Advanced Object-oriented Programming

Lecture 3

Fields and ADTs
# Battle of the Bands

<table>
<thead>
<tr>
<th>Name</th>
<th>Volume</th>
<th>Attitude</th>
<th>Cool</th>
<th>Eclecticism</th>
<th>Hair</th>
</tr>
</thead>
<tbody>
<tr>
<td>Goldfrapp</td>
<td>5</td>
<td>3</td>
<td>7</td>
<td>17</td>
<td>12</td>
</tr>
<tr>
<td>Squarepusher</td>
<td>6</td>
<td>7</td>
<td>6</td>
<td>17</td>
<td>23</td>
</tr>
</tbody>
</table>

Player 1: Volume: 5  
Player 2: Volume: 6  

Player 2 wins
# Battle of the Bands

<table>
<thead>
<tr>
<th>Name</th>
<th>Volume</th>
<th>Attitude</th>
<th>Cool</th>
<th>Eclecticism</th>
<th>Hair</th>
</tr>
</thead>
<tbody>
<tr>
<td>Goldfrapp</td>
<td>5</td>
<td>3</td>
<td>7</td>
<td>17</td>
<td>12</td>
</tr>
<tr>
<td>Squarepusher</td>
<td>6</td>
<td>7</td>
<td>6</td>
<td>17</td>
<td>23</td>
</tr>
</tbody>
</table>

Player 1
Volume: 5

Player 2
Volume: 6

Player 2 wins
Battle of the Bands

Name : Goldfrapp
Volume : 5
Attitude : 3
Cool : 7
Eclecticism : 17
Hair : 12

Name : Squarepusher
Volume : 6
Attitude : 7
Cool : 6
Eclecticism : 17
Hair : 23

Player 1
Volume: 5

Player 2
Volume: 6

Player 2 wins
Battle of the Bands

Name : Goldfrapp
Volume : 5
Attitude : 3
Cool : 7
Eclecticism : 17
Hair : 12

Name : Squarepusher
Volume : 6
Attitude : 7
Cool : 6
Eclecticism : 17
Hair : 23

Player 1
Volume: 5

Player 2
Volume: 6

Player 2 wins
Battle of the Bands

**Player 1**
Name: The Coral
Volume: 6
Attitude: 4
Cool: 14
Eclecticism: 44
Hair: 20

**Player 2**
Name: The Zutons
Volume: 7
Attitude: 10
Cool: 8
Eclecticism: 10
Hair: 22

Player 2 wins
Battle of the Bands

Name : The Coral
Volume : 6
Attitude : 4
Cool : 14
Eclecticism : 44
Hair : 20

Name : The Zutons
Volume : 7
Attitude : 10
Cool : 8
Eclecticism : 10
Hair : 22

Player 1
Hair: 20

Player 2
Hair: 22

Player 2 wins
Battle of the Bands

**Player 1**
- Name: The Coral
- Volume: 6
- Attitude: 4
- Cool: 14
- Eclecticism: 44
- Hair: 20

**Player 2**
- Name: The Zutons
- Volume: 7
- Attitude: 10
- Cool: 8
- Eclecticism: 10
- Hair: 22

Player 2 wins
<table>
<thead>
<tr>
<th></th>
<th>The Coral</th>
<th></th>
<th>The Zutons</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Name</td>
<td></td>
<td>Name</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Volume</td>
<td>6</td>
<td>Volume</td>
<td>7</td>
<td></td>
</tr>
<tr>
<td>Attitude</td>
<td>4</td>
<td>Attitude</td>
<td>10</td>
<td></td>
</tr>
<tr>
<td>Cool</td>
<td>14</td>
<td>Cool</td>
<td>8</td>
<td></td>
</tr>
<tr>
<td>Eclecticism</td>
<td>44</td>
<td>Eclecticism</td>
<td>10</td>
<td></td>
</tr>
<tr>
<td>Hair</td>
<td>20</td>
<td>Hair</td>
<td>22</td>
<td></td>
</tr>
</tbody>
</table>

Player 1
Hair: 20

Player 2
Hair: 22

Player 2 wins
Battle of the Bands: Requirements

Task:
implement a program that allows a user to play Battle of the Bands against the computer.

Each player holds a hand of cards. On each turn, the player whose turn it is selects an attribute from their uppermost card and states its value. Their opponent states the value of the same attribute of their uppermost card. The winner is the player with the higher value. They take the two uppermost cards and add them to the bottom of their hand. Play proceeds as above, with the winner taking the turn.
Battle of the Bands: Requirements

**Task:**
implement a program that allows a user to play Battle of the Bands against the computer.

Each player holds a hand of cards.
On each turn, the player whose turn it is selects an attribute from their uppermost card and states its value.
Their opponent states the value of the same attribute of their uppermost card.
The winner is the player with the higher value.
They take the two uppermost cards and add them to the bottom of their hand.
Play proceeds as above, with the winner taking the turn.
Battle of the Bands: Requirements

**Task:**
implement a program that allows a user to play Battle of the Bands against the computer.

Each player holds a hand of cards.
On each turn, the player whose turn it is selects an attribute from their uppermost card and states its value.
Their opponent states the value of the same attribute of their uppermost card.
The winner is the player with the higher value.
They take the two uppermost cards and add them to the bottom of their hand.
Play proceeds as above, with the winner taking the turn.
Candidate Classes

- **player**  More like a value: Player 1/2
- **hand**    This is data
- **card**    This is data
- **turn**    More like a value: Player 1/2
- **attribute**  Actually, this is a possibility: enum
- **value**   just use int, String, etc.
- **opponent**  More like a value: Player 1/2
- **winner**  More like a value: Player 1/2, whoever wins
- **bottom**  More like a value: index of last card in array

but why just these?
Candidate Classes

- player  More like a value: Player 1/2
- hand    This is data
- card    This is data
- turn    More like a value: Player 1/2
- attribute  Actually, this is a possibility: enum
- value   just use int, String, etc.
- opponent More like a value: Player 1/2
- winner  More like a value: Player 1/2, whoever wins
- bottom  More like a value: index of last card in array

but why just these?
Candidate Classes

- **player**  More like a value: Player 1/2
- **hand**   This is data
- **card**   This is data
- **turn**   More like a value: Player 1/2
- **attribute**  Actually, this is a possibility: enum
- **value**   just use int, String, etc.
- **opponent**   More like a value: Player 1/2
- **winner**   More like a value: Player 1/2, whoever wins
- **bottom**   More like a value: index of last card in array

but why just these?
Candidate Classes

- **player**  More like a value: Player 1/2
- **hand**    This is data
- **card**    This is data
- **turn**    More like a value: Player 1/2
- **attribute**  Actually, this is a possibility: enum
- **value**   just use int, String, etc.
- **opponent** More like a value: Player 1/2

- **winner**  More like a value: Player 1/2, whoever wins
- **bottom**  More like a value: index of last card in array

but why just these?
Candidate Classes

- **player** More like a value: Player 1/2
- **hand** This is data
- **card** This is data
- **turn** More like a value: Player 1/2
- **attribute** Actually, this is a possibility: enum
- **value** Just use int, String, etc.
- **opponent** More like a value: Player 1/2
- **winner** More like a value: Player 1/2, whoever wins
- **bottom** More like a value: index of last card in array

but why just these?
Candidate Classes

- **player**: More like a value: Player 1/2
- **hand**: This is data
- **card**: This is data
- **turn**: More like a value: Player 1/2
- **attribute**: Actually, this is a possibility: enum
- **value**: just use int, String, etc.
- **opponent**: More like a value: Player 1/2
- **winner**: More like a value: Player 1/2, whoever wins
- **bottom**: More like a value: index of last card in array

but why just these?
Candidate Classes

- player: More like a value: Player 1/2
- hand: This is data
- card: This is data
- turn: More like a value: Player 1/2
- attribute: Actually, this is a possibility: enum
- value: just use int, String, etc.
- opponent: More like a value: Player 1/2
- winner: More like a value: Player 1/2, whoever wins
- bottom: More like a value: index of last card in array

but why just these?
Candidate Classes

- player  More like a value: Player 1/2
- hand   This is data
- card   This is data
- turn   More like a value: Player 1/2
- attribute  Actually, this is a possibility: enum
- value  just use int, String, etc.
- opponent  More like a value: Player 1/2
- winner  More like a value: Player 1/2, whoever wins
- bottom  More like a value: index of last card in array

but why just these?
How to Design Classes

Principle 0

Java programmers build (most of) their classes on data.

This will be the topic of the next few lectures.
Card Requirements

Each card contains data for one band:

- **name** — the name of the band
- **volume** — how loudly they play
- **attitude** — how brash they are
- **cool** — how cool they are
- **eclecticism** — variety of styles
- **hair** — how cool their hair is

I.e., it is a *sextuple* (bunch of six things) of a particular type:

(String, int, int, int, int, int)
Card Requirements

Each card contains data for one band:

- **name** — the name of the band
- **volume** — how loudly they play
- **attitude** — how brash they are
- **cool** — how cool they are
- **eclecticism** — variety of styles
- **hair** — how cool their hair is

I.e., it is a *sestuple* (bunch of six things) of a particular *type*:

\[(\text{String}, \text{int}, \text{int}, \text{int}, \text{int}, \text{int})\]
ADTs, Properly

An *Abstract Data Type* is

a set of abstract data values
together with operations that act upon them.

Notes:

- ADTs are implemented in specific languages (Java, C, etc)
- How do we describe ADTs?
  - we need to describe the values
  - and the operations
ADTs, Properly

An *Abstract Data Type* is

<table>
<thead>
<tr>
<th>Definition of ADT</th>
</tr>
</thead>
<tbody>
<tr>
<td>a set of abstract data values together with operations that act upon them.</td>
</tr>
</tbody>
</table>

Notes:

- ADTs are **implemented** in specific languages (Java, C, etc)
- How do we describe ADTs?
  - we need to *describe* the values
  - and the operations
ADTs, Properly

An *Abstract Data Type* is

<table>
<thead>
<tr>
<th>Definition of ADT</th>
</tr>
</thead>
<tbody>
<tr>
<td>a set of abstract data values together with operations that act upon them.</td>
</tr>
</tbody>
</table>

Notes:

- ADTs are implemented in specific languages (Java, C, etc)
- How do we describe ADTs?
  - we need to *describe* the values
  - and the *operations*
ADTs, Properly

An *Abstract Data Type* is

A set of abstract data *values* together with *operations* that act upon them.

**Notes:**

- ADTs are implemented in specific languages (Java, C, etc)
- How do we describe ADTs?
  - We need to *describe* the *values*
  - And the *operations*
Example ADTs

### Lists

A **List** is a sequence of values with operations to:

- **add** a value to the start of the list
- **get** the value at the start of the list
- **get** the ‘tail’ of the list

### Stacks

A **Stack** is a sequence of values with operations to:

- **push** a value onto the top of the stack
- **get** the value at the top of the stack
- **pop** the first value off of the stack

how do we define these?
## Example ADTs

### Lists

A **List** is a sequence of values with operations to:
- **add** a value to the start of the list
- **get** the value at the start of the list
- **get** the ‘tail’ of the list

### Stacks

A **Stack** is a sequence of values with operations to
- **push** a value onto the top of the stack
- **get** the value at the top of the stack
- **pop** the first value off of the stack
Example ADTs

### Lists

A **List** is a *sequence of values* with operations to:

- **add** a value to the start of the list
- **get** the value at the start of the list
- **get** the ‘tail’ of the list

### Stacks

A **Stack** is a *sequence of values* with operations to:

- **push** a value onto the top of the stack
- **get** the value at the top of the stack
- **pop** the first value off of the stack

how do we define these?
**Maude** is a language for specifying abstract data types. Within a Maude module, you can specify:

- **sorts** — actually, just names for the sets of abstract data values
- **operations** — together with their types (types of arguments and ‘return type’)
- **equations** — that describe the behaviour of the operations
Maude is a language for specifying abstract data types. Within a Maude module, you can specify:

- **sorts** — actually, just names for the sets of abstract data values
- **operations** — together with their types (types of arguments and ‘return type’)
- **equations** — that describe the behaviour of the operations
Maude is a language for specifying abstract data types. Within a Maude module, you can specify:

- **sorts** — actually, just names for the sets of abstract data values
- **operations** — together with their types (types of arguments and ‘return type’)
- **equations** — that describe the behaviour of the operations
Maude is a language for specifying abstract data types. Within a Maude module, you can specify:

- **sorts** — actually, just names for the sets of abstract data values
- **operations** — together with their **types** (types of arguments and ‘return type’)
- **equations** — that describe the behaviour of the operations
Maude is a language for specifying abstract data types. Within a Maude module, you can specify:

- **sorts** — actually, just names for the sets of abstract data values
- **operations** — together with their types (types of arguments and ‘return type’)
- **equations** — that describe the behaviour of the operations
Maude Made Easy

The form of a Maude module is:

```
| fmod | NAME_OF_THE_MODULE  is |
|      | declarations (sorts, operations, equations) |
| endfm |
```

Sort Declarations

```
| sort | NameOfTheSort  . |
```

Operation Declarations

```
| op   | nameOfTheOperation : Sort₁…Sortₙ  →  Sort  . |
```

For ‘functional module’
(because ADTs are used in functional programming).
The form of a Maude module is:

```
[Maude modules]

fmod NAME_OF_THE_MODULE is
declarations (sorts, operations, equations)
endfm
```

**Sort Declarations**

```
sort NameOfTheSort .
```

**Operation Declarations**

```
op nameOfTheOperation : Sort_1...Sort_n -> Sort .
```

For ‘functional module’ (because ADTs are used in functional programming).
The form of a Maude module is:

```maude
fmod  NAME_OF_THE_MODULE  is
  declarations (sorts, operations, equations)
endfm
```

Sort Declarations

```maude
sort  NameOfTheSort .
```

Operation Declarations

```maude
op  nameOfTheOperation  :  Sort₁...Sortᵢ  →  Sort .
```

For ‘functional module’ (because ADTs are used in functional programming).
The form of a Maude module is:

```
fmod NAME_OF_THE_MODULE is
  declarations (sorts, operations, equations)
endfm
```

**Maude modules**

**Sort Declarations**

```
sort NameOfTheSort .
```

**Operation Declarations**

```
op nameOfTheOperation : Sort_1...Sort_n => Sort .
```

For ‘functional module’ (because ADTs are used in functional programming).
Imports modules called STRING and INT that specify sorts 
String (strings) and Int (integers).
This is the sort of (String,Int,Int,Int,Int,Int)-sextuples.
‘Constructor’ operation that makes a BandCard from the 
components of the sextuple.
Imports modules called **STRING** and **INT** that specify sorts **String** (strings) and **Int** (integers).

This is the sort of (String,Int,Int,Int,Int,Int)-sestuples. ‘Constructor’ operation that makes a **BandCard** from the components of the sestuple.
Imports modules called STRING and INT that specify sorts String (strings) and Int (integers).
This is the sort of (String,Int,Int,Int,Int,Int)-sextuples.
‘Constructor’ operation that makes a BandCard from the components of the sextuple.
file: BandCard.maude

```maude
fmod BAND_CARD is
  protecting STRING .  *** import strings
  protecting INT .    *** import integers
  sort BandCard .
  op newBandCard : String Int Int Int Int Int Int Int -> BandCard .
```

Imports modules called STRING and INT that specify sorts String (strings) and Int (integers).
This is the sort of (String,Int,Int,Int,Int,Int,Int)-sestuples.
‘Constructor’ operation that makes a BandCard from the components of the sestuple.
Cards in Maude

An ‘accessor’ operation that gives the name of a BandCard. It takes a BandCard as argument, and returns a String as result.

Get the volume-value of a BandCard — returns an Int. etc.
An ‘accessor’ operation that gives the name of a BandCard. It takes a BandCard as argument, and returns a String as result. Get the volume-value of a BandCard — returns an Int. etc.
op getName : BandCard -> String.

op getVolume : BandCard -> Int.

op getAttitude : BandCard -> Int.

op getCool : BandCard -> Int.

op getEclecticism : BandCard -> Int.

op getHair : BandCard -> Int.

An ‘accessor’ operation that gives the name of a BandCard. It takes a BandCard as argument, and returns a String as result. Get the volume-value of a BandCard — returns an Int. etc.
An ‘accessor’ operation that gives the name of a BandCard. It takes a BandCard as argument, and returns a String as result. Get the volume-value of a BandCard — returns an Int. etc.
### Cards in Maude

<table>
<thead>
<tr>
<th>Operation</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>getName</code></td>
<td>An ‘accessor’ operation that gives the name of a BandCard. Takes a BandCard as argument and returns a String as result.</td>
</tr>
<tr>
<td><code>getVolume</code></td>
<td>Get the volume-value of a BandCard — returns an Int.</td>
</tr>
<tr>
<td><code>getAttitude</code></td>
<td></td>
</tr>
<tr>
<td><code>getCool</code></td>
<td></td>
</tr>
<tr>
<td><code>getEclecticism</code></td>
<td></td>
</tr>
<tr>
<td><code>getHair</code></td>
<td></td>
</tr>
</tbody>
</table>

An ‘accessor’ operation that gives the name of a BandCard. It takes a BandCard as argument, and returns a String as result. Get the volume-value of a BandCard — returns an Int. etc.
An ‘accessor’ operation that gives the name of a BandCard. It takes a BandCard as argument, and returns a String as result.

Get the volume-value of a BandCard — returns an Int.

etc.
An ‘accessor’ operation that gives the name of a BandCard. It takes a BandCard as argument, and returns a String as result. Get the volume-value of a BandCard — returns an Int. etc.
An ‘accessor’ operation that gives the name of a BandCard. It takes a BandCard as argument, and returns a String as result.

Get the volume-value of a BandCard — returns an Int.

etc.
An ‘accessor’ operation that gives the name of a BandCard. It takes a BandCard as argument, and returns a String as result. Get the volume-value of a BandCard — returns an Int. etc.
Terms

Using the operations, and constants such as integers 12, 66, etc., we can write terms that represent abstract data values. For example:

```plaintext
op newBandCard : String Int Int Int Int Int Int Int -> BandCard.
newBandCard("Goldfrapp", 5, 3, 7, 17, 12)
is a term of sort BandCard.

Similarly,

```plaintext
op getVolume : BandCard -> Int.
getVolume(newBandCard("Goldfrapp", 5, 3, 7, 17, 12))
is a term of sort Int (it should be equal to 5).
```
Terms

Using the operations, and constants such as integers 12, 66, etc., we can write terms that represent abstract data values. For example:

```
op newBandCard : String Int Int Int Int Int Int -> BandCard.
```

newBandCard("Goldfrapp", 5, 3, 7, 17, 12)
is a term of sort BandCard.

Similarly,

```
op getVolume : BandCard -> Int.
```

getVolume(newBandCard("Goldfrapp", 5, 3, 7, 17, 12))
is a term of sort Int (it should be equal to 5).
Terms

Using the operations, and constants such as integers 12, 66, etc., we can write terms that represent abstract data values. For example:

```
op newBandCard : String Int Int Int Int Int -&gt; BandCard.
```

newBandCard("Goldfrapp", 5, 3, 7, 17, 12)

is a term of sort BandCard.

Similarly,

```
op getVolume : BandCard -&gt; Int.
```

getVolume(newBandCard("Goldfrapp", 5, 3, 7, 17, 12))

is a term of sort Int (it should be equal to 5).
Terms

Using the operations, and constants such as integers 12, 66, etc., we can write terms that represent abstract data values.

For example:

\[
\text{op } \text{newBandCard} : \text{String Int Int Int Int Int Int } \rightarrow \text{BandCard}.
\]

\[
\text{newBandCard("Goldfrapp", 5, 3, 7, 17, 12)}
\]

is a term of sort BandCard.

Similarly,

\[
\text{op } \text{getVolume} : \text{BandCard } \rightarrow \text{Int}.
\]

\[
\text{getVolume(newBandCard("Goldfrapp", 5, 3, 7, 17, 12))}
\]

is a term of sort Int (it should be equal to 5).
Terms

Using the operations, and constants such as integers 12, 66, etc., we can write terms that represent abstract data values. For example:

```plaintext
op newBandCard : String Int Int Int Int Int Int -> BandCard.
```

newBandCard("Goldfrapp", 5, 3, 7, 17, 12)

is a term of sort BandCard.

Similarly,

```plaintext
op getVolume : BandCard -> Int.
```

getVolume(newBandCard("Goldfrapp", 5, 3, 7, 17, 12))

is a term of sort Int (it should be equal to 5).
Terms

Using the operations, and constants such as integers 12, 66, etc., we can write terms that represent abstract data values.

For example:

```plaintext
op newBandCard : String Int Int Int Int Int Int -> BandCard.
```

`newBandCard("Goldfrapp", 5, 3, 7, 17, 12)` is a term of sort BandCard.

Similarly,

```plaintext
op getVolume : BandCard -> Int.
```

`getVolume(newBandCard("Goldfrapp", 5, 3, 7, 17, 12))` is a term of sort Int (it should be equal to 5).
Using the operations, and constants such as integers 12, 66, etc., we can write terms that represent abstract data values.

For example:

\begin{verbatim}
  op newBandCard : String Int Int Int Int Int -\rightarrow BandCard .
\end{verbatim}

```
newBandCard("Goldfrapp", 5, 3, 7, 17, 12)
```

is a term of sort BandCard.

Similarly,

\begin{verbatim}
  op getVolume : BandCard -\rightarrow Int .
\end{verbatim}

```
getVolume(newBandCard("Goldfrapp", 5, 3, 7, 17, 12))
```

is a term of sort Int (it should be equal to 5).
Terms

Using the operations, and constants such as integers 12, 66, etc., we can write terms that represent abstract data values.

For example:

\[
\text{op newBandCard : String Int Int Int Int Int } \rightarrow \text{ BandCard}.
\]

newBandCard("Goldfrapp", 5, 3, 7, 17, 12)

is a term of sort BandCard.

Similarly,

\[
\text{op getVolume : BandCard } \rightarrow \text{ Int}.
\]

getVolume(newBandCard("Goldfrapp", 5, 3, 7, 17, 12))

is a term of sort Int (it should be equal to 5).
Terms

Using the operations, and constants such as integers 12, 66, etc., we can write terms that represent abstract data values. For example:

```
op newBandCard : String Int Int Int Int Int Int --> BandCard .
```

everyBandCard("Goldfrapp", 5, 3, 7, 17, 12)

is a term of sort BandCard.

Similarly,

```
op getVolume : BandCard --> Int .
```

getVolume(everyBandCard("Goldfrapp", 5, 3, 7, 17, 12))

is a term of sort Int (it should be equal to 5).
Terms

Using the operations, and constants such as integers 12, 66, etc., we can write terms that represent abstract data values. For example:

\[
\text{op newBandCard : String Int Int Int Int Int Int \rightarrow BandCard .}
\]

\[
\text{newBandCard("Goldfrapp", 5, 3, 7, 17, 12)}
\]
is a term of sort BandCard.

Similarly,

\[
\text{op getVolume : BandCard \rightarrow Int .}
\]

\[
\text{getVolume(newBandCard("Goldfrapp", 5, 3, 7, 17, 12))}
\]
is a term of sort Int (it should be equal to 5).
Terms

Using the operations, and constants such as integers 12, 66, etc., we can write terms that represent abstract data values. For example:

\[
\text{op newBandCard : String Int Int Int Int Int → BandCard}.
\]

\[
\text{newBandCard("Goldfrapp", 5, 3, 7, 17, 12)}
\]
is a term of sort BandCard.

Similarly,

\[
\text{op getVolume : BandCard → Int}.
\]

\[
\text{getVolume(newBandCard("Goldfrapp", 5, 3, 7, 17, 12))}
\]
is a term of sort Int (it should be equal to 5).
Equations: making Maude move

Note we expect:
\[
\text{getVolume(newBandCard("Goldfrapp", 5, 3, 7, 17, 12))} = 5
\]

Similarly, we want:
\[
\text{getVolume(newBandCard("The Coral", 6, 4, 14, 44, 20))} = 6
\]

And for any values, S, I, J, K, L and M
\[
\text{getVolume(newBandCard(S, I, J, K, L, M))} = I
\]
Equations: making Maude move

Note we expect:
\[
\text{getVolume(newBandCard("Goldfrapp", 5, 3, 7, 17, 12))} = 5
\]

Similarly, we want:
\[
\text{getVolume(newBandCard("The Coral", 6, 4, 14, 44, 20))} = 6
\]

And for any values, S, I, J, K, L and M
\[
\text{getVolume(newBandCard(S, I, J, K, L, M))} = I
\]
Equations: making Maude move

Note we expect:
\[
\text{getVolume}(\text{newBandCard}("Goldfrapp", 5, 3, 7, 17, 12)) = 5
\]

Similarly, we want:
\[
\text{getVolume}(\text{newBandCard}("The Coral", 6, 4, 14, 44, 20)) = 6
\]

And for any values, S, I, J, K, L and M
\[
\text{getVolume}(\text{newBandCard}(S, I, J, K, L, M)) = I
\]
Equations: making Maude move

Note we expect:
\[ \text{getVolume}(\text{newBandCard}("Goldfrapp", 5, 3, 7, 17, 12)) = 5 \]

Similarly, we want:
\[ \text{getVolume}(\text{newBandCard}("The Coral", 6, 4, 14, 44, 20)) = 6 \]

And for any values, S, I, J, K, L and M
\[ \text{getVolume}(\text{newBandCard}(S, I, J, K, L, M)) = I \]
Equations: making Maude move

Note we expect:
\[ \text{getVolume(newBandCard("Goldfrapp", 5, 3, 7, 17, 12))} = 5 \]

Similarly, we want:
\[ \text{getVolume(newBandCard("The Coral", 6, 4, 14, 44, 20))} = 6 \]

And for any values, S, I, J, K, L and M
\[ \text{getVolume(newBandCard(S, I, J, K, L, M))} = I \]
Equations: making Maude move

Note we expect:
\[
\text{getVolume(newBandCard("Goldfrapp", 5, 3, 7, 17, 12))} = 5
\]

Similarly, we want:
\[
\text{getVolume(newBandCard("The Coral", 6, 4, 14, 44, 20))} = 6
\]

And for any values, S, I, J, K, L and M
\[
\text{getVolume(newBandCard(S, I, J, K, L, M))} = I
\]
Equations: making Maude move

\[
\begin{align*}
\text{var} & \quad S : \text{String} \quad \text{*** for all strings } S \\
\text{vars} & \quad I, J, K, L, M : \text{Int} \quad \text{*** for all ints } I, J, K, L \text{ and } M \\
\text{eq} & \quad \text{getName}(\text{newBandCard}(S, I, J, K, L, M)) = S \\
\text{eq} & \quad \text{getVolume}(\text{newBandCard}(S, I, J, K, L, M)) = I \\
\text{eq} & \quad \text{getAttitude}(\text{newBandCard}(S, I, J, K, L, M)) = J \\
\text{eq} & \quad \text{getCool}(\text{newBandCard}(S, I, J, K, L, M)) = K \\
\text{eq} & \quad \text{getEclecticism}(\text{newBandCard}(S, I, J, K, L, M)) = L \\
\text{eq} & \quad \text{getHair}(\text{newBandCard}(S, I, J, K, L, M)) = M \\
\text{endfm}
\end{align*}
\]
Equations: making Maude move

var S : String . *** for all strings S
vars I J K L M : Int . *** for all ints I, J, K, L and M

eq getName(newBandCard(S, I, J, K, L, M)) = S .
eq getVolume(newBandCard(S, I, J, K, L, M)) = I .
eq getAttitude(newBandCard(S, I, J, K, L, M)) = J .
eq getCool(newBandCard(S, I, J, K, L, M)) = K .
eq getEclecticism(newBandCard(S, I, J, K, L, M)) = L .
eq getHair(newBandCard(S, I, J, K, L, M)) = M .
endfm
Equations: making Maude move

```plaintext
var  S : String . *** for all strings S
vars I J K L M : Int . *** for all ints I, J, K, L and M

eq  getName(newBandCard(S, I, J, K, L, M))   =   S .
eq  getVolume(newBandCard(S, I, J, K, L, M))  =   I .
eq  getAttitude(newBandCard(S, I, J, K, L, M)) =   J .
eq  getCool(newBandCard(S, I, J, K, L, M))   =   K .
eq  getEclecticism(newBandCard(S, I, J, K, L, M)) = L .
eq  getHair(newBandCard(S, I, J, K, L, M))   =   M .
endfm
```
Equations: making Maude move

\begin{verbatim}
var S : String . *** for all strings S
vars I J K L M : Int . *** for all ints I, J, K, L and M

eq getName(newBandCard(S, I, J, K, L, M)) = S .
eq getVolume(newBandCard(S, I, J, K, L, M)) = I .
eq getAttitude(newBandCard(S, I, J, K, L, M)) = J .
eq getCool(newBandCard(S, I, J, K, L, M)) = K .
eq getEclecticism(newBandCard(S, I, J, K, L, M)) = L .
eq getHair(newBandCard(S, I, J, K, L, M)) = M .
endfm
\end{verbatim}
Equations: making Maude move

\[
\begin{align*}
\text{var} & \quad S : \text{String}. & \quad & \text{*** for all strings S} \\
\text{vars} & \quad I \ J \ K \ L \ M : \text{Int}. & \quad & \text{*** for all ints } I, J, K, L \text{ and } M \\
\text{eq} & \quad \text{getName}(\text{newBandCard}(S, I, J, K, L, M)) = S. \\
\text{eq} & \quad \text{getVolume}(\text{newBandCard}(S, I, J, K, L, M)) = I. \\
\text{eq} & \quad \text{getAttitude}(\text{newBandCard}(S, I, J, K, L, M)) = J. \\
\text{eq} & \quad \text{getCool}(\text{newBandCard}(S, I, J, K, L, M)) = K. \\
\text{eq} & \quad \text{getEclecticism}(\text{newBandCard}(S, I, J, K, L, M)) = L. \\
\text{eq} & \quad \text{getHair}(\text{newBandCard}(S, I, J, K, L, M)) = M. \\
\end{align*}
\]
That’s All, Folks!

Summary

- Abstract Data Types
- Maude

Next:

Implementing ADTs