Tutorial for exception handling

In this tutorial, you'll create a custom exception, and use it in validating user input.

Exceptions and exception handling are more than ways of handling unexpected errors in your application. A custom exception and a well-placed `catch` block can lead to code code which separates detecting problems from telling users about them.

To get started, import this project into IntelliJ.

Step One: Review the existing code

There are two class files provided: `CreditCard.java` and `Program.java`.

`CreditCard.java`

The `CreditCard` class represents a generic credit card.

It contains four fields:

- `lastName` - The last name of the cardholder.
- `firstName` - The first name of the cardholder.
- `number` - The card number.
- `securityCode` - The card security code.

The user enters values for each of the properties at runtime.

Along with the usual getters and setters, there are also three methods in the class:

- `validate()` - This method validates the field values entered by the user. Currently, the method only contains comments listing the validations that you'll write in this tutorial.
- `toString()` - The base `toString()` has been overridden to build a well-formatted string of the `CreditCard` class properties to display to the user.
- `isDigits(String str)` - This is a helper method which confirms that the `str` argument is all numeric or *digit* characters: `'0'`-`'9'`. It returns `true` if all the characters are digits, otherwise, it returns `false`.

`Program.java`

`Program.java` contains the `main(String[] args)` method and has no fields.

The `main()` method begins by creating an instance of a `CreditCard` object. It then starts the validation loop which runs until the user enters valid credit card information:

```
// Credit card validation loop
while (true) {
    // Prompt user for credit card information
    System.out.print("Last name: ");
    cc.setLastName(keyboard.nextLine());
    System.out.print("First name: ");
```

```
        cc.setFirstName(keyboard.nextLine());
        System.out.print("Number: ");
        cc.setNumber(keyboard.nextLine());
        System.out.print("Security code: ");
        cc.setSecurityCode(keyboard.nextLine());

        // Validate the credit card
        try {
            cc.validate();
            break; // No exception thrown, credit card is valid, break out of
    validation loop
        }
        catch (Exception ex) { // Step 3: Modify validation loop to catch
    CreditCardValidationException
            System.out.println("Card is invalid: " + ex.getMessage() + "\n");
        }
    }
```

The validation loop prompts the user to enter values for the `CreditCard` fields, `lastName`, `firstName`, `number`, and `securityCode`.

This information is then validated from inside a `try-catch`.

If the information is valid, meaning that an exception wasn't thrown, the `break` statement runs and the validation loop ends. Otherwise, an exception gets thrown and the validation error message is displayed. The validation loop continues from the top with prompting for the credit card information and validating it.

The validation loop only ends when the credit card information is valid. The valid `CreditCard` object displays the information, and the application ends.

Run the application, and enter whatever values you like when prompted. Since the `validate()` method has no validation code at this time, any values you enter are valid.

```
Last name: A
First name:
Number: 12345678901234567890123456789012345678901234567890
Security code: dog

Card is valid - Name: ' A', Number:
12345678901234567890123456789012345678901234567890, Security Code: dog
```

Step Two: Create user-defined `CreditCardValidationException` class

In order to signal a validation error, the `CreditCard.validate()` method needs to throw a `CreditCardValidationException` with a message detailing the validation error.

Create the `CreditCardValidationException` class, extend `Exception`, and provide the `CreditCardValidationException(string message)` constructor which accepts a message detailing the exception.

> By convention, custom exceptions always end with the word `Exception`.

```
public class CreditCardValidationException extends Exception {

    public CreditCardValidationException(String message) {
        super(message);
    }

}
```

Step Three: Update the validation loop to catch the `CreditCardValidationException`

Its always best practice to throw and catch the most specific exception you can.

Now that `CreditCardValidationException` is available, modify the `validate()` method in `CreditCard` to throw the more specific exception. Replace `throws Exception` with `throws CreditCardValidationException`.

```
// ...
public void validate() throws CreditCardValidationException { // Step 3: Throw and
catch CreditCardValidationException
    // ...
}
```

Then change the generic `catch (Exception ex)` in the validation loop to the specific `catch (CreditCardValidationException ex)` in the `main()` method.

```
// ...
catch (CreditCardValidationException ex) { // Step 3: Throw and catch
CreditCardValidationException
    // ...
}
// ...
```

Step Four: Validate cardholder name

The `firstName` and `lastName` fields can't be `null` or blank.

Add the following code in `CreditCard.java` after the **Step 4:** comment in the `validate()` method to validate the cardholder's first and last names:

```
// Step 4: Validate cardholder name
if (lastName == null || lastName.length() == 0 || firstName == null ||
firstName.length() == 0) {
```

```
    throw new CreditCardValidationException("'" + firstName + " " + lastName + "'
- Cardholder name is invalid, must provide first and last names.");
}
```

Run the application, and enter a last name and first name. The application shows the credit card is valid provided you enter values for first two prompts:

```
Last name: Testing
First name: Test
Number:
Security code:

Card is valid - Name: `Test Testing`, Number: , Security Code:
```

Step Five: Validate card number

The card number must be 13 or 16 characters in length, and all digits, `'0'`-`'9'`. Add this code after the **Step 5:** comment:

```
// Step 5: Validate card number
if (number == null || (number.length() != 13 && number.length() != 16) ||
!isDigits(number)) {
    throw new CreditCardValidationException("'" + number + "' - Card number is too
short or too long, or not all digits.");
}
```

Run the application, enter values for last name and first name, and a 13 or 16 digit numeric value for the Number. The application shows the credit card is valid if you enter values that fit the validation rules:

```
Last name: Testing
First name: Test
Number: 1234567890123456
Security code:

Card is valid - Name: 'Test Testing', Number: 1234567890123456, Security Code:
```

Step Six: Validate security code

The security code must be 3 characters in length, and all digits, `'0'` - `'9'`. Add this code after the **Step 6:** comment:

```
// Step 6: Validate security code
if (securityCode == null || securityCode.length() != 3 || !isDigits(securityCode))
{
```

```
    throw new CreditCardValidationException("'" + securityCode + "' - Security
code is too short or too long, or not all digits.");
}
```

Run the application, enter values for last name and first name, a 13 or 16 digit numeric value for the Number, and a 3 digit value for the security code. The application shows the credit card is valid if you enter values that fit the validation rules:

```
Last name: Testing
First name: Test
Number: 1234567890123456
Security code: 123

Card is valid - Name: 'Test Testing', Number: 1234567890123456, Security Code: 123
```

Step Seven: Create the `MasterCard` class and replace `CreditCard` instance

The `MasterCard` class extends `CreditCard` and overrides the `validate()` method.

In addition to the validation already performed by the base `validate()` method, the override checks that the card number begins with a `'5'`.

```
class MasterCard extends CreditCard {

    @Override
    public void validate() throws CreditCardValidationException {
        super.validate();
        // MasterCard numbers always begin with '5'.
        if (getNumber().charAt(0) != '5') {
            throw new CreditCardValidationException("'" + getNumber() + "' -
Invalid MasterCard card number, must begin with '5'.");
        }
    }

}
```

Once the `MasterCard` class is complete, replace the `CreditCard` instance with `MasterCard` in the `main()` method.

```
CreditCard cc = new MasterCard();
```

> Note: Only replace the `new CreditCard()`. The variable holding the instance of new `MasterCard` object remains of type `CreditCard`.

Run the application, enter the last and first names, a 13 or 16 digit numeric value with the first character as `'5'` for the Number, and a 3 digit value for the security code. The application shows the credit card is valid if you enter values that fit the validation rules:

```
Last name: Testing
First name: Test
Number: 5234567890123456
Security code: 123

Card is valid - Name: 'Test Testing', Number: 5234567890123456, Security Code: 123
```

Step Eight: Create the `Visa` class and replace `CreditCard` instance

The `Visa` class extends `CreditCard` and overrides the `validate()` method.

In addition to the validation already performed by the base `validate()` method, the override checks that the card number begins with a `'4'`.

```
public class Visa extends CreditCard {

    @Override
    public void validate() throws CreditCardValidationException {
        super.validate();
        // Visa numbers always begin with '4'.
        if (getNumber().charAt(0) != '4') {
            throw new CreditCardValidationException("'" + getNumber() + "' -
Invalid Visa card number, must begin with '4'.");
        }
    }

}
```

Once the `Visa` class is complete, replace the `MasterCard` instance with `Visa` in the `main()` method.

```
CreditCard cc = new Visa();
```

> Note: Only replace the `new MasterCard()`. The variable holding the instance of new `Visa` object remains of type `CreditCard`.

Run the application, enter the last and first names, a 13 or 16 digit numeric value with the first character as `'4'` for the Number, and a 3 digit value for the security code. The application shows the credit card is valid if you enter values that fit the validation rules:

```
Last name: Testing
First name: Test
```

```
Number: 4234567890123456
Security code: 123

Card is valid - Name: 'Test Testing', Number: 4234567890123456, Security Code: 123
```

The validation for all the supported credit cards is now complete.

Once the `CreditCardValidationException` class was created, and the `catch` inside the validation loop was changed from the generic `Exception` to the more specific `CreditCardValidationException`, the validation loop remained unchanged.

The validation rules are encapsulated in their respective classes, `CreditCard`, `MasterCard`, and `Visa`, while the user-defined exception `CreditCardValidationException` serves as a clean separation between the code that knows the validation rules from the code in the validation loop. The `CreditCardValidationException` is thrown by the rules and caught by the application UI.

Exceptions and exception handling isn't limited to smoothing the occasional bump in your application's path. With some planning, and a little code, exceptions can help you write code that keeps separate concerns separate.

Next steps

Long strings of digits like the value of the `number` field tend to be broken up into segments, for instance, `"0000-0000-0000-0000"`. Modify the `number` validation rules to accept hyphens, `'-'` while preserving the existing rules of a length of 13 or 16 digits, `'0'` - `'9'`.

Inspecting existing code is a great way to learn to program. In this tutorial, the method `isDigits()` was already there for you when you started. The method checks that the `String` argument contains only digit characters, and returns `true` is this is so, otherwise, it returns `false`.

It uses a foreach to loop through the character array of the `String` using the `toCharArray()` method and checking that each character falls within the range of `'0'` - `'9'`. Examine this method until you understand how it works.