Java Exceptions

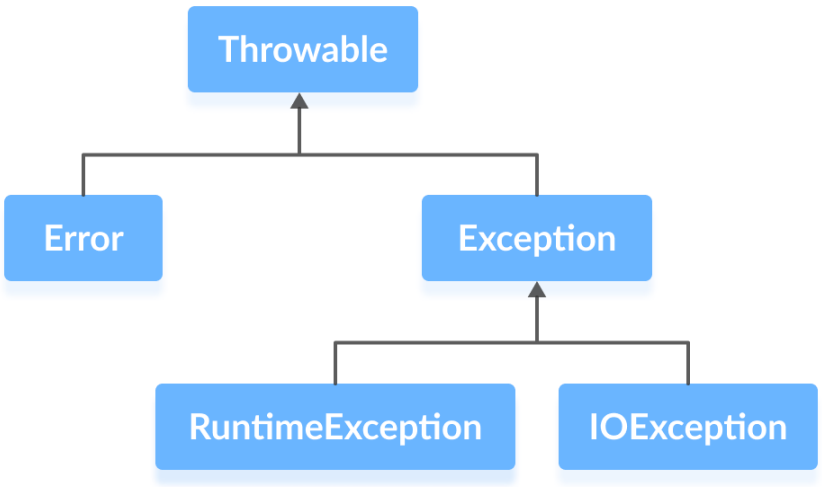
An exception is an unexpected event that occurs during program execution. It affects the flow of the program instructions which can cause the program to terminate abnormally.

An exception can occur for many reasons. Some of them are:

- Invalid user input
- Device failure
- Loss of network connection
- Physical limitations (out of disk memory)
- Code errors
- Opening an unavailable file

Java Exception hierarchy

Here is a simplified diagram of the exception hierarchy in Java.



As you can see from the image above, the `Throwable` class is the root class in the hierarchy.

Note that the hierarchy splits into two branches: Error and Exception.

Errors

Errors represent irrecoverable conditions such as Java virtual machine (JVM) running out of memory, memory leaks, stack overflow errors, library incompatibility, infinite recursion, etc.

Errors are usually beyond the control of the programmer and we should not try to handle errors.

Exceptions

Exceptions can be caught and handled by the program.

When an exception occurs within a method, it creates an object. This object is called the exception object.

It contains information about the exception such as the name and description of the exception and state of the program when the exception occurred.

Java Exception Types

The exception hierarchy also has two branches: RuntimeException and IOException.

1. RuntimeException

A **runtime exception** happens due to a programming error. They are also known as **unchecked exceptions**.

These exceptions are not checked at compile-time but run-time. Some of the common runtime exceptions are:

- Improper use of an API - IllegalArgumentException
- Null pointer access (missing the initialization of a variable) - NullPointerException
- Out-of-bounds array access - ArrayIndexOutOfBoundsException
- Dividing a number by 0 - ArithmeticException

You can think about it in this way. “If it is a runtime exception, it is your fault”.

The NullPointerException would not have occurred if you had checked whether the variable was initialized or not before using it.

An ArrayIndexOutOfBoundsException would not have occurred if you tested the array index against the array bounds.

2. IOException

An IOException is also known as a **checked exception**. They are checked by the compiler at the compile-time and the programmer is prompted to handle these exceptions.

Some of the examples of checked exceptions are:

- Trying to open a file that doesn't exist results in FileNotFoundException
- Trying to read past the end of a file

Java Exception Handling

In the last tutorial, we learned about Java exceptions. We know that exceptions abnormally terminate the execution of a program.

This is why it is important to handle exceptions. Here's a list of different approaches to handle exceptions in Java.

- try...catch block
- finally block
- throw and throws keyword

1. Java try...catch block

The try-catch block is used to handle exceptions in Java. Here's the syntax of try...catch block:

```
try {
  // code
}
catch(Exception e) {
  // code
}
```

Here, we have placed the code that might generate an exception inside the try block. Every try block is followed by a catch block.

When an exception occurs, it is caught by the catch block. The catch block cannot be used without the try block.

Example: Exception handling using try...catch

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

    try {

      // code that generate exception
      int divideByZero = 5 / 0;
      System.out.println("Rest of code in try block");
    }

    catch (ArithmeticException e) {
      System.out.println("ArithmeticException => " + e.getMessage());
    }
  }
}
```

Output

```
ArithmeticException => / by zero
```

In the example, we are trying to divide a number by 0. Here, this code generates an exception.

To handle the exception, we have put the code, 5 / 0 inside the try block. Now when an exception occurs, the rest of the code inside the try block is skipped.

The catch block catches the exception and statements inside the catch block is executed.

If none of the statements in the try block generates an exception, the catch block is skipped.

2. Java finally block

In Java, the finally block is always executed no matter whether there is an exception or not.

The finally block is optional. And, for each try block, there can be only one finally block.

The basic syntax of finally block is:

```
try {
  //code
}
catch (ExceptionType1 e1) {
  // catch block
}
finally {
  // finally block always executes
}
```

If an exception occurs, the finally block is executed after the try...catch block. Otherwise, it is executed after the try block. For each try block, there can be only one finally block.

Example: Java Exception Handling using finally block

```
class Main {
  public static void main(String[] args) {
    try {
      // code that generates exception
      int divideByZero = 5 / 0;
    }

    catch (ArithmeticException e) {
      System.out.println("ArithmeticException => " + e.getMessage());
    }

    finally {
      System.out.println("This is the finally block");
    }
  }
}
```

Output

```
ArithmeticException => / by zero
This is the finally block
```

In the above example, we are dividing a number by **0** inside the try block. Here, this code generates an ArithmeticException.

The exception is caught by the catch block. And, then the finally block is executed.

Note: It is a good practice to use the finally block. It is because it can include important cleanup codes like,

code that might be accidentally skipped by return, continue or break

closing a file or connection

3. Java throw and throws keyword

The Java throw keyword is used to explicitly throw a single exception.

When we throw an exception, the flow of the program moves from the try block to the catch block.

Example: Exception handling using Java throw

```
class Main {
  public static void divideByZero() {

    // throw an exception
    throw new ArithmeticException("Trying to divide by 0");
  }

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

Output

```
Exception in thread "main" java.lang.ArithmeticException: Trying to
divide by 0
        at Main.divideByZero(Main.java:5)
        at Main.main(Main.java:9)
```

In the above example, we are explicitly throwing the ArithmeticException using the throw keyword.

Similarly, the throws keyword is used to declare the type of exceptions that might occur within the method. It is used in the method declaration.

Example: Java throws keyword

```
import java.io.*;

class Main {
  // declareing the type of exception
  public static void findFile() throws IOException {

    // code that may generate IOException
    File newFile = new File("test.txt");
    FileInputStream stream = new FileInputStream(newFile);
  }

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

Output

java.io.FileNotFoundException: test.txt (The system cannot find the file specified)

When we run this program, if the file **test.txt** does not exist, FileInputStream throws a FileNotFoundException which extends the IOException class.

The findFile() method specifies that an IOException can be thrown. The main() method calls this method and handles the exception if it is thrown.

If a method does not handle exceptions, the type of exceptions that may occur within it must be specified in the throws clause.

Java try...catch

The try...catch block in Java is used to handle exceptions and prevents the abnormal termination of the program.

Here's the syntax of a try...catch block in Java.

```
try{
  // code
}
catch(exception) {
```

```
  // code
}
```

The try block includes the code that might generate an exception.

The catch block includes the code that is executed when there occurs an exception inside the try block.

Example: Java try...catch block

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

    try {
      int divideByZero = 5 / 0;
      System.out.println("Rest of code in try block");
    }

    catch (ArithmeticException e) {
      System.out.println("ArithmeticException => " + e.getMessage());
    }
  }
}
```

Output

```
ArithmeticException => / by zero
```

In the above example, notice the line,

```
int divideByZero = 5 / 0;
```

Here, we are trying to divide a number by **zero**. In this case, an exception occurs. Hence, we have enclosed this code inside the try block.

When the program encounters this code, ArithmeticException occurs. And, the exception is caught by the catch block and executes the code inside the catch block.

The catch block is only executed if there exists an exception inside the try block.

Note: In Java, we can use a try block without a catch block. However, we cannot use a catch block without a try block.

Java try...finally block

We can also use the try block along with a finally block.

In this case, the finally block is always executed whether there is an exception inside the try block or not.

Example: Java try...finally block

```
class Main {
  public static void main(String[] args) {
    try {
      int divideByZero = 5 / 0;
    }

    finally {
      System.out.println("Finally block is always executed");
    }
  }
}
```

Output

```
Finally block is always executed
Exception in thread "main" java.lang.ArithmeticException: / by zero
        at Main.main(Main.java:4)
```

In the above example, we have used the try block along with the finally block. We can see that the code inside the try block is causing an exception.

However, the code inside the finally block is executed irrespective of the exception.

Java try...catch...finally block

In Java, we can also use the finally block after the try...catch block. For example,

```
import java.io.*;

class ListOfNumbers {

  // create an integer array
  private int[] list = {5, 6, 8, 9, 2};

  // method to write data from array to a fila
  public void writeList() {
```

```
    PrintWriter out = null;

    try {
      System.out.println("Entering try statement");

      // creating a new file OutputFile.txt
      out = new PrintWriter(new FileWriter("OutputFile.txt"));

      // writing values from list array to Output.txt
      for (int i = 0; i < 7; i++) {
        out.println("Value at: " + i + " = " + list[i]);
      }
    }

    catch (Exception e) {
      System.out.println("Exception => " + e.getMessage());
    }

    finally {
      // checking if PrintWriter has been opened
      if (out != null) {
        System.out.println("Closing PrintWriter");
        // close PrintWriter
        out.close();
      }

      else {
        System.out.println("PrintWriter not open");
      }
    }

  }
}

class Main {
  public static void main(String[] args) {
    ListOfNumbers list = new ListOfNumbers();
    list.writeList();
  }
}
```

Output

```
Entering try statement
Exception => Index 5 out of bounds for length 5
Closing PrintWriter
```

In the above example, we have created an array named *list* and a file named *output.txt*. Here, we are trying to read data from the array and storing to the file.

Notice the code,

```
for (int i = 0; i < 7; i++) {
  out.println("Value at: " + i + " = " + list[i]);
}
```

Here, the size of the array is 5 and the last element of the array is at list[4]. However, we are trying to access elements at *a[5]* and a[6].

Hence, the code generates an exception that is caught by the catch block.

Since the finally block is always executed, we have included code to close the PrintWriter inside the finally block.

It is a good practice to use finally block to include important cleanup code like closing a file or connection.

Note: There are some cases when a finally block does not execute:

- Use of System.exit() method
- An exception occurs in the finally block
- The death of a thread

Multiple Catch blocks

For each try block, there can be zero or more catch blocks. Multiple catch blocks allow us to handle each exception differently.

The argument type of each catch block indicates the type of exception that can be handled by it. For example,

```
class ListOfNumbers {
  public int[] arr = new int[10];

  public void writeList() {

    try {
      arr[10] = 11;
    }

    catch (NumberFormatException e1) {
      System.out.println("NumberFormatException => " +
e1.getMessage());
    }
```

```
    catch (IndexOutOfBoundsException e2) {
      System.out.println("IndexOutOfBoundsException => " +
e2.getMessage());
    }

  }
}

class Main {
  public static void main(String[] args) {
    ListOfNumbers list = new ListOfNumbers();
    list.writeList();
  }
}
```

Output

```
IndexOutOfBoundsException => Index 10 out of bounds for length 10
```

In this example, we have created an integer array named arr of size **10**.

Since the array index starts from **0**, the last element of the array is at arr[9]. Notice the statement,

arr[10] = 11;

Here, we are trying to assign a value to the index **10**. Hence, IndexOutOfBoundException occurs.

When an exception occurs in the try block,

- The exception is thrown to the first catch block. The first catch block does not handle an IndexOutOfBoundsException, so it is passed to the next catch block.
- The second catch block in the above example is the appropriate exception handler because it handles an IndexOutOfBoundsException. Hence, it is executed.

Catching Multiple Exceptions

From Java SE 7 and later, we can now catch more than one type of exception with one `catch` block.

This reduces code duplication and increases code simplicity and efficiency.

Each exception type that can be handled by the `catch` block is separated using a vertical bar `|`.

Its syntax is:

```
try {
  // code
} catch (ExceptionType1 | Exceptiontype2 ex) {
  // catch block
}
```

Java try-with-resources statement

The **try-with-resources** statement is a try statement that has one or more resource declarations.

Its syntax is:

```
try (resource declaration) {
  // use of the resource
} catch (ExceptionType e1) {
  // catch block
}
```

The resource is an object to be closed at the end of the program. It must be declared and initialized in the try statement.

Let's take an example.

```
try (PrintWriter out = new PrintWriter(new
FileWriter("OutputFile.txt")) {
  // use of the resource
}
```

The **try-with-resources** statement is also referred to as **automatic resource management**. This statement automatically closes all the resources at the end of the statement.

Java throw and throws

In Java, exceptions can be categorized into two types:

- **Unchecked Exceptions**: They are not checked at compile-time but at run-time.For example: ArithmeticException, NullPointerException, ArrayIndexOutOfBoundsException, exceptions under Error class, etc.
- **Checked Exceptions**: They are checked at compile-time. For example, IOException, InterruptedException, etc.

Usually, we don't need to handle unchecked exceptions. It's because unchecked exceptions occur due to programming errors. And, it is a good practice to correct them instead of handling them.

This tutorial will now focus on how to handle checked exceptions using throw and throws.

Java throws keyword

We use the throws keyword in the method declaration to declare the type of exceptions that might occur within it.

Its syntax is:

```
accessModifier returnType methodName() throws ExceptionType1,
ExceptionType2 … {
  // code
}
```

As you can see from the above syntax, we can use `throws` to declare multiple exceptions.

Example 1: Java throws Keyword

```
import java.io.*;
class Main {
  public static void findFile() throws IOException {
    // code that may produce IOException
    File newFile=new File("test.txt");
    FileInputStream stream=new FileInputStream(newFile);
  }

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

```
      findFile();
    } catch(IOException e){
      System.out.println(e);
    }
  }
}
```

Output

```
java.io.FileNotFoundException: test.txt (No such file or directory)
```

When we run this program, if the file test.txt does not exist, FileInputStream throws a FileNotFoundException which extends the IOException class.

If a method does not handle exceptions, the type of exceptions that may occur within it must be specified in the throws clause so that methods further up in the call stack can handle them or specify them using throws keyword themselves.

The findFile() method specifies that an IOException can be thrown. The main() method calls this method and handles the exception if it is thrown.

Throwing multiple exceptions

Here's how we can throw multiple exceptions using the throws keyword.

```
import java.io.*;
class Main {
  public static void findFile() throws NullPointerException,
IOException, InvalidClassException {

    // code that may produce NullPointerException
    … … …

    // code that may produce IOException
    … … …

    // code that may produce InvalidClassException
    … … …
  }

  public static void main(String[] args) {
    try{
      findFile();
    } catch(IOException e1){
      System.out.println(e1.getMessage());
    } catch(InvalidClassException e2){
      System.out.println(e2.getMessage());
```

```
    }
  }
}
```

Here, the findFile() method specifies that it can throw NullPointerException, IOException, and InvalidClassException in its throws clause.

Note that we have not handled the NullPointerException. This is because it is an unchecked exception. It is not necessary to specify it in the throws clause and handle it.

throws keyword Vs. try...catch...finally

There might be several methods that can cause exceptions. Writing try...catch for each method will be tedious and code becomes long and less-readable.

throws is also useful when you have checked exception (an exception that must be handled) that you don't want to catch in your current method.

Java throw keyword

The throw keyword is used to explicitly throw a single exception.

When an exception is thrown, the flow of program execution transfers from the try block to the catch block. We use the throw keyword within a method.

Its syntax is:

```
throw throwableObject;
```

A throwable object is an instance of class Throwable or subclass of the Throwable class.

Example 2: Java throw keyword

```
class Main {
  public static void divideByZero() {
    throw new ArithmeticException("Trying to divide by 0");
  }

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

Output

```
Exception in thread "main" java.lang.ArithmeticException: Trying to
divide by 0
    at Main.divideByZero(Main.java:3)
    at Main.main(Main.java:7)
exit status 1
```

In this example, we are explicitly throwing an ArithmeticException.

Note: ArithmeticException is an unchecked exception. It's usually not necessary to handle unchecked exceptions.

Example 3: Throwing checked exception

```
import java.io.*;
class Main {
  public static void findFile() throws IOException {
    throw new IOException("File not found");
  }

  public static void main(String[] args) {
    try {
      findFile();
      System.out.println("Rest of code in try block");
    } catch (IOException e) {
      System.out.println(e.getMessage());
    }
  }
}
```

Output

```
File not found
```

The findFile() method throws an IOException with the message we passed to its constructor.

Note that since it is a checked exception, we must specify it in the throws clause.

The methods that call this findFile() method need to either handle this exception or specify it using throws keyword themselves.

We have handled this exception in the main() method. The flow of program execution transfers from the try block to catch block when an exception is thrown. So, the rest of the code in the try block is skipped and statements in the catch block are executed.

Java catch Multiple Exceptions

Before Java 7, we had to write multiple exception handling codes for different types of exceptions even if there was code redundancy.

Let’s take an example.

Example 1: Multiple catch blocks

```
class Main {
  public static void main(String[] args) {
    try {
      int array[] = new int[10];
      array[10] = 30 / 0;
    } catch (ArithmeticException e) {
      System.out.println(e.getMessage());
    } catch (ArrayIndexOutOfBoundsException e) {
      System.out.println(e.getMessage());
    }
  }
}
```

Output

```
/ by zero
```

In this example, two exceptions may occur:

- ArithmeticException because we are trying to divide a number by 0.
- ArrayIndexOutOfBoundsException because we have declared a new integer array with array bounds 0 to 9 and we are trying to assign a value to index 10.

We are printing out the exception message in both the catch blocks i.e. duplicate code.

The associativity of the assignment operator = is right to left, so an ArithmeticException is thrown first with the message / by zero.

Handle Multiple Exceptions in a catch Block

In Java SE 7 and later, we can now catch more than one type of exception in a single `catch` block.

Each exception type that can be handled by the `catch` block is separated using a vertical bar or pipe `|`.

Its syntax is:

```
try {
  // code
} catch (ExceptionType1 | Exceptiontype2 ex) {
  // catch block
}
```

Example 2: Multi-catch block

```
class Main {
  public static void main(String[] args) {
    try {
      int array[] = new int[10];
      array[10] = 30 / 0;
    } catch (ArithmeticException | ArrayIndexOutOfBoundsException e) {
      System.out.println(e.getMessage());
    }
  }
}
```

Output

```
/ by zero
```

Catching multiple exceptions in a single catch block reduces code duplication and increases efficiency.

The bytecode generated while compiling this program will be smaller than the program having multiple catch blocks as there is no code redundancy.

Note: If a catch block handles multiple exceptions, the catch parameter is implicitly final. This means we cannot assign any values to catch parameters.

Catching base Exception

When catching multiple exceptions in a single `catch` block, the rule is generalized to specialized.

This means that if there is a hierarchy of exceptions in the `catch` block, we can catch the base exception only instead of catching multiple specialized exceptions.

Let’s take an example.

Example 3: Catching base exception class only

```
class Main {
  public static void main(String[] args) {
    try {
      int array[] = new int[10];
      array[10] = 30 / 0;
    } catch (Exception e) {
      System.out.println(e.getMessage());
    }
  }
}
```

Output

```
/ by zero
```

We know that all the exception classes are subclasses of the Exception class. So, instead of catching multiple specialized exceptions, we can simply catch the Exception class.

If the base exception class has already been specified in the catch block, do not use child exception classes in the same catch block. Otherwise, we will get a compilation error.

Let’s take an example.

Example 4: Catching base and child exception classes

```
class Main {
  public static void main(String[] args) {
    try {
      int array[] = new int[10];
      array[10] = 30 / 0;
    } catch (Exception | ArithmeticException |
ArrayIndexOutOfBoundsException e) {
```

```
      System.out.println(e.getMessage());
    }
  }
}
```

Output

```
Main.java:6: error: Alternatives in a multi-catch statement cannot be
related by subclassing
```

In this example, ArithmeticException and ArrayIndexOutOfBoundsException are both subclasses of the Exception class. So, we get a compilation error.

Java try-with-resources

The try-with-resources statement automatically closes all the resources at the end of the statement. A resource is an object to be closed at the end of the program.

Its syntax is:

```
try (resource declaration) {
  // use of the resource
} catch (ExceptionType e1) {
  // catch block
}
```

As seen from the above syntax, we declare the try-with-resources statement by,

1. declaring and instantiating the resource within the try clause.
2. specifying and handling all exceptions that might be thrown while closing the resource.

Note: The try-with-resources statement closes all the resources that implement the AutoCloseable interface.

Let us take an example that implements the try-with-resources statement.

Example 1: try-with-resources

```
import java.io.*;

class Main {
  public static void main(String[] args) {
    String line;
    try(BufferedReader br = new BufferedReader(new
FileReader("test.txt"))) {
```

```
      while ((line = br.readLine()) != null) {
        System.out.println("Line =>"+line);
      }
    } catch (IOException e) {
      System.out.println("IOException in try block =>" +
e.getMessage());
    }
  }
}
```

Output if the test.txt file is not found.

```
IOException in try-with-resources block =>test.txt (No such file or
directory)
```

Output if the test.txt file is found.

```
Entering try-with-resources block
Line =>test line
```

In this example, we use an instance of *BufferedReader* to read data from the test.txt file.

Declaring and instantiating the *BufferedReader* inside the try-with-resources statement ensures that its instance is closed regardless of whether the try statement completes normally or throws an exception.

If an exception occurs, it can be handled using the exception handling blocks or the throws keyword.

Suppressed Exceptions

In the above example, exceptions can be thrown from the try-with-resources statement when:

- The file test.txt is not found.
- Closing the BufferedReader object.

An exception can also be thrown from the try block as a file read can fail for many reasons at any time.

If exceptions are thrown from both the try block and the try-with-resources statement, exception from the try block is thrown and exception from the try-with-resources statement is suppressed.

Retrieving Suppressed Exceptions

In Java 7 and later, the suppressed exceptions can be retrieved by calling the Throwable.getSuppressed() method from the exception thrown by the try block.

This method returns an array of all suppressed exceptions. We get the suppressed exceptions in the catch block.

```
catch(IOException e) {
  System.out.println("Thrown exception=>" + e.getMessage());
  Throwable[] suppressedExceptions = e.getSuppressed();
  for (int i=0; i<suppressedExceptions.length; i++) {
    System.out.println("Suppressed exception=>" +
suppressedExceptions[i]);
  }
}
```

Advantages of using try-with-resources

Here are the advantages of using try-with-resources:

1. finally block not required to close the resource

Before Java 7 introduced this feature, we had to use the `finally` block to ensure that the resource is closed to avoid resource leaks.

Here's a program that is similar to **Example 1**. However, in this program, we have used finally block to close resources.

Example 2: Close resource using finally block

```
import java.io.*;

class Main {
  public static void main(String[] args) {
    BufferedReader br = null;
    String line;

    try {
      System.out.println("Entering try block");
      br = new BufferedReader(new FileReader("test.txt"));
```

```
      while ((line = br.readLine()) != null) {
        System.out.println("Line =>"+line);
      }
    } catch (IOException e) {
      System.out.println("IOException in try block =>" +
e.getMessage());
    } finally {
      System.out.println("Entering finally block");
      try {
        if (br != null) {
          br.close();
        }
      } catch (IOException e) {
        System.out.println("IOException in finally block
=>"+e.getMessage());
      }

    }
  }
}
```

Output

```
Entering try block
Line =>line from test.txt file
Entering finally block
```

As we can see from the above example, the use of finally block to clean up resources makes the code more complex.

Notice the try...catch block in the finally block as well? This is because an IOException can also occur while closing the BufferedReader instance inside this finally block so it is also caught and handled.

The try-with-resources statement does **automatic resource management**. We need not explicitly close the resources as JVM automatically closes them. This makes the code more readable and easier to write.

2. try-with-resources with multiple resources

We can declare more than one resource in the try-with-resources statement by separating them with a semicolon ;

Example 3: try with multiple resources

```
import java.io.*;
import java.util.*;
class Main {
  public static void main(String[] args) throws IOException{
    try (Scanner scanner = new Scanner(new File("testRead.txt"));
      PrintWriter writer = new PrintWriter(new File("testWrite.txt")))
{
      while (scanner.hasNext()) {
        writer.print(scanner.nextLine());
      }
    }
  }
}
```

If this program executes without generating any exceptions, Scanner object reads a line from the testRead.txt file and writes it in a new testWrite.txt file.

When multiple declarations are made, the try-with-resources statement closes these resources in reverse order. In this example, the PrintWriter object is closed first and then the Scanner object is closed.

Java 9 try-with-resources enhancement

In Java 7, there is a restriction to the try-with-resources statement. The resource needs to be declared locally within its block.

```
try (Scanner scanner = new Scanner(new File("testRead.txt"))) {
  // code
}
```

If we declared the resource outside the block in Java 7, it would have generated an error message.

```
Scanner scanner = new Scanner(new File("testRead.txt"));
try (scanner) {
  // code
}
```

To deal with this error, Java 9 improved the try-with-resources statement so that the reference of the resource can be used even if it is not declared locally. The above code will now execute without any compilation error.

Java Annotations

Java annotations are metadata (data about data) for our program source code.

They provide additional information about the program to the compiler but are not part of the program itself. These annotations do not affect the execution of the compiled program.

Annotations start with `@`. Its syntax is:

```
@AnnotationName
```

Let's take an example of `@Override` annotation.

The @Override annotation specifies that the method that has been marked with this annotation overrides the method of the superclass with the same method name, return type, and parameter list.

It is not mandatory to use @Override when overriding a method. However, if we use it, the compiler gives an error if something is wrong (such as wrong parameter type) while overriding the method.

Example 1: @Override Annotation Example

```
class Animal {
  public void displayInfo() {
    System.out.println("I am an animal.");
  }
}

class Dog extends Animal {
  @Override
  public void displayInfo() {
    System.out.println("I am a dog.");
  }
}

class Main {
  public static void main(String[] args) {
    Dog d1 = new Dog();
    d1.displayInfo();
  }
}
```

Output

```
I am a dog.
```

In this example, the method displayInfo() is present in both the superclass *Animal* and subclass *Dog*. When this method is called, the method of the subclass is called instead of the method in the superclass.

Annotation formats

Annotations may also include elements (members/attributes/parameters).

1. Marker Annotations

Marker annotations do not contain members/elements. It is only used for marking a declaration.

Its syntax is:

```
@AnnotationName()
```

Since these annotations do not contain elements, parentheses can be excluded. For example,

```
@Override
```

2. Single element Annotations

A single element annotation contains only one element.

Its syntax is:

```
@AnnotationName(elementName = "elementValue")
```

If there is only one element, it is a convention to name that element as *value*.

```
@AnnotationName(value = "elementValue")
```

In this case, the element name can be excluded as well. The element name will be *value* by default.

```
@AnnotationName("elementValue")
```

3. Multiple element Annotations

These annotations contain multiple elements separated by commas.

Its syntax is:

```
@AnnotationName(element1 = "value1", element2 = "value2")
```

Annotation placement

Any declaration can be marked with annotation by placing it above that declaration. As of Java 8, annotations can also be placed before a type.

1. Above declarations

As mentioned above, Java annotations can be placed above class, method, interface, field, and other program element declarations.

Example 2: @SuppressWarnings Annotation Example

```
import java.util.*;

class Main {
  @SuppressWarnings("unchecked")
  static void wordsList() {
    ArrayList wordList = new ArrayList<>();

// This causes an unchecked warning
    wordList.add("programiz");

    System.out.println("Word list => " + wordList);
  }

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

Output

```
Word list => [programiz]
```

If the above program is compiled without using the @SuppressWarnings("unchecked") annotation, the compiler will still compile the program but it will give warnings like:

```
Main.java uses unchecked or unsafe operations.
Word list => [programiz]
```

We are getting the warning

```
Main.java uses unchecked or unsafe operations
```

because of the following statement.

```
ArrayList wordList = new ArrayList<>();
```

This is because we haven't defined the generic type of the array list. We can fix this warning by specifying generics inside angle brackets <>.

```
ArrayList<String> wordList = new ArrayList<>();
```

2. Type annotations

Before Java 8, annotations could be applied to declarations only. Now, type annotations can be used as well. This means that we can place annotations wherever we use a type.

Constructor invocations

```
new @Readonly ArrayList<>()
```

Type definitions

```
@NonNull String str;
```

This declaration specifies non-null variable *str* of type `String` to avoid `NullPointerException`.

```
@NonNull List<String> newList;
```

This declaration specifies a non-null list of type `String`.

```
List<@NonNull String> newList;
```

This declaration specifies a list of non-null values of type `String`.

Type casts

```
newStr = (@NonNull String) str;
```

extends and implements clause

```
class Warning extends @Localized Message
```

throws clause

```
public String readMethod() throws @Localized IOException
```

Type annotations enable Java code to be analyzed better and provide even stronger type checks.

Types of Annotations

1. **Predefined annotations**

1. `@Deprecated`
2. `@Override`
3. `@SuppressWarnings`
4. `@SafeVarargs`
5. `@FunctionalInterface`

2. **Meta-annotations**

1. `@Retention`
2. `@Documented`
3. `@Target`
4. `@Inherited`
5. `@Repeatable`

3. **Custom annotations**

These annotation types are described in detail in the Java Annotation Types tutorial.

Use of Annotations

- **Compiler instructions** - Annotations can be used for giving instructions to the compiler, detect errors or suppress warnings. The built-in annotations @Deprecated, @Override, @SuppressWarnings are used for these purposes.

- **Compile-time instructions** - Compile-time instructions provided by these annotations help the software build tools to generate code, XML files and many more.
- **Runtime instructions** - Some annotations can be defined to give instructions to the program at runtime. These annotations are accessed using Java Reflection.

Java Annotation Types

Java annotations are metadata (data about data) for our program source code. There are several predefined annotations provided by the Java SE. Moreover, we can also create custom annotations as per our needs.

These annotations can be categorized as:

1. **Predefined annotations**

- `@Deprecated`
- `@Override`
- `@SuppressWarnings`
- `@SafeVarargs`
- `@FunctionalInterface`

2. **Custom annotations**

3. **Meta-annotations**

- `@Retention`
- `@Documented`
- `@Target`
- `@Inherited`
- `@Repeatable`

Predefined Annotation Types

1. @Deprecated

The @Deprecated annotation is a marker annotation that indicates the element (class, method, field, etc) is deprecated and has been replaced by a newer element.

Its syntax is:

```
@Deprecated
accessModifier returnType deprecatedMethodName() { ... }
```

When a program uses the element that has been declared deprecated, the compiler generates a warning.

We use Javadoc @deprecated tag for documenting the deprecated element.

```
/**
 * @deprecated
 * why it was deprecated
 */
@Deprecated
accessModifier returnType deprecatedMethodName() { ... }
```

Example 1: @Deprecated annotation example

```
class Main {
  /**
   * @deprecated
   * This method is deprecated and has been replaced by newMethod()
   */
  @Deprecated
  public static void deprecatedMethod() {
    System.out.println("Deprecated method");
  }

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

Output

```
Deprecated method
```

2. @Override

The @Override annotation specifies that a method of a subclass overrides the method of the superclass with the same method name, return type, and parameter list.

It is not mandatory to use @Override when overriding a method. However, if we use it, the compiler gives an error if something is wrong (such as wrong parameter type) while overriding the method.

Example 2: @Override annotation example

```
class Animal {

  // overridden method
  public void display(){
    System.out.println("I am an animal");
  }
}

class Dog extends Animal {

  // overriding method
  @Override
  public void display(){
    System.out.println("I am a dog");
  }

  public void printMessage(){
    display();
  }
}

class Main {
  public static void main(String[] args) {
    Dog dog1 = new Dog();
    dog1.printMessage();
  }
}
```

Output

```
I am a dog
```

In this example, by making an object *dog1* of *Dog* class, we can call its method *printMessage()* which then executes the display() statement.

Since display() is defined in both the classes, the method of subclass *Dog* overrides the method of superclass *Animal*. Hence, the display() of the subclass is called.

3. @SuppressWarnings

As the name suggests, the @SuppressWarnings annotation instructs the compiler to suppress warnings that are generated while the program executes.

We can specify the type of warnings to be suppressed. The warnings that can be suppressed are compiler-specific but there are two categories of warnings: **deprecation** and **unchecked**.

To suppress a particular category of warning, we use:

```
@SuppressWarnings("warningCategory")
```

For example,

```
@SuppressWarnings("deprecated")
```

To suppress multiple categories of warnings, we use:

```
@SuppressWarnings({"warningCategory1", "warningCategory2"})
```

For example,

```
@SuppressWarnings({"deprecated", "unchecked"})
```

Category `deprecated` instructs the compiler to suppress warnings when we use a deprecated element.

Category `unchecked` instructs the compiler to suppress warnings when we use raw types.

And, undefined warnings are ignored. For example,

```
@SuppressWarnings("someundefinedwarning")
```

Example 3: @SuppressWarnings annotation example

```
class Main {
  @Deprecated
  public static void deprecatedMethod() {
    System.out.println("Deprecated method");
  }

  @SuppressWarnings("deprecated")
  public static void main(String args[]) {
    Main depObj = new Main();
    depObj. deprecatedMethod();
  }
}
```

Output

```
Deprecated method
```

Here, deprecatedMethod() has been marked as deprecated and will give compiler warnings when used. By using the @SuppressWarnings("deprecated") annotation, we can avoid compiler warnings.

4. @SafeVarargs

The @SafeVarargs annotation asserts that the annotated method or constructor does not perform unsafe operations on its varargs (variable number of arguments).

We can only use this annotation on methods or constructors that cannot be overridden. This is because the methods that override them might perform unsafe operations.

Before Java 9, we could use this annotation only on final or static methods because they cannot be overridden. We can now use this annotation for private methods as well.

Example 4: @SafeVarargs annotation example

```
import java.util.*;

class Main {

  private void displayList(List<String>... lists) {
    for (List<String> list : lists) {
      System.out.println(list);
    }
  }

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

    List<String> universityList = Arrays.asList("Tribhuvan
University", "Kathmandu University");
    obj.displayList(universityList);

    List<String> programmingLanguages = Arrays.asList("Java", "C");
    obj.displayList(universityList, programmingLanguages);
  }
}
```

Warnings

```
Type safety: Potential heap pollution via varargs parameter lists
Type safety: A generic array of List<String> is created for a varargs
 parameter
```

Output

```
Note: Main.java uses unchecked or unsafe operations.
[Tribhuvan University, Kathmandu University]
[Tribhuvan University, Kathmandu University]
[Java, C]
```

Here, List ... lists specifies a variable-length argument of type List. This means that the method displayList() can have zero or more arguments.

The above program compiles without errors but gives warnings when @SafeVarargs annotation isn't used.

When we use @SafeVarargs annotation in the above example,

```
@SafeVarargs
 private void displayList(List<String>... lists) { ... }
```

We get the same output but without any warnings. Unchecked warnings are also suppressed when we use this annotation.

5. @FunctionalInterface

Java 8 first introduced this `@FunctionalInterface` annotation. This annotation indicates that the type declaration on which it is used is a functional interface. A functional interface can have only one abstract method.

Example 5: @FunctionalInterface annotation example

```
@FunctionalInterface
public interface MyFuncInterface{
  public void firstMethod(); // this is an abstract method
}
```

If we add another abstract method, let's say

```
@FunctionalInterface
public interface MyFuncInterface{
  public void firstMethod(); // this is an abstract method
  public void secondMethod(); // this throws compile error
```

```
}
```

Now, when we run the program, we will get the following warning:

```
Unexpected @FunctionalInterface annotation
@FunctionalInterface ^ MyFuncInterface is not a functional interface
multiple non-overriding abstract methods found in interface
MyFuncInterface
```

It is not mandatory to use @FunctionalInterface annotation. The compiler will consider any interface that meets the functional interface definition as a functional interface.

We use this annotation to make sure that the functional interface has only one abstract method.

However, it can have any number of default and static methods because they have an implementation.

```
@FunctionalInterface
public interface MyFuncInterface{
  public void firstMethod(); // this is an abstract method
  default void secondMethod() { ... }
  default void thirdMethod() { ... }
}
```

Custom Annotations

It is also possible to create our own custom annotations.

Its syntax is:

```
[Access Specifier] @interface<AnnotationName> {
  DataType <Method Name>() [default value];
}
```

Here is what you need to know about custom annotation:

- Annotations can be created by using @interface followed by the annotation name.
- The annotation can have elements that look like methods but they do not have an implementation.
- The default value is optional. The parameters cannot have a null value.

- The return type of the method can be primitive, enum, string, class name or array of these types.

Example 6: Custom annotation example

```
@interface MyCustomAnnotation {
  String value() default "default value";
}

class Main {
  @MyCustomAnnotation(value = "programiz")
  public void method1() {
    System.out.println("Test method 1");
  }

  public static void main(String[] args) throws Exception {
    Main obj = new Main();
    obj.method1();
  }
}
```

Output

```
Test method 1
```

Meta Annotations

Meta-annotations are annotations that are applied to other annotations.

1. @Retention

The @Retention annotation specifies the level up to which the annotation will be available.

Its syntax is:

```
@Retention(RetentionPolicy)
```

There are 3 types of retention policies:

- **RetentionPolicy.SOURCE** - The annotation is available only at the source level and is ignored by the compiler.
- **RetentionPolicy.CLASS** - The annotation is available to the compiler at compile-time, but is ignored by the Java Virtual Machine (JVM).

- **RetentionPolicy.RUNTIME** - The annotation is available to the JVM.

For example,

```
@Retention(RetentionPolicy.RUNTIME)
public @interface MyCustomAnnotation{ ... }
```

2. @Documented

By default, custom annotations are not included in the official Java documentation. To include our annotation in the Javadoc documentation, we use the @Documented annotation.

For example,

```
@Documented
public @interface MyCustomAnnotation{ ... }
```

3. @Target

We can restrict an annotation to be applied to specific targets using the `@Target` annotation.

Its syntax is:

```
@Target(ElementType)
```

The ElementType can have one of the following types:

Element Type	Target
`ElementType.ANNOTATION_TYPE`	Annotation type
`ElementType.CONSTRUCTOR`	Constructors
`ElementType.FIELD`	Fields
`ElementType.LOCAL_VARIABLE`	Local variables
`ElementType.METHOD`	Methods
`ElementType.PACKAGE`	Package

`ElementType.PARAMETER`	Parameter
`ElementType.TYPE`	Any element of class

For example,

```
@Target(ElementType.METHOD)
public @interface MyCustomAnnotation{ ... }
```

In this example, we have restricted the use of this annotation to methods only.

Note: If the target type is not defined, the annotation can be used for any element.

4. @Inherited

By default, an annotation type cannot be inherited from a superclass. However, if we need to inherit an annotation from a superclass to a subclass, we use the @Inherited annotation.

Its syntax is:

```
@Inherited
```

For example,

```
@Inherited
public @interface MyCustomAnnotation { ... }

@MyCustomAnnotation
public class ParentClass{ ... }

public class ChildClass extends ParentClass { ... }
```

5. @Repeatable

An annotation that has been marked by `@Repeatable` can be applied multiple times to the same declaration.

```
@Repeatable(Universities.class)
public @interface University {
  String name();
}
```

The value defined in the @Repeatable annotation is the container annotation. The container annotation has a variable *value* of array type of the above repeatable annotation. Here, Universities are the containing annotation type.

```
public @interface Universities {
  University[] value();
}
```

Now, the @University annotation can be used multiple times on the same declaration.

```
@University(name = "TU")
@University(name = "KU")
private String uniName;
```

If we need to retrieve the annotation data, we can use the Reflection API.

To retrieve annotation values, we use getAnnotationsByType() or getAnnotations() method defined in the Reflection API.

Java Logging

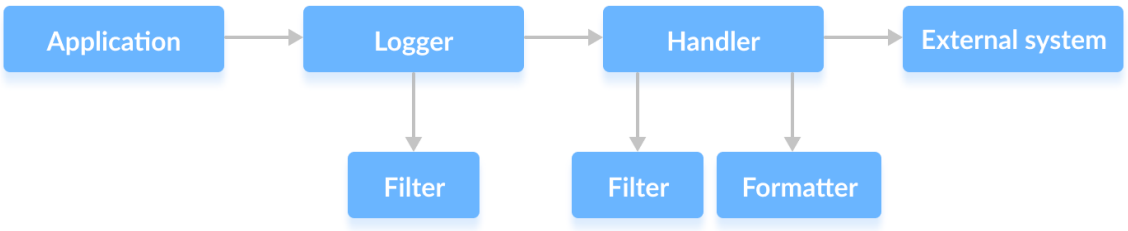
Java allows us to create and capture log messages and files through the process of logging.

In Java, logging requires frameworks and APIs. Java has a built-in logging framework in the java.util.logging package.

We can also use third-party frameworks like Log4j, Logback, and many more for logging purposes.

Java Logging Components

The figure below represents the core components and the flow of control of the Java Logging API (java.util.logging).



Java Logging

1. Logger

The Logger class provides methods for logging. We can instantiate objects from the Logger class and call its methods for logging purposes.

Let's take an example.

```
Logger logger = Logger.getLogger("newLoggerName");
```

The getLogger() method of the Logger class is used to find or create a new Logger. The string argument defines the name of the logger.

Here, this creates a new Logger object or returns an existing Logger with the same name.

It is a convention to define a Logger after the current class using class.getName().

```
Logger logger = Logger.getLogger(MyClass.class.getName());
```

Note: This method will throw NullPointerException if the passed name is null.

Each Logger has a level that determines the importance of the log message. There are 7 basic log levels:

Log Level (in descending order)	Use
SEVERE	serious failure
WARNING	warning message, a potential problem
INFO	general runtime information
CONFIG	configuration information

FINE	general developer information (tracing messages)
FINER	detailed developer information (tracing messages)
FINEST	highly detailed developer information(tracing messages)
OFF	turn off logging for all levels (capture nothing)
ALL	turn on logging for all levels (capture everything)

Each log level has an integer value that determines their severity except for two special log levels OFF and ALL.

Logging the message

By default, the top three log levels are always logged. To set a different level, we can use the following code:

```
logger.setLevel(Level.LogLevel);

// example
logger.setLevel(Level.FINE);
```

In this example, only level FINE and levels above it are set to be logged. All other log messages are dropped.

Now to log a message, we use the log() method.

```
logger.log(Level.LogLevel, "log message");

// example
logger.log(Level.INFO, "This is INFO log level message");
```

There are shorthand methods for logging at desired levels.

```
logger.info( "This is INFO log level message");
logger.warning( "This is WARNING log level message");
```

All log requests that have passed the set log level are then forwarded to the **LogRecord**.

Note: If a logger's level is set to null, its level is inherited from its parent and so on up the tree.

2. Filters

A filter (if it is present) determines whether the **LogRecord** should be forwarded or not. As the name suggests, it filters the log messages according to specific criteria.

A **LogRecord** is only passed from the logger to the log handler and from the log handler to external systems if it passes the specified criteria.

```
// set a filter
logger.setFilter(filter);

// get a filter
Filter filter = logger.getFilter();
```

3. Handlers(Appenders)

The log handler or the appenders receive the **LogRecord** and exports it to various targets.

Java SE provides 5 built-in handlers:

Handlers	Use
`StreamHandler`	writes to an `OutputStream`
`ConsoleHandler`	writes to console
`FileHandler`	writes to file
`SocketHandler`	writes to remote TCP ports
`MemoryHandler`	writes to memory

A handler can pass the **LogRecord** to a filter to again determine whether it can be forwarded to external systems or not.

To add a new handler, we use the following code:

```
logger.addHandler(handler);

// example
```

```
Handler handler = new ConsoleHandler();
logger.addHandler(handler);
```

To remove a handler, we use the following code:

```
logger.removeHandler(handler);

// example
Handler handler = new ConsoleHandler();
logger.addHandler(handler);
logger.removeHandler(handler);
```

A logger can have multiple handlers. To get all the handlers, we use the following code:

```
Handler[] handlers = logger.getHandlers();
```

4. Formatters

A handler can also use a **Formatter** to format the **LogRecord** object into a string before exporting it to external systems.

Java SE has two built-in **Formatters**:

Formatters	Use
`SimpleFormatter`	formats **LogRecord** to string
`XMLFormatter`	formats **LogRecord** to XML form

We can use the following code to format a handler:

```
// formats to string form
handler.setFormatter(new SimpleFormatter());

// formats to XML form
handler.setFormatter(new XMLFormatter());
```

LogManager

The **LogManager** object keeps track of the global logging information. It reads and maintains the logging configuration and the logger instances.

The log manager is a singleton, which means that only one instance of it is instantiated.

To obtain the log manager instance, we use the following code:

```
LogManager manager = new LogManager();
```

Advantages of Logging

Here are some of the advantages of logging in Java.

- helps in monitoring the flow of the program
- helps in capturing any errors that may occur
- provides support for problem diagnosis and debugging

Java Assertions

Assertions in Java help to detect bugs by testing code we assume to be true.

An assertion is made using the assert keyword.

Its syntax is:

```
assert condition;
```

Here, condition is a boolean expression that we assume to be true when the program executes.

Enabling Assertions

By default, assertions are disabled and ignored at runtime.

To enable assertions, we use:

```
java -ea:arguments
```

OR

```
java -enableassertions:arguments
```

When assertions are enabled and the condition is true, the program executes normally.

But if the condition evaluates to false while assertions are enabled, JVM throws an AssertionError, and the program stops immediately.

Example 1: Java assertion

```
class Main {
  public static void main(String args[]) {
    String[] weekends = {"Friday", "Saturday", "Sunday"};
    assert weekends.length == 2;
    System.out.println("There are " + weekends.length + "  weekends in
a week");
  }
}
```

Output

```
There are 3 weekends in a week
```

We get the above output because this program has no compilation errors and by default, assertions are disabled.

After enabling assertions, we get the following output:

```
Exception in thread "main" java.lang.AssertionError
```

Another form of assertion statement

```
assert condition : expression;
```

In this form of assertion statement, an expression is passed to the constructor of the AssertionError object. This expression has a value that is displayed as the error's detail message if the condition is false.

The detailed message is used to capture and transmit the information of the assertion failure to help in debugging the problem.

Example 2: Java assertion with expression example

```
class Main {
  public static void main(String args[]) {
    String[] weekends = {"Friday", "Saturday", "Sunday"};
    assert weekends.length==2 : "There are only 2 weekends in a week";
    System.out.println("There are " + weekends.length + "  weekends in
a week");
  }
}
```

Output

```
Exception in thread "main" java.lang.AssertionError: There are only 2
weekends in a week
```

As we see from the above example, the expression is passed to the constructor of the AssertionError object. If our assumption is false and assertions are enabled, an exception is thrown with an appropriate message.

This message helps in diagnosing and fixing the error that caused the assertion to fail.

Enabling assertion for specific classes and packages

If we do not provide any arguments to the assertion command-line switches,

```
java -ea
```

This enables assertions in all classes except system classes.

We can also enable assertion for specific classes and packages using arguments. The arguments that can be provided to these command-line switches are:

Enable assertion in class names

To enable assertion for all classes of our program Main,

```
java -ea Main
```

To enable only one class,

```
java -ea:AnimalClass Main
```

This enables assertion in only the `AnimalClass` in the `Main` program.

Enable assertion in package names

To enable assertions for package `com.animal` and its sub-packages only,

```
java -ea:com.animal... Main
```

Enable assertion in unnamed packages

To enable assertion in unnamed packages (when we don't use a package statement) in the current working directory.

```
java -ea:... Main
```

Enable assertion in system classes

To enable assertion in system classes, we use a different command-line switch:

```
java -esa:arguments
```

OR

```
java -enablesystemassertions:arguments
```

The arguments that can be provided to these switches are the same.

Disabling Assertions

To disable assertions, we use:

```
java -da arguments
```

OR

```
java -disableassertions arguments
```

To disable assertion in system classes, we use:

```
java -dsa:arguments
```

OR

```
java -disablesystemassertions:arguments
```

The arguments that can be passed while disabling assertions are the same as while enabling them.

Advantages of Assertion

1. Quick and efficient for detecting and correcting bugs.
2. Assertion checks are done only during development and testing. They are automatically removed in the production code at runtime so that it won't slow the execution of the program.
3. It helps remove boilerplate code and make code more readable.
4. Refactors and optimizes code with increased confidence that it functions correctly.

When to use Assertions

1. Unreachable codes

Unreachable codes are codes that do not execute when we try to run the program. Use assertions to make sure unreachable codes are actually unreachable.

Let's take an example.

```
void unreachableCodeMethod() {
  System.out.println("Reachable code");
  return;
  // Unreachable code
  System.out.println("Unreachable code");
  assert false;
}
```

Let's take another example of a switch statement without a default case.

```
switch (dayOfWeek) {
  case "Sunday":
    System.out.println("It's Sunday!");
    break;
  case "Monday":
    System.out.println("It's Monday!");
    break;
  case "Tuesday":
    System.out.println("It's Tuesday!");
    break;
  case "Wednesday":
```

```
    System.out.println("It's Wednesday!");
    break;
  case "Thursday":
    System.out.println("It's Thursday!");
    break;
  case "Friday":
    System.out.println("It's Friday!");
    break;
  case "Saturday":
    System.out.println("It's Saturday!");
    break;
}
```

The above switch statement indicates that the days of the week can be only one of the above 7 values. Having no default case means that the programmer believes that one of these cases will always be executed.

However, there might be some cases that have not yet been considered where the assumption is actually false.

This assumption should be checked using an assertion to make sure that the default switch case is not reached.

```
default:
    assert false: dayofWeek + " is invalid day";
```

If *dayOfWeek* has a value other than the valid days, an AssertionError is thrown.

2. Documenting assumptions

To document their underlying assumptions, many programmers use comments. Let's take an example.

```
if (i % 2 == 0) {
    ...
} else { // We know (i % 2 == 1)
    ...
}
```

Use assertions instead.

Comments can get out-of-date and out-of-sync as the program grows. However, we will be forced to update the assert statements; otherwise, they might fail for valid conditions too.

```
if (i % 2 == 0) {
   ...
} else {
    assert i % 2 == 1 : i;
    ...
}
```

When not to use Assertions

1. Argument checking in public methods

Arguments in public methods may be provided by the user.

So, if an assertion is used to check these arguments, the conditions may fail and result in AssertionError.

Instead of using assertions, let it result in the appropriate runtime exceptions and handle these exceptions.

2. To evaluate expressions that affect the program operation

Do not call methods or evaluate exceptions that can later affect the program operation in assertion conditions.

Let us take an example of a list *weekdays* which contains the names of all the days in a week.

```
ArrayList<String> weekdays = new ArrayList<>(Arrays.asList("Sunday",
"Monday", "Tuesday", "Wednesday", "Thursday", "Friday", "Saturday" ));

ArrayList<String> weekends= new ArrayList<>(Arrays.asList("Sunday",
"Saturday" ));

assert weekdays.removeAll(weekends);
```

Here, we are trying to remove elements Saturday and Sunday from the ArrayList *weekdays*.

If the assertion is enabled, the program works fine. However, if assertions are disabled, the elements from the list are not removed. This may result in program failure.

Instead, assign the result to a variable and then use that variable for assertion.

```
ArrayList<String> weekdays = new ArrayList<>(Arrays.asList("Sunday",
"Monday", "Tuesday", "Wednesday", "Thursday", "Friday", "Saturday" ));

ArrayList<String> weekends= new ArrayList<>(Arrays.asList("Sunday",
"Saturday" ));

boolean weekendsRemoved = weekdays.removeAll(weekends);
assert weekendsRemoved;
```

In this way, we can ensure that all the *weekends* are removed from the *weekdays* regardless of the assertion being enabled or disabled. As a result, it does not affect the program operation in the future.