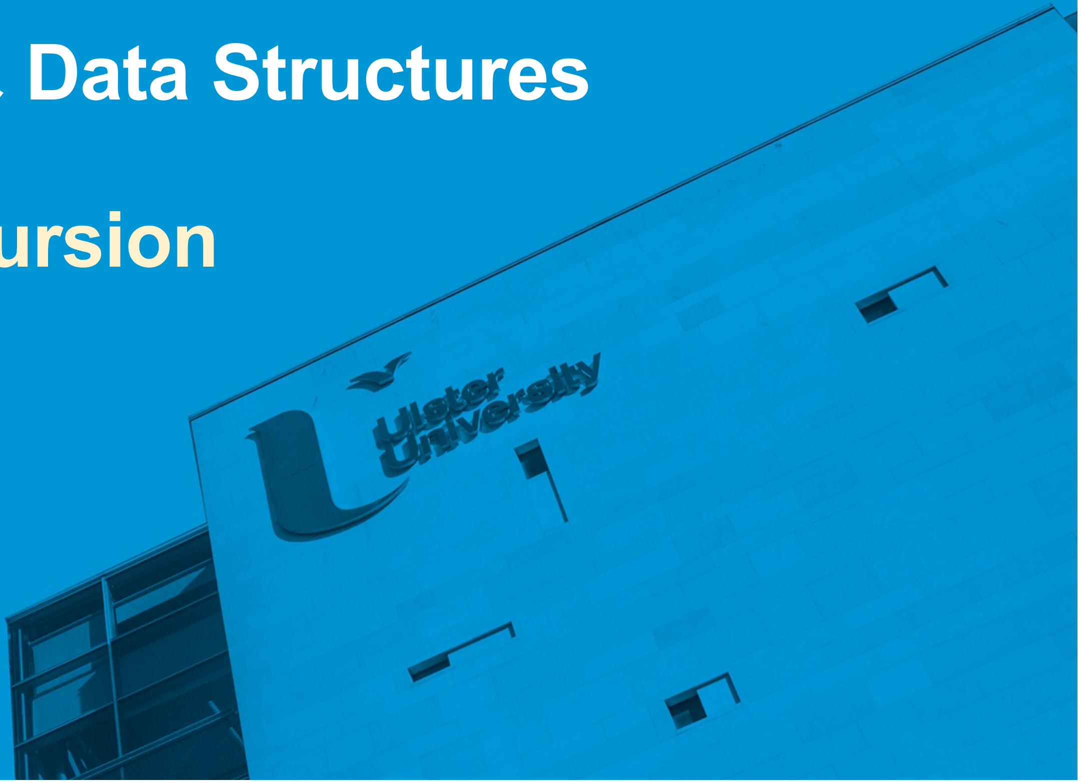
COM498 Algorithms & Data Structures

4.2 Implementing Recursion



Example: Binary Conversion

- Recursive method to convert an integer to binary:

```
public class Binary
{
   public static String toBinary(int n) {
      if (n == 1) return “1";
      else return toBinary(n / 2) + (n % 2);
   }

   public static void main(String[] args){
      System.out.println(toBinary(26));
   }
}
```

Example: Binary Conversion

- **Convert 6 to binary**
  - **6** divided by 2 is **3** remainder 0
  - **3** divided by 2 is **1** remainder 1
  - **1** divided by 2 is **0** remainder 1

Remainder values in reverse order gives **110**

- **Convert 10 to binary**
  - **10** divided by 2 is **5** remainder 0
  - **5** divided by 2 is **2** remainder 1
  - **2** divided by 2 is **1** remainder 0
  - **1** divided by 2 is **0** remainder 1

Remainder values in reverse order gives **1010**

Example: Binary Conversion

- Recursive method to convert an integer to binary:

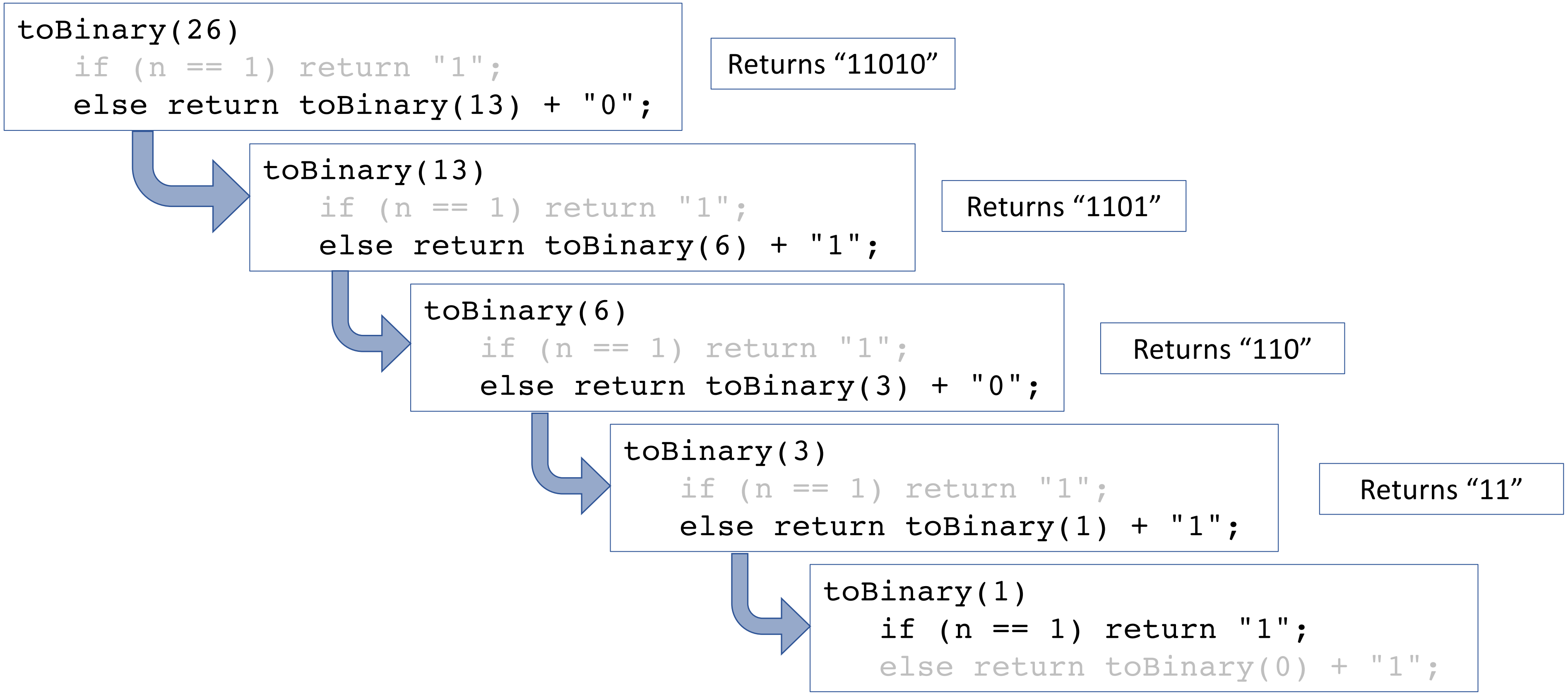
```
public class Binary
{
   public static String toBinary(int n) {
      if (n == 1) return “1";
      else return toBinary(n / 2) + (n % 2);
   }

   public static void main(String[] args){
      System.out.println(toBinary(26));
   }
}
```

Base Case

Recursive Call

Example: Binary Conversion



```
toBinary(26)
   if (n == 1) return "1";
   else return toBinary(13) + "0";
```

Returns “11010”

```
toBinary(13)
   if (n == 1) return "1";
   else return toBinary(6) + "1";
```

Returns “1101”

```
toBinary(6)
   if (n == 1) return "1";
   else return toBinary(3) + "0";
```

Returns “110”

```
toBinary(3)
   if (n == 1) return "1";
   else return toBinary(1) + "1";
```

Returns “11”

```
toBinary(1)
   if (n == 1) return "1";
   else return toBinary(0) + "1";
```

Scenario

- Implement a first recursive application
  - Create a new Java project called Recursion and create a new class called `Decimal2Binary` in a file called *Decimal2Binary.java*
  - In the new class, implement the static method `toBinary()` that takes a decimal integer as a parameter and returns a `String` that shows the binary representation of the parameter.
  - Verify the `toBinary()` method by providing a `main()` method that prints the result of 3 calls to `toBinary()` with different positive integer values.

Programming with Recursion

- Some typical bugs when programming with recursion:

```
public static double bad(int n)
{
   return bad(n – 1) + (1.0 / n);
}
```

No base case given!

```
public static double bad(int n)
{
   if (n == 1) return 1.0;
   else return bad(1 + n / 2) +
               (1.0 / n);
}
```

No convergence guarantee!
Consider n = 2

- Both cases potentially lead to infinite recursive loops!

Recursive Method Design Questions

```
public static void countDown(int n) {

    System.out.println(n);
    if (n > 1) countDown(n - 1);
}
```

- What part of the solution can you contribute directly?

  In `countDown()` the method displays the given integer as the part of the solution that it contributes directly

- What smaller but identical problem has a solution that, when taken with your contribution, provides the solution to the original problem?

  The smaller problem is counting down from `n` – 1. The method solves the smaller problem when it calls itself recursively

- When does the process end?

  The `if` statement asks if the process has reached the base case, which occurs when `n` = 1

Some Design Guidelines

- Method must be given an input value (usually as an argument)
- Method definition must contain logic that involves this input value and leads to different cases (such logic usually involves an `if` or `switch` statement)
- One or more of these cases should provide solution that does not require recursion (the base cases)
- One or more cases must include a recursive invocation of the method (these should in some sense make a step towards the base case by using solving smaller version of the task performed by the method)
- Where the method returns a value, each case **must** provide a value for the method to return

Recursive Methods That Return a Value

$$\sum_{i=1}^{n} i$$

- Compute the sum **1 + 2 + . . . +** ***n*** for any integer ***n*** > 0

- The given input for the problem is integer ***n***

- What small value of *n* results in a sum that you know immediately? (i.e. what is the base case? How do we know when we are finished?) One possible answer is 1 (if *n* = 1, sum = 1)

- What should the smaller version of the task be? Compute the sum **1 + . . . + (*n* – 1)** (adding **n** to this computation will solve the original problem)

- Leads to the following definition of `sumOf()`

```
public static int sumOf(int n) {

    if (n == 1) return 1;
    else return n + sumOf(n – 1);
}
```

Tracing the Recursive Method `sumOf()`

- Suppose we invoke the method with the statement: `sumOf(3);`
- The computation occurs as follows:

```
public static int sumOf(int n) {

    if (n == 1) return 1;
    else return n + sumOf(n – 1);
}
```

a) `sumOf(3)` is 3 + `sumOf(2)`
(`sumOf(3)` suspends execution and `sumOf(2)` begins)

b) `sumOf(2)` is 2 + `sumOf(1)` (`sumOf(2)` suspends execution and `sumOf(1)` begins)

c) `sumOf(1)` returns 1 (base case is reached so suspended execution resumes with most recent activation)

d) `sumOf(2)` returns 2 + 1 = 3

e) `sumOf(3)` returns 3 + 3 = 6

Scenario

- Add an additional recursive demonstration
  - In the `Recursion` project, add a new class `sumOf` in a file called *sumOf.java* that contains a method `sumOf()` that takes a single integer parameter and returns the sum of all positive integers up to and including that value.
  - Verify the `sumOf()` method by providing code to a `main()` method in the `sumOf` class that prints the result of 3 calls to `sumOf()` with different positive integer values.

Recursively Processing an Array

- Some of the more powerful searching and sorting algorithms are often stated recursively
- A common simple use of recursion is to display elements in an array (within a given range):

```
public static void displayArray(int[] array, int first, int last)
```

- There are a variety of ways to implement `displayArray()` recursively:

Starting with **`array[first]`**:

```
public static void displayArray(int arr[], int first, int last) {
    System.out.print( arr[first] + “ “);
    if (first <= last) displayArray (arr, first + 1, last);
}
```

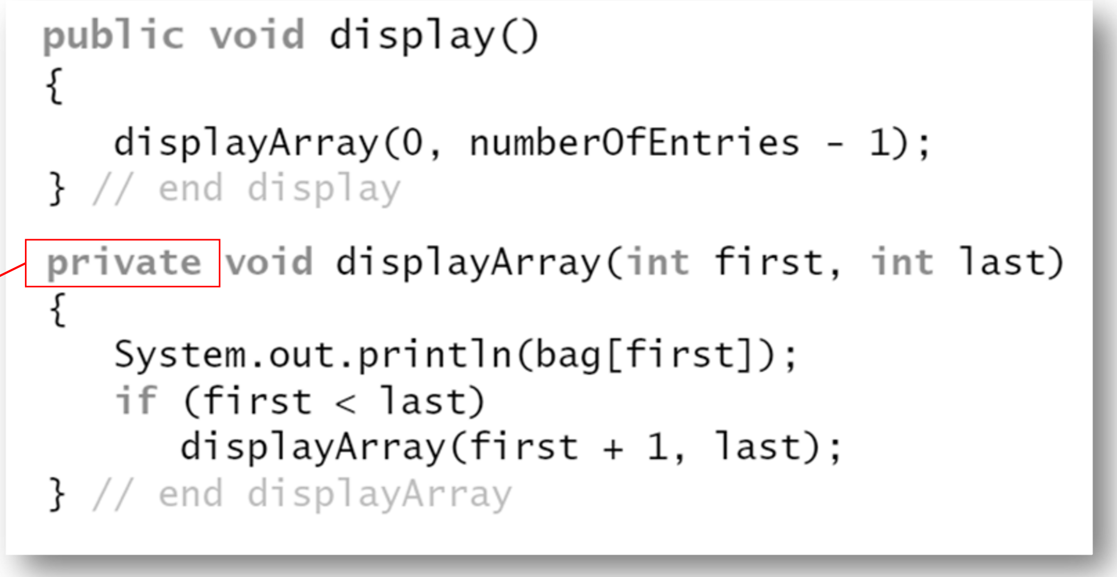
Starting with **`array[last]`**:

```
public static void displayArray(int arr[], int first, int last) {
    System.out.print( arr[last] + “ “);
    if (first <= last) displayArray (arr, first, last - 1);
}
```

Recursively Displaying a Bag

- Supposing our array-based implementation of the ADT bag has a `display()` method
- As it has no parameters, it must call another method that use the data fields of the bag:
- The arguments to the call would be 0 for first index and `numberOfEntries` - 1 for last index

A method that uses required knowledge of the underlying data structure of an ADT should be private

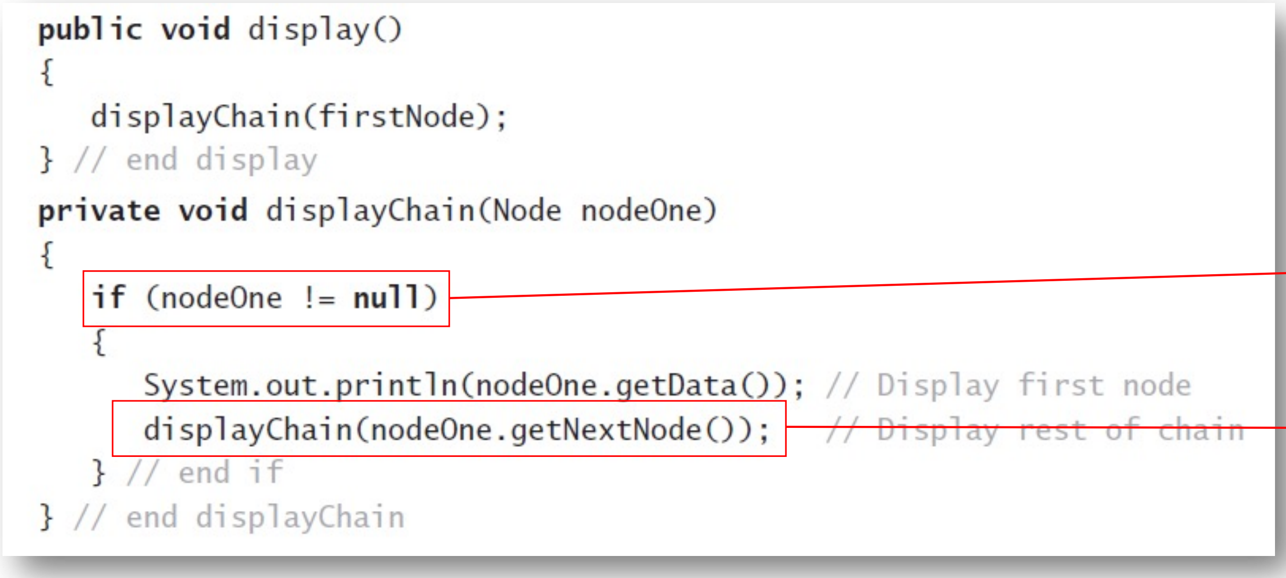


```
public void display()
{
   displayArray(0, numberOfEntries - 1);
} // end display

private void displayArray(int first, int last)
{
   System.out.println(bag[first]);
   if (first < last)
      displayArray(first + 1, last);
} // end displayArray
```

Recursively Processing a Linked Chain

- Recursion can also be used to process (display data) a linked chain of nodes
- Consider the linked chain implementation of ADT bag has a `display()` method:

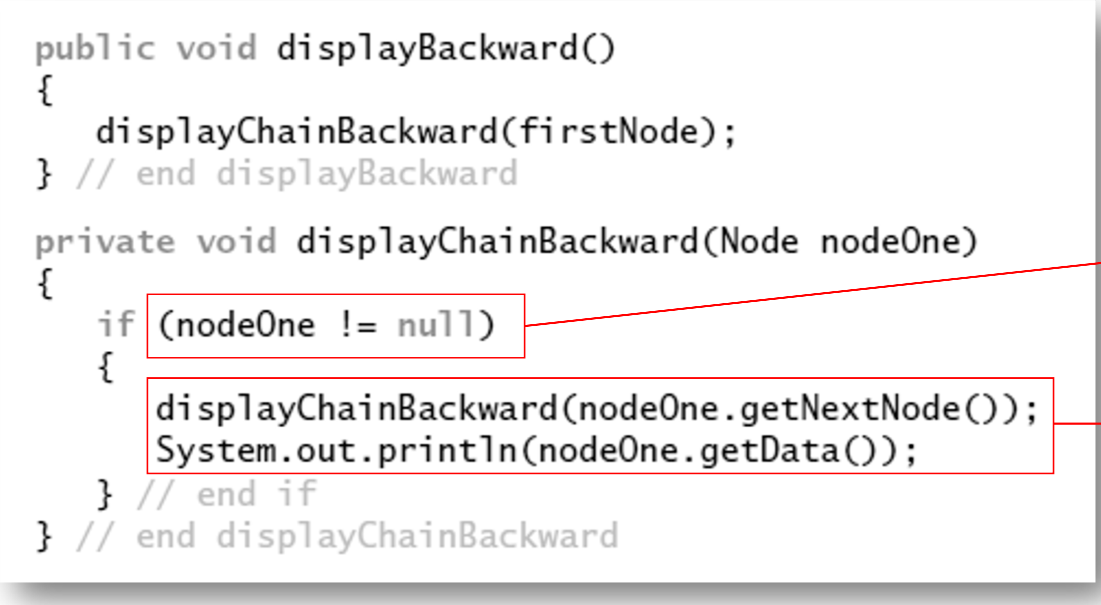


```
public void display()
{
   displayChain(firstNode);
} // end display
private void displayChain(Node nodeOne)
{
   if (nodeOne != null)
   {
      System.out.println(nodeOne.getData()); // Display first node
      displayChain(nodeOne.getNextNode());   // Display rest of chain
   } // end if
} // end displayChain
```

- Cannot access any particular node without traversal of chain from beginning (`nodeOne`)
- Empty chain (`nodeOne=null`) is the base case
- Approach displays data in `nodeOne` then recursively displays data in rest of chain

Recursively Processing a Linked Chain

- Recursion can also be used to easily traverse a linked chain in reverse order (difficult to do using iteration as each node only references the next node)
- Consider the implementation of ADT bag now has a `displayBackward()` method:



```
public void displayBackward()
{
   displayChainBackward(firstNode);
} // end displayBackward

private void displayChainBackward(Node nodeOne)
{
   if (nodeOne != null)
   {
      displayChainBackward(nodeOne.getNextNode());
      System.out.println(nodeOne.getData());
   } // end if
} // end displayChainBackward
```

- Again, empty chain (`nodeOne=null`) is the base case
- Approach traverses the chain *saving* references to each node, then uses the references to display the data in the nodes

Challenge

- **Adding functionality to the Bag class**
  - Revisit your `BagInterface` class and add the specification of a new void method `display()` that prints a representation of the `Bag` content
  - Provide an implementation of `display()` in both the `ArrayBag` and `LinkedBag` classes, using a recursive approach for each.
  - Test your implementation by modifying the `BagTest` class so that the line of code that prints the contents of the `Bag` (the first line of the `bagStatusReport()` method) calls the new `display()` method rather than an implicit call to `toString()`
    - Run the `BagTest` application twice – once for an instance of `ArrayBag` and once for an instance of `LinkedBag`