Data Structures and Algorithms in Java

Procedural Programming: Defining Functions

Outline

Function Definitions

Function Definitions

```
public|private static void|<type> <name>(<parameter1>, <parameter2>, ...) {
    <statement>
    ...
}
```

```
return; // to return from void functions

return <expression>; // to return (with a value) from non-void functions
```

Function Definitions · Control Flow

```
1  import stdlib.StdOut;
2
3  public class Square {
4      public static void main(String[] args) {
5          int x = Integer.parseInt(args[0]);
6          int y = square(x);
7          StdOut.println(y);
8      }
9
10     private static int square(int x) {
11         int y = x * x;
12         return y;
13     }
14 }
```

Variable Trace

line #	main::x	main::y	square::x	square::y

>_

```
$ _
```

Function Definitions · Control Flow

```
1  import stdlib.StdOut;
2
3  public class Square {
4      public static void main(String[] args) {
5          int x = Integer.parseInt(args[0]);
6          int y = square(x);
7          StdOut.println(y);
8      }
9
10     private static int square(int x) {
11         int y = x * x;
12         return y;
13     }
14 }
```

Variable Trace

line #	main::x	main::y	square::x	square::y

```
>_
$ java Square 13
```

Function Definitions · Control Flow

```
1  import stdlib.StdOut;
2
3  public class Square {
4      public static void main(String[] args) {
5          int x = Integer.parseInt(args[0]);
6          int y = square(x);
7          StdOut.println(y);
8      }
9
10     private static int square(int x) {
11         int y = x * x;
12         return y;
13     }
14 }
```

Variable Trace

line #	main::x	main::y	square::x	square::y
1				

```
>_
$ java Square 13
```

```
1  import stdlib.StdOut;
2
3  public class Square {
4      public static void main(String[] args) {
5          int x = Integer.parseInt(args[0]);
6          int y = square(x);
7          StdOut.println(y);
8      }
9
10     private static int square(int x) {
11         int y = x * x;
12         return y;
13     }
14 }
```

Variable Trace

line #	main::x	main::y	square::x	square::y
3				

```
$ java Square 13
```

Function Definitions · Control Flow

```
1  import stdlib.StdOut;
2
3  public class Square {
4      public static void main(String[] args) {
5          int x = Integer.parseInt(args[0]);
6          int y = square(x);
7          StdOut.println(y);
8      }
9
10     private static int square(int x) {
11         int y = x * x;
12         return y;
13     }
14 }
```

Variable Trace

line #	main::x	main::y	square::x	square::y
4				

>_

```
$ java Square 13
```

Function Definitions · Control Flow

```
1  import stdlib.StdOut;
2
3  public class Square {
4      public static void main(String[] args) {
5          int x = Integer.parseInt(args[0]);
6          int y = square(x);
7          StdOut.println(y);
8      }
9
10     private static int square(int x) {
11         int y = x * x;
12         return y;
13     }
14 }
```

Variable Trace

line #	main::x	main::y	square::x	square::y
5	13			

```
>_
$ java Square 13
```

Function Definitions · Control Flow

```java
import stdlib.StdOut;

public class Square {
    public static void main(String[] args) {
        int x = Integer.parseInt(args[0]);
        int y = square(x);
        StdOut.println(y);
    }

    private static int square(int x) {
        int y = x * x;
        return y;
    }
}
```

Variable Trace

line #	main::x	main::y	square::x	square::y
6	13			

```
$ java Square 13
```

Function Definitions · Control Flow

```java
import stdlib.StdOut;

public class Square {
    public static void main(String[] args) {
        int x = Integer.parseInt(args[0]);
        int y = square(x);
        StdOut.println(y);
    }

    private static int square(int x) {
        int y = x * x;
        return y;
    }
}
```

Variable Trace

line #	main::x	main::y	square::x	square::y
10	13		13	

```
$ java Square 13
```

Function Definitions · Control Flow

```java
import stdlib.StdOut;

public class Square {
    public static void main(String[] args) {
        int x = Integer.parseInt(args[0]);
        int y = square(x);
        StdOut.println(y);
    }

    private static int square(int x) {
        int y = x * x;
        return y;
    }
}
```

Variable Trace

line #	main::x	main::y	square::x	square::y
11	13		13	169

```
>_
$ java Square 13
```

```java
import stdlib.StdOut;

public class Square {
    public static void main(String[] args) {
        int x = Integer.parseInt(args[0]);
        int y = square(x);
        StdOut.println(y);
    }

    private static int square(int x) {
        int y = x * x;
        return y;
    }
}
```

Variable Trace

line #	main::x	main::y	square::x	square::y
12	13			

```
>_
$ java Square 13
```

```java
import stdlib.StdOut;

public class Square {
    public static void main(String[] args) {
        int x = Integer.parseInt(args[0]);
        int y = square(x);
        StdOut.println(y);
    }

    private static int square(int x) {
        int y = x * x;
        return y;
    }
}
```

Variable Trace

line #	main::x	main::y	square::x	square::y
6	13	169		

```
>_
$ java Square 13
```

```
1  import stdlib.StdOut;
2
3  public class Square {
4      public static void main(String[] args) {
5          int x = Integer.parseInt(args[0]);
6          int y = square(x);
7          StdOut.println(y);
8      }
9
10     private static int square(int x) {
11         int y = x * x;
12         return y;
13     }
14 }
```

Variable Trace

line #	main::x	main::y	square::x	square::y
7	13	169		

```
$ java Square 13
169
```

Function Definitions · Control Flow

```
1  import stdlib.StdOut;
2
3  public class Square {
4      public static void main(String[] args) {
5          int x = Integer.parseInt(args[0]);
6          int y = square(x);
7          StdOut.println(y);
8      }
9
10     private static int square(int x) {
11         int y = x * x;
12         return y;
13     }
14 }
```

Variable Trace

line #	main::x	main::y	square::x	square::y

```
$ java Square 13
169
$ _
```

- Arguments are passed by value

- Arguments are passed by value
- Function names can be overloaded

- Arguments are passed by value
- Function names can be overloaded
- A function has a single return value but may have multiple return statements

- Arguments are passed by value
- Function names can be overloaded
- A function has a single return value but may have multiple return statements
- A function can have side effects

- Arguments are passed by value
- Function names can be overloaded
- A function has a single return value but may have multiple return statements
- A function can have side effects
- Inputs to a function are generally its parameters

- Arguments are passed by value
- Function names can be overloaded
- A function has a single return value but may have multiple return statements
- A function can have side effects
- Inputs to a function are generally its parameters
- A function generally does not write any output

`HarmonicRedux.java`

Command-line input	n (int)
Standard output	the nth harmonic number, $H_n = 1 + \frac{1}{2} + \frac{1}{3} + \cdots + \frac{1}{n} \approx \ln(n) + 0.57721$

HarmonicRedux.java

Command-line input	n (int)
Standard output	the nth harmonic number, $H_n = 1 + \frac{1}{2} + \frac{1}{3} + \cdots + \frac{1}{n} \approx \ln(n) + 0.57721$

```
>_ ~/workspace/dsaj/programs

$ _
```

HarmonicRedux.java

Command-line input	n (int)
Standard output	the nth harmonic number, $H_n = 1 + \frac{1}{2} + \frac{1}{3} + \cdots + \frac{1}{n} \approx \ln(n) + 0.57721$

```
>_ ~/workspace/dsaj/programs

$ java HarmonicRedux 10
```

HarmonicRedux.java

Command-line input	n (int)
Standard output	the nth harmonic number, $H_n = 1 + \frac{1}{2} + \frac{1}{3} + \cdots + \frac{1}{n} \approx \ln(n) + 0.57721$

~/workspace/dsaj/programs

```
$ java HarmonicRedux 10
2.9289682539682538
$ _
```

HarmonicRedux.java

Command-line input	n (int)
Standard output	the nth harmonic number, $H_n = 1 + \frac{1}{2} + \frac{1}{3} + \cdots + \frac{1}{n} \approx \ln(n) + 0.57721$

>_ ~/workspace/dsaj/programs

```
$ java HarmonicRedux 10
2.9289682539682538
$ java HarmonicRedux 1000
```

HarmonicRedux.java

Command-line input	n (int)
Standard output	the nth harmonic number, $H_n = 1 + \frac{1}{2} + \frac{1}{3} + \cdots + \frac{1}{n} \approx \ln(n) + 0.57721$

```
>_ ~/workspace/dsaj/programs

$ java HarmonicRedux 10
2.9289682539682538
$ java HarmonicRedux 1000
7.485470860550343
$ _
```

HarmonicRedux.java

Command-line input	n (int)
Standard output	the nth harmonic number, $H_n = 1 + \frac{1}{2} + \frac{1}{3} + \cdots + \frac{1}{n} \approx \ln(n) + 0.57721$

```
>_ ~/workspace/dsaj/programs

$ java HarmonicRedux 10
2.9289682539682538
$ java HarmonicRedux 1000
7.485470860550343
$ java HarmonicRedux 10000
```

HarmonicRedux.java

Command-line input	n (int)
Standard output	the nth harmonic number, $H_n = 1 + \frac{1}{2} + \frac{1}{3} + \cdots + \frac{1}{n} \approx \ln(n) + 0.57721$

```
>_ ~/workspace/dsaj/programs

$ java HarmonicRedux 10
2.9289682539682538
$ java HarmonicRedux 1000
7.485470860550343
$ java HarmonicRedux 10000
9.787606036044348
$ _
```

Examples · Harmonic Numbers

</> HarmonicRedux.java

```java
import stdlib.StdOut;

public class HarmonicRedux {
    public static void main(String[] args) {
        int n = Integer.parseInt(args[0]);
        StdOut.println(harmonic(n));
    }

    private static double harmonic(int n) {
        double total = 0.0;
        for (int i = 1; i <= n; i++) {
            total += 1.0 / i;
        }
        return total;
    }
}
```

`CouponCollectorRedux.java`

Command-line input	n (int)
Standard output	number of coupons one must collect before obtaining at least one of the n unique coupons

CouponCollectorRedux.java

Command-line input	n (int)
Standard output	number of coupons one must collect before obtaining at least one of the n unique coupons

```
>_ ~/workspace/dsaj/programs
$ _
```

CouponCollectorRedux.java

Command-line input	n (int)
Standard output	number of coupons one must collect before obtaining at least one of the n unique coupons

```
>_ ~/workspace/dsaj/programs
$ java CouponCollectorRedux 1000
```

CouponCollectorRedux.java

Command-line input	n (int)
Standard output	number of coupons one must collect before obtaining at least one of the n unique coupons

>_ ~/workspace/dsaj/programs

```
$ java CouponCollectorRedux 1000
7462
$ _
```

CouponCollectorRedux.java

Command-line input	*n* (int)
Standard output	number of coupons one must collect before obtaining at least one of the *n* unique coupons

~/workspace/dsaj/programs

```
$ java CouponCollectorRedux 1000
7462
$ java CouponCollectorRedux 1000
```

CouponCollectorRedux.java

Command-line input	n (int)
Standard output	number of coupons one must collect before obtaining at least one of the n unique coupons

>_ ~/workspace/dsaj/programs

```
$ java CouponCollectorRedux 1000
7462
$ java CouponCollectorRedux 1000
9514
$ _
```

CouponCollectorRedux.java

Command-line input	n (int)
Standard output	number of coupons one must collect before obtaining at least one of the n unique coupons

>_ ~/workspace/dsaj/programs

```
$ java CouponCollectorRedux 1000
7462
$ java CouponCollectorRedux 1000
9514
$ java CouponCollectorRedux 1000000
```

CouponCollectorRedux.java

Command-line input	n (int)
Standard output	number of coupons one must collect before obtaining at least one of the n unique coupons

```
>_ ~/workspace/dsaj/programs

$ java CouponCollectorRedux 1000
7462
$ java CouponCollectorRedux 1000
9514
$ java CouponCollectorRedux 1000000
13368303
$ _
```

Examples · Coupon Collector Problem

</> CouponCollectorRedux.java

```java
import stdlib.StdOut;
import stdlib.StdRandom;

public class CouponCollectorRedux {
    public static void main(String[] args) {
        int n = Integer.parseInt(args[0]);
        StdOut.println(collect(n));
    }

    private static int collect(int n) {
        int count = 0;
        int collectedCount = 0;
        boolean[] isCollected = new boolean[n];
        while (collectedCount < n) {
            int value = getCoupon(n);
            count++;
            if (!isCollected[value]) {
                collectedCount++;
                isCollected[value] = true;
            }
        }
        return count;
    }

    private static int getCoupon(int n) {
        return StdRandom.uniform(0, n);
    }
}
```

Recursive Functions

Recursive Functions

A recursive function is one that

- Calls itself
- Has a base case
- Addresses smaller, non overlapping subproblems in each recursive call

$$n! = \begin{cases} n(n-1)! & \text{if } n > 0, \text{ and} \\ 1 & \text{if } n = 0 \end{cases}$$

$$n! = \begin{cases} n(n-1)! & \text{if } n > 0, \text{ and} \\ 1 & \text{if } n = 0 \end{cases}$$

```
private static int factorial(int n) {
    if (n == 0) {
        return 1;
    }
    return n * factorial(n - 1);
}
```

$$n! = \begin{cases} n(n-1)! & \text{if } n > 0, \text{ and} \\ 1 & \text{if } n = 0 \end{cases}$$

```
private static int factorial(int n) {
    if (n == 0) {
        return 1;
    }
    return n * factorial(n - 1);
}
```

Call trace for `factorial(5)`

```
int x = factorial(5);
```

Recursive Functions · Example (Factorial Function)

$$n! = \begin{cases} n(n-1)! & \text{if } n > 0, \text{ and} \\ 1 & \text{if } n = 0 \end{cases}$$

```java
private static int factorial(int n) {
    if (n == 0) {
        return 1;
    }
    return n * factorial(n - 1);
}
```

Call trace for `factorial(5)`

```
int x = factorial(5);
          return 5 * factorial(4)
```

$$n! = \begin{cases} n(n-1)! & \text{if } n > 0, \text{ and} \\ 1 & \text{if } n = 0 \end{cases}$$

```
private static int factorial(int n) {
    if (n == 0) {
        return 1;
    }
    return n * factorial(n - 1);
}
```

Call trace for `factorial(5)`

```
int x = factorial(5);
          return 5 * factorial(4)
                       return 4 * factorial(3)
```

$$n! = \begin{cases} n(n-1)! & \text{if } n > 0, \text{ and} \\ 1 & \text{if } n = 0 \end{cases}$$

```
private static int factorial(int n) {
    if (n == 0) {
        return 1;
    }
    return n * factorial(n - 1);
}
```

Call trace for `factorial(5)`

```
int x = factorial(5);
          return 5 * factorial(4)
                       return 4 * factorial(3)
                                    return 3 * factorial(2)
```

Recursive Functions · Example (Factorial Function)

$$n! = \begin{cases} n(n-1)! & \text{if } n > 0, \text{ and} \\ 1 & \text{if } n = 0 \end{cases}$$

```
private static int factorial(int n) {
    if (n == 0) {
        return 1;
    }
    return n * factorial(n - 1);
}
```

Call trace for `factorial(5)`

```
int x = factorial(5);
          return 5 * factorial(4)
                       return 4 * factorial(3)
                                    return 3 * factorial(2)
                                                 return 2 * factorial(1)
```

$$n! = \begin{cases} n(n-1)! & \text{if } n > 0, \text{ and} \\ 1 & \text{if } n = 0 \end{cases}$$

```
private static int factorial(int n) {
    if (n == 0) {
        return 1;
    }
    return n * factorial(n - 1);
}
```

Call trace for `factorial(5)`

```
int x = factorial(5);
          return 5 * factorial(4)
                       return 4 * factorial(3)
                                    return 3 * factorial(2)
                                                 return 2 * factorial(1)
                                                              return 1 * factorial(0)
```

$$n! = \begin{cases} n(n-1)! & \text{if } n > 0, \text{ and} \\ 1 & \text{if } n = 0 \end{cases}$$

```
private static int factorial(int n) {
    if (n == 0) {
        return 1;
    }
    return n * factorial(n - 1);
}
```

Call trace for `factorial(5)`

```
int x = factorial(5);
          return 5 * factorial(4)
                       return 4 * factorial(3)
                                    return 3 * factorial(2)
                                                 return 2 * factorial(1)
                                                              return 1 * factorial(0)
                                                                           return 1
```

Recursive Functions · Example (Factorial Function)

$$n! = \begin{cases} n(n-1)! & \text{if } n > 0, \text{ and} \\ 1 & \text{if } n = 0 \end{cases}$$

```
private static int factorial(int n) {
    if (n == 0) {
        return 1;
    }
    return n * factorial(n - 1);
}
```

Call trace for `factorial(5)`

```
int x = factorial(5);
          return 5 * factorial(4)
                       return 4 * factorial(3)
                                    return 3 * factorial(2)
                                                 return 2 * factorial(1)
                                                              return 1 * 1
```

Recursive Functions · Example (Factorial Function)

$$n! = \begin{cases} n(n-1)! & \text{if } n > 0, \text{ and} \\ 1 & \text{if } n = 0 \end{cases}$$

```
private static int factorial(int n) {
    if (n == 0) {
        return 1;
    }
    return n * factorial(n - 1);
}
```

Call trace for `factorial(5)`

```
int x = factorial(5);
          return 5 * factorial(4)
                       return 4 * factorial(3)
                                    return 3 * factorial(2)
                                                 return 2 * 1
```

$$n! = \begin{cases} n(n-1)! & \text{if } n > 0, \text{ and} \\ 1 & \text{if } n = 0 \end{cases}$$

```
private static int factorial(int n) {
    if (n == 0) {
        return 1;
    }
    return n * factorial(n - 1);
}
```

Call trace for `factorial(5)`

```
int x = factorial(5);
          return 5 * factorial(4)
                       return 4 * factorial(3)
                                    return 3 * 2
```

$$n! = \begin{cases} n(n-1)! & \text{if } n > 0, \text{ and} \\ 1 & \text{if } n = 0 \end{cases}$$

```
private static int factorial(int n) {
    if (n == 0) {
        return 1;
    }
    return n * factorial(n - 1);
}
```

Call trace for `factorial(5)`

```
int x = factorial(5);
          return 5 * factorial(4)
                       return 4 * 6
```

$$n! = \begin{cases} n(n-1)! & \text{if } n > 0, \text{ and} \\ 1 & \text{if } n = 0 \end{cases}$$

```
private static int factorial(int n) {
    if (n == 0) {
        return 1;
    }
    return n * factorial(n - 1);
}
```

Call trace for `factorial(5)`

```
int x = factorial(5);
          return 5 * 24
```

$$n! = \begin{cases} n(n-1)! & \text{if } n > 0, \text{ and} \\ 1 & \text{if } n = 0 \end{cases}$$

```
private static int factorial(int n) {
    if (n == 0) {
        return 1;
    }
    return n * factorial(n - 1);
}
```

Call trace for `factorial(5)`

```
int x = 120;
```

Factorial.java

Command-line input	n (int)
Standard output	$n!$

Factorial.java

Command-line input	n (int)
Standard output	$n!$

~/workspace/dsaj/programs

```
$ _
```

Factorial.java

Command-line input	n (int)
Standard output	$n!$

~/workspace/dsaj/programs

```
$ java Factorial 0
```

Factorial.java

Command-line input	n (int)
Standard output	$n!$

~/workspace/dsaj/programs

```
$ java Factorial 0
1
$ _
```

Factorial.java

Command-line input	n (int)
Standard output	$n!$

~/workspace/dsaj/programs

```
$ java Factorial 0
1
$ java Factorial 5
```

Factorial.java

Command-line input	n (int)
Standard output	$n!$

~/workspace/dsaj/programs

```
$ java Factorial 0
1
$ java Factorial 5
120
$ _
```

Recursive Functions · Example (Factorial Function)

</> Factorial.java

```java
import stdlib.StdOut;

public class Factorial {
    public static void main(String[] args) {
        int n = Integer.parseInt(args[0]);
        StdOut.println(factorial(n));
    }

    private static int factorial(int n) {
        if (n == 0) {
            return 1;
        }
        return n * factorial(n - 1);
    }
}
```

$$\text{fib}(n) = \begin{cases} \text{fib}(n-1) + \text{fib}(n-2) & \text{if } n > 1, \text{ and} \\ 1 & \text{if } n = 1, \text{ and} \\ 0 & \text{if } n = 0 \end{cases}$$

$$\text{fib}(n) = \begin{cases} \text{fib}(n-1) + \text{fib}(n-2) & \text{if } n > 1, \text{ and} \\ 1 & \text{if } n = 1, \text{ and} \\ 0 & \text{if } n = 0 \end{cases}$$

```
private static int fibonacci(int n) {
    if (n < 2) {
        return n;
    }
    return fibonacci(n - 1) + fibonacci(n - 2);
}
```

Fibonacci.java

Command-line input	n (int)
Standard output	nth Fibonacci number

Fibonacci.java

Command-line input	n (int)
Standard output	nth Fibonacci number

~/workspace/dsaj/programs

```
$ _
```

Fibonacci.java

Command-line input	n (int)
Standard output	nth Fibonacci number

~/workspace/dsaj/programs

```
$ java Fibonacci 0
```

Fibonacci.java

Command-line input	n (int)
Standard output	nth Fibonacci number

~/workspace/dsaj/programs

```
$ java Fibonacci 0
0
$ _
```

Fibonacci.java

Command-line input	n (int)
Standard output	nth Fibonacci number

~/workspace/dsaj/programs

```
$ java Fibonacci 0
0
$ java Fibonacci 1
```

Fibonacci.java

Command-line input	n (int)
Standard output	nth Fibonacci number

~/workspace/dsaj/programs

```
$ java Fibonacci 0
0
$ java Fibonacci 1
1
$ _
```

Fibonacci.java

Command-line input	n (int)
Standard output	nth Fibonacci number

~/workspace/dsaj/programs

```
$ java Fibonacci 0
0
$ java Fibonacci 1
1
$ java Fibonacci 2
```

Fibonacci.java

Command-line input	n (int)
Standard output	nth Fibonacci number

~/workspace/dsaj/programs

```
$ java Fibonacci 0
0
$ java Fibonacci 1
1
$ java Fibonacci 2
1
$ _
```

Fibonacci.java

Command-line input	n (int)
Standard output	nth Fibonacci number

~/workspace/dsaj/programs

```
$ java Fibonacci 0
0
$ java Fibonacci 1
1
$ java Fibonacci 2
1
$ java Fibonacci 3
```

Fibonacci.java

Command-line input	n (int)
Standard output	nth Fibonacci number

~/workspace/dsaj/programs

```
$ java Fibonacci 0
0
$ java Fibonacci 1
1
$ java Fibonacci 2
1
$ java Fibonacci 3
2
$ _
```

Fibonacci.java

Command-line input	n (int)
Standard output	nth Fibonacci number

~/workspace/dsaj/programs

```
$ java Fibonacci 0
0
$ java Fibonacci 1
1
$ java Fibonacci 2
1
$ java Fibonacci 3
2
$ java Fibonacci 4
```

Fibonacci.java

Command-line input	n (int)
Standard output	nth Fibonacci number

~/workspace/dsaj/programs

```
$ java Fibonacci 0
0
$ java Fibonacci 1
1
$ java Fibonacci 2
1
$ java Fibonacci 3
2
$ java Fibonacci 4
3
$ _
```

Fibonacci.java

Command-line input	n (int)
Standard output	nth Fibonacci number

~/workspace/dsaj/programs

```
$ java Fibonacci 0
0
$ java Fibonacci 1
1
$ java Fibonacci 2
1
$ java Fibonacci 3
2
$ java Fibonacci 4
3
$ java Fibonacci 10
```

Fibonacci.java

Command-line input	n (int)
Standard output	nth Fibonacci number

~/workspace/dsaj/programs

```
$ java Fibonacci 0
0
$ java Fibonacci 1
1
$ java Fibonacci 2
1
$ java Fibonacci 3
2
$ java Fibonacci 4
3
$ java Fibonacci 10
55
$ _
```

Recursive Functions · Example (Fibonacci Function)

</> Fibonacci.java

```java
import stdlib.StdOut;

public class Fibonacci {
    public static void main(String[] args) {
        int n = Integer.parseInt(args[0]);
        StdOut.println(fibonacci(n));
    }

    private static int fibonacci(int n) {
        if (n < 2) {
            return n;
        }
        return fibonacci(n - 1) + fibonacci(n - 2);
    }
}
```

Recursive Functions · Example (Fibonacci Function)