Week 2

Basic Process

This week's topics

- Procedural programming
- Key/framework approach
- Incremental goals
- Testing
- Debugging

Procedural programming

A stepping stone to OO programming…

Break down a program into procedures

- A small program has one goal = one method.
- A large program has many sub-goals = many methods.

Specification: Read in a size. Show a diamond of that size.

```
Size: 4
   *
  * *
 * * *
* * * *
 * * *
  * *
   *
```

Show a diamond: main procedure

```
public static void main(String[] args) {
    System.out.print(“Size: ”);
    int size = In.nextInt();
    showDiamond(size);
}
```

```
Size: 4
   *
  * *
 * * *
* * * *
 * * *
  * *
   *
```

Show a diamond: showDiamond procedure

```
public static void showDiamond(int size) {
    showTop(size);
    showMiddle(size);
    showBottom(size);
}
```

```
Size: 4
   *
  * *
 * * *
* * * *
 * * *
  * *
   *
```

Show a diamond: showTop procedure

```
public static void showTop(int size) {
    for (int length = 1; length < size; length++)
        showLine(length, size);
}
```

```
e.g. size = 4

    *       length = 1
   * *      length = 2
  * * *     length = 3
```

Show a diamond: showLine procedure

```
public static void showLine(int howManyStars, int size) {
    int howManySpaces = size - howManyStars;
    repeat(howManySpaces, “ ”);
    repeat(howManyStars, “* ”);
    System.out.println();
}
e.g. size = 4

    *      howManyStars = 1   howManySpaces = 3
   * *     howManyStars = 2   howManySpaces = 2
  * * *    howManyStars = 3   howManySpaces = 1
```

Show a diamond: repeat procedure

```
public static void repeat(int howMany, String what) {
    for (int i = 0; i < howMany; i++)
        System.out.print(what);
}
```

That’s the end of the chain...

Show a diamond: showMiddle and showBottom

```
public static void showMiddle(int size) {
    showLine(size, size);
}

public static void showBottom(int size) {
    for (int length = size - 1; length >= 1; length--)
        showLine(length, size);
}
```

Easy! Just **reuse** the showLine procedure.

Never repeat code. Always reuse code.

- Code reuse is the main benefit of splitting code into small methods.
- Put each goal in a separate method so that it can be reused.

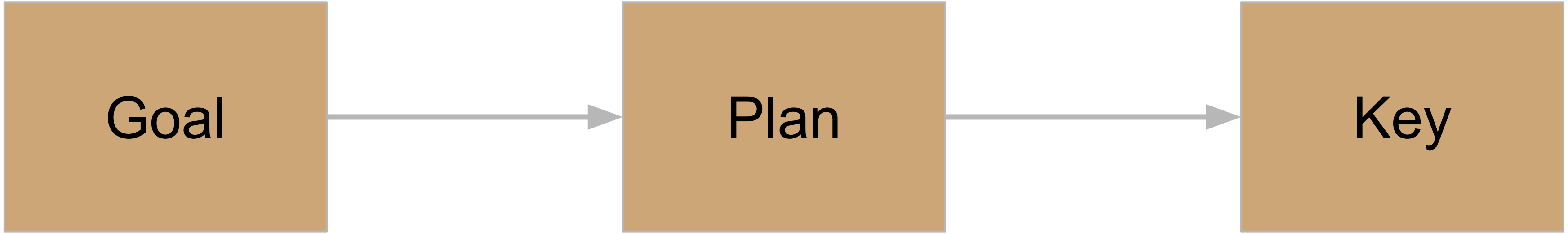
```
public static void showTop(int size) {
    for (int length = 1; length < size; length++)
        showLine(length, size);
}
public static void showMiddle(int size) {
    showLine(size, size);
}
public static void showBottom(int size) {
    for (int length = size - 1; length >= 1; length--)
        showLine(length, size);
}
```

Process #1
Key/framework approach

How to find a solution when you don’t have a pattern.

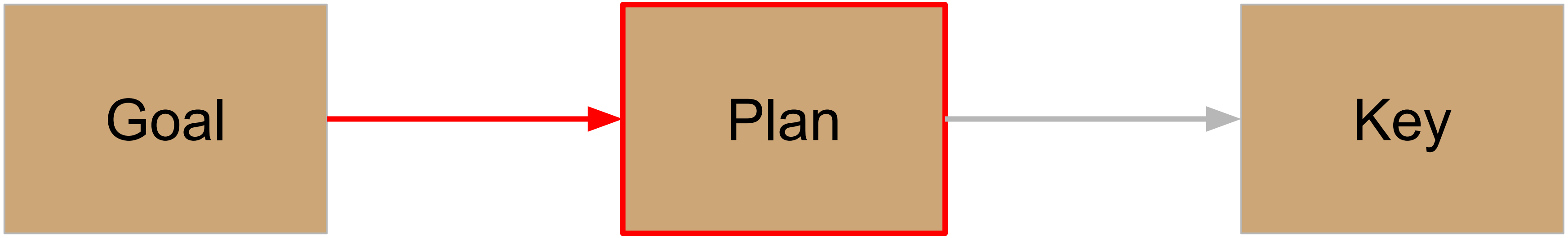
Goal, Plan and Key

- **Goal**: what your program should achieve
- **Plan**: a series of steps to achieve the goal
- **Key**: the key line of code that achieves the goal



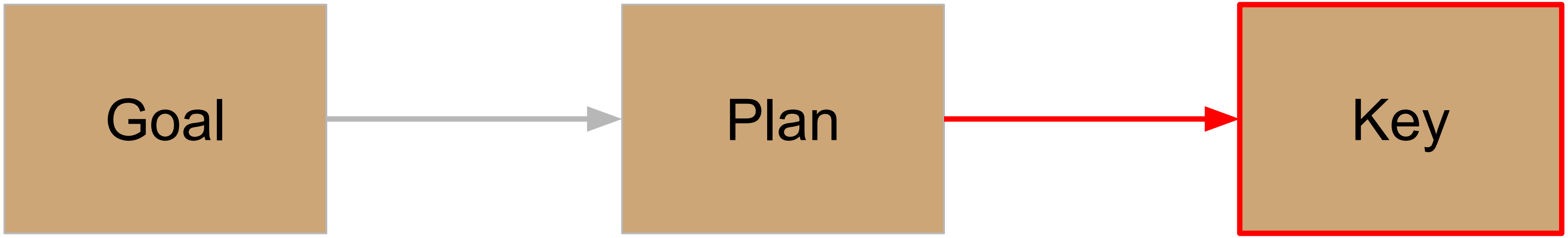
Goal, Plan and Key

Every goal needs a plan



Goal, Plan and Key

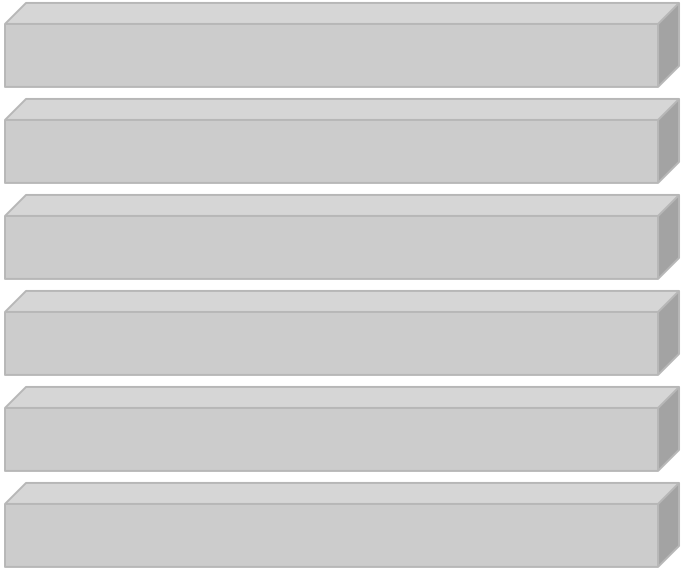
And every plan has a “key”



Where does a programmer start?

Goal: Show the total rainfall for a period.

{

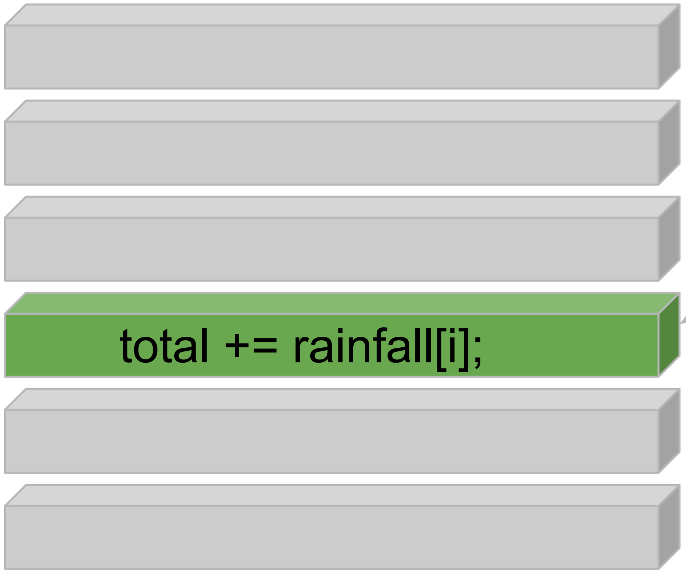


}

Where does a programmer start?

Goal: Show the total rainfall for a period.

{



```
total += rainfall[i];
```

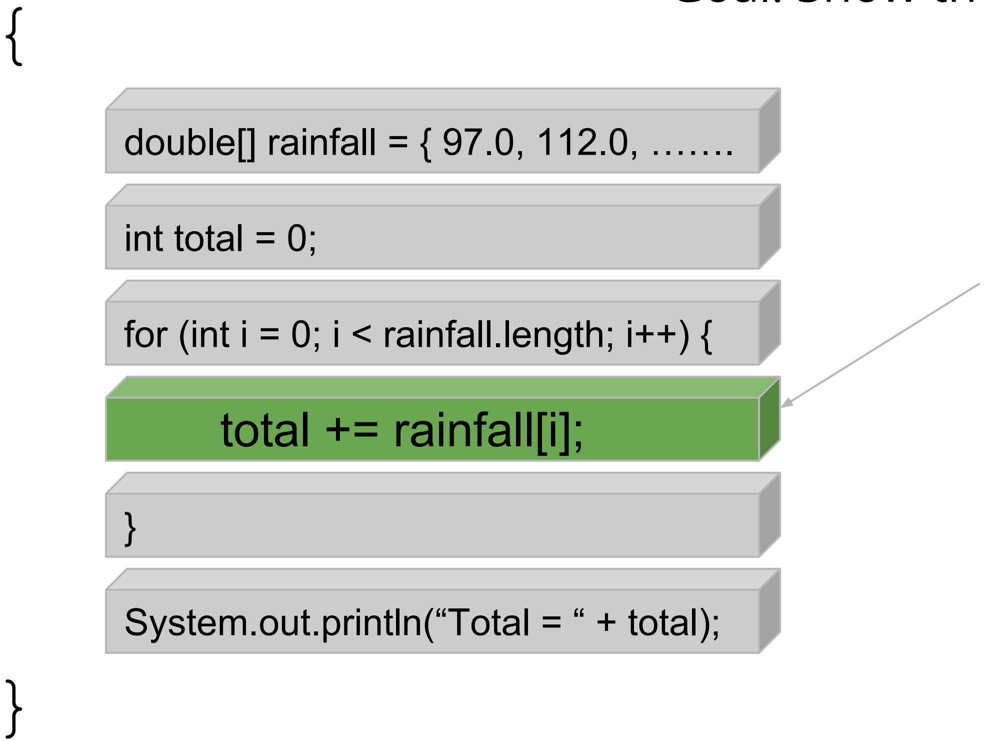
}

The “key” line of code.

Start with the **key**.

Where does a programmer start?

Goal: Show the total rainfall for a period.



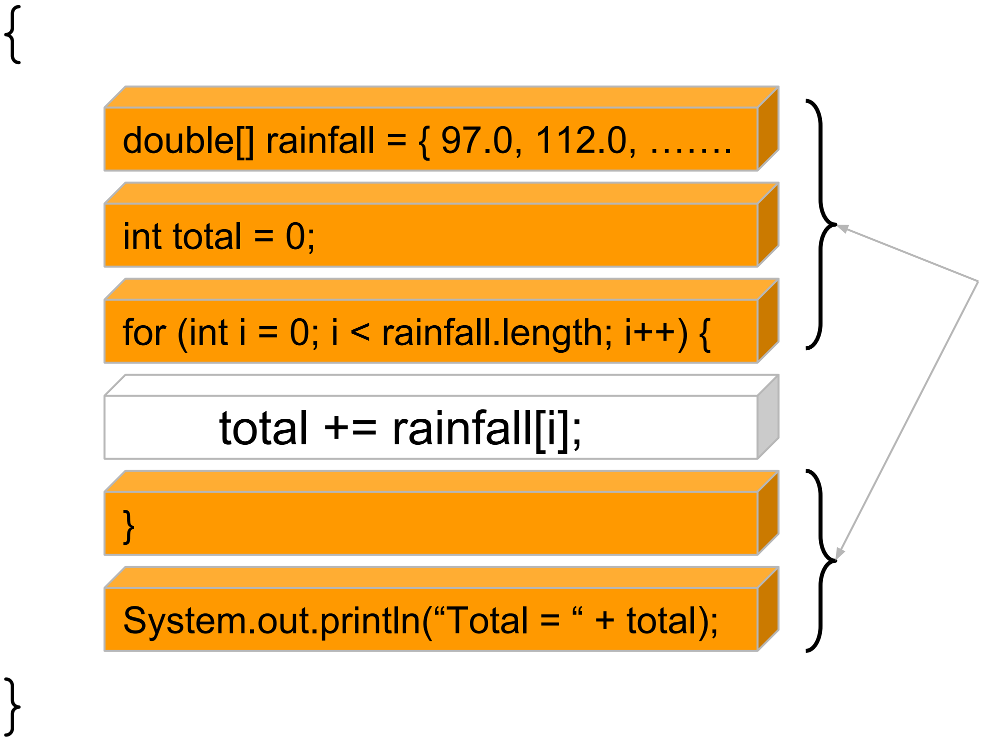
```
{
    double[] rainfall = { 97.0, 112.0, …….
    int total = 0;
    for (int i = 0; i < rainfall.length; i++) {
        total += rainfall[i];
    }
    System.out.println("Total = " + total);
}
```

The "key" line of code.

Start with the **key**.

Build the solution around the key.

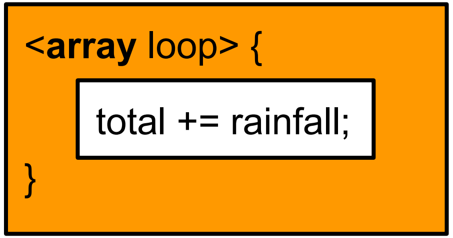
The framework



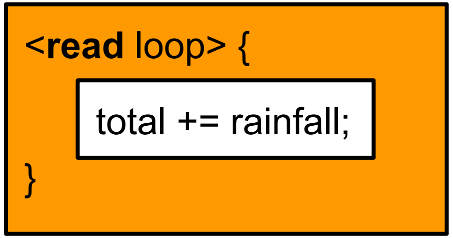
```
{
    double[] rainfall = { 97.0, 112.0, …….
    int total = 0;
    for (int i = 0; i < rainfall.length; i++) {
            total += rainfall[i];
    }
    System.out.println(“Total = “ + total);
}
```

The “framework” is whatever code supports the key.

Same key, different frameworks



```
<array loop> {
    total += rainfall;
}
```



```
<read loop> {
    total += rainfall;
}
```

The framework may vary according to the data source.

- From an array
- From user input
- From a file...

Key/framework step-by-step

Start with the key code

```
total += rainfall;
```

Add the start value

```
double total = 0.0;



    total += rainfall;
```

Add the loop

```
double total = 0.0;


<loop> {
    total += rainfall;


}
```

Decision point.

What kind of loop?

Read loop

```
double total = 0.0;


while (rainfall != -1.0) {
    total += rainfall;



}
```

Add the first read

```
double total = 0.0;
System.out.print("Rainfall: ");
double rainfall = In.nextDouble();
while (rainfall != -1.0) {
    total += rainfall;



}
```

Add the second read

```
double total = 0.0;
System.out.print(“Rainfall: “);
double rainfall = In.nextDouble();
while (rainfall != -1.0) {
    total += rainfall;
    System.out.print(“Rainfall: “);
    rainfall = In.nextDouble();
}
```

Add the output

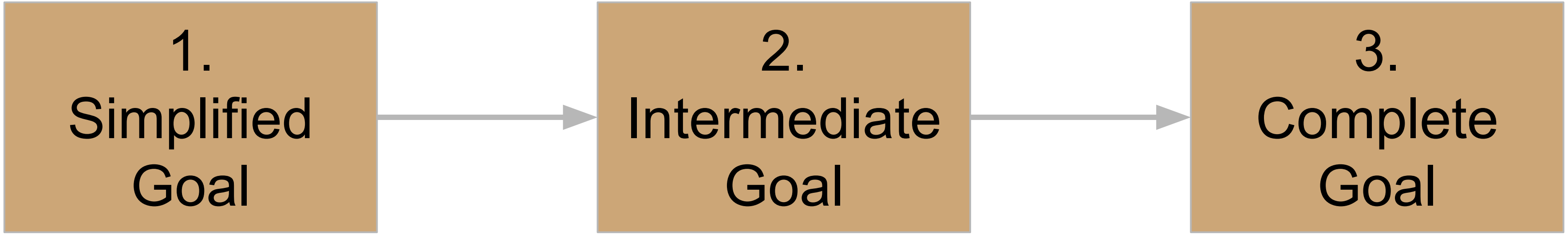
```
double total = 0.0;
System.out.print(“Rainfall: “);
double rainfall = In.nextDouble();
while (rainfall != -1.0) {
    total += rainfall;
    System.out.print(“Rainfall: “);
    rainfall = In.nextDouble();
}
System.out.println(“Total rainfall = “ + total);
```

Process #2
Incremental goals

How to tackle a large problem!

Incremental Goals

- Sometimes the goal is too difficult
- Start with a simplified goal and devise a plan for that
- Gradually add more, until the complete goal is achieved



1. Simplified Goal

2. Intermediate Goal

3. Complete Goal

Specification

Read in integers less than 100 until the user enters -1. Show the frequency of integers in each group: 0-9, 10-19, 20-29, 30-39, ..., 90-99.

```
Number: 17
Number: 19
Number: 45
Number: 87
Number: 49
Number: -1
```

```
00’s: 0
10’s: 2
20’s: 0
30’s: 0
40’s: 2
50’s: 0
60’s: 0
70’s: 0
80’s: 1
90’s: 0
```

Incremental goal #1

Goal: Show the frequency of integers in **one group**: 0-9

Patterns / key code:

- read loop
- output
- count
- if (value < 10)

Solution:

```
int count = 0;
System.out.print(“Integer: “);
int value = In.nextInt();
while (value != -1) {
     if (value < 10)
          count++;
     System.out.print(“Integer: “);
     value = In.nextInt();
}
System.out.println(“00’s: “ + count);
```

Incremental goal #2

Goal: Show the frequency of integers in **two groups**: 0-9 and 10-19.

Patterns / key code:

- read loop
- output
- count0
- count1
- if (value < 10)
- else if (value < 20)

Solution:

```
int count0 = 0, count1 = 0;
System.out.print(“Integer: “);
int value = In.nextInt();
while (value != -1) {
    if (value < 10)
        count0++;
    else if (value < 20)
        count1++;
    System.out.print(“Integer: “);
    value = In.nextInt();
}
System.out.println(“00’s: “ + count0);
System.out.println(“10’s: “ + count1);
```

Incremental goal #3: Complete

Goal: show the frequency of integers in **many groups**: 0-9, 10-19, ... 90-99.

- Design question: how can we store the count for many groups?
  Solution an array.
- How big is the array?
  10 groups. Positions start from zero: [0] [1] [2] [3] [4] [5] [6] [7] [8] [9]
- Read integer N. How do we decide which group to count?
  e.g. 95 goes into [9]. 47 goes into [4]. 31 goes into [3]. So, divide by 10:
  95 / 10 is 9. 47 / 10 is 4. 31 / 10 is 3.
- **What is the “key” code?**
  `count[value/10]++;`

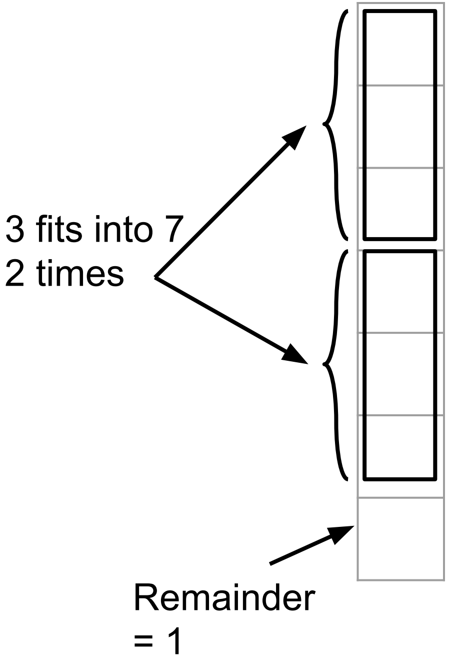
Complete Solution

```
int[] count = new int[10];
System.out.print(“Integer: “);
int value = In.nextInt();
while (value != -1) {
    count[value / 10]++;
    System.out.print(“Integer: “);
    value = In.nextInt();
}
for (int i = 0; i < count.length; i++) {
    System.out.println(i + “0’s: “ + count[i]);
}
```

Integer division and remainder

- `7 / 3 = 2`

- `7 % 3 = 1`



The modulo operator: %

- Numbers wrap around a “modulus”. e.g. 12 hour time is modulo 12.
- (11 o’clock + 2 hours) modulo 12 = 1 o’clock

  (11 + 2) % 12 = 1

n	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20
n % 3	0	1	2	0	1	2	0	1	2	0	1	2	0	1	2	0	1	2	0	1	2
n % 7	0	1	2	3	4	5	6	0	1	2	3	4	5	6	0	1	2	3	4	5	6
n % 10	0	1	2	3	4	5	6	7	8	9	0	1	2	3	4	5	6	7	8	9	0
n % 12	0	1	2	3	4	5	6	7	8	9	10	11	0	1	2	3	4	5	6	7	8

Extracting digits from a number

- The last digit of a number is: number % 10
    - e.g. What is the last digit of 92873?
    - 92873 % 10 -> 3
- number / 10 will remove a digit from the right
    - 92873 / 10 -> 9287
- To obtain the first digit of an N-digit number, divide by 10 N-1 times.
    - e.g. What is the first digit of 92873?
    - 92873 / 10 / 10 / 10 / 10 = 9 (in other words, 92873 / 10000 = 9)
- To obtain a digit in the middle, combine /10 and %10:
    - e.g. What is the middle digit of 92873?
    - Divide by 10 two times: 92873 -> 9287 -> 928.
    - Now the digit we want is the last digit.
    - 928 % 10 = 8.

3/5. Testing

How to test

- Don’t test every possible input
- Devise a finite set of test cases that are representative of all scenarios

e.g. Your program stores values into this array:

[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]

1. Middle case: Test storing into position [4]
   No need to also test [5] and [6]... etc. One middle value is representative.
2. Edge case: Test storing into position [0]
3. Edge case: Test storing into position [9]

Off-by-one error

A common programming mistake is to be “off by one”.

Loops from [0] to [9]

```
for (int i = 0; i < 10; i++)
```

Loops from [1] to [10]

```
for (int i = 1; i <= 10; i++)
```

- The left loop works
- The right loop gives ArrayIndexOutOfBoundsException: 10
- This error is picked up by testing edge cases.

Testing example

Goal: Read in integers less than 100 until the user enters -1. Show the frequency of integers in each group: 0-9, 10-19, 20-29, 30-39, ..., 90-99.

What are the edge and middle cases?

Test the beginning/middle/end of the array

Goal: Read in integers less than 100 until the user enters -1. Show the frequency of integers in each group: 0-9, 10-19, 20-29, 30-39, ..., 90-99.

- edge case: input value 4. Expected array position: [0]
- middle case: input value 42. Expected array position: [4]
- edge case: input value 97. Expected array position: [9]

Test the beginning/middle/end of a range

Goal: Read in integers less than 100 until the user enters -1. Show the frequency of integers in each group: 0-9, 10-19, 20-29, 30-39, ..., 90-99.

- edge case: input the value 30.
- middle case: input the value 35.
- edge case: input the value 39.

Expected array position: [3]

Testing Invalid Inputs

Goal: Read in integers less than 100 until the user enters -1. Show the frequency of integers in each group: 0-9, 10-19, 20-29, 30-39, ..., 90-99.

Test just above and below the highest and lowest valid input:

- bottom invalid case: input -2
- top invalid case: input 100

NOTE: In this subject, we generally do not require testing for invalid inputs, unless explicitly stated.

This means your program usually will not need to test for invalid inputs.

4/5. Debugging

The “poor person’s” debugger

- If you lack debugging tools, insert temporary `println` statements into your code.
- e.g. Does the key code `count[value/10]++` actually work?
- Print value/10 to see if it points to the correct array position:

Change this…

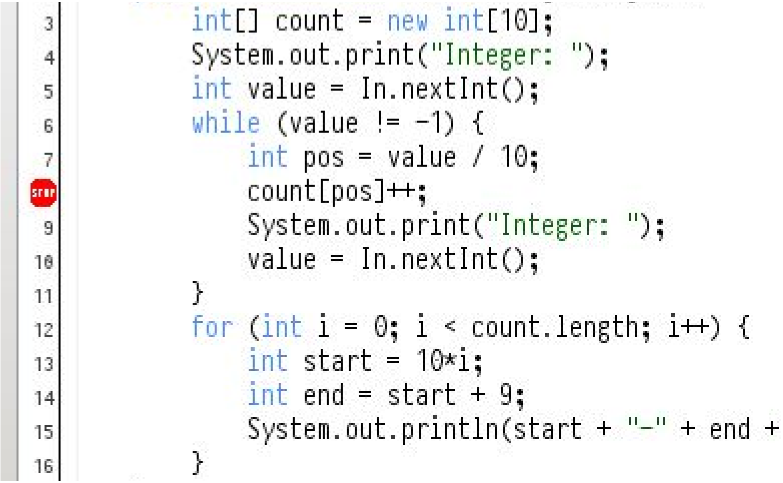
```
while (value != -1) {
     count[value / 10]++;
     System.out.print(“Integer:
“);
     value = In.nextInt();
}
```

To this…

```
while (value != -1) {
     int pos = value / 10;
     System.out.println(“pos = “ + pos);
     count[pos]++;
     System.out.print(“Integer: “);
     value = In.nextInt();
}
```

The BlueJ Debugger

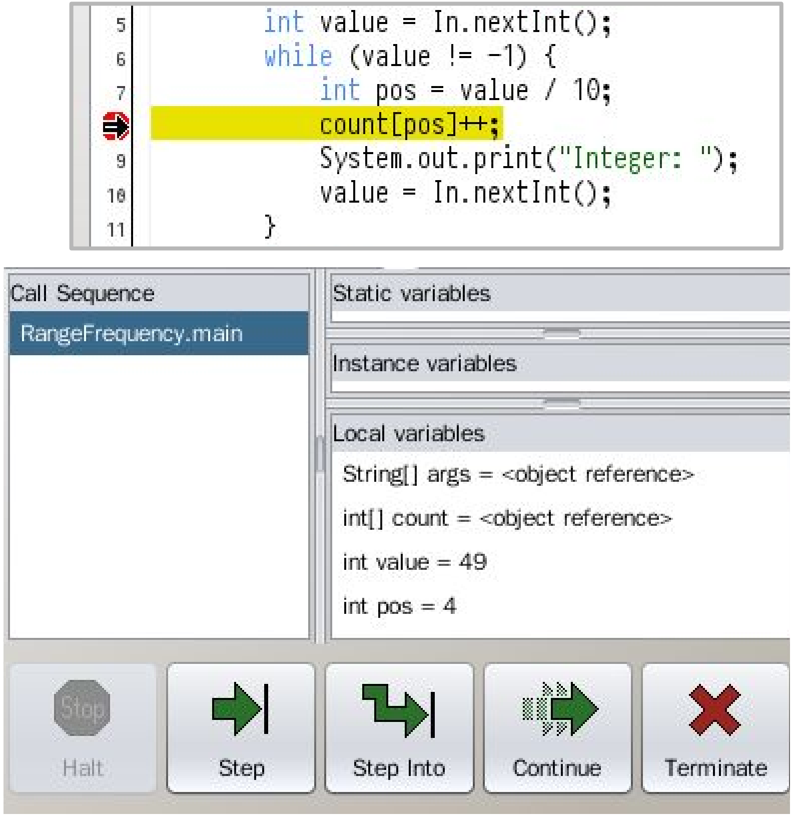
- Set a break point
  Click on the left-margin
  Must be executable code
  A red stop sign appears
  The run stops at this point
- See What code is executed
  See the call stack
  Trace the execution
- See the values of the variables
  at any point in execution



```
int[] count = new int[10];
System.out.print("Integer: ");
int value = In.nextInt();
while (value != -1) {
    int pos = value / 10;
    count[pos]++;
    System.out.print("Integer: ");
    value = In.nextInt();
}
for (int i = 0; i < count.length; i++) {
    int start = 10*i;
    int end = start + 9;
    System.out.println(start + "-" + end +
}
```

A checkpoint

- The run stops at the check point
  You see a black arrow.
- You see the current variable values
  `value` is 49
  `value / 10` is stored in `pos`
  We can see `pos` has the correct value 4
- You step through the code
  the arrow moves forward
  the variable values change
- “Step” moves to the next line.
  “Step Into” moves into a method.



```
        int value = In.nextInt();
        while (value != -1) {
            int pos = value / 10;
            count[pos]++;
            System.out.print("Integer: ");
            value = In.nextInt();
        }
```

Call Sequence

RangeFrequency.main

Static variables

Instance variables

Local variables

String[] args = <object reference>

int[] count = <object reference>

int value = 49

int pos = 4