

List of Strings

```
; A list-of-string is either
; - empty
; - (cons string list-of-string)

; length : list-of-string -> num
(define (length los)
  (cond
    [(empty? los) 0]
    [(cons? los) (+ 1 (length (rest los)))])))
```

List of Strings

```
interface IListOfString {  
    int length();  
}
```

```
class EmptyOfString implements IListOfString {  
    EmptyOfString() { }  
    public int length() { return 0; }  
}
```

```
class ConsOfString implements IListOfString {  
    String first;  
    IListOfString rest;  
    ConsOfString(String first, IListOfString rest) {  
        this.first = first;  
        this.rest = rest;  
    }  
    public int length() { return 1 + this.rest.length(); }  
}
```

Lists

```
; A list-of-X is either
; - empty
; - (cons X list-of-X)

; length : list-of-X -> num
(define (length los)
  (cond
    [(empty? los) 0]
    [(cons? los) (+ 1 (length (rest los)))])))
```

Lists

There are two approaches to **list-of-X** in Java:

- Use **Object**
- Use generics

➤ **Everything is an Object**

➤ **Generics**

➤ **Functions in Java**

Object

- The **Object** class is built into Java
- Every class **extends Object**

The declaration

```
class Posn {  
    . . . .  
}
```

is actually a shorthand for

```
class Posn extends Object {  
    . . . .  
}
```

List of Objects

```
interface IList {  
    int length();  
}
```

```
class Empty implements IList {  
    Empty() { }  
    public int length() { return 0; }  
}
```

```
class Cons implements IList {  
    Object first;  
    IList rest;  
    Cons(Object first, IList rest) {  
        this.first = first;  
        this.rest = rest;  
    }  
    public int length() { return 1 + this.rest.length(); }  
}
```

[Copy](#)

Object from a List

```
interface IList {
```

```
    ....
```

```
    Object ref(int n);
```

```
}
```

```
class Empty implements IList {
```

```
    ....
```

```
    public Object ref(int n) { throw new Error("no"); }
```

```
}
```

```
class Cons implements IList {
```

```
    ....
```

```
    public Object ref(int n) {
```

```
        if (n == 0)
```

```
            return this.first;
```

```
        else
```

```
            return this.rest.ref(n - 1);
```

```
    }
```

```
}
```


Object from a List

```
class Posn { .....  
    Posn flip() { ..... }  
}
```

```
Posn park = new Posn(6, 11);  
IList l = new Cons(home, new Empty());
```

```
park.flip();           // ok  
l.ref(0).flip();     // not ok
```

Casts

A **cast** refines the type of an expression:

```
Posn park = new Posn(6, 11);  
IList l = new Cons(home, new Empty());  
  
Posn place = (Posn)l.ref(0);  
place.flip(); // ok
```

Casts

A **cast** refines the type of an expression:

```
Posn park = new Posn(6, 11);  
IList l = new Cons(home, new Empty());
```

```
Posn place = (Posn)l.ref(0);  
place.flip(); // ok
```

A cast is (then
type then) before
an expression

Casts

A **cast** refines the type of an expression:

```
Posn park = new Posn(6, 11);  
IList l = new Cons(home, new Empty());
```

```
Posn place = (Posn)l.ref(0);  
place.flip(); // ok
```

```
(IList)l.ref(0); // run-time error
```

Objects and Numbers

```
IList l = new Cons(7, new Empty());
```

```
l.ref(0); // ok
```

```
(int)l.ref(0); // not ok
```

```
(Integer)l.ref(0); // ok
```

➤ **Everything is an Object**

➤ **Generics**

➤ **Functions in Java**

Lists

We'd like a solution closer to Racket, where the caller gets to pick an **X**:

```
; A list-of-X is either  
; - empty  
; - (cons X list-of-X)
```

```
; length : list-of-X -> num
```

```
(define (length los)
```

```
  (cond
```

```
    [(empty? los) 0]
```

```
    [(cons? los) (+ 1 (length (rest los)))]))
```

Lists

```
interface IList<X> {  
    int length();  
}
```

```
class Empty<X> implements IList<X> {  
    Empty() { }  
    public int length() { return 0; }  
}
```

```
class Cons<X> implements IList<X> {  
    X first;  
    IList<X> rest;  
    Cons(X first, IList<X> rest) {  
        this.first = first;  
        this.rest = rest;  
    }  
    public int length() { return 1 + this.rest.length(); }  
}
```


X from a List

```
interface IList<X> {
    ....
    X ref(int n);
}

class Empty<X> implements IList<X> {
    ....
    public X ref(int n) { throw new Error("no"); }
}

class Cons<X> implements IList<X> {
    ....
    public X ref(int n) {
        if (n == 0)
            return this.first;
        else
            return this.rest.ref(n - 1);
    }
}
```

Posn from a List

```
Posn park = new Posn(6, 11);  
IList<Posn> emptyPL = new Empty<Posn>();  
IList<Posn> l = new Cons<Posn>(home, emptyPL);  
  
park.flip(); // ok  
l.ref(0).flip(); // ok
```

Using generics instead of `Object` is usually better, because a cast is a potential run-time error

- **Everything is an Object**
- **Generics**
- **Functions in Java**

Counting All Characters

Determine the total number of characters in a list of strings

```
IList<String> emptySL
    = new Empty<String>();
IList<String> fruit1
    = new Cons<String>("apple", empty);
IList<String> fruit2
    = new Cons<String>("banana", fruit1);

t.checkexpect(fruit2.countChars(), 11);
```

Counting All Characters

```
interface IList<X> {
    ....
    int countChars();
}

class Empty<X> implements IList<X> {
    ....
    public int countChars() { return 0; }
}

class Cons<X> implements IList<X> {
    ....
    public int countChars() {
        .... this.first ....
        .... this.rest.countChars() ....
    }
}
```

Counting All Characters

```
interface IList<X> {  
    ....  
    int countChars();  
}
```

```
class Empty<X> implements IList<X> {  
    ....  
    public int countChars() { return 0; }  
}
```

```
class Cons<X> implements IList<X> {  
    ....  
    public int countChars() {  
        .... this.first ....  
        .... this.rest.countChars() ....  
    }  
}
```

not necessarily a
String

Counting All Characters

Falling back to Racket (i.e., functional) style:

```
class Count {
  static int countChars(IList<String> sl) {
    if (sl instanceof Empty)
      return 0;
    else {
      Cons<String> c = (Cons<String>)sl;
      return c.first.length() + countChars(c.rest);
    }
  }
}
```

- **static** declares a function instead of a method
- the function name is **Count.countChars**

Counting All Characters

Falling back to Racket (i.e., functional) style:

```
class Count {
  static int countChars(IList<String> sl) {
    if (sl instanceof Empty)
      return 0;
    else {
      Cons<String> c = (Cons<String>)sl;
      return c.first.length() + countChars(c.rest);
    }
  }
}
```

- `instanceof Empty` is analogous to `empty?`
- `c.first` is analogous to `(first c)`

...but both are poor style in Java

Accessors

```
interface IList<X> {  
    ....  
    boolean isEmpty();  
    X getFirst();  
    IList<X> getRest();  
}
```

```
class Empty<X> implements IList<X> {  
    ....  
    public boolean isEmpty() { return true; }  
    public X getFirst() { throw new Error("fail"); }  
    public IList<X> getRest() { throw new Error("fail"); }  
}
```

```
class Cons<X> implements IList<X> {  
    ....  
    public boolean isEmpty() { return false; }  
    public X getFirst() { return this.first; }  
    public IList<X> getRest() { return this.rest; }  
}
```

Counting All Characters

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

... and that's *almost* what Java programmers do