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 = 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);
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 the snake is lighter than the given number
(define (snake-lighter? s n)
  (< (snake-weight s) n))

Java:

// Determines whether the snake is lighter than the given number
boolean isLighter(double n) {
    return this.weight < n;
}
Methods in Java

Comparing just the function and method:

Racket:
```
; snake-lighter :: number -> boolean
; Determines if this snake is lighter than n lbs
(define (snake-lighter n)
  (< (snake-weight) n))
```

Java:
```
// Determines if this snake is lighter than 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 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:

```racket
: snake-lighter? : snake num -> bool
  < n lbs
  (not (boolean this.weight < n))

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

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.
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 s is < n lbs
(define (snake-lighter? s n)
  (< (snake-weight s) n))

Java:

// Determines whether this snake is lighter than n lbs
boolean isLighter(Snake s, int 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-lighter? s n)
  (< (snake-weight s) n))

Java:

// Determines whether the snake is lighter than n lbs
boolean snakeLighter(double n) {
  return this.weight < n;
}
Methods in Java, Step-by-Step

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
boolean isLighter(double n) {
    return
}
```

First the purpose, starting with `//`
Methods in Java, Step-by-Step

Inside the `class` declaration...

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

Then the result type
Methods in Java, Step-by-Step

Inside the `class` declaration...

```java
// Determines whether it's < n lbs
boolean isLighter(double n) {
  // method implementation
  return <condition>;
}
```

Then the method name (not capitalized, by convention)
Methods in Java, Step-by-Step

Inside the `class` declaration...

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

Start arguments
with (`

```java
result = isLighter(100); // false
```
Methods in Java, Step-by-Step

Inside the `class` declaration...

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

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

End arguments with `)`
Methods in Java, Step-by-Step

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
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...

```java
// 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...

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

End with `}`
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:

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

Java:

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

Equivalent, using constant definitions:

Racket:

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

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

Java:

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

t.checkLighter(slimly, 10, false);
```

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

Equivalent, using constant definitions:

Racket:

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

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

Java:

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

t.checkException(SlinkyLighterException.class, 10, false);
```

Then the name
Method Calls in Java

Equivalent, using constant definitions:

Racket:

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

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

Java:

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

t.checkExpect(slinky, false);
```
Method Calls in Java

Equivalent, using constant definitions:

Racket:

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

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

Java:

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

Then an expression
```
Method Calls in Java

Equivalent, using constant definitions:

Racket:

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

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

Java:

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

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

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:

\[
\text{(define slinky (make-snake 'Slinky 10 'rats))}
\]

\[
\text{(check-expect (snake-lighter? slinky 10) false)}
\]

Java:

```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:

\[
\text{(define slinky (make-snake 'Slinky 10 'rats))}
\]

\[
\text{(check-expect (snake-lighter? slinky 10)
\qquad \text{false})}
\]

Java:

\[
\text{Snake slinky = new Snake("Slinky", 10, "rats");}
\]

\[
\text{t.checkExpect(slinky.isLighter(10), false);}\]

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:

\[
\text{(define slinky (make-snake 'Slinky 10 'rats))}
\]

\[
\text{(check-expect (snake-lighter? slinky 10)
  false)}
\]

Java:

\[
\text{Snake slinky = new Snake("Slinky", 10, "rats");}
\]

\[
\text{t.checkExpect(slinky.isLighter(10), false)};
\]

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);
```
Testing Framework

• A file with tests has

```java
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:

```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 \texttt{feed} method for \texttt{Snake} which takes an amount of food in pounds and produces a fatter snake.

Implement a \texttt{feed} method for \texttt{Dillo} and \texttt{Ant}.

Implement a \texttt{feed} method for \texttt{Animal}. 
Lists in Java

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