LEARN PROGRAMMING IN JAVA™

Learning OO Java™ as a Beginner – Enums

COURSE AGENDA

- What is an Object
- The Basics
- Operators
- Control Flow
- Implementation Hiding
- Reuse
- Interfaces
- Polymorphism
- Collections
- Functional Programming
- Streams
- Exceptions
- Enums

INSTRUCTOR – HUGO SCAVINO

- 30 Years of IT Experience
- Using Java since the beta
- Taught Java and OOP in USA, UK, and France to Fortune 500
- Senior Software Architect with
  - DTCC
  - Penske
  - HSBC
  - Government Agencies

https://www.linkedin.com/in/hugoscavino/

REQUIRED SOFTWARE

- JDK™
  - JDK 8 or Newer (Open Source, Amazon, or Oracle)
  - The beginner course focuses on core language constructs; feel free to use 11, 17, 21, etc.

- IDE
  - IntelliJ Community Edition (Free) or Enterprise (Paid)
  - While you can use any IDE with Java, the course and labs are explicitly made with IntelliJ in mind

RECOMMENDED TEXTS

- Java 5 Book
  - Thinking in Java – 4th Edition - Free – PDF - GitHub
  - Thinking in Java – 4th Edition – Hard Cover - Amazon

- Recommended Java 8 Book
  - Bruce Eckel on Java 8
    - Contains references to newer Java syntax
- #1 Best Seller in Beginner's Guide to Java Programming
  - Head First Java: A Brain-Friendly Guide 3rd Edition

TOPIC DESCRIPTION

Enums: …

KEY CONCEPTS

- `Throwable`: The root class of all errors and exceptions.
- `Error`: Represents serious problems that applications should not attempt to handle (e.g., `OutOfMemoryError`).
- `Exception`: Represents issues that applications can handle.
- Checked Exceptions: Must be declared in the throws clause or handled using a `try-catch` block (e.g., `IOException`, `SQLException`).
- Unchecked Exceptions: Runtime exceptions that do not require explicit handling (e.g., `NullPointerException`, `ArrayIndexOutOfBoundsException`).

- *Note, in C# everything is an unchecked exception. We can discuss the better choice now that we have been using exception handling for years.

KEY CONCEPTS

- Throwing an Exception: Generating an exception object using the `throw` keyword.
- Catching an Exception: Handling an exception using a `try-catch` block.
- Propagating an Exception: Passing the exception up the call stack if not caught locally.

```
Caused by: java.sql.SQLException: Violation of unique constraint MY_ENTITY_UK_1: duplicate value(s) for column(s)
MY_COLUMN in statement [...]
    at org.hsqldb.jdbc.Util.throwError(Unknown Source)
    at org.hsqldb.jdbc.jdbcPreparedStatement.executeUpdate(Unknown Source)
    at com.mchange.v2.c3p0.impl.NewProxyPreparedStatement.executeUpdate(NewProxyPreparedStatement.java:105)
    at org.hibernate.id.insert.AbstractSelectingDelegate.performInsert(AbstractSelectingDelegate.java:57)
    ... 54 more
```

EXCEPTION HANDLING MECHANICS

try {..}:

It contains the code that might throw an exception.

catch (Exception e):

Handles the exception(s).

finally (optional):

Contains code that will always execute, even if an exception is thrown

throws:

Used in method declarations to specify the exceptions that a method can throw. For Checked exceptions.

EXAMPLE

```java
try {
    int result = 10 / 0; // This will throw an ArithmeticException
} catch (ArithmeticException e) {
    System.out.println("Cannot divide by zero: " + e.getMessage());
} finally {
    System.out.println("This block always executes.");
}
```

* The catch clause can catch multiple Exceptions and not constrained to one like above

CLASSIC EXAMPLE

```java
try {
    int result = 10 / 0; // This will throw an ArithmeticException
} catch (ArithmeticException e) {
    System.out.println("Cannot divide by zero: " + e.getMessage());
} finally {
    System.out.println("This block always executes.");
}
```

* The catch clause can catch multiple Exceptions and not constrained to one like above

MULTI CATCH EXAMPLE

```
try {
    int result = 10 / 0; // This will throw an ArithmeticException
    if(true) {
        throw new SQLException("Something went wrong in DB");
    }
} catch (ArithmeticException  | SQLException e){
    System.out.println(e.getMessage());
}
```

* A ‘finally’ is not required

CUSTOM EXCEPTIONS

```java
public class CustomerException extends Exception {

    final int customerId;

    public CustomerException(String message, int customerId) {
        super(message);
        this.customerId = customerId;
    }

    public int getCustomerId() {
        return customerId;
    }
```

Be careful when creating an exotic `Exception` hierarchy. Do you need to capture the metadata?

RETHROWING AN EXCEPTION

Rethrow: When you want to peek (perhaps log) at the Exception before sending the Exception back out of the stack.

```
void demo() throws CustomerCheckedException {
    try {
        foreverThrowing();
    }
    catch (CustomerCheckedException ce){
        // Do something with the Exception or don't bother
        System.out.println("Caught CustomerException: " + ce.getMessage());
        throw ce;
    }
}
```

You can re-throw a different Exception

RUNTIME EXCEPTIONS

`RuntimeException`: Inherit from it, and your Exception will not be checked.

```
// From the JavaDoc of RuntimeException
// RuntimeException and its subclasses are unchecked exceptions.
// Unchecked exceptions do not need to be declared in a method
// or constructor's throws clause if they can be thrown by
// the execution of the method or constructor and propagate
// outside the method or constructor boundary.
public class CustomRuntimeException extends RuntimeException {
}
```

CHECKED AND UNCHECKED METHODS

```java
// No Exception in the signature
void mayOrMayThrow()  {
    boolean someCondition = new Random().nextBoolean();
    if(someCondition){
        throw new CustomRuntimeException();
    }

}
// The exception is part of the signature
void foreverThrowing() throws CustomerCheckedException {
    throw new CustomerCheckedException("I always throw!", 100);
}
```

Exceptions are part of your design. Think through the consequences, especially if you are building a library meant for reuse!

DO WE CARE ABOUT FINALLY THESE DAYS?

In the good old days, resources were scarce and error-prone. Java programmers had to release, close, and otherwise manage the lifetime of certain objects like files, databases, network connections, and especially hardware devices.

Do we care now?

There are far fewer reasons why closing a connection is required. When it is, use the `finally` block otherwise….

HARDWARERESOURCE CLASS

```java
// Wrapper over some scare resource like a socket or sprinkler
public HardwareResource() throws Exception {
    boolean available = true;
    if (!available) {
        throw new Exception("Hardware resource not available");
    }
}

public void start(){
    // open connections
    // Call some hardware API
    System.out.println("Hardware resource started");
}

public void stop(){
    // Close connections
    // Call some hardware API
    System.out.println("Hardware resource stopped");
}
```

ATTEMPTING TO CLOSE THE RESOURCE

Using finally

```
void startAllDevices() throws Exception {

    try {
        hw = new HardwareResource();
        hw.start();
    } catch (Exception e) {
        throw new Exception("Hardware Device Failed" + e);
    } finally {

        if (hw != null){
            hw.stop();
            System.out.println("Attempting to stop hardware");
```

Looks like old-style C programming. But what happens if `stop()` can throw an `Exception`?

ATTEMPTING TO CLOSE THE RESOURCE

Using finally and yet another try

```
} finally {

    if (hw != null){
        System.out.println("Attempting to stop hardware");
        // What if stop throws an Exception!
        // This code style is madness!
        try {
            hw.stop();
            System.out.println("Stopped hardware");
        }
        catch (Exception e) {
            System.out.println("Hardware Failure to Stop");
        }

    }
```

DON'T GOBBLE EXCEPTIONS!

Using finally and forgetting the catch

```
try {
    hw = new HardwareResource();
    hw.start();
}
finally {
  hw.stop();
    System.out.println("Stopped hardware");
}
```

You do not have to use the `catch` statement and accidentally only rely on the `finally` clause. Good luck debugging this code.

TRY-WITH-RESOURCES TO THE RESCUE

What if objects created in the `try-with-resources` definition clause (within the parentheses) implement the interface `java.lang.AutoCloseable`?

If you do, then Java promises to call the interface on your behalf, and we can stop the final madness.

EXAMPLE

```
// Class now implements AutoCloseable and can be used in a try-with-resource block
public class CloseableHardwareResource implements AutoCloseable{


void startAllDevicesWithResources() throws Exception {

    try (CloseableHardwareResource hw = new CloseableHardwareResource()) {
        hw.start();
    }
    catch (Exception e) {
        throw new Exception("Hardware Device Failed to start" + e);
    }
    // no need for the finally!

}
```

JDK 9 EXAMPLE

```java
// Class now implements AutoCloseable and can be used in a try-with-resource block
public class CloseableHardwareResource implements AutoCloseable{

 void startAllDevicesWithResourcesJdk9() throws Exception {
     CloseableHardwareResource hw = new CloseableHardwareResource();
     AnotherCloseableHardwareResource ahw = new AnotherCloseableHardwareResource();
     try (hw ; ahw) {
         hw.start();
     }
     catch (Exception e) {
         throw new Exception("Hardware Device Failed to start" + e);
     }
     // no need for the 'finally' clause!
 }
```

EXCEPTION BEST PRACTICES

Following best practices when using and designing with exceptions in Java ensures clarity, maintainability, and robustness in your code.

Use Specific Exceptions

Catch specific exceptions rather than general ones like Exception or Throwable. This makes the code more predictable and easier to debug.

```
try {
    // risky code
} catch (IOException e) {
    // Handle IO exception
} catch (NullPointerException e) {
    // Handle null pointer exception
}
```

EXCEPTION BEST PRACTICES

Don't Use Exceptions for Control Flow

Don’t use exceptions to control program logic. This is inefficient and makes the code harder to read.

```
// Bad
try {
    int index = list.indexOf(element);
   // Using Exception for unnecessary error logic
} catch (Exception e) {
    // BE proactive instead
}

// Better - Check for the condition first
if (list.contains(element)) {
    int index = list.indexOf(element);
}
```

EXCEPTION BEST PRACTICES

Always Clean Up Resources

Use the `finally` block or `try-with-resources` to release resources like files, sockets, or database connections.

```
FileReader fr = new FileReader("file.txt");

try (
    BufferedReader reader = new BufferedReader(fr)) {

    String line = reader.readLine();

} catch (IOException e) {
    …
}
// No need for finally block; resources are auto-closed
```

EXCEPTION BEST PRACTICES

Provide Meaningful Messages

```
throw new IllegalArgumentException("Age must be greater than 0.");
```

Use Custom Exceptions When Necessary

- Create custom exceptions to represent specific error conditions in your application.
- Extend `Exception` for checked exceptions.
- Extend `RuntimeException` for unchecked exceptions.

EXCEPTION BEST PRACTICES

Avoid Swallowing Exceptions

```
throw new IllegalArgumentException("Age must be greater than 0.");
```

```
try {
    // some code
} catch (Exception e) {
    // print to stack trace
    // ignore the entire situation
    // write to a bogus file, repeatedly
}
```

EXCEPTION BEST PRACTICES

Log Exceptions Properly

Use a logging framework (e.g., SLF4J, Log4J) to log exceptions instead of `System.out` or `e.printStackTrace()`.

```
try {
    // some important code
} catch (Exception e) {
    e.printStackTrace();
}
```

EXCEPTION BEST PRACTICES

Rethrow Exceptions with Context

If you catch an exception but cannot handle it, rethrow it with additional context to help future you or the next maintainer.

```
try {
    // some important code
} catch (Exception e) {
    throw new RuntimeException("CustId" + ci +
    " " + e.printStackTrace();
}
```

EXCEPTION BEST PRACTICES

Keep Exception Hierarchies Simple

Avoid overly complex inheritance hierarchies for many custom exceptions (many of which solely inherit from Exception and do little else.

Example of a simple structure:

- ApplicationException (base exception)
- ValidationException (specific type)
- DatabaseException (specific type)

EXCEPTION BEST PRACTICES

Separate Checked and Unchecked Exceptions

- Use checked exceptions when the caller can reasonably recover from the issue (e.g., `IOException`).
- Use unchecked exceptions for programming errors (e.g., `NullPointerException`, `IllegalArgumentException`).

EXCEPTION BEST PRACTICES

Separate Checked and Unchecked Exceptions

- Use checked exceptions when the caller can reasonably recover from the issue (e.g., `IOException`).
- Use unchecked exceptions for programming errors (e.g., `NullPointerException`, `IllegalArgumentException`).

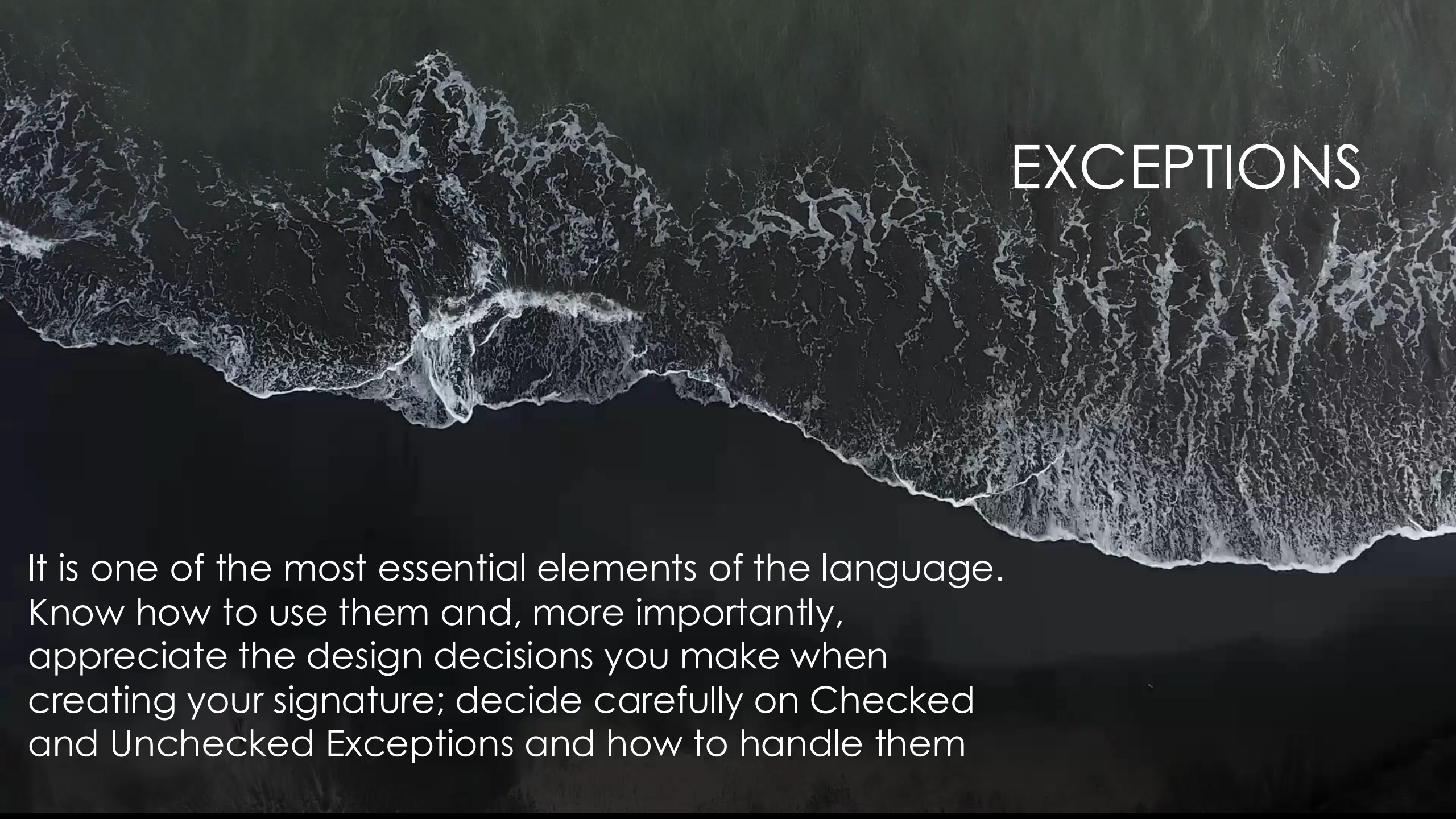
Catch Exceptions Only When Necessary

- Avoid catching exceptions if you cannot do anything meaningful with them. Let them propagate to higher layers for better handling. Let go!

EXCEPTION BEST PRACTICES

Avoid Overusing Checked Exceptions

- Too many checked exceptions can make code verbose and hard to maintain. Consider using unchecked exceptions for simpler logic, like the team from C#. We can all learn from experience.



EXCEPTIONS

It is one of the most essential elements of the language. Know how to use them and, more importantly, appreciate the design decisions you make when creating your signature; decide carefully on Checked and Unchecked Exceptions and how to handle them