Question 1: What are the 4 pillars of OOPS?

Answer: 4 pillars of OOPS are:

1. Abstraction
2. Encapsulation
3. Inheritance
4. Polymorphism

Let's take a look at them:

1. **Abstraction**: Abstraction is a process of hiding the implementation details and showing only functionality to the user.

   Real world examples:

   - TV remote: To start the TV, you have to press the power button, you don't have to know about the internal circuit operations like how infrared waves are passing.
   - Car gears: We know what happens when we change the gear. But we don't know how changing gear works under the hood, that information is irrelevant to us, so it is abstracted.

   In java, Abstraction can be achieved in two ways:

   - Abstract classes
   - Interfaces

2. **Encapsulation**: Encapsulation is a process of *Binding data and methods within a class.* Think of it like showing the essential details of a class by using the access control modifiers (*public, private, protected*). So, we can say that Encapsulation leads to the desired level of Abstraction.

   Example:

   Java Bean, where all data members are made private and you define certain public methods to the outside world to access them.

3. **Inheritance**: Using inheritance means defining a parent-child relationship between classes, by doing so, you can reuse the code that is already defined in the parent class. Code reusability is the biggest advantage of Inheritance.

Java does not allow multiple inheritance through classes but it allows it through interfaces.

4. **Polymorphism**: Poly means many and Morph means forms. Polymorphism is the process in which an object or function takes different forms. There are 2 types of Polymorphism :

- Compile Time Polymorphism (Method Overloading)
- Run Time Polymorphism (Method Overriding)

In Method overloading, two or more methods in one class have the same method name but different arguments. It is called as *Compile time polymorphism* because it is decided at compile time which overloaded method will be called.

Overriding means when we have two methods with same name and same parameters in parent and child class. Through overriding, child class can provide specific implementation for the method which is already defined in the parent class.

Question 2: What is an abstract class?

Answer: A class that is declared using "abstract" keyword is known as abstract class. It can have abstract methods (methods without body) as well as concrete methods (methods with body).

Some points to remember:

- An abstract class cannot be instantiated, which means you are not allowed to create an object of the abstract class. This also means, an abstract class has no use unless it is extended by some other class

- If there is any abstract method in a class then that class must be declared abstract
- The first non-abstract class which is extending from an abstract class will have to give implementation of the abstract methods defined in abstract class

Example:

```
package com.tech;

abstract class MyAbstractClass {

    //abstract method
    abstract void print();

    //concrete method
    public void display() {
        System.out.println("In display method");
    }

}
```

```
public class AbstractDemo extends MyAbstractClass {

    @Override
    void print() {
        System.out.println("In print method");
    }

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

}
```

Output:

```
In print method
In display method
```

Question 3: Does Abstract class have constructor?

Answer: This is a famous interview question and the answer is: Yes, abstract classes have constructor. Either you can provide it or the default one will be provided by Java. Now, you must be wondering if you cannot create an object of abstract class then what is the need of a constructor.

One thing you must know is that the constructors are used when you are creating an object of a class, to initialize the data members of that class and your abstract class can have data members.

Now, when your class extends abstract class then the same abstract class will become super class for your extending class and remember when you have constructor of your class then first line of your constructor is always a call to super class constructor and this is the time when your abstract class constructor will get called.

Example 1:

```
package com.tech;

abstract class MyAbstractClass {

    public MyAbstractClass() {
        System.out.println("inside MyAbstractClass constructor");
    }

}

public class AbstractDemo extends MyAbstractClass {

    public AbstractDemo() {
        System.out.println("inside AbstractDemo constructor");
    }

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

}
```

Output:

```
inside MyAbstractClass constructor
inside AbstractDemo constructor
```

Example 2:

```
package com.tech;

abstract class MyAbstractClass {

    public int a;
    public int b;

    public MyAbstractClass(int a, int b) {
        this.a = a;
        this.b = b;
    }

    public void print() {
        System.out.println("a = " + a);
        System.out.println("b = " + b);
    }

}
```

```
public class AbstractDemo extends MyAbstractClass {

    public AbstractDemo(int x, int y) {
        super(x, y);

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

}
```

Output:

```
a = 5
b = 10
```

Question 4: What is an Interface?

Answer:

- An interface in Java is a blueprint of a class. It has static constants and abstract methods.
- Interface specify what a class must do but not how to do
- An interface is like defining a contract that is fulfilled by implementing classes
- An interface is used to achieve full abstraction.
- All methods in an interface are public and abstract by default and all variables declared in an interface are constants i.e. public, static and final
- A class which implements an interface will have to provide implementation of all the methods that are defined in the interface
- A class can implement more than one interface, this is how Java allows multiple inheritance.
- Since Java 8, we can have default and static methods in an interface

Question 5: Difference between abstract class and interface

Answer: The differences are:

- Abstract class can have both abstract and concrete methods but interface can only have abstract methods (Java 8 onwards, it can have default and static methods as well)
- Abstract class methods can have access modifiers other than public but interface methods are implicitly public and abstract
- Abstract class can have final, non-final, static and non-static variables but interface variables are only static and final

- A subclass can extend only one abstract class but it can implement multiple interfaces
- An Abstract class can extend one other class and can implement multiple interfaces but an interface can only extend other interfaces

In this question, the interviewer may try to confuse you by saying that from Java 8 onwards, you can have static and default methods in an Interface so now what is the difference between abstract class and interface and the answer you should tell is – We can still extend only one class but can implement multiple interfaces.

Question 6: What to choose – interface or abstract class

Answer: Consider these points while choosing between the two:

- When you want to provide default implementation to some of the common methods that can be used directly by the sub-classes then you can use abstract class because it can have concrete methods also, this is not the case with Interface because the child classes that are implementing this interface will have to provide implementation for all the methods that are declared in the interface
- If your contract keeps on changing then Interface will create problems because then you will have to provide implementation of those new methods in all the implementing classes, whereas with abstract class you can provide one default implementation to the new methods and only change those implementing classes that are actually going to use these new methods

Most of the times, interfaces are a good choice. It is also one of the best practices, when you code in terms of interfaces.

Question 7: Why Java 8 has introduced default methods?

Answer: To extend the capability of an already existing interface, default methods are introduced in Java 8. Let's understand this by one example:

Consider there are 100 classes that are implementing one interface. Now you want to define one new method inside your interface. In this case you will have to change all the implementation classes to fulfill the interface contract. So, Java introduced default methods, here you can provide default implementation of that new method inside your interface and as it is not mandatory to provide implementation of default methods by the implementing classes, all the 100 classes can use the default implementation or if they want they can provide their own implementation by overriding the default method.

Now consider one interesting scenario: You have two interfaces, *Interface1* and *Interface2* both having default method *hello()* and one class is implementing these 2 interfaces without giving implementation to this default method. You see the problem here? Yes, it is the famous ***Diamond Problem*** (Refer to *Question 9*, if you're not already familiar with this problem).

```java
interface Interface1 {
    default void hello() {
        System.out.println("Hello from Interface1");
    }
}

interface Interface2 {
    default void hello() {
        System.out.println("Hello from Interface2");
    }
}

public class Child implements Interface1, Interface2 {

}
```

So, to avoid this error, it is mandatory to provide implementation for common default methods of interfaces

```java
public class Child implements Interface1, Interface2 {
    @Override
    public void hello() {
        System.out.println("inside Child class hello method");
        Interface1.super.hello();
    }

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

Output:

```
inside Child class hello method
Hello from Interface1
```

Question 8: Why Java 8 has introduced static methods?

Answer: Consider an example where you want to define a utility class, what you usually do is you define a class which contains static methods and then you call these methods using class name. Now, Java 8 onwards you can do the same thing using an Interface by giving only static methods inside your interface. This way of using Interface for defining utility classes is better as it helps in performance also, because using a class is more expensive operation than using an interface.

Question 9: Why Java does not allow multiple inheritance?

Answer: Multiple inheritance occurs when a class has more than one parent classes.

Why Java does not allow this: let us consider there are 2 parent classes having a method named *hello()* with same signature and one child class is extending these 2 classes, if you call this *hello()* method which is same in both parents, which parent class method will get executed – it results into an ambiguous situation, this is also called **Diamond Problem**.

You will get a compile time error if you try to extend more than one class.

```
class Parent1 {
    public void hello() {
        System.out.println("Hello from Parent1 class");
    }
}

class Parent2 {
    public void hello() {
        System.out.println("Hello from Parent2 class");
    }
}

public class Child extends Parent1, Parent2 {

}
```

Question 10: What are the rules for Method Overloading and Method Overriding?

Answer: **Method Overloading Rules:** Two methods can be called overloaded if they follow below rules:

- Both have same method name
- Both have different arguments

If both methods follow above two rules, then they may or may not:

- Have different access modifiers
- Have different return types
- Throw different checked or unchecked exceptions

Method Overriding Rules: The overriding method of child class must follow below rules:

- It must have same method name as that of parent class method
- It must have same arguments as that of parent class method
- It must have either the same return type or covariant return type (child classes are covariant types to their parents)
- It must not throw broader checked exceptions
- It must not have a more restrictive access modifier (if parent method is public, then child method cannot be private/protected)

Question 11: Can we override final methods?

Answer: No, final methods cannot be overridden.

Question 12: Can constructors and private methods be overridden?

Answer: No

Question 13: What is final keyword and where it can be used?

Answer: If you use final with a primitive type variable, then its value cannot be changed once assigned.

If you use final with a method, then you cannot override it in the subclass.

If you use final with class, then that class cannot be extended.

If you use final with an object type, then that object cannot be referenced again.

Question 14: What is exception and exception handling?

Answer: An exception is an event that disrupts the normal flow of the program. It is an object which is thrown at runtime, so exception handling is a mechanism by which normal flow of the program is maintained.

Program showing the exception is thrown:

```
public class TestException {
    public static void main(String[] args) {
        System.out.println("Program Started");
        int a = 15/0;
        System.out.println("Program End");
    }
}
```

Output:

```
Program Started
Exception in thread "main" java.lang.ArithmeticException: / by zero
        at com.exception.TestException.main(TestException.java:6)
```

Program showing that the exception is handled:

```
public class TestException {
    public static void main(String[] args) {
        System.out.println("Program Started");
        try{
            int a = 15/0;
        } catch (ArithmeticException e) {
            System.out.println("Exception handled");
        }
        System.out.println("Program End");
    }
}
```

Output:

```
Program Started
Exception handled
Program End
```

Question 15: Difference between error and exception

Answer: **Error**: Errors in a program are irrecoverable, they indicate that something severe has gone wrong in the application and the program gets terminated in case of error occurrence e.g. running out of memory: *OutOfMemoryError*, making too many recursive calls: *StackOverflowError* etc.

Exception: Exceptions on the other hand are something that we can recover from by handling them properly e.g.: trying to access a property/method from a null object: *NullPointerException*, dividing an integer by zero: *ArithmeticException* etc.

Question 16: What are the different types of exceptions?

Answer: There are 2 types of exceptions:

- **Checked Exceptions:** All exceptions other than *RuntimeException* and Error are known as Checked exception. These exceptions are checked by the compiler at the compile time itself. E.g. when you are trying to read from a file, then compiler enforces us to handle the *FileNotFoundException* because it is possible that the file may not be present. Some other checked exceptions are *SQLException*, *IOException* etc.

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

        try {
            FileReader f = new FileReader("C:\\temp\\dummy.txt");
        } finally {
            System.out.println("Inside finally block");
        }

    }
}
```

Errors (1 item)

Unhandled exception type FileNotFoundException

- **Unchecked Exceptions:** Runtime Exceptions are known as Unchecked exceptions. Compiler does not force us to handle these exceptions but as a programmer, it is our responsibility to handle runtime exceptions e.g. *NullPointerException*, *ArithmeticException*, *ArrayIndexOutOfBoundException* etc.

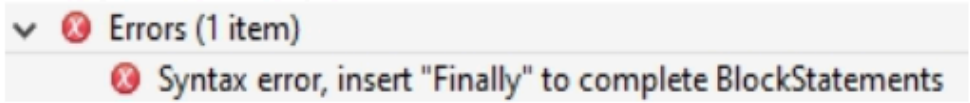
Question 17: How exception handling is done in java?

Answer: try-catch block is used for exception handling. If you think that certain statements may throw an exception, surround them with try block.

A try block is always followed by a catch block or finally or both.

You cannot use the try block alone:

```
public class DemoException {

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

}
```



Question 18: Can we write a try block without catch block?

Answer: Yes, we can write a try block with finally, but we cannot write a try block alone.

Question 19: How to handle multiple exceptions together?

Answer: You can write multiple catch blocks one after another for each exception or you can write a single catch block using a pipe symbol (|) to separate the exceptions.

While writing multiple catch block, you have to follow the below rule:

- Handle the most specific exception and then move down to the most generic ones, means you cannot handle Exception (base class of exception) before FileNotFoundException

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

        try {
            FileReader f = new FileReader("C:\\temp\\dummy.txt");
        } catch (Exception e) {
            e.printStackTrace();
        }catch (FileNotFoundException e) {
            e.printStackTrace();
        } finally {
            System.out.println("Inside finally block");
        }

    }
}
```

Suppose, your method is throwing more than one exception and you want to perform some specific action based on the exception thrown, you should use multiple catch blocks in this case.

Example using multiple catch blocks:

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

        try {
            FileReader f = new FileReader("C:\\temp\\dummy.txt");
        } catch (FileNotFoundException e) {
            System.out.println("Action when File is not found");
        } catch (NullPointerException e) {
            System.out.println("Action for NullPointerException");
        } catch (Exception e) {
            System.out.println("Action for exceptions other than "
                    + "FileNotFoundException/NullPointerException");
        } finally {
            System.out.println("Inside finally block");
        }

    }
}
```

When using pipe (|) symbol:

```
public class DemoException {

    public static void main(String[] args) {

        try {
            FileReader f = new FileReader("C:\\temp\\dummy.txt");
        } catch (FileNotFoundException | NullPointerException e ) {
            System.out.println("Same action for both");
        }  finally {
            System.out.println("Inside finally block");
        }

    }

}
```

Question 20: When finally block will not get executed

Answer:

- when System.exit() is called
- when JVM crashes

Question 21: Difference between throw and throws keyword. And discuss Exception Propagation

Answer:

- ***throw*** is a keyword which is used to explicitly throw an exception in the program, inside a function or inside a block of code, whereas ***throws*** is a keyword which is used with the method signature to declare an exception which might get thrown by the method while executing the code
- throw keyword is followed by an instance of an Exception class whereas throws is followed by Exception class names
- You can throw one exception at a time but you can declare multiple exceptions using throws keyword

- Using throw keyword, only unchecked exceptions are propagated, whereas using throws keyword both checked and unchecked exceptions can be propagated.

Exception propagation:

An exception is first thrown from the top of the stack and if it is not caught, it drops down the call stack to the previous method, If not caught there, the exception again drops down to the previous method, and so on until they are caught or until they reach the very bottom of the call stack. This is called exception propagation.

- When method m1() calls method m2() which calls method m3(), a stack is formed which gets unfold from the top, so if method m3() throws an exception and it is not handled there, it will drop down the call stack to method m2(), if it is not handled there, it will drop down the call stack to method m1(), this happens until we reach the bottom of the stack or until the exception is caught. This is called Exception Propagation in java.

Example: unchecked exception is thrown and it can be seen from the call stack that it is propagated

```
public class DemoException {

    public static void method1() {
        method2();
    }

    public static void method2() {
        throw new ArithmeticException("Arithmetic Exception from method2");
    }

    public static void main(String[] args) {
        method1();
    }

}
```

Output:

```
Exception in thread "main" java.lang.ArithmeticException: Arithmetic Exception from method2
        at com.tech.DemoException.method2(DemoException.java:10)
        at com.tech.DemoException.method1(DemoException.java:6)
        at com.tech.DemoException.main(DemoException.java:14)
```

Example: **Here the unchecked exception is handled**

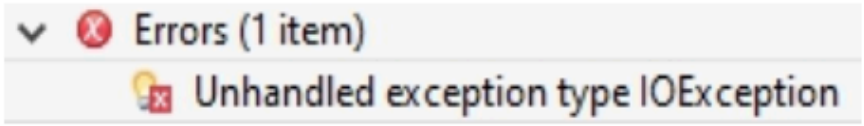
```
public class DemoException {

    public static void method1() {
        method2();
    }

    public static void method2() {
        throw new ArithmeticException("Arithmetic Exception from method2");
    }

    public static void main(String[] args) {
        try {
            method1();
        } catch (ArithmeticException e) {
            System.out.println("Exception handled");
        }
    }

}
```

Output:

```
Exception handled
```

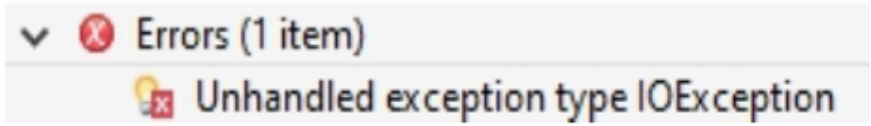
Example: ***checked exceptions are not propagated down the call chain by default,***

```
public class DemoException {

    public static void method1() {
        throw new IOException("IO Exception occurred");
    }

    public static void main(String[] args) {
        method1();
    }

}
```



You have to use **throws keyword** if you want to propagate the checked exception, like

```
public class DemoException {

    public static void method1() throws IOException {
        throw new IOException("IO Exception occurred");
    }

    public static void main(String[] args) {
        method1();
    }

}
```



Notice in the above example that the checked exception is propagated and now it is the responsibility of the caller method to either handle the exception or throw it further. Below example is showing that the **checked exception is handled**,

```
public class DemoException {
    public static void method1() throws IOException {
        throw new IOException("IO Exception occurred");
    }

    public static void main(String[] args) {
        try {
            method1();
        } catch (IOException e) {
            System.out.println("Exception handled");
        }
    }
}
```

Output:

```
Exception handled
```

throws can be used with unchecked exceptions also, though it is of no use because unchecked exceptions are by default propagated. See this in the below program:

```java
public class DemoException {
    public static void method1() throws ArithmeticException{
        int a = 10/0;
    }

    public static void main(String[] args) {
        try {
            method1();
        } catch (ArithmeticException e) {
            System.out.println("Exception handled");
        }
    }
}
```

Output:

```
Exception handled
```

Unchecked exceptions are by default propagated:

```java
public class DemoException {
    public static void method1() {
        int a = 10/0;
    }

    public static void main(String[] args) {
        try {
            method1();
        } catch (ArithmeticException e) {
            System.out.println("Exception handled");
        }
    }
}
```

Output:

```
Exception handled
```

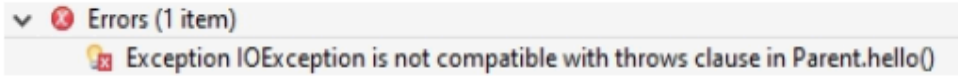
Question 22: Exception handling w.r.t. method overriding

Answer:

- If the parent class method does not declare an exception then child class overridden method cannot declare checked exceptions but it can declare unchecked exceptions
- If the parent class method declares an exception, then child class overridden method
  - can declare no exception
  - can declare same exception
  - can declare a narrower exception (more broader exception declaration than parent one is not allowed)

Example when parent class method does not declare an exception and child class declares checked exception:

```java
package com.exception;

import java.io.IOException;

class Parent {
    public void hello() {
        System.out.println("Parent class hello method");
    }
}

class Child extends Parent {
    public void hello() throws IOException{
        System.out.println("Child class hello method");
    }
}
```



Example when parent class method does not declare an exception and child class declares unchecked exception:

```java
package com.exception;

class Parent {
    public void hello() {
        System.out.println("Parent class hello method");
    }
}

class Child extends Parent {
    public void hello() throws ArithmeticException{
        System.out.println("Child class hello method");
    }
}

public class TestException {
    public static void main(String[] args) {
        Parent p = new Child();
        p.hello();
    }
}
```

Output:

Child class hello method

Example when Child class overridden method throws a broader exception than the parent one:

```java
package com.exception;

class Parent {
    public void hello() throws ArithmeticException {
        System.out.println("Parent class hello method");
    }
}

class Child extends Parent {
    public void hello() throws Exception {
        System.out.println("Child class hello method");
    }
}
```

Example when child class overridden method throws same exception as the parent one:

```
package com.exception;

class Parent {
    public void hello() throws Exception {
        System.out.println("Parent class hello method");
    }
}

class Child extends Parent {
    public void hello() throws Exception {
        System.out.println("Child class hello method");
    }
}
```

```
public class TestException {
    public static void main(String[] args) {
        Parent p = new Child();
        try {
            p.hello();
        } catch (Exception e) {
            System.out.println("Handled");
        }
    }
}
```

Output:

```
Child class hello method
```

Example when child class overridden method declares a narrower exception than the parent one:

```java
import java.io.IOException;

class Parent {
    public void hello() throws Exception {
        System.out.println("Parent class hello method");
    }
}

class Child extends Parent {
    public void hello() throws IOException {
        System.out.println("Child class hello method");
    }
}
```

```java
public class TestException {
    public static void main(String[] args) {
        Parent p = new Child();
        try {
            p.hello();
        } catch (Exception e) {
            System.out.println("Handled");
        }
    }
}
```

Output:

```
Child class hello method
```

Example when parent class method declares an exception and child class overridden method does not declare any exception:

```java
package com.exception;

class Parent {
    public void hello() throws Exception {
        System.out.println("Parent class hello method");
    }
}

class Child extends Parent {
    public void hello() {
        System.out.println("Child class hello method");
    }
}
```

```java
public class TestException {
    public static void main(String[] args) {
        Parent p = new Child();
        try {
            p.hello();
        } catch (Exception e) {
            System.out.println("Handled");
        }
    }
}
```

Output:

```
Child class hello method
```

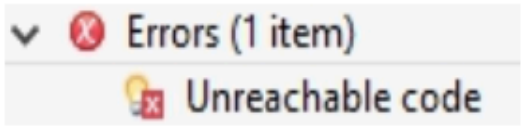
Question 23: Programs related to Exception handling and return keyword

Answer: These questions are asked a lot of times to the new programmers.

One thing you should remember is, if you write anything after return statement / throw exception statement, then that will give Compile time error as 'Unreachable code'.

Program 1:

```
package com.tech;

public class DemoException {
    public static int method1() {
        try {
            throw new ArithmeticException();
            return 1;
        } catch(Exception e) {
            return 2;
        } finally {
            return 3;
        }
    }

    public static void main(String[] args) {
        int result = method1();
        System.out.println(result);
    }
}
```



Program 2: finally block is always executed

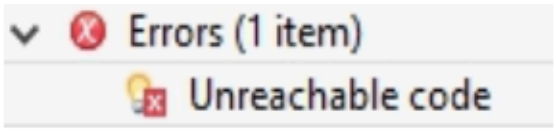
```
package com.tech;

public class DemoException {
    public static int method1() {
        try {
            return 1;
        } catch(Exception e) {
            return 2;
        } finally {
            return 3;
        }
    }
    public static void main(String[] args) {
        int result = method1();
        System.out.println(result);
    }
}
```

Output:

```
3
```

Program 3: program execution returns from the finally block

```
public class DemoException {
    public static int method1() {
        try {
            return 1;
        } catch(Exception e) {
            return 2;
        } finally {
            return 3;
        }
        return 4;
    }
    public static void main(String[] args) {
        int result = method1();
        System.out.println(result);
    }
}
```



Program 4:

```
public class DemoException {
    public static int method1() {
        try {
            int a = 15/0;
            return 1;
        } catch(Exception e) {
            return 2;
        } finally {
            return 3;
        }
    }
    public static void main(String[] args) {
        int result = method1();
        System.out.println(result);
    }
}
```

Output:

```
3
```

Program 5:

```
public class DemoException {
    public static int method1() {
        try {
            int a = 15/0;
            return 1;
        } catch(Exception e) {
            return 2;
        }
    }
    public static void main(String[] args) {
        int result = method1();
        System.out.println(result);
    }
}
```

Output:

2

Question 24: How to make your own custom exception class?

Answer: In java, you can create your own custom exception class which are basically derived classes from the Exception class.

For Example:

```
package com.tech;

class MyException extends Exception {
    public MyException(String s) {
        super(s);
    }
}
```

```
public class DemoException {

    public static void main(String[] args) {
        try {
            method1();
        } catch (MyException e) {
            System.out.println(e.getMessage());
        }
        System.out.println("Program end");
    }

    public static void method1() throws MyException {
        throw new MyException("My own custom exception");
    }
}
```

Output:

```
My own custom exception
Program end
```