Cấu trúc dữ liệu trong Java

Bộ môn công nghệ phần mềm

Khoa ccông nghệ thông tin

Trường ĐHCN, ĐHQG Hà Nội

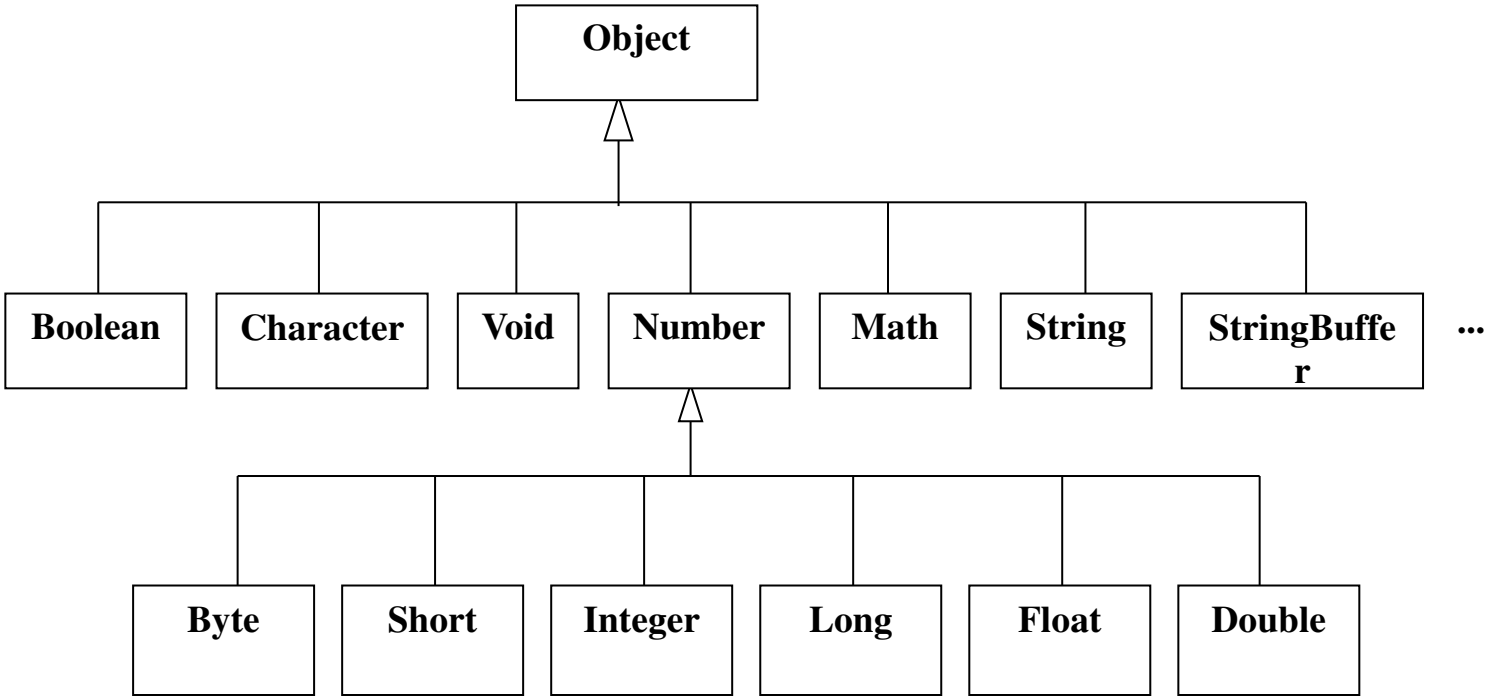
Nội dung

- Lớp `Object`
- Lớp `Wrapper`
- Lớp `String`
- Lớp `StringBuffer`
- Lớp `Math`
- Mảng
- Giao diện `Collection` và các lớp con
- Giao diện `Map` và các lớp con

Lớp `Object`

Cây thừa kế của lớp `Object`

Lớp `Object` là lớp cha của tất cả các lớp đối tượng trong Java



Các phương thức phổ biến

- `Class getClass()`: trả lại lớp của đối tượng hiện tại

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

- `boolean equals(Object obj)`: trả về `true` nếu đối tượng gọi hàm `equal` “bằng” đối tượng `obj`, ngược lại trả về `false`
  - Phương thức này thường định nghĩa lại (override) cho phù hợp với tiêu chí so sánh
- `String toString()`: trả lại biểu diễn của đối tượng dưới dạng xâu ký tự

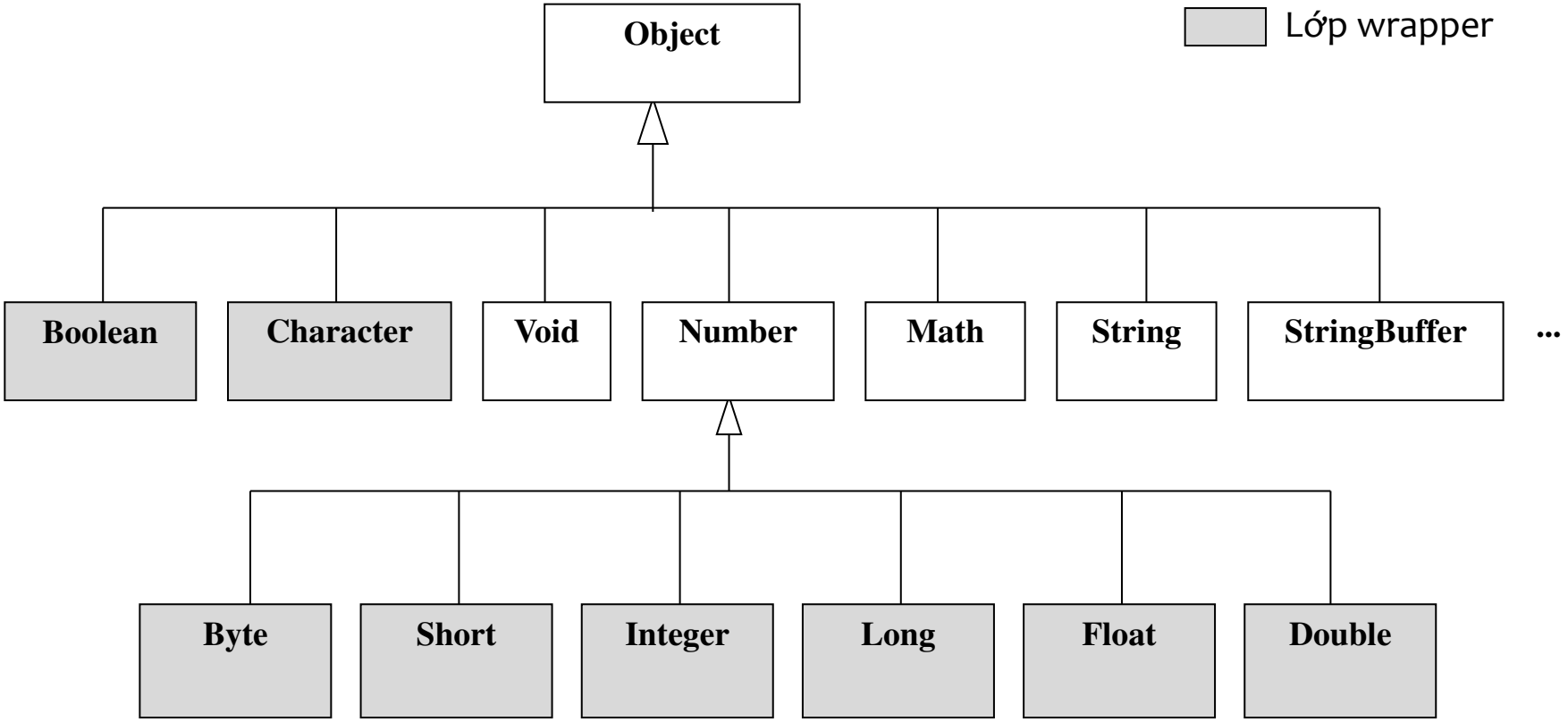
Lớp Wrapper

Định nghĩa

- Lớp wrapper (wrapper class), còn gọi là lớp gói, được dùng để bọc một kiểu dữ liệu nguyên thủy sao cho trông như một đối tượng
- Bảng bên phải mô tả các lớp wrapper phổ biến
- Bài giảng làm rõ hơn về lớp Integer và lớp `Character`

Kiểu dữ liệu nguyên thủy	*Lớp wrapper tương ứng*
byte	Byte
short	Short
int	Integer
long	Long
float	Float
double	Double
boolean	Boolean
char	Character

Cây thừa kế của lớp wrapper



Lớp `Integer`

- Lớp `Integer` là một lớp wrapper phổ biến, được dùng để bọc dữ liệu có kiểu dữ liệu `int`
- Các phương thức phổ biến
  - `Integer valueOf(String str)`: chuyển str thành số nguyên, và bọc trong một đối tượng kiểu `Integer`

```
Integer k = Integer.valueOf( "12"); // k = 12
```

  - `int intValue()`: trả lại giá trị kiểu int của đối tượng `Integer`

```
int i = k.intValue();  // i = 12
```

  - `int parseInt(String str)`: chuyển xâu ký tự str thành giá trị kiểu `int`

```
int i = Integer.parseInt("12");  // i = 12
```

Lớp `Integer`

- Hằng số
  - `Integer.MAX_VALUE`: Giá trị lớn nhất trong khoảng số nguyên (2^31 – 1)
  - `Integer.MIN_VALUE`: Giá trị nhỏ nhất trong khoảng số nguyên (-2^31)

Lớp Integer

- Ví dụ
  - Tạo một đối tượng `Integer` để bọc lấy dữ liệu kiểu `int`

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

  - Giá trị kiểu int có thể được lấy qua hàm `intValue()`

```
int z= o.intValue() // "100"
System.out.println(z*z);// "10000"
```

Lớp `Character`

- Tương tự như lớp `Integer`, lớp `Character` là một lớp wrapper dùng để bọc kiểu dữ liệu kiểu `char`
- Các phương thức phổ biến
  - `isUppercase(char ch)`: trả về true nếu `ch` là kí tự in hoa
  - `isLowercase(char ch)`: trả về true nếu `ch` là kí tự in thường
  - `isDigit(char ch)`: trả về true nếu `ch` là chữ số
  - `isLetter(char ch)`: trả về true nếu `ch` là chữ cái
- Ví dụ: Tạo một đối tượng `Character` bọc lấy kí tự 'a'

```
Character o = new Character('a');
```

Lớp `String`

Lớp `String`

- Đối tượng `String` được dùng để lưu một xâu kí tự
  - Xâu ký tự có thể hiểu là một mảng các ký tự, trong đó kích thước mảng là cố định
  - Xâu ký tự sẽ không thể sửa đổi được
  - Tất cả những thay đổi đối với đối tượng String đều tạo ra một đối tượng `String` mới
- Để cho đơn giản, gọi vắn tắt đối tượng `String` là xâu

Ký tự đặc biệt

- Xâu có thể chứa các ký tự đặc biệt
- Các ký tự đặc biệt trong xâu bắt đầu với ký tự '\'
- Bảng sau trình bày một vài ký tự đặc biệt phổ biến

Ký tự	**Giải thích**
\r\n	Ký tự xuống dòng ở hệ điều hành Windows
\n	Ký tự xuống dòng ở hệ điều hành kiểu Unix
\t	kí tự tab
\f	kí tự ngăn cách giữa các trang (form feed). Bộ soạn thảo có thể dùng kí tự này khi người dùng chèn thêm một trang mới.
\b	kí tự backspace

Khởi tạo xâu

- Có nhiều cách để khởi tạo một xâu
- `String()`: khởi tạo một xâu với độ dài bằng 0 (xâu rỗng)
  - Nếu xâu bị sửa đổi, một vùng nhớ mới được cấp phát để lưu nội dung xâu mới
  - Do đó, khởi tạo mặc định cho đối tượng `String` là ***vô nghĩa!***
- `String(String str)`: khởi tạo xâu từ xâu str bằng cách tạo một bản sao của xâu `str`

Phương thức so sánh hai xâu

- `boolean equals(String)`: trả về true nếu hai xâu bằng nhau
- `int compareTo(String str)`:
  - trả về 0 nếu hai xâu bằng nhau
  - trả về giá trị < 0 nếu xâu đang so sánh nhỏ hơn str
  - trả về giá trị > 0 nếu xâu đang so sánh lớn hơn str
- Ví dụ

```
String myStr1 = ”A";
String myStr2 = ”B";
System.out.println(myStr1.compareTo(myStr2));// -1
```

Phương thức kết nối hai xâu

- `String concat(String str)`: tạo xâu mới có nội dung là xâu hiện tại ghép với xâu `str`

```
String firstName = "John ";
String lastName = "Doe";
System.out.print(firstName.concat(lastName));
// “John Doe”
```

- Có thể sử dụng phép toán “+” để tạo một xâu mới từ hai xâu đã có

Tìm kiếm thông tin trong xâu

Cách 1

- Tìm kiếm theo chiều từ trái sang phải (hay từ đầu xâu đến cuối xâu)
- `int indexOf(String str)`: trả về vị trí xuất hiện đầu tiên của str trong xâu kí tự
- Ví dụ

```
String s="đường đi mua đường";
System.out.print(s.indexOf("đường"));
// in ra vị trí "0"
```

Cách 2

- Tìm kiếm theo chiều từ phải sang trái (hay từ cuối xâu về đầu xâu)
- `int lastIndexOf(String str)`: trả về vị trí xuất hiện cuối cùng của str trong xâu kí tự
- Ví dụ

```
String s = "đường đi mua đường";
System.out.print(s.lastIndexOf("đường"));
// in ra vị trí "13"
```

Các phương thức khác

- Thay thế
  - `String replace(char oldChar, char newChar)`: trả lại một xâu ký tự mới bằng cách thay thế `oldChar` bằng `newChar`
- Xâu con
  - `String trim()`: trả lại xâu ký tự sau khi cắt xén những khoảng trắng ở đầu và cuối
  - `String substring(int startIndex)`: cắt xâu từ vị trí `startIndex`

Lớp `StringBuffer`

Lớp `StringBuffer`

- Lớp `StringBuffer` dùng để lưu một xâu các kí tự
  - Tuy nhiên, có thể thay đổi giá trị các ký tự trong xâu kí tự này mà không tạo bản sao như lớp `String`
  - Xâu có thể chứa các ký tự đặc biệt
- Điểm khác biệt giữa lớp `StringBuffer` và lớp `String`: Xâu tạo bởi lớp `StringBuffer` cho phép chỉnh sửa, trong khi xâu tạo bởi lớp `String` không cho phép chỉnh sửa
  - Nên dùng lớp này khi nội dung của xâu không phải cố định
  - Thích hợp khi cần thao tác chỉnh sửa với xâu có độ dài lớn

Khởi tạo lớp `StringBuffer`

- Có nhiều cách để khởi tạo một xâu
  - `StringBuffer(String)`
  - `StringBuffer(int length)`
  - `StringBuffer()`: kích cỡ mặc định là 16

Phương thức sửa nội dung xâu

Có thể thêm, xóa, sửa nội dung xâu

- `append(String str)`: ghép xâu hiện tại với xâu `str` (hiện đang lưu trong lớp đối tượng `String`)
- `insert(int offset, String str)`: chèn xâu `str` (hiện đang lưu trong lớp đối tượng `String`) vào xâu hiện tại ở vị trí `offset`
- `delete(int start, int end)`: xóa các kí tự từ vị trí `start` đến vị trí `(end – 1)`
- `void setCharAt(int index, char ch)`: thay thế kí tự ở vị trí `index` bởi kí tự `ch`

Các phương thức phổ biến khác

- `int length()`: lấy độ dài của xâu
- `char charAt(int index)`: lấy kí tự ở vị trí `index`
- `String toString()`: chuyển sang thực thể kiểu `String`
- `reverse()`: đảo ngược xâu kí tự

Lớp `Math`

Lớp `Math`

- Lớp `Math` chứa các hàm toán học phổ biến và các hằng số thường gặp
- Lớp `Math` được định nghĩa là lớp `final`, tức là lớp này không thể được thừa kế để mở rộng bởi người dùng
- Các hằng số phổ biến
  - `Math.E` (2.718281828459045)
  - `Math.PI` (3.141592653589793)

Phương thức so sánh

- `type max(type num1, type num2)`: lấy giá trị lớn nhất giữa hai số `num1` và `num2` có cùng kiểu `type`
- `type min(type num1, type num2)`: lấy giá trị nhỏ nhất giữa hai số `num1` và `num2` có cùng kiểu `type`
- Kiểu `type` có thể là `int`, `float`, `double`, v.v.
- Ví dụ

```
// so sánh hai số nguyên 5 và 10
//-> trả về số nguyên 10
System.out.println(Math.max(5, 10));
// so sánh hai số thực 5.0 và 10.0 -> trả về số thực 10.0
System.out.println(Math.max(5.0, 10));
```

Phương thức làm tròn

- `double ceil(double num)`
  - trả về số nguyên nhỏ nhất mà lớn hơn hoặc bằng tham số `num`, trong đó giá trị trả về ở dạng `double`
- `double floor(double num)`
  - trả về số nguyên lớn nhất mà nhỏ hơn hoặc bằng tham số `num`, trong đó giá trị trả về ở dạng `double`
- `int round(float num)`
  - trả về giá trị kiểu int gần nhất với số `num`
- Ví dụ

```
System.out.println(Math.ceil(5.4)); // 6.0
System.out.println(Math.round(5.4)); // 5
System.out.println(Math.floor(5.4)); // 5.0
```

Mảng

Mảng

- Mảng có những đặc trưng sau
  - Kích thước của mảng phải được xác định tường minh khi khởi tạo
  - Mảng có thể lưu giá trị nguyên thủy hoặc các đối tượng có cùng kiểu
  - Mảng có thể có một chiều hoặc đa chiều
- JDK có lớp `Arrays` chứa các phương thức tối ưu cho xử lý trên mảng
- Chú ý phân biệt sự khác biệt giữa mảng và đối tượng `ArrayList`
  - Kích thước của mảng là cố định
  - Kích thước của đối tượng `ArrayList` không cố định (phần sau sẽ nói rõ hơn về lớp `ArrayList`)

Mảng một chiều

- Các phần tử trong một mảng một chiều được đánh số từ 0 đến `n – 1`, trong đó `n` là độ dài của mảng
  - Trong đó, phần tử có chỉ số 0 là phần tử đầu tiên
  - Phần tử có chỉ số `n– 1` là phần tử cuối cùng
- Để truy cập phần tử thứ i của mảng một chiều `arr`, sử dụng cú pháp `arr[i]`

Khởi tạo mảng một chiều

- Để khởi tạo mảng một chiều, sử dụng từ khóa new với cú pháp như sau:

```
<kiểu mảng> <tên mảng>[] = new <kiểu mảng>[<số phần tử>];
```

- Ví dụ
  - `int a[] = new int[10]`: tạo mảng một chiều lưu 10 số nguyên
  - `String a[] = new String[10]`: tạo mảng một chiều lưu 10 xâu ký tự

Ví dụ sao chép mảng một chiều

- Mảng sử dụng làm đối của hàm và trả lại giá trị
  - Hàm `myCopy` nhận vào mảng số nguyên `a`
  - Mảng `a` là mảng 1 chiều
  - Tạo mảng `b` bằng cách copy từng phần tử của mảng `a`

```
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);
```

Mảng đa chiều

- Mảng có thể có nhiều hơn 1 chiều
- Để truy cập phần tử (i, j) của mảng hai chiều `arr`, viết `arr[i][j]`
  - `i` là chỉ số của chiều thứ nhất
  - `j` là chỉ số của chiều thứ hai

```
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 w: a[0])
    System.out.print(w + " ");

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

Sao chép mảng đa chiều

`System.arraycopy(src, s_off, des, d_off, len)`

Trong đó:

- `src`: mảng nguồn
- `s_off`: vị trí bắt đầu sao chép trên mảng nguồn src
- `des`: mảng đích
- `d_off`: vị trí bắt đầu đặt dữ liệu sao chép vào trên mảng đích
- `len`: chiều dài của các phần tử được copy

- Nội dung của các phần tử sẽ được sao chép
  - Giá trị nguyên thủy
  - Tham chiếu đối tượng

Ví dụ:

```
int arr1[] = { 0, 1, 2, 3, 4, 5 };
int arr2[] = { 5, 10, 20, 30, 40, 50 };
// chép 2 phần tử từ vị trí 0 trên arr1
// sau đó, đặt vào arr2 từ vị trí thứ 1
System.arraycopy(arr1, 0, arr2, 1, 2);
// array2 = [5, 0, 1, 30, 40, 50]
// phần tử 10, 20 trên arr2 bị ghi đè
```

Lớp `Arrays`

`Arrays` là lớp tiện ích để xử lý mảng

Các phương thức phổ biến

- `sort(type[] arr)`: trả về chính mảng `arr` đã được sắp xếp theo giá trị tăng dần
  - Thao tác sắp xếp không tạo bản sao mảng (thực hiện trên chính mảng gốc)

`Type` có thể là dữ liệu nguyên thủy hoặc lớp đối tượng

Ví dụ:

```
int[] arr = {1, 3, 2};
Arrays.sort(arr);
// arr = [1, 2, 3]
```

Lớp `Arrays`

- `equals(type[] arr1, type[] arr2)`: trả về `true` nếu hai mảng `arr1` và `arr2` (có cùng kiểu `type[]`) giống nhau

Ví dụ:

```
int[] arr1 = new int[] {1, 2, 3};
int[] arr2 = new int[] {1, 2, 3};
Arrays.equals(arr1, arr2); // true
```

Lớp `Arrays`

- `binarySearch(type[] arr, data_type key)`: trả về vị trí của `key` trong mảng `arr`
  - Mảng `arr` phải được sắp xếp trước khi gọi hàm `binarySearch`
  - Sử dụng phương pháp kiếm nhị phân
  - Sẽ báo lỗi nếu mảng chưa được sắp xếp

Ví dụ:

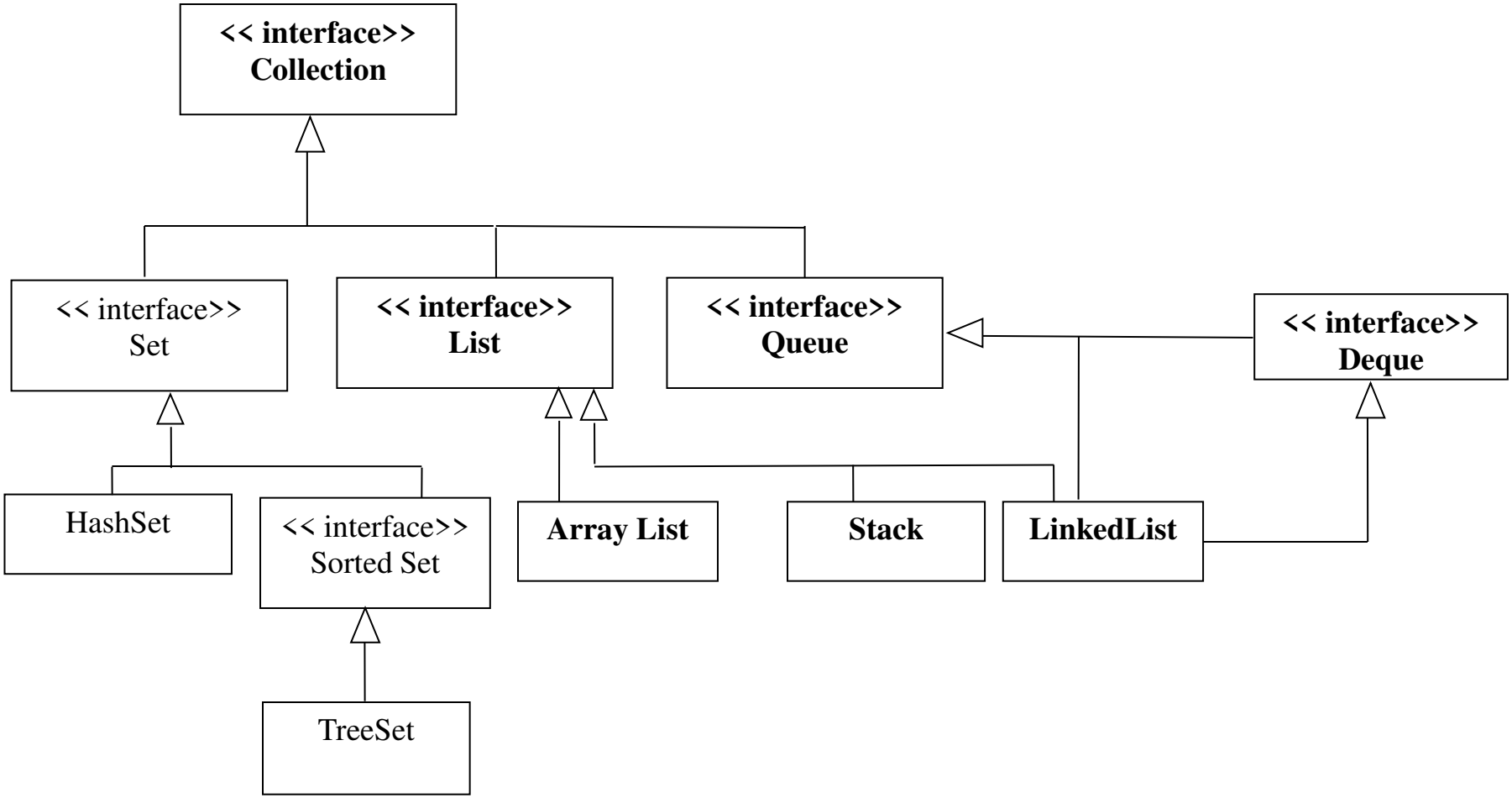
```
int[] arr = {1, 3, 2};
Arrays.binarySearch(arr1, 3);
// 1
```

Giao diện `Collection` và các lớp con

Giao diện `Collection`

- Giao diện `Collection` (hay `interface Collection`) thỏa mãn tính chất sau:
  - Có khả năng chứa một tập các đối tượng (các đối tượng có thể lặp lại hoặc không lặp lại)
  - Có thể duyệt các phần tử trong tập các đối tượng bằng cách sử dụng `for-each`

Cây thừa kế của `Collection`



Giao diện `List`

- Thừa kế giao diện `Collection`
- Phải thỏa mãn các tính chất sau:
  - Có các tính chất của giao diện `Collection` (do `extends Collection`)
  - Là một danh sách có thứ tự (tức mỗi phần tử đều có vị trí xác định trong danh sách)
  - Người dùng điều khiển chính xác được phần tử được chèn vào đâu trong danh sách
  - Có thể truy cập phần tử bằng chỉ mục

Lớp `ArrayList`

- Lớp `ArrayList` là một lớp cài đặt giao diện `List`
- Khởi tạo:

```
ArrayList names = new ArrayList();
```

- Đối tượng `ArrayList` là một danh sách có thứ tự và kích thước không xác định
- So sánh với mảng
  - Số phần tử mảng phải xác định khi khởi tạo
  - Số phần tử của lớp đối tượng `ArrayList` không phải xác định khi khởi tạo

  => lớp đối tượng `ArrayList` linh hoạt hơn mảng

Lớp `ArrayList`

- Phương thức thêm và xóa
  - `add(E element)`: thêm phần tử có kiểu E vào cuối danh sách
  - `get(int index)`: trả lại phần tử ở vị trí xác định
  - `remove(E element)`: xóa phần tử `element` khỏi danh sách
  - Ví dụ phương thức `add()`:

```
ArrayList<String> cars = new ArrayList<String>();
cars.add("Volvo"); cars.add("BMW");
cars.remove(" BMW "); cars.add("Mazda");
System.out.println(cars);
// [Volvo, Mazda]
```

Lớp `ArrayList`

- Phương thức tìm kiếm `indexOf(Object obj)`
  - trả về vị trí xuất hiện đầu tiên của phần tử `obj`; hoặc -1 nếu phần tử `obj` không có trong danh sách
  - Vì thế, khi ta muốn tìm kiếm một đối tượng trong danh sách, thay vì dùng vòng lặp, có thể sử dụng `indexOf`
  - Ví dụ: `list.indexOf("Biff")`; nếu trả về -1 tức là Biff không có trong danh sách list

Lớp `ArrayList`

- Có thể duyệt danh sách bằng vòng lặp tương tự như mảng
  - Ví dụ: xét danh sách `list` có kiểu `ArrayList`

```
int size =list.size();
// size(): cho biết số phần tử trong danh sách
for(int i=0; i<size; i++){
    System.out.prinltn(list.get(i));
}
```

Lớp `Stack`

- Biểu diễn một tập các đối tượng hoạt động theo cơ chế vào trước ra sau (last-in-first out: LIFO)
- Lớp `Stack` có 5 phương thức
  - `push(e Item)`: đưa đối tượng dữ liệu lên đầu ngăn xếp
  - `pop()`: xóa đối tượng dữ liệu khỏi đầu ngăn xếp và trả về đối tượng xóa đó
  - `peek()`: trả lại đối tượng trên đỉnh ngăn xếp nhưng không xóa nó đi
  - `empty()`: trả về `true` nếu ngăn xếp rỗng
  - `search(Object o)`: tìm kiếm và trả lại vị trí của đối tượng `o`

Lớp Stack

- Ví dụ: Phát triển một hệ thống email
  - khi email server nhận được một email mới nó sẽ đặt email lên trên đầu tiên để người dùng luôn đọc email mới nhất

```
Stack newsFeed = new Stack();
newsFeed.push("Tin sáng");
newsFeed.push("Tin chiều");
newsFeed.push("Tin tối");
String breakingNews = (String) newsFeed.pop();
System.out.println(breakingNews); // "Tin tối"
String moreNews = (String) newsFeed.pop();
System.out.println(moreNews); // "Tin chiều"
```

Giao diện `Queue`

- `Queue` chỉ là một `interface` không phải lớp
- Hai phương thức
  - `add(E element)`: thêm một phần tử vào hàng đợi
  - `poll()`: truy cập và xóa phần tử ở đầu của hàng đợi
- Không giống `Stack`, hàng đợi (`Queue`) hoạt động theo cơ chế vào trước ra trước (First In First Out: FIFO)

Giao diện `Deque`

- Thừa kế giao diện `Queue`
- Là kiểu đặc biệt của hàng đợi, nó có 2 đầu
  - Có thể thêm hoặc xóa phần tử từ cả hai đầu của `Deque`
- Cùng với 2 phương thức trong `Queue`, `Deque` cung cấp một số phương thức nữa
  - `addFirst(E element)`: chèn vào đầu
  - `addLast(E element)`: chèn vào cuối
  - `pollFirst()`: trả lại và đồng thời xóa phần tử đầu tiên
  - `pollLast()`: trả lại và đồng thời xóa phần tử cuối cùng

Lớp `LinkedList`

- Java có một vài lớp cài đặt giao diện `Queue`, nhưng quen thuộc nhất là `LinkedList`
- Ví dụ:

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

Giao diện `Iterator`

- Cho phép một chương trình có thể duyệt tập hợp và xóa phần tử trong khi duyệt
- `Iterator` có các phương thức
  - `hasNext()`: trả về true nếu có phần tử kế tiếp
  - `next()`: trả về phần tử kế tiếp
  - `remove()`: loại bỏ phần tử cuối cùng
- Các tập hợp đều cung cấp phương thức để lấy interator bắt đầu của tập hợp

Giao diện Iterator

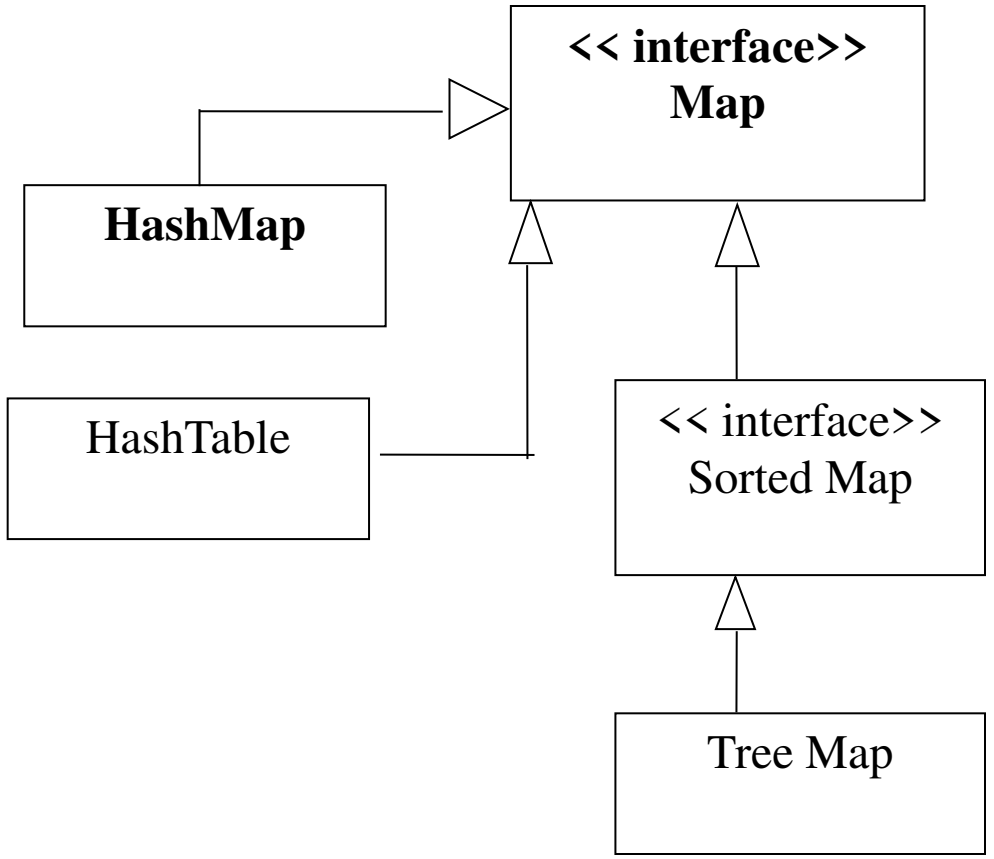
```
public static void main(String args[]){
      Collection list = new LinkedList();

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

      Iterator i = list.iterator();// lấy iterator bắt đầu
      while (i.hasNext()) { // kiểm tra liệu có phần tử kế tiếp
            System.out.println(i.next());
            // lấy phần tử kế tiếp và tăng iterator lên 1 đơn vị duyệt
      }
}
```

3
2
1
0
go!

Giao diện `Map` và các lớp con

Phả hệ của `Map`

- Giao diện `Map` dùng để lưu các phần tử có dạng (key, value), trong đó có tính chất sau:
  - Không có hai phần tử có giá trị key giống nhau
  - key và value là đối tượng
  - Nếu muốn lưu giá trị key có dạng nguyên thủy, cần bọc giá trị key trong lớp wrapper tương ứng



Lớp `HashMap`

- Là cài đặt của giao diện `Map`
  - Nên có các tính chất của giao diện `Map`
- Ưu điểm so với `ArrayList`
  - Có thể tìm một đối tượng ngay tức thì mà không cần phải sử dụng vòng lặp
  - Từ đó, tiết kiệm đáng kể thời gian chạy với lượng dữ liệu tìm kiếm lớn

Lớp HashMap

- Ví dụ: Viết chương trình tìm kiếm sách trong thư viện, trong đó lớp `Book` lưu chi tiết về sách như sau:

```
public class Book{
    String title;
    String author;
    int numberOfPages;
    // ...
}
```

- Hai cách phổ biến
  - Cách 1: Dùng `ArrayList`
  - Cách 2: Dùng `HashMap`

Lớp HashMap

- Cách 1: Dùng ArrayList
  - Tạo một lớp Library lưu sách như sau:

```
public class Library{ private ArrayList<Book> allBooks; /* ... */ }
```

  - Tiếp theo, để tìm kiếm một cuốn sách, bạn cần sử dụng vòng lặp để so sánh ISBN của mỗi cuốn sách với cuốn bạn đang tìm

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

  - Không hiệu quả khi số lượng sách rất lớn (lên tới hàng triệu cuốn sách)

Lớp HashMap

- Cách 2: Sử dụng HashMap
  - Khai báo thư viện như sau

```
public class Library{
      private HashMap<String, Book> allBooks
                        = new HashMap<String, Book>();
      /*...*/
}
```

  - Thêm các đối tượng Book vào HashMap, ví dụ:

```
Book takeOfTwoCities = new Book();
allBooks.put("1234567", taleOfTwoCities);
```

  - Tìm sách theo ISBN

```
Book findBookByISBN(String isbn){
    Book book = allBooks.get(isbn);
    // truy cập tức thời đối tượng -> TỐT HƠN so với cách 1
    return book;
}
```

Tổng kết

- Lớp wrapper có thể làm cho một dữ liệu nguyên thủy hoạt động như một đối tượng và cũng có thể chuyển ngược lại
- Trong Java cung cấp các tập hợp (giao diện, lớp)
  - `List`, `Queue` ...
  - `Stack`, `ArrayList`, `LinkedList`, `HashMap`...
  - Hỗ trợ tất cả các thao tác trên dữ liệu như tìm kiếm, sắp xếp, xóa, chèn...
  - Hiểu được các cấu trúc dữ liệu làm việc như thế nào giúp quyết định hiệu quả khi viết chương trình phần mềm
- v.v.