

 **Visitors**

 **Classes in Racket**

Functions

One way to define functions outside of the datatype's class:

```
class Count {
    static int countChars(IList<String> sl) {
        if (sl.isEmpty())
            return 0;
        else
            return sl.getFirst().length()
                + countChars(sl.getRest());
    }
}
```

Visitor

A more “purely object-oriented” approach:

```
interface IListVisitor<X> {  
    int visitEmpty();  
    int visitCons(X first, IList<X> rest);  
}
```

```
interface IList<X> { ....  
    int visit(IListVisitor<X> visitor);  
}
```

```
class Empty<X> implements IList<X> { ....  
    int visit(IListVisitor<X> visitor) {  
        return visitor.visitEmpty();  
    }  
}
```

```
class Cons<X> implements IList<X> { ....  
    int visit(IListVisitor<X> visitor) {  
        return visitor.visitCons(this.first, this.rest);  
    }  
}
```

Counting Visitor

```
class CountingVisitor implements IListVisitor<String> {  
    public int visitEmpty() {  
        return 0;  
    }  
  
    public int visitCons(String first, IList<String> rest) {  
        return first.length() + rest.visit(new CountingVisitor());  
    }  
}
```

More Generic Visitor

```
interface IListVisitor<X, Y> {
    Y visitEmpty();
    Y visitCons(X first, IList<X> rest);
}

interface IList<X> { ....
    <Y> Y visit(IListVisitor<X, Y> visitor);
}

class Empty<X> implements IList<X> { ....
    <Y> Y visit(IListVisitor<X, Y> visitor) {
        return visitor.visitEmpty();
    }
}

class Cons<X> implements IList<X> { ....
    <Y> Y visit(IListVisitor<X, Y> visitor) {
        return visitor.visitCons(this.first, this.rest);
    }
}
```

Generic Counting Visitor

```
class CountingVisitor implements IListVisitor<String, Integer> {  
    public Integer visitEmpty() {  
        return 0;  
    }  
  
    public Integer visitCons(String first, IList<String> rest) {  
        return first.length() + rest.visit(new CountingVisitor());  
    }  
}
```

Appending Visitor

```
class AppendingVisitor implements IListVisitor<String, String> {  
    public String visitEmpty() {  
        return "";  
    }  
  
    public String visitCons(String first, IList<String> rest) {  
        return first.concat(rest.visit(new AppendingVisitor()));  
    }  
}
```

Reversing Visitor

Use a field in the visitor for an accumulator:

```
class ReversingVisitor<X> implements IListVisitor<X, IList<X>> {
    IList<X> accum;

    ReversingVisitor(IList<X> accum) {
        this.accum = accum;
    }

    public IList<X> visitEmpty() {
        return accum;
    }

    public IList<X> visitCons(X first, IList<X> rest) {
        IList<X> nextAccum = new Cons<X>(first, this.accum);
        return rest.visit(new ReversingVisitor<X>(nextAccum));
    }
}
```


➤ **Visitors**

➤ **Classes in Racket**

Racket Classes

The full Racket language includes a Java-like class system

```
(define room%  
  (class object%  
    (init-field left  
                right)  
    (super-new)  
  
    (define/public (escape-path player)  
      (define le (send left escape-path player))  
      (define re (send right escape-path player))  
      (cond  
        [le (cons 'left le)]  
        [re (cons 'right re)]  
        [else false])))
```

Racket Mixins

The Racket `class` form is an expression, so it can go inside a function:

```
(define (room-mixin %)  
  (class %  
    (init-field left  
              right)  
    (super-new)  
  
    (define/public (escape-path player)  
      (define le (send left escape-path player))  
      (define re (send right escape-path player))  
      (cond  
        [le (cons 'left le)]  
        [re (cons 'right re)]  
        [else false])))
```