# Exceptions & Java 8 Features II

SDET Program

# Outline

- Exception
  - Categories of Exception
  - Exception Handling
  - Customized Exception
- Java 8 Features
  - Optional Class
  - Commonly used Functional Interfaces
  - Default and static methods for interfaces
  - Stream API
  - forEach() method in Iterable interface
- Java I/O
- Serialization

# Exceptions in Java

# Exceptions in Java

- An exception in Java is an unwanted or unexpected event, which occurs during the execution of a program (i.e., at runtime), that disrupts the normal flow of the program's instructions.
- Exceptions can arise due to various of situations:
  - Trying to access the 11th element of an array whose length is 10 (ArrayIndexOutOfBoundsException)
  - Division by zero (ArithmeticException)
  - Accessing a file which is not there (FileNotFoundException)
  - A thread is interrupted (InterruptedException)
  - Failure of I/O operations (IOException)
  - Dereferencing null (NullPointerException)
  - …

# Check Error Message

- Read error logs in IDE
  - When an error happens, the stack trace appears in the console at the bottom
  - The first line tells you what exception occurred and why
  - The lines below show the call stack (how the program got there)

```
Exception in thread "main" java.lang.ArrayIndexOutOfBoundsException Create breakpoint : 11
    at D1_Exception.ExceptionDemo.method3(ExceptionDemo.java:32)
    at D1_Exception.ExceptionDemo.main(ExceptionDemo.java:9)
```

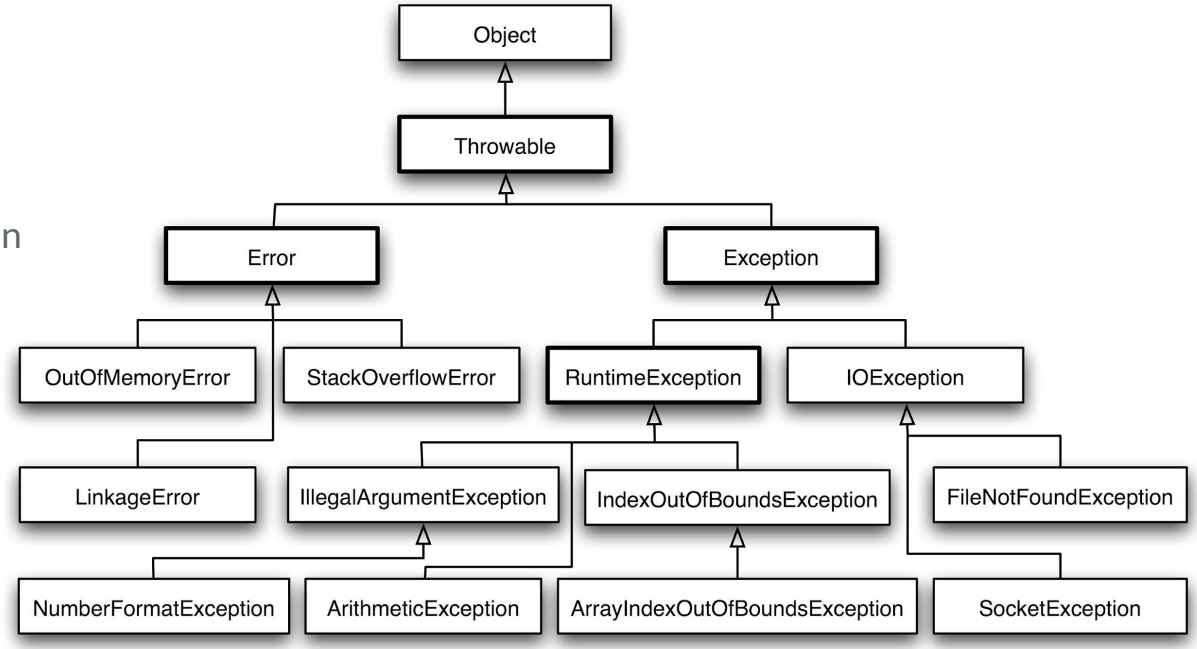
```
Exception in thread "main" java.net.UnknownHostException Create breakpoint : jsonmock.hackerrank2.com
    at java.base/sun.nio.ch.NioSocketImpl.connect(NioSocketImpl.java:572)
    at java.base/java.net.SocksSocketImpl.connect(SocksSocketImpl.java:327)
    at java.base/java.net.Socket.connect(Socket.java:633)
    at java.base/sun.security.ssl.SSLSocketImpl.connect(SSLSocketImpl.java:304)
    at java.base/sun.security.ssl.BaseSSLSocketImpl.connect(BaseSSLSocketImpl.java:174)
    at java.base/sun.net.NetworkClient.doConnect(NetworkClient.java:183)
    at java.base/sun.net.www.http.HttpClient.openServer(HttpClient.java:533)
    at java.base/sun.net.www.http.HttpClient.openServer(HttpClient.java:638)
    at java.base/sun.net.www.protocol.https.HttpsClient.<init>(HttpsClient.java:266)
    at java.base/sun.net.www.protocol.https.HttpsClient.New(HttpsClient.java:380)
    at java.base/sun.net.www.protocol.https.AbstractDelegateHttpsURLConnection.getNewHttpClient(AbstractDelegateHttpsURLConnection.java:193)
    at java.base/sun.net.www.protocol.http.HttpURLConnection.plainConnect0(HttpURLConnection.java:1257)
    at java.base/sun.net.www.protocol.http.HttpURLConnection.plainConnect(HttpURLConnection.java:1143)
    at java.base/sun.net.www.protocol.https.AbstractDelegateHttpsURLConnection.connect(AbstractDelegateHttpsURLConnection.java:179)
    at java.base/sun.net.www.protocol.http.HttpURLConnection.getInputStream0(HttpURLConnection.java:1702)
    at java.base/sun.net.www.protocol.http.HttpURLConnection.getInputStream(HttpURLConnection.java:1626)
    at java.base/sun.net.www.protocol.https.HttpsURLConnectionImpl.getInputStream(HttpsURLConnectionImpl.java:224)
    at com.beaconfire.DiscountedPriceBarcode.getDiscountedPrice(DiscountedPriceBarcode.java:26)
    at com.beaconfire.DiscountedPriceBarcode.main(DiscountedPriceBarcode.java:14)
```

# Exception vs. Error

- There is another kind of problem we might encounter when running Java programs - Error.
- An Error indicates serious problem that a reasonable application should not try to recover from.
    - Mostly Error is thrown by JVM in a **fatal** scenario. For example, OutOfMemoryError, StackOverflowError

# Java Exception Hierarchy

- The **Throwable** class is the superclass of all errors and exceptions in the Java language
- Throwable
  - Error
  - Exception
    - Checked Exception
    - Unchecked Exception



# Types of Exception - Definition

- Unchecked exception
  - Any classes extends **RuntimeException**
  - checked by the JVM at run time
  - e.g., ArrayIndexOutOfBoundsException, NullPointerException, …

- Checked exception
  - Any other **Exception**s that are not unchecked exception
  - checked by the compiler at compile time
    - Programmers are required to handle any possible checked exceptions
  - e.g., IOException, SQLException, InterruptedException, …

# Exception Handling

- try-catch
- throw/throws
- finally

# Try-catch

- try-catch block can be placed within any method that you feel can throw exceptions
- Code that might throw an exception goes inside the try block
- The catch block handles any exception that arises from the try block
- If an exception occurs in the try block, execution stops at that point, and control is immediately transferred to the matching catch block
    - You can write multiple catch blocks after a single try block to handle different types of exceptions, but only **the first matching catch block** will execute

# Try-catch

- Order of catch is important
  - catch having super class types should be defined later than the catch clauses with subclass types
  - most general handling must come last
- Nested try-catch block is allowed.
- Multi catch is available since Java 7
  - `catch (Exception1 | Exception2 | Exception3){...}`

```
try {
    // risky code
} catch (IOException e) {
    System.out.println("File error: " + e.getMessage());
} catch (ArithmeticException e) {
    System.out.println("Math error: " + e.getMessage());
} catch (Exception e) {
    System.out.println("General error: " + e.getMessage());
}
```

# Finally

- `finally` block can be added after try or try-catch
- Used to define code that must be executed after a try or try-catch block
- It executes in all circumstances, whether:
  - An exception occurs
  - No exception occurs
  - A method returns using return
- Typical use cases
  - When we need to release some resources (open a file, start a database connection, etc.) in the try block
  - An alternative way is using try-with-resources statement

# Throw, Throws

- Throw
  - Used to explicitly throw an exception
  - “Something is going wrong”
  - Syntax: `throw ThrowableInstance`
    - For example, `throw new InvalidCredentialException();`
- Throws
  - Added to the ***method signature*** to let the caller know about what exception the called method (or callee) can throw
  - It is the responsibility of the caller to either handle the exception (using try...catch mechanism) or it can also pass the exception (by specifying throws clause in its method declaration)
  - If all the methods in a program pass the exception to their callers (including **main( )**), then ultimately the exception passes to the default exception handler
- Throwing and catching errors are doable, but strongly discouraged

# Exception Propagation

- What is Exception Propagation?
  - Refers to how an exception is passed from one method to another in the call stack.
  - If a method does not handle an exception, it is propagated to the calling method.
- How does propagation work?
  - When an exception occurs, the JVM searches for a matching catch block in the current method
  - If no catch block is found, the exception is propagated to the calling method
  - This process continues until the exception is handled or reach the main() method
  - If unhandled, the JVM terminates the program and prints the stack trace
- Key points
  - Checked exceptions must be explicitly declared using throws and handled by the caller
  - Unchecked exceptions are propagated automatically without requiring declaration

# Customized Exception

- A class which is a subclass of Exception (or RuntimeException)
- Provide constructor as needed
- Use throw keyword to throw exception whenever you want to raise the exception
- Can be either checked or unchecked exception
    - Extend Exception class if you want to create checked exception
    - Extend RuntimeException class if you want to create unchecked exception

# Java 8 Features

# Java 8 Features

- Functional Interface and Lambda Expressions
- Default and static methods for interfaces
- Optional Class
- Stream API for Bulk Data Operations on Collections
- forEach() method in Iterable interface (and more!)

# Optional Class

- Optional is a **container object** used to contain not-null objects. Optional object is used to represent null with absent value

```
String input = null;
Optional<String> name = Optional.ofNullable(input);
if (name.isPresent()) {// Check if value is present
    System.out.println("Name: " + name.get());
} else {
    System.out.println("Name is missing");
}
// Provide default value
System.out.println(name.orElse("Default Name"));
```

- This class has various utility methods to facilitate code to handle values as ‘available’ or ‘not available’ instead of checking null values
- Useful for circumventing handling of NullPointerException

# Optional Class - Useful Methods

- https://docs.oracle.com/javase/8/docs/api/java/util/Optional.html

| Method Signature | Description |
| --- | --- |
| `static <T> Optional<T> empty()` | Returns an empty `Optional` instance. |
| `static <T> Optional<T> of(T value)` | Returns an `Optional` with the specified present non-null value. |
| `static <T> Optional<T> ofNullable(T value)` | Returns an `Optional` describing the value if non-null, otherwise an empty `Optional`. |
| `void ifPresent(Consumer<? super T> consumer)` | Invokes the consumer if a value is present; does nothing otherwise. |
| `boolean isPresent()` | Returns `true` if a value is present, otherwise `false`. |
| `T orElse(T other)` | Returns the value if present, otherwise returns the given default value. |
| `T orElseGet(Supplier<? extends T> other)` | Returns the value if present, otherwise invokes the supplier and returns the result. |
| `Optional<T> filter(Predicate<? super T> predicate)` | Returns the value if it matches the predicate, otherwise returns an empty `Optional`. |

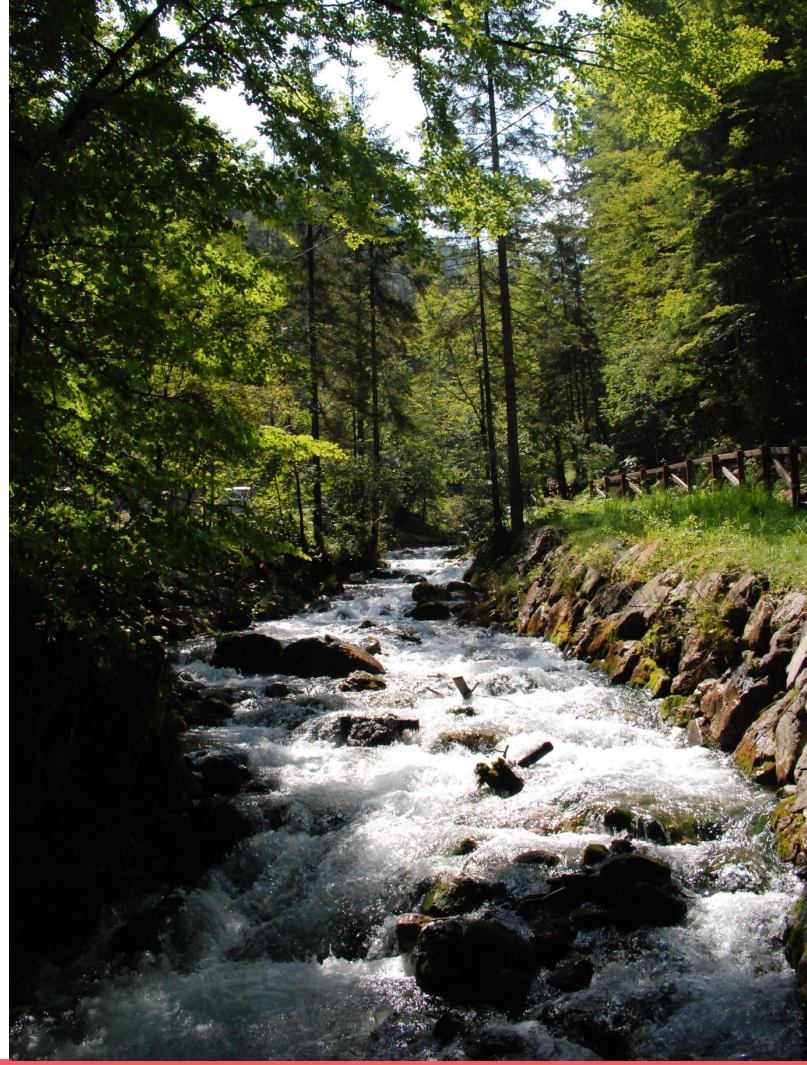
# Functional Interfaces

- Consumer<T>
  - Represents an operation that accepts a single input argument and returns no result.
  - `void accept(T t)`
    `Consumer<T> andThen(Consumer<? super T> after)`
- Supplier<T>
  - Represents a supplier of results.
  - `T get()`
- Function<T,R>
  - Represents a function that accepts one argument and produces a result.
  - `R apply(T t)...`
- Predicate<T>
  - Represents a predicate (boolean-valued function) of one argument.
  - `boolean test(T t)...`

# Stream API

- Introduced in Java 8, the Stream API is used to process collections of objects.

- A stream is a **sequence of objects** that supports various methods which can be pipelined to produce the desired result.



# Stream API - Main features

- A Stream is not a data structure, meaning it does not store data.
- Instead, it takes input from sources like Collections, Arrays, or I/O channels.
- Streams do not modify the original data source. They produce a new result based on a pipeline of operations.
- We can do two types of operations on Stream:
    - Intermediate Operation: takes a stream, returns a stream. Hence various intermediate operations can be chained.
    - Terminal Operation: marks the end of the stream, and return the actual result.

# Stream API - Example

- Let's take a look at an example:

```
public class Person {
  public String name;
  public int age;
  // constructors

  public static void main(String[] args) {
    List<Person> l = new ArrayList<>();
    l.add(new Person("Alice", 24));
    l.add(new Person("Bob", 22));
    l.add(new Person("Carson", 17));
  }
}
```

- Suppose there is a list of people, we would like to get a list of names of all adults, and the result should be in all uppercase and sorted in Alphabetical order

# Stream API - Example

- Solution using Stream API
  - `stream()` to create a stream
  - Multiple intermediate methods
  - `collect()` to collect the result

```
public List<String> solve(List<Person> input) {
  return input.stream() // Stream<Person>
    .filter(p -> p.age >= 18) // Stream<Person>, adults
    .map(p -> p.name) // Stream<String>
    .map(String::toUpperCase) // Stream<String>
    .sorted() // based on ?
    .collect(Collectors.toList()); // List<String>
}
```

# Intermediate VS. Terminal Operations

- Return value
  - Intermediate operations return a stream as a result and terminal operations return non-stream values like primitive or object or collection or may not return anything.
- Intermediate operations are lazily executed
  - When you call intermediate operations, they are actually not executed. They are just stored in the memory and executed when the terminal operation is called on the stream.
  - For example, if you write a statement like:
    - `Stream<Person> stream = input.stream().filter(p -> p.age >= 18);`
- Example methods
  - Intermediate operations: map(), filter(), distinct(), sorted(), limit(), skip()
  - Terminal operation: forEach(), toArray(), collect(), min(), count(), anyMatch()

# Java I/O

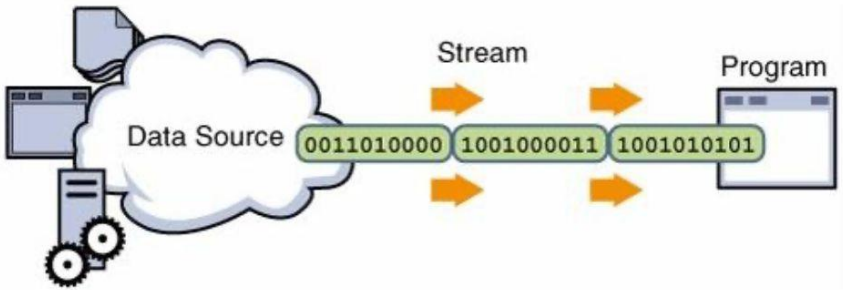
# Java I/O

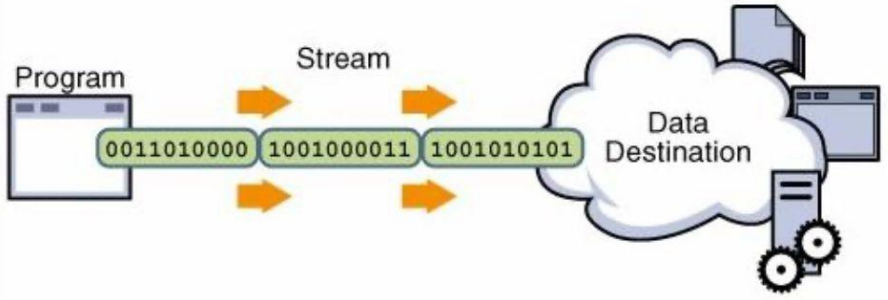
- Most application need to process some data and produce some output based on the input. Java I/O (Input and Output) is used to process the input and produce the output (read and write data).
- Here, I/O is relative to the Java program:
  - Input refers to data coming into the program, such as from a user, file, or network
  - Output refers to data going out from the program, such as writing to a file, console, or sending data over a network

- The java.io package contains all the classes required for input and output operations

# Streams

- Java uses the concept of STREAM to make I/O operation fast
- A stream is a conceptually endless flow of data. We can either read from a stream or write to a stream
  - “Input” Stream: connected to a data source
  - “Output” Stream: connected to a data destination
- Stream does not have concept of an index (like array), nor can we typically move forward and backward (like array). It is just a flow of data.



# Different Types of Streams

- In Java, stream can either be byte based (reading and writing bytes) or character based (reading and writing character)
  - 1 byte = 8 bits, 1 character = 16 bits (java uses UTF-16 encoding)
- Byte Stream
  - Naming convention: **xxxInputStream**, **xxxOutputStream**
  - It provides a convenient means for handling input and output of byte. Used when working with bytes or other binary objects
- Character Stream
  - Naming convention: **xxxInputStreamReader**, **xxxOutputStreamWriter**
  - Character stream uses Unicode and therefore can be internationalized. Used when working with characters or strings.

# I/O Stream

- InputStream: an abstract class. FileInputStream, FilterInputStream, …
  - read() — read a byte from input stream
    - read() method returns actual number of bytes that were successfully read or -1 if end of the file (EOF) is reached.
  - read(byte[] b) — read from input into a byte array b
  - close() — closes the input stream
- OutputStream: an abstract class. FileOutputStream, PrintStream, …
  - write() — write a byte to output stream
  - write(byte[] b) — write all bytes in b into output stream
  - close() — Closes the output stream
  - flush() — flushes the output stream. It forces the buffered data to be physically written to the destination (e.g., file, network)

# Buffer, Buffered I/O Stream

- A buffer is a temporary memory area used to hold data while it's being moved between two places (e.g., from disk to memory).
  - It reduces the number of direct reads/writes, improving performance.
- BufferedInputStream, BufferedOutputStream
  - Wraps a basic InputStream / OutputStream to improve efficiency when reading/writing data.
  - The default chunk size is 8 KB (8192 bytes)

```
BufferedInputStream bis = new BufferedInputStream(new FileInputStream("in.txt"));
BufferedOutputStream bos = new BufferedOutputStream(new FileOutputStream("out.txt"));
```

# File I/O (java.io)

- FileInputStream, FileOutputStream
- FileReader, FileWriter

- FileReader(String filePath) – Reads a file from the given file path.
- FileReader(File fileObj) – Reads from a File object
- FileWriter(String fileName) – Creates or overwrites the file with new content.
- FileWriter(String fileName, boolean append)
    - If append is true, data is written at the end of the file
    - If append is false, the file is overwritten.

# java.io.File

- The java.io.File class represents a file or directory (folder) path in the filesystem. It is part of the java.io package and provides methods to:
  - Create new files or directories
  - Check if a file exists
  - Get file metadata (name, size, path, etc.)
  - Delete or rename files
  - Traverse directories
- APIs
  - File(String pathname) – Creates a new File instance from a file or directory path
  - file.exists() – Checks if the file or directory exists
  - file.isFile() – Returns true if it's a file
  - file.isDirectory() – Returns true if it's a directory

# java.io.File - Example

```
import java.io.File;

public class FileInfo {
    public static void main(String[] args) {
        // Accept file name or directory name through command line args
        String fname = args[0];

        // Pass the filename or directory name to File object
        File f = new File(fname);

        // Apply File class methods on File object
        System.out.println("File name: " + f.getName());
        System.out.println("Path: " + f.getPath());
        System.out.println("Absolute path: " + f.getAbsolutePath());
        System.out.println("Parent: " + f.getParent());
        System.out.println("Exists: " + f.exists());

        if (f.exists()) {
            System.out.println("Is writeable: " + f.canWrite());
            System.out.println("Is readable: " + f.canRead());
            System.out.println("Is a directory: " + f.isDirectory());
            System.out.println("File Size in bytes: " + f.length());
        }
    }
}
```

# Serialization

# Serialization

- Is a built-in mechanism in Java
- **Serialization**: convert java object to bytes, including the type and data fields information
- Once serialized, object will be stored in somewhere as byte. These data can be read by other input streams and convert back into Java Objects. This process is called **deserialization**.
- Why is this important?
    - Serialization in Java allows an object's state to be converted into a byte stream so it can be saved or sent over a network.
    - This process is platform independent. An object can be serialized on one machine and deserialized on another, even if they have different operating systems.
    - Later on we will learn JSON format, which is another common, language-independent way to exchange data between systems (e.g., between a Java app and a web service).

# Serializable

- java.io.Serializable is a marker interface
  - Marker interface: has no methods or constants inside it. It provides run-time type information about objects, so the compiler and JVM have additional information about the object.
  - e.g., Cloneable, Serializable
- Classes that do not implement this interface will not have any of their state serialized or deserialized.
  - `ObjectOutputStream.writeObject(user);`
  - `user = (User) ObjectInputStream.readObject();`
  - These methods will throw an exception if User class is not serializable.

# Transient keyword

- What if we don’t want to serialize certain fields?
  - **transient** keyword can be used to exclude specific fields from serialization

```
public class User implements Serializable {
    private String username;
    private transient String password;  // This will NOT be serialized

    public User(String username, String password) {
        this.username = username;
        this.password = password;
    }
}
```

- When a user object is serialized:
  - username is saved.
  - password is not saved (it will be null when deserialized).

# Deserialization

- Deserialization Flow (Reading Objects from a File)
  - FileInputStream → ObjectInputStream → Java objects
  - The deserialized objects are returned as type Object.
  - It's the developer's responsibility to cast them to the correct class.
- Process
  - During deserialization, the JVM loads the class and restores the object's state from the byte stream.
  - Constructors are not called for classes that implement Serializable.
  - Constructors are called only for non-serializable superclass(es) in the object's hierarchy.
- Transient variables are given null or default values (0, false, null) for primitives.

# SerializationUID

- serialVersionUID is a special version control identifier used during Java serialization and deserialization.
- Why is it important?
  - When an object is deserialized, Java checks if the serialVersionUID of the serialized class (in the file) matches the current class in your code.
  - If they don't match, you get an InvalidClassException.
- Not frequently used
  - It is not recommended to change the structure of existing classes (violates solid). Usually new classes are created and new tables are created.

```
class Student implements Serializable {
    private static final long serialVersionUID = 1L;
    String name;
}
```

# Thank You!