Files and Collections

CIS*2430 (Fall 2021)

File Input/Output

- A *stream* is an object that enables the flow of data between a program and some I/O device or file.
- Input streams can flow from the keyboard or from a file:
  - **System.in** is an input stream that connects to the keyboard:

    Scanner keyboard = new Scanner(System.in);
- Output streams can flow to a screen or to a file:
  - **System.out** is an output stream that connects to the screen:

    System.out.println("Output stream");

Text Files

- Files that are designed to be read by human beings, and that can be read or written with an editor are called *text files:*
  - Text files are also called ASCII files because the data they contain uses an ASCII encoding scheme.
  - An advantage of text files is that they are usually the same on all computers, and as a result can be moved from one computer to another.

Binary Files

- Files that are designed to be read by programs and that consist of a sequence of binary digits are called *binary files:*
  - Binary files are designed to be read on the same type of computer and with the same programming language as the computer that created the file.
  - An advantage of binary files is that they are *more efficient to process* than text files.
  - Unlike most binary files, Java binary files have the advantage of being platform independent also.

Writing to a Text File

- The class **PrintWriter** is a stream class that can be used to write to a text file.
- All file I/O classes are in the package **java.io** and need to be imported:

```
import java.io.PrintWriter;
import java.io.FileOutputStream;
import java.io.FileNotFoundException;
```

- The class **PrintWriter** has no constructor that takes a file name as its argument:
  - It uses another class, **FileOutputStream**, to convert a file name to an object that can be used as the argument to its constructor.

Writing to a Text File

- The process of connecting a stream to a file is called *opening the file:*

```
PrintWriter outputStream =
    new PrintWriter(new FileOutputStream(FileName));
```

- If the file already exists, then doing this causes the old contents to be lost.
- If the file does not exist, then a new, empty file named ***FileName*** is created.

Writing Output File

```
import java.io.PrintWriter;
import java.io.FileOutputStream;
import java.io.FileNotFoundException;

public class TextFileOutputDemo {
   public static void main(String[] args) {
       PrintWriter outputStream = null;
       try {
          outputStream = new PrintWriter(new FileOutputStream("stuff.txt"));
       } catch (FileNotFoundException e) {
          System.out.println("Error opening the file stuff.txt.");
          System.exit(0);
       }
       outputStream.println("The quick brown fox");
       outputStream.println("jumped over the lazy dog.");
       outputStream.close();
   }
}
```

Buffered Output

- Output streams connected to files are usually *buffered:*
  - Rather than physically writing to the file as soon as possible, the data are saved in a temporary memory location (*buffer*).
  - When enough data accumulates, or when the method **flush** is invoked, the buffered data are written to the file all at once.
  - This is more efficient, since physical writes to a file can be slow.

Closing a File

- The method **close** invokes the method **flush**, thus ensuring that all the data are written to the file:
  - Although Java closes a file automatically when a program ends, it is safer to close it explicitly.
  - If a program relies on Java to close the file, and the program terminates abnormally, then any output that was buffered may not get written to the file.
  - Also, if a program writes to a file and later reopens it to read from the same file, it will have to be closed first anyway.

Pitfall: a `try` Block is a Block

- Since opening a file can result in an exception, it should be placed inside a **try** block.

- If the variable for a **PrintWriter** object needs to be used outside that block, then the variable must be declared outside the block:
  - Otherwise, it would be local to the block, and could not be used elsewhere.
  - If it were declared in the block and referenced elsewhere, the compiler will generate a message indicating that it is an undefined identifier.

Appending to a Text File

- To create a **PrintWriter** object and connect it to a text file for *appending*, a second argument can be used in the constructor for the **FileOutputStream** object:

  ***outputStreamName* = new PrintWriter(new**
  FileOutputStream(*FileName*, true));

  - After this statement, the methods **print**, **println** and/or **printf** can be used to write to the file
  - The new text will be written *after the old text* in the file

Reading from a Text File

- The class **Scanner** can be used for reading from the keyboard as well as reading from a text file:

```
Scanner StreamObject =
    new Scanner(new FileInputStream(FileName));
```

- Methods of the **Scanner** class for reading input behave the same whether reading from the keyboard or reading from a text file:
  - For example, the **nextInt** and **nextLine** methods.

Reading Input File (1/4)

Display 10.3 Reading Input from a Text File Using Scanner

```
import java.util.Scanner;
import java.io.FileInputStream;
import java.io.FileNotFoundException;

public class TextFileScannerDemo
{
    public static void main(String[] args)
    {
        System.out.println("I will read three numbers and a line");
        System.out.println("of text from the file morestuff.txt.");

        Scanner inputStream = null;

        try
        {
            inputStream =
               new Scanner(new FileInputStream("morestuff.txt"));
        }
```

(continued)

Reading Input File (2/4)

Display 10.3 **Reading Input from a Text File Using Scanner**

```
        catch(FileNotFoundException e)
        {
            System.out.println("File morestuff.txt was not found");
            System.out.println("or could not be opened.");
            System.exit(0);
        }
            int n1 = inputStream.nextInt( );
            int n2 = inputStream.nextInt( );
            int n3 = inputStream.nextInt( );

            inputStream.nextLine(); //To go to the next line

            String line = inputStream.nextLine( );

```

(continued)

Reading Input File (3/4)

Display 10.3 Reading Input from a Text File Using Scanner

```
            System.out.println("The three numbers read from the file are:");
            System.out.println(n1 + ", " + n2 + ", and " +  n3);

            System.out.println("The line read from the file is:");
            System.out.println(line);

            inputStream.close( );
        }
    }
```

File morestuff.txt

```
1 2
3 4
Eat my shorts.
```

This file could have been made with a text editor or by another Java program.

(continued)

Reading Input File (4/4)

Display 10.3 **Reading Input from a Text File Using** Scanner

Screen Output

```
I will read three numbers and a line
of text from the file morestuff.txt.
The three numbers read from the file are:
1, 2, and 3
The line read from the file is:
Eat my shorts.
```

Test for End of File

- A program that tries to read beyond the end of a file using methods of the **Scanner** class will cause an exception to be thrown.
- Instead of relying on an exception to signal the end of a file, it's better to use the testing methods such as **hasNextInt** and **hasNextLine** in the Scanner class:
  - These methods check if the next token is a suitable element of the appropriate type before the real input.

Testing with hasNextInt (1/2)

Display 10.5 Checking for the End of a Text File with `hasNextInt`

```
import java.util.Scanner;
import java.io.FileInputStream;
import java.io.FileNotFoundException;

public class HasNextIntDemo
{
    public static void main(String[] args)
    {
        Scanner inputStream = null;

        try
        {
            inputStream =
                new Scanner(new FileInputStream("data.txt"));
        }
        catch(FileNotFoundException e)
        {
            System.out.println("File data.txt was not found");
            System.out.println("or could not be opened.");
            System.exit(0);
        }
```

(continued)

Testing with hasNextInt (2/2)

Display 10.5 **Checking for the End of a Text File with `hasNextInt`**

```
        int next, sum = 0;
        while (inputStream.hasNextInt( ))
        {
            next = inputStream.nextInt( );
            sum = sum + next;
        }

        inputStream.close( );

        System.out.println("The sum of the numbers is " + sum);
    }
}
```

```
File data.txt

1  2
3  4 hi 5
```

*Reading ends when either the end of the file is reach or a token that is not an **`int`** is reached. So, the **`5`** is never read.*

Screen Output

```
The sum of the numbers is 10
```

Path Names

- A *path name* not only gives the name of the file, but also the directory or folder in which the file exists.

- A *full path name* gives a complete path name, starting from the root directory.

- A *relative path name* gives the path to the file, starting with the directory in which the program is located.

Path Names

- The way path names are specified depends on the operating system:
  - A typical UNIX path name that could be used as a file name argument is:

    "/user/joe/data/data.txt“

  - A **Scanner** input stream connected to this file is created as follows:

    Scanner inputStream = new Scanner(new FileInputStream("/user/joe/data/data.txt"));

Path Names

- The Windows operating system specifies path names in a different way:
  - A typical Windows path name is the following:

    C:\dataFiles\goodData\data.txt

  - A **Scanner** input stream connected to this file is created as follows:

```
Scanner inputStream = new Scanner(new
    FileInputStream("C:\\dataFiles\\goodData\\data.txt"));
```

Path Names

- A double backslash (\\) must be used for a Windows path name enclosed in a quoted string:
  - This problem does not occur with path names read in from the keyboard.
- Problems with escape characters can be avoided altogether by always using UNIX conventions when writing a path name:
  - A Java program will accept a path name written in either Windows or Unix format regardless of the operating system on which it is run.

The File Class

- The **File** class is like a wrapper class for files:
  - The constructor for the class **File** takes a name as a string argument and produces an object that represents the file with that name.
  - The **File** object and methods of the class **File** can be used to find information about the file and its properties.

Methods in File Class (1/5)

Display 10.12 **Some Methods in the Class `File`**

`File` is in the `java.io` package.
`public File(String File_Name)`
Constructor. *File_Name* can be either a full or a relative path name (which includes the case of a simple file name). *File_Name* is referred to as the **abstract path name**.
`public boolean exists()`
Tests whether there is a file with the abstract path name.
`public boolean canRead()`
Tests whether the program can read from the file. Returns `true` if the file named by the abstract path name exists and is readable by the program; otherwise returns `false`.

(continued)

Methods in File Class (2/5)

Display 10.12 Some Methods in the Class `File`

```
public boolean setReadOnly()
```

Sets the file represented by the abstract path name to be read only. Returns `true` if successful; otherwise returns `false`.

```
public boolean canWrite()
```

Tests whether the program can write to the file. Returns `true` if the file named by the abstract path name exists and is writable by the program; otherwise returns `false`.

```
public boolean delete()
```

Tries to delete the file or directory named by the abstract path name. A directory must be empty to be removed. Returns `true` if it was able to delete the file or directory. Returns `false` if it was unable to delete the file or directory.

(continued)

Methods in File Class (3/5)

Display 10.12 Some Methods in the Class `File`

```
public boolean createNewFile() throws IOException
```

Creates a new empty file named by the abstract path name, provided that a file of that name does not already exist. Returns `true` if successful, and returns `false` otherwise.

```
public String getName()
```

Returns the last name in the abstract path name (that is, the simple file name). Returns the empty string if the abstract path name is the empty string.

```
public String getPath()
```

Returns the abstract path name as a `String` value.

`public boolean renameTo(File` *New_Name*`)`

Renames the file represented by the abstract path name to *New_Name*. Returns `true` if successful; otherwise returns `false`. *New_Name* can be a relative or absolute path name. This may require moving the file. Whether or not the file can be moved is system dependent.

(continued)

Methods in File Class (4/5)

Display 10.12 **Some Methods in the Class `File`**

```
public boolean isFile()
```

Returns `true` if a file exists that is named by the abstract path name and the file is a normal file; otherwise returns `false`. The meaning of *normal* is system dependent. Any file created by a Java program is guaranteed to be normal.

```
public boolean isDirectory()
```

Returns `true` if a directory (folder) exists that is named by the abstract path name; otherwise returns `false`.

(continued)

Methods in File Class (5/5)

Display 10.12 Some Methods in the Class `File`

`public boolean mkdir()`

Makes a directory named by the abstract path name. Will not create parent directories. See `mkdirs`. Returns `true` if successful; otherwise returns `false`.

`public boolean mkdirs()`

Makes a directory named by the abstract path name. Will create any necessary but nonexistent parent directories. Returns `true` if successful; otherwise returns `false`. Note that if it fails, then some of the parent directories may have been created.

`public long length()`

Returns the length in bytes of the file named by the abstract path name. If the file does not exist or the abstract path name names a directory, then the value returned is not specified and may be anything.

Java Collections

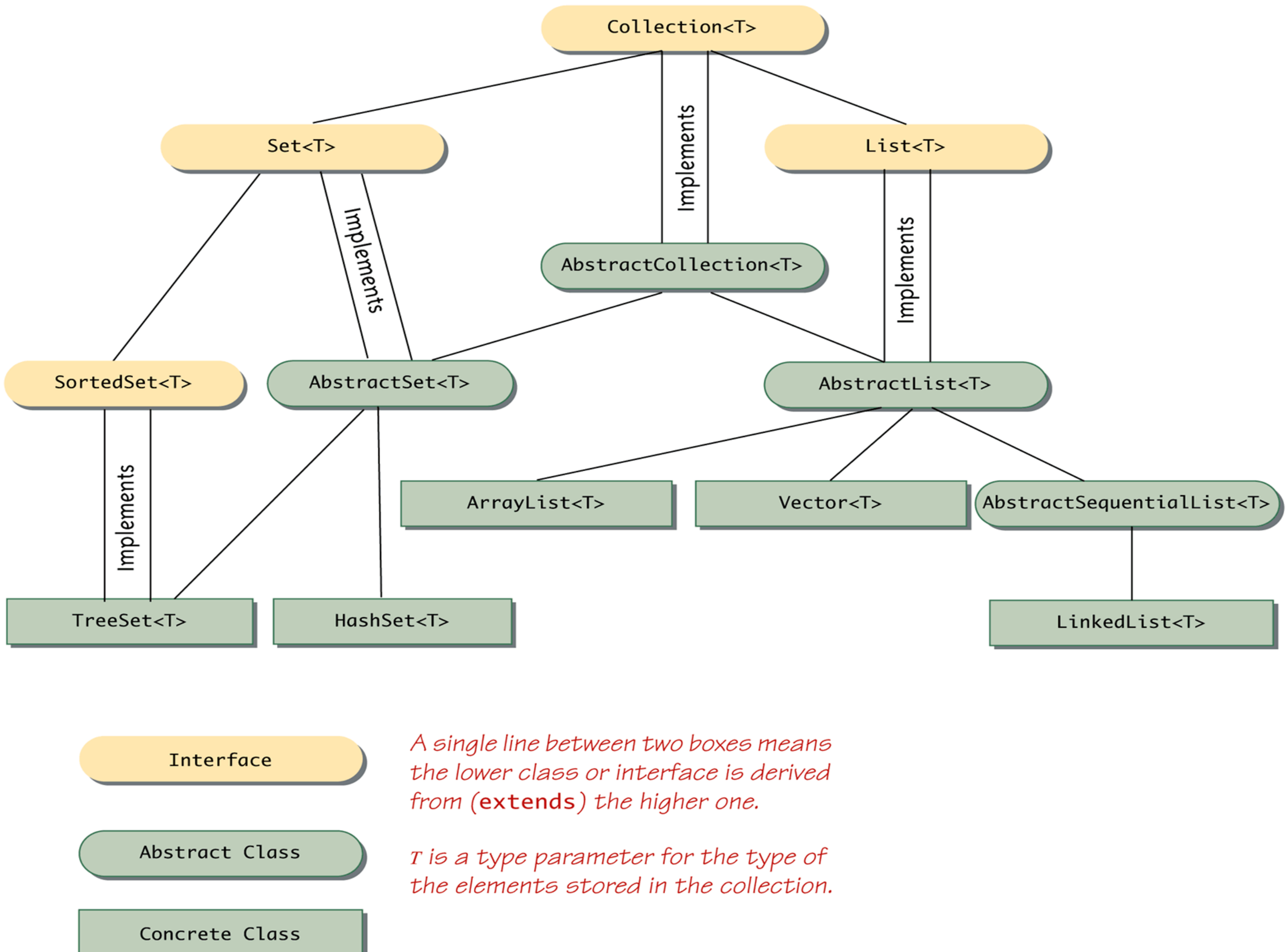
- A Java collection is any class that holds objects and implements the **Collection** interface:
  - Example: the **ArrayList<T>** class.

- A collection can be accessed by an iterator, which is an object that cycles through all the elements in the collection.

- All the collection classes can be found in package java.util.

The Collection Landscape

Display 16.1 The Collection Landscape



Collection Relationships

- Classes that implement the **List<T>** interface have their elements ordered as on a list:
  - Elements are indexed starting from zero.
  - A class that implements the **List<T>** interface allows elements to occur more than once.
  - The **List<T>** interface has more methods than the **Collection<T>** interface.
  - The **ArrayList<T>** class implements the **List<T>** interface.

Iterators

- An iterator is an object used with a collection to provide sequential access to the collection elements:
  - This access allows examination and possible modification of the elements.
- An iterator imposes an ordering on the elements of a collection even if the collection itself does not impose any order on its elements:
  - If the collection does impose an ordering on its elements, then the iterator will use the same ordering.

Iterator<T> Interface

- Java provides an **Iterator<T>** interface:
  - Any object of any class that satisfies the **Iterator<T>** interface is an **Iterator<T>**.
- An **Iterator<T>** does not stand on its own:
  - It must be associated with some collection object using the method **iterator.**
  - If **c** is an instance of a collection class (e.g., **HashSet<String>**), the following obtains an iterator for **c**:

```
Iterator<String> iteratorForC = c.iterator();
```

Iterator Example (1/3)

Display 16.8 An Iterator

```
import java.util.HashSet;
import java.util.Iterator;

public class HashSetIteratorDemo
{
    public static void main(String[] args)
    {
        HashSet<String> s = new HashSet<String>();

        s.add("health");
        s.add("love");
        s.add("money");

        System.out.println("The set contains:");
```

(continued)

Iterator Example (2/3)

Display 16.8 **An Iterator**

```
        Iterator<String> i = s.iterator();
        while (i.hasNext())
            System.out.println(i.next());

        i.remove();

        System.out.println();
        System.out.println("The set now contains:");

        i = s.iterator();
        while (i.hasNext())
            System.out.println(i.next());

        System.out.println("End of program.");
    }
}
```

You cannot "reset" an iterator "to the beginning." To do a second iteration, you create another iterator.

(continued)

Iterator Example (3/3)

Display 16.8 An Iterator

SAMPLE DIALOGUE

```
The set contains:
money
love
health

The set now contains:
money
love
End of program.
```

The `HashSet<T>` object does not order the elements it contains, but the iterator imposes an order on the elements.

For-Each Loops as Iterators (1/2)

Display 16.9 For-Each Loops as Iterators

```
import java.util.HashSet;
import java.util.Iterator;

public class ForEachDemo
{
    public static void main(String[] args)
    {
        HashSet<String> s = new HashSet<String>();

        s.add("health");
        s.add("love");
        s.add("money");

        System.out.println("The set contains:");
```

(continued)

For-Each Loops as Iterators (2/2)

Display 16.9 For-Each Loops as Iterators

```
        String last = null;
        for (String e : s)
        {
            last = e;
            System.out.println(e);
        }

        s.remove(last);

        System.out.println();
        System.out.println("The set now contains:");

        for (String e : s)
            System.out.println(e);

        System.out.println("End of program.");
    }
}
```

The output is the same as in Display 16.8.

Iterators for ArrayLists (1/4)

Display 16.12 An Iterator Returns a Reference

The class **Date** *is defined in Display 4.13, but you can easily guess all you need to know about* **Date** *for this example.*

```
import java.util.ArrayList;
import java.util.Iterator;

public class IteratorReferenceDemo
{
    public static void main(String[] args)
    {
        ArrayList<Date> birthdays = new ArrayList<Date>();


        birthdays.add(new Date(1, 1, 1990));
        birthdays.add(new Date(2, 2, 1990));
        birthdays.add(new Date(3, 3, 1990));


        System.out.println("The list contains:");
```

(continued)

Iterators for ArrayLists (2/4)

Display 16.12 **An Iterator Returns a Reference**

```
Iterator<Date> i = birthdays.iterator();
while (i.hasNext())
    System.out.println(i.next());

i = birthdays.iterator();
Date d = null; //To keep the compiler happy.
System.out.println("Changing the references.");
while (i.hasNext())
{
    d = i.next();
    d.setDate(4, 1, 1990);
}
```

(continued)

Iterators for ArrayLists (3/4)

Display 16.12 An Iterator Returns a Reference

```
        System.out.println("The list now contains:");

        i = birthdays.iterator();
        while (i.hasNext())
            System.out.println(i.next());

        System.out.println("April fool!");
    }
}
```

(continued)

Iterators for ArrayLists (4/4)

Display 16.12 **An Iterator Returns a Reference**

SAMPLE DIALOGUE

```
The list contains:
January 1, 1990
February 2, 1990
March 3, 1990
Changing the references.
The list now contains:
April 1, 1990
April 1, 1990
April 1, 1990
April fool!
```

Maps

- The Java *map* framework deals with collections of ordered pairs:
  - Ordered pair: a key and an associated value.
- Objects in the map framework can implement mathematical functions and relations, so can be used to construct database classes.
- The map framework uses the **Map<K,V>** interface and the two concrete classes are **TreeMap<K,V>** and **HashMap<K,V>**.

HashMap Example (1/3)

```
import java.util.HashMap;
import java.util.Scanner;
import java.util.Iterator;

public class HashMapDemo {
  public static void main(String[] args) {
     HashMap<String, Employee> employees =
          new HashMap<String, Employee>(10);

     employees.put(“Joe”,
          new Employee(“Joe”, new Date(“September”, 15, 1970)));
     employees.put(“Andy”,
          new Employee(“Andy”, new Date(“August”, 22, 1971)));
     employees.put(“Greg”,
          new Employee(“Greg”, new Date(“March”, 9, 1972)));
     employees.put(“Kiki”,
          new Employee(“Kiki”, new Date(“October”, 8, 1970)));
     System.out.println(“Added Joe, Andy, Greg, and Kiki to the map.”);
```

HashMap Example (2/3)

```
        Scanner keyboard = new Scanner(System.in);
        String name = “”;
        do {
            System.out.print(“\nEnter a name to look up the map. ”);
            System.out.println(“Press enter to quit.”);
            name = keyboard.nextLine();
            if (employees.containsKey(name)) {
              Employee e = employees.get(name);
              System.out.println(“Name found: “ + e.toString());
            } else if (!name.isEmpty()) {
              System.out.println(“Name not found”);
            }
        } while (!name.isEmpty());
        System.out.println(“Created HashMap: ” + employees);
        Iterator<HashMap.Entry<String, Employee>> iter = employees.entrySet().iterator();
        while (iter.hasNext()) {
            HashMap.Entry<String, Employee> entry = iter.next();
            System.out.println(entry.getKey() + “: ” + entry.getValue().toString());
        }
    }
}
```

HashMap Example (3/3)

Sample Dialogue:

```
Added Joe, Andy, Greg, and Kiki to the map.
Enter a name to look up in the map.  Press enter to quit.
Joe
Name found: Joe September 15, 1970
Enter a name to look up in the map.  Press enter to quit.
Frank
Name not found
Enter a name to look up in the map.  Press enter to quit.

Created HashMap: {Joe=Joe September 15, 1970, Greg=Greg March 9,
1972, Andy=Andy August 22, 1971, Kiki=Kiki October 8, 1970}
Joe : Joe September 15, 1970
Greg : Greg March 9, 1972
Andy : Andy August 22, 1971
Kiki : Kiki October 8, 1970
```

Indices and Search

- Given the name "toronto bank", we can extract two keywords "toronto" and "bank ".
- If "toronto" appears in investments 0, 5, 7, 12, and 15 of the ArrayList, we can add the following entry to a HashMap index:

  key: "toronto"

  value: a list of [0, 5, 7, 12, 15]
- If we also know that "bank" appears in investments [0, 3, 7, 10], then the search for "toronto bank" will be the intersection of [0, 5, 7, 12, 15] and [0, 3, 7, 10], which returns investments [0, 7] as the result.

Creating a HashMap Index

- Suitable structure:

```
HashMap<String, ArrayList<Integer>> index =
    new HashMap<String, ArrayList<Integer>>();
```

- Given the list of investments in an ArrayList, the index will look like:

0	toronto bank
1	toronto star
2	royal bank
	…

Changed to

<toronto, [0]> → <toronto, [0, 1]>
<bank, [0]> → <bank, [0, 2]>
<star, [1]>
<royal, [2]>
…

Maintaining the Index

- Adding a new product "royal dutch shell" at location 8:

  <toronto, [0, 1]>
  <bank, [0, 2]>
  <star, [1]>
  <royal, [2]> ➔ <royal, [2, 8]>
  <dutch, [8]>
  <shell, [8]>
  …

- Deleting the product "toronto star" at location 1:

  <toronto, [0, 1]> ➔ <java, [0]>
  <bank, [0, 2]> ➔ <bank, [0, 1]>
  <~~star, [1~~]> deleted
  <royal, [2, 8]> ➔ <royal, [1, 7]>
  <dutch, [8]> ➔ <dutch, [7]>
  <shell, [8]> -> <shell, [7]
  …

0	toronto bank
~~1~~	~~toronto star~~
1	royal bank
	…