

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
```

Functions in 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;
    }

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

```
new Snake("Slinky", 10, "rats").isLighter(10)
"should be" false
```

Functions in 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;
    }

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

A function becomes a **method** that is in the class

```
new Snake("Slinky", 10, "rats").isLighter(10)
"should be" false
```

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;  
}
```

Methods in Java

Comparing just the function and method:

Scheme:

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

A method in **Snake** has an implicit **Snake this** argument

Java:

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

Methods in Java

Comparing just the function and method:

Scheme:

```
; snake-lighter? :> bool  
; Determines if a snake is lighter than lbs  
(define (snake-lighter? snake lbs)  
  (< (snake-weight snake) lbs))
```

All other arguments are explicit, and the type is next to the name, as in `double n`

Java:

```
// Determines whether a snake is lighter than 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;  
}
```

The result type is
boolean

Methods in Java

Comparing just the function and method:

Scheme:

```
; snake-lighter? : snake num -> bool  
                  < n lbs
```

Since the method takes a **Snake** and **double** and produces a **boolean**, the contract is

Java:

Snake double -> boolean

and we don't write it as a comment

```
// Determine whether it is < 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))
```

Purpose comment is as
in Scheme, but
comments start with //

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 s is lighter than n lbs  
boolean isLighter(Snake s, int n) {  
  return s.weight < n;  
}
```

Instead of
(snake-weight s)
use
this.weight

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:

```
// Determine if lighter (double n)
boolean isLighter(double n) {
    return this.weight < n;
}
```

Explicitly designate the result with
return

Methods in Java, Step-by-Step

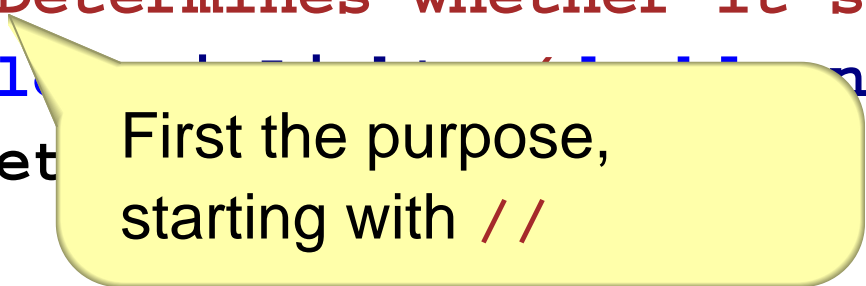
Inside the `class` declaration...

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

Methods in Java, Step-by-Step

Inside the `class` declaration...

```
// Determines whether it's < n lbs  
boolean isLighterThan(int weight, int n) {  
    return weight < n;  
}
```



First the purpose,
starting with `//`

Methods in Java, Step-by-Step

Inside the `class` declaration...

```
// Determines whether it's < n lbs  
boolean isLighter(double n) {  
    return true; //   
}
```

Then the result
type

Methods in Java, Step-by-Step

Inside the `class` declaration...

```
// Determines whether it's < n lbs  
boolean isLighter(double n) {
```

Then the method name (not capitalized, by convention)

Methods in Java, Step-by-Step

Inside the `class` declaration...

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

Start arguments
with (

Methods in Java, Step-by-Step

Inside the `class` declaration...

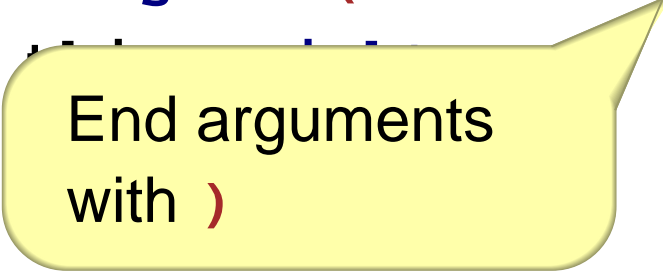
```
// Determines whether it's < n lbs  
boolean isLighter(double n) {  
    // ...  
    ht < n;  
}
```

Arguments
except for `this`
– use a type for
each argument,
and separate
multiple
arguments with `,`

Methods in Java, Step-by-Step

Inside the `class` declaration...

```
// Determines whether it's < n lbs  
boolean isLighter(double n) {  
    return true;  
}
```



End arguments
with)

Methods in Java, Step-by-Step

Inside the `class` declaration...

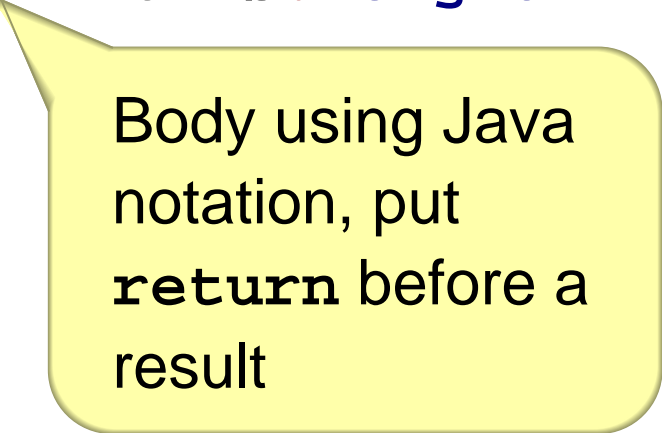
```
// Determines whether it's < n lbs  
boolean isLighter(double n) {  
    return true;  
}
```

Then a {

Methods in Java, Step-by-Step

Inside the `class` declaration...

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



Body using Java notation, put `return` before a result

Methods in Java, Step-by-Step

Inside the `class` declaration...

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




Put `;` after a
result

Methods in Java, Step-by-Step

Inside the `class` declaration...

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



End with }

Method Calls in Java

Original tests:

Scheme:

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

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");
```

```
slinky
```

```
"shoul
```

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
```

Then the name

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.
```

```
"should
```

Then an
expression

Method Calls in Java

Equivalent, using constant definitions:

Scheme:

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

```
(snake-lighter? SLINKY 10)
```

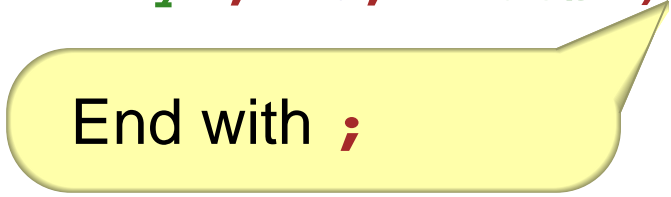
```
"should be" false
```

Java:

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

```
slinky.isLighter(10)
```

```
"should be" false
```



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:

```
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)
```

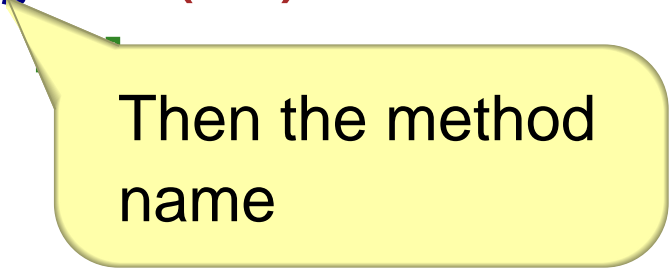
```
"should be" false
```

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:

```
(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)
```

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) ...)
```

Templates

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