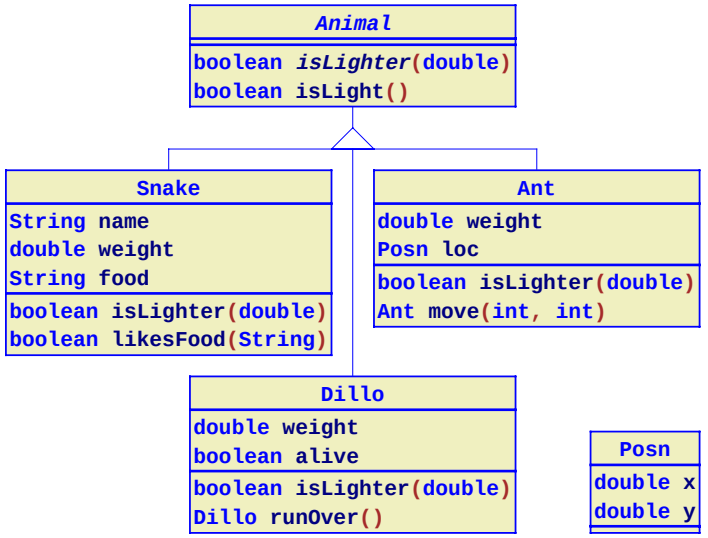
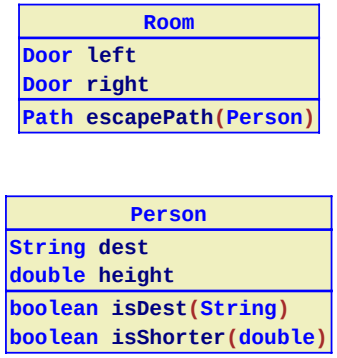
- **Class Diagrams**
- **Nesting Variants to Refine Contracts**
- **Common Functionality in Abstract Classes**
- **Nesting without Abstract**

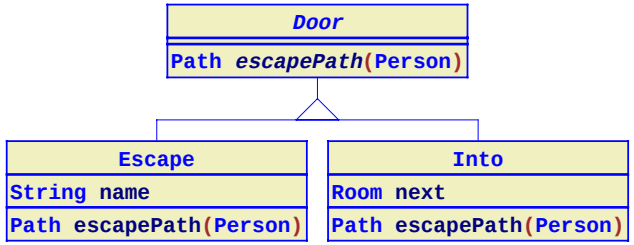
Animal Classes



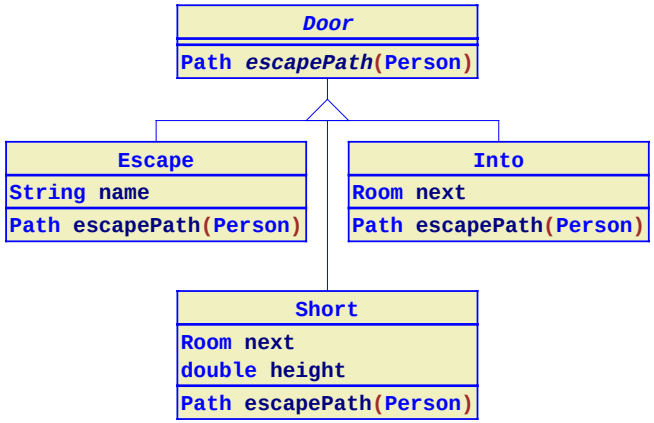
Some Maze Classes



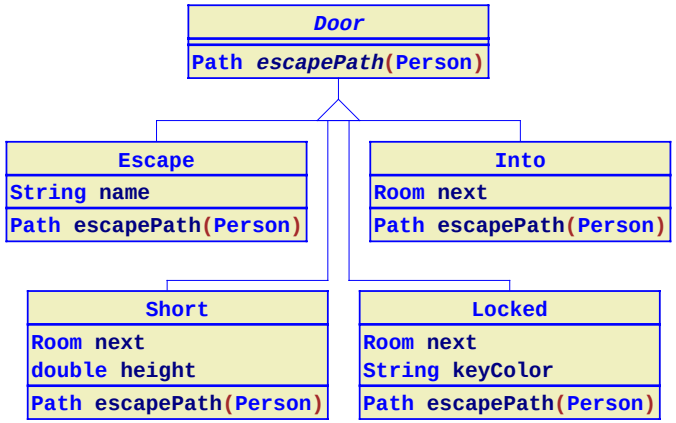
Door Classes



Door Classes

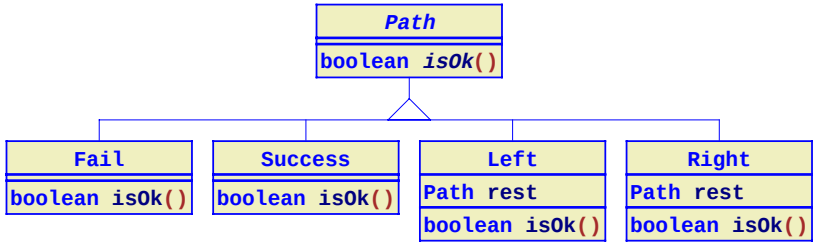


Door Classes



- Class Diagrams
- **Nesting Variants to Refine Contracts**
- Common Functionality in Abstract Classes
- Nesting without Abstract

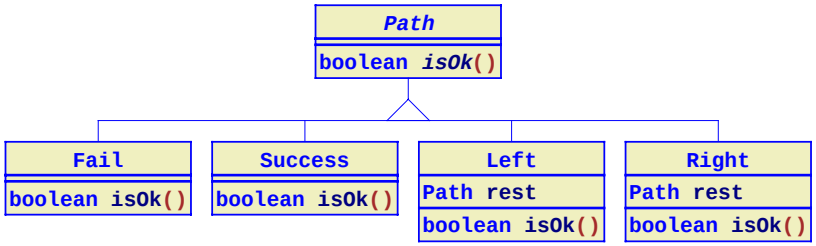
Path Classes



No escape:

```
new Fail()
```

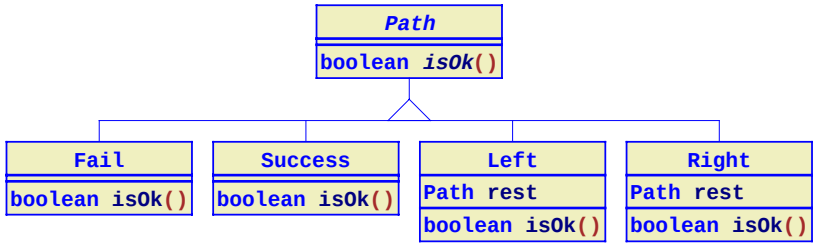
Path Classes



Door is an immediate escape:

```
new Success()
```

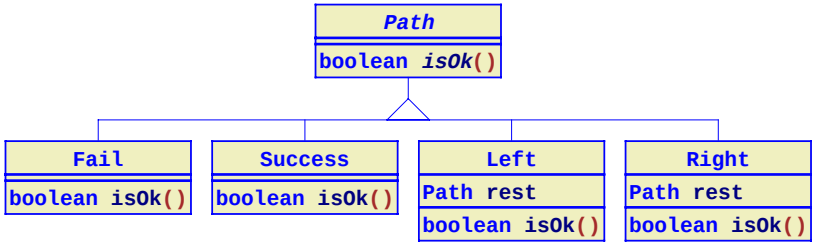
Path Classes



Turn left, then right, then you're there:

```
new Left(new Right(new Success()))
```

Path Classes



What's this?

```
new Left(new Right(new Fail()))
```

We'd prefer to ensure that `Left` and `Right` to extend only successful paths

Paths Reconsidered

Our current definition:

- A path is either
  - failure
  - immediate success
  - left followed by a path
  - right followed by a path

A better definition:

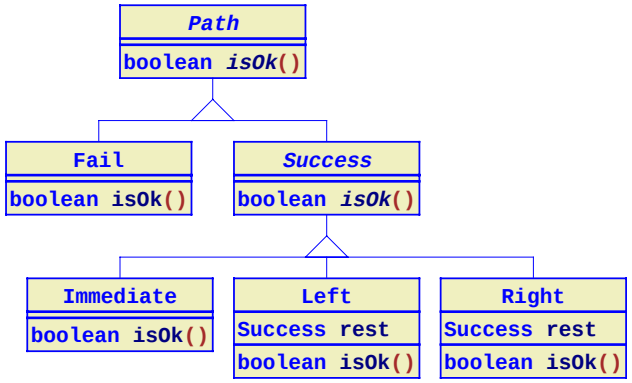
- A path is either
  - failure
  - success
- A success is either
  - immediate
  - left followed by success
  - right followed by success

Nested Variants

- A path is either
  - failure
  - success
- A success is either
  - immediate
  - left followed by success
  - right followed by success

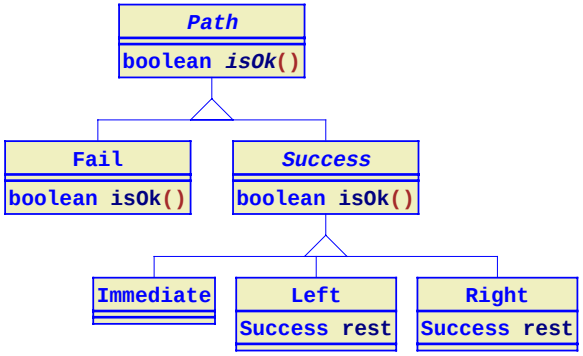
To translate this into Java, a variant of the abstract class **`Path`** must itself be an abstract class with variants

Revised Path Classes



For the **`Success`** classes, the **`isOk`** method always returns **`true`**, so we can simplify...

Revised Path Classes



Revised Path Class Code

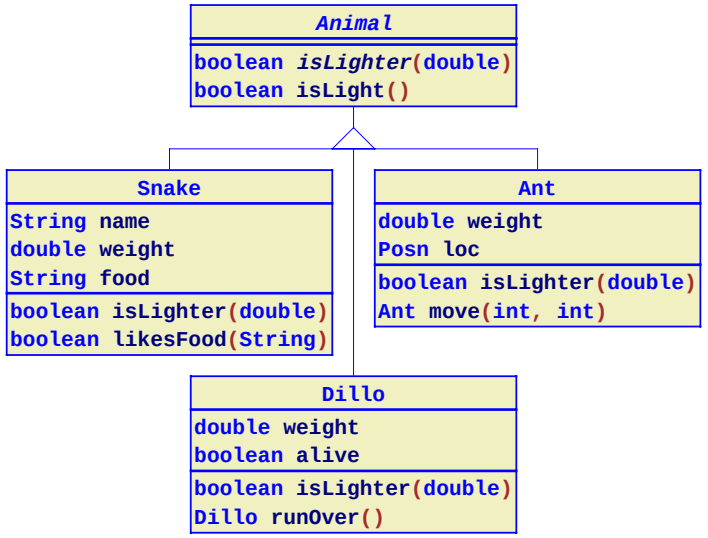
```
abstract class Path {
  abstract boolean isOk();
}

class Fail extends Path {
  Fail() { }
  boolean isOk() { return false; }
}


abstract class Success extends Path {
  boolean isOk() { return true; }
}

class Immediate extends Success {
  Immediate() { }
}

class Right extends Success {
  Success rest;
  Right(Success rest) { this.rest = rest; }
}

class Left extends Success {
  Success rest;
  Left(Success rest) { this.rest = rest; }
}
```

Copy

- Class Diagrams
- Nesting Variants to Refine Contracts
- **Common Functionality in Abstract Classes**
- Nesting without Abstract

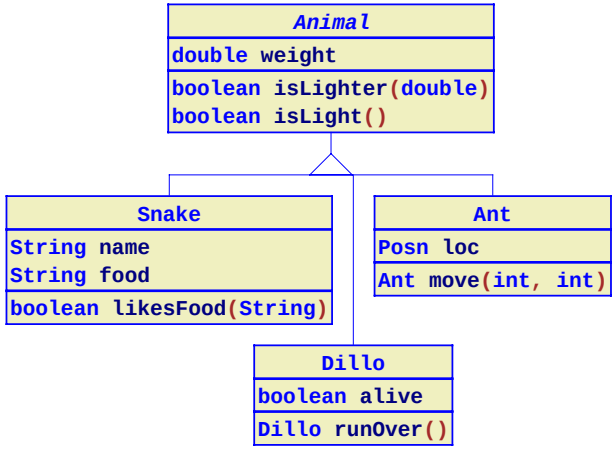
Common Animal Behavior

All animals have a **`weight`** field:



Common Animal Behavior

We can move the common field into the **`Animal`** class:



Fields in Abstract Classes

An **`abstract`** class with a field needs a constructor:

```
abstract class Animal {
  double weight;
  Animal(double weight) {
    this.weight = weight;
  }
  boolean isLighter(int n) {
    return this.weight < n;
  }
  boolean isLight() {
    return this.isLighter(10);
  }
}
```

Copy

Classes that extend a Class with Fields

Extensions of **`Animal`** must now supply the **`super`** class with its field:

```
class Snake extends Animal {
  String name;
  String food;
  Snake(String name, double weight, String food) {
    super(weight);
    this.name = name;
    this.food = food;
  }
  boolean likesFood(String s) {
    return this.food.equals(s);
  }
}
```

Copy

Classes that extend a Class with Fields

Extensions of **`Animal`** must now supply the **`super`** class with its field:

```
class Snake extends Animal {
  String name;
  String food;
  Snake(String name, double weight, String food) {
    super(weight);
    this.
    this.f
  }
  boolean
    return
  }
}
```

The **`super`** keyword in a constructor calls the extended class's constructor

Copy

Classes that extend a Class with Fields

Extensions of **`Animal`** must now supply the **`super`** class with its field:

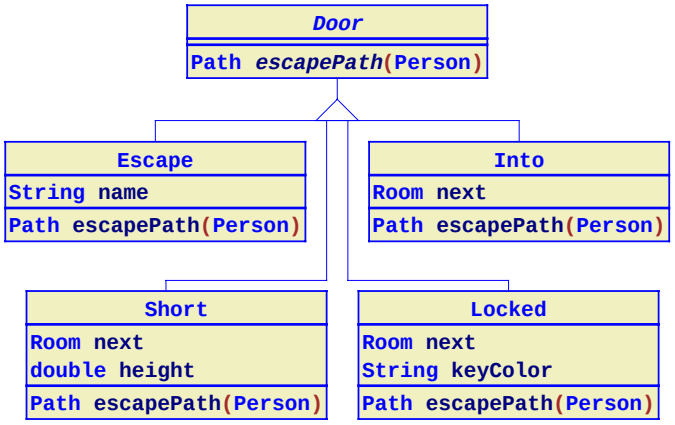
```
class Snake extends Animal {
  String name;
  String food;
  Snake(String name, double weight, String food) {
    super(weight);
    this.
    this.f
  }
  boolean
    return
  }
}
```

A **`super`** call must appear before the others statements

Copy

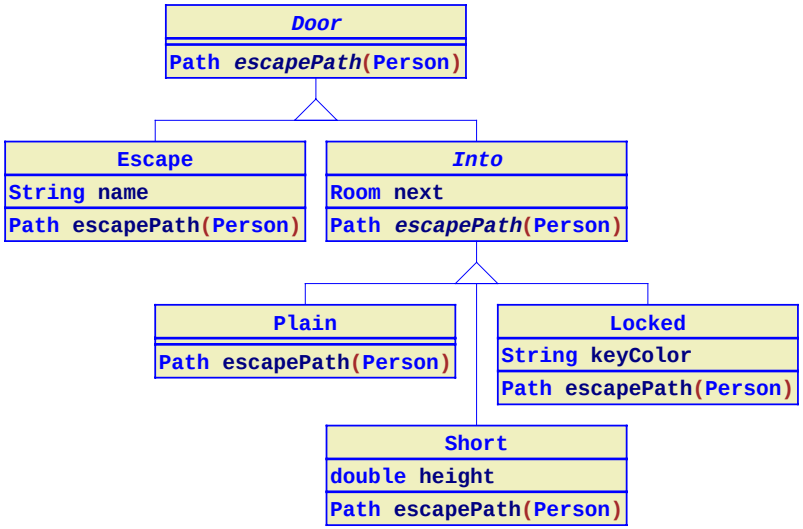
- **Class Diagrams**
- **Nesting Variants to Refine Contracts**
- **Common Functionality in Abstract Classes**
- **Nesting without Abstract**

More Common Features



Most new kinds of door will have a **`next`** field, like **`Into`**

Doors



The **`escapePath`** method isn't always the same, but the **`this.next.escapePath(p)`** part is always the same...

Method Parts in Abstract Classes

```
abstract class Into extends Door {
  Room next;
  Into(Room next) {
    this.next = next;
  }
  Path escapePath(Person p) {
    return this.next.escapePath(p);
  }
}
```

Copy

Note that **`escapePath`** is not **`abstract`**

Chaining to a Super Method

```
class Short extends Into {
  double height;
  Short(Room next, double height) {
    super(next);
    this.height = height;
  }
  Path escapePath(Person p) {
   if (p.height <= this.height)
     return super.escapePath(p);
   else
     return new Fail();
  }
}
```

Copy

Chaining to a Super Method

```
class Short extends Into {
  double height;
  Short(Room next, double height) {
    super(next);
    this.height = height;
  }
  Path escapePath(Person p) {
   if (p.height <= this.height)
     return super.escapePath(p);
   else
     return new Fail();
  }
}
```

Copy

The **`escapePath`** in **`Short`** ***overrides*** the method in **`Into`**

Chaining to a Super Method

```
class Short extends Into {
  double height;
  Short(Room next, double height) {
    if (p.height <= this.height)
      return super.escapePath(p);
    else
      return new Fail();
  }
}
```

Using the **`super`** keyword in **`super.escapePath`** means to call the extended class's method

Copy

The **`escapePath`** in **`Short`** ***overrides*** the method in **`Into`**

Chaining to a Super Method

```
class Short extends Into {
  double height;
  Short(Room next, double height) {
    super(next);
    this.height = height;
  }
  Path escapePath(Person p) {
   if (p.height <= this.height)
     return super.escapePath(p);
   else
     return new Fail();
  }
}
```

Copy

The **`escapePath`** in **`Short`** ***overrides*** the method in **`Into`**

Plain Door

```
class Plain extends Into {
  Plain(Room next) {
    super(next);
  }
  Path escapePath(Person p) {
   return super.escapePath(p);
  }
}
```

Plain Door

```
class Plain extends Into {
  Plain(Room next) {
    super(next);
  }
  Path escapePath(Person p) {
   return super.escapePath(p);
  }
}
```

The overriding **escapePath** merely chains to **super**, so it isn't needed

Plain Door

```
class Plain extends Into {
  Plain(Room next) {
    super(next);
  }
}
```

The overriding **escapePath** merely chains to **super**, so it isn't needed

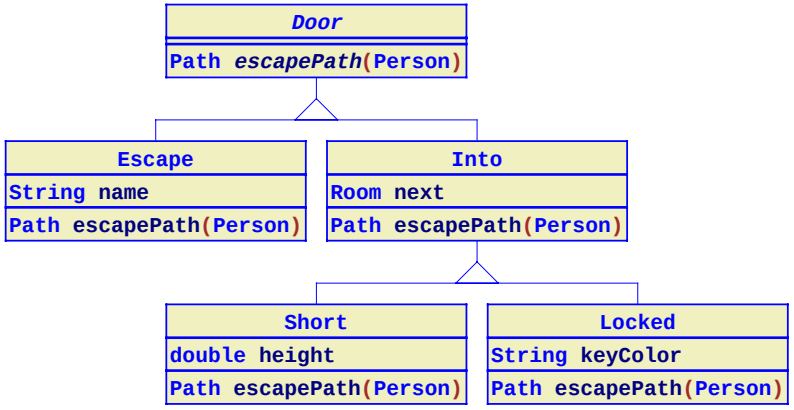
Plain Door

```
class Plain extends Into {
  Plain(Room next) {
    super(next);
  }
}
```

The overriding **escapePath** merely chains to **super**, so it isn't needed

In fact, we can do away with the **Plain** class completely, and just make **Into** non-**abstract**

Doors Revised



Summary

- An **`abstract class`** can extend an **`abstract class`**
- An **`abstract class`** can declare fields
- A **`class`** can extend a **`class`**
- Use **`super(...)`** when the extended class has a constructor
- Use **`super.method(...)`** to chain to an overridden method