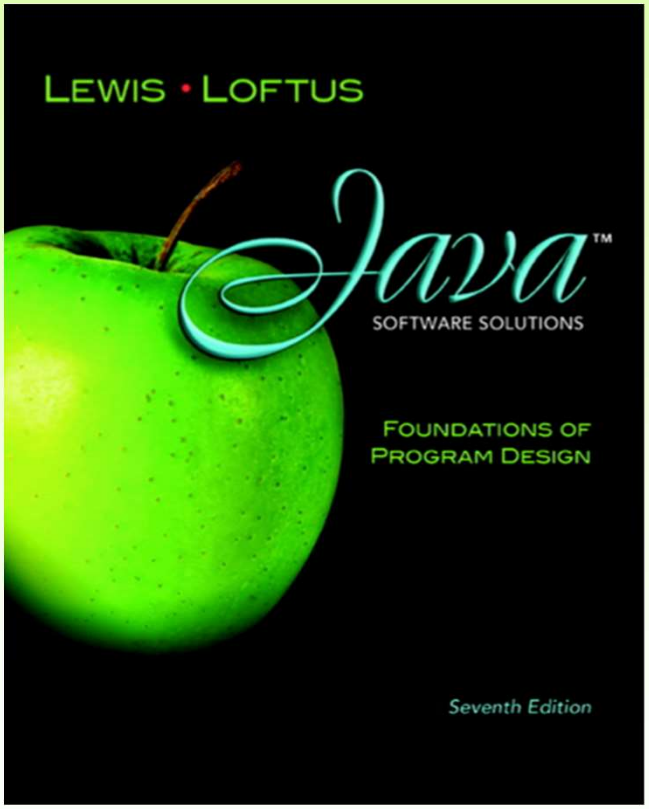
Chapter 13
Collections



Java Software Solutions

Foundations of Program Design

Seventh Edition

John Lewis
William Loftus

Addison-Wesley
is an imprint of
PEARSON

Collections

- A collection is an object that helps us organize and manage other objects

- Chapter 13 focuses on:
  - the concept of a collection
  - separating the interface from the implementation
  - dynamic data structures
  - linked lists
  - queues and stacks
  - trees and graphs
  - generics

Outline

→ Collections and Data Structures

Dynamic Representations

Linear Structures (Queues & Stacks)

Non-Linear Structures (Trees & Graphs)

The Java Collections API

Collections

- A *collection* is an object that serves as a repository for other objects
- A collection provides services for 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*, containing all the same type of objects, and sometimes they are *heterogeneous*

Abstraction

- Collections can be implemented in many different ways
- Collections should be *abstractions*
- That is, they should hide unneeded details
- We want to separate the interface of the structure from its underlying implementation
- This helps manage complexity and makes it possible to change the implementation without changing the interface

Abstract Data Types

- 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
- In one sense, as long as the ADT fulfills the promises of the interface, it doesn't matter how the ADT is implemented
- Objects are a good programming mechanism to create ADTs because their internal details are *encapsulated*

Outline

Collections and Data Structures

Dynamic Representations

Linear Structures (Queues & Stacks)

Non-Linear Structures (Trees & Graphs)

The Java Collections API

Dynamic Structures

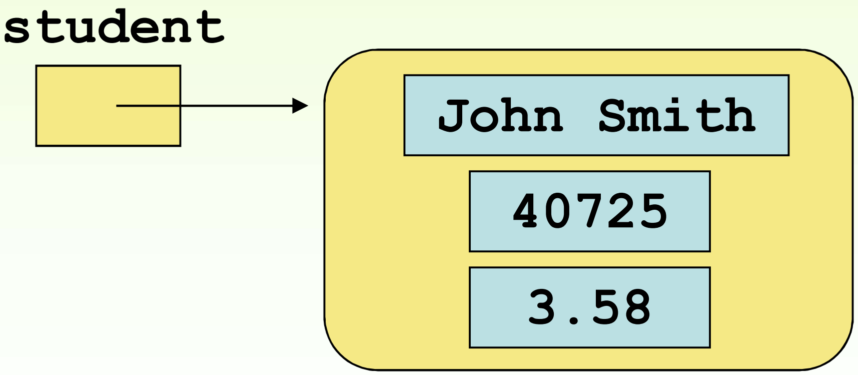
- A *static* data structure has a fixed size
- This meaning is different from the meaning of the `static` modifier
- Arrays are static; once you define the number of elements it can hold, the size doesn’t change
- A *dynamic data structure* grows and shrinks at execution time as required by its contents
- A dynamic data structure is implemented using object references as *links*

Object References

- 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:



References as Links

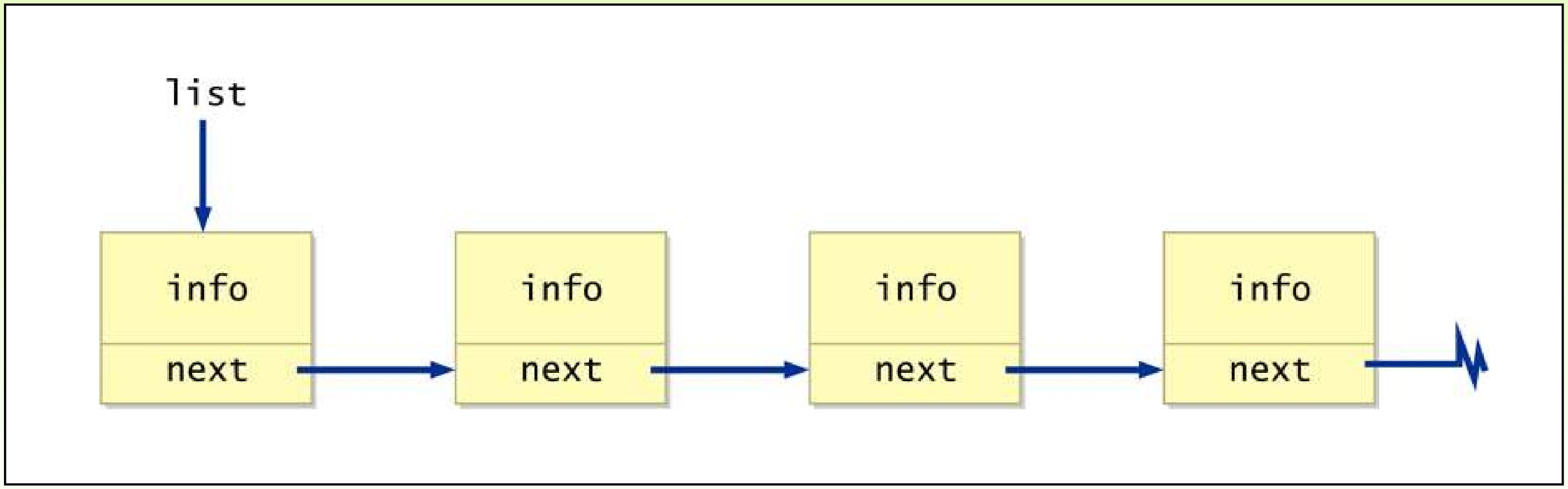
- Object references can be used to create *links* between objects

- Suppose a class contains a reference to another object of the same class:

```
class Node
{
   int info;
   Node next;
}
```

References as Links

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



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, use a separate node class with two parts:
  - a reference to an independent object
  - a link to the next node in the list
- The internal representation becomes a linked list of nodes

Magazine Collection

- Let’s explore an example of a collection of `Magazine` objects, managed by the `MagazineList` class, which has an private inner class called `MagazineNode`

- See `MagazineRack.java`
- See `MagazineList.java`
- See `Magazine.java`

```java
//********************************************************************
//  MagazineRack.java       Author: Lewis/Loftus
//
//  Driver to exercise the MagazineList collection.
//********************************************************************

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);
   }
}
```

```
//***************                                    ****************
//  MagazineRack.
//
//  Driver to exe
//***************                                    ****************

public class Maga
{
   //-----------------------------------------------------------------
   //  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);
   }
}
```

Output

```
Time
Woodworking Today
Communications of the ACM
House and Garden
GQ
```

```java
//********************************************************************
//  MagazineList.java       Author: Lewis/Loftus
//
//  Represents a collection of magazines.
//********************************************************************

public class MagazineList
{
   private MagazineNode list;

   //-----------------------------------------------------------------
   //  Sets up an initially empty list of magazines.
   //-----------------------------------------------------------------
   public MagazineList()
   {
      list = null;
   }
```

continue

continue

```
//----------------------------------------------------------------
//  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;
      while (current.next != null)
         current = current.next;
      current.next = node;
   }
}
```

continue

continue

```
   //----------------------------------------------------------------
   //  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;
   }
```

continue

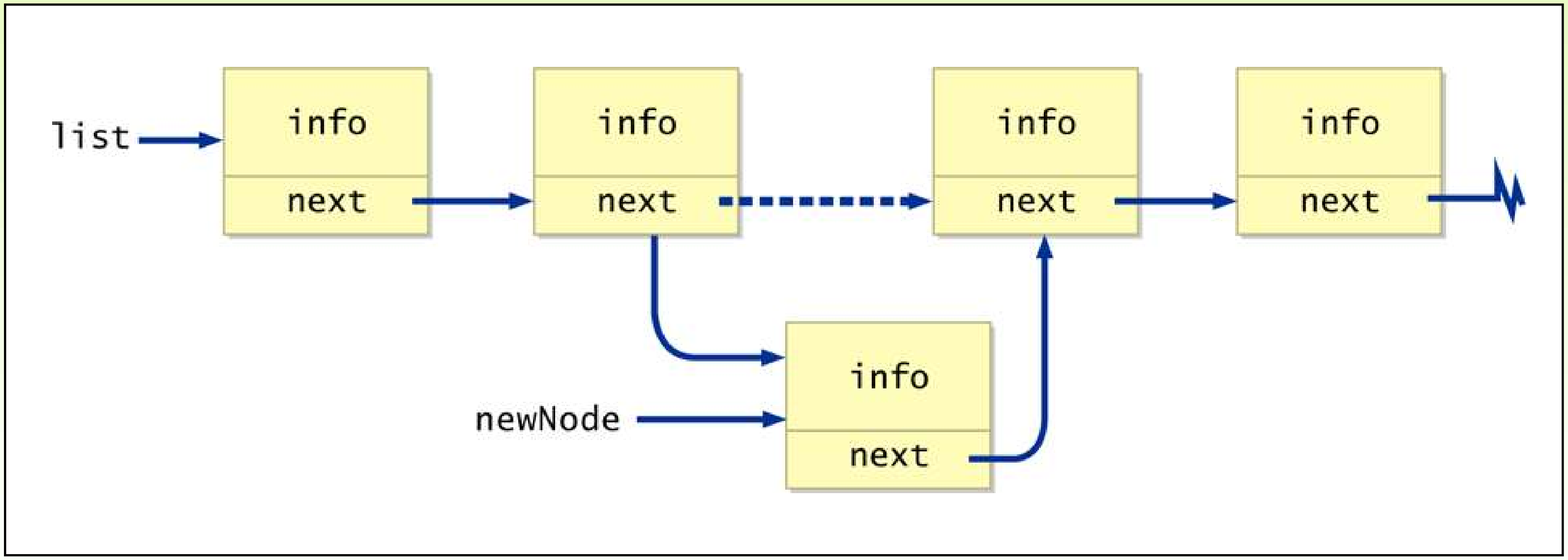
continue

```
   //*****************************************************************
   //  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       Author: Lewis/Loftus
//
//  Represents a single magazine.
//********************************************************************

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;
   }
}
```

Inserting a Node

- A node can be inserted into a linked list with a few reference changes:



Quick Check

Write code that inserts `newNode` after the node pointed to by `current`.

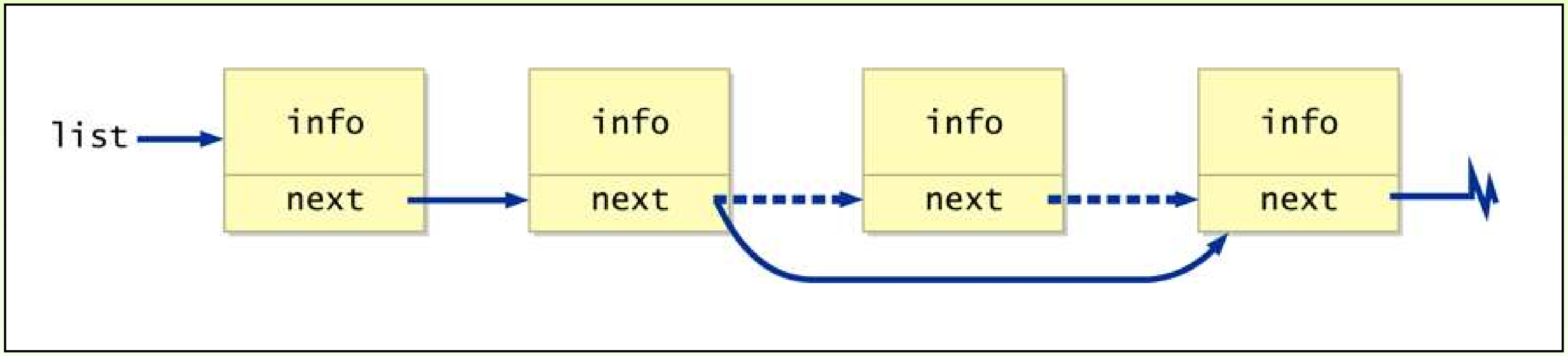
Quick Check

Write code that inserts `newNode` after the node pointed to by `current`.

```
newNode.next = current.next;

current.next = newNode;
```

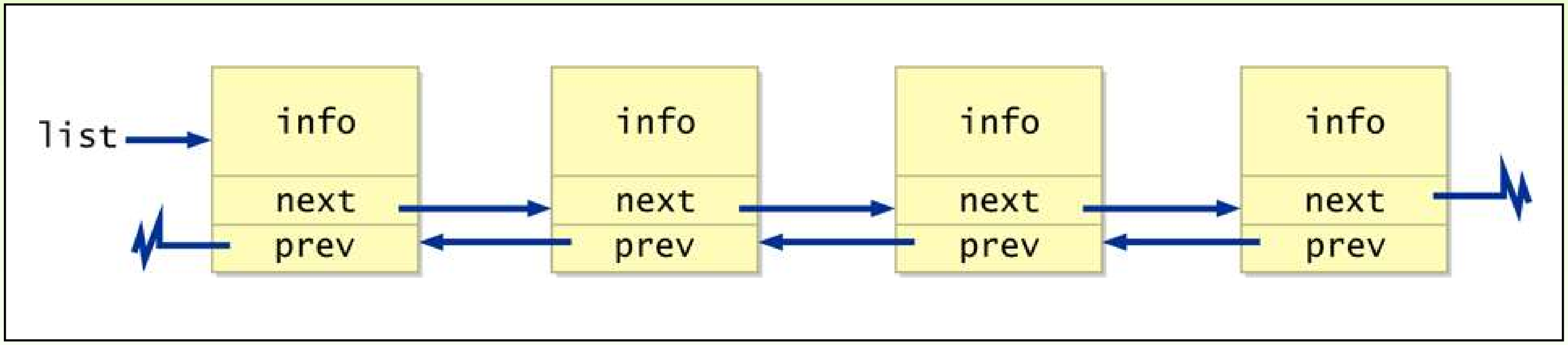
Deleting a Node

- Likewise, a node can be removed from a linked list by changing the `next` pointer of the preceding node:



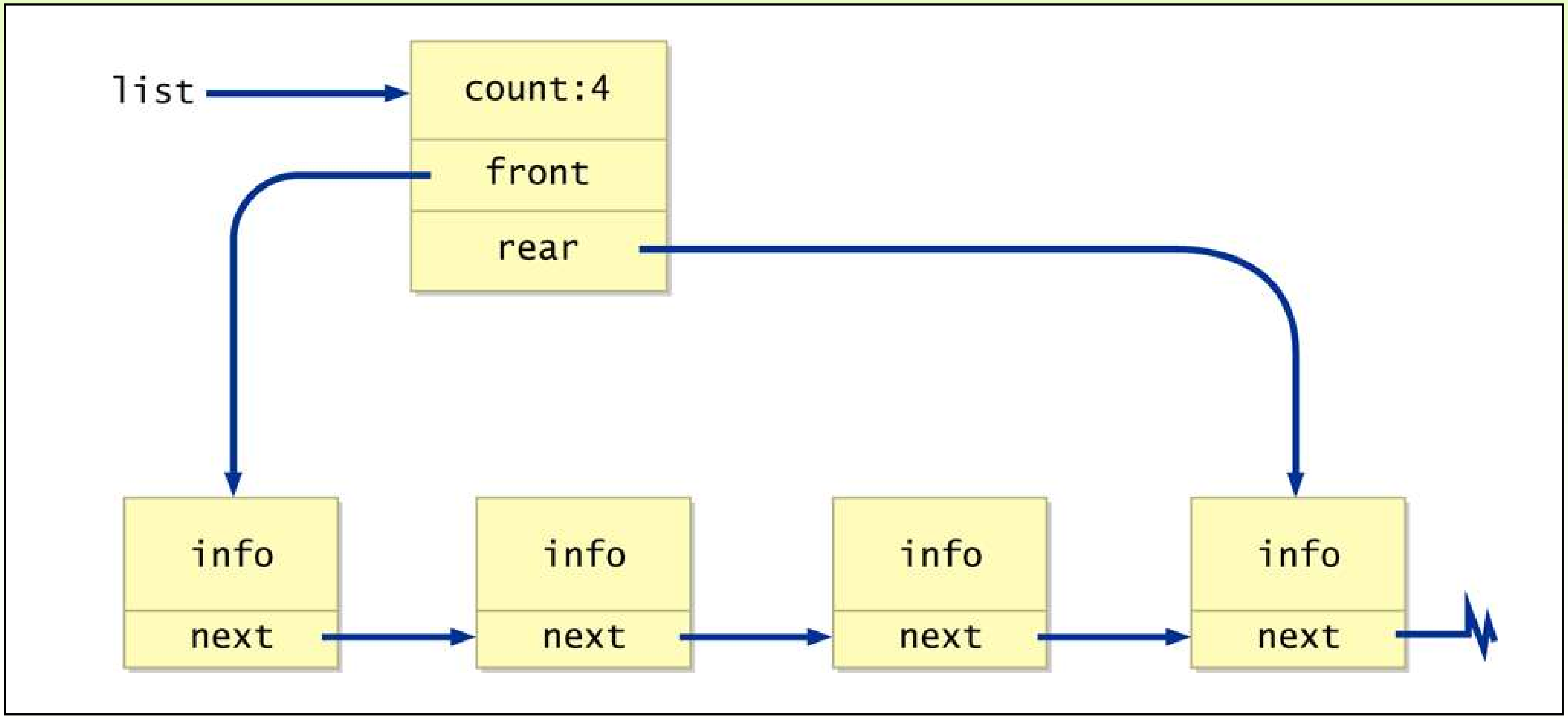
Other Dynamic Representations

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



Other Dynamic Representations

- Another approach is to use a separate *header node*, with a count and references to both the front and rear of the list:



Outline

Collections and Data Structures

Dynamic Representations

➡ Linear Structures (Queues & Stacks)

Non-Linear Structures (Trees & Graphs)

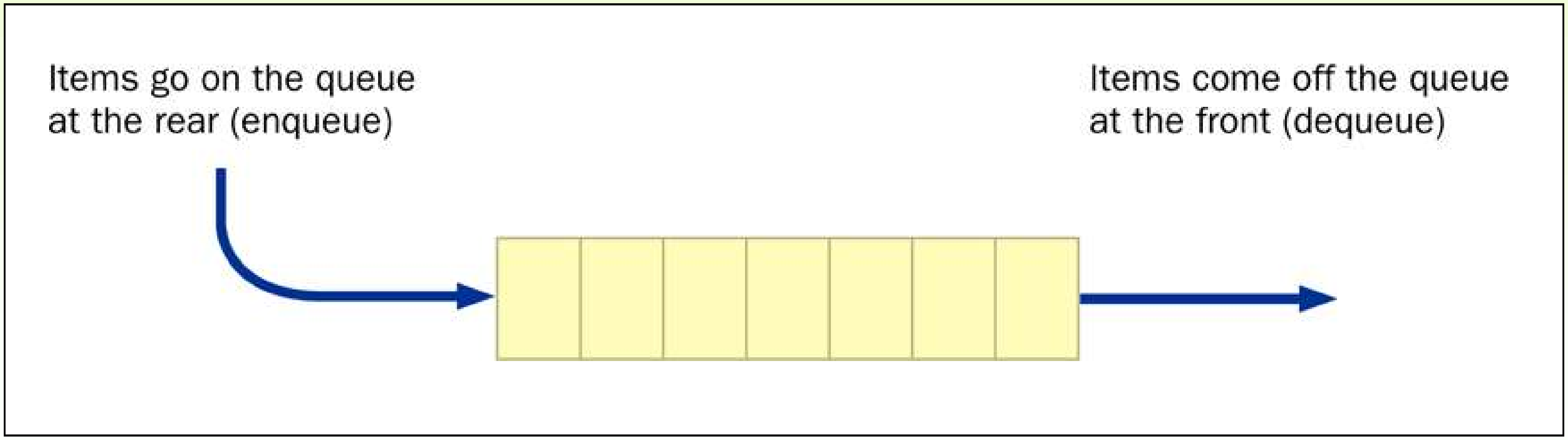
The Java Collections API

Classic Data Structures

- Now we'll examine some common data structures that are helpful in many situations
- Classic *linear data structures* include *queues* and *stacks*
- Classic *nonlinear data structures* include *trees* and *graphs*

Queues

- A *queue* is a list that adds items only to the rear of the list and removes them only from the front
- It is a FIFO data structure: First-In, First-Out
- Analogy: a line of people at a bank teller's window


Items go on the queue at the rear (enqueue)
Items come off the queue at the front (dequeue)

Queues

- Classic 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
  - empty - returns true if the queue is empty
- Queues often are helpful in simulations or any situation in which items get “backed up” while awaiting processing

Queues

- 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

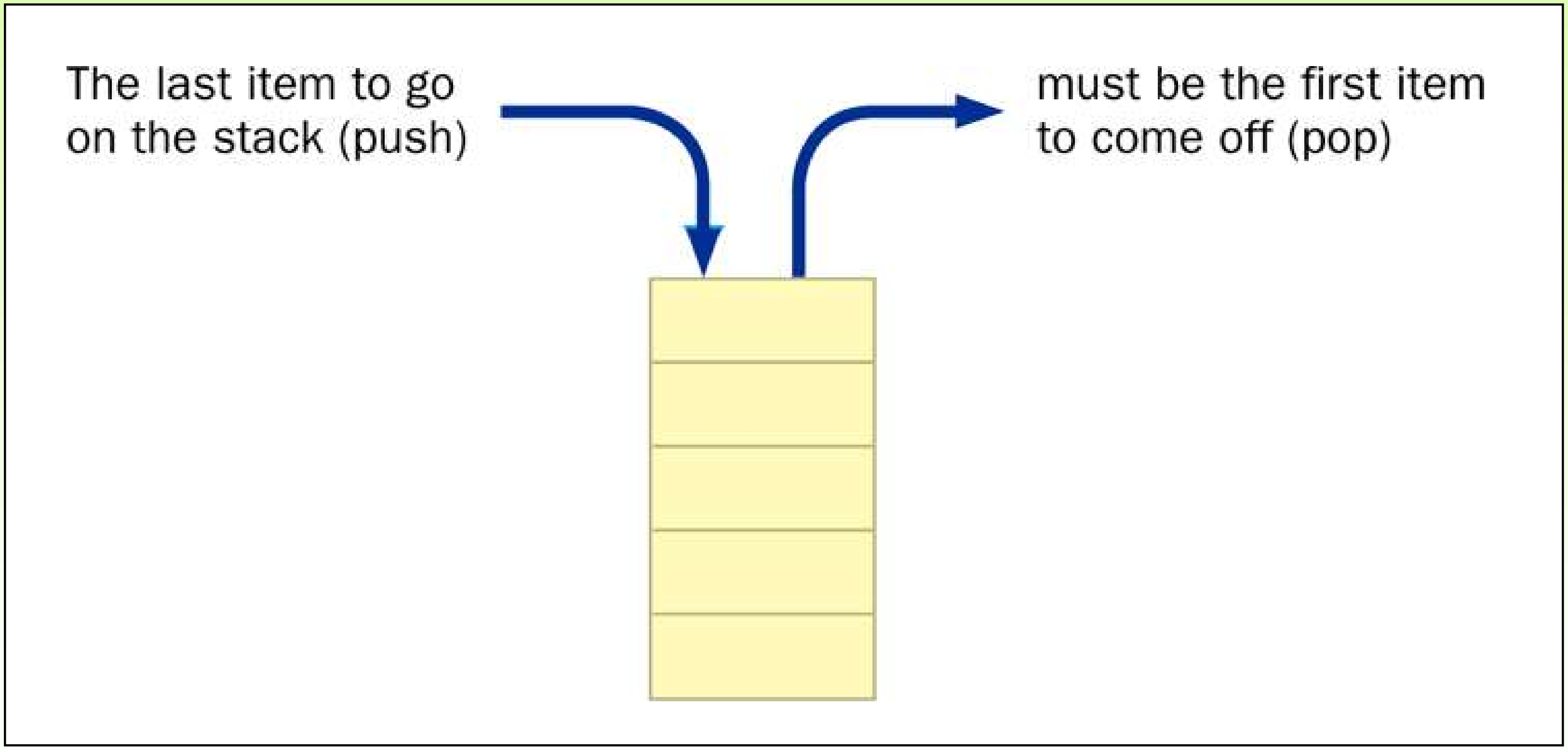
- A queue can be represented by an array, using the remainder operator (%) to “wrap around” when the end of the array is reached and space is available at the front of the array

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 or a stack of books

Stacks

- Stacks often are drawn vertically:



Stacks

- Clasic 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
- A stack can be represented by a singly-linked list, with the first node in the list being to top element on the stack
- A stack can also be represented by an array, with the bottom of the stack at index 0

Stacks

- The `java.util` package contains a `Stack` class
- The `Stack` operations operate on `Object` references
- Suppose a message has been encoded by reversing the letters of each word
- See `Decode.java`

```
//********************************************************************
//  Decode.java       Author: Lewis/Loftus
//
//  Demonstrates the use of the Stack class.
//********************************************************************

import java.util.*;

public class Decode
{
   //-----------------------------------------------------------------
   //  Decodes a message by reversing each word in a string.
   //-----------------------------------------------------------------
   public static void main (String[] args)
   {
      Scanner scan = new Scanner (System.in);

      Stack word = new Stack();

      String message;
      int index = 0;

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

continue

continue

```
      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();
   }
}
```

continue

```
      while (inde
      {
         // Push w
         while (in                                   harAt(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();
   }
}
```

Sample Run

```
Enter the coded message:
artxE eseehc esaelp
The decoded message is:
Extra cheese please
```

Outline

Collections and Data Structures

Dynamic Representations

Linear Structures (Queues & Stacks)

⇨ Non-Linear Structures (Trees & Graphs)

The Java Collections API

Trees

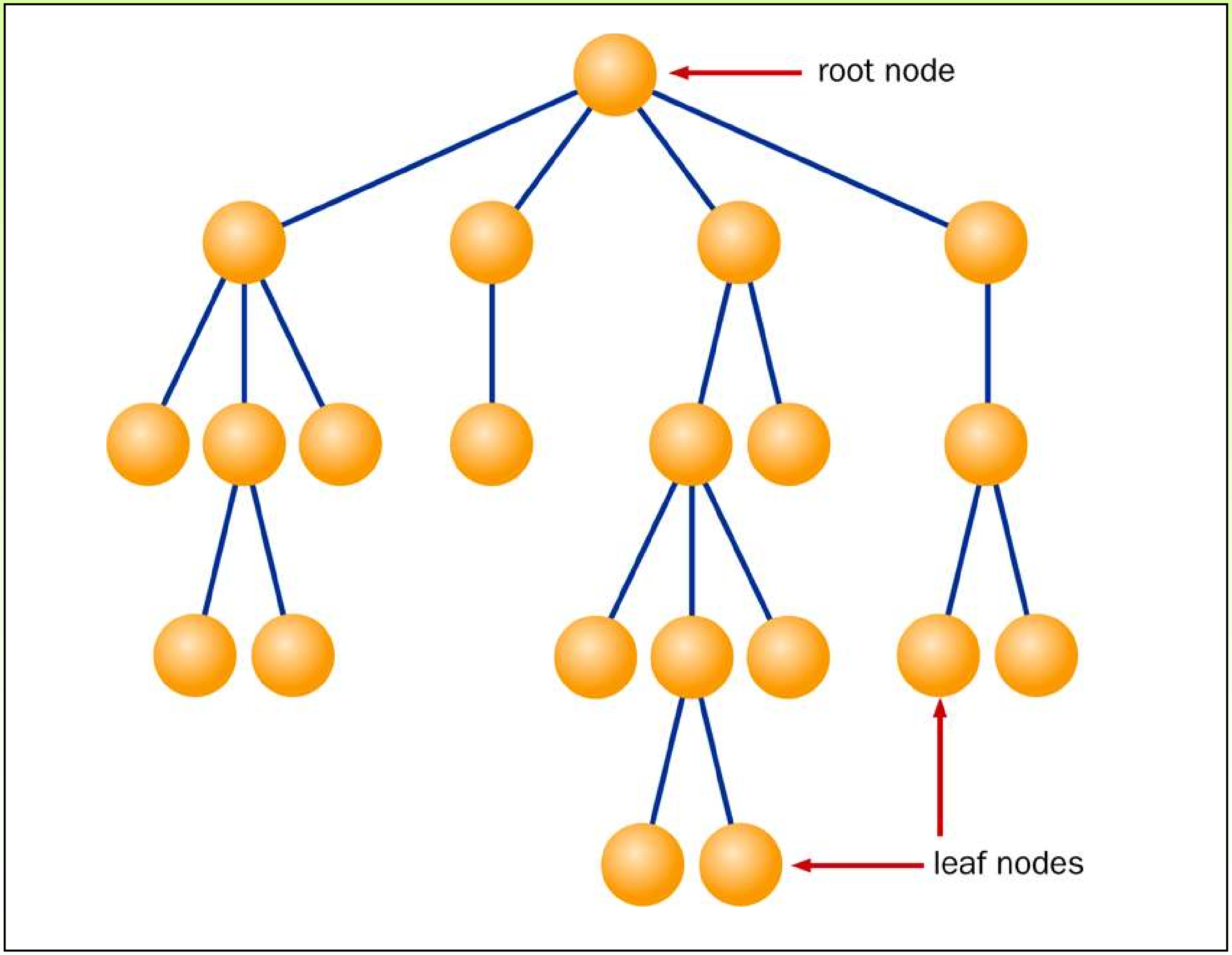
- A *tree* is a non-linear data structure that consists of a *root node* and potentially many levels of additional nodes that form a hierarchy

- Nodes that have no children are called *leaf nodes*

- Nodes except for the root and leaf nodes are called *internal nodes*

- In a general tree, each node can have many child nodes

A General Tree



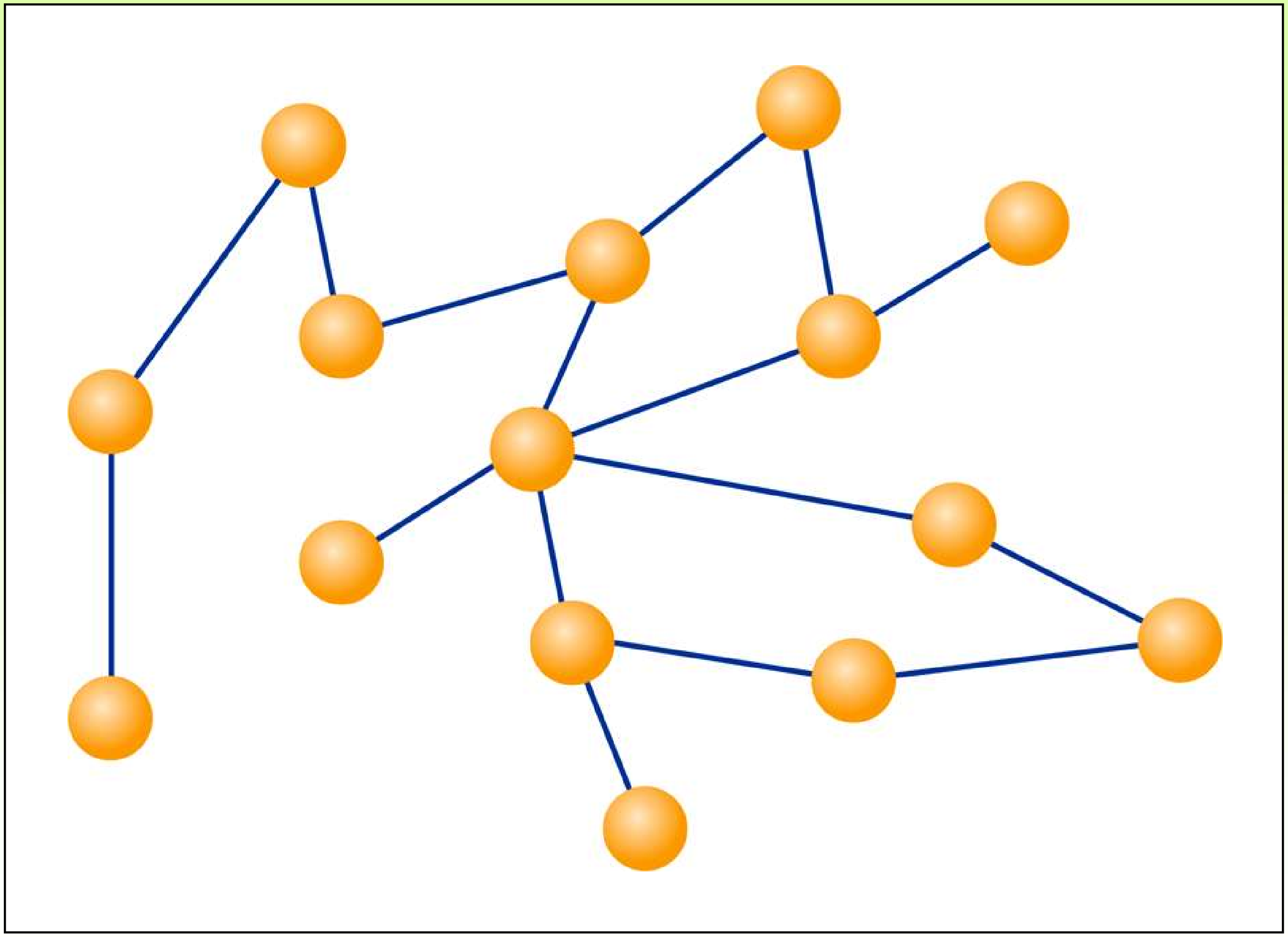
Binary Trees

- In a *binary tree*, each node can have no more than two child nodes
- Trees are typically are represented using references as dynamic links
- For binary trees, this requires storing only two links per node to the left and right child

Graphs

- A *graph* is another non-linear structure
- Unlike a tree, a graph does not have a root
- Any node in a graph can be connected to any other node by an *edge*
- Analogy: the highway system connecting cities on a map

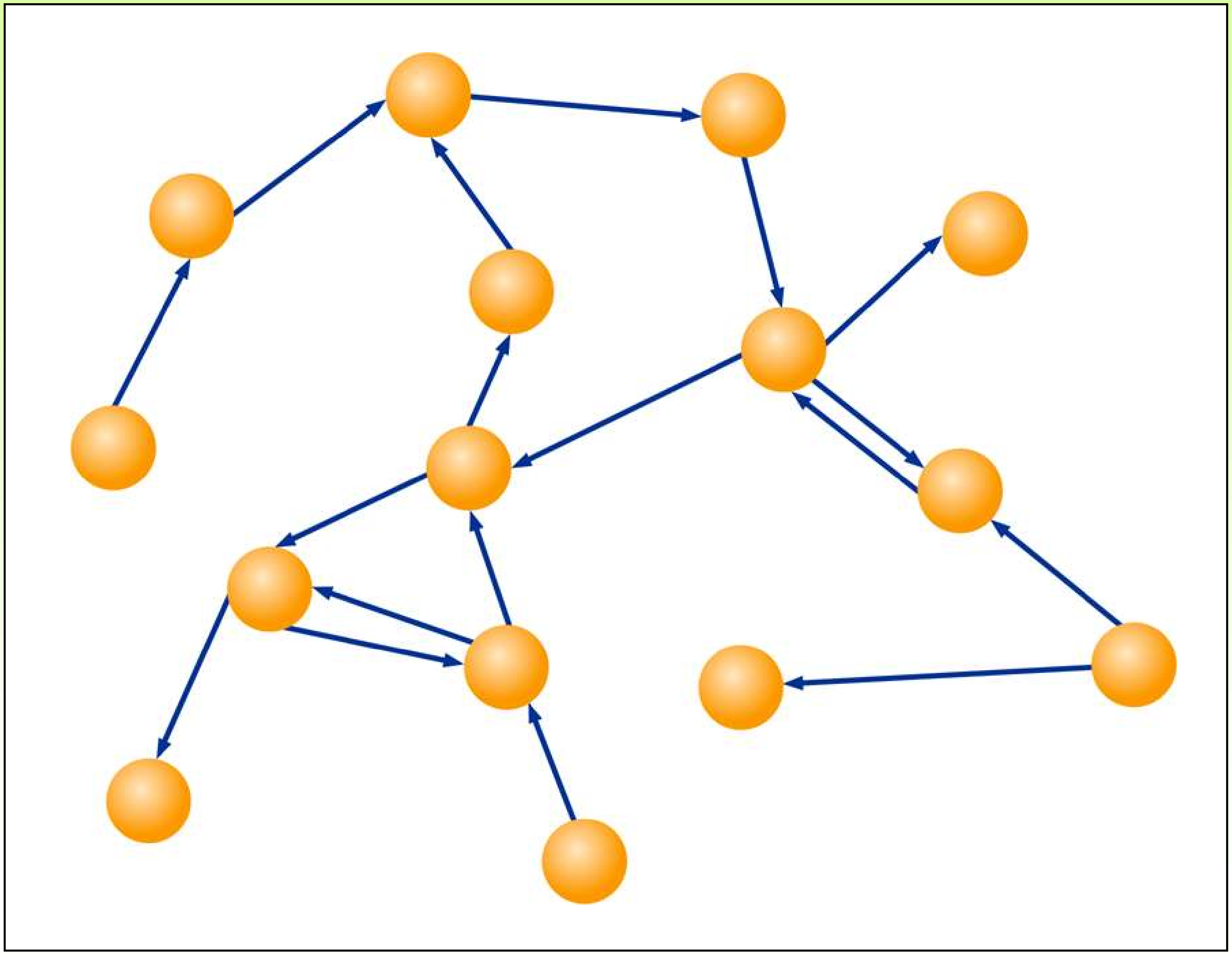
Graphs



Digraphs

- In a *directed graph* or *digraph*, each edge has a specific direction.
- Edges with direction sometimes are called *arcs*
- Analogy: airline flights between airports

Digraphs



Representing Graphs

- Both graphs and digraphs can be represented using dynamic links or using arrays.
- As always, the representation should facilitate the intended operations and make them convenient to implement

Outline

Collections and Data Structures

Dynamic Representations

Linear Structures (Queues & Stacks)

Non-Linear Structures (Trees & Graphs)

➡ The Java Collections API

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`

Generics

- As mentioned in Chapter 5, Java supports *generic types,* which are useful when defining collections

- A class can be defined to operate on a generic data type which is specified when the class is instantiated:

```
LinkedList<Book> myList =
 new LinkedList<Book>();
```

- By specifying the type stored in a collection, only objects of that type can be added to it

- Furthermore, when an object is removed, its type is already established

Summary

- Chapter 13 has focused on:
  - the concept of a collection
  - separating the interface from the implementation
  - dynamic data structures
  - linked lists
  - queues and stacks
  - trees and graphs
  - generics