A maze consists of rooms and doors:

- An door is either
  - a door into a room
  - an escape to a particular place
- A room has two doors, left and right
abstract class Door {
}

class Into extends Door {
    Room next;
    Into(Room next) {
        this.next = next;
    }
}

class Escape extends Door {
    String name;
    Escape(String name) {
        this.name = name;
    }
}
class Room {
    Door left;
    Door right;
    Room(Door left, Door right) {
        this.left = left;
        this.right = right;
    }
}

Room Data Definition

Copy
Factory for Examples

class Factory {
    Factory() { }
    Room Example() {
        Door meadow = new Escape("meadow");
        Door street = new Escape("street");
        Room ms = new Room(meadow, street);
        Room planets = new Room(new Escape("mars"),
                               new Escape("venus"));
        return new Room(new Into(ms),
                         new Into(planets));
    }
}

Local definitions ⇒ Intermediate Java
Finding Paths

- Implement the `Door` method `canEscape` that takes a string and returns a boolean indicating whether an escape with the given name is available.
- Replace the `canEscape` method with a `escapePath` method that takes a string and returns either a path of "left" and "right" leading to the escape, or a failure value.

```java
Path escapePath(String dest)
```
Paths

A path result is either

• failure

• immediate success

• left followed by a (successful) path

• right followed by a (successful) path

We'll need a **Path** abstract class with an **isOk** method
abstract class Path {
    abstract boolean isOk();
}

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

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

class Right extends Path {
    Path rest;
    Right(Path rest) { this.rest = rest; }
    boolean isOk() { return true; }
}

class Left extends Path {
    Path rest;
    Left(Path rest) { this.rest = rest; }
    boolean isOk() { return true; }
}
Door Variations and Person Attributes

Eventually, we want locked doors, short doors, magic doors, and other kinds of doors.

Finding an escape will depend on having keys, being a certain height, etc.

Instead of adding more and more arguments to `escapePath`, let's introduce a `Person` to carry attributes.

- Replace the destination-string argument of `escapePath` with a `Person` argument, where a `Person` has a destination and height.
Short Doors

• Add a new kind of door, a short door, where a person must be less that the door's height to pass

Adding a short door requires only the declaration of a Short class — no other code changes!
Locked Doors

• Add a new kind of door, a locked door, where a person must have a key to pass

Besides adding Locked, we change Person to add the notion of keys to the person

In contrast to adding new variants, adding new operations requires changing the class
Scheme versus Java

Scheme:

- New variant ⇒ change old functions
- New function ⇒ no changes to old code

Java:

- New variant ⇒ no changes to old code
- New method ⇒ change old classes

This is the essential difference between *functional* programming and *object-oriented* programming