Creating Exceptions

Est. Learner Time: 3 hours 45 min

Terminal LOs:

- Examine whether to throw an existing exception or implement a new exception for a given error case
- Write code that appropriately transforms exceptions by chaining or translating exceptions given a scenario (new LO)
  - Explain how and when to chain (or wrap) an exception to hide implementation details from the caller (combined LOs)
  - Explain how and when to translate exceptions to intentionally drop details of original cause of exceptions (combined LOs)
  - Explain why making custom exceptions can be helpful in the debugging process
- Create custom exceptions
  - Design and implement an exception class hierarchy for a code base
  - Understand that it is a good practice to provide an Exception subclass with the same public constructor signatures as the base java.lang.Exception class
  - Design and implement an exception that describes an error case specific to a service
  - Produce a unique serialVersionUID when creating a new exception type
  - Outline the public constructors of the Exception class
- Define error cases
  - Explain how to define error cases given a set of requirements

Introduction:

Review Link:

Est. Learner Time: 10 min

Slide Number	Visual Content/Text/Assets	Text	Notes
1	**What is an Exception?** Decorative image: Blue screen of death	Remember: an exception is an error. It denotes something that "didn't follow the rules". There are several common times when a method may hit an issue causing it to throw an exception. Fatal errors include anything that causes your program to unexpectedly stop running. The infamous Windows "blue screen of death" is an example of a fatal system error in the Windows operating system that could not be caught and handled. Improper arguments or invalid data are other reasons to throw an exception, indicating that your method has not received the proper input. An example of an exception is a ResourceUnavailableException. This indicates that the method could not reach a resource it expected to find, for example, if the connection to a required database is lost. Anything else that causes the method to deviate from the way it was designed should also throw an exception. Java includes and makes use of a variety of pre-defined exceptions in the JDK but as you are developing your code you may find a need for your own, custom exceptions. Please refer to the lesson on exception handling for the pre-defined exceptions.	
2		In the lesson on exception handling, you learned what an exception is, and how to handle one. You learned that	

		exceptions are part of an exception hierarchy, and that Java exceptions extend the *java.lang.Exception* class.	
3	Show same chart used in Exception Handling lesson that shows Exception Hierarchy	Built-in exceptions can be Checked or Unchecked Exceptions, and each have slightly different rules for handling or propagating. Remember with checked exceptions the Java compiler enforces the code to have certain handling rules. For example, declaring that the method throws the exception. Unchecked exceptions don't have those compiler-enforced restrictions. As you see in the chart, exceptions rely on inheritance to create a hierarchy of detail, allowing some exceptions to be general and others to locate more specific error conditions. It is an important design choice to identify whether to handle the exception locally or instead propagate the exception up to the calling method with logging or other helpful information.	
4		In this lesson, you will learn to manipulate exceptions and even create your own. Developers often need to create their own exceptions when there are exceptions specific to the workflows or business logic they build. These help the users or developers understand more about the exact problems that can come up in the program. Developers also manipulate or customize existing exceptions. The same exceptions Java provides may need special treatment so that you adjust or provide different information than the exception ordinarily conveys.	
5	**What's Next?**	• Instructional Lesson 1: Transforming Exceptions • Activity 1: Transforming Exceptions • Instructional Lesson 2:	

Glossary:

- **Checked Exceptions –** These are exceptions checked by the compiler at compile-time. These are declared in the code as part of method signatures and represent errors developers should be prepared to manage.
- **Unchecked Exceptions –** These are exceptions that are thrown at runtime. Because of their nature, they do not have to be declared for the code to continue. These do not have the same handling or propagation rules as checked exceptions.
- **Chaining Exceptions –** Using one exception to relate to another exception. Commonly described as wrapping one exception in the "cause" field of a new exception.
- **Translating Exceptions** – A type of exception transformation that obscures information. This is usually done for security reasons, so that information is hidden from users. This does make it more difficult for developers to track down errors, so this is used less often than chained or other custom exceptions.

Instructional Lesson 1: Transforming Exceptions

Review Link:

Est. Learner Time: 20 min

LOs:

- Examine whether to throw an existing exception or implement a new exception for a given error case
- Write code that appropriately transforms exceptions by chaining or translating exceptions given a scenario (new LO)
    - Explain how and when to chain (or wrap) an exception to hide implementation details from the caller (combined LOs)
    - Explain how and when to translate exceptions to intentionally drop details of original cause of exceptions (combined LOs)
    - Explain why making custom exceptions can be helpful in the debugging process

Slide Number	Visual Content/Text/Assets	Text	Notes
1		So far, you've encountered exceptions thrown by code you were calling. You've had to determine how to proceed when an exception has been thrown. As you write more complex code, *you* will be in charge of deciding what is an error condition and when to throw an exception.	
2		All exceptions in Java extend java.lang.Exception. Any class that extends Exception (or a subclass) can be handled and thrown just like one of the built-in exceptions. Any new class extending Exception can be used to respond to error conditions. There are many reasons why manipulating or customizing exceptions for your methods is helpful in the debugging process. • **Clarity:** Exceptions are situations where your method is not executing properly. Thus, the more specific you can make an exception, the easier it will be to track down what is going wrong in the code. Project-specific error codes can help trace an error back to a specific API or feature set. An API is an integration with a service. You'll learn more about using APIs later in the curriculum. • **Detail:** Using a custom exception can allow you to include additional detail or data fields in the exception itself about what caused an error, which could also help in resolving the issue.	

		• **Functionality:** Custom exceptions can include utility methods to manipulate specific data formats and assist in debugging. • **Organization:** API-specific exceptions can help you tell exactly where the exception is coming from. • **Hide Implementation:** In rare cases, you may wish to use a custom Exception to obscure the exact file causing your method to throw an exception. You would want to avoid using an exception type that would indicate that you're accessing the file system directly, as this might pose a security risk by revealing more implementation details than necessary.	
3	**Diving Deeper - Detecting Errors**	For example, imagine a service that manages gym memberships. Every membership ID follows the pattern last name followed by 5 digits. If the service receives a request for ID ombrellaro123, you already know that won't work, since it only has 3 digits at the end. Instead of doing any more processing, you can code the program to throw a new IllegalArgumentException. This way, you know at the earliest point where something has gone wrong, which makes it easier to debug the root cause (a bad input). Imagine instead you tried to process the bad ID. Something else in the code would fail eventually, but it could be very far away (logically) from the input to the service. Connecting the dots back to that bad input could be difficult.	
4	**Transforming Exceptions**	You may find yourself in a situation where it will be useful to catch one exception and throw a different exception type. The IllegalArgumentException above may not be the exception you wish to convey – instead, it might be better to throw an InvalidMemberIDException.	

		This often means wrapping the original exception inside a new one. This could be to hide an implementation, as described in the previous section. It might also be to provide clarity, detail, functionality, or organization in a situation that the error case is detected by catching an exception, rather than testing for an error condition. Exception messages may need to contain more or different information than the exception you're catching. Or, your application may need to be more secure, so you may need to monitor and adjust what information your exceptions convey. There are two strategies to accomplish this: **chaining exceptions** and **translating exceptions**.	
5	**Chaining Exceptions**	Chaining exceptions, or wrapping exceptions, means using one exception to relate to another exception. This is commonly described as wrapping one exception in the "cause" field of a new exception. That means the exception is actually passed in as an additional constructor argument. For example, consider an exception where a method throws an ArithmeticException because of an attempt to divide by zero. However, the actual cause of the error is an I/O error which caused the divisor to be zero. The code only throws the Arithmetic Exception – so the developer does not know about the actual *cause* of the exception. Chained exceptions are a great solution here. Wrapping exceptions like this allows each exception to retain its own stack trace. That allows a developer to see where the exception originated. If an error might cause issues in multiple layers of the program, chaining	

		exceptions helps locate where the error began, as well as trace its path through the code, even when it’s wrapped. Chaining exceptions can preserve or even add information to exceptions.	

6	`public void setBirthday(int year, int month, int day) throws InvalidBirthdayException {`
	`    try {`
	`        LocalDate birthday = LocalDate.of(year, month, day);`
	`    } catch (DateTimeException e) {`
	`        throw new InvalidBirthdayException(String.format("One of the Date of Birth components is invalid. Year: " + year + " Month: " + month + " Day: " + day), e);`
	`    }`
	`}`
	
	`public class InvalidBirthdayException extends Exception {`
	
	`    public InvalidBirthdayException(String message, Throwable cause) {`
	`        super(message, cause);`
	`    }`
	`}`
		Here’s a simple example. In this code, the setBirthday() method takes in 3 ints specifying the date of birth and attempts to construct a LocalDate object. There may be several reasons why this may fail, for example, if one of the date of birth arguments is negative. The method above catches a DateTimeException and throws a new custom exception called InvalidBirthdayException.
		
		InvalidBirthdayException is defined below.
		
		The InvalidBirthdayException takes in both a String, message, and a Throwable, cause. Throwable is a superclass of Exception that implements the Serializable interface (we'll introduce serialization in the a later reading). Any exception you pass into InvalidBirthdayException will be a subclass of Throwable, so this is a safe way to accept previously caught exceptions. It also corresponds to the relevant constructor in the superclass, Exception, which is a good model to follow when creating your custom exception class constructors. In the setBirthDay() method, the original caught DateTimeException is passed in to InvalidBirthdayException's constructor to provide details on where exactly the parsing failed.	

7	**Translating Exceptions**	While chained exceptions preserve the cause of the exception, translating exceptions drop the original cause of the exception. Wanting to do this is rare, because dropping the cause means losing the original stack trace. Where chaining exceptions preserves information, translating exceptions intentionally loses information.	
8		The most common reason to do this is if the original exception might expose specific, security-sensitive information, perhaps even a security hole, and you're returning the exception to a code base you don't own. This may happen with database exceptions and file IO exceptions, by revealing the specific storage systems or versions being used – especially if these have known vulnerabilities. Since translating the exception loses all previous stack trace information, it is important to log enough information to know where the translated exception was thrown from, so that the developer can track down the error.	

9	`public class ItemAccessUtilty {`
	`    private Logger logger;`
	
	`    // Constructor and other methods omitted`
	
	`    /**`
	`     * Retrieves a record from some unspecified datastore.`
	`     * @throws RecordAccessFailedException when attempt to access data fails`
	`     */`
		This is an example of a translated exception. The method getRecord() catches the original AmazonDynamoDbException, an exception thrown from accessing a database. However, imagine in this case that there are business and security reasons not to expose the fact that an Amazon database is being used. The message from the AmazonDynamoDbException is logged, then a custom RecordAccessNotFoundException is thrown with a message that gives the calling routine a chance to handle the exception but does not disclose the details of the original exception for security reasons.	

<table>
<tr><td></td><td><pre>   public Record getRecord() throws RecordAccessFailedException {

        //

        try {

            Record record = getItemFromDynamoDbDatabaseAndConvertToRecord();

        } catch (AmazonDynamoDbException e) {

            logger.log(Level.INFO, e.getMessage(), e);

            throw new RecordAccessFailedException ("The record could not be accessed.");

        }

    return record;

    }

}</pre></td><td></td><td></td></tr>
<tr><td>10</td><td>Knowledge Check</td><td></td><td></td></tr>
</table>

Activity 1: Transforming Exceptions

Est. Learner Time: 30 min

Activity Explanation: NEW ACTIVITY. Can we build something that practices transforming exceptions? Both chaining and translating? This should be pretty low-Blooms, we should tell them to “transform the following exception using X info” I think.

- TLO: Write code that appropriately transforms exceptions by chaining or translating exceptions given a scenario (new LO)

Instructional Lesson 2: Designing Custom Exceptions

Review Link:

Est. Learner Time: 30 min

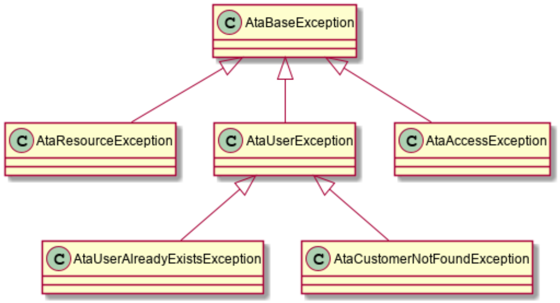
TLO:

- Create custom exceptions
  - Design and implement an exception class hierarchy for a code base
  - Understand that it is a good practice to provide an Exception subclass with the same public constructor signatures as the base java.lang.Exception class
  - Design and implement an exception that describes an error case specific to a service
  - Produce a unique serialVersionUID when creating a new exception type
  - Outline the public constructors of the Exception class
- Define error cases
  - Explain how to define error cases given a set of requirements

Slide Number	Visual Content/Text/Assets	Text	Notes
1	**Defining Exceptions**	In the a prior lesson, you modified exceptions that already exist. In this lesson, you will design your own. When designing a class or method, it is important to look at how it can fail as well as how it handles the happy path. Ask the following questions: 1. What can possibly go wrong? What are the edge cases that you might need to check or handle? Include possible scenarios that might make your method fail, and how your method should respond. 2. Should the method handle or throw an exception? 3. If the method should throw an exception: a. Is there an existing exception that covers the situation? If so, use it! b. Does it make sense to add any additional details to the exception? You may want to create a custom exception. 4. Plan to include enough information in the exception via the exception type, exception message, additional fields, or the wrapped original cause exception for another developer to diagnose the error condition if it occurs.	
2		Custom exceptions are useful in Java because they allow a developer to add methods or attributes that are not part of the standard Java exceptions. These	

		might include specific ways of handling an error or provide specific error messages.	
3	**Naming the Exception**	Exceptions should be named clearly and to help you and other developers interpret what is going on quickly. The format name SomethingException, including Exception at the end of the name, is a widespread convention. For example, InvalidBirthdayException from the previous lesson identifies the problem (InvalidBirthdate) and the fact that the object is an Exception.	
4	Show same chart used in Exception Handling lesson that shows Exception Hierarchy	Recall that Exceptions all extend the base Exception class. Java has two types of Exceptions, Checked Exceptions and Unchecked Exceptions. When building your own exception, you must determine where it fits in the exception hierarchy. Remember that Checked exceptions must be explicitly handled or propagated, as enforced by the Java compiler. Checked exceptions are included as part of a method's call signature, informing other users which checked exceptions must be handled. When designing your own exceptions, it makes sense to use a checked exception if there's a good way to manage or recover from the error in the code. Otherwise, why enforce the handling of the error if there's nothing that can be done with a catch block? One of the most common use cases for checked exceptions is to inform a caller their request is invalid. These are often associated with 4xx response codes. Unchecked exceptions do not explicitly require handling or propagation, and are propagated by default. Because unchecked exceptions can occur	

		unexpectedly, there’s no need to include them in the method signature. The primary downside of unchecked exceptions is that our caller is far less aware of which unchecked exceptions our code might throw, and is not forced to handle them when they occur. As such, unchecked exceptions are less useful than checked exceptions if we expect our caller to manage or recover from an error. Only use unchecked exceptions when encountering an issue that the caller can’t recover from, or when the unexpected occurs. Here are common cases for throwing unchecked exceptions: • Programming errors, like indexing beyond the size of an array or accessing data from a null pointer • Attempting to access an unavailable resource or dependency • Arithmetic errors, like attempting to divide by 0. Note that these are issues that are difficult to recover from. The request the caller made is valid, but an unexpected error occurred within the code or service when the request was handled. Always consider whether an existing exception class is available before creating your own. See if a standard Java RuntimeException subclass is suitable for the use case before writing a custom unchecked exception.	

		Follow any conventions or standards of the development team when developing custom exceptions. Some teams at Amazon write custom checked exceptions that inherit from a common service-specific base class. In a larger programming project, there might be many services interacting with one another. Each service might have its own base Exception subclass, which may in turn inherit from a global base exception.	
5	 AtaBaseException AtaResourceException AtaUserException AtaAccessException AtaUserAlreadyExistsException AtaCustomerNotFoundException	It's best to name exceptions so they are consistent with the code they reside in. This code resides in the AtaCreatingExceptions package. That uses a base Exception subclass for the package called AtaBaseException. This will be used in the activity for this lesson as well. AtaBaseException is a checked exception. Therefore, all of the subclasses are checked exceptions. Any issue must be handled so that the program can continue. Within this package is a client library, used to call information about users and resources they access. Imagine building a service that requires customers to login in before using its features. This diagram illustrates one possible hierarchy. AtaBaseException has 3 subclass exceptions. Each represents types of error conditions: • AtaUserException – This is thrown if a user unexpectedly exists or unexpectedly doesn't exist. • AtaCustomerNotFoundException – This is thrown if the user does not appear to exist in the system (maybe the data was entered inaccurately).	I've gotten all this from the current reading. However, our learners are not currently trained on DynamoDB – we get that in the next phase. I think we can modify this exception class to create something not built as a DB, but it might be a pain, so please modify if this does not work.

		• AtaUserAlreadyExistsException – This is thrown when a duplicate user exists but the system is trying to create a new one. • AtaResourceException – This is thrown if an error associated with a resource (for example, a file) occurs. • AtaAccessException – This is thrown if the user doesn't have privileges for the requested operation. Each user can access their own data, but not all users can access someone else's data.	

5	serialVersionUID
	`public class AtaCustomerNotFoundException extends AtaUserException {`
	`    private static final long serialVersionUID = 1952705374572855798L;`
	`    private String username;`
		Every custom exception will include a private static final long constant, serialVersionUID. This is because the Java Exception class implements the interface Serializable. This is essentially a specific number created just for the current exception class's design. It is used when exceptions are passed back from server to client. This number ensures all codebases have compatible versions of the class. You'll learn more about Serializable later, but it's important to use here when declaring custom exceptions.
		When you declare the custom exception, it looks something like this:
		Don't worry, you won't have to create a crazy long number yourself. You do need to declare the serialVersionUID explicitly, but IntelliJ can calculate the parameter value for you. To do this, go to IntelliJ's Preferences, then Editor, Inspections, Java, and Serialization Issues. There should be an option for "Serializable class without serialVersionUID." There you will find the option to check a box and click	

		“OK.” This will make IntelliJ tell you when a Serializable class is missing the serialVersionUID field, and it will allow you to generate the field in the IDE’s interface.	

6	`package com.amazon.ata.creatingexceptions.prework;`
	
	`public class AtaCustomerNotFoundException extends AtaUserException {`
	
	`    private static final long serialVersionUID = 1952705374572855798L;`
	`    private String username;`
	
	`    /**`
	`     * Constructs exception with username.`
	`     * @param username - username representing customer ID`
	`     */`
	`    public AtaCustomerNotFoundException(String username) {`
	`        super("User with " + username + " cannot be found.");`
	`        this.username = username;`
	`    }`
	
	`    /**`
		As a developer, you are working on a customer lookup method using username as the search parameter. You have identified a common error scenario: the customer not found in the existing data. When this occurs, there are two unrecoverable errors that could occur:
		• A method could return a null string that a calling method might not support
		• The search engine might keep looking for a customer that is never found.
		
		Several recovery options exist for this situation. One is to have the software prompt the customer to create a new account if it doesn’t exist. Another is to display an error message requesting the customer re-enter the account ID. In either case, the developer worked with the product manager and determined that this is not the job of the lookup algorithm. Instead, the program should throw a AtaCustomerNotFoundException to inform the calling method that a customer-facing recovery action should occur. The customer facing user interface then calls it’s own customer lookup and can handle it from there.
		
		Custom Exceptions can contain additional fields as needed. This one includes the customer ID field, username.
			Code came from the current lesson. Please modify if this is overtly service-y. I don’t see it but I may be missing something.
			
			The scenario, however, is API-y. I have modified it to remove the API bits but it sounds a little weird now. I’m open to something better.

```
     * Constructs exception with username, message and cause.

     * @param username - username representing customer ID

     * @param message - Description of the error encountered, in this case the requested customer could not be found.

     * @param cause - The Exception that caused this exception to be thrown. Used in Exception chaining.

     */

    public AtaCustomerNotFoundException(String username, String message, Throwable cause) {

        super("Username " + username + " cannot be found. " + message, cause);

        this.username = username;

    }


    /**

     * Constructs exception with username and message.

     * @param username - username representing customer ID

     * @param message - Description of the error encountered, in this case the requested customer could not be found.

     */

    public AtaCustomerNotFoundException(String username, String message) {
        super("Username " + username + " cannot be found. " + message);
        this.username = username;
    }


    /**
     * Constructs exception with username and cause.
     * @param username - username representing customer ID
     * @param cause - The Exception that caused this exception to be thrown. Used in Exception chaining.
     */
    public AtaCustomerNotFoundException(String username, Throwable cause) {
        super("User with " + username + " cannot be found.", cause);
        this.username = username;
    }


    public String getUsername() {
        return username;
    }
}
```

7	**Constructors**	Exceptions typically announce that the error has occurred. They don't really handle any details about	

	Show picture of the code above – point these out in the code. I'm thinking we'll show this side by side with numbers indicating the places in the image where these items are defined.	the error – that's done in the catch block. However, the Exception class does need to define constructors. Define as many of the following as are relevant to your custom exception. Compare this to the AtaCustomerNotFoundException: 1. Exception() – this constructs a new exception without a cause. It does not provide a message, so it's not that helpful on its own. The AtaCustomerNotFoundException includes AtaCustomerNotFoundException(String username), which includes the class field and username. This does not accept a message or cause, but identifies the exception. 2. Exception (String message) – This constructs a new exception with a specified detailed message. AtaCustomerNotFoundException has AtaCustomerNotFoundException(String username, String message). This accepts a username and a message. 3. Exception (Throwable cause) – This constructs a new exception with the specified cause, but the default message is null. In AtaCustomerNotFoundException, you see AtaCustomerNotFoundException(String username, Throwable cause). This accepts only a username and a cause. 4. Exception (String message, Throwable cause) – This constructs a new exception with the specified message detail and cause. AtaCustomerNotFoundException offers AtaCustomerNotFoundException(String username, String message, Throwable cause).	

		This accepts all the standard Exception arguments, along with the custom username field. Exception classes may have simple getters, but it is rare for them to have any other methods. This is because handling the exception is done by the code using the exception rather than the exception class itself.	

8	**Declaring the New Exception**
	@Test
	public void searchUser_customerNotFound_exceptionThrownWithExpectedMessage() {
	    try {
	        customerNotFound();
	    } catch (AtaCustomerNotFoundException e) {
	        Assertions.assertEquals("The customer was not found.", e.getMessage(), "Wrong Exception message");
	    }
	    // code not shown that would cause the test to fail if the Exception didn’t get thrown
	}
	/**
	 * Simple method to ensure a AtaCustomerNotFoundException can be thrown.
		The customerNotFound() method tries out the new exception. This declaration throws AtaCustomerNotFoundException. The JUnit test then asserts the message contents, and uses a try catch block to verify the Exception message.	

	* @throws AtaCustomerNotFoundException - Stores the username ID
	* and informs the caller ID that it wasn't found with associated message.
	*/
	public void customerNotFound() throws AtaCustomerNotFoundException {
	    throw new AtaCustomerNotFoundException("badusername", "The customer was not found.");
	}		

9	Knowledge Check	Questions 3 & 4	
10	What’s Next?	In this lesson, you learned to create your own custom exceptions. Return to the LMS to practice creating your own exceptions in code!	

Activity 2: Creating Exceptions

Est. Learner Time: 15 min

Activity Explanation: MIGRATE EXISTING ACTIVITY. If this will work – and I think it will. It doesn't seem to be service heavy.

- TLO:
    - Create custom exceptions
    - Define error cases
- Instructions: https://code.amazon.com/packages/ATAClassroomSnippets_U3/blobs/heads/C2021Aug/--/src/main/java/com/amazon/ata/creatingexceptions/prework/README.md
- Code Snippets: https://code.amazon.com/packages/ATAClassroomSnippets_U3/trees/heads/C2021Aug/--/src/main/java/com/amazon/ata/creatingexceptions/prework

Activity 3: TBD

Est. Learner Time: 60 min

Activity Explanation: TBD.

- TLO:
    - Examine whether to throw an existing exception or implement a new exception for a given error case
    - Create custom exceptions
    - Write code that appropriately transforms exceptions by chaining or translating exceptions given a scenario

Lesson Wrap-Up:

- *Est. Learner Time: 30 min*
- Questions: (link or actual questions)

- Summary Prose: Errors in code are inevitable. Developers must determine the best way to recover from errors. Using and handling existing exceptions properly provides a foundation that works for many errors. However, developers must often transform errors to provide a more useful message or recovery path. In addition, there are cases where creating a customized exception makes sense. A seasoned developer must examine code to determine whether existing exceptions provide a path forward, or whether it’s better to transform or customize an exception for special circumstances. Regardless, understanding exceptions and handling them appropriately in your code will help you create more flexible and useful programs less prone to failure.