2.1.1. Constraint definition properties

A constraint definition may have attributes that are specified at the time the constraint is applied to a JavaBean. The properties are mapped as annotation elements. The annotation element names `message`, `groups` and `payload` are considered reserved names; annotation elements starting with `valid` are not allowed; a constraint may use any other element name for its attributes.

2.1.1.1. message

Every constraint annotation must define a `message` element of type `String`.

```
String message() default
"{com.acme.constraint.MyConstraint.message}";
```

The `message` element value is used to create the error message. See Section 4.3, "Message interpolation" for a detailed explanation. It is recommended to default `message` values to resource bundle keys to enable internationalization. It is also recommended to use the following convention: the resource bundle key should be the fully qualified class name of the constraint annotation concatenated to `.message` as shown in the previous program listing.
Built-in Bean Validation constraints follow this convention.

2.1.1.2. groups

Every constraint annotation must define a `groups` element that specifies the processing groups with which the constraint declaration is associated.

```
    Class<?>[] groups() default {};
```

The default value must be an empty array.
If no group is specified when declaring the constraint on an element, the `Default` group is considered declared.
See Section 4.1.2, "groups" for more information.
Groups are typically used to control the order in which constraints are evaluated, or to perform validation of the partial state of a JavaBean.

2.1.1.3. payload

Constraint annotations must define a `payload` element that specifies the payload with which the the constraint declaration is associated.

```
    Class<? extends Payload>[] payload() default {};
```

The default value must be an empty array.
Each attachable payload extends `Payload`.

```
/**
 * Payload type that can be attached to a given
 * constraint declaration.
 * Payloads are typically used to carry on metadata
information
 * consumed by a validation client.
 *
 * Use of payloads is not considered portable.
 *
 * @author Emmanuel Bernard
 * @author Gerhard Petracek
 */
public interface Payload {
}
```

Payloads are typically used by validation clients to associate some metadata information with a given constraint declaration. Payloads are typically non-portable. Describing payloads as interface extensions as opposed to a string-based approach allows an easier and more type-safe approach.

One use case for payload shown in Example 2.1, “Use of payload to associate severity to a constraint” is to associate a severity to a constraint. This severity can be exploited by a presentation framework to adjust how a constraint failure is displayed.

Example 2.1. Use of payload to associate severity to a constraint

```
package com.acme.severity;

public class Severity {
    public static class Info implements Payload {};
    public static class Error implements Payload {};
}

public class Address {
    @NotNull(message="would be nice if we had one",
payload=Severity.Info.class)
    public String getZipCode() {...}

    @NotNull(message="the city is mandatory",
payload=Severity.Error.class)
    String getCity() {...}
}
```

The `payload` information can be retrieved from error reports via the

`ConstraintDescriptor` either accessed through the `ConstraintViolation` objects (see Section 4.2, “ConstraintViolation”) or through the metadata API (see Section 5.5, “ConstraintDescriptor”).

2.1.1.4. Constraint specific parameter

The constraint annotation definitions may define additional elements to parameterize the constraint. For example, a constraint that validates the length of a string can use an annotation element named `length` to specify the maximum length at the time the constraint is declared.

2.1.2. Examples

Example 2.2. Simple constraint definition

```
package com.acme.constraint;

/**
 * Mark a String as representing a well formed order
number
 */
@Documented
@Constraint(validatedBy = OrderNumberValidator.class)
@Target({ METHOD, FIELD, ANNOTATION_TYPE,
CONSTRUCTOR, PARAMETER })
@Retention(RUNTIME)
public @interface OrderNumber {
    String message() default
"{com.acme.constraint.OrderNumber.message}";
    Class<?>[] groups() default {};
    Class<? extends Payload>[] payload() default {};
}
```

Example 2.2, “Simple constraint definition” marks a String as a well-formed order number. The constraint Validator is implemented by `OrderNumberValidator`.

Example 2.3. Constraint definition with default parameter

```
package com.acme.constraint;

/**
 * A frequency in Hz as audible to human ear.
 * Adjustable to the age of the person.
 * Accept Numbers.
 */
```

```
@Documented
@Constraint(validatedBy = AudibleValidator.class)
@Target({ METHOD, FIELD, ANNOTATION_TYPE,
CONSTRUCTOR, PARAMETER })
@Retention(RUNTIME)
public @interface Audible {
    Age age() default Age.YOUNG;
    String message() default
"{com.acme.constraint.Audible.message}";
    Class<?>[] groups() default {};
    Class<? extends Payload>[] payload() default {};

    public enum Age {
        YOUNG,
        WONDERING
        OLD
    }
}
```

Example 2.3, "Constraint definition with default parameter" ensures that a given frequency is within the scope of human ears. The constraint definition includes an optional parameter that may be specified when the constraint is applied.

Example 2.4. Constraint definition with mandatory parameter

```
package com.acme.constraint;

/**
 * Defines the list of values accepted
 * Accepts int or Integer objects
 */
@Documented
@Constraint(validatedBy =
DiscreteListOfIntegerValidator.class)
@Target({ METHOD, FIELD, ANNOTATION_TYPE,
CONSTRUCTOR, PARAMETER })
@Retention(RUNTIME)
public @interface Acceptable {
    int[] value();
    String message() default
"{com.acme.constraint.Acceptable.message}";
    Class<?>[] groups() default {};
    Class<? extends Payload>[] payload() default {};
}
```

Example 2.4, “Constraint definition with mandatory parameter” defines a list of acceptable values expressed as an array: the `value` property must be specified when the constraint is applied.