Class Diagrams

- Nesting Variants to Refine Contracts
- Common Functionality in Abstract Classes
- Nesting without Abstract
Animal Classes

- **Animal**
  - `boolean isLighter(double)`
  - `boolean isLight()`

- **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()`

- **Posn**
  - `double x`
  - `double y`
## Some Maze Classes

<table>
<thead>
<tr>
<th>Room</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Door left</td>
<td></td>
</tr>
<tr>
<td>Door right</td>
<td></td>
</tr>
<tr>
<td>Path escapePath(Person)</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Person</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>String dest</td>
<td></td>
</tr>
<tr>
<td>double height</td>
<td></td>
</tr>
<tr>
<td>boolean isDest(String)</td>
<td></td>
</tr>
<tr>
<td>boolean isShorter(double)</td>
<td></td>
</tr>
</tbody>
</table>
Door Classes

- **Door**
  - Path `escapePath(Person)`

- **Escape**
  - String `name`
  - Path `escapePath(Person)`

- **Into**
  - Room `next`
  - Path `escapePath(Person)`
Door Classes

- **Door**
  - Path `escapePath(Person)`

- **Escape**
  - String `name`
  - Path `escapePath(Person)`

- **Into**
  - Room `next`
  - Path `escapePath(Person)`

- **Short**
  - Room `next`
  - Double `height`
  - Path `escapePath(Person)`
Door Classes

Door

Path escapePath(Person)

Escape

String name
Path escapePath(Person)

Into

Room next
Path escapePath(Person)

Short

Room next
double height
Path escapePath(Person)

Locked

Room next
String keyColor
Path escapePath(Person)
➢ Class Diagrams

➢ Nesting Variants to Refine Contracts

➢ Common Functionality in Abstract Classes

➢ Nesting without Abstract
Path Classes

Path

boolean isOk()

Fail

boolean isOk()

Success

boolean isOk()

Left

Path rest

boolean isOk()

Right

Path rest

boolean isOk()

No escape:

new Fail()
Path Classes

```
Path
  boolean isOk()

  Fail
    boolean isOk()

  Success
    boolean isOk()

  Left
    Path rest
    boolean isOk()

  Right
    Path rest
    boolean isOk()
```

Door is an immediate escape:

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

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

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

Path

boolean isOk()
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; }
}
Class Diagrams

Nesting Variants to Refine Contracts

Common Functionality in Abstract Classes

Nesting without Abstract
Common Animal Behavior

All animals have a **weight** field:

```
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

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

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

```
Snake
  String name
  String food
  boolean likesFood(String)
```

```
Ant
  Posn loc
  Ant move(int, int)
```

```
Dillo
  boolean alive
  Dillo runOver()
```
Fields in Abstract Classes

An abstract class with a field needs a constructor:

```java
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);
    }
}
```
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.name = name;
        this.food = food;
    }
    boolean likesFood(String s) {
        return this.food.equals(s);
    }
}

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

Extensions of \texttt{Animal} must now supply the \texttt{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);
    }
}
```

A \texttt{super} call must appear before the others statements.
- 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`
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);
    }
}
```

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();
    }
}
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();
    }
}

The escapePath in Short overrides the method in Into
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();
  }
}

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

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) {
        super(next);
        this.height = height;
    }
    Path escapePath(Person p) {
        if (p.height <= 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
Doors Revised

```
Door
Path escapePath(Person)

Escape
String name
Path escapePath(Person)

Into
Room next
Path escapePath(Person)

Short
double height
Path escapePath(Person)

Locked
String keyColor
Path escapePath(Person)
```
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