From Scheme to Java

So far, we've translated data definitions:

; A snake is
; (make-snake sym num sym)
(define-struct snake (name weight food))

⇒

class Snake {
    String name;
    double weight;
    String food;
    Snake(String name, double weight, String food) {
        this.name = name;
        this.weight = weight;
        this.food = food;
    }
}


Functions in Scheme

; A snake is
; (make-snake sym num sym)
(define-struct snake (name weight food))

; snake-lighter? : snake num -> bool
; Determines whether s is < n lbs
(define (snake-lighter? s n)
  (< (snake-weight s) n))

(snake-lighter? (make-snake 'Slinky 10 'rats) 10) "should be" false
(snake-lighter? (make-snake 'Slimey 5 'grass) 10) "should be" false
class Snake {
    String name;
    double weight;
    String food;
    Snake(String name, double weight, String food) {
        this.name = name;
        this.weight = weight;
        this.food = food;
    }

    // Determines whether it's < n lbs
    boolean isLighter(double n) {
        return this.weight < n;
    }
}

new Snake("Slinky", 10, "rats").isLighter(10) "should be" false
class Snake {
    String name;
    double weight;
    String food;
    Snake(String name, double weight, String food) {
        this.name = name;
        this.weight = weight;
        this.food = food;
    }

    // Determines whether it's < n lbs
    boolean isLighter(double n) {
        return this.weight < n;
    }
}

new Snake("Slinky", 10, "rats").isLighter(10)
"should be" false
Comparing just the function and method:

Scheme:

; snake-lighter? : snake num -> bool
; Determines whether s is < n lbs
(define (snake-lighter? s n)
  (< (snake-weight s) n))

Java:

// Determines whether it's < n lbs
boolean isLighter(double n) {
  return this.weight < n;
}
Comparing just the function and method:

Scheme:

; snake-lighter? : snake num -> bool
; Determines whether s is < n lbs
(define (snake-lighter? s n)
  (< (snake-weight s) n))

Java:

// Determines whether it's < n lbs
boolean isLighter(double n) {
  return this.weight < n;
}
Methods in Java

Comparing just the function and method:

Scheme:

```
;; snake-lighter? : num -> bool
;; Determines whether s is < n lbs
(define (snake-lighter? s n)
  (< (snake-weight s) n))
```

Java:

```
// Determines whether it’s < n lbs
boolean isLighter(double n) {
    return this.weight < n;
}
```

All other arguments are explicit, and the type is next to the name, as in double n.
Comparing just the function and method:

Scheme:

; snake-lighter? : snake num -> bool
; Determines whether s is < n lbs
(define (snake-lighter? s n)
  (< (snake-weight s) n))

Java:

// Determines whether it's < n lbs
boolean isLighter(double n) {
  return this.weight < n;
}
Comparing just the function and method:

Scheme:

; snake-lighter? : snake num -> bool
(determine whether it's < n lbs)

Java:

// Determines whether it's < n lbs
boolean isLighter(double n) {
    return this.weight < n;
}
Methods in Java

Comparing just the function and method:

Scheme:

; snake-lighter? : snake num -> bool
; Determines whether s is < n lbs
(define (snake-lighter? s n)
  (< (snake-weight s) n))

Java:

// Determines whether it's < n lbs
boolean isLighter(double n) {
  return this.weight < n;
}
Comparing just the function and method:

Scheme:

\[
\text{; snake-lighter? : snake num -> bool}
\text{; Determines whether s is < n lbs}
\]

\[
\text{(define (snake-lighter? s n)}
\text{  (\< (snake-weight s) n))}
\]

Java:

// Determines whether s is < n lbs

boolean isLighter(double n) {
    return this.weight < n;
}
Comparing just the function and method:

Scheme:

```scheme
; snake-lighter? : snake num -> bool
; Determines whether s is < n lbs
(define (snake-lighter? s n)
  (< (snake-weight s) n))
```

Java:

```java
// Determines whether it's < n lbs
boolean isLighter(double n) {
    return this.weight < n;
}
```

Explicitly designate the result with `return`
Inside the `class` declaration...

```java
// Determines whether it's < n lbs
boolean isLighter(double n) {
    return this.weight < n;
}
```
Methods in Java, Step-by-Step

Inside the `class` declaration...

```java
// Determines whether it's < n lbs
public boolean isLighter(double n) {
    return this.weight < n;
}
```

First the purpose, starting with `//`
Inside the `class` declaration...

```java
// Determines whether it's < n lbs
boolean isLighter(double n) {
    return this.weight < n;
}
```

Then the result type
Inside the `class` declaration...

```java
// Determines whether it's < n lbs
boolean isLighter(double n) {
    return this.weight < n;
}
```

Then the method name (not capitalized, by convention)
Inside the `class` declaration...

```java
// Determines whether it's < n lbs
boolean isLighter(double n) {
    return this.weight < n;
}
```

Start arguments with (`)
Inside the `class` declaration...

```java
// Determines whether it's < n lbs
boolean isLighter(double n) {
    return this.weight < n;
}
```

Arguments except for `this`—use a type for each argument, and separate multiple arguments with ,.
Inside the `class` declaration...

```java
// Determines whether it's < n lbs
boolean isLighter(double n) {
    return this.weight < n;
}
```

End arguments with `)`
Inside the `class` declaration...

```java
// Determines whether it's < n lbs
boolean isLighter(double n) {
    return this.weight < n;
}
```

Then a {
Inside the `class` declaration...

```java
// Determines whether it's < n lbs
boolean isLighter(double n) {
    return this.weight < n;
}
```

Body using Java notation, put `return` before a result.
Inside the `class` declaration...

```java
// Determines whether it's < n lbs
boolean isLighter(double n) {
    return this.weight < n;
}
```

Put `;` after a result
Inside the `class` declaration...

```java
// Determines whether it's < n lbs
boolean isLighter(double n) {
    return this.weight < n;
}
```

End with `}`
Method Calls in Java

Original tests:

Scheme:

```scheme
(scheme-lighter? (make-snake 'Slinky 10 'rats) 10)
"should be" false
```

Java:

```java
new Snake("Slinky", 10, "rats").isLighter(10)
"should be" false
```
Method Calls in Java

Equivalent, using constant definitions:

Scheme:

```
(define SLINKY (make-snake 'Slinky 10 'rats))

(snake-lighter? SLINKY 10)
"should be" false
```

Java:

```
Snake slinky = new Snake("Slinky", 10, "rats");

slinky.isLighter(10)
"should be" false
```
Method Calls in Java

Equivalent, using constant definitions:

Scheme:

```
(define SLINKY (make-snake 'Slinky 10 'rats))

(snake-lighter? SLINKY 10)
"should be" false
```

Java:

```
Snake slinky = new Snake("Slinky", 10, "rats");
```

Constant definition starts with the constant's type
Method Calls in Java

Equivalent, using constant definitions:

Scheme:

```
(define SLINKY (make-snake 'Slinky 10 'rats))
```

```
(snake-lighter? SLINKY 10)
"should be" false
```

Java:

```
Snake slinky = new Snake("Slinky", 10, "rats");
```

```
slinky.isLighter(10)
"should be" false
```
Method Calls in Java

Equivalent, using constant definitions:

Scheme:

\[
\begin{align*}
&\text{(define SLINKY (make-snake 'Slinky 10 'rats))} \\
&(\text{snake-lighter? SLINKY 10})
\end{align*}
\]

"should be" false

Java:

```java
Snake slinky = new Snake("Slinky", 10, "rats");

slinky.isLighter() // Then =
"should be" false
```
Method Calls in Java

Equivalent, using constant definitions:

Scheme:

```
(define SLINKY (make-snake 'Slinky 10 'rats))
(snake-lighter? SLINKY 10)
"should be" false
```

Java:

```java
Snake slinky = new Snake("Slinky", 10, "rats");
slinky.
"should be" false
```
Method Calls in Java

Equivalent, using constant definitions:

Scheme:

\[
\begin{align*}
&(define \text{SLINKY} \ (\text{make-snake} \ '\text{Slinky} \ 10 \ '\text{rats})) \\
&(\text{snake-lighter?} \ \text{SLINKY} \ 10) \\
&"\text{should be}" \ false
\end{align*}
\]

Java:

```java
Snake slinky = new Snake("Slinky", 10, "rats");

slinky.isLighter(10) \\
"should be" false
```

End with ;
Method Calls in Java

Equivalent, using constant definitions:

Scheme:

```
(define SLINKY (make-snake 'Slinky 10 'rats))

(snake-lighter? SLINKY 10)
"should be" false
```

Java:

```java
Snake slinky = new Snake("Slinky", 10, "rats");

slinky.isLighter(10)
"should be" false
```

Method call starts with an expression for the implicit this argument
Method Calls in Java

Equivalent, using constant definitions:

Scheme:

```
(define SLINKY (make-snake 'Slinky 10 'rats))

(snake-lighter? SLINKY 10)
"should be" false
```

Java:

```
Snake slinky = new Snake("Slinky", 10, "rats");

slinky.isLighter(10)
"should be" false

Then .
```
Method Calls in Java

Equivalent, using constant definitions:

Scheme:

```
(define SLINKY (make-snake 'Slinky 10 'rats))
```

```
(snake-lighter? SLINKY 10)
"should be" false
```

Java:

```
Snake slinky = new Snake("Slinky", 10, "rats");
```

```
slinky.isLighter(10)
"should be" false
```

Then the method name
Method Calls in Java

Equivalent, using constant definitions:

Scheme:

```scheme
(define SLINKY (make-snake 'Slinky 10 'rats))

(scheme-lighter? SLINKY 10)
"should be" false
```

Java:

```java
Snake slinky = new Snake("Slinky", 10, "rats");

slinky.isLighter(10)
"should be" false
```
Method Calls in Java

Equivalent, using constant definitions:

Scheme:

```
(define SLINKY (make-snake 'Slinky 10 'rats))

(snake-lighter? SLINKY 10)
"should be" false
```

Java:

```java
Snake slinky = new Snake("Slinky", 10, "rats");

slinky.isLighter(10)
"should be" false
```

Then expressions for the explicit arguments separated by ,
Method Calls in Java

Equivalent, using constant definitions:

Scheme:

```
(define SLINKY (make-snake 'Slinky 10 'rats))

(snake-lighter? SLINKY 10)
"should be" false
```

Java:

```
Snake slinky = new Snake("Slinky", 10, "rats");

slinky.isLighter(10)
"should be" false
```

Then
Templates

In Scheme:

; A snake is
; (make-snake sym num sym)
(define-struct snake (name weight food))

; func-for-snake : snake -> ...
(define (func-for-snake s)
  ... (snake-name s)
  ... (snake-weight s)
  ... (snake-food s) ...)
Same idea works for Java:

class Snake {
    String name;
    double weight;
    String food;
    Snake(String name, double weight, String food) {
        this.name = name;
        this.weight = weight;
        this.food = food;
    }

    ... methodForSnake(...) {
        ... this.name
        ... this.weight
        ... this.food ...
    }
}
More Examples

- Implement a `feed` method for `Snake` which takes an amount of food in pounds and produces a fatter snake
- Implement a `feed` method for `Dillo` and `Ant`
- Implement a `feed` method for `Animal`
Lists in Java

• Translate the `list-of-num` data definition to Java and implement a `length` method