562.613 APPLIED DATA STRUCTURES

Lecture 01

Dr. Abbas Malik
muhammad.malik@manukau.ac.nz

M. G. Abbas Malik

- PhD in Computer Science and Artificial Intelligence
  - Major in Machine Learning and Data Analytics
  - University of Grenoble, France
- Experienced Software and Mobile Application Designer and Developer
- Research Interests: Machine Learning, Data Analytics, Data Security, Wireless Sensor Networks, and Software Defined Networks

Class Introduction

Source

562.613 Applied Data Structures

- **Course Assessments**
  - Test 1 — 40% weight (in **Week 4**)
  - Test 2 — 40% weight (in **Week 7**)
  - Project — 20% weight (released by the end of **week 2** and due in **week 6**)
- **Slides and Lab material – MIT Canvas**

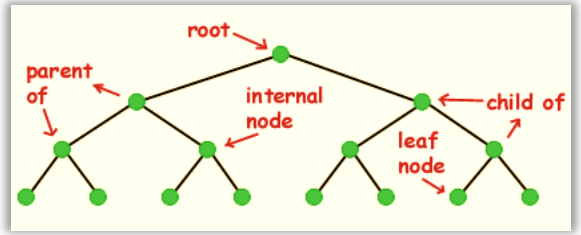
Course Outline and Class Setup

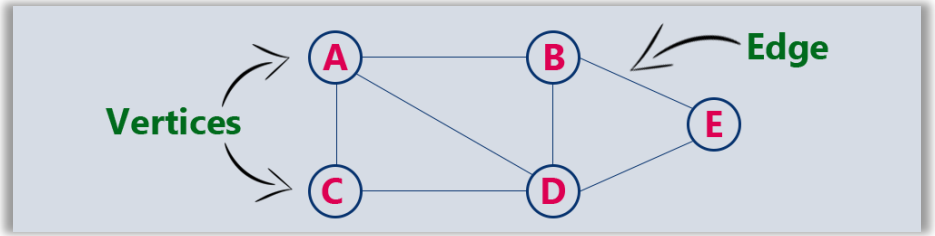
- A complete course outline - MIT Canvas
- 2 classes every week
  - Monday at 9:00 – 12:00
  - Thursday at 9:00 – 12:00
- A Class - Lecture + Programming Practice
- Language of Implementation - C#
- Microsoft Visual Studio IDE

Purpose of ADS Course

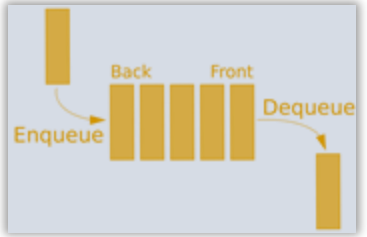
- **Data Structure**: a specific way to store and organize data in a computer
- **Efficiently using data** in our program and software
- **Data values** with **relations** among them and **functions/methods** that can be applied on data

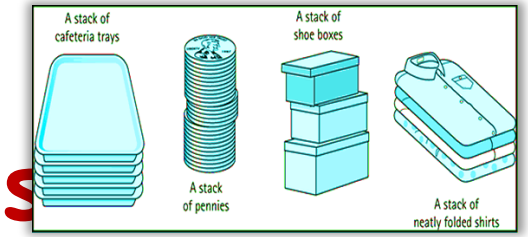
Purpose of Course

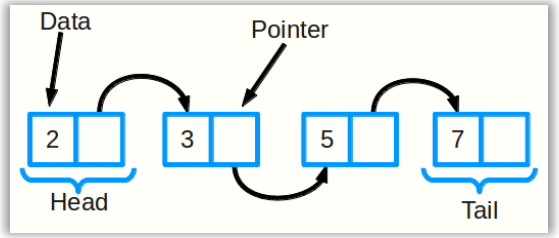




- Student's Data
  - Name, ID, address, mobile, program, gender, …
- Course Data
  - Name, Code, level, assessments, program, …
- Cannot be stored in **Primitive Data Types**
  - int, String, double, char, float, …







ADT – Abstract Data Type

Purpose of Course

- **Algorithms**: step by step instructions to solve a problem
- **Sorting Algorithms** – Bubble, Insertion, Quick, Heap, Merge, Counting
- **Searching Algorithms** – Linear, Binary
- …

Course References and Reading Material

- C# Guide
- C# Language Reference
- Programming Guide
- Visual Studio Documentation
- .Net Framework API Reference 4.7.1
- .Net Core API Reference

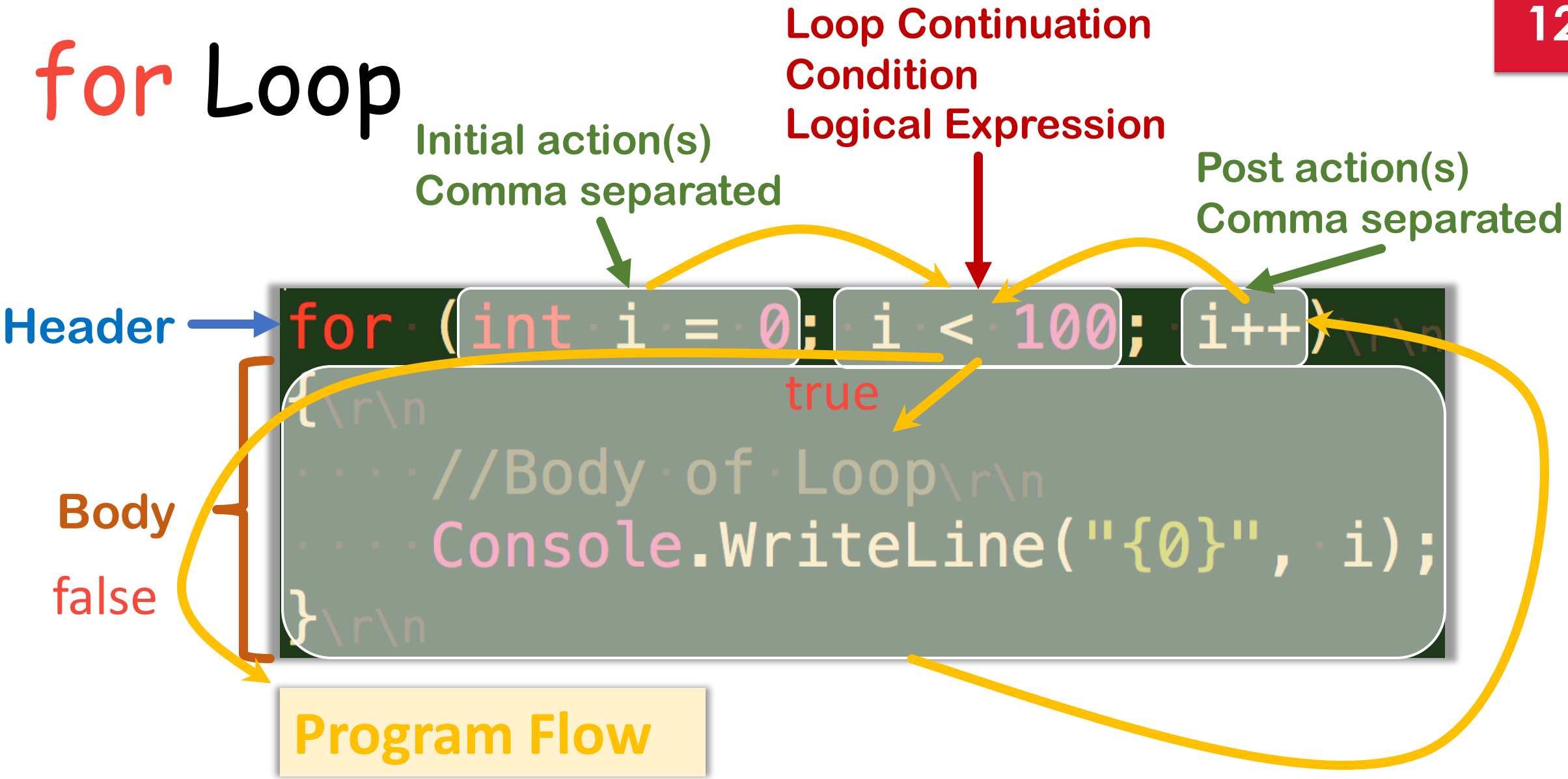
Lecture 01

- Loops – FOR, WHILE, DO … WHILE
- Algorithms
  - Factorial
  - Fibonacci Numbers
  - Compute Power
- Recursion and Recursive Algorithms

Loops in C#

- Three Types
  - for loop
  - while loop
  - do-while loop
- break
- continue

for Loop


Loop Continuation
Condition
Logical Expression
Initial action(s)
Comma separated
Post action(s)
Comma separated
Header
for (int i = 0; i < 100; i++)\r\n
{\r\n
//Body of Loop\r\n
Console.WriteLine("{0}", i);
}\r\n
true
Body
false
Program Flow

for Loop - Infinite Loop

- Loop Continuation Condition is always **true**

```
for (int i = 0; true; i++)\r\n
{\r\n
    Console.WriteLine("{0}", i);
}\r\n
```

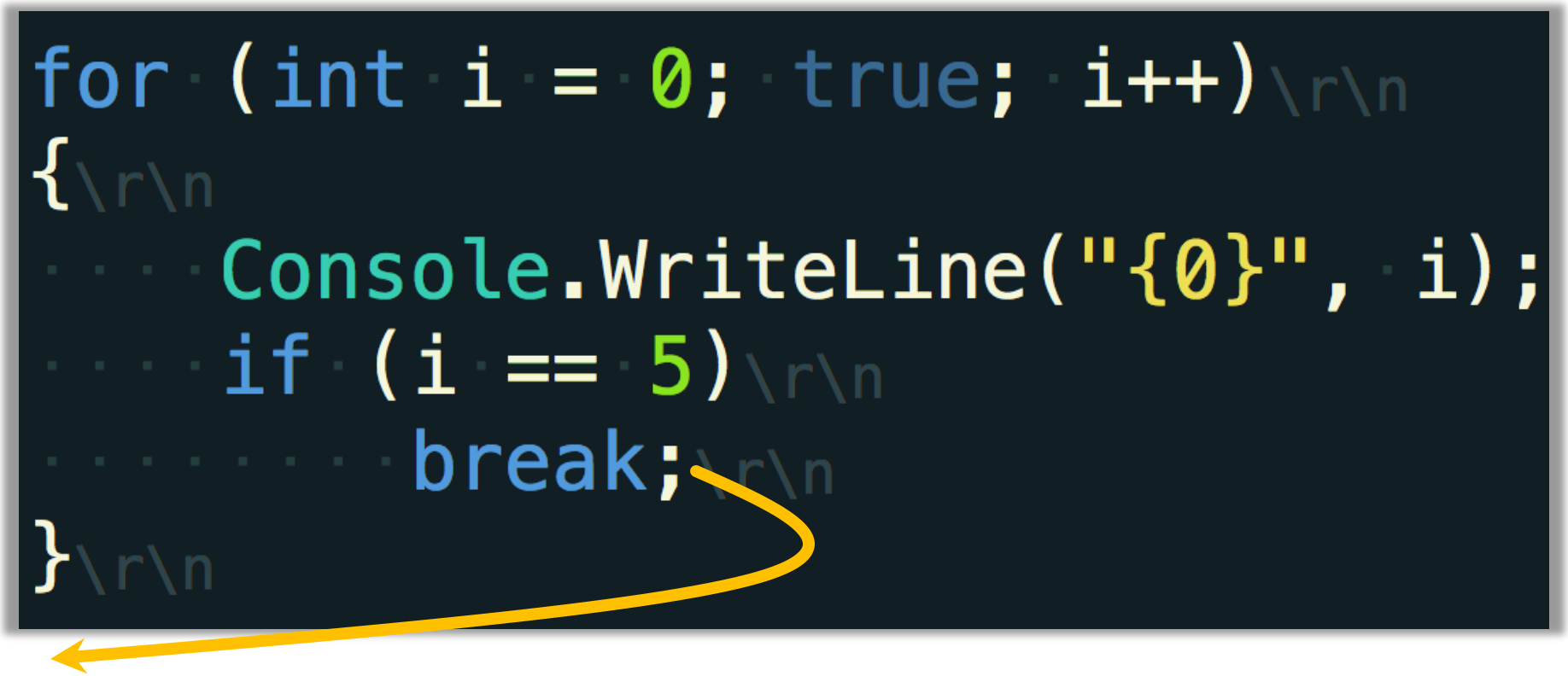
0
1
2
3
4
.
.
.

Press control c (CTRL + C) to finish program

for Loop - break

- Loop will finish once, **break** is executed

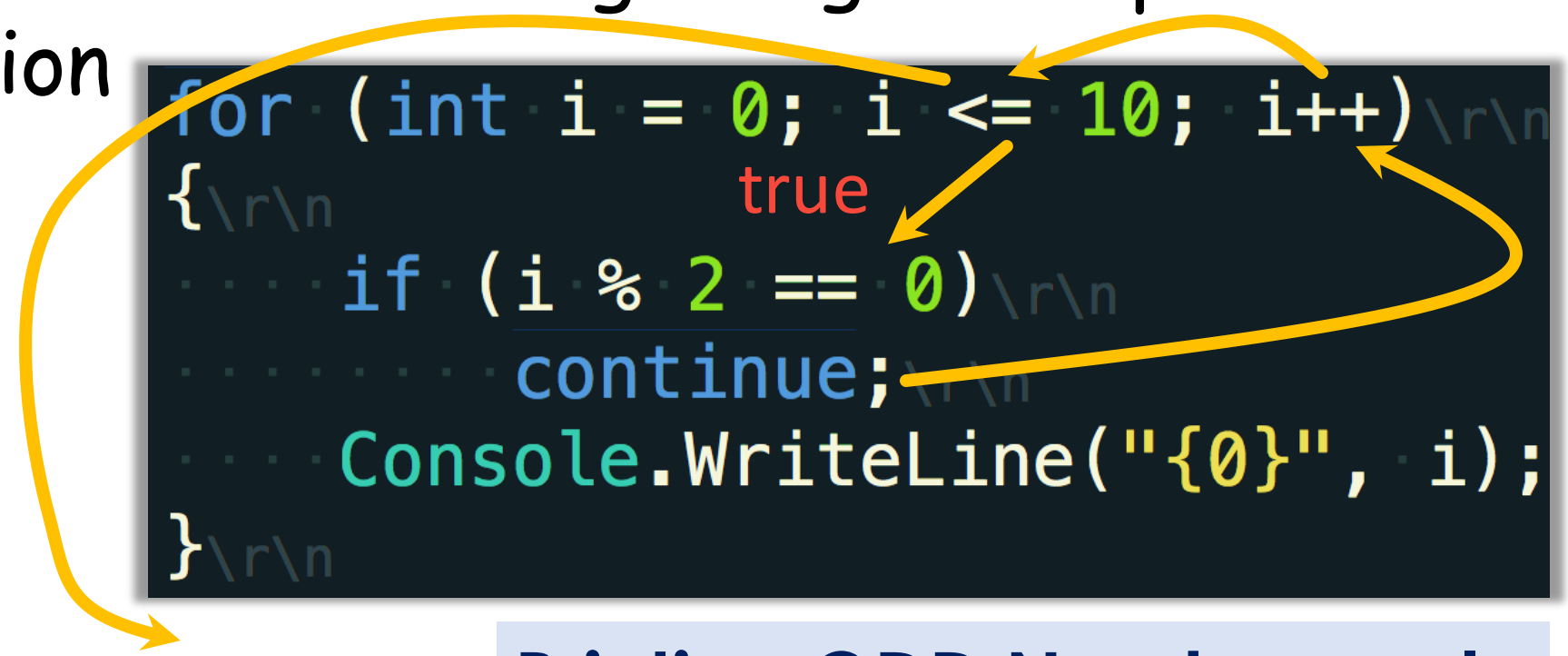
```
for (int i = 0; true; i++)\r\n
{\r\n
    Console.WriteLine("{0}", i);
    if (i == 5)\r\n
        break;\r\n
}\r\n
```



```
0
1
2
3
4
5
```

for Loop - continue

- Jump Back to the beginning of loop for next iteration

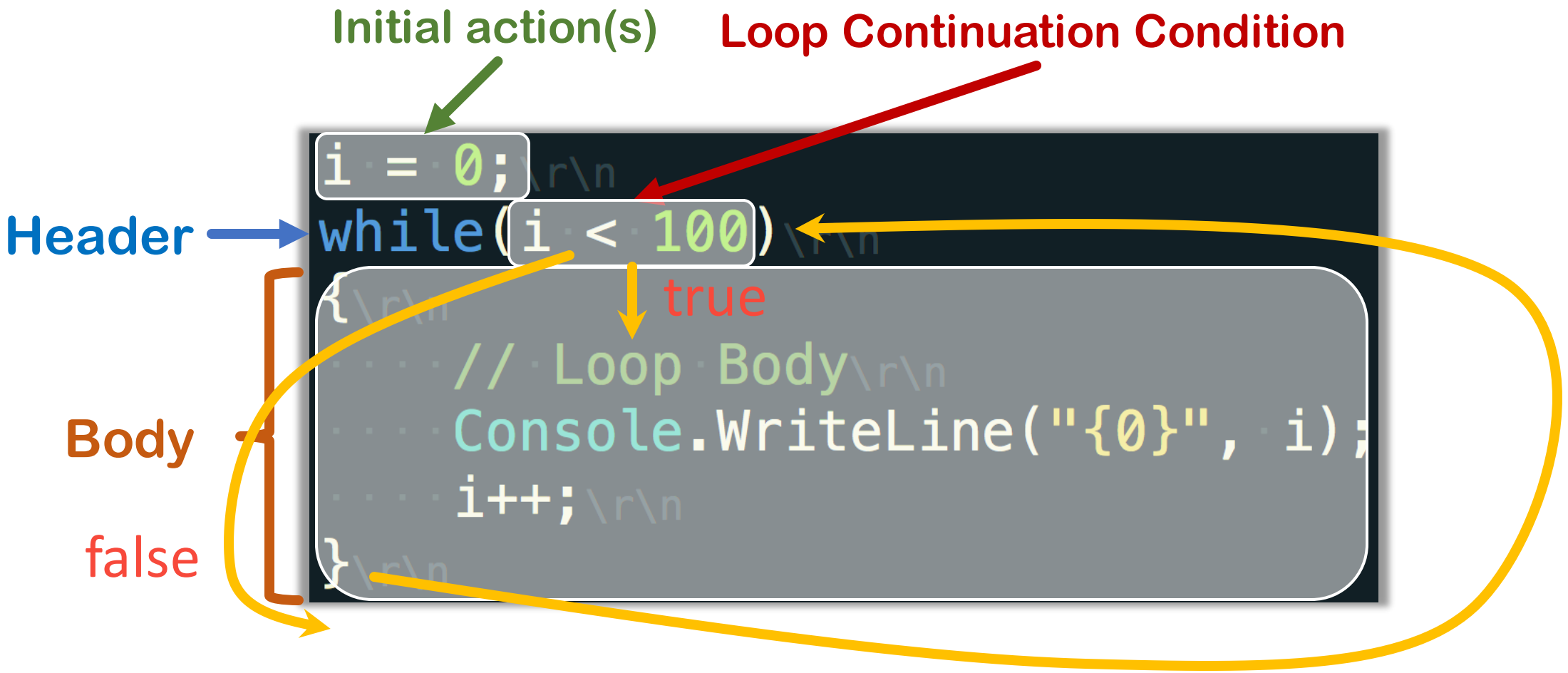


1
3
5
7
9

Printing ODD Numbers only

while Loop

Program Flow



while Loop

```
i = 0;\r\n
while(true)\r\n
{\r\n
    // Loop Body\r\n
    Console.WriteLine("{0}", i);
    i++;\r\n
    if (i == 5)\r\n
        break;\r\n
}\r\n
```

```
0
1
2
3
4
```

while Loop

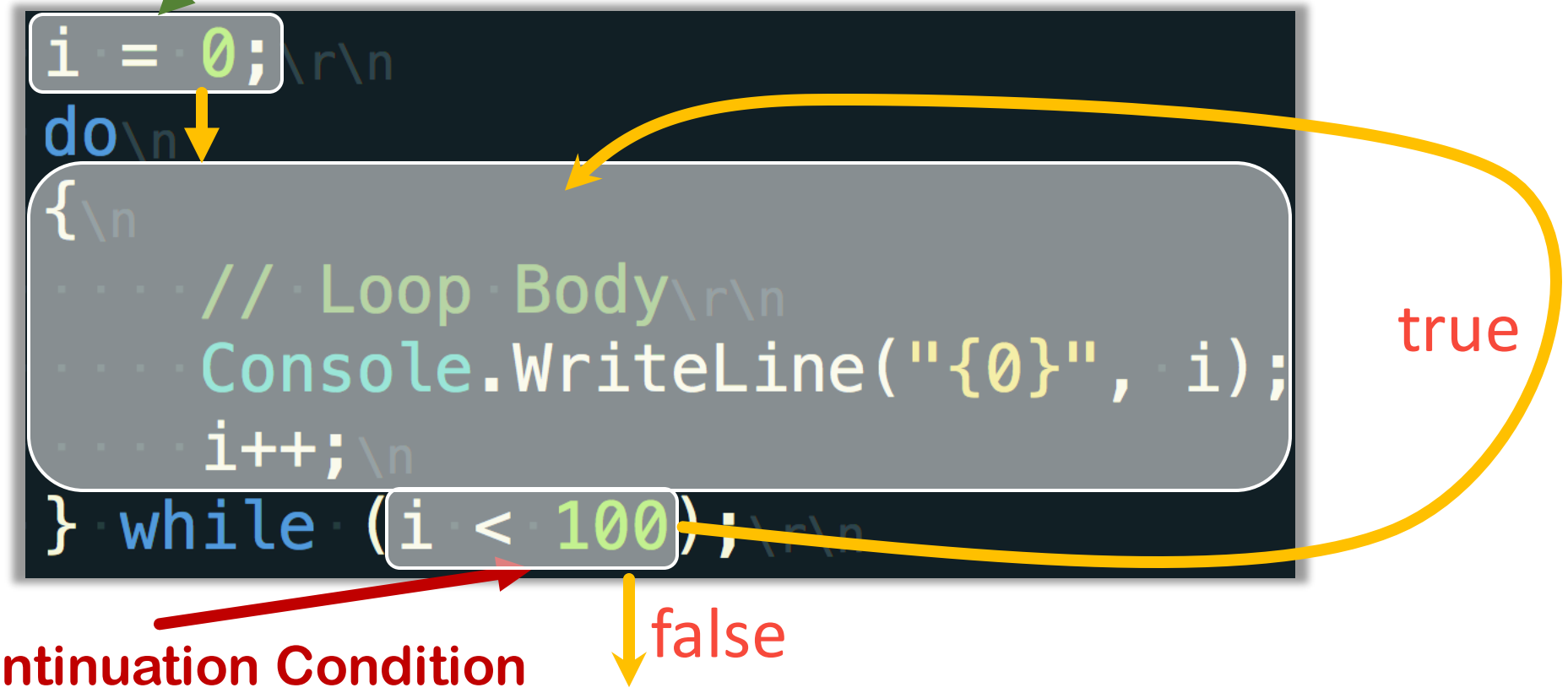
```
i = 0;\r\n
while (i<=10)\r\n
{\r\n
    // Loop Body\r\n
    i++;\r\n
    if (i % 2 == 0)\r\n
        continue;\r\n
    Console.WriteLine("{0}", i);
}\r\n
```

```
1
3
5
7
9
11
```

do...while Loop

Initial action(s)

Program Flow



do...while Loop

```
i = 0;\r\n
do\r\n
{\r\n
    // Loop Body\r\n
    Console.WriteLine("{0}", i);
    i++;\r\n
    if (i == 5)\r\n
        break;\r\n
} while (true);\r\n
```

0
1
2
3
4

do...while Loop

```
i = 0;\r\n
do\r\n
{\r\n
    // Loop Body\r\n
    i++;\r\n
    if (i%2==0)\r\n
        continue;\r\n
    Console.WriteLine("{0}", i);
} while (i<=10);\r\n
```

```
1
3
5
7
9
11
```

Algorithms

- A finite sequence of instructions to solve a problem
- Describing an Algorithm – Two ways
- Pseudocode
- Flow Chart

Algorithms

Pseudocode

Recipe
CHOCOLATE CAKE

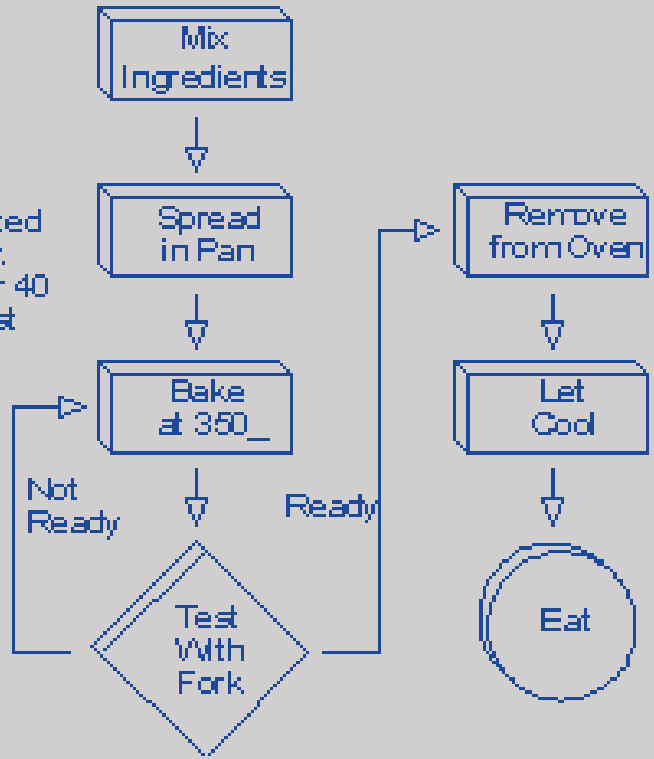
4 oz. chocolate
1 cup butter
2 cups sugar
3 eggs
1 tsp. vanilla
1 cup flour

Melt chocolate and butter. Stir sugar into melted chocolate. Stir in eggs and vanilla. Mix in flour. Spread mix in greased pan. Bake at 350_ for 40 minutes or until inserted fork comes out almost clean. Cool in pan before eating.

Program Code

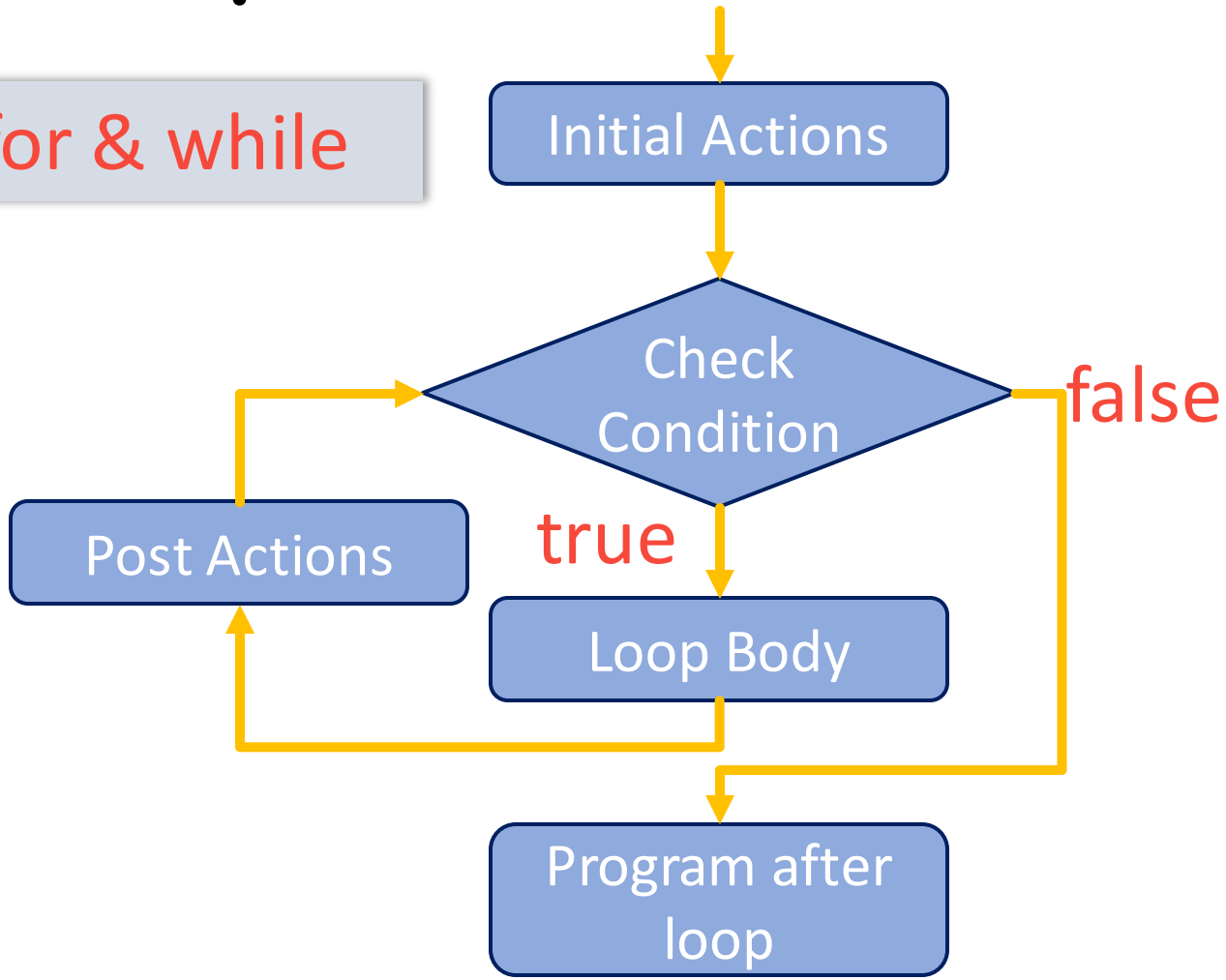
```
Declare variables:
    chocolate   eggs        mix
    butter      vanilla
    sugar       flour
mix = melted ((4*chocolate) + butter)
mix = stir (mix + (2*sugar))
mix = stir (mix + (3*eggs) + vanilla)
mix = mix + flour
spread (mix)
While not clean (fork)
bake (mix, 350)
```

Flow Chart

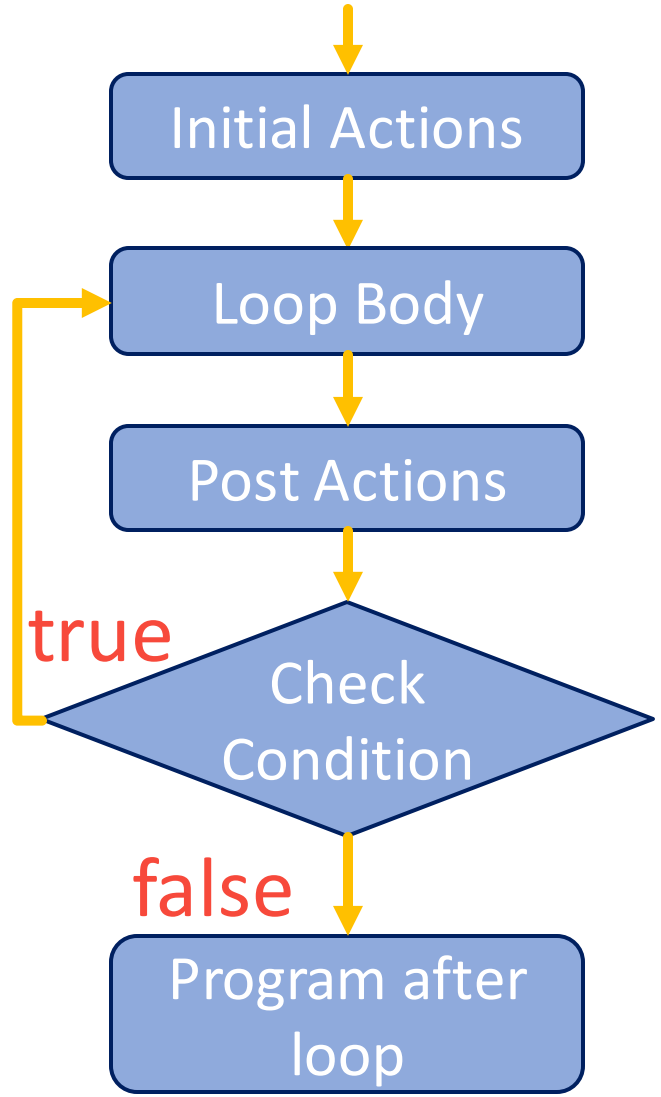


Loops - Flow Charts

for & while



do...while



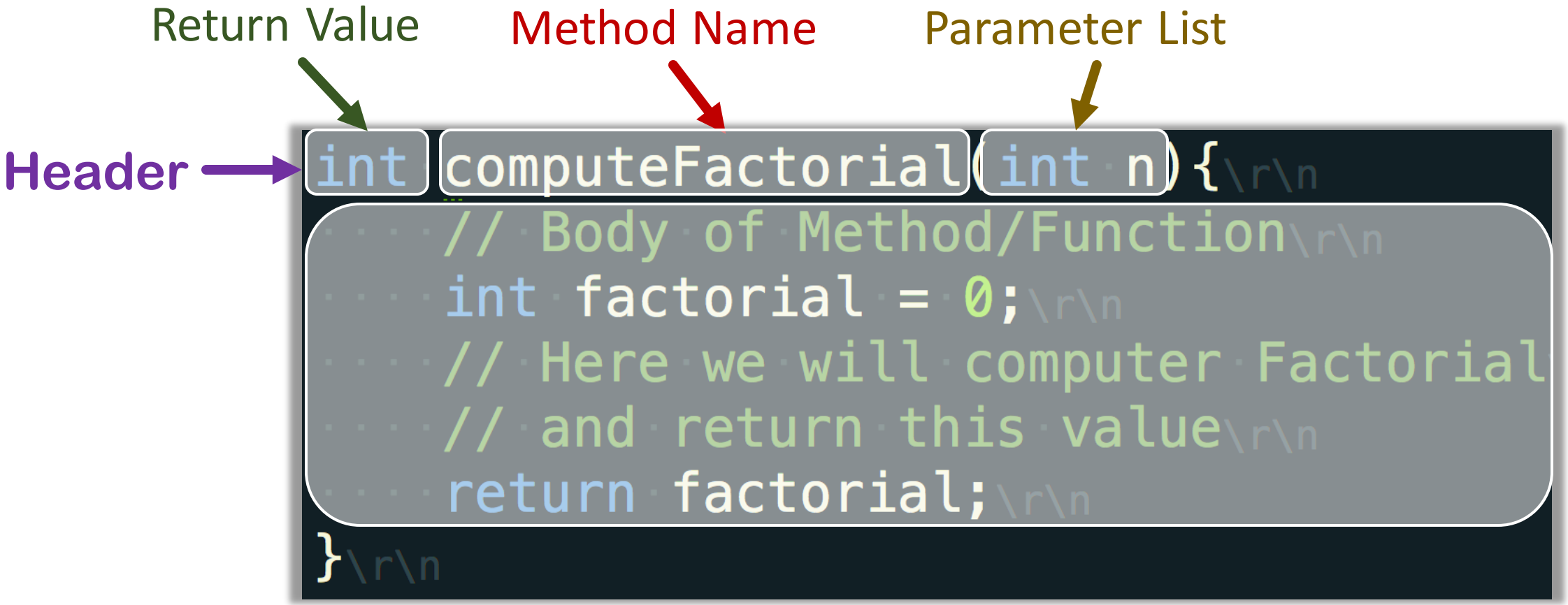
Method/Function

- A reusable code that can take multiple parameters as input and can return ZERO or ONE return value
- Two Properties
- Generalization
- Encapsulation

Method/Function

- Method Signature/Header
  - Method's Name
  - List of Parameters (0 or more)
  - Return Value (0 or 1)
- Body of Method
  - Contains steps/operations that will be performed on parameters to produce results

Method/Function



Factorial: Non-Negative Integer

- Factorial of 1, 2, 3, …
- $n! = n(n-1)(n-2)(n-3) \ldots 3.2.1$
- $5! = 5 \times 4 \times 3 \times 2 \times 1 = 120$
- $0! = 1$
- Design an algorithm to compute factorial of a non-negative number.
- Implement it in C#

Factorial: Non-Negative Integer

Algorithm 1

Input: n *is a nonnegative integer*

Output: *Factorial of* n

1. $factorial \leftarrow 1$
2. $i \leftarrow n$
3. $factorial \leftarrow factorial \times i$
4. $i \leftarrow i - 1$
5. $if(i > 0)$
6. *Go to step 2*
7. *Else*
8. *return factorial*

Algorithm 2

Input: n *is a nonnegative integer*

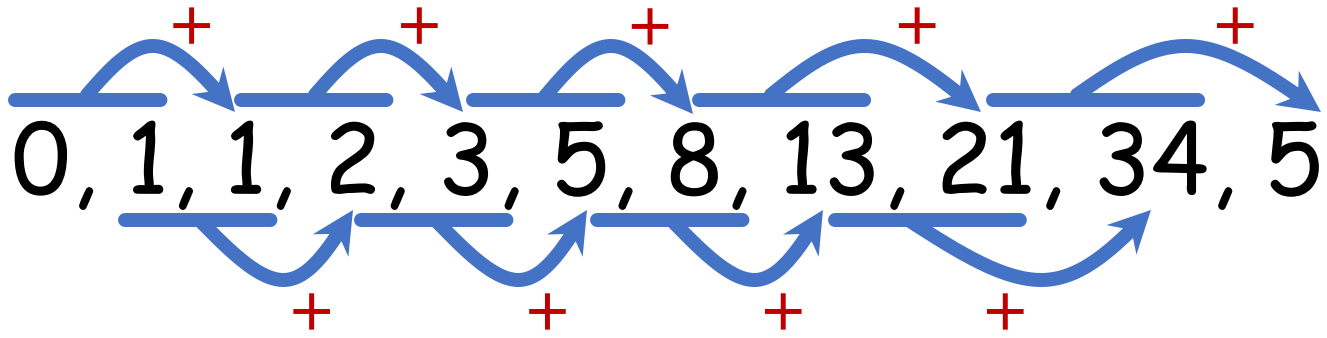
Output: *Factorial of* n

1. $factorial \leftarrow 1$
2. $i \leftarrow 1$
3. $factorial \leftarrow factorial \times i$
4. $i \leftarrow i + 1$
5. $if(i \leq n)$
6. *Go to step 3*
7. *Else*
8. *return factorial*

Factorial: Non-Negative Integer

```
static long computeFactorial(int n)
{\r\n
    long factorial = 1;\r\n
    for (int i = n; i > 0; i--)\r\n
    {\r\n
        factorial = factorial * i;\r
    }\r\n
    return factorial;\r\n
}\r\n
```

Fibonacci Numbers



- How to compute n^{th} Fibonacci Number?
- Design an algorithm to compute n^{th} Fibonacci Number?

Fibonacci Numbers

Input: n is a nonnegative integer
Output: n^{th} Fibonacci Number

1. $prev\_fibo\_num \leftarrow 0$
2. $fibo\_num \leftarrow 1$
3. $i \leftarrow 2$
4. $temp \leftarrow fibo\_num$
5. $fibo\_num \leftarrow prev\_fibo\_num + fibo\_num$
6. $prev\_fibo\_num \leftarrow temp$
7. $i \leftarrow i + 1$
8. $if(i < n)$ *Go to step 4*
9. $if(n = 1)$: **return 0**
10. *Else* $if(n = 2)$: **return 1**
11. *Else*: **return fibo_num**

Fibonacci Numbers

```
static long computeFibonacciNumber(int n){\r\n
    long fibo_num = 1, prev_fibo_num = 0, temp = -1;
    int i = 2;\r\n
    while(i < n){\r\n
        temp = fibo_num;\r\n
        fibo_num = fibo_num + prev_fibo_num;\r\n
        prev_fibo_num = temp;\r\n
        i++;\r\n
    }\r\n
    if (n == 1)\r\n
        return 0;\r\n
    else if (n == 2)\r\n
        return 1;\r\n
    else\r\n
        return fibo_num;\r\n
}\r\n
```

Computer Power x^n

- n is a non-negative integer
- x is an integer
- How to computer x^n?
- $5^4 = 5 \times 5 \times 5 \times 5 = 625$
- $(-3)^5 = (-3) \times (-3) \times (-3) \times (-3) \times (-3) = -243$

Computer Power x^n

Input: n *is a nonnegative integer*
x *is an integer*
Output: n^{th} *power of* x

1. $result \leftarrow 1$
2. $i \leftarrow n$
3. $result \leftarrow result \times x$
4. $i \leftarrow i - 1$
5. $if(i > 1)$: *go to step* 3
6. $if(n = 0)$: *return* 1
7. *else*: *return result*

Computer Power x^n

```
static long computeNPowerOfNumber(int n, int x)
{\r\n
    long result = x;\r\n
    for (int i = n; i > 1; i--){\r\n
        result = result * x;\r\n
    }\r\n
    if(n==0)\r\n
        return 1;\r\n
    else\r\n
        return result;\r\n
}\r\n
```

Recursive Functions

- **Factorial**

$$f(n) = \begin{cases} nf(n-1) \\ 1 \quad for\ n = 0 \end{cases}$$

- **Fibonacci Numbers**

$$f(n) = \begin{cases} f(n-1) + f(n-2) \\ 1 \quad for\ n = 2 \\ 0 \quad for\ n = 1 \end{cases}$$

- **Power**

$$f(x,n) = \begin{cases} xf(x,n-1) \\ 1 \quad for\ n = 0 \end{cases}$$

Recursive Algorithms

$$f(n) = \begin{cases} nf(n-1) \\ 1 \quad for\ n = 0 \end{cases}$$

- Algorithm is calling itself to solve the problem
- Must have two parts

1. **Base Case**: Algorithm's **termination condition**
2. **Recursive call**: if base case is not reached, a recursive call is generated such that algorithm is approaching the Base Case

Recursive Factorial

Base Condition

```
static long recursiveFactorial(int n)\n
{\r\n
    if (n == 0)\r\n
        return 1;\r\n
    return n * recursiveFactorial(n - 1);
}\r\n
```

Recursive Call

Recursive Fibonacci Numbers

Base Condition

```
static long recursiveFibonacci(int n)\r\n
{\r\n
    if (n == 1)\r\n
        return 0;\r\n
    else if (n == 2)\r\n
        return 1;\r\n
    return recursiveFibonacci(n-1) + recursiveFibonacci(n-2);
}\r\n
```

Recursive Call

Recursive Power x^n

Base Condition

```
static long recursiveComputeNPowerOfNumber(int n, int x)
{\r\n
    if (n == 0)\r\n
        return 1;\r\n
    return x * recursiveComputeNPowerOfNumber(n - 1, x);
}\r\n
```

Recursive Call

Recursion

- Straightforward implementation
- Recursive calls are managed by OS
- For small problem, recursion is fast

Loops (iteration) Vs Recursion

- Recursive solutions are slower than iterative solutions
- Recursive solutions are easier to understand and to implement than iterative solutions

Recursive Call Tree

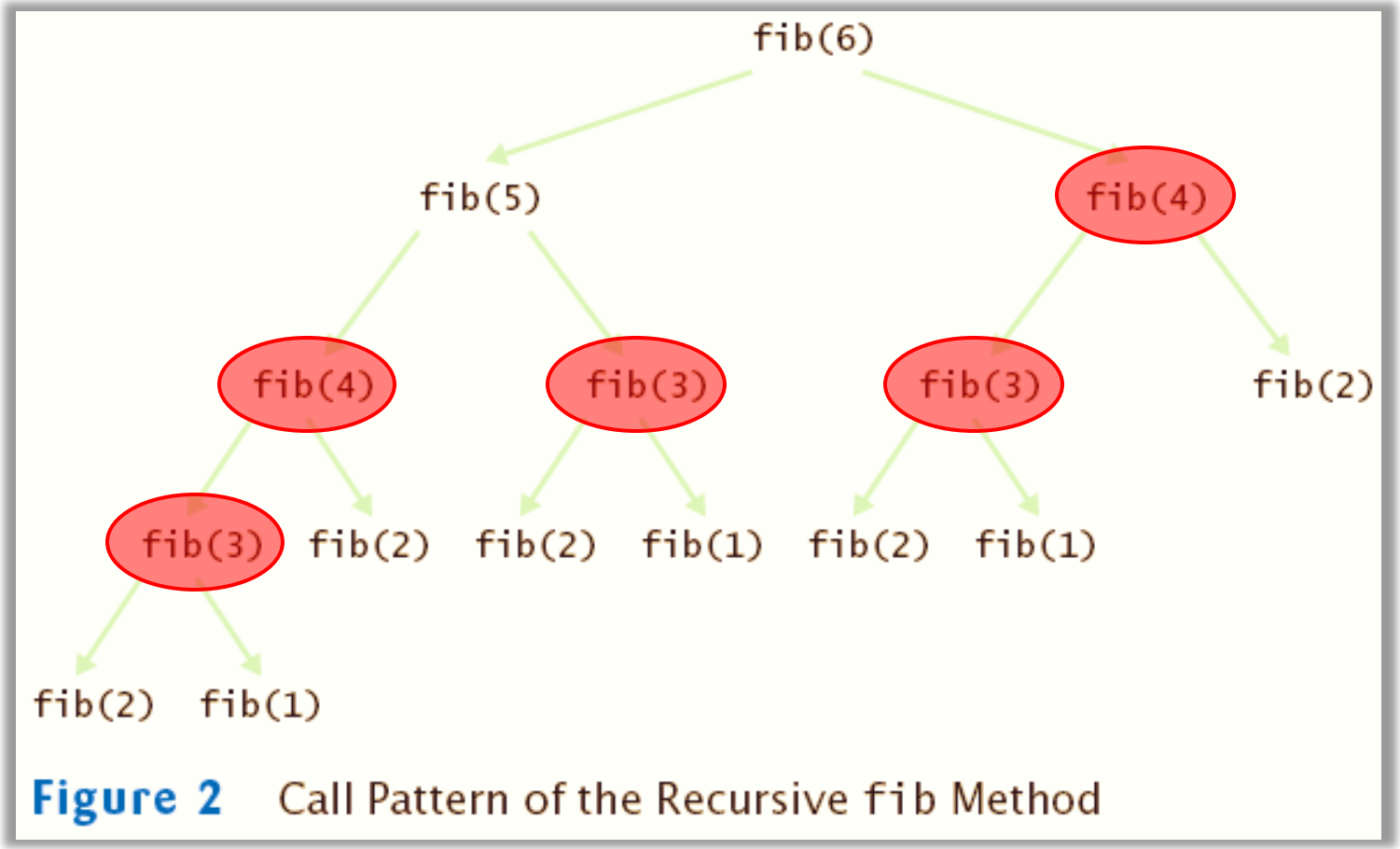


Figure 2 Call Pattern of the Recursive `fib` Method

Reference and Reading Material

- C# Tutorial: http://www.tutorialsteacher.com/csharp/csharp-tutorials
- Methods: https://docs.microsoft.com/en-us/dotnet/csharp/programming-guide/classes-and-structs/methods
- Recursion: https://www.iro.umontreal.ca/~pift1025/bigjava/Ch18/ch18.html