Nesting Variants to Refine Contracts

Common Functionality in Abstract Classes

Nesting without Abstract
Path Classes

```
IPath
booleanisOk()
```

Fail

```
booleanisOk()
```

Success

```
booleanisOk()
```

Left

```
IPath rest
booleanisOk()
```

Right

```
IPath rest
booleanisOk()
```

No escape:

```
new Fail()
```
Path Classes

```
<table>
<thead>
<tr>
<th>Class</th>
<th>Method</th>
</tr>
</thead>
<tbody>
<tr>
<td>IPath</td>
<td>Boolean</td>
</tr>
<tr>
<td></td>
<td>isOk()</td>
</tr>
<tr>
<td>Fail</td>
<td>Boolean</td>
</tr>
<tr>
<td></td>
<td>isOk()</td>
</tr>
<tr>
<td>Success</td>
<td>Boolean</td>
</tr>
<tr>
<td></td>
<td>isOk()</td>
</tr>
<tr>
<td>Left</td>
<td>IPath</td>
</tr>
<tr>
<td></td>
<td>rest</td>
</tr>
<tr>
<td></td>
<td>Boolean</td>
</tr>
<tr>
<td></td>
<td>isOk()</td>
</tr>
<tr>
<td>Right</td>
<td>IPath</td>
</tr>
<tr>
<td></td>
<td>rest</td>
</tr>
<tr>
<td></td>
<td>Boolean</td>
</tr>
<tr>
<td></td>
<td>isOk()</td>
</tr>
</tbody>
</table>
```

Door is an immediate escape:

```
new Success()
```
Path Classes

```
IPath
  boolean isOk()

Fail
  boolean isOk()

Success
  boolean isOk()

Left
  IPath rest
  boolean isOk()

Right
  IPath rest
  boolean isOk()
```

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 interface `IPath` must itself be an interface with variants
Revised Path Classes

![Diagram of IPath and ISuccess classes with their isOk() methods.](image-url)
Revised Path Class Code

```java
interface IPath {
    boolean isOk();
}

class Fail implements IPath {
    Fail() { }
    public boolean isOk() { return false; }
}

interface ISuccess extends IPath {
}

class Immediate implements ISuccess {
    Immediate() { }
    public boolean isOk() { return true; }
}

class Right implements ISuccess {
    ISuccess rest;
    Right(ISuccess rest) { this.rest = rest; }
    public boolean isOk() { return true; }
}

class Left implements ISuccess {
    ISuccess rest;
    Left(ISuccess rest) { this.rest = rest; }
    public boolean isOk() { return true; }
}
```
➤ Nesting Variants to Refine Contracts
➤ Common Functionality in Abstract Classes
➤ Nesting without Abstract
Common Animal Behavior

All animals have a **weight** field:

```
IAAnimal
    boolean isLighter(double)
```

**Snake**
- String name
- double weight
- String food
- boolean isLighter(double)
- boolean likesFood(String)

**Ant**
- double weight
- Posn loc
- boolean isLighter(double)
- Ant move(int, int)

**Dillo**
- double weight
- boolean alive
- boolean isLighter(double)
- Dillo runOver()
Common Animal Behavior

Move the common field into the **Animal** abstract class

Also move **isLighter**, since it uses only **weight**

```
IANimal

boolean isLighter(double)

Animal

double weight

boolean isLighter(double)

Snake

String name
String food

boolean likesFood(String)

Dillo

boolean alive
Dillo runOver()

Ant

Posn loc

Ant move(int, int)
```
Interface

An interface:

<table>
<thead>
<tr>
<th>IAnimal</th>
</tr>
</thead>
<tbody>
<tr>
<td>boolean isLighter(double)</td>
</tr>
</tbody>
</table>

- No fields
- Methods declared, but not implemented
- new IAnimal() doesn’t work
- Use with implements

```java
interface IAnimal { ... }

class Snake implements IAnimal { ... }
```
Abstract Class

An **abstract class**:

```
Animal
  double weight
  boolean isLighter(double)
```

- Can have fields
- Methods implemented
- **new Animal()** doesn’t work
- Use with **extends**

```java
abstract class Animal implements IAnimal { ... }

class Snake extends Animal { ... }
```
Fields in Abstract Classes

An **abstract class** needs a constructor:

```java
abstract class Animal implements IAnimal {
    double weight;
    Animal(double weight) {
        this.weight = weight;
    }
    boolean isLighter(int n) {
        return this.weight < n;
    }
}
```
Classes that extend a Class with Fields

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

```java
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);
    }
}
```
Classes that extend a Class with Fields

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

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

The *super* keyword in a constructor calls the extended class’s constructor.
Classes that extend a Class with Fields

Extensions of \textit{Animal} must now supply the \textit{super} class with its field:

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

A \textit{super} call must appear before the other statements
➤ 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**
The `escapePath` method isn’t always the same, but the `this.next.escapePath(p)` part is always the same...
Method Parts in Abstract Classes

```java
abstract class Into extends Door {
    Room next;
    Into(Room next) {
        this.next = next;
    }
    Path escapePath(Person p) {
        return this.next.escapePath(p);
    }
}
```
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.isShorter(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.isShorter(this.height))
            return super.escapePath(p);
        else
            return new Fail();
    }
}

The escapePath in Short overrides the method in Into
Chaining to a Super Method

```java
class Short extends Into {
    double height;
    Short(Room next, double height) {
        Using the `super` keyword in `super.escapePath` means to call the extended class’s method
        if (p instanceof this.height)
            return super.escapePath(p);
        else
            return new Fail();
    }
}
```

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.isShorter(this.height))
            return super.escapePath(p);
        else
            return new Fail();
    }
}

The escapePath in Short overrides the method in Into
class Plain extends Into {
    Plain(Room next) {
        super(next);
    }

    Path escapePath(Person p) {
        return super.escapePath(p);
    }
}

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
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
Summary

• An **interface** can extend an **interface**

• An **abstract class** can implement an **interface**

• 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`