4 Validation et mise en forme

Nous aurons tôt ou tard besoin de traiter des données fournies par l'utilisateur, cependant il sera nécessaire de vérifier qu'elles respectent un format précis. Ce chapitre introduit la validation de formulaire et le formatage des données (classes `Validator` et `Formatter` du framework Flex).

L'objet `Validator` est un diffuseur d'événement qui contrôle un champ afin de s'assurer qu'il respecte le format requis. Le framework fourni des validateurs prédéfinis (dates, cartes de crédit, emails, …)

Flex Language Reference, package **mx.validators** :

```
http://help.adobe.com/fr_FR/AS3LCR/Flex_4.0/mx/validators/package-
detail.html
```

L'objet `Formatter` quant à lui a pour but de mettre en forme une valeur afin qu'elle soit présentée dans un format spécifique. Il existe également des formatteurs prédéfinis dans le framework.

Flex Language Reference, package **mx.formatters** :

```
http://help.adobe.com/fr_FR/AS3LCR/Flex_4.0/mx/formatters/package-
detail.html
```

4.1 DateValidator

Dans un premier temps, nous allons mettre en œuvre le validateur de date `DateValidator`. L'objectif est de vérifier que le format de saisie de deux dates est `DD.MM.YYYY`. Pour ce faire, nous allons procéder de plusieurs manières :

4.1.1 Validateur pour chaque date

Cette première manière de faire présente l'utilisation d'un validateur par champ à vérifier.

```
<fx:Declarations>
    <mx:DateValidator
        id="date1Validator"
        source="{inputDate1}"
        property="text"
        inputFormat="DD.MM.YYYY"/>
    <mx:DateValidator
        id="date2Validator"
        source="{inputDate2}"
        property="text"
        inputFormat="DD.MM.YYYY"/>
</fx:Declarations>
```

Cela signifie que chaque validateur s'occupe de contrôler son champ date. Les propriétés `source` et `property` permettent de définir la propriété du composant à valider. En l'occurrence, les validateurs s'intéressent à la propriété `text` des deux composants `TextInput`.

Ajoutons ensuite deux `TextInput` à l'interface avec chacun leur écouteur d'événement lorsqu'ils perdent le focus. En effet, c'est à cet instant qu'il faudra vérifier si la date est valide.

```
<!-- Date 1 -->
<s:HGroup
    verticalAlign="middle">
    <s:Label
        text="Date Input 1"/>
    <s:TextInput
        id="inputDate1"
        focusOut="date1Validate(event)"
        width="100"/>
</s:HGroup>

<!-- Date 2 -->
<s:HGroup
    verticalAlign="middle">
    <s:Label
        text="Date Input 2"/>
    <s:TextInput
        id="inputDate2"
        focusOut="date2Validate(event)"
        width="100"/>
</s:HGroup>
```

Event handler

```
private function date1Validate(event : FocusEvent):void
{
    vResult = date1Validator.validate();
    if (vResult.type == ValidationResultEvent.VALID)
        trace("Valid date");
    else
        trace("Invalid date");
}

private function date2Validate(event : FocusEvent):void
{
    vResult = date2Validator.validate();
    if (vResult.type == ValidationResultEvent.VALID)
        trace("Valid date");
    else
        trace("Invalid date");
}
```

Comme nous pouvons le constater, `date1Validator` et `date2Validator` jouent exactement le même rôle, c'est pourquoi il pourrait être intéressant de n'utiliser qu'un validateur pour contrôler la validité des deux dates.

4.1.2 Validateur partagé pour toutes les dates

Cette méthode présente la seconde manière de faire permettant de valider plusieurs dates à l'aide d'un seul et unique validateur. Tout d'abord, créer un `DateValidator` sans spécifier les propriétés `source` et `property`.

```
<fx:Declarations>
    <mx:DateValidator
        id="dateValidator"
        inputFormat="DD.MM.YYYY"/>
</fx:Declarations>
```

Ajoutons ensuite deux TextInput à l'interface avec un écouteur d'événement lorsqu'ils perdent le focus.

```
<s:TextInput
    id="inputDate1"
    focusOut="dateValidate(event)"
    width="100"/>
…
<s:TextInput
    id="inputDate2"
    focusOut="dateValidate(event)"
    width="100"/>
```

Pour terminer, la méthode `dateValidate()` permet de démarrer la validation du champ qui vient de perdre le focus.

```
private function dateValidate(event : FocusEvent):void
{
    dateValidator.listener = event.currentTarget;
    vResult = dateValidator.validate(event.currentTarget.text);

    if (vResult.type == ValidationResultEvent.VALID)
        trace("Valid date");
    else
        trace("Invalid date");
}
```

Etant donné que nous utilisons un validateur pour plusieurs champs, il est nécessaire de définir le composant sur lequel les éventuelles erreurs de validation seront affichées (`dateValidator.listener`).

Il est possible de personnaliser les messages d'erreurs en définissant les propriétés suivantes :

```
invalidCharError="The date contains invalid characters."
wrongDayError="Enter a valid day for the month."
wrongLengthError="Type the date in the format inputFormat."
wrongMonthError="Enter a month between 1 and 12."
wrongYearError="Enter a year between 0 and 9999."
```

4.2 DateFormatter

Nous allons maintenant mettre en oeuvre le formateur de date. Pour ce faire commençons par déclarer un objet DateFormatter.

```
<fx:Declarations>
    <mx:DateFormatter
        id="dateFormatter"/>
</fx:Declarations>
```

Ensuite, définir le format d'affichage désiré en utilisant les « lettres de modèle » provenants de la documentation.

```
private function onComplete():void{
    var today:Date = new Date();

    dateFormatter.formatString = "DD.MM.YYYY";
    lblDateT_1.text = dateFormatter.formatString;
    lblDate_1.text = dateFormatter.format(today);

    dateFormatter.formatString = "EEEE D MMMM YYYY";
    lblDateT_2.text = dateFormatter.formatString;
    lblDate_2.text = dateFormatter.format(today);
}
```

L'affichage des dates s'effectue dans les labels suivants :

```
<mx:HBox>
    <s:Label id="lblDateT_1"/>
    <s:Label id="lblDate_1"/>
</mx:HBox>

<mx:HBox>
    <s:Label id="lblDateT_2"/>
    <s:Label id="lblDate_2"/>
</mx:HBox>
```

Flex Language Reference, Class **DateFormatter** :

```
http://help.adobe.com/fr_FR/AS3LCR/Flex_4.0/mx/formatters/DateFormatter.h
tml
```

4.3 Custom Validator

Nous allons créer un validateur personnalisé qui va s'occuper de vérifier que le format d'une String correspond à un « Language Code » (fr_FR, en_US, …). La vérification s'effectuera à l'aide d'une expression régulière.

Pour commencer, créer un fichier `LanguageCodeValidator.as` qui étend la classe `mx.validators.Validator`. Ce validateur définit une expression régulière définissant la syntaxe d'un code de language.

```
public class LanguageCodeValidator extends Validator
{
    private var languageCode:RegExp = /^[a-z]{2}([_-][A-Z]{2})?$/;
    …
```

Il est nécessaire de surcharger la méthode `doValidation()` afin d'effectuer le traitement de validation. Cette méthode doit retourner un tableau d'objets `ValidationResult`.

Méthode surchargée doValidation()

```
override protected function doValidation(value:Object):Array
{
    // Initialise le tableau de résultats
    var results:Array = results = [];

    if (languageCode.test(value as String))
        return results;
    else
    {
        // Ajout d'une erreur de validation
        var err:ValidationResult = new ValidationResult(true, "",
                "", "Please enter a correct Country Code");
        results.push(err);
        return results;
    }
}
```

Si la valeur passée en paramètre à la méthode `doValidation()` ne correspond pas à l'expression régulière, un nouvel objet `ValidationResult` est créé. La vérification de la correspondance s'effectue par la méthode `test()` de notre objet `RegExp`.

L'outil « *Flex 3 Regular Expression Explorer* » propose un grand nombre d'exemples d'expressions régulières et permet notamment de tester nos propres expressions.

```
http://ryanswanson.com/regexp/#start
```

4.4 Custom Formatter

Nous allons créer ici un formateur personnalisé qui va accepter une chaîne de caractère afin de la retourner dans un format spécifique. Il permettra de mettre en forme des numéros ISBN (10 ou 13 chiffres). Pour commencer, créer un fichier `ISBNFormatter.as` qui étend la classe `mx.formatters.Formatter`, et qui déclare une variable `formatString`.

```
public class ISBNFormatter extends Formatter
{
    public var formatString : String;// = "####-##-####";
    …
```

Il est nécessaire de surcharger la méthode `format()` afin d'effectuer notre traitement de mise en forme. Ceci se fait en plusieurs étapes :

- Vérification de la longueur de la String à mettre en forme
- Vérification du nombre de # dans le formatString
- Formattage du contenu à l'aide de l'utilitaire `SwitchSymbolFormatter`

Méthode format()

```
override public function format(value:Object):String
{
    // Vérifie la longueur de la string ISBN (10 ou 13 caractères)
    if (!(value.toString().length == 10 ||
          value.toString().length == 13))
    {
        error = "Invalid String Length";
        return "";
    }
    // Vérifie le nombre de # dans la formatString (10 ou 13)
    var numCharCnt:int = 0;
    for (var i:int = 0; i<formatString.length; i++)
    {
        if ( formatString.charAt(i) == "#")
            numCharCnt++;
    }
    if (!(numCharCnt == 10 || numCharCnt == 13))
    {
        error = "Invalid Format String";
        return "";
    }
    // Si la valeur et le formatString  sont valides, formatter le contenu
    var dataFormatter:SwitchSymbolFormatter = new SwitchSymbolFormatter();
    return dataFormatter.formatValue(formatString, value);
}
```

La classe utilitaire SwitchSymbolFormatter est très utilisée lors de la création de formateurs personnalisés. Elle permet de substituer les caractères réservés d'une chaine par des nombres issus d'une autre chaîne. Par exemple, l'appel de `dataFormatter.formatValue("#-#","12")` retourne une String `"1-2"`.

4.5 Exercice

Créer un validateur personnalisé permettant de vérifier qu'un groupe de checkbox contient entre *min* et *max* éléments cochés.