Java Exceptions and Error Handling

1. Four access modifiers in Java:

   i) Public
   ii) Protected
   iii) Default
   iv) Private

   ## Public:

   - When a class, method, or variable is marked as public, it can be accessed from any other class in the same package or from any other package.
   - This is the broadest level of access, allowing unrestricted visibility and accessibility.

   ## Features:

   - Members are accessible from anywhere in the program.
   - Provides the broadest level of accessibility.
   - Ideal for creating reusable components and exposing APIs.

   ## Example:

```
public class PublicExample {

public int publicVar = 42;
}
```

   ## Protected:

   - Members (methods and variables) with the protected access modifier are accessible within the same package and by subclasses (whether they are in the same package or not).
   - Outside the package, protected members can only be accessed by subclasses of the class in which they are declared.

Features:

- Members are accessible within the same package and subclasses, regardless of package
- Encourages controlled sharing among related classes and subclasses.
- Balances encapsulation with limited accessibility.

Example:

```
package mypackage;
class Parent {
protected int protectedVar = 42;
}
class Child extends Parent {
void someMethod() {
 System.out.println(protectedVar); // Accessible in subclass
}
}
```

Default (no modifier):

- If no access modifier is specified, the default access modifier applies.
- Members with default access are accessible only within the same package.
- They are not accessible outside the package, even to subclasses.

Features:

- Members are accessible only within the same package.
- Provides a level of encapsulation within the package.
- Helpful in creating package-level functionality or restricting access to certain components.

Example:

```
package mypackage;
class DefaultExample {
int defaultVar = 42; // Default access modifier
}
```

Private:

- Members marked as private are accessible only within the same class.
- They cannot be accessed by any other class, including subclasses.

Features:

- Members are accessible only within the same class.
  Ensures the highest level of encapsulation.
- Protects sensitive information from unauthorized access.

  Example:

```
class PrivateExample {
 private int privateVar = 42;

 }
```

The significance of these access modifiers in terms of class, method, and variable accessibility can be summarized as follows:

Class Accessibility:

- Public classes can be accessed from anywhere.
- Default and protected classes can be accessed only within the same package.
- Private classes cannot be accessed from outside the declaring class.

Method Accessibility:

- Public methods can be called from anywhere.
- Protected methods can be called within the same package or by subclasses, including those outside the package.
- Default methods can be called only within the same package.
- Private methods can be called only within the declaring class.

Variable Accessibility:

- Public variables can be accessed from anywhere.
- Protected variables can be accessed within the same package or by subclasses, including those outside the package.
- Default variables can be accessed only within the same package.
- Private variables can be accessed only within the declaring class.

Overall, access modifiers provide control over the visibility and accessibility of classes, methods, and variables, contributing to encapsulation, security, and proper design in Java programming.

2. Difference between Exception and Error:

Aspect	Exception	Error
Nature	Typically unexpected and recoverable occurrences.	Often indicates a severe problem that might not be recoverable.
Handling	Can be caught and handled using try-catch blocks.	Generally not intended to be caught or handled programmatically.
Type	Subclass of Exception.	Subclass of Throwable.
Examples	NullPointerException, ArrayIndexOutOfBoundsException.	OutOfMemoryError, StackOverflowError.

Causes	Usually caused by faulty logic or exceptional conditions.	Often caused by system failures, resource exhaustion, or critical errors.
Impact	Generally less severe and can be anticipated in code.	Can lead to application crashes or unexpected termination.
Resolution	Often resolved through exception handling mechanisms.	Requires fixing underlying issues in code or environment.
Message	Provides information about what went wrong in the program.	Typically signifies a critical problem without detailed explanation.
Handling Strategy	Encourages programmers to handle exceptional situations gracefully.	Often requires debugging and fixing the root cause.

Example:

Scenario	Exception Handling	Error Handling
File not found	Handle with try-catch block, prompt user to provide correct file path.	Log error, notify user about inability to proceed.

Out of memory	Unlikely to handle programmatically, might require architectural changes.	Log error, perform necessary cleanup, terminate gracefully.
Divide by zero	Handle with try-catch block, provide alternative computation or error message.	Log error, notify user, investigate code for mathematical logic.

Example Programs:

1. Exception Example:

```
public class DivideByZeroExample {
   public static void main(String[] args) {
      int numerator = 10;
      int denominator = 0;
      try {
         int result = numerator / denominator;
         System.out.println("Result: " + result);
      } catch (ArithmeticException e) {
         System.out.println("Error: Division by zero");
      }
   }
}
```

In this example, if the denominator is 0, it will throw an `ArithmeticException`, which is a type of exception. We handle this exception by catching it with a try-catch block.

2. Error Example:

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

public class OutOfMemoryExample {
```

```
    public static void main(String[] args) {
        List<Integer> list = new ArrayList<>();
        try {
            while (true) {
                list.add(100);     // Keep adding integers to the list
            }
        } catch (OutOfMemoryError e) {
            System.out.println("Error: Out of memory");
        }
    }
}
```

In this example, we keep adding integers to an ArrayList until we run out of memory. When this happens, it throws an `OutOfMemoryError`, which is a type of error. We catch this error with a try-catch block, but typically, errors like these are not caught and handled programmatically in real-world applications.

3. Difference between Checked Exception and Unchecked Exception:

Aspect	Checked Exception	Unchecked Exception
Definition	Exceptions that are checked at compile time.	Exceptions that are not checked at compile time.
Handling Requirement	Must be either caught or declared in the method's	Can be caught or ignored. They are not required
	signature with a throws clause.	to be declared.

Examples	IOException, ClassNotFoundException, SQLException	NullPointerException, ArrayIndexOutOfBoundsExcepti on,
		IllegalArgumentException
Forced Handling	Must be handled by either catching the exception	Not necessarily handled by the programmer. They
	or declaring it in the method's signature.	may be left unhandled.
Compile-Time Checking	Compiler enforces the handling or declaration of	Compiler does not enforce handling or declaration
	checked exceptions.	of unchecked exceptions.
Use Case	Usually used for recoverable conditions like I/O	Typically used for programming errors or
	errors, database errors, etc.	conditions that could have been avoided by proper
		programming practices.
Impact on Code Clarity	Often makes code more verbose due to required	Code may be less cluttered as handling code is not
	handling or declaration.	forced.

Example:

Checked Exception Example (IOException):

```
import java.io.*;

public class FileReaderExample {
   public static void main(String[] args) {
      try {
         readFile("example.txt");
      } catch (FileNotFoundException e) {
         System.out.println("File not found: " + e.getMessage());
      } catch (IOException e) {
         System.out.println("Error reading file: " + e.getMessage());
      }
   }

   public static void readFile(String fileName) throws IOException {
      BufferedReader br = null;
      try {
         br = new BufferedReader(new FileReader(fileName));
         String line;
         while ((line = br.readLine()) != null) {
            System.out.println(line);
         }
      } finally {
         if (br != null) {
            try {
               br.close();
            } catch (IOException e) {
               System.out.println("Error closing BufferedReader: " +
e.getMessage());
            }
         }
      }
   }
}
```

In this example, `FileNotFoundException` is a checked exception that must be either caught or declared in the method signature. It is thrown when a file with the specified pathname does not exist. `IOException` is also a checked exception and is thrown during the file reading process if an error occurs.

Unchecked Exception Example (NullPointerException):

```
public class NullPointerExceptionExample {
    public static void main(String[] args) {
        String str = null;
        try {
            int length = str.length();
            System.out.println("Length of the string: " + length);
        } catch (NullPointerException e) {
            System.out.println("NullPointerException caught: " + e.getMessage());
        }
    }
}
```

In this example, `NullPointerException` is an unchecked exception that occurs at runtime. It is thrown when attempting to access or perform operations on an object reference that is `null`. In this case, trying to access the `length()` method of a `null` string causes a `NullPointerException`. Since it's an unchecked exception, it's not mandatory to catch it or declare it in the method signature.