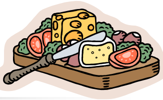
# CSI1102: Introduction to Software Design

**Chapter 12:**
**Data Structures**

# Learning objective: Data Structures

- Some convenient techniques for **organizing** and **managing** information
- Understand what the following entails:
  - Collections in Java
  - Abstract Data Types (ADTs)
  - dynamic structures and linked lists
  - Linear data structures: queues and stacks

# What is a Collection?



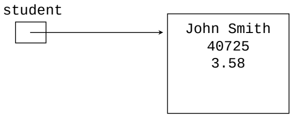
- A *collection* is **an object** that serves as a repository for other objects,
  - e.g. collection of students, CD, magazines, food
- A collection usually provides services such as adding, removing, and otherwise managing the elements it contains
  - Sometimes the elements in a collection are ordered, sometimes they are not
  - Sometimes collections are *homogeneous*, sometimes the are *heterogeneous*

# Abstract Data Types: Implementing a collection

- An *abstract data type* (ADT) is
  - an organized collection of information and
  - a set of operations used to manage that information
- The set of operations defines the *interface* to the ADT
- We implement an ADT using a *dynamic data structure*
  - A *dynamic data structure* grows and shrinks at execution time as required by its contents
  - A dynamic data structure is implemented using *links*
- *Question*: Is an Array a dynamic data structure?

# Object References: Used for ADTs

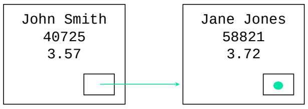
- Recall that an *object reference* is a **variable that stores the address of an object**
  - A reference also can be called a *pointer*
- References often are depicted graphically:



```
Student john = new Student(“John Smith…”);
```

# Object References as Links

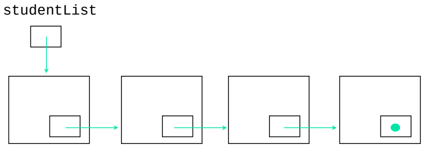
- Suppose a `Student` class contains a reference to another `Student` object



```
class Student
{
   STRecord info; // info about the student
   Student next;  // link to another Student object
}

Student john = new Student(“John Smith…”, null);
Student jane = new Student(“Jane Jones…”, null);
john.next = jane;
```

# References as Links: The Linked List

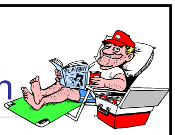
- References can be used to create a variety of linked structures, such as a *linked list*:



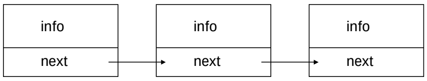
# The content of the Intermediate Nodes

- The objects being stored should not be concerned with the details of the data structure in which they may be stored
  - For example, the `Student` **class** should not have to store a link to the next `Student` **object** in the list
- Instead, we can use a separate **node class** with two parts:
  - 1) a reference to an independent object and
  - 2) a link to the next node in the list
- The internal representation becomes a linked list of nodes

# An example: A Magazine Collection



- Let's explore an example of a collection of `Magazine` objects
- The collection is managed by the `MagazineList` class, which has an private inner class called `MagazineNode`
- Because the `MagazineNode` is private to `MagazineList`, the `MagazineList` methods can directly access `MagazineNode` data without violating encapsulation



# MagazineRack.java

```
public class MagazineRack
   {
      //  Creates a MagazineList object, adds several magazines to the
      //  list, then prints it.

     public static void main (String[] args)
      {
         MagazineList rack = new MagazineList();

         rack.add (new Magazine("Time"));
         rack.add (new Magazine("Woodworking Today"));
         rack.add (new Magazine("Communications of the ACM"));
         rack.add (new Magazine("House and Garden"));
         rack.add (new Magazine("GQ"));

         System.out.println (rack);
      }
   }
```

# MagazineList.java

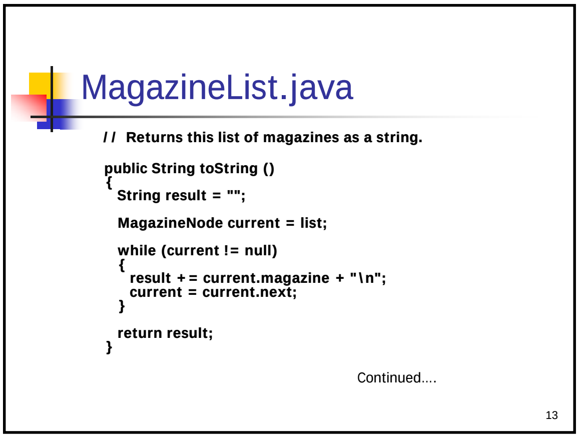
```
public class MagazineList
   {
     private MagazineNode list;

     //  Sets up an initially empty list of magazines.

      MagazineList()
     {
        list = null;
     }
```

Continued....

# MagazineList.java

```
//  Creates a new MagazineNode object and adds it to the end of the linked list.

      public void add (Magazine mag)
        {

         MagazineNode node = new MagazineNode (mag);
         MagazineNode current;

         if (list == null)    list = node;
         else
         {
            current = list;                    // we are at the list's beginning
           while (current.next != null)        // walk through the list to the end
                current = current.next;
            current.next = node;
         }
       }
```

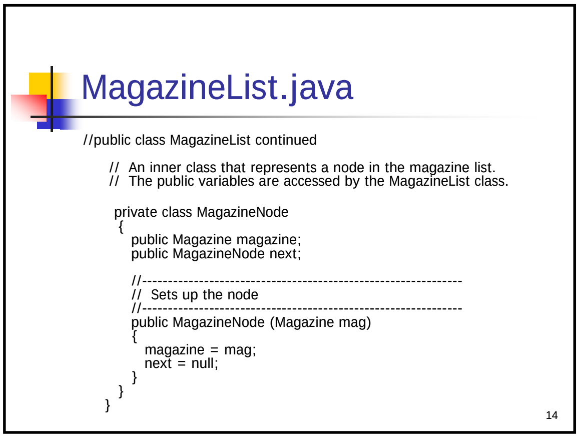
Continued....



## MagazineList.java

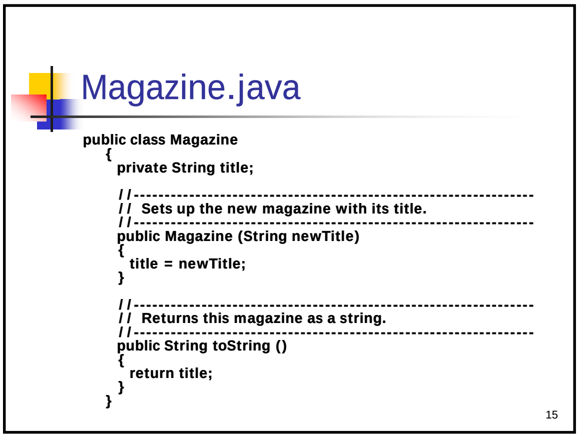
```
// Returns this list of magazines as a string.

public String toString ()
{
  String result = "";

  MagazineNode current = list;

  while (current != null)
  {
    result += current.magazine + "\n";
    current = current.next;
  }

  return result;
}
```

Continued....



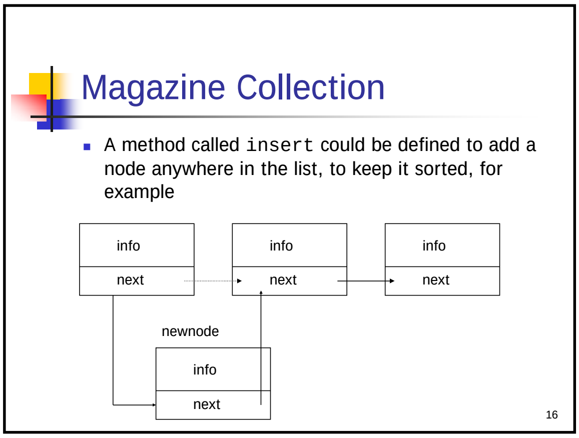
## MagazineList.java

```
//public class MagazineList continued

   //  An inner class that represents a node in the magazine list.
   //  The public variables are accessed by the MagazineList class.

   private class MagazineNode
   {
      public Magazine magazine;
      public MagazineNode next;

      //--------------------------------------------------------------
      //  Sets up the node
      //--------------------------------------------------------------
      public MagazineNode (Magazine mag)
      {
         magazine = mag;
         next = null;
      }
   }
}
```



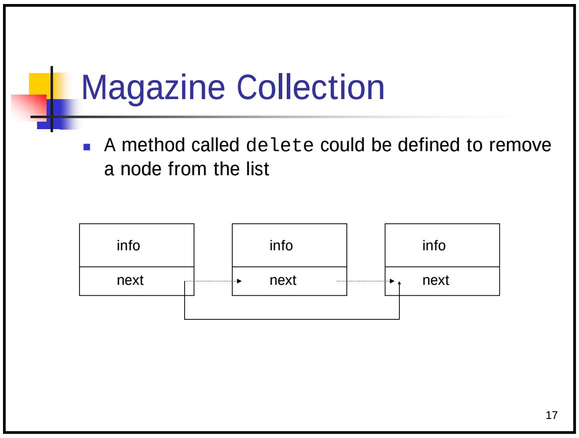
## Magazine.java

```
public class Magazine
{
   private String title;

   //-----------------------------------------------------------------
   //  Sets up the new magazine with its title.
   //-----------------------------------------------------------------
   public Magazine (String newTitle)
   {
     title = newTitle;
   }

   //-----------------------------------------------------------------
   //  Returns this magazine as a string.
   //-----------------------------------------------------------------
   public String toString ()
   {
     return title;
   }
}
```



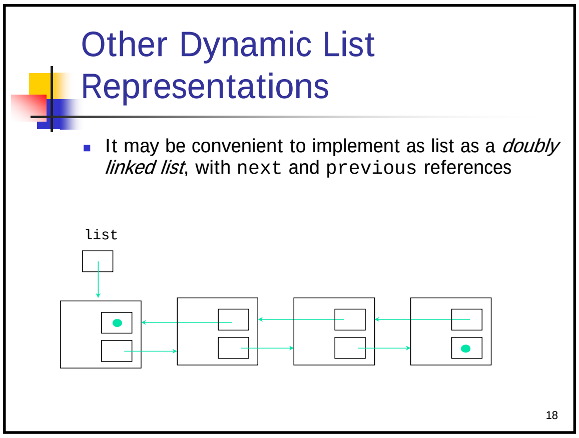
## Magazine Collection

- A method called `insert` could be defined to add a node anywhere in the list, to keep it sorted, for example



## Magazine Collection

- A method called `delete` could be defined to remove a node from the list

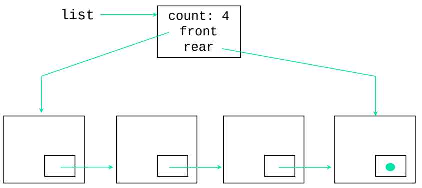


## Other Dynamic List Representations

- It may be convenient to implement as list as a *doubly linked list*, with `next` and `previous` references

## Other Dynamic List Implementations

- It may be convenient to use a separate *header node*, with a count and references to both the front and rear of the list



## Other Dynamic List Implementations

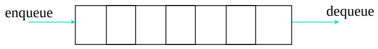
- A linked list can be *circularly linked* in which case the last node in the list points to the first node in the list
- If the linked list is *doubly linked*, the first node in the list also points to the last node in the list
- Choice of linking:
  - The representation should
    - facilitate the intended operations and
    - make them easy to implement

## Other Classic Data Structures

- Classic *linear data structures* include *queues* and *stacks*
- Classic *nonlinear data structures* include *trees*, *binary trees*, *graphs*, and *digraphs*
- *CSI2114 explores Data Structures in much more detail Introduction to abstract data types. Trees, binary search trees, balanced trees. Searching. Sorting. Simple examples of complexity analysis. Graphs, simple graph algorithms: depth-first and breadth-first search, minimum spanning tree, shortest path. (Lab work will be done in the Java programming language). Prerequisite: CSI1101 or CSI1102*

## Linear data structure 2: Queues

- A *queue* is similar to a list but adds items only to the rear of the list and removes them only from the front
- It is called a FIFO data structure: First-In, First-Out
- Analogy:
  - a line of people at a bank teller's window
- Used quite a lot in **Operating Systems**
- Queues often are helpful in simulations or any situation in which items get "backed up" while awaiting processing

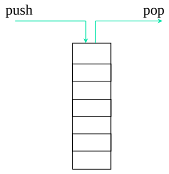


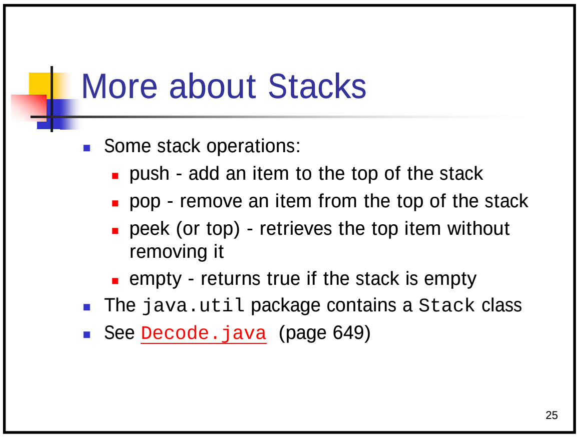
## More about Queues

- We can define the operations for a queue
  - enqueue - add an item to the rear of the queue
  - dequeue (or serve) - remove an item from the front of the queue
  - isEmpty - returns true if the queue is empty
- A queue can be represented by a *singly-linked list*; it is most efficient if the references point from the front toward the rear of the queue

## Linear data structure 2: Stacks

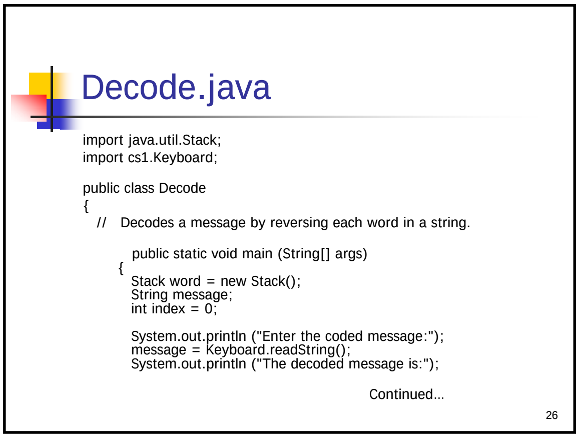
- A *stack* ADT is also linear, like a list or a queue
- Items are added and removed from only one end of a stack
- It is therefore LIFO: Last-In, First-Out
- Analogies:
  - a stack of plates in a cupboard,
  - a stack of bills to be paid,
  - or a stack of hay bales in a barn





## More about Stacks

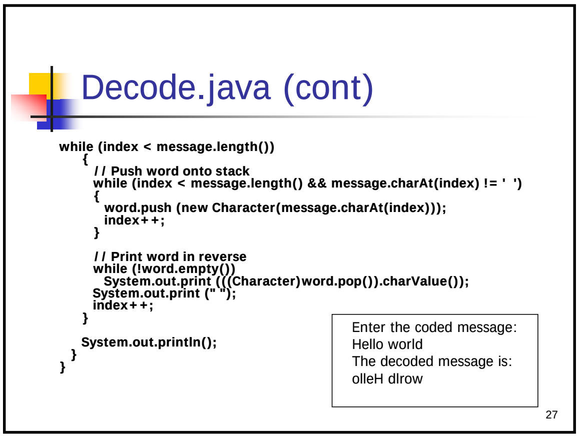
- Some stack operations:
  - push - add an item to the top of the stack
  - pop - remove an item from the top of the stack
  - peek (or top) - retrieves the top item without removing it
  - empty - returns true if the stack is empty
- The `java.util` package contains a `Stack` class
- See `Decode.java` (page 649)



## Decode.java

```
import java.util.Stack;
import cs1.Keyboard;

public class Decode
{
   //   Decodes a message by reversing each word in a string.

         public static void main (String[] args)
        {
         Stack word = new Stack();
         String message;
         int index = 0;

         System.out.println ("Enter the coded message:");
         message = Keyboard.readString();
         System.out.println ("The decoded message is:");
```

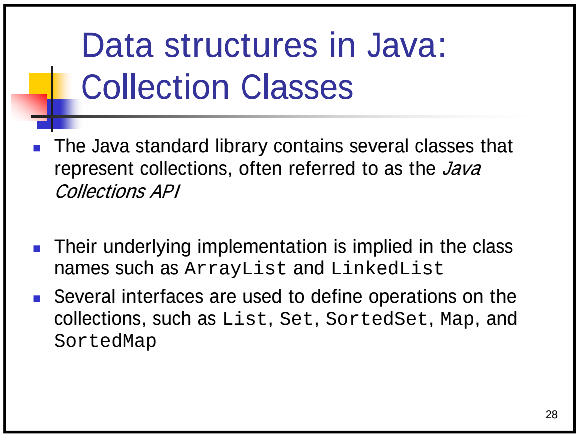
Continued...



## Decode.java (cont)

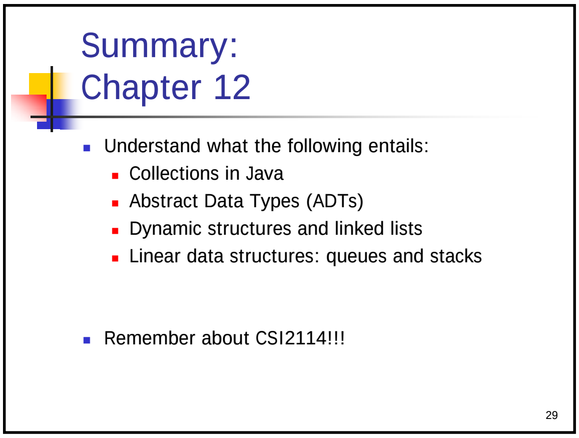
```
while (index < message.length())
    {
      // Push word onto stack
      while (index < message.length() && message.charAt(index) != '  ')
      {
        word.push (new Character(message.charAt(index)));
        index++;
      }

      // Print word in reverse
      while (!word.empty())
        System.out.print (((Character)word.pop()).charValue());
      System.out.print (" ");
      index++;
    }

    System.out.println();
  }
}
```

```
Enter the coded message:
Hello world
The decoded message is:
olleH dlrow
```



## Data structures in Java: Collection Classes

- The Java standard library contains several classes that represent collections, often referred to as the *Java Collections API*
- Their underlying implementation is implied in the class names such as `ArrayList` and `LinkedList`
- Several interfaces are used to define operations on the collections, such as `List`, `Set`, `SortedSet`, `Map`, and `SortedMap`



## Summary: Chapter 12

- Understand what the following entails:
  - Collections in Java
  - Abstract Data Types (ADTs)
  - Dynamic structures and linked lists
  - Linear data structures: queues and stacks
- Remember about CSI2114!!!