

# Java Basic for Tester

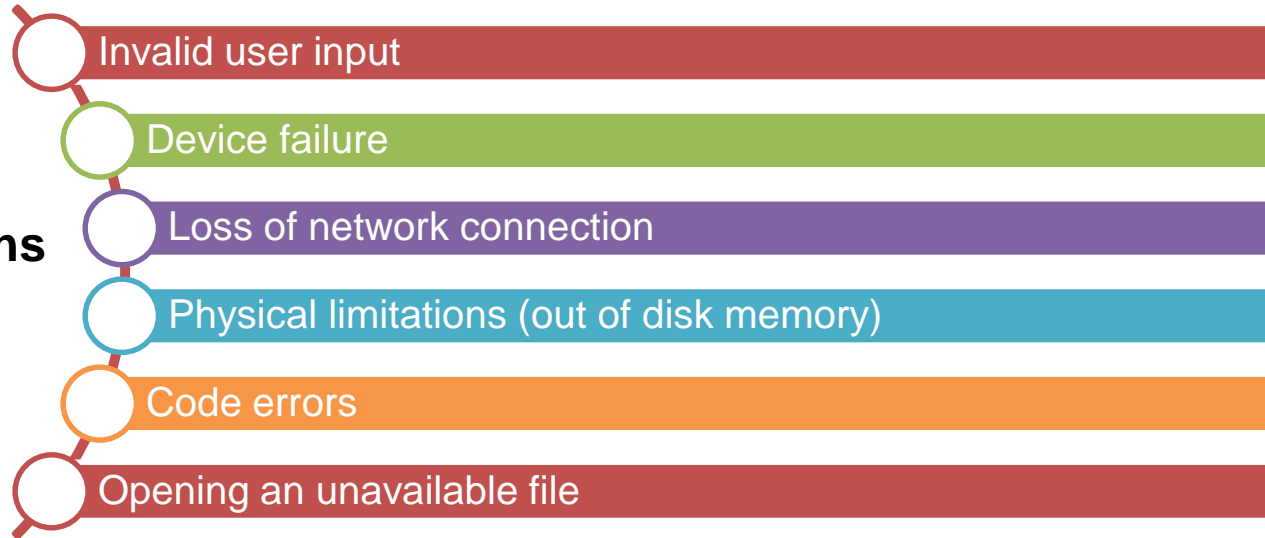
## *Java Exceptions*



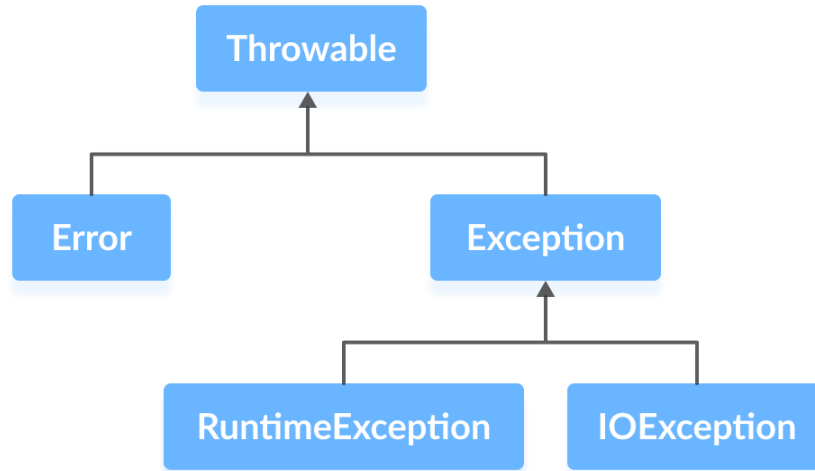
- *Java Exceptions*
- *Java Exception Handling*
- *Java throw and throws*
- *Java catch Multiple Exceptions*
- *Java try-with-resources*
- *Java Annotations*
- *Java Annotation Types*
- *Java Logging*
- *Java Assertions*

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.

## Many reasons

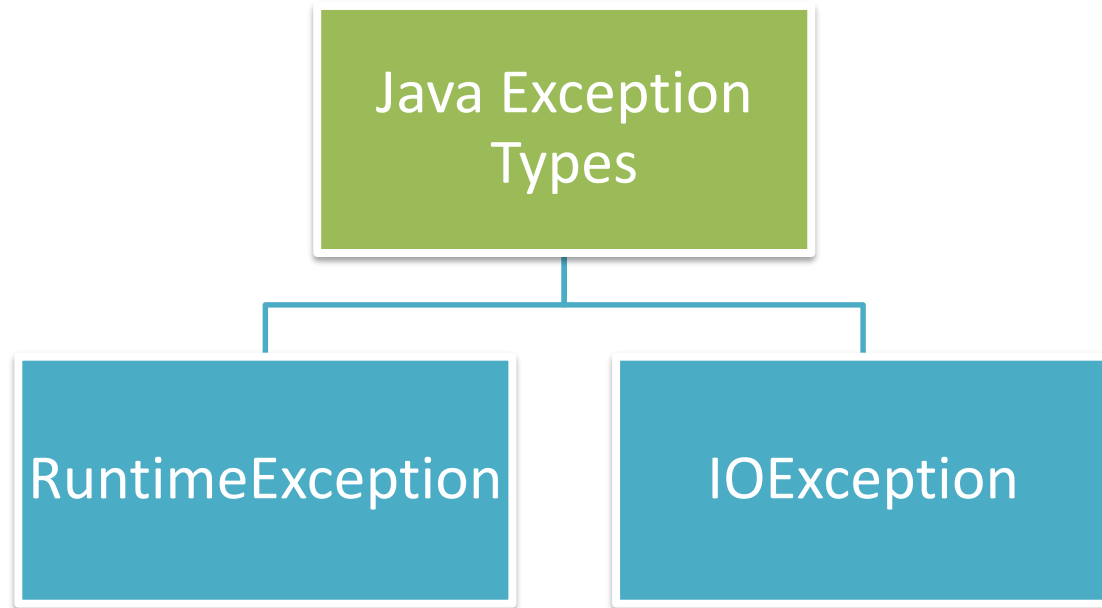


## Java Exception hierarchy



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

**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.



**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`

**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

## Catching and handling exceptions

In Java, we use the exception handler components try, catch and finally blocks to handle exceptions.

To catch and handle an exception, we place the **try...catch...finally** block around the code that might generate an exception. The finally block is optional.

```
try {  
    // code  
} catch (ExceptionType e) {  
    // catch block  
} finally {  
    // finally block  
}
```



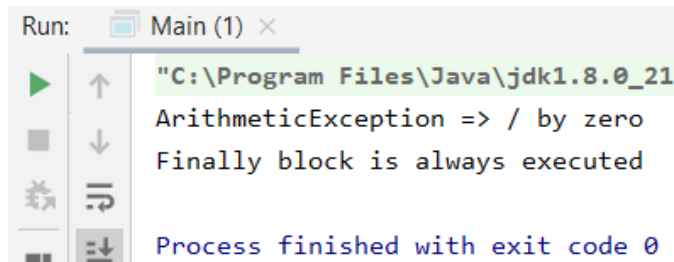
# Java Exception Handling

```
1 package OOP1.JavaException;
2
3 public class Main {
4     public static void main(String[] args) {
5
6         try {
7             int divideByZero = 5 / 0;
8             System.out.println("Rest of code in try block");
9         } catch (ArithmeticException e) {
10             System.out.println("ArithmeticException => " + e.getMessage());
11         }
12     }
13 }
14 }
```

Run: Main (1) x

```
▶ ↑ "C:\Program Files\Java\jdk1.8.0_21
■ ↓ ArithmeticException => / by zero
⚙ ⏮ Process finished with exit code 0
```

```
1 package OOP1.JavaException;
2
3 public class Main {
4     public static void main(String[] args) {
5
6         try {
7             int divideByZero = 5 / 0;
8             System.out.println("Rest of code in try block");
9         } catch (ArithmeticException e) {
10             System.out.println("ArithmeticException => " + e.getMessage());
11         } finally {
12             System.out.println("Finally block is always executed");
13         }
14     }
15 }
16 }
```



```
Run: Main (1) x
C:\Program Files\Java\jdk1.8.0_21
ArithmeticException => / by zero
Finally block is always executed
Process finished with exit code 0
```

## 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.

```
try {  
    // code  
} catch (ExceptionType1 | ExceptionType2 ex) {  
    // catch block  
}
```

In Java, exceptions can be categorized into two types:

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

## Java throws keyword

```
1 package OOP1.JavaException;
2
3 import java.io.File;
4 import java.io.FileInputStream;
5 import java.io.IOException;
6
7 public class Main1 {
8     public static void findFile() throws IOException {
9         // code that may produce IOException
10         File newFile=new File( pathname: "test.txt");
11         FileInputStream stream=new FileInputStream(newFile);
12     }
13
14     public static void main(String[] args) {
15         try{
16             findFile();
17         } catch(IOException e){
18             System.out.println(e);
19         }
20     }
21 }
```

Run: Main1 (1) ×

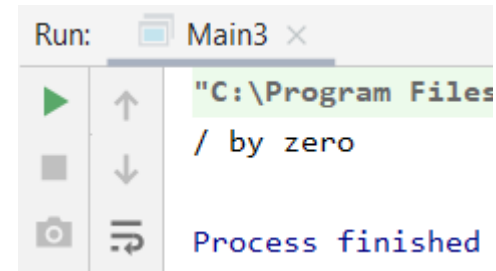
```
"C:\Program Files\Java\jdk1.8.0_211\bin\java.exe" ...
java.io.FileNotFoundException: test.txt (The system cannot find the file specified)

Process finished with exit code 0
```

# 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.

```
1 package OOP1.JavaException;
2
3 public class Main3 {
4     public static void main(String[] args) {
5         try {
6             int array[] = new int[10];
7             array[10] = 30 / 0;
8         } catch (ArithmeticException e) {
9             System.out.println(e.getMessage());
10        } catch (ArrayIndexOutOfBoundsException e) {
11            System.out.println(e.getMessage());
12        }
13    }
14 }
```



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.

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

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.

```
1 package OOP1.JavaException;
2
3 import java.io.BufferedReader;
4 import java.io.FileReader;
5 import java.io.IOException;
6
7 public class Main4 {
8     public static void main(String[] args) {
9         String line;
10        try(BufferedReader br = new BufferedReader(new FileReader( fileName: "test.txt"))) {
11            while ((line = br.readLine()) != null) {
12                System.out.println("Line =>" + line);
13            }
14        } catch (IOException e) {
15            System.out.println("IOException in try block =>" + e.getMessage());
16        }
17    }
18 }
```



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.

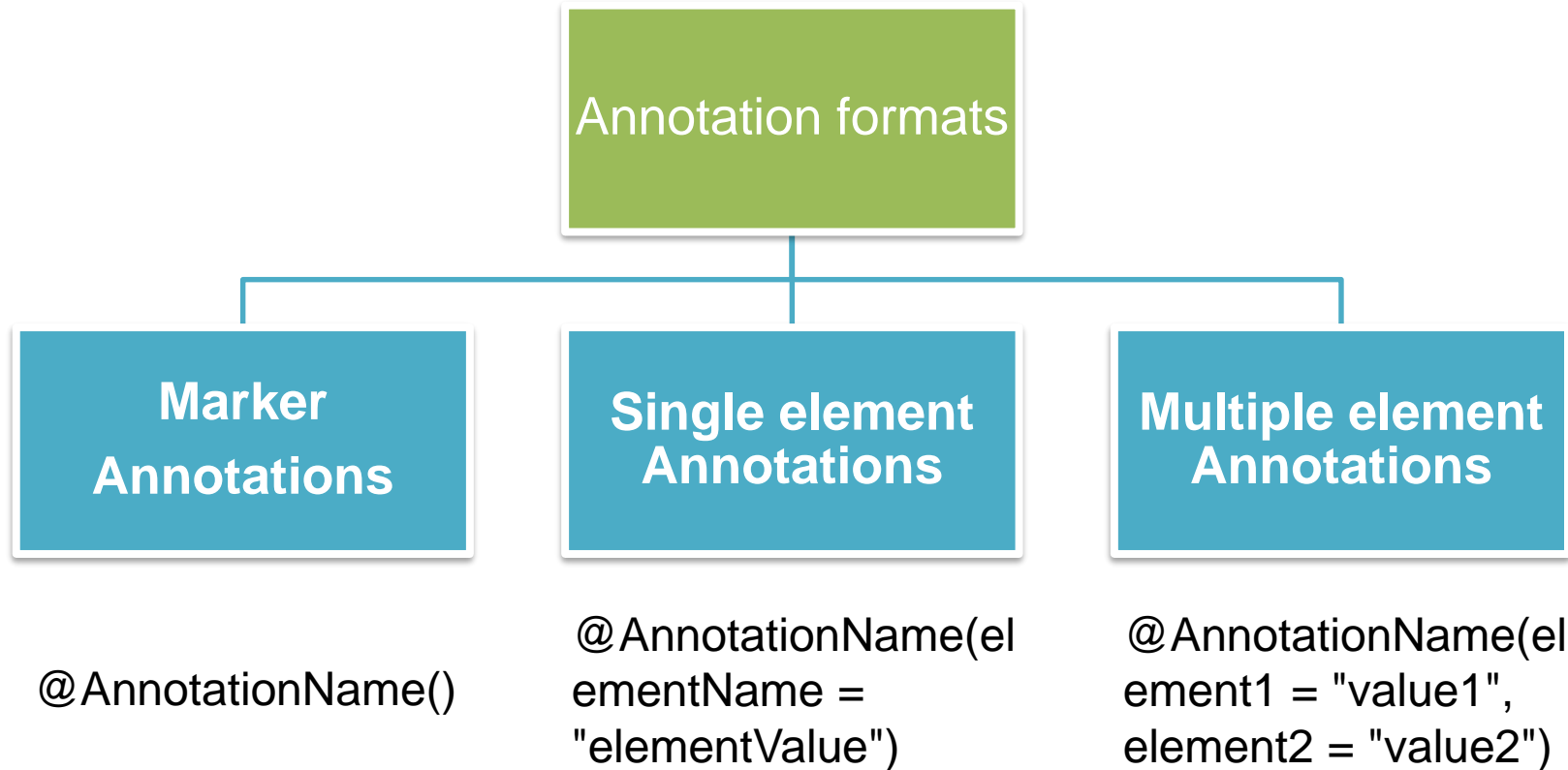
Annotations start with @. Its syntax is: @AnnotationName

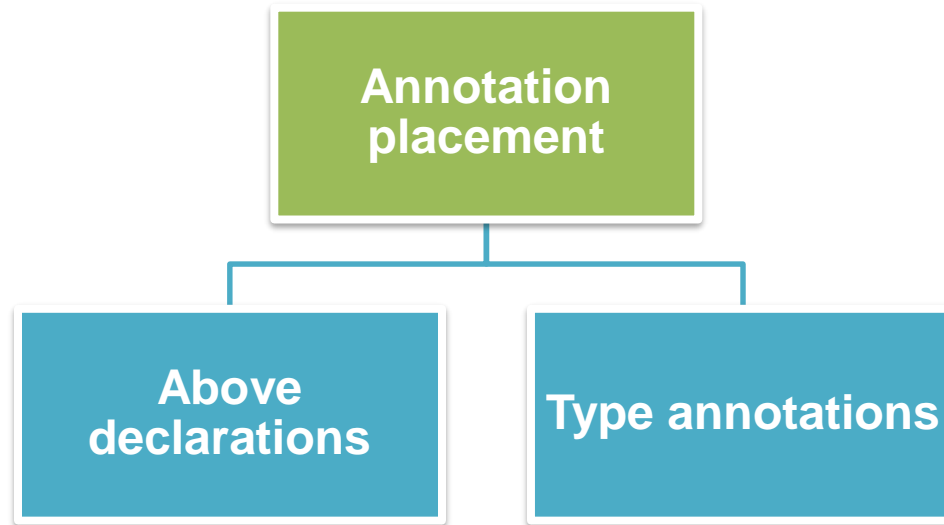
@Deprecated

@Override

@SuppressWarnings

@Overload





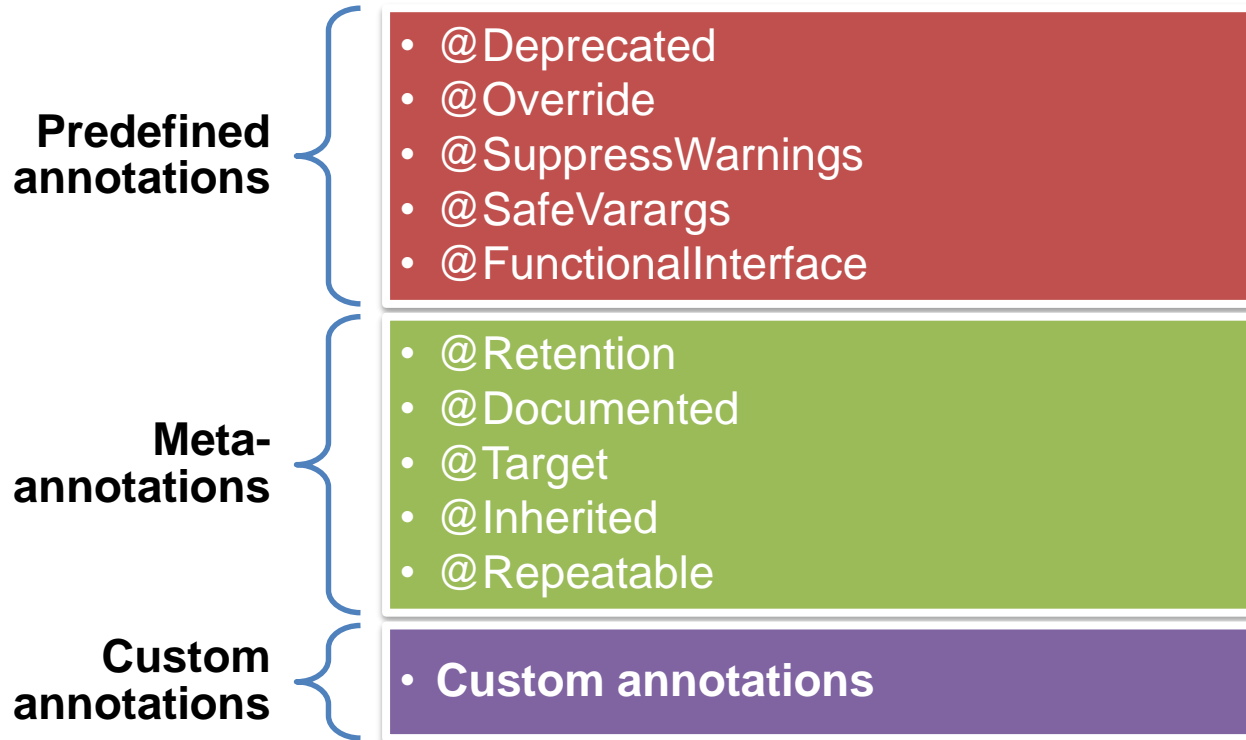
```
@Override  
public void displayInfo() {  
    System.out.println("LamNS3");  
}
```

## Use of Annotations

**Compiler instructions** - Annotations can be used for giving instructions to the compiler, detect errors or suppress warnings. The built-in annotations `@SuppressWarnings`, `@SuppressWarnings("unchecked")`, 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.



## Predefined Annotation Types

### @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.

### @Deprecated

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

## Predefined Annotation Types

### @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.

### @SuppressWarnings

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

## Custom Annotations

It is also possible to create our own custom annotations.

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

1. Annotations can be created by using **@interface** followed by the annotation name.
2. The annotation can have elements that look like methods but they do not have an implementation.
3. The default value is optional. The parameters cannot have a null value.
4. The return type of the method can be primitive, enum, string, class name or array of these types.



## @Retention

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

@Retention(RetentionPolicy)

**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.

## @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.

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

## @Target

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

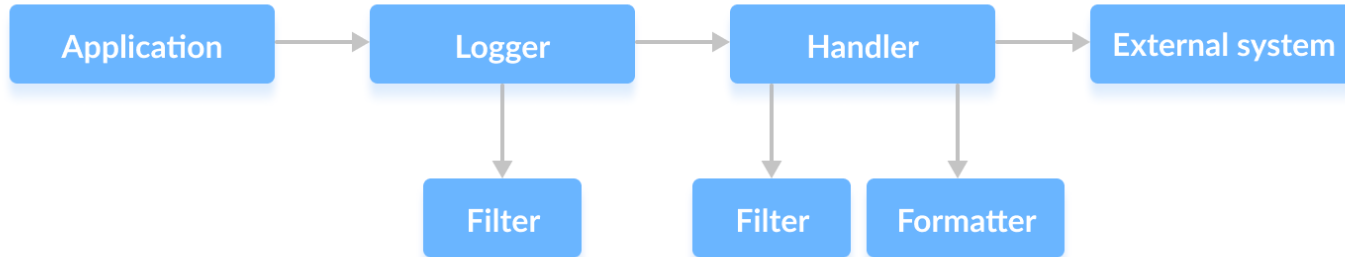
@Target(ElementType)

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

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

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.



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

```
Logger logger = Logger.getLogger(MyClass.class.getName());  
  
logger.info( "This is INFO log level message");
```

Log Level (in descending order)	Use
<b>WARNING</b>	warning message, a potential problem
<b>INFO</b>	general runtime information
<b>CONFIG</b>	configuration information
<b>FINE</b>	general developer information (tracing messages)
<b>OFF</b>	turn off logging for all levels (capture nothing)

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

//To add a new handler

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

//To remove a handler

```
logger.removeHandler(handler);
```

//A logger can have multiple handlers

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

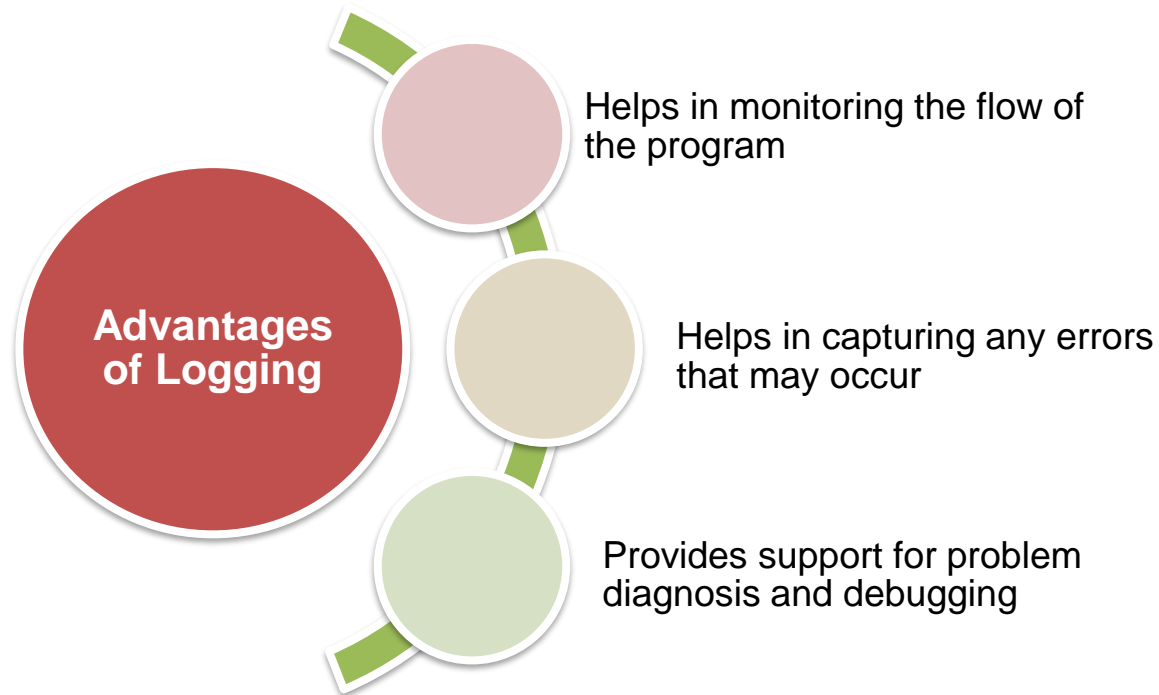
Handlers	Use
StreamHandler	writes to an OutputStream
ConsoleHandler	writes to console
FileHandler	writes to file
SocketHandler	writes to remote TCP ports
MemoryHandler	writes to memory

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

Formatters	Use
<code>SimpleFormatter</code>	formats <b>LogRecord</b> to string
<code>XMLFormatter</code>	formats <b>LogRecord</b> to XML form

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

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





# Thank you

