R2.01 : Object-oriented development (OOD)

Coordinator : Isabelle Blasquez

My name: Cristina Onete

cristina.onete@gmail.com

Slides : https://www.onete.net/teaching.html

Collections

Beyond arrays

- Seen so far:
  - homogeneous arrays: variables instantiated as the same type
  - heterogeneous arrays: we use polymorphism

- Arrays can be very useful!
- ... however, they also present some disadvantages:

  It is compulsory to declare the length of an array

  Massively overestimating the length is costly

  Once defined, the length cannot be changed

  To add/remove elements we must indicate their index

Let's see an example!

- Take a group of users whose exact number is not known
  - ... and for which the number of users is hard to estimate
  - Say the number of users of a given social network

- We want to be able to:
  - **Store** in a single object all the objects representing users
  - **Add** and **remove** users at will
  - **Dynamically modify** the **size** of the container

- Idea: We could design a class NetUser which models users

Using an array of NetUsers

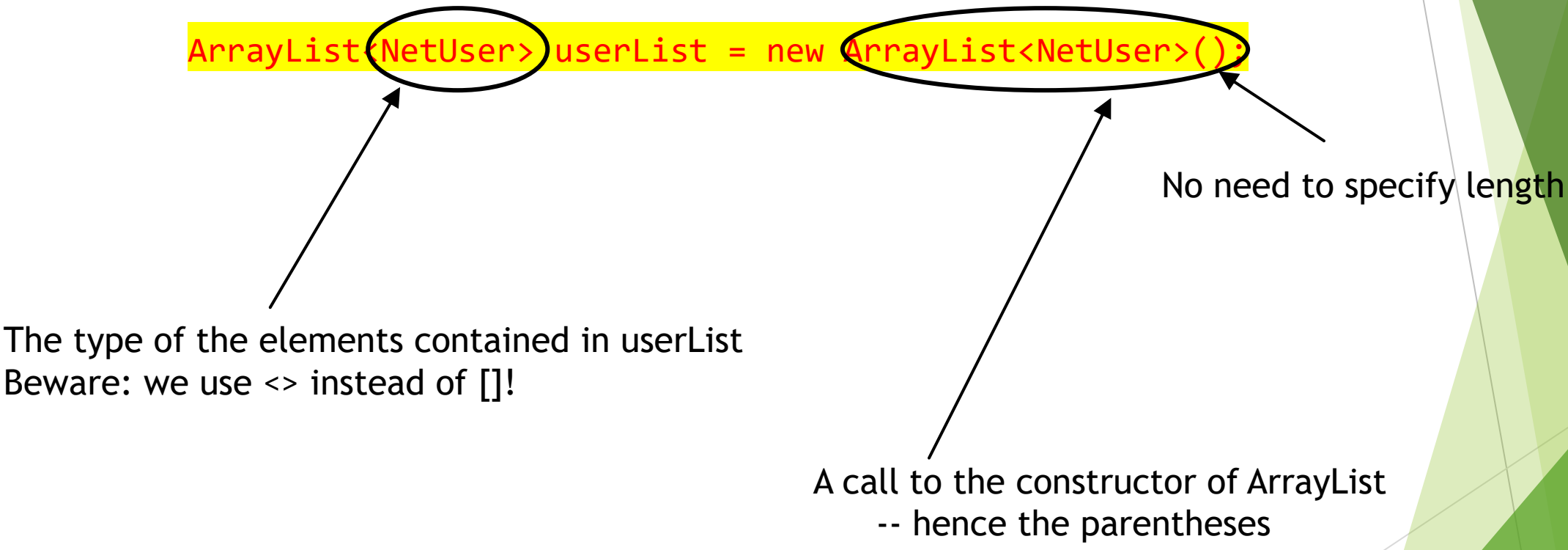
- First question: what would be the length of the array ?
  - Say 7 billion ? (all the users in the world)
  - But: even Facebook only has about 2.1 billion users...
  - That being said, the number of people using social networks keeps increasing every day

- Adding a user in the array:
  - Find the first open position and add the user there
- Changing the length of the array :
  - Declare and instantiate a new array of the updated length
  - Copy the non-null elements of the original array in the new one

A better alternative: ArrayList

- An ArrayList is a collection of items of a single type:

```
ArrayList<NetUser> userList = new ArrayList<NetUser>();
```



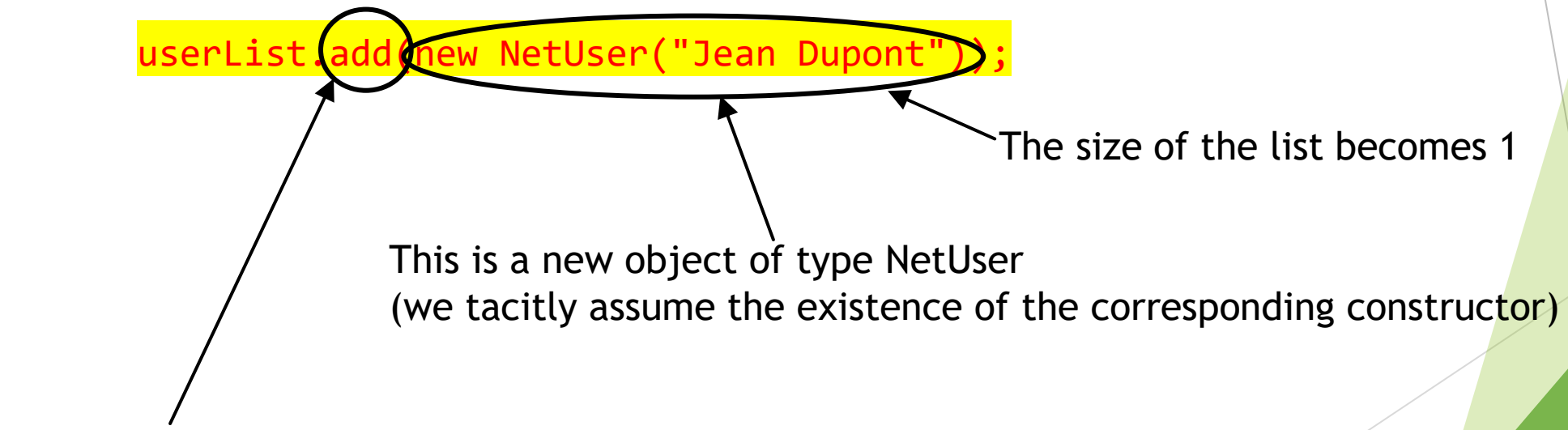
A better alternative: ArrayList

- An ArrayList is a collection of items of a single type:

```
ArrayList<NetUser> userList = new ArrayList<NetUser>();
```

- Adding an element (a user):

```
userList.add(new NetUser("Jean Dupont"));
```



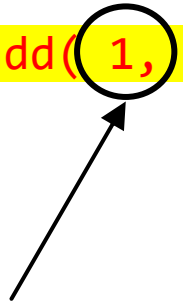
A better alternative: ArrayList

- An ArrayList is a collection of items of a single type:

```
ArrayList<NetUser> userList = new ArrayList<NetUser>();
```

- Adding an element (a user):

```
userList.add(new NetUser("Jean Dupont"));
userList.add( 1, new NetUser("Jean Dupont"));
```



The methode add(index, Element) adds a new element at the given index
All other elements are shifted to the right
The size of the ArrayList increases by 1

A better alternative: ArrayList

- An ArrayList is a collection of items of a single type:

```
ArrayList<NetUser> userList = new ArrayList<NetUser>();
```

- Adding an element (a user):

```
userList.add(new NetUser("Jean Dupont"));
userList.add(1, new NetUser("Jean Dupont"));
```

- Removing an element (a user):

```
userList.remove(user);
```

A previously-instantiated object of type NetUser

A better alternative: ArrayList

- An ArrayList is a collection of items of a single type:

```
ArrayList<NetUser> userList = new ArrayList<NetUser>();
```

- Adding an element (a user):

```
userList.add(new NetUser("Jean Dupont"));
userList.add(1, new NetUser("Jean Dupont"));
```

- Removing an element (a user):

```
userList.remove(user);
```

- We can also use preprogrammed methods to find elements, clone the list, return an element situated at a given position...

Collections in general

- An ArrayList is just one example of a *Collection*
- *Java.util.Collection* is an interface in Java
  - ...which allows us to "collect" elements within a single structure
  - The elements of a collection MUST be objects
    - We cannot collect variables of primitive types
- Like any other interface, `Collection` only contains abstract methods
  - These are specified by the classes implementing the interface

The Java collection framework

Collections

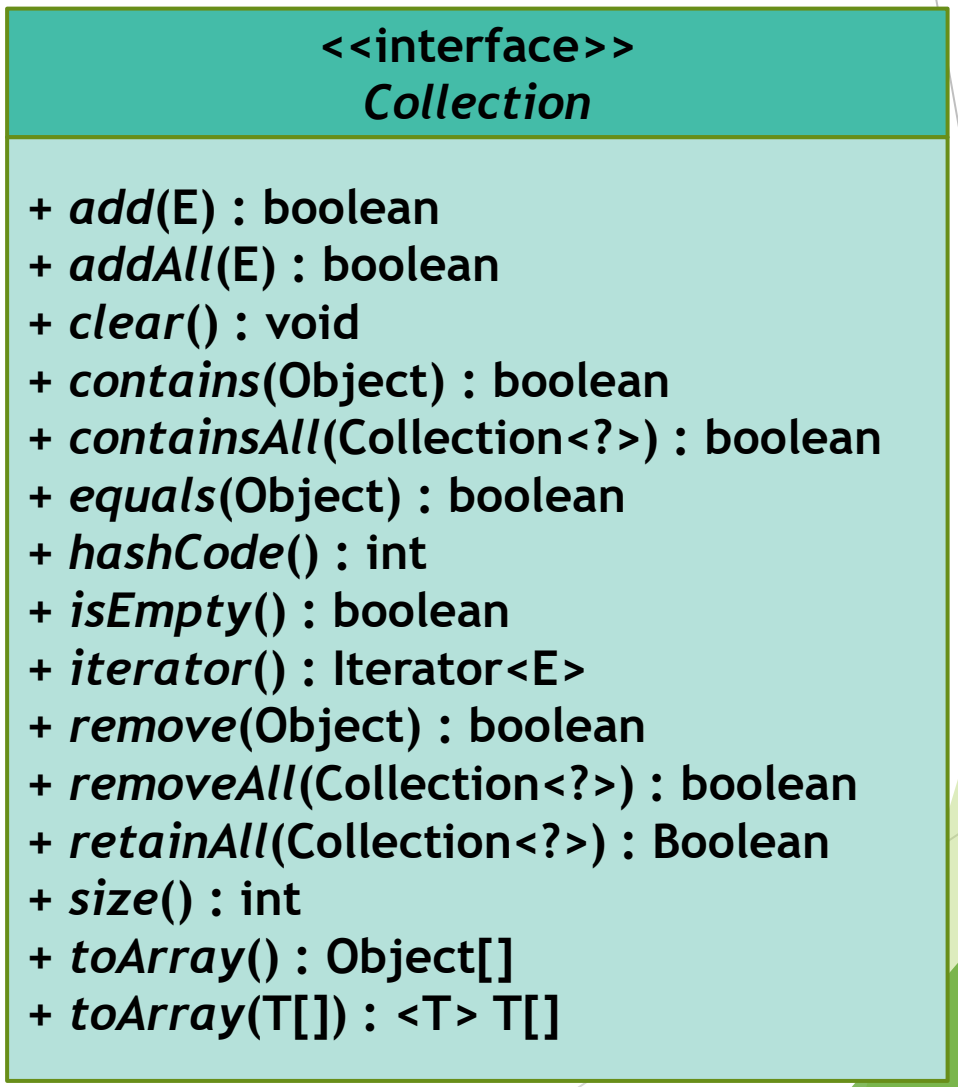
- The Java Collection Framework is a programming framework
  - It contains a hierarchy of interfaces for manipulating collections
  - The abstract methods in the interfaces are implemented concretely for different types of collections

- Why use collections:
  - Collections have variable sizes
  - Data structures and methods already provided in Java
  - A more efficient manipulation of the data
  - Interoperability

The generic structure of Collection

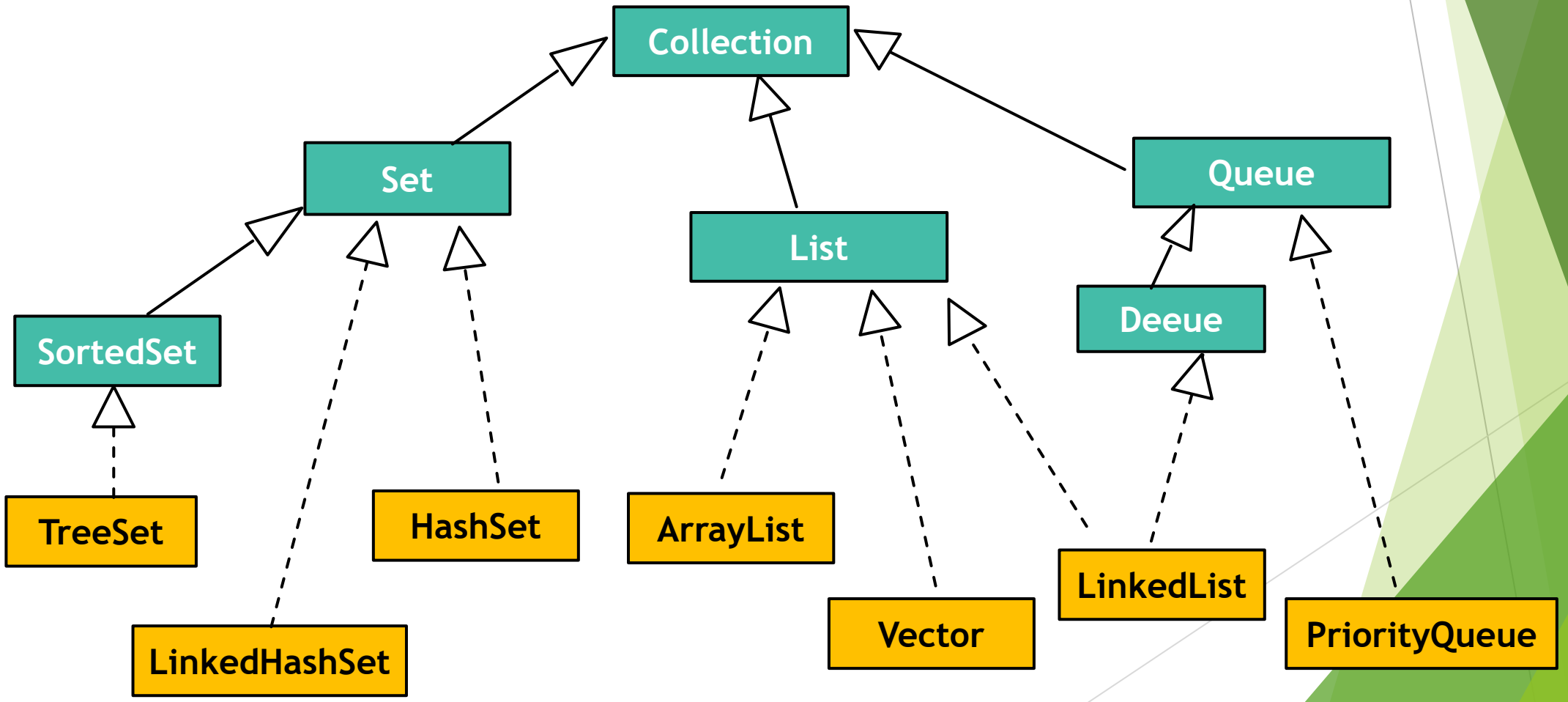
- The `Collection<E>` interface extends `Iterable<E>`

  In its turn `Iterable<E>` is useful in allowing us to generate iterators over the collection



The Java Collection Framework hierarchy

- Two main interfaces: **Collection** and **Map**

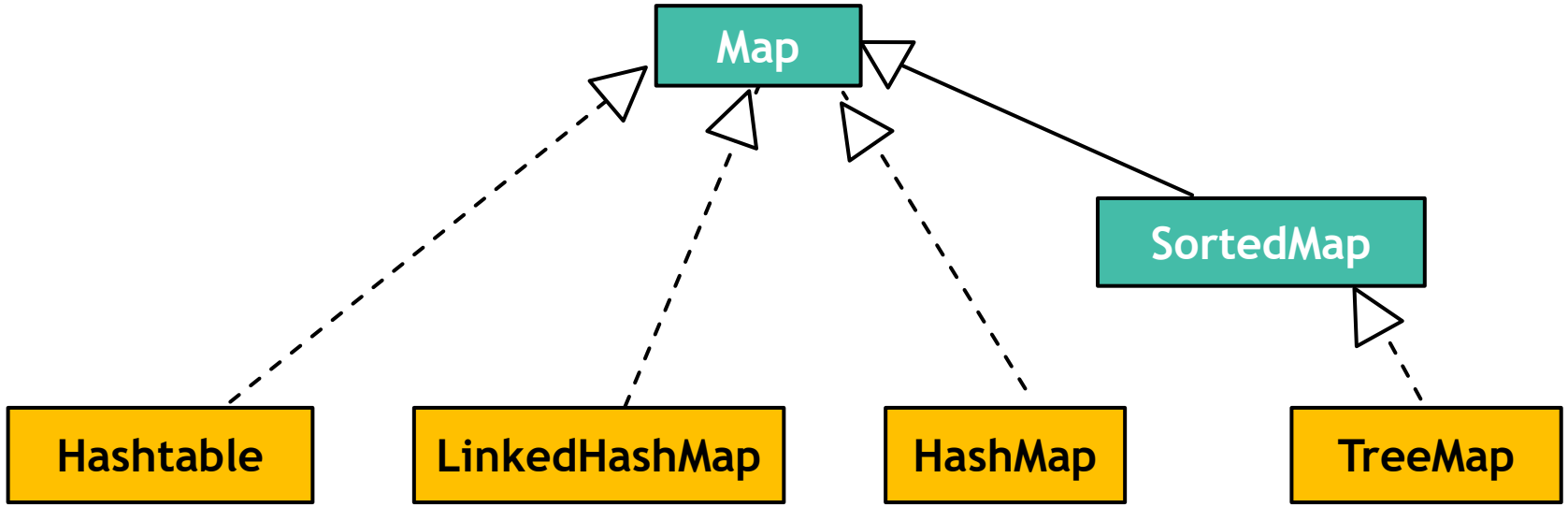


The Collection interface

- A generic interface for collections of objects

- Different collections implement different subinterfaces:
  - **Set**: a collection without repeated elements
  - **SortedSet**: a set whose elements are ordered increasingly
  - **List**: a collection allowing for duplication
    the elements are indexed
  - **Queue** : a collection that stores elements awaiting processing
  - **Deque** (double-ended queue) : a queue allowing to insert and process elements at both ends of the queue

The Map interface



- The Map et SortedMap interfaces:
  - **Map**: associates values to keys
  - **SortedMap** : a map that is sorted by key values

Some collections in practice

Sets

- An unindexed collection with no repeated elements
  - We can never access an element by using its "position"
- Set vs. array:
  - Sets can have an arbitrary, modifiable size
  - Elements are easier to access in arrays, by using their indexes
  - Sets have inbuilt methods for simple element manipulation: adding/removing elements, checking if an element is in the set...

Using a set

- A collection with no repetitions
- An interface with specific methods of interface Collection, with restrictions on repetitions
- Classes implementing Set:
  - HashSet: the elements are stored in a hash table (yielding constant-time operations)
  - TreeSet: the elements are stored in order in a tree
  - LinkedHashSet: the elements are stored in order given their moment of insertion

```
import java.util.Collection;
import java.util.HashSet;
import java.util.Set;

public class LearnSets{
    public static void main(String[] args){
        final Set<String> mySet = new
HashSet<String>();
        mySet.add("John");
        mySet.add("Gabi");
        mySet.add("John"); // not added (repetition)
        System.out.println(mySet); // prints {John,
Gabi} or {Gabi, John}
        System.out.println(mySet.size());   //
prints 2

}
```

Iterators and printing

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

public class LearnSets{
    public static void main(String[] args){
        final Set<String> mySet = new
HashSet<String>();
        mySet.add("John");
        mySet.add("Gabi");

        Iterator<String> iterator =
mySet.iterator();
        while (iterator.hasNext()){
            String currentItem = iterator.next();
            System.out.println(currentItem);
            if (currentItem.equals("John"))
                System.out.println("It's John");
        }
    }
}
```

Iterators are useful for iterating through an unordered collection

The iterator is specific to `mySet`

The method `hasNext()` checks if iterator still has room to iterate

The `next()` method returns the object the iterator will encounter next

Compares elements

Iterators and printing

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

public class LearnSets{
    public static void main(String[] args){
        final Set<String> mySet = new
HashSet<String>();
        mySet.add("John");
        mySet.add("Gabi");

        Iterator<String> iterator = mySet.iterator();
        while (iterator.hasNext()){
            String currentItem = iterator.next();
            System.out.println(currentItem);
            if (currentItem.equals("John"))
                System.out.println("It's John");
        }
    }
}
```

We can view the set as a text, in which the words are set one space apart

```
Gabi John
```

An iterator is like a cursor in the text

Evtery time we call next() :

- the cursor moves one position to the right
- the object between its former and current positions becomes currentItem

Other operations with sets

```
import java.util.Collection;
import java.util.HashSet;
import java.util.Set;

public class LearnSets{
    public static void main(String[] args){
        final Set<String> mySet = new
HashSet<String>();
        mySet.add("John");
        mySet.add("Gabi");
        System.out.println(mySet.size());   //
prints 2

        mySet.remove("Gabi");
        System.out.println(mySet); // prints John
        mySet.add("Alice");
        if (mySet.contains("Gabi"))
            System.out.println("Gabi is still
here.");
    }
}
```

removes an element

checks if the set contains a given element -- no need to actually iterate through it!

Using lists

- Lists are indexed
- Repetitions are allowed
- Additional methods:
  - To handle elements whose index we know
  - To partition a list in sublists
  - To sort, mix, invert, copy or find elements
- Types of lists:
  - ArrayList: a resizable array
  - LinkedList : a list that is manipulable at both ends

```
import java.util.Collection;
import java.util.List;
import java.util.ArrayList;

public class LearnLists{
    public static void main(String[] args){
        final List<Integer> myList = new
ArrayList<Integer>();
        myList.add(Integer.valueOf(10));
        myList.add(Integer.valueOf(5));
        myList.add(Integer.valueOf(8));
        myList.add(Integer.valueOf(10)); //added
        System.out.println(myList); // prints
{10,5,8,10}
        System.out.println(myList.get(2));   //
prints 8, as indexing starts at 0
}
```

Integer: a class that encapsulates an int into an object
Remember: collections only contain objects!

More ways to handle lists

```
import java.util.Collection;
import java.util.List;
import java.util.ArrayList;
import java.util.Collections;

public class LearnLists{
    public static void main(String[] args){
        final List<Integer> myList = new
ArrayList<Integer>();
        myList.add(Integer.valueOf(10));
         myList.add(Integer.valueOf(5));
         myList.add(Integer.valueOf(8));
         myList.add(Integer.valueOf(10));

        System.out.println(myList.indexOf(10));
// prints 0
        System.out.println(myList.lastIndexOf(10));
// prints 3
        Collection.sort(myList);
        System.out.println(myList); //prints
{5,8,10,10}
}
```

Part of the Java Collection Framework
Includes many abstract methods, including sorting

Finds the first index of a given element

Finds the last index of a given element

A method that works on all lists with sortable elements
Including lists on Strings and Integers

Using a Map

- A map consists of pairs (key, value) such that keys are unique
  - Example of keys: student numbers
  - The keys are used to index the data
- Methods allowing us to manipulate elements by keys, or keys themselves
- Types of maps:
  - HashMap: keys/elements stored in a hash table (more efficient)
  - TreeMap: elements stored in a tree (and ordered)
  - LinkedHashMap: tradeoff between HashMap (efficient) et TreeMap (ordered)

```
import java.util.HashMap;
import java.util.Map;

public class LearnMaps{
    public static void main(String[] args){
        final Map<String, String> 00Agents = new
HashMap<>();
        00Agents.put("002", "Bill Fairbanks");
        00Agents.put("006", "Alec Trevelyan");
        00Agents.put("007", "James Bond");
        00Agents.put("002", "Johnny English"); // not
added, duplicate key
        System.out.println(00Agents); // prints {002=Bill
Fairbanks, 006=Alec Trevelyan, 007=James Bond}
}
```

Other operations with Maps

```
import java.util.HashMap;
import java.util.Map;

public class LearnMaps{
    public static void main(String[] args){
        final Map<String, String> 00Agents = new
HashMap<>();
        00Agents.put("002", "Bill Fairbanks");
        00Agents.put("006", "Alec Trevelyan");
        00Agents.put("007", "James Bond");

        System.out.println(00Agents.get("007"));
        System.out.println(00Agents.size());
        if (00Agents.containsKey("001")){
            System.out.println(00Agents.get("001"));
        }
        else{
            System.out.println("Vacant position.");
        }
        System.out.println(00Agents.keySet());
        System.out.println(00Agents.values());
}
```

Find the value associated to the key "007"

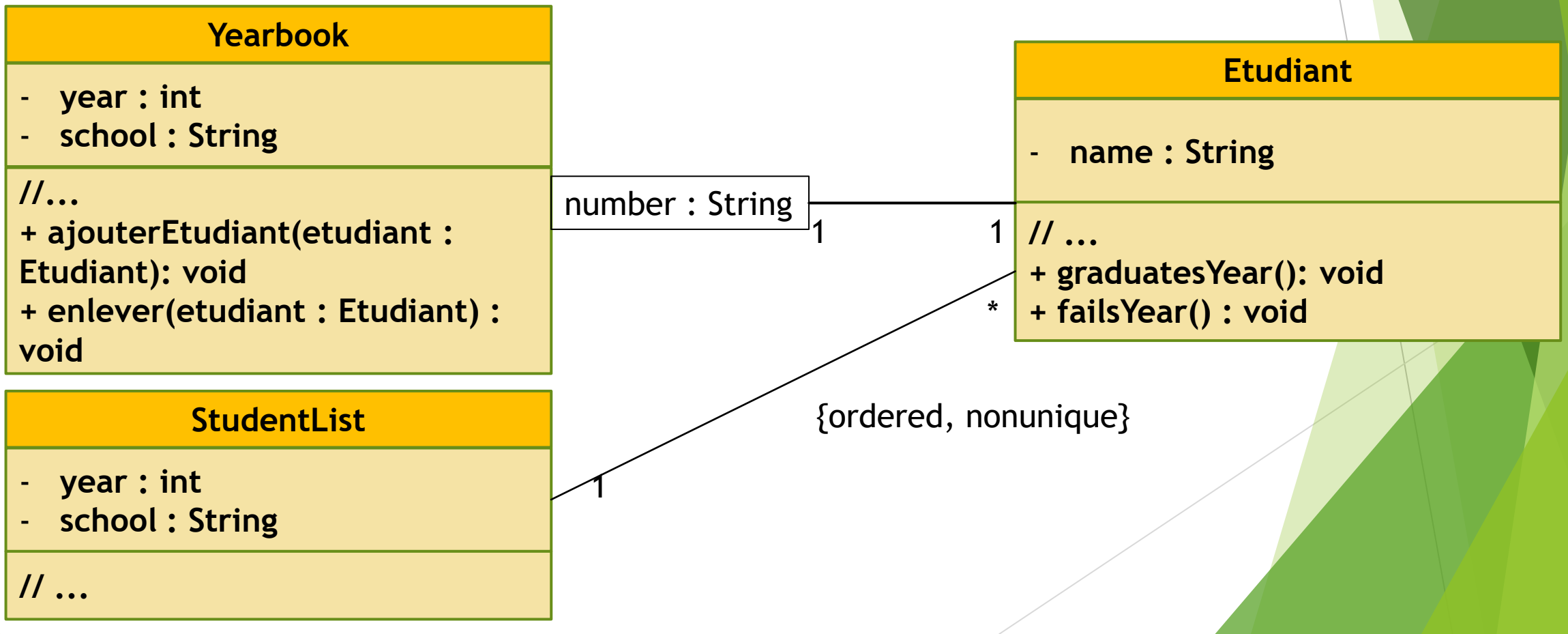
Searches through the keys
We can search values by using **contains()**

Prints a set of keys: **{"002", "006", "007"}**

Prints a set of values

Class diagrams

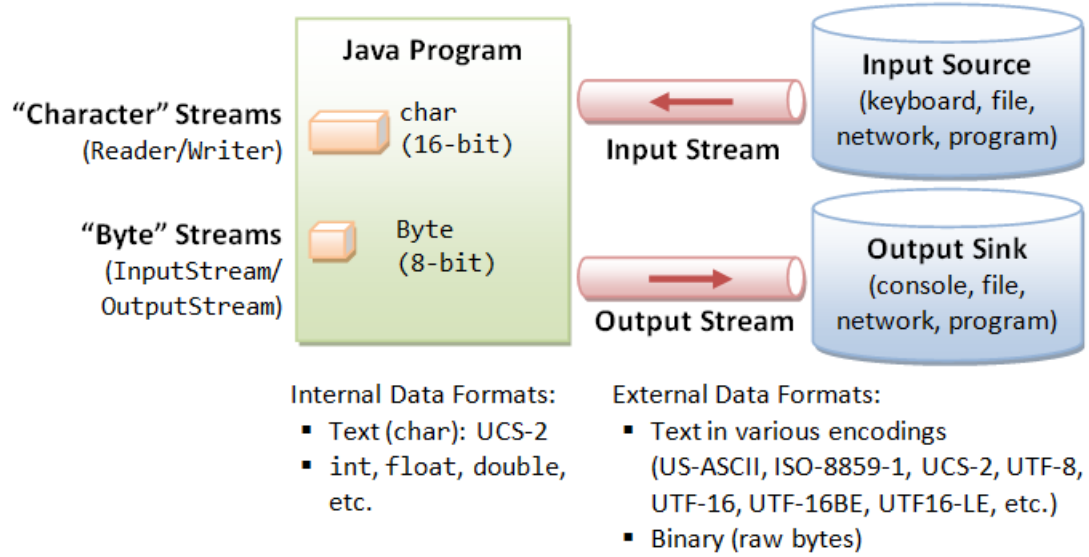
- Collections are generally indicated by means of associations
  - A less standard notation: use attributes directly



Input and output streams

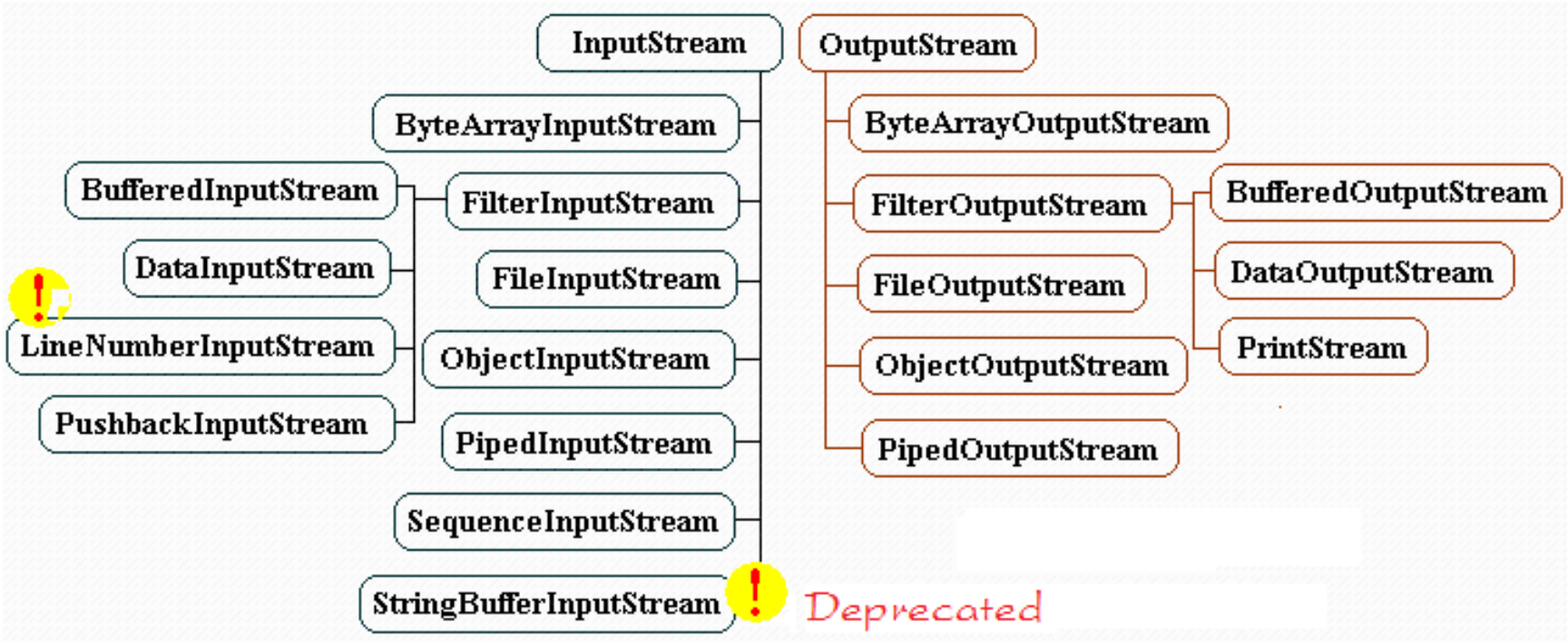
The java.io package

- Input/output **streams** are essential in Java
  - Such streams are unidirectional
  - There are multiple input sources and output destinations in Java
  - Input sources: keyboard input, file, network input, input from another program
  - Output destinations: Java console, file, another program, output on network
- Two types of streams : binary and character streams



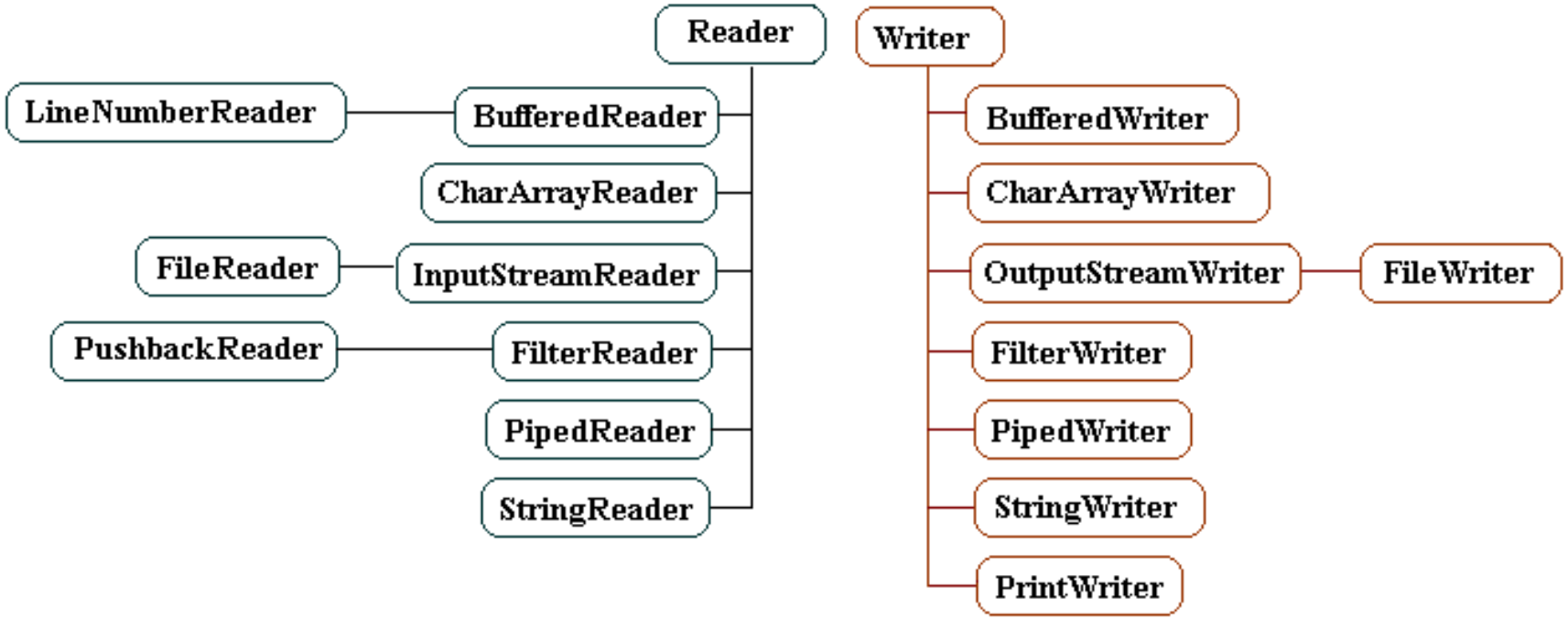
Source : https://www.ntu.edu.sg

Binary Streams in Java



Source: https://o7planning.org/

Character streams



Source:
https://o7planning.org/

Reading a text file

- A correct manipulation of files in Java involves:
  - Correctly opening files
    - With a correct treatment of possible exceptions, including FileNotFound
  - Manipulation (for instance read/write operations)
  - Correctly closing the files
    - All files **must** be closed after use
    - In particular: even if an error occurs or an exception is triggered, the file must be correctly closed while the execution proceeds!
- Let's look at these steps one by one

Opening and reading a file

- A file can be read with a fileReader
  - We will use the constructor of class FileReader with the signature

    public FileReader(String fileName)

    throws FileNotFoundException
  - The methods in which we instantiate the FileReader must take into account the exceptions!

- Useful methods in class FileReader:
  - public int read(): reads a single character, throws an IOException
  - public int read(char[] buffer) : readers characters from a character array

    better efficiency, throws an IOException

Closing a file

- Always close files when you've finished using them
- 2 ways of doing this:
  - the finally block of a try-catch-finally block is always run
  - Since Java 7 try-catch suffices, due to the interface java.lang.Autocloseable
    - an interface implemented by most readers/writers and I/O streams
    - However, it is still a good idea to concretely close the files

```
import java.io.*;

public class Reading{
    public static void main(String[] args){
        try (FileReader reader = new
FileReader("C:/Documents/File.txt")) {
            // read character by character
            int character;
            while ( (character = reader.read()) != -1  )
                System.out.print((char)character);
        }
        catch(IOException e) {
            e.printStackTrace();
        }
}
```

Optimising reading

- Reading character by character is inefficient
- To improve efficiency we can read windows of several characters at a time
  - We can choose the window size
- For large file, the second method can make all the difference

```
import java.io.*;

public class Reading{
     public static void main(String[] args){
          try{
               FileReader reader = new
FileReader("C:/Documents/File.txt");
               char[] window = new char[128];
               while ( (reader.read(window)) != -1  ){
                    for (int i=0; i<128; i++){
                         System.out.print(window[i]);
                    }
               }
          }
          catch(IOException e) {
               e.printStackTrace();
          }
}
```

Writing to a file

- Use a file writer: FileWriter
- Two interesting FileWriter constructors:
  - public FileWriter(String filename) : throws IOException
  - public FileWriter(String filename, boolean append) : throws IOException
    - if append == true, then the input data is written at the end of the file; oterwise they are written at the start of it
    - very useful in a log or in a file in which order is important
- We can end the line and start on a fresh line by using "\n".

Writing into the file

```
import java.io.*;

public class Writing{
    public static void main(String[] args){
        try{
            FileWriter writer = new FileWriter("C:/Documents/File.txt", true);
            writer.write("I'm continuing to write in this file \n and here is the rest
of my sentence");
        }
        catch(IOException e) {
            e.printStackTrace();
        }
}
```

Of further use...

- The BufferedReader class optimizes reading for files
- Class File

- More information :
  - https://docs.oracle.com/javase/7/docs/api/java/io/BufferedReader.html
  - https://docs.oracle.com/javase/7/docs/api/java/io/File.html

Questions ?