Lecture 4
Implementing ADTs
Equations: making Maude move

Recall the equations from Bandcard.maude:

<table>
<thead>
<tr>
<th>var</th>
<th>S : String</th>
<th>*** for all strings S</th>
</tr>
</thead>
<tbody>
<tr>
<td>vars</td>
<td>I J K L M : Int</td>
<td>*** for all ints I, J, K, L and M</td>
</tr>
<tr>
<td>eq</td>
<td>getName(newBandCard(S, I, J, K, L, M)) = S .</td>
<td></td>
</tr>
<tr>
<td>eq</td>
<td>getVolume(newBandCard(S, I, J, K, L, M)) = I .</td>
<td></td>
</tr>
<tr>
<td>eq</td>
<td>getAttitude(newBandCard(S, I, J, K, L, M)) = J .</td>
<td></td>
</tr>
<tr>
<td>eq</td>
<td>getCool(newBandCard(S, I, J, K, L, M)) = K .</td>
<td></td>
</tr>
<tr>
<td>eq</td>
<td>getEclecticism(newBandCard(S, I, J, K, L, M)) = L .</td>
<td></td>
</tr>
<tr>
<td>eq</td>
<td>getHair(newBandCard(S, I, J, K, L, M)) = M .</td>
<td></td>
</tr>
</tbody>
</table>
The Maude command **reduce** causes Maude to rewrite terms, using the equations in a module.

For example

```plaintext
var S : String .
vars I J K L M : Int .
eq getName(newBandCard(S, I, J, K, L, M)) = S .

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

The term matches the left-hand side of the equation... ...and is equal to the right-hand side.
The Maude command `reduce` causes Maude to rewrite terms, using the equations in a module.

For example

```plaintext
var S : String .
vars I J K L M : Int .
eq getName(newBandCard(S, I, J, K, L, M)) = S .

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

The term matches the left-hand side of the equation... ...and is equal to the right-hand side.
Equations: making Maude move

The Maude command \texttt{reduce} causes Maude to rewrite terms, using the equations in a module.

For example

\begin{verbatim}
var S : String .
vars I J K L M : Int .
eq getName(newBandCard(S, I, J, K, L, M)) = S .
\end{verbatim}

\texttt{getName(newBandCard("Goldfrapp", 5, 3, 7, 17, 12)) = "Goldfrapp"}

The term matches the left-hand side of the equation. . .

. . . and is equal to the right-hand side.
Equations: making Maude move

The Maude command **reduce** causes Maude to rewrite terms, using the equations in a module.

For example

```plaintext
var S : String .
vars I J K L M : Int .
eq getName(newBandCard(S, I, J, K, L, M)) = S .
```

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

The term matches the left-hand side of the equation... ...and is equal to the right-hand side.
Equations: making Maude move

The Maude command **reduce** causes Maude to rewrite terms, using the equations in a module.

For example

```plaintext
var S : String .
vars I J K L M : Int .

eq getName(newBandCard(S, I, J, K, L, M)) = S .
```

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

The term matches the left-hand side of the equation... ...and is equal to the right-hand side.
Equations: making Maude move

The Maude command **reduce** causes Maude to rewrite terms, using the equations in a module.

For example

```plaintext
var S : String .
vars I J K L M : Int .
eq getName(newBandCard(S, I, J, K, L, M)) = S .
```

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

The term matches the left-hand side of the equation... . . .

...and is equal to the right-hand side.
Equations: making Maude move

The Maude command **reduce** causes Maude to rewrite terms, using the equations in a module.

For example

```plaintext
var S : String .
vars I J K L M : Int .
eq getName(newBandCard(S, I, J, K, L, M)) = S .
```

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

The term matches the left-hand side of the equation . . .

. . . and is equal to the right-hand side.
Equations: making Maude move

The Maude command **reduce** causes Maude to rewrite terms, using the equations in a module.

For example

```maude
var S : String .
vars I J K L M : Int .

eq getName(newBandCard(S, I, J, K, L, M)) = S .
```

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

The term matches the left-hand side of the equation... ...and is equal to the right-hand side.
Equations: making Maude move

The Maude command **reduce** causes Maude to rewrite terms, using the equations in a module.

For example

```plaintext
var S : String  .
vars I J K L M : Int  .

eq getName(newBandCard(S, I, J, K, L, M)) = S  .
```

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

The term matches the left-hand side of the equation... 
...and is equal to the right-hand side.
Equations: making Maude move

The Maude command `reduce` causes Maude to rewrite terms, using the equations in a module.

For example

```maude
var S : String .
vars I J K L M : Int .
eq getName(newBandCard(S, I, J, K, L, M)) = S .
```

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

The term matches the left-hand side of the equation... ...and is equal to the right-hand side.
[grantlocalhost] maude

-- Welcome to Maude --

Maude> load BandCard

Maude> reduce getVolume( newBandCard("Goldfrapp",5,3,7,17,12) ) .

...(some stuff omitted)...

result NzNat: 5

Maude> q

Bye.

[grantlocalhost]
Maude Moves!

[grantlocalhost] maude
\|/                   |
-- Welcome to Maude --
\|/                   |
Maude 2.3 built: Feb 14 2007 17:53:50
Copyright 1997-2007 SRI International
Maude> load BandCard
Maude> reduce getVolume(
    newBandCard("Goldfrapp",5,3,7,17,12)) .
...(some stuff omitted)...
result NzNat: 5
Maude> q
Bye.
[grantlocalhost]
[grantlocalhost] maude

-- Welcome to Maude --

Maude 2.3 built: Feb 14 2007 17:53:50
Copyright 1997-2007 SRI International

Maude> load BandCard
Maude> reduce getVolume(newBandCard("Goldfrapp",5,3,7,17,12)) .
...(some stuff omitted)...
result NzNat: 5
Maude> q
Bye.

[grantlocalhost]
Maude Moves!

[grantlocalhost] maude
\|\|\|\|\|\|\|\|\|\|\|\|\|\|\|\|\|\|\|\|
-- Welcome to Maude --
/\|\|\|\|\|\|\|\|\|\|\|\|\|\|\|\|\|\|\|\|
Maude 2.3 built:  Feb 14 2007 17:53:50
Copyright 1997-2007 SRI International
Maude> load BandCard
Maude> reduce getVolume(
  newBandCard("Goldfrapp",5,3,7,17,12)) .
...(some stuff omitted)...
result NzNat: 5
Maude> q
Bye.
[grantlocalhost]
Maude Moves!

[grantlocalhost] maude
/\--------------------------------------------------------------------------------------/
-- Welcome to Maude --
/\--------------------------------------------------------------------------------------\
Maude 2.3 built: Feb 14 2007 17:53:50
Copyright 1997–2007 SRI International
Maude> load BandCard
Maude> reduce getVolume(newBandCard("Goldfrapp",5,3,7,17,12)) .
...(some stuff omitted)...
result NzNat: 5
Maude> q
Bye.
[grantlocalhost]
Maude Moves!

[grantlocalhost] maude
\|/|
-- Welcome to Maude --
/\|
Maude 2.3 built: Feb 14 2007 17:53:50
Copyright 1997-2007 SRI International
Maude> load BandCard
Maude> reduce getVolume(
newBandCard("Goldfrapp",5,3,7,17,12)) .
...(some stuff omitted)...
result NzNat: 5
Maude> q
Bye.
[grantlocalhost]
[grantlocalhost] maude
\|/|
-- Welcome to Maude --
\|/|
Maude 2.3 built: Feb 14 2007 17:53:50
Copyright 1997–2007 SRI International
Maude> load BandCard
Maude> reduce getVolume(
newBandCard("Goldfrapp",5,3,7,17,12)) .
...(some stuff omitted)...
result NzNat: 5
Maude> q
Bye.
[grantlocalhost]
Maude Moves!

[grantlocalhost] maude
\/\…………/\…………/\…………/\…………/
-- Welcome to Maude --
/\…………/\…………/\…………/\…………/
Maude 2.3 built: Feb 14 2007 17:53:50
Copyright 1997–2007 SRI International
Maude> load BandCard
Maude> reduce getVolume(
newBandCard("Goldfrapp",5,3,7,17,12)) .
...(some stuff omitted)...  
result NzNat: 5
Maude> q
Bye.
[grantlocalhost]
Maude Moves!

```
[grantlocalhost] maude
\|/----------/----------/----------/----------/----------/
-- Welcome to Maude --
/\|/----------/----------/----------/----------/----------/\nMaude 2.3 built:  Feb 14 2007 17:53:50
Copyright 1997-2007 SRI International
Maude> load BandCard
Maude> reduce getVolume(
   newBandCard("Goldfrapp",5,3,7,17,12)) .
...(some stuff omitted)...
result NzNat:  5
Maude> q
Bye.
[grantlocalhost]
```
Ready, Steady . . . Code!

Now we know exactly what we have to implement in Java.

Principle 1

In Java, fields do the work of data representation.
Ready, Steady . . . Code!

Now we know exactly what we have to implement in Java.

**Principle 1**

In Java, fields do the work of data representation.
Ready, Steady ... Code!

(String, Int, Int, Int, Int, Int)-tuples:

class BandCard {
    String name;
    int volume;
    int attitude;
    int cool;
    int eclecticism;
    int hair;
    ...
}

file: BandCard.java
(String, Int, Int, Int, Int, Int)-tuples:

class BandCard {
    String name;
    int volume;
    int attitude;
    int cool;
    int eclecticism;
    int hair;
    ...
}
Ready, Steady . . . Code!

(String, Int, Int, Int, Int, Int)-tuples:

class BandCard {
    String name;
    int volume;
    int attitude;
    int cool;
    int eclecticism;
    int hair;
    ...
}
(String, Int, Int, Int, Int, Int)-tuples:

```java
file: BandCard.java

class BandCard {
    String name;
    int volume;
    int attitude;
    int cool;
    int eclecticism;
    int hair;
    ...
}
```
Ready, Steady . . . Code!

(String, Int, Int, Int, Int, Int)-tuples:

```java
class BandCard {
    String name;
    int volume;
    int attitude;
    int cool;
    int eclecticism;
    int hair;
    ...
}
```
Ready, Steady . . . Code!

(String, Int, Int, Int, Int)-tuples:

```
@file: BandCard.java

class BandCard {
  String name;
  int volume;
  int attitude;
  int cool;
  int eclecticism;
  int hair;
  ...
}
```
Ready, Steady . . . Code!

(String, Int, Int, Int, Int, Int)-tuples:

```java
class BandCard {
    String name;
    int volume;
    int attitude;
    int cool;
    int eclecticism;
    int hair;
    ...
}
```
Ready, Steady . . . Code!

(String, Int, Int, Int, Int, Int)-tuples:

```java
class BandCard {
    String name;
    int volume;
    int attitude;
    int cool;
    int eclecticism;
    int hair;
...
}
```
In Java, constructors are (often) used to instantiate fields.

...and, of course, to create (‘construct’) instances.
Ready, Steady . . . Code!

Principle 2

In Java, constructors are (often) used to instantiate fields.

. . . and, of course, to create (‘construct’) instances.
Constructor:

class BandCard constructor

BandCard(String n,
           int v, int a, int c, int e, int h) {
    name = n;
    volume = v;
    attitude = a;
    cool = c;
    eclecticism = e;
    hair = h;
}
Ready, Steady . . . Code!

Constructor:

```java
class BandCard constructor
BandCard(String n,
           int v, int a, int c, int e, int h) {
    name = n;
    volume = v;
    attitude = a;
    cool = c;
    eclecticism = e;
    hair = h;
}
```

Principle 2.1

one argument for each field
Constructor:

```java
class BandCard constructor

BandCard(String n,
           int v, int a, int c, int e, int h) {
    name = n;
    volume = v;
    attitude = a;
    cool = c;
    eclecticism = e;
    hair = h;
}
```

Principle 2.1

one argument for each field
Constructor:

```java
class BandCard constructor

BandCard(String n, int v, int a, int c, int e, int h) {
    name = n;
    volume = v;
    attitude = a;
    cool = c;
    eclecticism = e;
    hair = h;
}
```

Principle 2.1

one argument for each field
Constructors:

```java
class BandCard constructor

BandCard(String n, int v, int a, int c, int e, int h) {
    name = n;
    volume = v;
    attitude = a;
    cool = c;
    eclecticism = e;
    hair = h;
}
```

Principle 2.1

one argument for each field
Constructor:

```java
class BandCard constructor

BandCard(String n,
           int v, int a, int c, int e, int h) {
    name = n;
    volume = v;
    attitude = a;
    cool = c;
    eclecticism = e;
    hair = h;
}
```

Principle 2.1

one argument for each field
Ready, Steady ... Code!

Constructor:

```java
class BandCard constructor

BandCard(String n, 
          int v, int a, int c, int e, int h) {
    name = n;
    volume = v;
    attitude = a;
    cool = c;
    eclecticism = e;
    hair = h;
}
```

Principle 2.1

one argument for each field
Ready, Steady . . . Code!

Principle 3

In Java, methods implement the operations of the ADT in the way specified by the equations.
Ready, Steady . . . Code!

from BandCard.maude

\[
\begin{align*}
\text{op} & \quad \text{name} : \text{BandCard} \rightarrow \text{String} . \\
\text{eq} & \quad \text{name}(\text{newBandCard}(S, I, J, K, L, M)) = S . \\
\end{align*}
\]

is implemented as

```java
public String getName() {
    return name;
}
```
Ready, Steady . . . Code!

from BandCard.maude

\textbf{op} \quad \texttt{getName} : \texttt{BandCard} \rightarrow \texttt{String}.
\textbf{eq} \quad \texttt{getName}(\texttt{newBandCard}(S, I, J, K, L, M)) = S.

is implemented as

\textbf{method} \quad \texttt{Bandcard#getName()}

\begin{verbatim}
public String getName() {
    return name;
}
\end{verbatim}
Ready, Steady . . . Code!

From BandCard.maude

\textbf{op} \texttt{getName} : \texttt{BandCard} \rightarrow \texttt{String}.

\textbf{eq} \texttt{getName(newBandCard(S, I, J, K, L, M))} = S.

Is implemented as

\textbf{public} \texttt{String getName()} {
    \texttt{return name;}
}
Ready, Steady . . . Code!

from BandCard.maude

\textbf{op} \text{getName} : \text{BandCard} \rightarrow \text{String} . \\
\textbf{eq} \text{getName(newBandCard(S, I, J, K, L, M))} = S .

is implemented as

\textbf{method} \text{Bandcard#getName}()

\begin{verbatim}
public String getName() {
    return name;
}
\end{verbatim}
Ready, Steady . . . Code!

from BandCard.maude

```
op  getName : BandCard  \rightarrow  String .
eq  getName(newBandCard(S, I, J, K, L, M))  =  S .
```

is implemented as

```java
public String getName() {
    return name;
}
```
Ready, Steady . . . Code!

From BandCard.maude

\textbf{op} \texttt{getVolume : BandCard -> Int} .

\textbf{eq} \texttt{getVolume(newBandCard(S, I, J, K, L, M))} = I .

is implemented as

\texttt{method Bandcard\#getVolume()}

\begin{verbatim}
public int getVolume() {
    return volume;
}
\end{verbatim}

etc.
Ready, Steady ... Code!

```
from BandCard.maude

op getVolume : BandCard → Int .
eq getVolume(newBandCard(S, I, J, K, L, M)) = I .
```

is implemented as

```
method Bandcard#getVolume()

public int getVolume() {
    return volume;
}
```

etc.
Ready, Steady ... Code!

from BandCard.maude

```
op getVolume : BandCard -> Int .
eq getVolume(newBandCard(S, I, J, K, L, M)) = I .
```

is implemented as

```
public int getVolume() {
    return volume;
}
```

etc.
Ready, Steady . . . Code!

from BandCard.maude

\textbf{op} \textit{getVolume} : \textit{BandCard} \rightarrow \textit{Int} .

is implemented as

\textbf{method} Bandcard\#\textit{getVolume}()

\texttt{public int \textit{getVolume}() {}

\hspace{1em} \texttt{return \textit{volume};}

\}

etc.
from BandCard.maude

\textbf{op} \textit{getVolume} : \textit{BandCard} \rightarrow \textit{Int} .


is implemented as

\textbf{method} \textit{Bandcard#getVolume}()

\text{public int getVolume() { \}
\text{    return volume; \}
\text{}}

\text{etc.}
Ready, Steady . . . Code!

from BandCard.maude

```plaintext
op getVolume : BandCard → Int.
eq getVolume(newBandCard(S, I, J, K, L, M))  =  I.
```

is implemented as

```java
method Bandcard#getVolume()

public int getVolume() {
    return volume;
}
```

e etc.
We use Maude to specify ADTS:

- the abstract data values — terms
- the operations:

```maude
example operation from BandCard.maude

op newBandCard : String Int Int Int Int Int Int -> BandCard .
var S : String .
vars I J K L M : Int .
eq getName(newBandCard(S, I, J, K, L, M)) = S .
```

- their names
- the types of their arguments
- their return types
- and their behaviour — equations.
We use Maude to specify ADTS:

- the abstract data values — terms
- the operations:

```maude
do

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

var S : String.
vars I J K L M : Int.

eq getName(newBandCard(S, I, J, K, L, M)) = S.
```

- their names
- the types of their arguments
- their return types
- and their behaviour — equations.
Testing, testing

We use Maude to specify ADTS:

- the abstract data values — terms
- the operations:

example operation from BandCard.maude

```
op newBandCard : String Int Int Int Int Int Int -> BandCard .
var S : String .
vars I J K L M : Int .
eq getName(newBandCard(S, I, J, K, L, M)) = S .
```

- their names
- the types of their arguments
- their return types
- and their behaviour — equations.
Testing, testing

We use Maude to specify ADTS:

- the abstract data values — terms
- the operations:

```
example operation from BandCard.maude

op newBandCard : String Int Int Int Int Int Int --> BandCard.
var S : String.
vars I J K L M : Int.
eq getName(newBandCard(S, I, J, K, L, M)) = S.
```

- their names
- the types of their arguments
- their return types
- and their behaviour — equations.
We use Maude to specify ADTS:

- the abstract data values — terms
- the operations:

  example operation from BandCard.maude

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

  \[
  \text{var S : String .}
  \]

  \[
  \text{vars I J K L M : Int .}
  \]

  \[
  \text{eq getName(newBandCard(S, I, J, K, L, M)) } = \text{ S .}
  \]

- their names
- the types of their arguments
- their return types
- and their behaviour — equations.
We use Maude to specify ADTS:

- the abstract data values — terms
- the operations:

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

var S : String.
vars I J K L M : Int.

eq getName(newBandCard(S, I, J, K, L, M)) = S.
```

- their names
- the types of their arguments
- their return types
- and their behaviour — equations.
We use Maude to specify ADTS:

- the abstract data values — terms
- the operations:

```
example operation from BandCard.maude

op newBandCard : String Int Int Int Int Int Int -> BandCard.
var S : String.
vars I J K L M : Int.
eq getName(newBandCard(S, I, J, K, L, M)) = S.
```

- their names
- the types of their arguments
- their return types
- and their behaviour — equations.
In other words, Maude is used to specify (very precisely) *what is to be implemented* and Java is used to actually *implement it*.

Both languages are *executable*:
- we run Java programs; and
- we reduce Maude terms

So (if we get everything right) Maude reductions should be the touchstone for what our Java programs do.
In other words, Maude is used to specify (very precisely) \textit{what is to be implemented}
and Java is used to actually \textit{implement it}.

Both languages are \textit{executable}:

- we run Java programs; and
- we reduce Maude terms

So (if we get everything right) Maude reductions should be the touchstone for what our Java programs do.
In other words, Maude is used to specify (very precisely) what is to be implemented and Java is used to actually implement it.

Both languages are executable:

- we run Java programs; and
- we reduce Maude terms

So (if we get everything right) Maude reductions should be the touchstone for what our Java programs do.
In other words, Maude is used to specify (very precisely) what is to be implemented and Java is used to actually implement it.

Both languages are executable:

- we run Java programs; and
- we reduce Maude terms

So (if we get everything right) Maude reductions should be the touchstone for what our Java programs do.
Maude and States

from BandCard.maude

\[
\text{var} \; \text{S} : \text{String} . \\
\text{vars} \; I \; J \; K \; L \; M : \text{Int} .
\]

\[
\text{eq} \; \text{getName} (\text{newBandCard(S, I, J, K, L, M)}) = \text{S} .
\]

means that

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

So let’s use this to test the correctness of our implementation.
Maude and States

from BandCard.maude

\[
\begin{align*}
\textbf{var} & \quad S : \text{String} . \\
\textbf{vars} & \quad I \, J \, K \, L \, M : \text{Int} . \\
\textbf{eq} & \quad \text{getName(newBandCard}(S, I, J, K, L, M)) = S .
\end{align*}
\]

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

So let’s use this to test the correctness of our implementation.
Maude and States

from BandCard.maude

\begin{verbatim}
var S : String.
vars I J K L M : Int.

eq getName(newBandCard(S, I, J, K, L, M)) = S.
\end{verbatim}

means that

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

So let's use this to test the correctness of our implementation.
Testing for Real

in class BandCard

```java
class BandCard {
    private String name;
    private int year;
    private int genre;
    private int rating;
    private int popularity;

    public BandCard(String name, int year, int genre, int rating, int popularity) {
        this.name = name;
        this.year = year;
        this.genre = genre;
        this.rating = rating;
        this.popularity = popularity;
    }

    public String getName() {
        return name;
    }
}
```

```java
class TestBandCard {
    public static void main(String[] args) {
        BandCard bc = new BandCard("Goldfrapp", 5, 3, 7, 17, 12);
        System.out.println(bc.getName());
    }
}
```

Maude remix

```maude
red getName(newBandCard("Goldfrapp", 5, 3, 7, 17, 12)) .
```

short for ‘reduce’

We should see the result "Goldfrapp" for both Java and Maude.
public static void main(String[] args) {
    BandCard bc = new BandCard("Goldfrapp", 5, 3, 7, 17, 12);
    System.out.println(bc.getName());
}

Maude remix
red getName(newBandCard("Goldfrapp", 5, 3, 7, 17, 12)) .

short for 'reduce'
We should see the result "Goldfrapp" for both Java and Maude
in class BandCard

```java
public static void main(String[] args) {
    BandCard bc = new BandCard("Goldfrapp", 5, 3, 7, 17, 12);
    System.out.println(bc.getName());
}
```

Maude remix

```maude
red getName(newBandCard("Goldfrapp", 5, 3, 7, 17, 12)) .
```

short for ‘reduce’

We should see the result "Goldfrapp" for both Java and Maude.
public static void main(String[] args) {
    BandCard bc = new BandCard("Goldfrapp", 5, 3, 7, 17, 12);
    System.out.println(bc.getName());
}

Maude remix
red getName(newBandCard("Goldfrapp", 5, 3, 7, 17, 12)) .

short for ‘reduce’
We should see the result "Goldfrapp" for both Java and Maude
public static void main(String[] args) {
    BandCard bc = new BandCard("Goldfrapp", 5, 3, 7, 17, 12);
    System.out.println(bc.getName());
}

red getName(newBandCard("Goldfrapp", 5, 3, 7, 17, 12)).

short for ‘reduce’
We should see the result "Goldfrapp" for both Java and Maude
public static void main(String[] args) {
    BandCard bc = new BandCard("Goldfrapp", 5, 3, 7, 17, 12);
    System.out.println(bc.getName());
}

Maude remix
red getName(newBandCard("Goldfrapp", 5, 3, 7, 17, 12)).

short for ‘reduce’

We should see the result "Goldfrapp" for both Java and Maude.
public static void main(String[] args) {
    BandCard bc = new BandCard("Goldfrapp", 5, 3, 7, 17, 12);
    System.out.println(bc.getName());
}

red getName(newBandCard("Goldfrapp", 5, 3, 7, 17, 12)) .

short for ‘reduce’
We should see the result "Goldfrapp" for both Java and Maude
public static void main(String[] args) {
    BandCard bc = new BandCard("Goldfrapp", 5, 3, 7, 17, 12);
    System.out.println(bc.getName());
}

red getName(newBandCard("Goldfrapp", 5, 3, 7, 17, 12)) .

short for ‘reduce’
We should see the result "Goldfrapp" for both Java and Maude
A reasonable set of test cases would be:

```java
public static void main(String[] args) {
    BandCard bc = new BandCard("Goldfrapp", 5, 3, 7, 17, 12);
    System.out.println(bc.getName());
    System.out.println(bc.getVolume());
    System.out.println(bc.getAttitude());
    System.out.println(bc.getCool());
    System.out.println(bc.getEclecticism());
    System.out.println(bc.getHair());
}
```
The values we should expect from these test cases are given by Maude reductions:

Maude remix

red getName(newBandCard("Goldfrapp", 5, 3, 7, 17, 12)) .
red getVolume(newBandCard("Goldfrapp", 5, 3, 7, 17, 12)) .
red getAttitude(newBandCard("Goldfrapp", 5, 3, 7, 17, 12)) .
red getCool(newBandCard("Goldfrapp", 5, 3, 7, 17, 12)) .
red getEclecticism(newBandCard("Goldfrapp", 5, 3, 7, 17, 12)) .
red getHair(newBandCard("Goldfrapp", 5, 3, 7, 17, 12)) .
That’s All, Folks!

Summary

- Abstract Data Types
- Maude

Next:

Modifiers and Scope