Well-Formed vs. Valid

- Well-Formed
  - can determine start position and length
  - bits are convertible to the expected simple type
- Valid
  - Schema Valid - data obeys facets and max/min occurs
  - Schematron Valid - data obeys additional rules
- Parsing should accept well-formed data
- Parsing should *not* reject invalid data
  - Why? - Forensics: so you can look at the invalid data!
    - Without this you can't even see the data - you can't parse it into memory
  - Second why: Composition properties
    - DFDL assumes a backtracking parser.
    - Use of dfdl:checkConstraints(.) function will cause backtracking to try other alternatives when data is well-formed (just invalid)

Avoid using dfdl:checkConstraints()

- dfdl:checkConstraints(.) escalates XSD facet check failures into DFDL parse errors

- Proper uses: To check facets that are associated with well-formed data.
- Example 1: data has 3 kinds of records encoded by a single-character code which must be A, F, or Q.
  - If the code is not one of those, it's an error and you can't parse the data
  - Express this as XSD enumeration facets
  - Use dfdl:checkConstraints(.) so the parse fails if the A, F, or Q are not found.
- Example 2: data has a length or count (array number of items) field which is 32 bits, but the maximum value is 9999. Longer lengths or larger counts cannot be represented.
  - Express this as a maxInclusive facet on an unsigned int type.
  - Use dfdl:checkConstraints(.) to ensure the length/count is not excessive.

Parse Errors vs. "Fail Fast" approach

- You should not use DFDL parse errors as a 'fail fast' way to reject invalid data.
- Why? Composition properties: A DFDL parse error doesn't 'fail' the parse, it causes backtracking to other alternatives of the parse.
- If you didn't write those other parts of the schema (e.g., because your schema is being used as a component in a larger schema) it is unclear what this backtracking will cause, but it is almost certainly not 'failing fast'.
- Often you will get 'left over data' problems, if the parse succeeds up to the invalid data.

Avoid dfdl:checkConstraints

- Look at simple type "longitude_degrees" below.
- Forcing parsing to only succeed on valid data (-180 to 180) turns out to be a mistake.

```
<simpleType name="longitude_degrees"
            dfdl:binaryFloatRep="ieee"
            dfdl:lengthKind="implicit">
  <annotation><appinfo source="http://www.ogf.org/dfdl">
    <!--
        bad idea. Don't do this to check valid values of numbers.
        These numbers are well formed even if out of range.
        Turn on Daffodil's validation, or use a separate validator process.
     -->
    <dfdl:assert>{ dfdl:checkConstraints(.) }</dfdl:assert>
  </appinfo></annotation>
  <restriction base="xs:float">
    <minInclusive value="-180.0"/>
    <maxInclusive value="180.0"/>
  </restriction>
</simpleType>
```

Avoid using dfdl:checkConstraints()

- If you are just validating the data, turn on Daffodil validation, and test for validation errors at end of parse.
  - validation 'limited' (or 'daffodil', meaning built-in) is done efficiently by Daffodil as it traverses the data
  - validation 'full' (or 'xerces') outputs XML and calls Xerces to validate it. This is less efficient.

Unparsing

- In dfdl:outputValueCalc expressions, accept any value that is well-formed, not only valid values.
- Why?
  - Allows generation of invalid data for testing
  - Symmetry with parsing - what the parser can create, the unparser should be able to serialize back

- Exception: Reject Elements
  - Schemas can be designed to tolerate bad data and create reject elements
    - Especially for large file formats where failing to parse the whole file on one bad data item may be undesirable.
  - Reject elements should be designed to always be invalid