Dr David A. McMeekin

Looping & Arrays

COMP1007

Acknowledgement of Country

"We acknowledge the Wadjuk Noongar people as the traditional custodians of the land on which Curtin University's Perth Campus sits, and pay our respect to elders past, present and emerging."

COMP1007 - Unit Learning Outcomes

- Identify appropriate primitive data types required for the translation of pseudocode algorithms into Java;
- Design in pseudocode simple classes and implement them in Java in a Linux command-line environment;
- Design in pseudocode and implement in Java structured procedural algorithms in a Linux command-line environment;
- Apply design and programming skills to implement known algorithms in real world applications; and
- Reflect on design choices and communicate design and design decisions in a manner appropriate to the audience.

Outline

- WHILE loops;
- DO-WHILE loops;
- Validation;
- FOR loops;
- Arrays;
- for-each loops; and
- Exceptions

WHILE Loops

Repetition AKA Looping

- Loop: a block of code repeated 0 to many times;
- Three available loops are: repetition control is different:
  1. WHILE:
     - Executes zero or more times
  2. DO-WHILE:
     - Executes one or more times
  3. FOR:
     - Executes a fixed number of times
- Choose the appropriate loop based what you want to do.

WHILE Loop

- Repetition controlled by a logical expression at top of loop;
- If logical expression is true, enter the loop execute the code;

```
WHILE boolExpression DO
    Body of loop
ENDWHILE
```

```
while(boolExpression)
{
    statements;
}
```

WHILE Loop (2)

- Logical expression checked before entering the loop:
  - If logical expression **false** loop is **NOT** entered, program jumps to first statement after loop's body;
  - If logical expression **true**, loop **IS** entered, code in loop body executed.
- After executing the loop code, logical expression checked again:
  - If logical expression **IS** still **true**, code in loop body executed again;
  - If logical expression **false** the loop is **NOT** entered, program jumps to first statement after loop's body.

WHILE Loop: Menu Example - Pseudocode

```
WHILE NOT close DO
   OUTPUT 'Enter Choice'
   INPUT choice
   CASE choice OF
      a OR A
         OUTPUT 'You entered' choice
      e OR E
         OUTPUT 'You entered' choice
         close = TRUE
      DEFUALT
         OUTPUT 'Invalid Choice'
   ENDCASE
 ENDWHILE
```

WHILE Loop: Menu Example – Java

```java
import java.util.*;
public class WhileLoop
{
    public static void main(String[] args)
    {
        char choice;
        boolean close = false;
        while(!close)
        {
            Scanner input = new Scanner(System.in);
            System.out.print("Enter letter: ");
            choice = input.next().charAt(0);
            switch(choice)
            {
                case 'a': case 'A':
                    System.out.println("You entered:" + choice);
                break;
                case 'e': case 'E':
                    System.out.println("You entered:" + choice);
                    close = true;
                break;
                default:
                    System.out.println("Invalid Choice");
            }
        }
    }
}
```

Infinite Loop

- One that can never end;
- Three major causes:
  1. Logical expression can never be false (logical error);
  2. Variable within logical expression never changes inside loop code (logical error); or
  3. Semi-colon in the wrong place (Syntax error).

Infinite Loop (2)

- Good assertion statements usually mean that:
  - Infinite Loops rarely occur within your algorithm;
  - Infinite Loops occur because of typos;
  - REASON: you see what should be true for the loop to stop.

Logical Error (1)

```
x = 0
WHILE x NOT EQUAL TO 11 DO
    OUTPUT x
    INCREMENT x BY 2
ENDWHILE
ASSERTION: x is equal to 11
```

```
x = 0
WHILE x < 11 DO
    OUTPUT x
    INCREMENT x BY 2
ENDWHILE
ASSERTION: x >= 11
```

Logical Error (2)

- Variable in logical expression never changes, will never be false.

```
INPUT x
WHILE x < 0 OR x > 10 DO
    OUTPUT "Invalid Input"
ENDWHILE
ASSERTION: 0 <= x <= 10
```

```
x = 0
WHILE x < 11 DO
    OUTPUT x
ENDWHILE
ASSERTION: x >= 11
```

```
INPUT x
WHILE x < 0 OR x > 10 DO
    OUTPUT "Invalid Input"
    INPUT x
ENDWHILE
ASSERTION: 0 <= x <= 10
```

```
x = 0
WHILE x < 11 DO
    OUTPUT x
    INCREMENT x BY 2
ENDWHILE
ASSERTION: x >= 11
```

Syntax Error

- Semi-colon in wrong place, loop body is now outside the loop;

```
evensSum = 0;
nextNo = 0;
while(nextNo <= 100);
{
    evensSum = evensSum + nextNo;
    nextNo += 2; // add two to nextNo
} // Assertion: nextNo > 100
System.out.println(evensSum);
```

- Loop ends after semi-colon following while loop i.e., hence, no statements in the loop;
- Boolean expression continually checked, nothing else.

DO-WHILE Loops

DO-WHILE

- Repetition controlled by a logical expression at bottom of loop;
- Loop body executes once before logical expression is checked;
- If logical expression is true the loop code executes again.

```
DO
    Body of loop
WHILE boolExpression
```

```
do
 {
     statements;
 } while(boolExpression);
```

DO-WHILE (2)

- Logical expression **NOT** checked before entering the loop:
  - Loop executed once prior to logical expression evaluation;
  - If logical expression **true**, execute code inside loop again;
  - If logical expression is **false**, loop is finished, moves to first statement after loop body.
- Logical expression is repeatedly checked after last statement in loop is executed.

Example Algorithm

- What is potentially wrong with this algorithm?

```
DO
    INPUT age
WHILE age <= 0 OR age >= 110
ASSERTION: 0 < age < 110
```

Example: Java

```
int age;
Scanner sc = new Scanner(System.in);
do
{
    System.out.println("Enter Age");
    age = sc.nextInt();
} while((age <= 0) || (age >= 110));
// Assertion: 0 < age < 110
```

- The logic is correct, but no indication given to the user of what went wrong

Example: Possible Solution

- A possible starting template for you to use:
- NB: it will evolve as we cover methods, and Exceptions.

```
DO
    DISPLAY 'Please enter a value between X & Y'
    num = GET num
WHILE((num < x) OR (num > y))
ASSERTION: lower <= value <= upper
```

- The displayed message (prompt) works even on first loop;
- Creating accurate messages here is difficult.

Loop Equivalency

- A WHILE loop can be expressed as a DO-WHILE loop:

```
WHILE x < 10 DO
    INCREMENT x BY 2
ENDWHILE
ASSERTION: x >= 10
```

```
IF x < 10 THEN
    DO
        INCREMENT x BY 2
    WHILE x < 10
    ASSERTION: x >= 10
ENDIF
ASSERTION: x >= 10
```

- A DO-WHILE loop can be expressed as a WHILE loop:

```
DO
    INCREMENT x BY 2
WHILE x < 10
ASSERTION: x >= 10
```

```
INCREMENT x BY 2
WHILE x < 10 DO
    INCREMENT x BY 2
ENDWHILE
ASSERTION: x >= 10
```

Think before you design

FOR Loops

FOR Loop

- Is an extremely useful loop;
- Pseudo Code:

```
FOR count = startVal TO stopVal CHANGEBY increment
    OTHER_ACTIONS
ENDFOR
```

- The variable increment can be positive or negative; and
- count is known as the for loop index.

Properties of a FOR Loop

- Loop index should always be a local variable;
- Loop index **NEVER** a Real number;
- Loop index **NEVER** explicitly modified inside the loop;
- The value of the loop index is undefined outside of the loop;
- for loop never executes if:
  - positive increment and stopVal ‹ startVal;
  - negative increment and startVal ‹ stopVal.

FOR Loops in Java

- Syntax:

```
for(initialisation; booleanExpression; increment)
{
    body_of_loop;
}
```

- Example:

```
sum = 0;
for(int count = 0; count < 10; count++)
{
    System.out.println("Count is: " + count);
    sum += count;
}
```

Declaring Loop Indexes

- Good programming practice: all variables declared at method block start;
- Loop index is an exception, never referred to outside the for loop;
- Java allows variable declarations anywhere:

```
sum = 0;
for(int count = 0; count < 10; count++)
{
    System.out.println("Count is: " + count);
    sum += count;
}
```

- Referring to loop index outside for loop incurs a compiler error.

What NOT to Use in Loops

- Three statements that should NOT be used but should in loops:
  - **break** exit loop
  - **continue** skip to next iteration of the loop
  - **goto** go to LABEL (but not in Java).

```
for( ; ; )                          // Infinite Loop
{
    ...
    if(cond1) continue;             // Start the next Iteration
    else if(cond2) break;           // Exit the loop now
    else if(cond3) goto FRED;       // Go to label FRED (Somewhere)
    ...                             // Neither cond1 or cond2 true
}
```

- Programming languages allow poor algorithm design and programming style;
- Design great algorithms using great programming style.

FOR Loop Example

- Pseudocode:

```
FOR index = 0 TO userNumber LENGTH CHANGEBY 1
  OUTPUT userNumber * 2
ENDFOR
```

- Java:

```java
Scanner input = new Scanner(System.in);
System.out.print("Enter an integer: "); //Prompt for user input
int userNumber = input.nextInt();
for(int i = 0; i < userNumber; i++)
{
    System.out.println(i * 2);
}
//ASSERTION: Output from 0 - userNumber will be doubled
```

FOR Loop Example (2): Pseudocode

```
ALGORITHM:
nFactorial = 1
FOR i = 2 TO n CHANGEBY 1
    nFactorial = nFactorial * i
ENDFOR

ALTERNATE ALGORITHM:
nFactorial = 1
FOR i = n DOWNTO 2 CHANGEBY -1
    nFactorial = nFactorial * i
ENDFOR
```

FOR Loop Example (2): Java

```java
/*
* ASSERTION: if n 0 or negative, then nFactorial is 1 *
*/
int n = 5;
long nFactorial = 1;
for(int i = 2; i <= n; i++)
{
    nFactorial *= (long)i;
    System.out.println(nFactorial);
}
// ----------------------------------------------------
// Try this one
long nFactorial = 1;
for(int i = n; i >= 2; i--)
{
    nFactorial *= (long)i;
}
```

Arrays

Arrays

- Variables represent a single item:
  - e.g., `int numTimes;` is a single integer number
- We also work with sets of similar data:
  - e.g., a list of student marks in PDI;
  - How do we work with this?
  - `double student1Mark, student2Mark, ..., studentXMark;`
- Calculating the average involves a massive amount of typing;
- Can't conveniently pass the student set around.

Arrays (2)

- Arrays solve this problem;
- A simple data structure to store sets of data;
- Arrays are built-in to all programming languages; and
- An array is a variable that contains many elements.

Array Properties

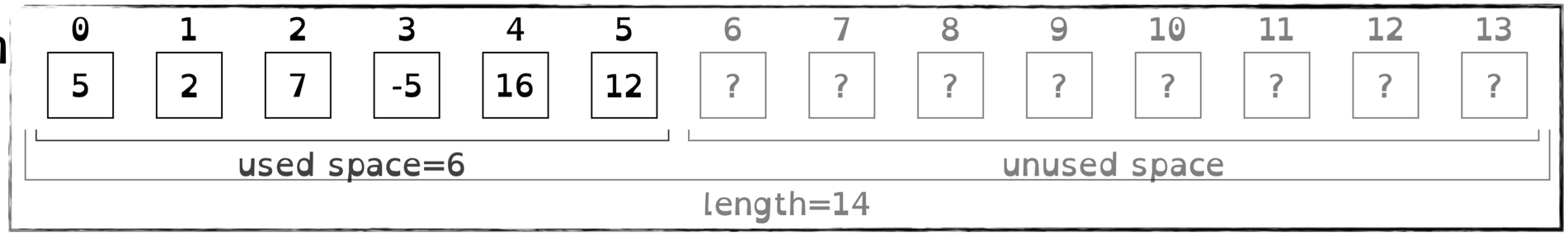
- Elements located sequentially in memory:
  - an array is a contiguous block of memory.
- All elements must have same data type:
  - e.g., double
- Arrays can be initialised to any size (within memory limits);
- Once initialised they CANNOT be resized;
- A new array must be created and the old array contents copied over to it.

More Array Properties

- Array capacity (length) vs actually used elements;
- An array initialised to `length = 20;` reserves space for 20 elements;
- Initialisation is also referred to as allocation;
- Typically you need to keep track of how many array elements are actually used in the array:
  - i.e., the count of used elements, as distinct from array size;
  - Generally allocate more space than thought required, as arrays have a fixed capacity and cannot be resized.

Arrays – Accessing Elements

- Once created, you need to work with the array elements;
- Elements are accessed via an index or subscript;
- The index is the element number in the array;
- Arrays are numbered: **0 to N-1**, **N** is the allocated length;
- To access an element: **theArrayName[elementNumber]**
- In th



Declaring and Allocating Arrays in Java

- Declaring arrays use: []
  ```
  double[] theArray;
  ```
  - Arrays of any data type can be created (including classes);

- Allocating arrays: use keyword **new** with [] syntax:
  ```
  theArray = new double[100];
  ```

- Declare and allocate at the same time also possible:
  ```
  double [] theArray = new double[100];
  ```

  theArray now has 100 elements of data type **double**;

Accessing and Copying Arrays

- To access elements: **theArray[index]**, **index** must be a positive int:
  - **index** is the element to access in the array;
  - Negative indexes or indexes past the array end (i.e., **>=** length) cause a runtime error.
- Assignment: **sameArray = theArray;** does NOT copy **theArray** into **sameArray;**
  - **sameArray** points to **theArray;**
  - The L.H.S variable points to the R.H.S variable;
  - Same when an array is a method parameter (covered later).

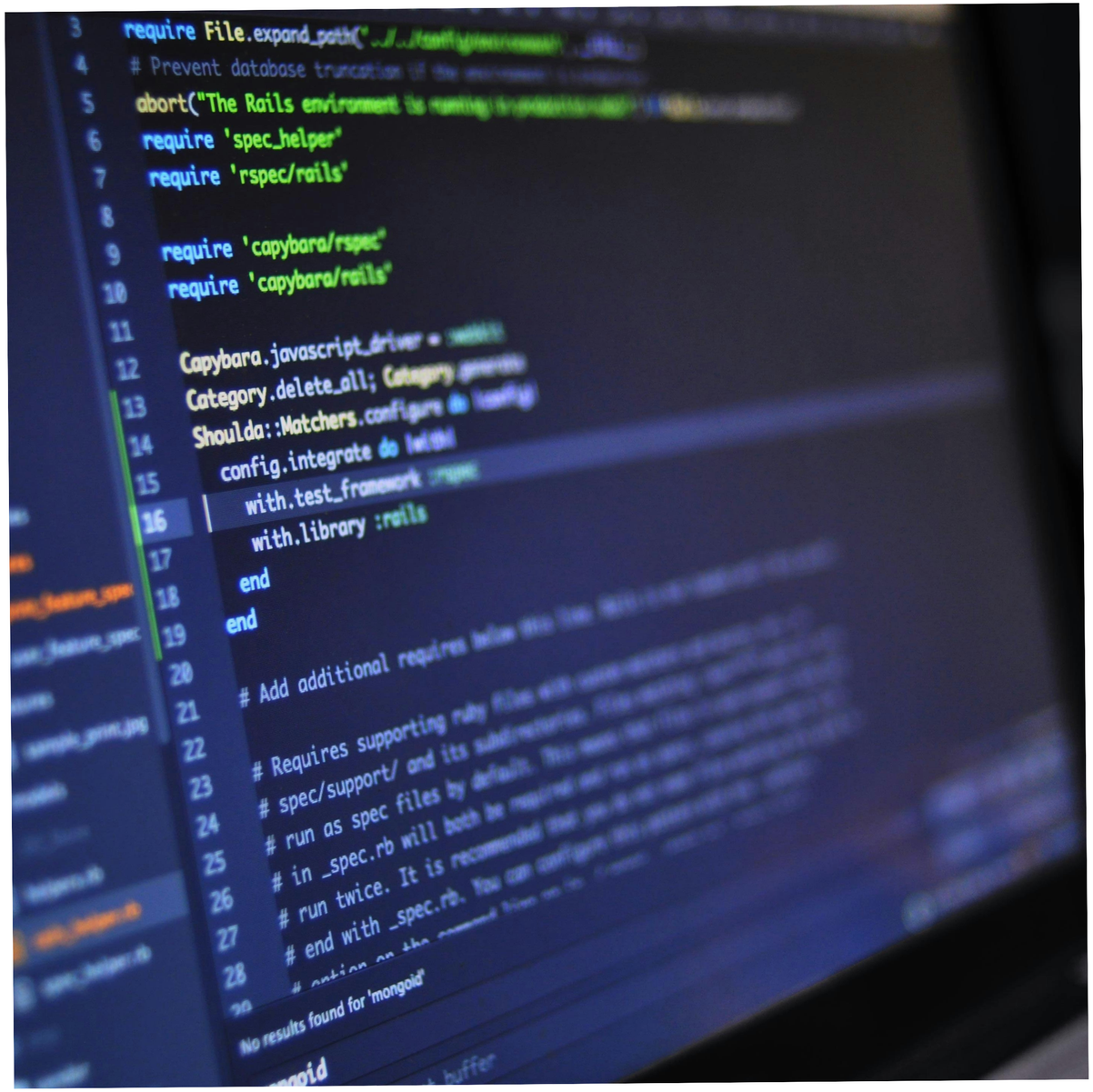
Java - Arrays

```
import java.util.*;
public class UsingArrays
{
    public static void main(String[] args)
    {
        int [] theArray;
        theArray = new int[100];
        int theArrayLength = theArray.length;

        for(int i = 0; i < theArrayLength; i++)
        {
            theArray[i] = i * i;
        }

        for(int i = 0; i < theArrayLength; i++)
        {
            System.out.println("theArray["+i+"] is: " + theArray[i]);
        }
    }
}
```

for-each Loop



for-each Loop

- also known as the **enhanced for loop;**
- used to iterate over arrays and collections;
- simple easily readable (compared to traditional for loop, some say).

Structure of the for-each Loop

```
for (Type var : array)
{
    // statements using var
}
```

- **Type:** the type of the items in the array/collection;
- **var:** the variable representing current item in each iteration;
- **array:** the array or collection to iterate over

Code Example: for-each Loop

```java
public class MyForEach
{
    public static void main(String[] args)
    {
        // Example of For-Each Loop
        // for-each loop iterates over an array of integers:

        int[] numbers = {1, 2, 3, 4, 5};
        for (int number : numbers)
        {
            System.out.println(number);
        }
    }
}
```

- Prints each number in the **numbers** array on a new line.

for-each: Advantages & Limitations

- Advantages:
  - Succinct & readable iterating over entire arrays/collections;
  - Eliminates bugs related to incorrect initialisation or incrementing loop counters;

- Limitations:
  - Cannot be used to initialise or modify array/collection elements;
  - Cannot be used to iterate over multiple collections or arrays in parallel;
  - Cannot obtain the current element index;

Validation & Exceptions

Validating User Input

- Programs must protect against a unique error: ID10T;
- Is the input correct?
- Is the input actually correct?
- Are you sure the input is really correct?

Can You Repeat that Please?

- Not all users get it right the first time;
- Not all files actually exist;
- Not all files that exist permit you to access them;
- Not all data files contain the correct data;
- Validating the input is crucial and can save lives.

Validation: IF Statement - Pseudocode

```
number = 0
'Enter number between 1 and 6'
GET number
IF number between 1 and 6 THEN
    the magic happens here here
    run your code
ELSE
    PRINT 'Input was not in the required range'
ENDIF
```

Validation: IF Statement - Java

```
import java.util.*;
public class IfValidation
{
    public static void main(String[] args)
    {
        int number = 0;
        Scanner sc = new Scanner(System.in);
        System.out.print("Enter a number between 1 and 6: ");
        number = sc.nextInt();
        if(number < 7 && number > 0)
        {
            System.out.print("Look what happens now: AMAZING");
        }
        else
        {
            System.out.println("Input was outside of range!");
        }
    }
}
```

Validation: Using a loop - Pseudocode

```
DO
    'Enter number between 1 and 6'
    GET number
    the magic happens here here
    run your code
WHILE number NOT between 1 and 6
END DO-WHILE
```

Validation Using a loop - Java

```
import java.util.*;
public class WhileValidation
{
   public static void main(String[] args)
   {
        int number = 0;
        Scanner sc = new Scanner(System.in);
        do
        {
            System.out.print("Enter a number between 1 and 6: ");
            number = sc.nextInt();
        }while(number > 7 && number < 0);
   }
}
```

Exceptions

- An exception is an event that DOES NOT follow a rule;
- It is something that was NOT meant to happen;
- Possible exceptions:
  - user enters wrong data type;
  - file doesn't exist;
  - divide by zero.
- Dealing with it when it happens is called Exception Handling.

Java and Exceptions

- Java handles errors using exceptions;
- When an error occurs the program “throws an exception”;
- The program should “catch” the exception;
  - The calling method can “catch” the exception; (methods covered later)
  - If the caller doesn’t catch it, exception is thrown to the next highest caller;
- If the program does not “catch” the exception it will cause the program to crash.

Catching Exceptions

- Often all caught at the one place in the calling method;
- Often close to main (not always);
- Convenient to handle errors all in one place;
- Most languages use **try**, **catch()** and **finally** blocks;
  - **try:** define a set of statements whose exceptions will be handled by the catch block associated with this try;
  - **catch():** processing to do if an exception is thrown;
  - **finally:** processing to always do, regardless of exception or not.

try and catch

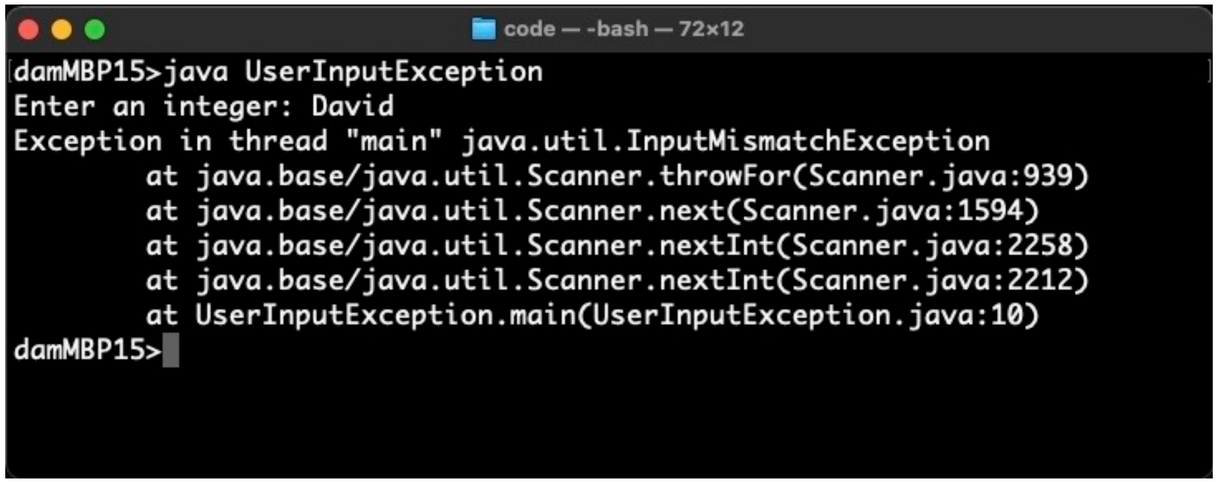
```
try
{
    //code that may throw an exception
}
catch(Exception someException)
{
    //code execute when exception happens
}
```

A Java Program - no Exception Handling

```
import java.util.*;
public class UserInputException
{
    public static void main(String[] args)
    {
        int number = 0;
        Scanner sc = new Scanner(System.in);
        System.out.print("Enter an integer: ");
        number = sc.nextInt();
        System.out.println("The integer is:" + number);
    }
}
```

Exception Message

- Displayed by Java when the error was preventable:



- Program asked for a number, user entered ‘David’

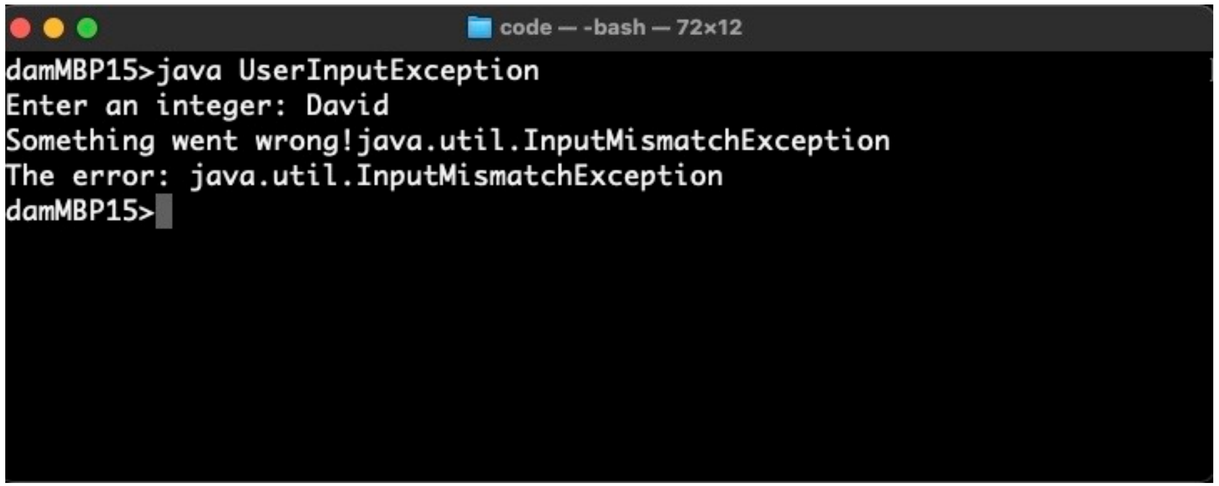
```
Exception in thread "main" java.util.InputMismatchException
```

A Java Program – with Exception Handling

```
import java.util.*;
public class UserInputException2
{
    public static void main(String[] args)
    {
        int number = 0;
        Scanner sc = new Scanner(System.in);
        try
        {
            System.out.print("Enter an integer: ");
            number = sc.nextInt();
            System.out.println("The integer is:" + number);
        }
        catch(InputMismatchException error)
        {
            System.out.println("Something went wrong!" + error);
            System.out.println("The error: " + error);
        }
    }
}
```

Error Handled by Exception Handling

- A clean error message to the user;
- Program did **NOT** crash;
- Program exited gracefully;
- Message could be written to log file.



Arrays and Exceptions

- Arrays are of fixed length;
- In Java, memory outside the array can NOT be accessed;

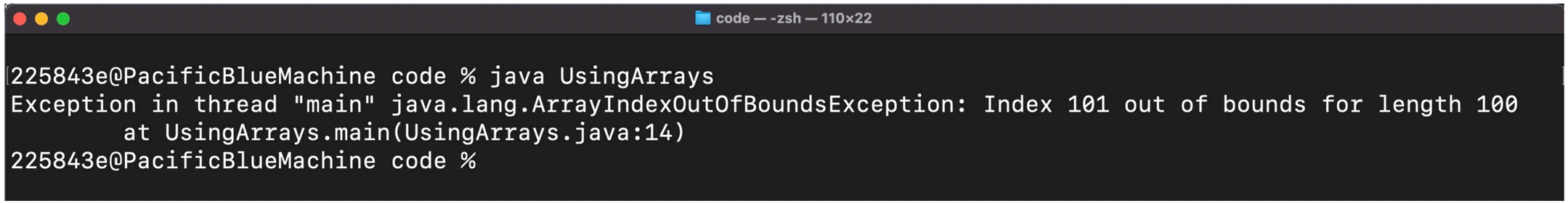
```
int [] theArray;
theArray = new int[100];
int theArrayLength = theArray.length;

for(int i = 0; i < theArrayLength; i++)
{
    theArray[i] = i * i;
}
System.out.println("theArray[101] is: " + theArray[101]);
```

- **theArray[101]** is invalid;
- An exception will be thrown.

Unhandled Exception Message

- Displayed by Java:



- Array has 100 elements, attempted to access element 101:

  `Exception in thread "main" java.lang.ArrayIndexOutOfBoundsException`

Exception Handling with Arrays

```
import java.util.*;
public class UsingArrays2
{
    public static void main(String[] args)
    {
        int [] theArray;
        theArray = new int[100];
        int theArrayLength = theArray.length;
        try
        {
            for(int i = 0; i < theArrayLength; i++)
            {
                theArray[i] = i * i;
            }
            System.out.println("theArray[101] is: " + theArray[101]);
            for(int i = 0; i < theArrayLength; i++)
            {
                System.out.println("theArray["+i+"] is: " + theArray[i]);
            }
        }
        catch(ArrayIndexOutOfBoundsException error)
        {
            System.out.println("Something went wrong!");
            System.out.println("The error: " + error);
        }
    }
}
```

Image References

https://www.imdb.com/title/tt4788946/mediaviewer/rm1262069505/

Photo by **Luis Zambrano** from **Pexels**

Photo by **Parakit Keng Eos** from **Pexels**

Photo by **Ricardo Esquivel** from **Pexels**

Photo by Jason Leung on Unsplash