

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)
 true)
```

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

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

```
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  
(define (snake-lighter? s n)  
  (< (snake-weight s) n))
```

Java:

```
// Determines whether  
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-lighter? : bool  
; Determines if a snake is lighter than n 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 if a 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;  
}
```

The result type is
boolean

Methods in Java

Comparing just the function and method:

Racket:

```
: snake-lighter? : snake num -> bool  
< 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-lighter? s n)  
  (< (snake-weight s) n))
```

Purpose comment is as in Racket, 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:

Racket:

```
; 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:

Racket:

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

Java:

```
// Determine if snake is lighter than n lbs
boolean isLighter(Snake s, int 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...

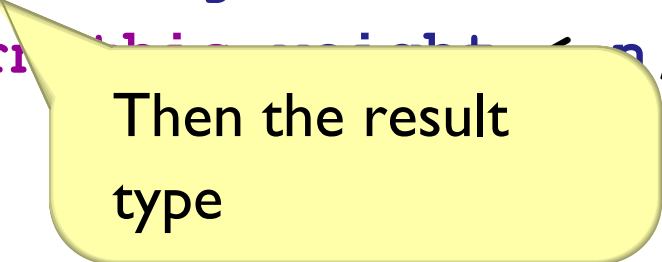
```
// Determines whether it's < n lbs  
boolean isLighter(double n) {  
    return  
}
```

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 this.weight < 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...

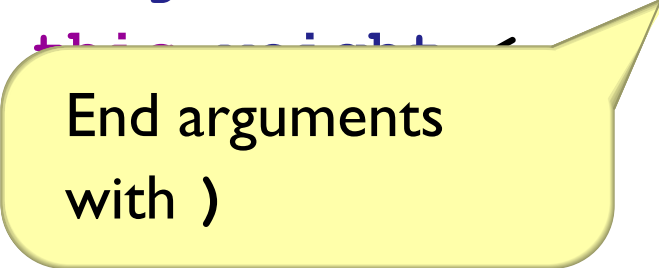
```
// Determines whether it's < n lbs  
boolean isLighter(double n) {  
    return this.weight < n;  
}
```

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 < n;  
}
```



End arguments
with)

Methods in Java, Step-by-Step

Inside the **class** declaration...

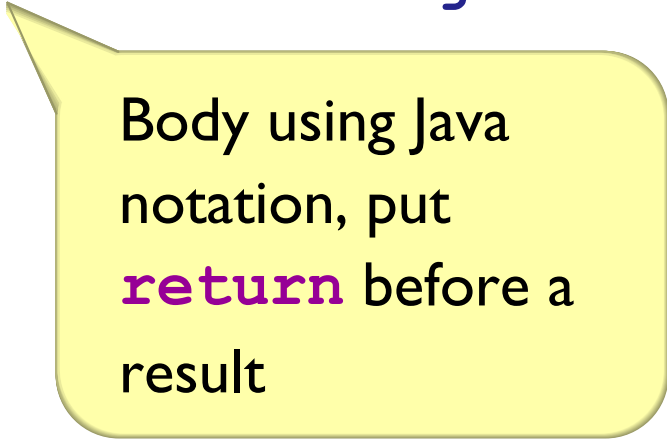
```
// Determines whether it's < n lbs  
boolean isLighter(double n) {  
    return this.weight < n;  
}
```

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;  
}
```

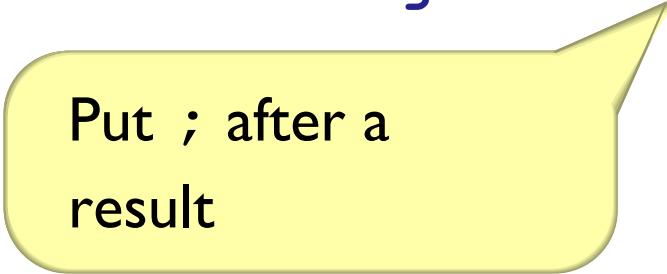


Body using Java notation, put **return** before a result

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.checkLighter(slinky, 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 Then the name ter(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, Then = , 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.check(Then an expression, lighter(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.isLight
```

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

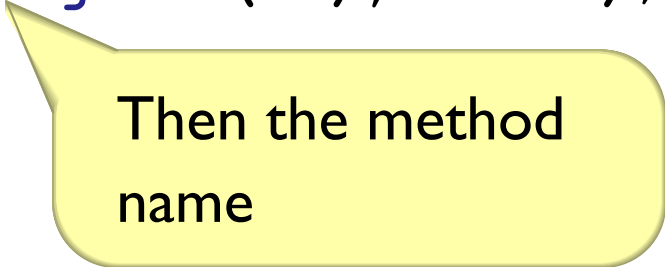
Equivalent, using constant definitions:

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

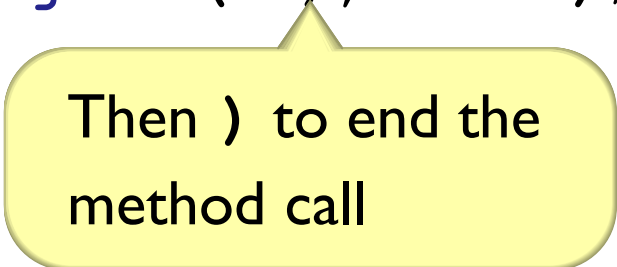
Equivalent, using constant definitions:

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

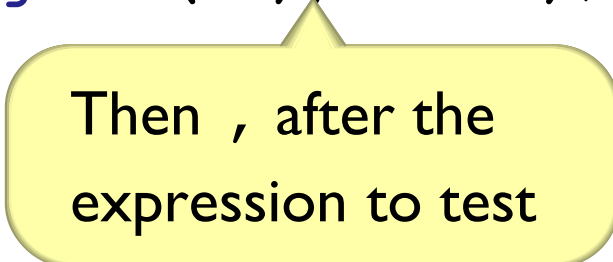
Equivalent, using constant definitions:

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 an **isLighter** method for **Dillo** and **Ant**

Implement a **isLighter** method for **IAnimal**

Functions with Variants

```
; An animal is either
; - snake
; - dillo
; - ant

; animal-is-lighter? : animal num -> bool
(define (animal-is-lighter? a n)
  (cond
    [(snake? a) (snake-is-lighter? s n)]
    [(dillo? a) (dillo-is-ligheter? s n)]
    [(ant? a) (ant-is-lighter? s n)]))

; snake-is-lighter? : snake num -> bool
(define (snake-is-lighter? s n) ...)

; dillo-is-lighter? : dillo num -> bool
(define (dillo-is-lighter? d n) ...)

; ant-is-lighter? : ant num -> bool
(define (ant-is-lighter? a n) ...)
```

Methods with Variants

```
interface IAnimal {
    boolean isLighter(double n);
}

class Snake implements IAnimal {
    ...
    boolean isLighter(double n) { ... }
}

class Dillo implements IAnimal {
    ...
    boolean isLighter(double n) { ... }
}

class Ant implements IAnimal {
    ...
    boolean isLighter(double n) { ... }
}
```

Translating Functions to Methods

```
; An animal is either
; - snake
; - dillo
; - ant

; animal-is-lighter? : animal num -> bool
(define (animal-is-lighter? a n)
  (cond
    [(snake? a) (snake-is-lighter? s n)]
    [(dillo? a) (dillo-is-lighter? s n)]
    [(ant? a) (ant-is-lighter? s n)]))

; snake-is-lighter? : snake num -> bool
(define (snake-is-lighter? s n) ...)

; dillo-is-lighter? : dillo num -> bool
(define (dillo-is-lighter? d n) ...)

; ant-is-lighter? : ant num -> bool
(define (ant-is-lighter? a n) ...)
```

```
interface IAnimal {
  boolean isLighter(double n);
}

class Snake implements IAnimal {
  ...
  boolean isLighter(double n) { ... }
}

class Dillo implements IAnimal {
  ...
  boolean isLighter(double n) { ... }
}

class Ant implements IAnimal {
  ...
  boolean isLighter(double n) { ... }
}
```


Translating Functions to Methods

```
; An animal is either  
; - snake  
; - dillo  
; - ant
```

```
; animal-is-lighter? : animal num -> bool  
(define (animal-is-lighter? a n)  
  (cond  
    [(snake? a) (snake-is-lighter? s n)]  
    [(dillo? a) (dillo-is-lighter? s n)]  
    [(ant? a) (ant-is-lighter? s n)]))
```

```
; snake-is-lighter? : snake num -> bool  
(define (snake-is-lighter? s n) ...)
```

```
; dillo-is-lighter? : dillo num -> bool  
(define (dillo-is-lighter? d n) ...)
```

```
; ant-is-lighter? : ant num -> bool  
(define (ant-is-lighter? a n) ...)
```

Data definition turns into class declarations

```
interface IAnimal {  
  boolean isLighter(double n);  
}
```

```
class Snake implements IAnimal {  
  ...  
  boolean isLighter(double n) { ... }  
}
```

```
class Dillo implements IAnimal {  
  ...  
  boolean isLighter(double n) { ... }  
}
```

```
class Ant implements IAnimal {  
  ...  
  boolean isLighter(double n) { ... }  
}
```

Translating Functions to Methods

```
; An animal is either
; - snake
; - dillo
; - ant

; animal-is-lighter? : animal num -> bool
(define (animal-is-lighter? a n)
  (cond
    [(snake? a) (snake-is-lighter? s n)]
    [(dillo? a) (dillo-is-lighter? s n)]
    [(ant? a) (ant-is-lighter? s n)]))

; snake-is-lighter? : snake num -> bool
(define (snake-is-lighter? s n) ...)

; dillo-is-lighter? : dillo num -> bool
(define (dillo-is-lighter? d n) ...)

; ant-is-lighter? : ant num -> bool
(define (ant-is-lighter? a n) ...)
```

Variant functions turn into variant methods — all with the same contract after the implicit argument

```
interface IAnimal {
  boolean isLighter(double n);
}

class Snake implements IAnimal {
  ...
  boolean isLighter(double n) { ... }
}

class Dillo implements IAnimal {
  ...
  boolean isLighter(double n) { ... }
}

class Ant implements IAnimal {
  ...
  boolean isLighter(double n) { ... }
}
```

Translating Functions to Methods

```
; An animal is either  
; - snake  
; - dillo  
; - ant
```

```
; animal-is-lighter? : animal num -> bool  
(define (animal-is-lighter? a n)  
  (cond  
    [(snake? a) (snake-is-lighter? s n)]  
    [(dillo? a) (dillo-is-lighter? s n)]  
    [(ant? a) (ant-is-lighter? s n)]))
```

```
; snake-is-lighter? : snake num -> bool  
(define (snake-is-lighter? s n) ...)
```

```
; dillo-is-lighter? : dillo num -> bool  
(define (dillo-is-lighter? d n) ...)
```

```
; ant-is-lighter? : ant num -> bool  
(define (ant-is-lighter? a n) ...)
```

Function with variant-based **cond**
turns into just an **interface**
method declaration

```
interface IAnimal {  
  boolean isLighter(double n);  
}  
  
class Snake implements IAnimal {  
  ...  
  boolean isLighter(double n) { ... }  
}  
  
class Dillo implements IAnimal {  
  ...  
  boolean isLighter(double n) { ... }  
}  
  
class Ant implements IAnimal {  
  ...  
  boolean isLighter(double n) { ... }  
}
```

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 **IAnimal**

Lists in Java

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

Lists of Things

```
interface IListOfThing {  
    int length();  
}
```

```
class EmptyListOfThing implements IListOfThing {  
    EmptyListOfThing() { }  
    public int length() { return 0; }  
}
```

```
class ConsListOfThing implements IListOfThing {  
    Thing first;  
    IListOfThing rest;  
    ConsListOfThing(Thing first, IListOfThing rest) {  
        this.first = first;  
        this.rest = rest;  
    }  
    public int length() { return 1 + this.rest.length(); }  
}
```