Data structures

Vũ Thị Hồng Nhạn

(vthnhan@vnu.edu.vn)

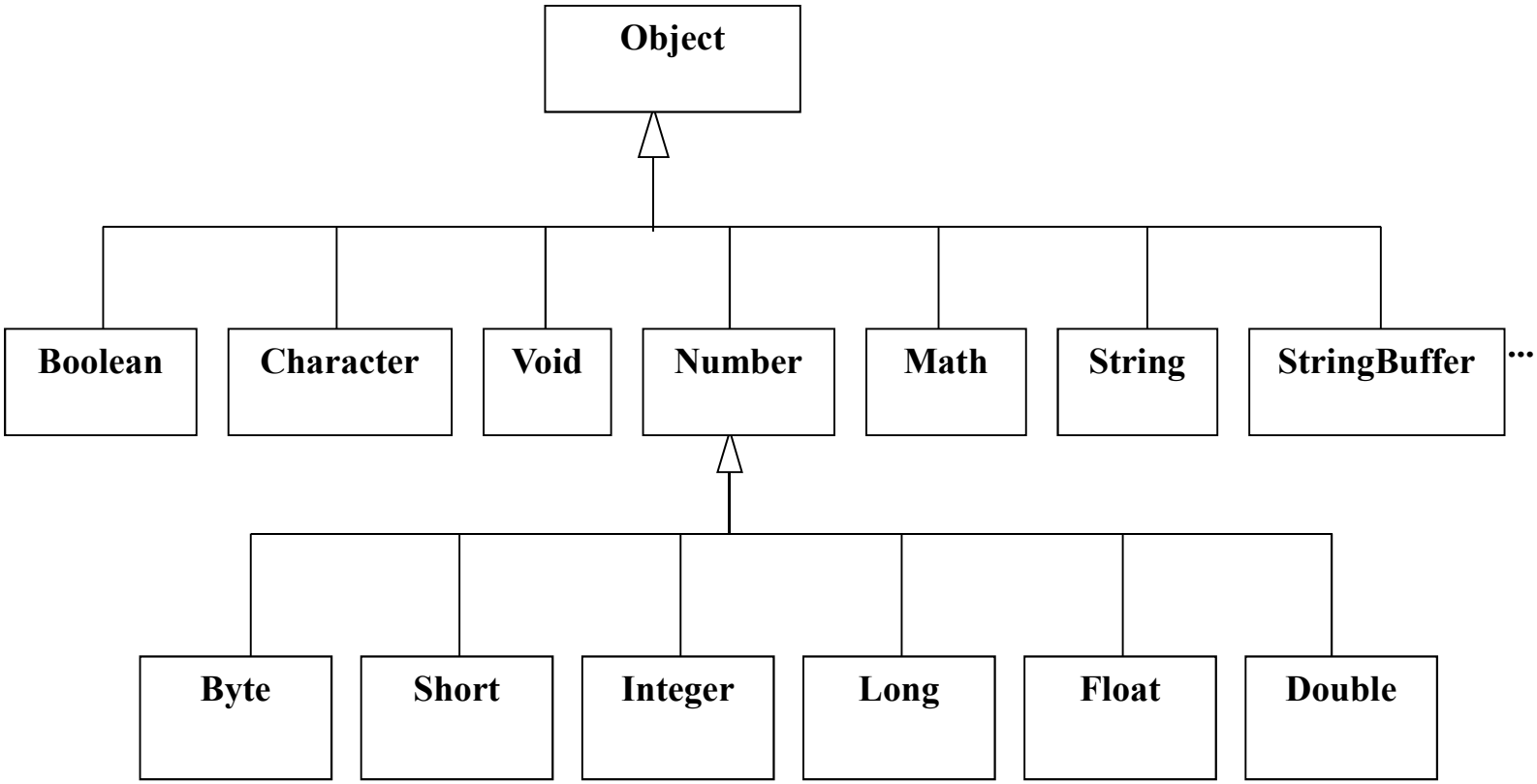
Dept. of Software Engineering, FIT, UET,

Vietnam National Univ., Hanoi

Content

- Primitive data types
- String
- Math class
- Arrays
- Collections (container classes)
  - Lists
  - ArrayLists
  - Stacks
  - Queues
- HashMap
- Summary

Basic classes



The Object class

- Class getClass() returns class name of the current objects.

```
Cat a = new Cat(”Tom”);
Class c = a.getClass();
System.out.println(c);
```

- boolean equals(Object) compares objects, is usually redefined.
- String toString() returns text representation of the object, is usually redefined.

Wrapper classes

- A wrapper class wraps around a data type and gives it an object appearance
- Wrapper classes include methods to unwrap the object and give back the data type
- Example

```
int x = 100;
Integer o = new Integer(x)
```

  - The int data type x is converted into an object, *o* using Integer class
- Unwrap an object
  - int y= o.intValue()
  - System.out.println(y*y);// 10000

Primitive data types

- Utility methods:
  - **valueOf(String s)** returns an object of the corresponding type holding the value of String s.

    ```
    Integer k = Integer.valueOf( "12"); // k = 12
    ```
  - **typeValue()** returns primitive value of the object

    ```
    int i = k.intValue();  // i = 12
    ```
  - **static parseType(String s)** converts a string into a value of the corresponding primitive type

    ```
    int i = Integer.parseInt("12");  // i = 12
    ```
- constants
  - **Type.MAX_VALUE, Type.MIN_VALUE**

The Character class

- Methods
  - static boolean isUppercase(char ch)
  - static boolean isLowercase(char ch)
  - static boolean isDigit(char ch)
  - static boolean isLetter(char ch)
  - static boolean isLetterOrDigit(char ch)
  - static char toUpperCase(char ch)
  - static char toLowerCase(char ch)

The String class

- **String**: unmodifiable sequence of characters.
- Initialize
  - **String(String)**
  - **String(StringBuffer)**
  - **String(byte[])**
  - **String(char[])**
- Methods
  - **int length** the length of the string
  - **char charAt(int index)** returns the character at position index

The String class

- Comparison
  - **boolean equals(String)**
  - **boolean equalsIgnoreCase(String)**
  - **boolean startWith(String)**
  - **boolean endWith(String)**
  - **int compareTo(String)**
- Conversion
  - **String toUpperCase()**
  - **String toLowerCase()**
- Concatenation
  - **String concat(String)**
  - operator “+”

The String class

- Search forwards
  - **int indexOf(int ch)**
  - **int indexOf(int ch, int from)**
  - **int indexOf(String)**
  - **int indexOf(String s, int from)**
- Search backwards
  - **int lastIndexOf(int ch)**
  - **int lastIndexOf(int ch, int from)**
  - **int lastIndexOf(String)**
  - **int lastIndexOf(String, int)**

The String class

- Replace
  - **String replace(char oldChar, char newChar)** returns a new string resulting from replacing all occurrences of oldChar with newChar
- Substring
  - **String trim()** returns a copy of the string, with leading and trailing white space omitted.
  - **String substring(int startIndex)**
  - **String substring(int start, int end)**

StringBuffer

StringBuffer: modifiable sequence of characters

- Initialize
  - **StringBuffer(String)**
  - **StringBuffer(int length)**
  - **StringBuffer()**: default size is 16
- Utilities
  - **int length()**
  - **char charAt(int index)**
  - **void setCharAt(int index, char ch)**
  - **String toString()**

StringBuffer

- Edit
  - **append(String)**
  - **append(type t)** appends t's string representation
  - **insert(int offset, String s)**
  - **insert(int offset, char[] chs)**
  - **insert(int offset, type t)**
  - **delete(int start, int end)** deletes a substring
  - **delete(int index)** deletes 01 character
  - **reverse()**

The Math class

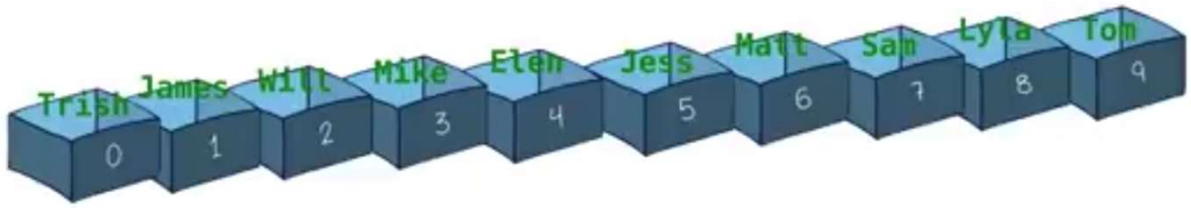
- Constants
  - **Math.E**
  - **Math.PI**
- Static methods
  - **type abs(type):** absolute value of int/double/long
  - **double ceil(double)**
  - **double floor(double)**
  - **int round(float)**
  - **long round(double)**
  - **type max(type, type), type min(type, type)**

The Math class

- Static methods (cont.)
  - `double random()` generates random value in the range **[0.0,1.0]**
  - `double pow(double, double)`
  - `double exp(double)` *e* raised to the power of a value.
  - `double log(double)` natural logarithm (base *e*)
  - `double sqrt(double)`
- trigonometric
  - `double sin(double)` returns sine of an angle
  - `double cos(double)` returns cosine of an angle
  - `double tan(double)` returns tangent of an angle

Arrays

- We already know the best way to store multiple items of the same type is to use arrays
- Array simply allocates a bunch of cells to store each item **separately**
  - but the whole array is treated as if it's **a single variable**
- Each cell in the array has an index that start from zero and these indices allow you to access each item individually inside the array



String[] names = new String[10]

```
names[0] ⟶ Trish
names[1] ⟶ James
names[2] ⟶ Will
```

Array

- An array is an object must be created (**new**) before use

```
int a[];
a = new int[10];
for (int i = 0; i < a.length; i++) a[i] = i * i;
for (int j: a)
    System.out.print(j + " ");

int b[] = {2, 3, 5, 7};
a = b;
int m, n[];
double[] arr1, arr2;
```

Array as argument and return value

```
int[] myCopy(int[] a)
{
    int b[] = new int[a.length];
    for (i=0; i<a.length; i++)
            b[i] = a[i];
    return b;
}
...
int a[] = {0, 1, 1, 2, 3, 5, 8};
int b[] = myCopy(a);
```

Multi-dimensional arrays

```
int a[][];
a = new int[10][20];
a[2][3] = 10;
for (int i=0; i<a[0].length; i++)
    a[0][i] = i;
for (int j: a[0])
    System.out.print(j + " ");

int b[][] = { {1 , 2}, {3, 4} };
int c[][] = new int[2][];
c[0] = new int[5];
c[1] = new int[10];
```

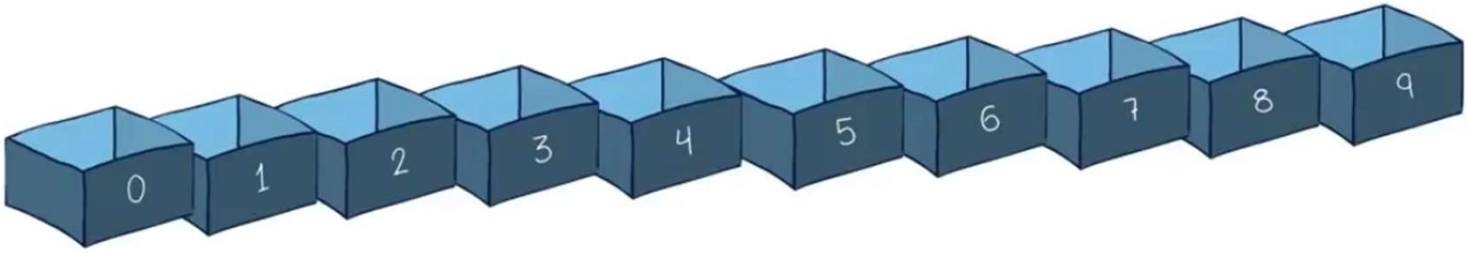
Array copy

- `System.arraycopy(src, s_off, des, d_off, len)`
  - **`src`**: source array, **`s_off`**: source array's offset
  - **`des`**: destination array, **`d_off`**: destination array's offset
  - **`len`**: number of entries to be copied
- Entry's content is copied
  - Primitive value
  - Object reference.

The Array class

- In the package **`java.util`**
- Four static methods
  - `fill()` initializes array entries with one same value
  - `sort()` sorts array
    - works with arrays of primitive values
    - works with classes that implement **`Comparable`**
  - `equals()` compares two arrays
  - `binarySearch()` performs binary search in sorted arrays, creates logic error if used for unsorted arrays.

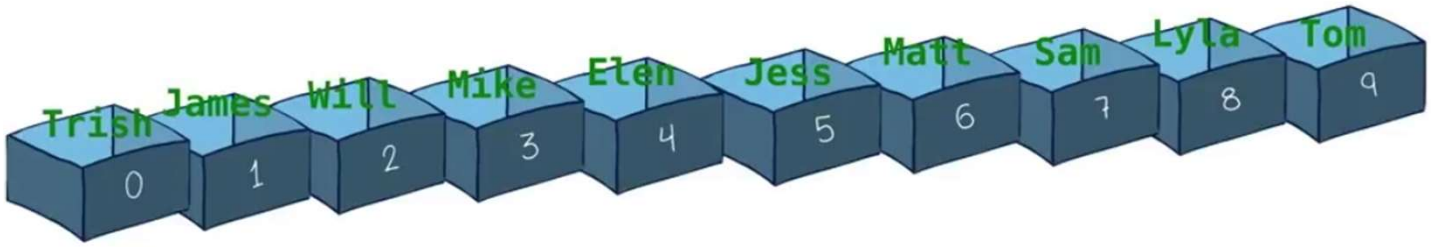
Limitations




```
String[] names = new String[10];
```

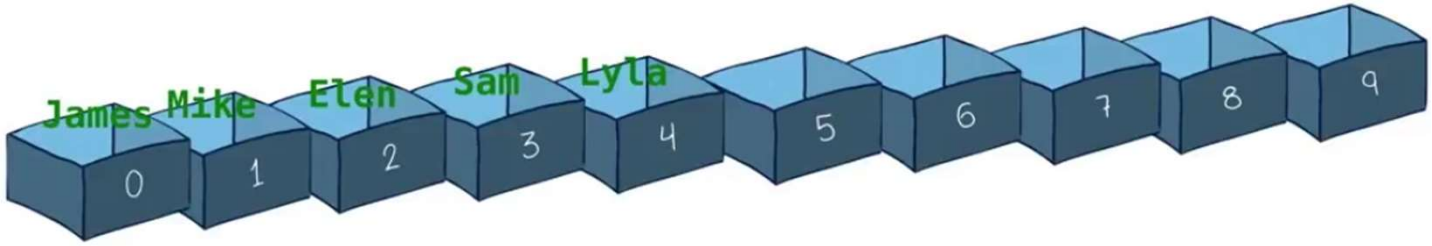
- Firstly, you need to know exactly the number of items you're going to be using in that array while you're initializing it, which is even before you start using it
- Once you initialize an array with a specific number you're not allowed to add or remove any cells
  - i.e., if you try to access an index that doesn't exists within the bounds of the array it will show you a runtime error
  - e.g., names[100]= "Mike" **!!!**

Limitations



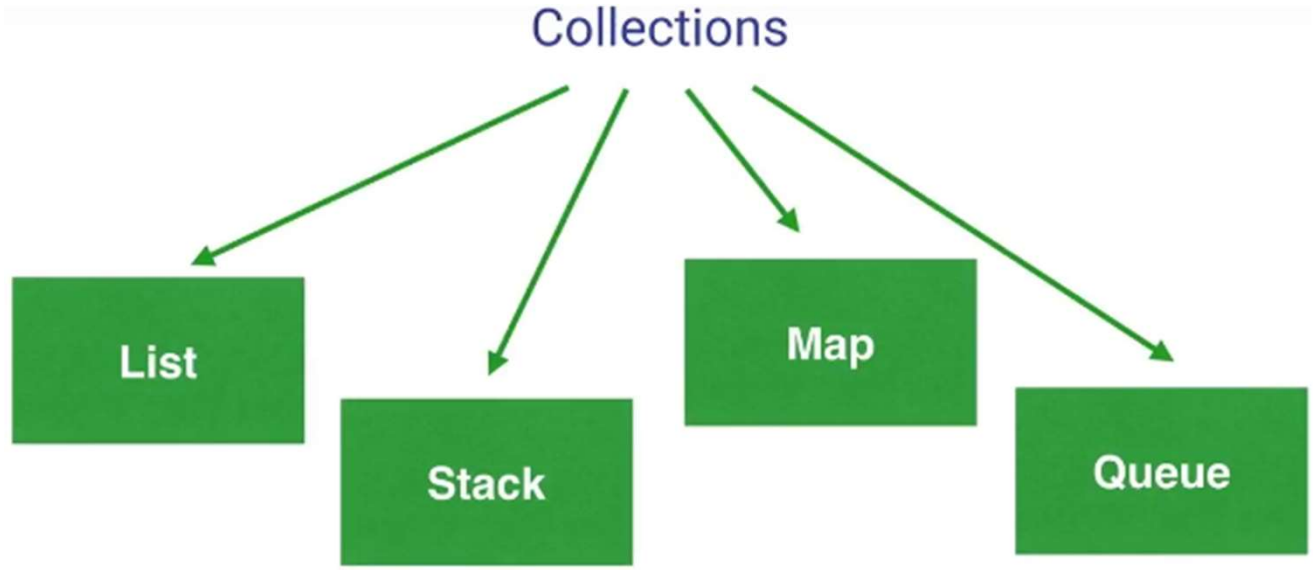
- Another limitation is when you start filling that array, you're not allowed to add any extra items without replacing an existing one

- When you start removing items it create these weird gaps in the array
  - and you have to shift items manually to fill those gaps



Collections

- Collections are simply **a bunch of different classes** and i**nterfaces** that Java offers you to use to simplify your dealing with multiple items of the same type



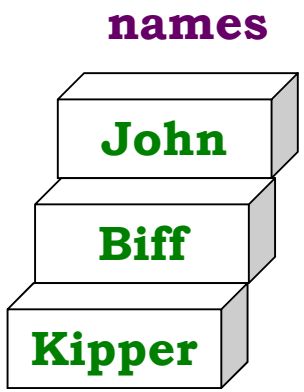
List

- A **List** in Java is an **interface t**hat behaves very similarly to an array
  - It's an ordered collection (also known as a sequence)
  - The user of this interface has **precise control** over where each item is inserted in the list
  - The user can access items by their **integer index** (position in the list)
  - The user can search for items in the list by looping over the items in it

- ArrayList is the most common class that implements the **List interface**
  - **It** uses an array internally

ArrayList

- is simply a wrapper around an array
- allows you to initialize this collection variable without specifying the number of items that you will need in it
  - ArrayList **names**= **new** ArrayList();
  - an array called **names** is created



- You could use the add method that exists in the ArrayList class to continue to add items without *worrying about any indices* or any array implementation in the background
  - E.g., names.add("**John**"); names.add("**Biff**"); names.add("**Kipper**");
- You can also use the remove method inside the ArrayList class which would take care of removing the items as well as shifting all the other items back to reorganizing your indices
  - E.g, names.remove("**Biff**");

ArrayList…

- It provides really powerful methods that are dealing with the array much simpler,
  - check the link for the full list of methods
- E.g.,
  - **add(E element**): appends the specified element to the end of this list
  - **add(int index, E element):** appends the specified element to the specified index of this list
  - **get(int index)**: returns the element at the specified position in this list
  - **contains(Object o):** returns true if this lists contains the specified element
  - **remove(int index)**: removes the element at the specified position in this list
  - **size():** returns the number of elements in the list
  - **clear()**: clear the entire list

Loops

- Just like with arrays, the best way **to access** each and every element in an ArrayList is to create a loop and use the loop counter as an index

```
int size =list.size();
for(int i=0; i<size; i++){
        System.out.prinltn(list.get(i));//get element at index "i" and print it
}
```

- Another type of loop that's a shorthand for loops
  - consists of 2 parts, declaring item variable followed by a colon " **:** " then the ArrayList variable (or any collection type)

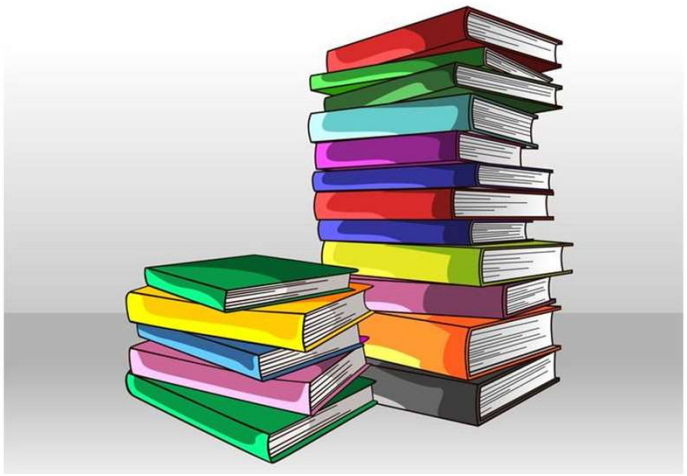
```
for(String  i: list){
        System.out.prinltn(i);//get element at index "i" and print it
```

Loops

- Method indexOf(Object o)
  - returns the index of the first occurrence of the specified element in this list
  - or -1 if that element is not contained in the list
- So when we want to search if an object is in the list, instead of using a loop we can replace it with just one line
  - list.indexOf("Biff");
  - if -1 is returned, then Biff is not in the list
- Check the link for more information in the ArrayList class

Stacks

- The stack collection represents a last-in-first out (LIFO) stack of objects



- **Stack class** includes these 5 methods
  - **push(e Item):** add an item onto the top of the stack
  - **pop():** remove the object at the top of the stack and return that object
  - **peek():** returns the object at the top without removing it from the stack
  - **empty()**: checks if this stack is empty
  - **search(Object o):** searches for an object in the stack and returns its position

Example

- An example of when a stack is useful would be
  - when developing something like an email system
  - once the email server receives a new email, it would add this email to the top of the stack of emails so that the user will read the latest email first
- E.g.,

```
Stack newsFeed = new Stack();
newsFeed.push("Morning news");
newsFeed.push("Afternoon news");
newsFeed.push("Everning news");
String breakingNews = (String) newsFeed.pop();
    System.out.println(breakingNews);
String moreNews = (String) newsFeed.pop(); System.out.println(moreNews);
```

What will the code print?

Queue interface

- is another common collection type in Java
- unlike Stack, it's First In First Out (FIFO) data type
  - the first element added to the queue is the first element to be accessed or removed

- The Queue is only an **interface** and not a Class by itself, however, it defines 2 important methods for all classes that do implement the Queue interface
  - **add(E element)**: inserts the specified element into the queue
  - **poll():** retrieve and remove the head of this queue

Deque & LinkedList classes

- Deque is a special type of Queue and is a double-ended queue
  - i.e., you can **add** or **remove** from either end of a Deque (Front or End)

- Along with 2 Queue methods, **a Deque** also offers these methods
  - **addFirst(E element)**: inserts the specified element at the front of this queue
  - **addLast(E element)**: inserts the specified element at the end of this deque
  - **pollFirst()**: retrieves and removes the first element of this queue
  - **pollLast():** retrieves and removes the last element of this queue

- Java also provides **a few classes** that implement the **Queue interface,** perhaps the most popular of all this is the LinkedList

Example of LinkedList

```
import java.util.Queue;
import java.util.LinkedList;
        Queue orders = new LinkedList();
        orders.add("order1");
        orders.add("order2");
        orders.add("order3");
        System.out.print(orders.poll());
        System.out.print(orders.poll());
        System.out.print(orders.poll());
```

- What will the above code print?

Optimization: HashMap

Optimization

- To use the correct data structure for a variable or collection is **performance**
- A single program can be implemented in so many different ways, but only some will run smoothly and fast enough that users will want to continue using it
  - usually, users consider a program “non-responding” if it takes more than 10 seconds to complete an operation

Hashmaps

- **Hashmap** is *a type of collection* that was introduced in Java to speed up the search process in an **ArrayList**
- In some sense, it's just another collection of **items** (*String* or *Integers* or *any other Object*),
  - but the way it stores those items are **unique**
- Hashmaps allow you to store **a key** for **every** item you want to add
  - This **key** has to be **unique** for the entire list,
  - very much like the index of a typical array, except that *this key* can be *any Object of any Type*!
- The point is *to be able to find an item* in this collection **instantly** without having to loop through all the items inside → save the precious run-time

Example

- Consider a class called Book which contains every detail about such book

```
public class Book{
        String title;
        String author;
        int numberOfPages;
        int pulishedYears;
        int edition;
        String ISBN
}
```

Example: using ArrayList

- If you were to create a Library class that will simulate a virtual library of all the books that exist in the world,
  - you can easily create **an ArrayList of Books**, and fill it up with all the book details that you may have

```
public class Library{
        ArrayList<Book> allBooks;
        ……
}
```

- Now, to search for a book by its ISBN, you'll need to create **a loop** to compare the ISBN of each book with the one you're looking for

```
Book findBookByISBN(String isbn){
    for(Book book: allBooks)
        if(book.ISBN.equals(isbn))
            return book;
}
```

Example: using HashMap

- A more optimal solution is to use **a HashMap** instead of ArrayLists
- To use HashMap, you need to import it at the every top of your Java file

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

- This is how you declare HashMap

```
public class Library{
        HashMap<String, Book> allBooks;
        ……
}
```

  - The above declaration means that we are creating a collection of Book with a key of type String

Example: using HashMap...

- To initialize this hash map, use the default constructor like so
  - allBooks = **new** HashMap<String, Book>();
- Then, to add items to the HashMap
  - Book *takeOfTwoCities* = new Book();
  - allBooks.**put**("1234567", *taleOfTwoCities*);
  - ……..
- To search for a book using its ISBN

```
Book findBookByISBN(String isbn){
        Book book = allBooks.get(isbn);
        return book;
}
```

This code will use the String key(ISBN) to find the book instantly in the entire collection of books leading to a much better performing java program

Iterator

- An iterator allows a program to walk through a collection and remove elements from the collection during the iteration.
- Java `Iterator` interface
  - `hasNext()`
  - `next()`
  - `remove()`
- Collection classes implement **`Iterator`**

```
import java.util.*;

public class TestList {
   static public void main(String args[])
   {
      Collection list = new LinkedList();

      list.add(3);
      list.add(2);
      list.add(1);
      list.add(0);
      list.add("go!");

      Iterator i = list.iterator();
      while (i.hasNext()) {
         System.out.println(i.next());
      }
   }
}
```

```
3
2
1
0
go!
```

Summary

- Wrapper classes make the primitive type data to act as objects
- There are two uses with wrapper classes
  - convert simple data types into objects by using a constructor
  - convert strings into data types , methods of type parseXXX() are used
- Features of the Java wrapper classes
  - convert numeric strings into numeric values
  - store primitive data in an object
  - valueOf() method is available in all wrapper classes except Character
  - all wrapper classes have typeValue() method. This method returns the value of the object as its primitive type

Summary...

- Collection in Java is a framework that provides an architecture to store and manipulate the group of objects
- Java collections can achieve all the operations that you perform on a data such as searching, sorting, insertion, manipulation, and deletion
- Java collection framework provides
  - many interfaces such as List, Queue ...
  - many classes suh chas Stack, ArrayList, LinkedList, HashMap...
- Understanding how different data structures work in Java will help you become a great software developer