# Java's Built-in Data Definitions

- **`int`**

```
1    5999    -10
```

- **`double`**

```
1.1    5999.33    -10.01
```

- **`boolean`**

```
true    false
```

- **`String`**

```
"hello"    "See you later!"
```

# Compound Data in Java

Racket:

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

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;
  }
}
```

# Compound Data in Java

Racket:

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

Java:

```
class Snake {
  Str  name;
  doub
  Strin
  Snake                          le weight, String food) {
    thi
    thi
    thi
  }
}
```

**`class`** starts a data definition, or a ***class declaration*** in Java terminology

# Compound Data in Java

Racket:

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

Java:

```
class Snake {
  String na e;
  double we
  String foo
  Snake(Str                eight, String food) {
    this.nam
    this.wei
    this.foo
  }
}
```

Next is the name for the data definition; by convention, the name is captalized

# Compound Data in Java

Racket:

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

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;
  }
}
```

Put { after the name

# Compound Data in Java

Racket:

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

Java:

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

For each part of the compound value, write **type** then **name** then **;**, one line for each part; this is a ***field***

# Compound Data in Java

Racket:

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

Java:

```
class S
  Strin
  doubl
  Stri
  Snake(String name, double weight, String food) {
    this.name = name;
    this.weight = weight;
    this.food = food;
  }
}
```

After the parts, write the defined name again; this starts the ***constructor***

# Compound Data in Java

Racket:

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

Java:

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

Then a (

# Compound Data in Java

Racket:

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

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;
  }
}
```

Write each field again, but this time separate with , — these are the ***constructor arguments***

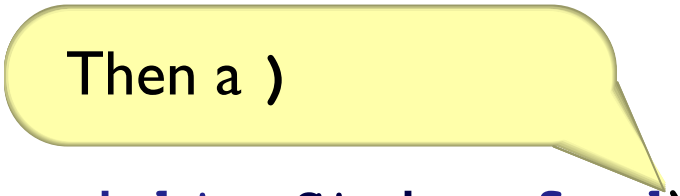
# Compound Data in Java

Racket:

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

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;
  }
}
```



# Compound Data in Java

Racket:

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

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;
  }
}
```

Then a {

# Compound Data in Java

Racket:

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

Java:

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

Each field, one more time... `this` then `.` then `name` then `=` then `name` then `;`

# Compound Data in Java

Racket:

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

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;
  }
}
```

Closing } for the constructor

# Compound Data in Java

Racket:

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

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;
  }
}
```

Closing } for the class declaration

# Instances of Compound Data Types

Racket:

```
(make-snake 'Slinky 12 'rats)
(make-snake 'Slimey 5 'grass)
```

Java:

```
new Snake("Slinky", 12, "rats")
new Snake("Slimey", 5, "grass")
```

# Instances of Compound Data Types

Racket:

```
(make-snake 'Slinky 12 'rats)
(make-snake 'Slimey 5 'grass)
```

Java:

```
new Snake("Slinky", 12, "rats")
new Snake("Slimey", 5, "grass")
```

`new` starts an instance (a value) of a class

# Instances of Compound Data Types

Racket:

```
(make-snake 'Slinky 12 'rats)
(make-snake 'Slimey 5 'grass)
```

Java:

```
new Snake("Slinky", 12, "rats")
new Snake("Slimey", 5, "grass")
```

Next is the class name

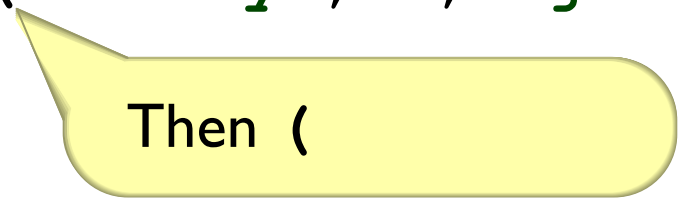
# Instances of Compound Data Types

Racket:

```
(make-snake 'Slinky 12 'rats)
(make-snake 'Slimey 5 'grass)
```

Java:

```
new Snake("Slinky", 12, "rats")
new Snake("Slimey", 5, "grass")
```



# Instances of Compound Data Types

Racket:

```
(make-snake 'Slinky 12 'rats)
(make-snake 'Slimey 5 'grass)
```

Java:

```
new Snake("Slinky", 12, "rats")
new Snake("Slimey", 5, "grass")
```

Then field values separated by ,

# Instances of Compound Data Types

Racket:

```
(make-snake 'Slinky 12 'rats)
(make-snake 'Slimey 5 'grass)
```

Java:

```
new Snake("Slinky", 12, "rats")
new Snake("Slimey", 5, "grass")
```

Then )

# Armadillos

```
class Dillo {
  double weight;
  boolean alive;
  Dillo(double weight, boolean alive) {
    this.weight = weight;
    this.alive = alive;
  }
}

new Dillo(2, true)
new Dillo(3, false)
```

# Posns

```
class Posn {
  int x;
  int y;
  Posn(int x, int y) {
   this.x = x;
   this.y = y;
  }
}

new Posn(0, 0)
new Posn(1, -2)
```

# Ants

```
class Ant {
  double weight;
  Posn loc;
  Ant(double weight, Posn loc) {
    this.weight = weight;
    this.loc = loc;
  }
}

new Ant(0.0001, new Posn(0, 0))
new Ant(0.0002, new Posn(1, -2))
```

# Data with Variants

Racket:

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

Java:

```
interface IAnimal {
}

class Snake implements IAnimal {
  ... as before ...
}
class Dillo implements IAnimal {
  ... as before ...
}
class Ant implements IAnimal {
  ... as before ...
}
```

# Data with Variants

Racket:

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

Java:

```
interface IAnimal {
}

class Snake implements IAnimal {
  ... as before ...
}

class Dillo implements IAnimal {
  ... as before ...
}

class Ant implements IAnimal {
  ... as before ...
}
```

`interface` for a data definition with variants

# Data with Variants

Racket:

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

Java:

```
interface IAnimal {
}

class                                IAnimal {
  ...
}
class                                IAnimal {
  ... as before ...
}
class Ant implements IAnimal {
  ... as before ...
}
```

No fields and no constructor when a class merely groups variants

# Data with Variants

Racket:

```
; An animal is either
;
;
;
```

Java:

```
inter
}
```

Change the class for each variant by adding `implements` then the grouping class name, all before {

```
class Snake implements IAnimal {
  ... as before ...
}
class Dillo implements IAnimal {
  ... as before ...
}
class Ant implements IAnimal {
  ... as before ...
}
```

# Data with Variants

Racket:

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

Java:

```
interface IAnimal {
}

class Snake implements IAnimal {
  ... as before ...
}
cl                    ts IAnimal {
  .
}
class Ant implements IAnimal {
  ... as before ...
}
```

Nothing else changes

# Variants in Java

- A data definition with variants must refer only to other data definitions (which are not built in)

```
; A grade is either
;  - false
;  - num
```

⇒

```
; A grade is either
;  - no-grade
;  - num-grade

; A no-grade is
;  (make-no-grade)
(define-struct no-grade ())

; A num-grade is
;  (make-num-grade num)
(define-struct num-grade (n))
```

- A data definition can be a variant in at most one other data definition