- **Loops**
- **When to Use Loops**

Iterating with Numbers

Many computations involve iterating over numbers:

- Checking each item in an array
- Computing sums or products from `0` to `n`

One way to write such loops:

Step 1. Set `i` to the starting number

Step 2. If `i` is too big, stop

Otherwise, do something with `i`

Step 3. Change `i` to the next value and go to **Step 2**

For Loops

Java supports this pattern with `for`

```
int sum(int n) {
  int res = 0;
  for (int i = 0; i <= n; i = i + 1) {
    res = res + i;
  }
  return res;
}
```

For Loops

Java supports this pattern with `for`

```
int sum(int n) {
  int res = 0;
  for (int i = 0; i <= n; i = i + 1) {
    res = res +
  }
  return res;
}
```

Set `i` to the starting number

For Loops

Java supports this pattern with **for**

```
int sum(int n) {
  int res = 0;
  for (int i = 0; i <= n; i = i + 1) {
    res = res + i;
  }
  return res;
}
```

If **i** isn’t too big...

For Loops

Java supports this pattern with **for**

```
int sum(int n) {
  int res = 0;
  for (int i = 0; i <= n; i = i + 1) {
    res = res + i;
  }
  return res;
}
```

Do something with **i**

For Loops

Java supports this pattern with **for**

```
int sum(int n) {
  int res = 0;
  for (int i = 0; i <= n; i = i + 1) {
    res = res + i;
  }
  return res;
}
```

Change **i** to the next value

Another Example

```
int sumElements(int[] a) {
  int res = 0;
  for (int i = 0; i < a.length(); i = i + 1) {
    res = res + a[i];
  }
  return res;
}
```

Another Example

```
int maxElement(int[] a) {
  int res = a[0];
  for (int i = 1; i < a.length(); i = i + 1) {
    if (res < a[i])
       res = a[i];
  }
  return res;
}
```

Another Example

```
int isArrayMember(Object o, Object[] a) {
  for (int i = 0; i < a.length(); i = i + 1) {
    if (o.equals(a[i]))
      return true;
  }
  return false;
}
```

Looping with Values Other than Numbers

With suitable methods and helpers, the same pattern can work for list-shaped data:

```
int isListMember(Object o, List lst) {
  for (Enumerator e = lst.elements();
       e.hasMoreElements();
       ) {
    Object elem = e.nextElement();
    if (o.equals(elem))
      return true;
  }
  return false;
}
```

While Loops

```
while (test) { ... }
```

is a shorthand for

```
for (; test; ) { ... }
```

```
int isListMember(Object o, List lst) {
  Enumerator e = lst.elements();
  while (e.hasMoreElements()) {
    Object elem = e.nextElement();
    if (o.equals(elem))
      return true;
  }
  return false;
}
```

Do/While Loops

```
do { ... } while (test);
```

is a shorthand for

```
for (boolean ok=true; ok; ) { ... ok = test; }
```

```
int tryUntil(List lst, Tester t) {
  Enumerator e = lst.elements();
  do {
    Object elem = e.nextElement();
  } while (!t.tryIt(elem));
}
```

- Loops
- **When to Use Loops**

  ... and why Java needs a special form for loops

When to Use Loops

Use **`for`**, **`while`** and **`do`** like you would use **`filter`** or **`map`**

- In other words, it's a question of reusing a pattern

Using **`map`** in Scheme is always optional, but sometimes you really *must* use **`for`** in Java

This is a design flaw in Java that you'll have to live with

As someone who knows how to design programs, you should understand

- why **`for`** is necessary
- how to convert recursive programs to use **`for`**

Cost of Computation, Revisited

```
; sum : num -> num
; Sums the numbers from 0 to n
(define (sum n)
  (cond
    [(zero? n) 0]
    [else (+ n (sum (- n 1)))]))
```

How long does **`(sum n)`** take?

$$\mathbf{T}(0) = k_1$$
$$\mathbf{T}(n) = k_2 + \mathbf{T}(n\text{-}1)$$

So it takes $k_1 + k_2 n$, i.e., proportional to n

Cost of Computation, Revisited

```
; sum : num -> num
; Sums the numbers from 0 to n
(define (sum n)
  (cond
    [(zero? n) 0]
    [else (+ n (sum (- n 1)))]))
```

How much *space* does `(sum n)` take?

```
(sum n)
→→(+ n (sum n-1))
→→(+ n (+ n-1 (sum n-2)))
→→→(+ n (+ n-1 (+ n-2 ... 0)))
```

So it takes space proportional to *n*

Cost of Computation with an Accumulator

```
; asum : num num -> num
; Sums the numbers from 0 to n, added to res
(define (asum n res)
  (cond
    [(zero? n) res]
    [else (asum (- n 1) (+ res n))]))
```

How long does `(asum n 0)` take?

Still proportional to *n*

Cost of Computation with an Accumulator

```
; asum : num num -> num
; Sums the numbers from 0 to n, added to res
(define (asum n res)
  (cond
    [(zero? n) res]
    [else (asum (- n 1) (+ res n))]))
```

How much *space* does `(asum n 0)` take?

```
(asum n 0)
→→(asum n-1 n)
→→(asum n-2 2n-1)
→→→(asum 0 n^2/2+n/2)
```

So it takes constant space, independent of *n*

Time and Space

- Weeks ago, we saw how an accumulator can save lots of time
- Less frequently, an assumulator can save space (sometimes even when it saves no time)

Accumulators save space when the result of a recursive call is the result for the current call:

```
(define (asum n res)
  (cond
    [(zero? n) res]
    [else (asum (- n 1) (+ res n))]))
```

As it turns out, recursive-call space in Java tends to be more scarce than other space, so this kind of saving is often important

Cost of Computation in Java

```
class Summer {
  int sum(int n) {
    if (n == 0)
      return 0;
    else
      return n+this.sum(n-1);
  }
}
```

How long does **new** `Summer.sum(n)` take?

Still proportional to *n*

Cost of Computation in Java

```
class Summer {
  int sum(int n) {
    if (n == 0)
      return 0;
    else
      return n+this.sum(n-1);
  }
}
```

How much *space* does **new** `Summer.sum(n)` take?

```
s.sum(n)
→ → return n+s.sum(n-1)
→ → return n+(return n-1+s.sum(n-2))
→ → → return n+(return n-1+(return n-2+...0))
```

Again, space proportional to *n*

Cost of Computation in Java

```
class Summer {
  int asum(int n, int res) {
    if (n == 0)
      return res;
    else
      return this.asum(n-1, res+n);
  }
}
```

How long does **new** `Summer.asum(n, 0)` take?

Still proportional to *n*

Cost of Computation in Java

```
class Summer {
  int asum(int n, int res) {
    if (n == 0)
      return res;
    else
      return this.asum(n-1, res+n);
  }
}
```

How much *space* does **new** `Summer.asum(n, 0)` take?

```
s.asum(n, 0)
→ → return s.asum(n-1, n)
→ → return return s.asum(n-2, 2n-1))
→ → → return return ... return n²/2+n/2
```

Still space proportional to *n*, due to all the `return`*s*

Tail Calls in Java

```
class Summer {
  int asum(int n, int res) {
    if (n == 0)
      return res;
    else
      return this.asum(n-1, res+n);
  }
}
```

The **`return`** explanation reflects the actual semantics of Java: redundant **`return`***s* do not get dropped

- To allow constant-space loops, languages like Java provide a special form
- The special form only works for loops with no arguments

Getting Rid of Arguments

```
(define (sum n)
  (local [(define (asum i res)
            (cond
              [(zero? i) res]
              [else (asum (- i 1) (+ res i))]))]
    (asum n 0)))
```

Equivalent Scheme code (in extremely poor style):

```
(define (sum n)
  (local [(define res 0)
          (define i n)
          (define (asum)
            (cond
              [(zero? i) (void)]
              [else (set! res (+ res i))
                    (set! i (- i 1))
                    (asum)]))]
    (asum)
    res))
```

Loops in Java

```
(define (sum n)
  (local
   [(define res 0)
    (define i n)
    (define (asum)
      (cond
        [(zero? i) (void)]
        [else (set! res (+ res i))
              (set! i (- i 1))
              (asum)]))]
   (asum)
   res))
```

```
class Summer {
  int sum(int n) {
    int res = 0;
    int i = n;
    while (true)
      if (i == 0)
        break;
      else {
        res = res + i;
        i = i - 1;
      }
    }
    return res;
  }
}
```

The **`while`** form is like a recursive function that always either returns void or calls itself with no arguments

Loops in Java, Slightly Better Style

`while (true) { if (`*`test`*`) break; else ... }`
⇒ **`while (`*`test`*`) { ... }`**

```
class Summer {
  int sum(int n) {
    int res = 0;
    int i = n;
    while (true) {
      if (i == 0)
        break;
      else {
        res = res + i;
        i = i - 1;
      }
    }
    return res;
  }
}
```

```
class Summer {
  int sum(int n) {
    int res = 0;
    int i = n;
    while (i == 0) {
      res = res + i;
      i = i - 1;
    }
    return res;
  }
}
```

Loops in Java, Good Style

```
init; while (test) { ... incr; }
⇒ for (init; test; incr) { ... }
```

```
class Summer {
  int sum(int n) {
    int res = 0;
    int i = n;
    while (true) {
      if (i == 0)
        break;
      else {
        res = res + i;
        i = i - 1;
      }
    }
    return res;
  }
}
```

```
class Summer {
  int sum(int n) {
    int res = 0;
    for (int i = n;
         i != 0;
         i = i - 1) {
      res = res + i;
    }
  }
}
```

When Loops Don't Work

Converting tree recusion into a loop usually won't work, because there are multiple recursive calls for each call

Some algorithms, such as **`merge-sort`**, also involve multiple recursive calls

Technically, any program can be converted by manually creating ***continuations***, but that's a topic for CS 3520

Loops in Java

Conclusion:

- Use **`for`** and **`while`** to make your code look and run better
- When in doubt, write the recursive version first