        StringBuilder sb2 = new StringBuilder("hello");

        if(sb1.equals(sb2)) {
            System.out.println("Equal");
        } else {
            System.out.println("Not Equal");
        }

    }
}
```

Output:

```
Not Equal
```

Answer: This is another very famous interview question. If you were expecting 'Equal' as output, then you were wrong. The output is not 'Equal' because StringBuffer and StringBuilder does not override equals and hashcode methods. In the above program, Object's class equals() method is getting used and as it compares the reference of two objects, the output of above program is 'Not Equal'.

Since hashcode is used in data structures that use hashing algorithm to store the objects. Examples are HashMap, HashSet, HashTable, ConcurrentHashMap etc. and all these data structures requires their keys not to be changed so that stored values can be found by using hashcode method but StringBuffer/StringBuilder are mutable objects. This makes them a very poor choice for this role.

You must have heard about the equals and hashcode contract in Java, which states that if two objects are equals according to equals() method then their hashcode must be same, vice-versa is not true. Now, had the equals method for StringBuilder/StringBuffer be overridden, their corresponding hashcode method would also need to be overridden to follow that rule. But as explained earlier, these classes don't need to have their own hashcode implementation and hence same is with their equals method.

Question 33: When to use String/StringBuffer/StringBuilder

Answer: You should use String class if you require immutability, use StringBuffer if you require mutability + Thread safety and use StringBuilder if you require mutability and no thread safety.

Question 34: Explain equals and hashcode contract

Answer: The equals and hashcode contract says:

- If two objects are equals according to equals() method, then their hashcode must be same but reverse is not true i.e. if two objects have same hashcode then they may/may not be equals.

Question 35: What is Marker Interface?

Answer: Marker interface is an interface which is empty. Some of the Marker interfaces are Cloneable, Serializable, Remote etc. If you have read that Marker interfaces indicate something to the compiler or JVM, then you have read it wrong, it has nothing to do with JVM.

Consider the example of Cloneable:

It is said that you cannot call clone() method on a class object unless the class implements Cloneable interface. Well this statement is true, because when you call clone() method then the first statement in clone() is, *obj instanceOf Cloneable*.

The class object on which clone() method is getting called is checked, whether the class implements Cloneable interface or not, by using instanceOf operator. If class does not implement Cloneable interface then CloneNotSupportedException is thrown.

Same is true with writeObject(Object) method of ObjectOutputStream class. Here, *obj instanceOf Serializable* is used to check whether the class implements Serializable interface or not. If class does not implement Serializable interface then NotSerializableException is thrown.

Question 36: Can you write your own custom Marker interface?

Answer: Yes. As you already know that Marker interfaces have got nothing to do with indicating some signal to JVM or compiler, instead it is just a mere check of using instanceOf operator to know whether the class implements Marker interface or not.

In your method, you can put a statement like: object instanceOf MyMarkerInterface.

You can use Marker interface for classification of your code.

Question 37: What is Comparable and Comparator?

Answer: Both Comparable and Comparator interfaces are used to sort the collection of objects. These interfaces should be implemented by your custom classes, if you want to use Arrays/Collections class sorting methods.

Comparable interface has compareTo(Obj) method, you can override this method in your class, and you can write your own logic to sort the collection.

General rule to sort a collection of objects is:

If 'this' object is less than passed object, return negative integer.

If 'this' object is greater than passed object, return positive integer.

If 'this' object is equal to the passed object, return zero.

Comparable Example:

Employee.java

```
public class Employee implements Comparable<Employee> {

    private int id;
    private String name;
    private int age;
    private long salary;

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

    public int getId() { return id; }
    public String getName() { return name; }
    public int getAge() { return age; }
    public long getSalary() { return salary; }

    public void setId(int id) { this.id = id; }
    public void setName(String name) { this.name = name; }
    public void setAge(int age) { this.age = age; }
    public void setSalary(long salary) { this.salary = salary; }

    @Override
    public int compareTo(Employee obj) {
        return this.id - obj.id;
    }

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

ComparableDemo.java:

```
public class ComparableDemo {

    public static void main(String[] args) {
        List<Employee> empList = new ArrayList<>();

        empList.add(new Employee(4, "Dave", 25, 28000));
        empList.add(new Employee(20, "Mike", 20, 10000));
        empList.add(new Employee(9, "Abhi", 32, 5000));
        empList.add(new Employee(1, "Lisa", 40, 19000));

        Collections.sort(empList);
        System.out.println(empList);
    }

}
```

Output:

```
[Employee [id=1, name=Lisa, age=40, salary=19000]
, Employee [id=4, name=Dave, age=25, salary=28000]
, Employee [id=9, name=Abhi, age=32, salary=5000]
, Employee [id=20, name=Mike, age=20, salary=10000]
]
```

Here, we have sorted the Employee list based on 'id' attribute.

Now, if we want to sort the employee list based on any other attribute, say name, we will have to change our compareTo() method implementation for this. So, Comparable allows only single sorting mechanism.

But Comparator allows sorting based on multiple parameters. We can define another class which will implement Comparator interface and then we can override it's compare(Obj, Obj) method.

Suppose we want to sort the Employee list based on name and salary.

NameComparator.java:

```
public class NameComparator implements Comparator<Employee> {

    @Override
    public int compare(Employee emp1, Employee emp2) {
        return emp1.getName().compareTo(emp2.getName());
    }

}
```

String class already implements Comparable interface and provides a lexicographic implementation for compareTo() method which compares 2 strings based on contents of characters or you can say in lexical order. Here, Java will determine whether passed String object is less than, equal to or greater than the current object.

ComparatorDemo.java:

```
public class ComparatorDemo {

    public static void main(String[] args) {
        List<Employee> empList = new ArrayList<>();

        empList.add(new Employee(4, "Dave", 25, 28000));
        empList.add(new Employee(20, "Mike", 20, 10000));
        empList.add(new Employee(9, "Abhi", 32, 5000));
        empList.add(new Employee(1, "Lisa", 40, 19000));

        Collections.sort(empList, new NameComparator());
        System.out.println(empList);
    }

}
```

Output:

```
[Employee [id=9, name=Abhi, age=32, salary=5000]
, Employee [id=4, name=Dave, age=25, salary=28000]
, Employee [id=1, name=Lisa, age=40, salary=19000]
, Employee [id=20, name=Mike, age=20, salary=10000]
]
```

The output list is sorted based on employee's names.

SalaryComparator.java:

```
import java.util.Comparator;

public class SalaryComparator implements Comparator<Employee> {

    @Override
    public int compare(Employee emp1, Employee emp2) {
        return (int) (emp1.getSalary() - emp2.getSalary());
    }

}
```

ComparatorDemo.java:

```
public class ComparatorDemo {

    public static void main(String[] args) {
        List<Employee> empList = new ArrayList<>();

        empList.add(new Employee(4, "Dave", 25, 28000));
        empList.add(new Employee(20, "Mike", 20, 10000));
        empList.add(new Employee(9, "Abhi", 32, 5000));
        empList.add(new Employee(1, "Lisa", 40, 19000));

        Collections.sort(empList, new SalaryComparator());
        System.out.println(empList);
    }

}
```

Output:

```
[Employee [id=9, name=Abhi, age=32, salary=5000]
, Employee [id=20, name=Mike, age=20, salary=10000]
, Employee [id=1, name=Lisa, age=40, salary=19000]
, Employee [id=4, name=Dave, age=25, salary=28000]
]
```

Question 38: How to compare a list of Employees based on name and age such that if name of the employee is same then sorting should be based on age

Answer: When comparing by name, if both names are same, then comparison will give 0. If the compare result is 0, we will compare based on age.

NameAgeComparator.java:

```
public class NameAgeComparator implements Comparator<Employee> {

    @Override
    public int compare(Employee emp1, Employee emp2) {
        int flag = emp1.getName().compareTo(emp2.getName());
        if(flag == 0) {
            flag = emp1.getAge() - emp2.getAge();
        }
        return flag;
    }

}
```

ComparatorDemo.java:

```
public class ComparatorDemo {

    public static void main(String[] args) {
        List<Employee> empList = new ArrayList<>();

        empList.add(new Employee(4, "Dave", 25, 28000));
        empList.add(new Employee(20, "Mike", 20, 10000));
        empList.add(new Employee(9, "Abhi", 32, 5000));
        empList.add(new Employee(1, "Lisa", 40, 19000));
        empList.add(new Employee(15, "Mike", 25, 15000));
        empList.add(new Employee(8, "Mike", 30, 20000));

        Collections.sort(empList, new NameAgeComparator());
        System.out.println(empList);
    }

}
```

Output:

```
[Employee [id=9, name=Abhi, age=32, salary=5000]
, Employee [id=4, name=Dave, age=25, salary=28000]
, Employee [id=1, name=Lisa, age=40, salary=19000]
, Employee [id=20, name=Mike, age=20, salary=10000]
, Employee [id=15, name=Mike, age=25, salary=15000]
, Employee [id=8, name=Mike, age=30, salary=20000]
]
```

Question 39: Difference between Comparable and Comparator

Answer:

- Comparable interface can be used to provide single way of sorting whereas Comparator interface is used to provide multiple ways of sorting
- Comparable interface is present in 'java.lang' package whereas Comparator interface is present in 'java.util' package
- For using Comparable, the class needs to implement Comparable interface whereas for using Comparator, there is no need to make changes in the class
- Comparable provides compareTo() method to sort elements, whereas Comparator provides compare() method to sort elements
- We can sort the list elements of Comparable type by using Collections.sort(listObj) method, whereas to sort the list elements of Comparator type, we have to provide a Comparator object like, Collections.sort(listObj, Comparator)

At times, when you are using any third-party classes or the classes where you are not the author of the class, then in that case Comparator is the only choice to sort those objects

Question 40: Different methods of Object class

Answer: Object class sits at the top of class hierarchy tree. Every class is a child of Object class. Below methods are present inside Object class:

```
//Creates and returns a copy of this object
protected native Object clone() throws CloneNotSupportedException

//Indicates whether some other object is "equal to" this one
public boolean equals(Object obj)

//Returns a hash code value for the object
public native int hashCode()

//Returns the runtime class of an Object
public final native Class<?> getClass()

//Returns a string representation of the object
public String toString()

//Called by the garbage collector on an object when garbage collection
//determines that there are no more references to the object
protected void finalize() throws Throwable

//Wakes up a single thread that is waiting on this object's monitor
public final native void notify()

//Wakes up all threads that are waiting on this object's monitor
public final native void notifyAll()

public final native void wait(long timeout) throws InterruptedException

public final void wait(long timeout, int nanos) throws InterruptedException

public final void wait() throws InterruptedException
```

Question 41: What type of arguments are allowed in System.out.println() method?

Answer: println() method of PrintStream class is overloaded, and it accepts below arguments :

```
public class DemoPrintln {

    public static void main(String[] args) {

        System.out.println;

    }

}
```

println() : void - PrintStream
println(boolean x) : void - PrintStream
println(char x) : void - PrintStream
println(char[] x) : void - PrintStream
println(double x) : void - PrintStream
println(float x) : void - PrintStream
println(int x) : void - PrintStream
println(long x) : void - PrintStream
println(Object x) : void - PrintStream
println(String x) : void - PrintStream

Question 42: Explain System.out.println() statement

Answer:

- System is a class in java.lang package
- out is a static member of System class and is an instance of java.io.PrintStream
- println() is a method of PrintStream class

Question 43: Explain Auto-boxing and Un-boxing

Answer: In Java 1.5, the concepts of Auto-boxing and Un-boxing were introduced to automatically convert primitive type to object and vice-versa.

When Java automatically converts a primitive type, like int into its corresponding wrapper class object i.e. Integer, then this is called Auto-boxing. While the opposite of this is called Un-boxing, where an Integer object is converted into primitive type int.

Example:

```
public class Test {
    public static void main(String[] args) {
        ArrayList<Integer> list = new ArrayList<>();

        //Auto-boxing : int to Integer
        list.add(5);
        list.add(10);

        //Unboxing : Integer to int
        int a = list.get(0);

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

Output:

```
5
```

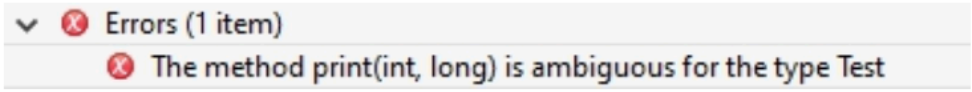
Question 44: Find the output of below program

```
public class Test {
    public void print(int a, long b) {
        System.out.println("Method 1");
    }

    public void print(long a, int b) {
        System.out.println("Method 2");
    }

    public static void main(String[] args) {
        Test obj = new Test();
        obj.print(5, 10);
    }
}
```

Answer: Here, the compiler is confused as to which method to be called, so it throws Compile Time error.



Question 45: Can you pass primitive long value in switch statement?

Answer: No, switch works only with 4 primitives and their wrappers, as well as with the enum type and String class:

- byte and Byte
- short and Short
- char and Character
- int and Integer
- enum
- String

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

        long num = 5;

        switch(num) {

        }
    }
}
```

Question 46: Explain static keyword in Java

Answer: In Java, a static member is a member of a class that isn't associated with an instance of a class. Instead, the member belongs to the class itself.

In Java, Static is applicable for the following:

- Variable
- Method

- Block
- Nested class

Static Variable: if any variable is declared as static, then it is known as 'static variable'. Only single copy of the variable gets created and all instances of the class share same static variable. The static variable gets memory only once in the class area at the time of class loading.

When to use static variable: static variables should be used to declare common property of all objects as only single copy is created and shared among all class objects, for example, the company name of employees etc.

Static Method: When a method is declared with static keyword then it is known as static method. These methods belong to the class rather than the object of the class. As a result, a static method can be directly accessed using class name without the need of creating an object.

One of the basic rules of working with static methods is that you can't access a non-static method or field from a static method because the static method doesn't have an instance of the class to use to reference instance methods or fields. Another restriction is, 'this' and 'super' cannot be used in static context.

For example: main() method is static, Java Runtime uses this method to start an application without creating an object.

Static Block: Static block gets executed exactly once when the class is first loaded, use static block to initialize the static variables.

Static nested classes:

Static nested classes are a type of inner class in java where the inner class is static. Static nested classes can access only the static members of the outer class. The advantage of using static nested classes is that it makes the code more readable and maintainable.

In the case of normal inner class, you cannot create inner class object without first creating the outer class object, but in the case of static inner class, there can be a static inner class object without the outer class object.

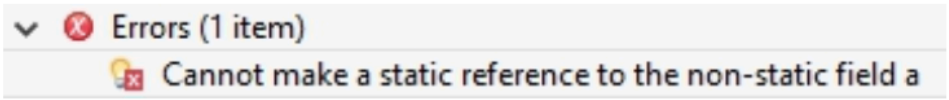
How to create object of static inner class:

```
OuterClass.StaticNestedClass nestedClassObject = new OuterClass.StaticNestedClass();
```

Compile Time Error comes when we try to access non-static member inside static nested class:

```
class OuterClass {
    int a = 10;
    static int b = 20;
    private static int c = 30;

    static class InnerClass {
        void print() {
            System.out.println("Outer class variable a : " + a);
            System.out.println("Outer class variable b : " + b);
            System.out.println("Outer class variable c : " + c);
        }
    }
}
```



Using inner class object:

```
class OuterClass {
    int a = 10;
    static int b = 20;
    private static int c = 30;

    static class InnerClass {
        void print() {
            //System.out.println("Outer class variable a : " + a);
            System.out.println("Outer class variable b : " + b);
            System.out.println("Outer class variable c : " + c);
        }
    }
}
public class StaticNestedTestClass {
    public static void main(String[] args) {
        OuterClass.InnerClass innerClassObject=new OuterClass.InnerClass();
        innerClassObject.print();
    }
}
```

Output:

```
Outer class variable b : 20
Outer class variable c : 30
```

If you have static members in your Static Inner class then there is no need to create the inner class object:

```
class OuterClass {
    static int x = 20;

    static class InnerClass {
        static int y = 30;

        static void display() {
            System.out.println("Outer x : " + x);
        }
    }
}
public class StaticNestedTestClass {
    public static void main(String[] args) {
        OuterClass.InnerClass.display();
        System.out.println(OuterClass.InnerClass.y);
    }
}
```

Output:

```
Outer x : 20
30
```

Question 47: What is an Inner Class in Java, how it can be instantiated and what are the types of Inner Classes?

Answer: In Java, when you define one non-static class within another class, it is called Inner Class/ Nested Class. Inner class enables you to logically group classes that are only used in one place, thus, this increases the use of encapsulation and creates more readable and maintainable code.

Java inner class is associated with the object of the class and they can access all the variables and methods of the outer class. Since inner classes are associated with the instance, we can't have any static variables in them.

The object of java inner class is part of the outer class object and to create an instance of the inner class, we first need to create an instance of outer class.

Java Inner classes can be instantiated like below:

OuterClass outerObject = new OuterClass();

OuterClass.InnerClass innerObject = outerObject.new InnerClass();

Example:

```
class OuterClass {
    static int outer_x = 10;
    int outer_y = 20;
    private int outer_z = 30;

    class InnerClass {
        void print() {
            System.out.println("outer_x : " + outer_x);
            System.out.println("outer_y : " + outer_y);
            System.out.println("outer_z : " + outer_z);
        }
    }
}
public class InnerClassDemo {
    public static void main(String[] args) {
        OuterClass outerClass = new OuterClass();
        OuterClass.InnerClass innerClass = outerClass.new InnerClass();

        innerClass.print();
    }
}
```

Output:

```
outer_x : 10
outer_y : 20
outer_z : 30
```

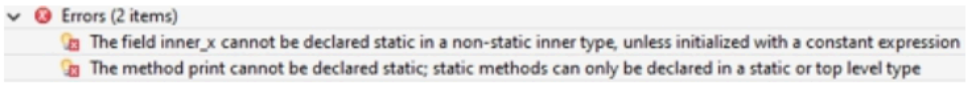
Compile time error when static variable and static method is present in Inner class:

```
class OuterClass {

    class InnerClass {
        static int inner_x = 20;

        static void print() {

        }
    }
}
```



There are 2 special types of Inner Classes:

- local inner class
- anonymous inner class

local inner class: Local Inner Classes are the inner classes that are defined inside a block. Generally, this block is a method body. Sometimes this block can be a *for loop*, or an if clause. Local Inner classes are not a member of any enclosing classes. They belong to the block in which they are defined in, due to which local inner classes cannot have any access modifiers associated with them.

However, they can be marked as final or abstract. These classes have access to the fields of the class enclosing it. Local inner class must be instantiated in the block they are defined in.

Points to remember:

- local inner class cannot be instantiated from outside of the block where they are defined
- local inner class has access to the members of the enclosing class
- till Java 1.7, local inner class can access only final local variable of the enclosing block where they are defined. But from Java 1.8 onwards, it is possible to access the non-final local variable of the enclosing block
- the scope of local inner class is restricted to the block where they are defined
- A local inner class can extend an abstract class or can implement an interface

Example:

```
class OuterClass {
    int outer_x = 10;
    static int outer_y = 20;
    private String outer_name = "Mike";

    public void print() {
        int non_final_block_level_j = 30;
        final int final_block_level_k = 40;

        class Inner {
            public void display() {
                System.out.println("outer_x : " + outer_x);
                System.out.println("outer_y : " + outer_y);
                System.out.println("outer_name : " + outer_name);

                System.out.println("non_final_block_level_j : " + non_final_block_level_j);
                System.out.println("non_final_block_level_j : " + final_block_level_k);
            }
        }

        Inner inner = new Inner();
        inner.display();
    }
}
```

```
public class InnerClassDemo {
    public static void main(String[] args) {
        OuterClass outerClass = new OuterClass();
        outerClass.print();
    }
}
```

Output:

```
outer_x : 10
outer_y : 20
outer_name : Mike
non_final_block_level_j : 30
non_final_block_level_j : 40
```

Remember, you can only access the block level variables, and cannot change them. You will get compile time error if you try to change them:

```
class OuterClass {
    public void print() {
        int print_j = 10;
        final int print_k = 20;

        class Inner {
            public void display() {
                print_j = 50;
            }
        }

        Inner inner = new Inner();
        inner.display();
    }
}
```

Errors (1 item)

Local variable print_j defined in an enclosing scope must be final or effectively final

A variable whose value is not changed once initialized is called as ***effectively final variable***.

Anonymous inner class:

An inner class that does not have any name is called Anonymous Inner class. You should use them when you want to use local class only once. It does not have a constructor since there is no class name and it cannot be declared as static.

Generally, they are used when you need to override the method of a class or an interface.

***When to use*:**

Example 1 : Let's understand this by an example, Suppose you are want to return a list of employee class objects and they should be sorted based on employee name, now for this you can write a comparator in a separate class and pass its object inside the Collections.sort(list, comparatorObject)

Instead, you can use the anonymous inner class and you don't have to create a new class just for writing a comparator that you are not going to use later on.

Program:

Employee.java:

```
public class Employee {
    private int id;
    private String name;
    private int age;
    private long salary;

    public Employee(int id, String name, int age, long salary) {
        this.id = id;
        this.name = name;
        this.age = age;
        this.salary = salary;
    }
    public int getId() { return id; }
    public String getName() { return name; }
    public int getAge() { return age; }
    public long getSalary() { return salary; }

    public void setId(int id) { this.id = id; }
    public void setName(String name) { this.name = name; }
    public void setAge(int age) { this.age = age; }
    public void setSalary(long salary) { this.salary = salary; }

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

AnonymousInnerDemo.java:

```
public class AnonymousInnerDemo {
    public static void main(String[] args) {
        List<Employee> empList = new ArrayList<>();

        empList.add(new Employee(4, "Dave", 25, 28000));
        empList.add(new Employee(20, "Mike", 20, 10000));
        empList.add(new Employee(9, "Abhi", 32, 5000));
        empList.add(new Employee(1, "Lisa", 40, 19000));
        empList.add(new Employee(15, "Mike", 25, 15000));
        empList.add(new Employee(8, "Mike", 30, 20000));

        Collections.sort(empList, new Comparator<Employee>() {

            @Override
            public int compare(Employee emp1, Employee emp2) {
                return emp1.getName().compareTo(emp2.getName());
            }

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

Output:

```
[Employee [id=9, name=Abhi, age=32, salary=5000]
, Employee [id=4, name=Dave, age=25, salary=28000]
, Employee [id=1, name=Lisa, age=40, salary=19000]
, Employee [id=20, name=Mike, age=20, salary=10000]
, Employee [id=15, name=Mike, age=25, salary=15000]
, Employee [id=8, name=Mike, age=30, salary=20000]
]
```

Example 2: Using anonymous inner class, you can implement a Runnable also

```
public class RunnableInnerDemo {

    public static void main(String[] args) {

        Thread t = new Thread(new Runnable() {

            @Override
            public void run() {
                System.out.println("Thread 1");
            }
        });

        t.start();

        System.out.println("Main Thread");
    }
}
```

Output:

```
Main Thread
Thread 1
```

Example 3: You can use anonymous inner class in situations where you want to override the parent class method without creating a separate child class:

```
package com.demo.inner;

class Parent {
    public void display() {
        System.out.println("display method in parent class");
    }
}
public class TestAnonymousInner {
    public static void main(String[] args) {

        Parent p = new Parent() {
            public void display() {
                System.out.println("display method in anonymous inner class");
            }
        };

        p.display();
    }
}
```

Output:

```
display method in anonymous inner class
```

Some points to remember:

- anonymous class does not have a constructor as it does not have any class name
- in anonymous class, we cannot have static members except static constants
- an anonymous class has access to the members of its enclosing class
- an anonymous class cannot access local variables in its enclosing scope(block) that are not final or effectively final

Anonymous Inner Class Example:

```
package com.demo.inner;

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

public class TestAnonymousInner {

    private int x = 10;
    private static String HELLO = "Hello";
    public static void main() {
        System.out.println("Overloaded main method");
    }
    public void dummy() {
        System.out.println("dummy");
    }
    private void print() {
        int y = 20;
        final int z = 30;
        int w = 40;
        Parent p = new Parent() {
            static final int w = 50;
            public void display() {
                System.out.println("display method in anonymous inner class");
                System.out.println("Enclosing class x : " + x);
                System.out.println("Enclosing class constant : " + HELLO);
                main();
                dummy();
                System.out.println("Enclosing block y : " + y);
                System.out.println("Enclosing block z : " + z);
                System.out.println("w variable shadowing : " + w);
            }
        };

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

Output: