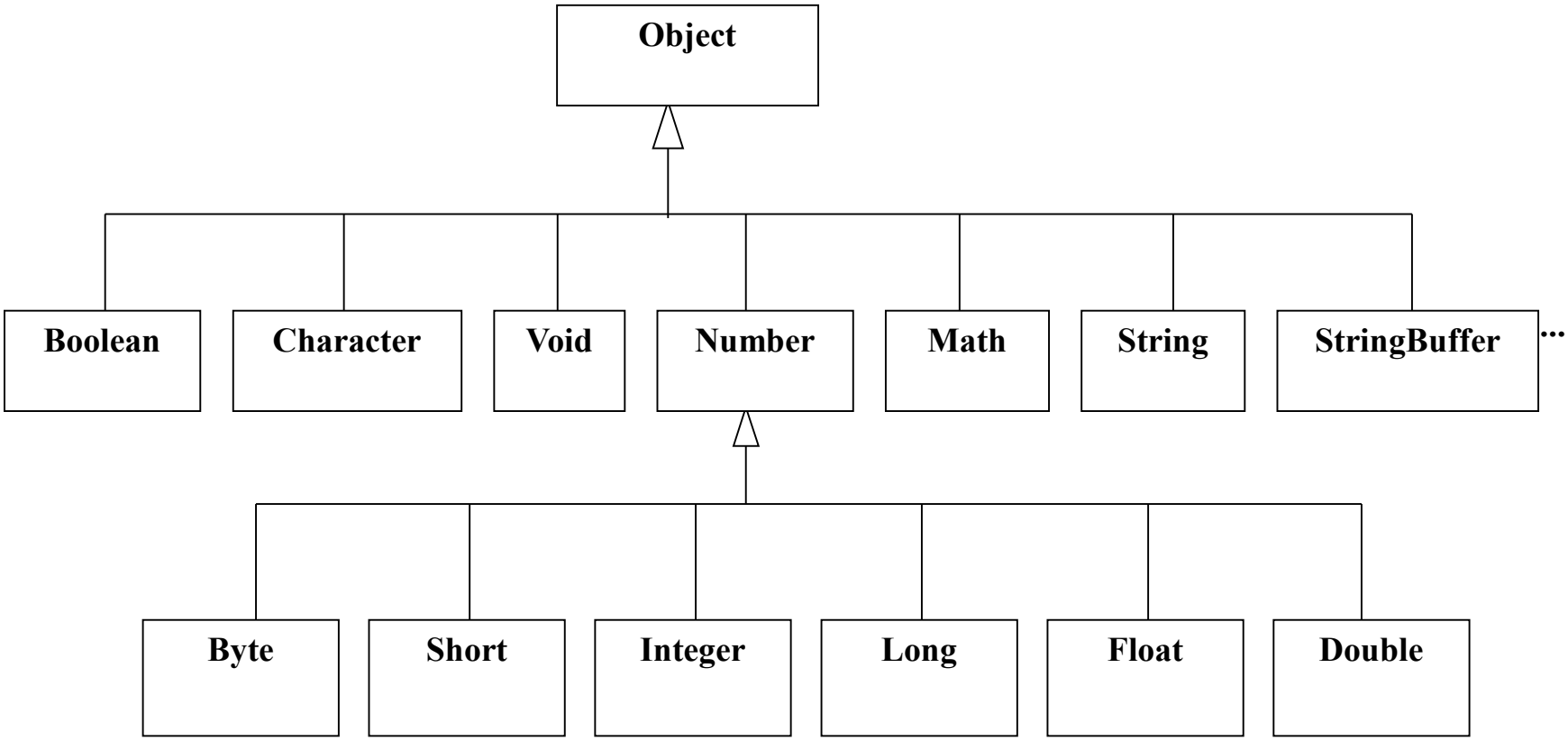
Cấu trúc dữ liệu trong Java

Bộ môn công nghệ phần mềm
Khoa công nghệ thông tin
Trường ĐHCN, ĐHQG Hà Nội

Nội dung

- Kiểu dữ liệu nguyên thủy
- String
- Math class
- Arrays
- Tập hợp (collection)
  - Lists
  - ArrayLists
  - Stacks
  - Queues
- HashMap
- Tổng kết

Các lớp cơ bản



Lớp Object

- Phương thức getClass() trả lại tên lớp của đối tượng hiện tại

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

- boolean equals(Object) so sánh các đối tượng, thường là được tái định nghĩa
- String toString() trả lại biểu diễn của đối tượng dưới dạng xâu ký tự, thường được định nghĩa lại

Wrapper classes

- 'Lớp bao bọc' là lớp gói quanh một kiểu dữ liệu nguyên thủy sao cho trông như một đối tượng
- Wrapper classes có phương thức có thể mở đối tượng ra và trả về kiểu ban đầu

- Ví dụ

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

- Mở đối tượng
  - int z= o.intValue()
  - System.out.println(z*z);// 10000

Kiểu nguyên thủy

- Các phương thức tiện ích
  - **valueOf(String s)** trả lại một đối tượng có kiểu tương ứng giá trị lưu trong biến xâu ký tự

    Integer k = Integer.valueOf("12"); // k = 12
  - **typeValue()** trả lại giá trị primitive value of the object

    int i = k.intValue(); // i = 12
  - **static parseType(String s)** chuyển xâu ký tự thành giá trị của kiểu nguyên thủy tương ứng

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

Lớp các ký tự

- Các phương thức
  - 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)

Lớp String

- **String**: xâu ký tự không thể sửa đổi được (vd. chiều dài xâu)
  - Tất cả những thay đổi đối với đối tượng String đều tạo một đối tượng mới
- Hàm tạo
  - **String(String)**
  - **String(StringBuffer)**
  - **String(byte[])**
  - **String(char[])**
- Các phương thức
  - **int length()** cho biết chiều dài của xâu
  - **char charAt(int index)** trả lại ký tự ở vị trí index

Lớp String

- So sánh
  - **boolean equals(String)**
  - **boolean equalsIgnoreCase(String)**
  - **boolean startWith(String)**
  - **boolean endWith(String)**
  - **int compareTo(String)**
- Chuyển đổi giữa ký tự hoa và thường
  - **String toUpperCase()**
  - **String toLowerCase()**
- Kết nối xâu
  - **String concat(String)**
  - operator "+"

Lớp String....

- Tìm kiếm về phía trước
  - **int indexOf(int ch)**
  - **int indexOf(int ch, int from)**
  - **int indexOf(String s)**
  - **int indexOf(String s, int from)**

- Tìm kiếm giật lùi
  - **int lastIndexOf(int ch)**
  - **int lastIndexOf(int ch, int from)**
  - **int lastIndexOf(String)**
  - **int lastIndexOf(String, int)**

Lớp String...

- 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 ký cắt xén những khoảng trắng ở đầu và cuối
  - **String substring(int startIndex)**
  - **String substring(int start, int end)**

StringBuffer

StringBuffer: xâu ký tự có thể thay đổi được

- Hàm tạo
  - **StringBuffer(String)**
  - **StringBuffer(int length)**
  - **StringBuffer():** kích cỡ mặc định là 16

- Các hàm tiện ích
  - **int length()**
  - **void setLength()**
  - **char charAt(int index)**
  - **void setCharAt(int index, char ch)**
  - **String toString()**

StringBuffer

- Soạn thảo
  - **append(String)**
  - **append(type t)** thêm vào biểu diễn xâu của t
  - **insert(int offset, String s)**
  - **insert(int offset, char[] chs)**
  - **insert(int offset, type t)**
  - **delete(int start, int end)** xoa xâu con
  - **delete(int index)** xóa một ký tự
  - **reverse()**

Lớp Math

- Constants
  - **Math.E**
  - **Math.PI**
- Các phương thức tĩnh
  - **type abs(type):** trả lại giá trị tuyệt đối của int/double/long
  - **double ceil(double)**
  - **double floor(double)**
  - **int round(float)**
  - **long round(double)**
  - **type max(type, type), type min(type, type)**

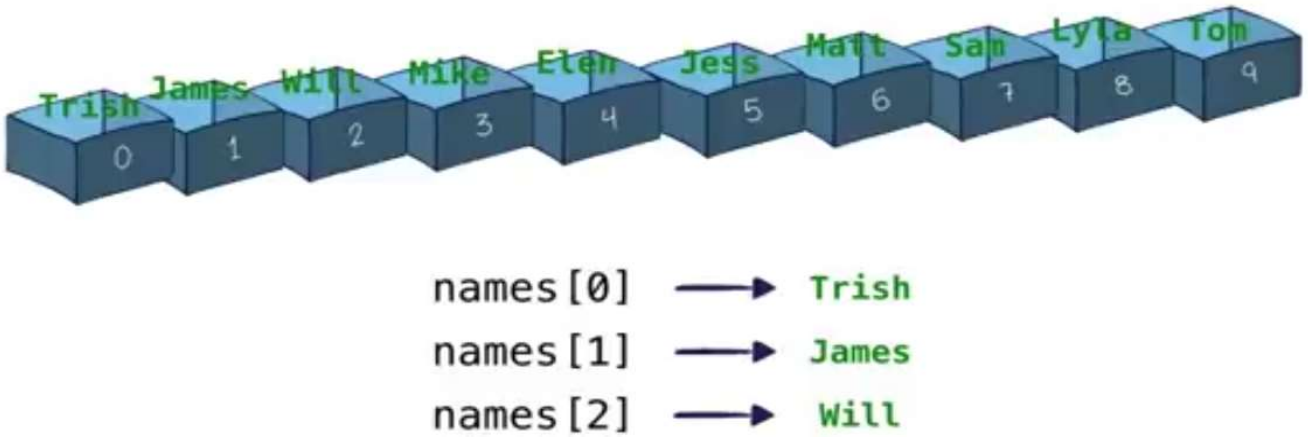
Lớp Math …

- Các phương thức static (tiếp)
  - **`double random()` sinh ngẫu nhiên giá trị trong [0.0,1.0]**
  - **`double pow(double, double)`**
  - **`double exp(double)`** *tính e mũ*
  - **`double log(double)`** log cơ số e
  - **`double sqrt(double)`**

- Hàm lượng giác
  - **`double sin(double)` trả lại giá trị sin của góc**
  - **`double cos(double)`**
  - **`double tan(double)`**

Mảng

- Để lưu nhiều phần tử có cùng kiểu ta có thể sử dụng mảng
  - cả một mảng được xử lý như một biến đơn
- Mỗi phần tử trong mảng có một chỉ mục bắt đầu từ 0 và cho phép truy cập dữ liệu thông qua chỉ mục đó



Mảng…

- Mảng là một đối tượng phải được tao ra sử dụng từ khoa **new**

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

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

Mảng sử dụng làm đối của hàm và trả lại giá trị

```
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

```
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];
```

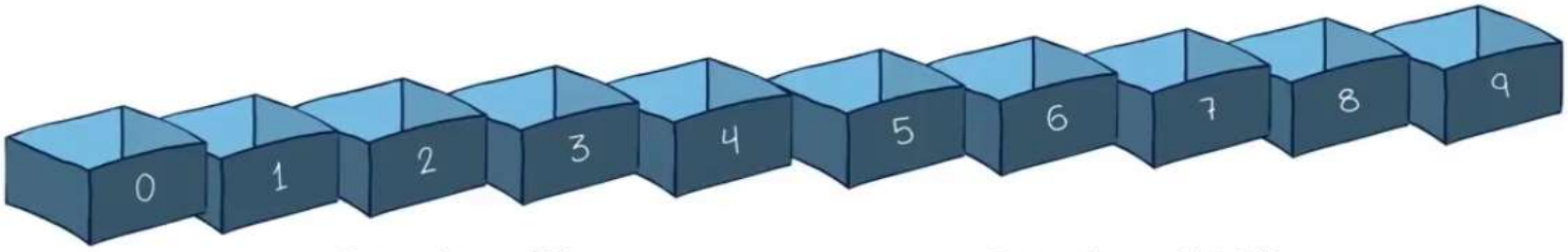
Copy mảng

- **`System.arraycopy(src, s_off, des, d_off, len)`**
  - **`src`**: mảng nguồn, **`s_off`**: offset của mảng nguồn
  - **`des`**: mảng đích, **`d_off`**: offset của mảng đích
  - **`len`**: chiều dài của các phần tử được copy
- Nội dung của các phần tử được copy
  - Giá trị nguyên thủy
  - Tham chiếu đối tượng

Lớp Array

- Trong gói **`java.util`**
- 4 phương thức static
  - **`fill()`** khởi tạo các phần tử với cùng giá trị
  - **`sort()`** sắp xếp mảng
    - Làm việc với mảng các giá trị nguyên thủy
    - Làm việc với các lớp và cài đặt interface **`Comparable`**
  - **`equals()`** so sánh 2 mảng
  - **`binarySearch()`** tìm kiếm nhị phân trong một mảng được sắp xếp, sẽ báo lỗi nếu mảng chưa được săp xếp

Hạn chế của mảng

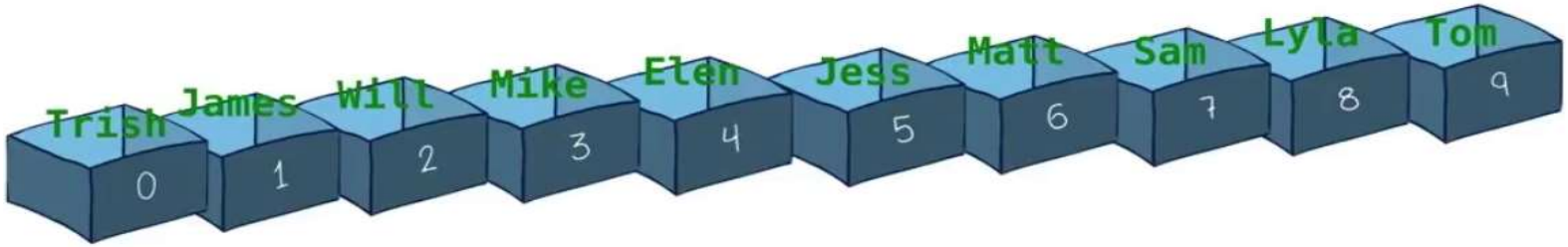



```
String[] names = new String[10];
```

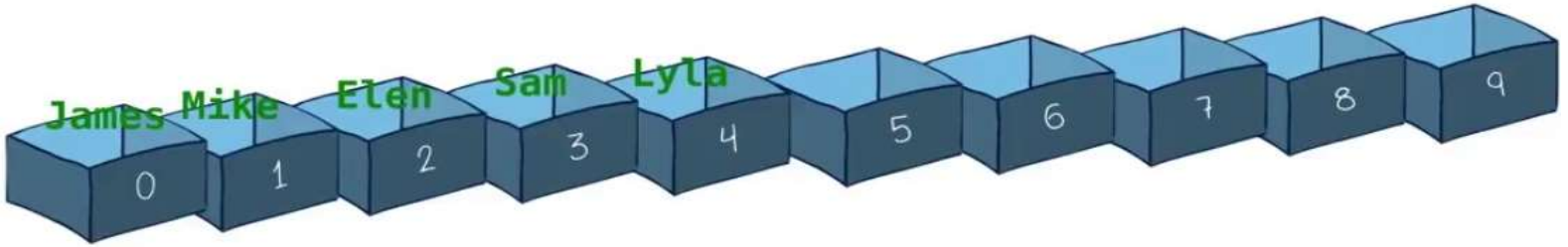
- Cần biết chính xác số phần tử sẽ dùng khi khởi tạo mảng (trước cả khi sử dụng mạng đó)
- Một khi đã khởi tạo mảng với số lượng xác định sẽ không được phép thêm hay xóa phần tử đi
  - Vd. có lỗi khi chạy chương trình: names[100]= "Mike";

Hạn chế…

- Khi đã gán giá trị cho các phần tử rồi sẽ không thể đưa giá trị mới mà không phải thay thế giá trị cũ

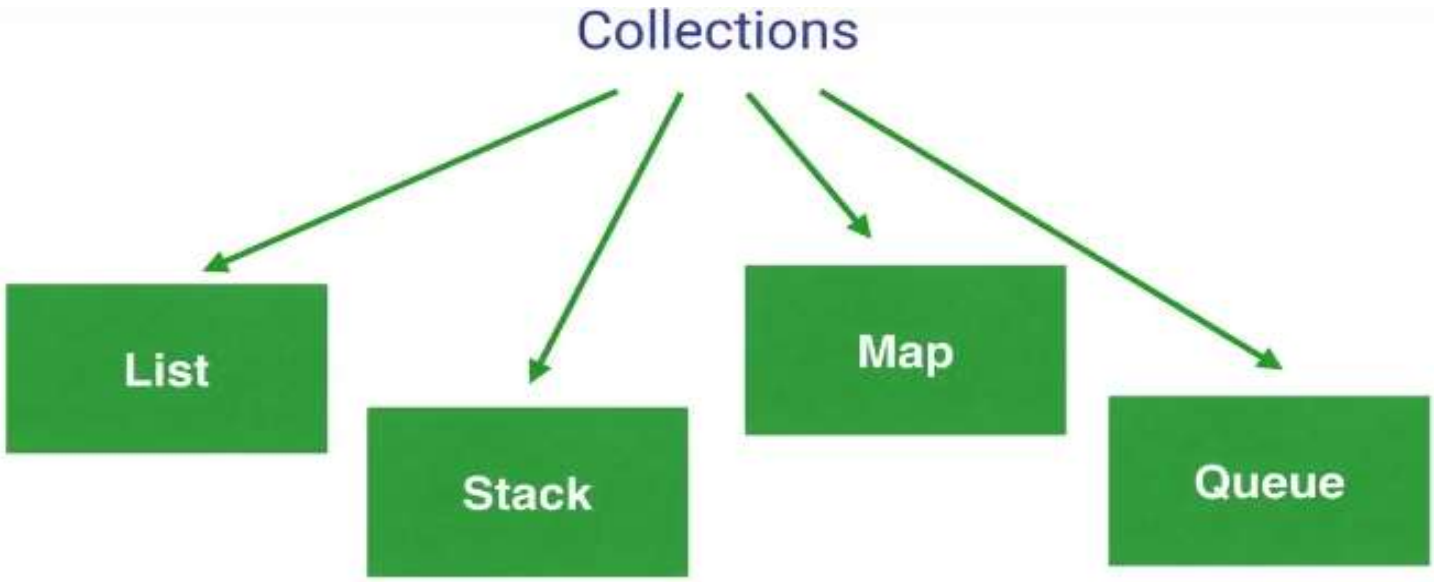


- Khi xóa giá trị của phần tử đi sẽ tao ra các lỗ hổng trong mảng, cần phải thực hiện dịch chuyển thủ công để lấp đầy chỗ trống

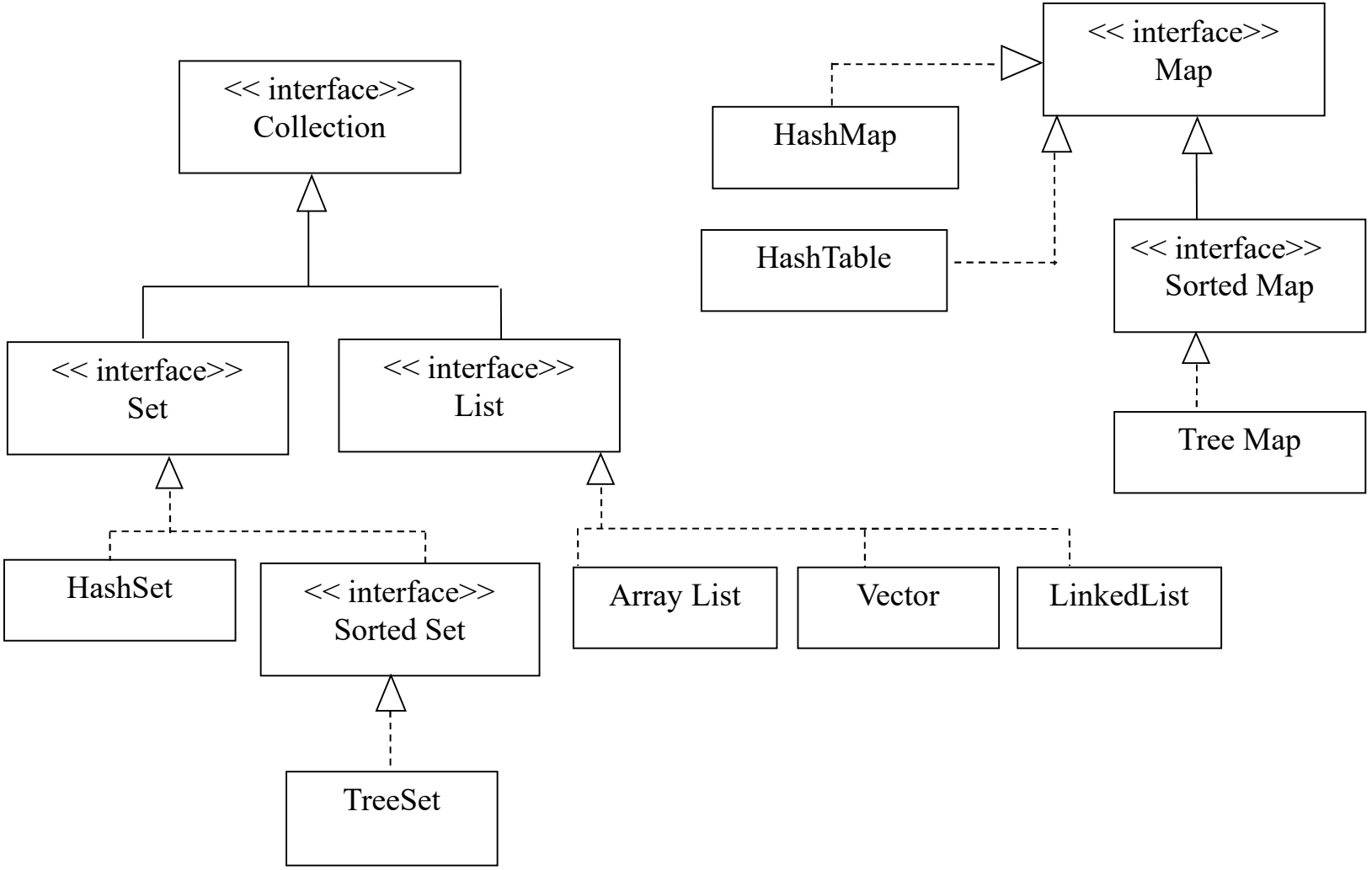


Tập hợp (collections)

- Bao gồm một số các lớp và giao diện mà Java cung cấp để bạn xử lý nhiều phần tử cùng kiểu đơn giản hơn



Phả hệ tập hợp trong Java



Danh sách (List)

- **List** trong Java là một **interface** hoạt động tương tự như mảng
  - Là một tập có thứ tự (xem như chuỗn tuần tự)
  - 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
  - và tìm kiếm phần tử trong mảng bằng vòng lặp

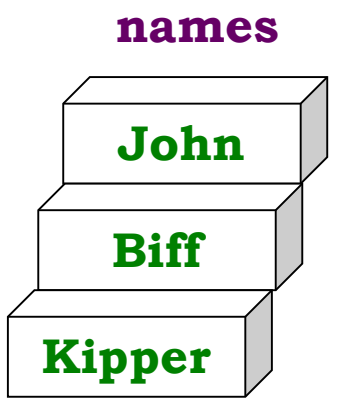
- ArrayList là một lớp cài đặt giao diện **List**
  - **Nó có một mảng ở bên trong**

ArrayList

- Đơn giản là gói bao bọc bên ngoài một mảng
- Cho phép bạn khởi tạo một biến tập hợp xác định số lượng phần tử mà bạn cần
  - ArrayList **names**= **new** ArrayList();
  - Mảng **names** được tạo ra

❖ Sử dụng phương thức add() của lớp ArrayList để tiếp tục đưa phần tử vào mà không cần chỉ mục hoặc cài đặt mảng như thế nào

- Vd: names.add("**John**"); names.add("**Biff**"); names.add("**Kipper**");



❖ Có thể dùng remove() để xóa phần tử và tự đồng dịch chuyển cũng như điều chỉnh chỉ mục

- Vd: names.remove("**Biff**");

ArrayList…

- Xem đường link để biết thêm chi tiết
- Vd.,
  - **add(E element**): thêm phần tử vào cuối danh sách
  - **add(int index, E element):** thêm phần tử vào vị trí xác định
  - **get(int index)**: trả lại phần tử ở vị trí xác định
  - **contains(Object o):** tra giá trị true nếu danh sách chứa đối tượng xác định
  - **remove(int index)**: xóa phần tử ở vị trí xác định
  - **size():** cho biết số phẩn tử trong danh sách
  - **clear()**: xóa sạch danh sách

Duyệt danh sách bằng vòng lặp

- Giống với mảng, cách tốt nhất để truy cập phần tử trong một ArrayList là dùng vòng lặp và biến đếm

```
int size =list.size();
for(int i=0; i<size; i++){
    System.out.prinltn(list.get(i));
}
```

Vòng lặp...

- Phương thức indexOf(Object o)
  - trả về chỉ mục của lần xuất hiện đầu tiên của một phần tử xác định trong danh sách
  - hoặc -1 nếu phần tử đó không có trong danh sách
- Vì thế, khi ta muốn tìm kiếm xem một đối tượng trong danh sách thay vì dùng vòng lặp, có thể sử dụng
  - Vd. list.indexOf("Biff");
  - nếu trả về -1 tức là Biff không có trong danh sách

- Xem thêm link về ArrayList class

Ngăn xếp (Stacks)

- 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()**
  - **search(Object o):** tìm kiếm và trả lại vị trí của đối tượng o

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);
String moreNews = (String) newsFeed.pop(); System.out.println(moreNews);
```

Kết quả?

Giao diện Queue

- là một kiểu tập hợp trong java
- không giống Stack, hàng đợi (Queue) nó hoạt động theo cơ chế vào trước ra trước (First In First Out: FIFO)

- Queue chỉ là một **interface** không phải lớp, nhưng nó định nghĩa 2 phương thức quan trọng cho tất cả các lớp cài đặt giao diện này
  - **add(E element):** thêm một phần tử vào hàng đợi
  - **poll():** truy cập và xóa phần tử ở đầu của hàng đợi

Lớp Deque

- là kiểu đặc biệt của hàng đợi, nó có 2 đầu
  - Bạn 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

LinkedList

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

```
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());
```

Kết quả là gì?

Tối ưu

- Sử dụng cấu trúc dữ liệu khác nhau mang lại hiệu suất khác nhau
- Một chương trình có thể có nhiều cách cài đặt khác nhau nhưng chỉ một vài chạy suôn sẻ và đủ nhanh theo mong muốn của người dùng

Hashmaps

- **Hashmap** là một kiểu tập hợp dùng với mục đích làm tăng tốc quy trình tìm kiếm trong **ArrayList**
- Có thể hiểu là một tập hợp các đối tượng (*String, Integers, hay bất cứ kiểu đối tượng nào khác*),
  - nhưng chúng được lưu sao cho xác định duy nhất
- Hashmaps cho phép bạn lưu một **key** đối với mọi phần tử mà bạn thêm vào
  - **key** là duy nhất trong danh sách
  - khá giống chỉ mục của một mảng, ngoại trừ khóa có thể là một đối tượng thuộc kiểu nào đó
- Điểm mấu chốt là có thể tìm một đối tượng ngay tức thì mà không cần phải sử dụng vòng lặp → tiết kiệm đáng kể thời gian chạy

Ví dụ

- Một lớp Book lưu chi tiết về sách

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

Nếu dùng ArrayList

- Tạo một lớp Library lưu sách
  - có thể dùng **ArrayList**

```
public class Library{
    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;
}
```

Sử dụng HashMap

- Cần import

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

- KHai báo lớp

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

  - khóa có kiểu String

Sử dụng HashMap…

- Khởi tạo dùng default constructor
  - allBooks = **new** HashMap<String, Book>();

- Thêm phần tử vào HashMap
  - 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
        return book;
}
```

Lớp Iterator

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

```
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!
```

Tổng kết

- Wrapper classes 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