From Racket 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 Racket

```
; 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))

(check-expect
 (snake-lighter? (make-snake 'Slinky 10 'rats) 10)
 false)
(check-expect
 (snake-lighter? (make-snake 'Slimey 5 'grass) 10)
 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;
  }
}

t.checkExpect(new Snake("Slinky", 10, "rats").isLighter(10),
              false);
```

Functions in Java

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

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

t.checkExpect(new Snake("Slinky", 10, "rats").isLighter(10),
              false);
```

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

Methods in Java

Comparing just the function and method:

Racket:

```
; 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:

Racket:

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

Java:

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

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

Methods in Java

Comparing just the function and method:

Racket:

```
; snake-lighte                    -> bool
;  Determines                     n lbs
(define (snake
  (< (snake-we
```

Java:

```
// Determines                     < 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`

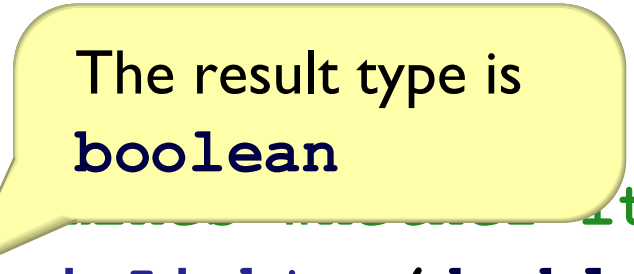
Methods in Java

Comparing just the function and method:

Racket:

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

Java:




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

Methods in Java

Comparing just the function and method:

Racket:

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

Since the method takes a **Snake** and **double** and produces a **boolean**, the contract is **Snake double -> boolean** and we don't write it as a comment

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

Methods in Java

Comparing just the function and method:

Racket:

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

Java:

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

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

Methods in Java

Comparing just the function and method:

Racket:

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

Java:

```
// Determines whetl
boolean isLighter(
  return this.weight < n;
}
```

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

Methods in Java

Comparing just the function and method:

Racket:

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

Java:

```
// Dete
boolean
  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
bool                              n) {
   ret
}
```

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 thisWeight < n;
}
```

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 this.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) {
  return this.weight < 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 this.weight <
}
```

End arguments with)

Methods in Java, Step-by-Step

Inside the `class` declaration...

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

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:

Racket:

```
(check-expect
 (snake-lighter? (make-snake 'Slinky 10 'rats) 10)
 false)
```

Java:

```
t.checkExpect(new Snake("Slinky", 10, "rats").isLighter(10),
              false);
```

Method Calls in Java

Equivalent, using constant definitions:

Racket:

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

(check-expect (snake-lighter? slinky 10)
              false)
```

Java:

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

t.checkExpect(slinky.isLighter(10), false);
```

Method Calls in Java

Equivalent, using constant definitions:

Racket:

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

(check-expect (snake-lighter? slinky 10)
              false)
```

Java:

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

t.chec                                ighter(10), false);
```

Constant definition starts with the constant's type

Method Calls in Java

Equivalent, using constant definitions:

Racket:

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

(check-expect (snake-lighter? slinky 10)
              false)
```

Java:

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

t.checkExp                                ter(10), false);
```

Then the name

Method Calls in Java

Equivalent, using constant definitions:

Racket:

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

(check-expect (snake-lighter? slinky 10)
              false)
```

Java:

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

t.checkExpect(sl                    , false);
```

Then =

Method Calls in Java

Equivalent, using constant definitions:

Racket:

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

(check-expect (snake-lighter? slinky 10)
              false)
```

Java:

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

t.check                          ghter(10), false);
```

Then an expression

Method Calls in Java

Equivalent, using constant definitions:

Racket:

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

(check-expect (snake-lighter? slinky 10)
              false)
```

Java:

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

t.checkExpect(slinky.isLigh
```

End with ;

Method Calls in Java

Equivalent, using constant definitions:

Racket:

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

(check-expect (snake-lighter? slinky 10)
              false)
```

Java:

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

t.checkExpect(slinky.isLighter(10), false);
```

`t.checkExpect` in **`tests`** of an **`Examples`** class starts a test using the **`tester`** library

Method Calls in Java

Equivalent, using constant definitions:

Racket:

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

(check-expect (snake-lighter? slinky 10)
              false)
```

Java:

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

t.checkExpect(slinky.isLighter(10), false);
```

Then (

Method Calls in Java

Equivalent, using constant definitions:

Racket:

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

(check-expect (snake-lighter? slinky 10)
              false)
```

Java:

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

t.checkExpect(slinky.isLighter(10), false);
```

Method call starts with an expression for the implicit **`this`** argument

Method Calls in Java

Equivalent, using constant definitions:

Racket:

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

(check-expect (snake-lighter? slinky 10)
              false)
```

Java:

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

t.checkExpect(slinky.isLighter(10), false);
```

Then .

Method Calls in Java

Equivalent, using constant definitions:

Racket:

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

(check-expect (snake-lighter? slinky 10)
              false)
```

Java:

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

t.checkExpect(slinky.isLighter(10), false);
```

Then the method name

Method Calls in Java

Equivalent, using constant definitions:

Racket:

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

(check-expect (snake-lighter? slinky 10)
              false)
```

Java:

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

t.checkExpect(slinky.isLighter(10), false);
```

Then (

Method Calls in Java

Equivalent, using constant definitions:

Racket:

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

(check-expect (snake-lighter? slinky 10)
              false)
```

Java:

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

t.checkExpect(slinky.isLighter(10), false);
```

Then expressions for the explicit arguments separated by `,`

Method Calls in Java

Equivalent, using constant definitions:

Racket:

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

(check-expect (snake-lighter? slinky 10)
              false)
```

Java:

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

t.checkExpect(slinky.isLighter(10), false);
```

Then) to end the method call

Method Calls in Java

Equivalent, using constant definitions:

Racket:

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

(check-expect (snake-lighter? slinky 10)
              false)
```

Java:

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

t.checkExpect(slinky.isLighter(10), false);
```

Then , after the expression to test

Method Calls in Java

Equivalent, using constant definitions:

Racket:

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

(check-expect (snake-lighter? slinky 10)
              false)
```

Java:

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

t.checkExpect(slinky.isLighter(10), false);
```

Then an expression for the expected result

Method Calls in Java

Equivalent, using constant definitions:

Racket:

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

(check-expect (snake-lighter? slinky 10)
              false)
```

Java:

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

t.checkExpect(slinky.isLighter(10), false);
```

Then `);`

Testing Framework

- A file with tests has

```
import tester.*;

class Examples {
  Snake slinky = new Snake("Slinky", 10, "rats");
  ....

  void tests(Tester t) {
    t.checkExpect(slinky.isLighter(10), false);
  }
}
```

- You have to tell your Java environment to start with **`tester.Main`**
- Names that contain **`Examples`** and **`tests`** are magic when you use **`tester.Main`**

Templates

In Racket:

```
; 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

The 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