Pre-OOP Course

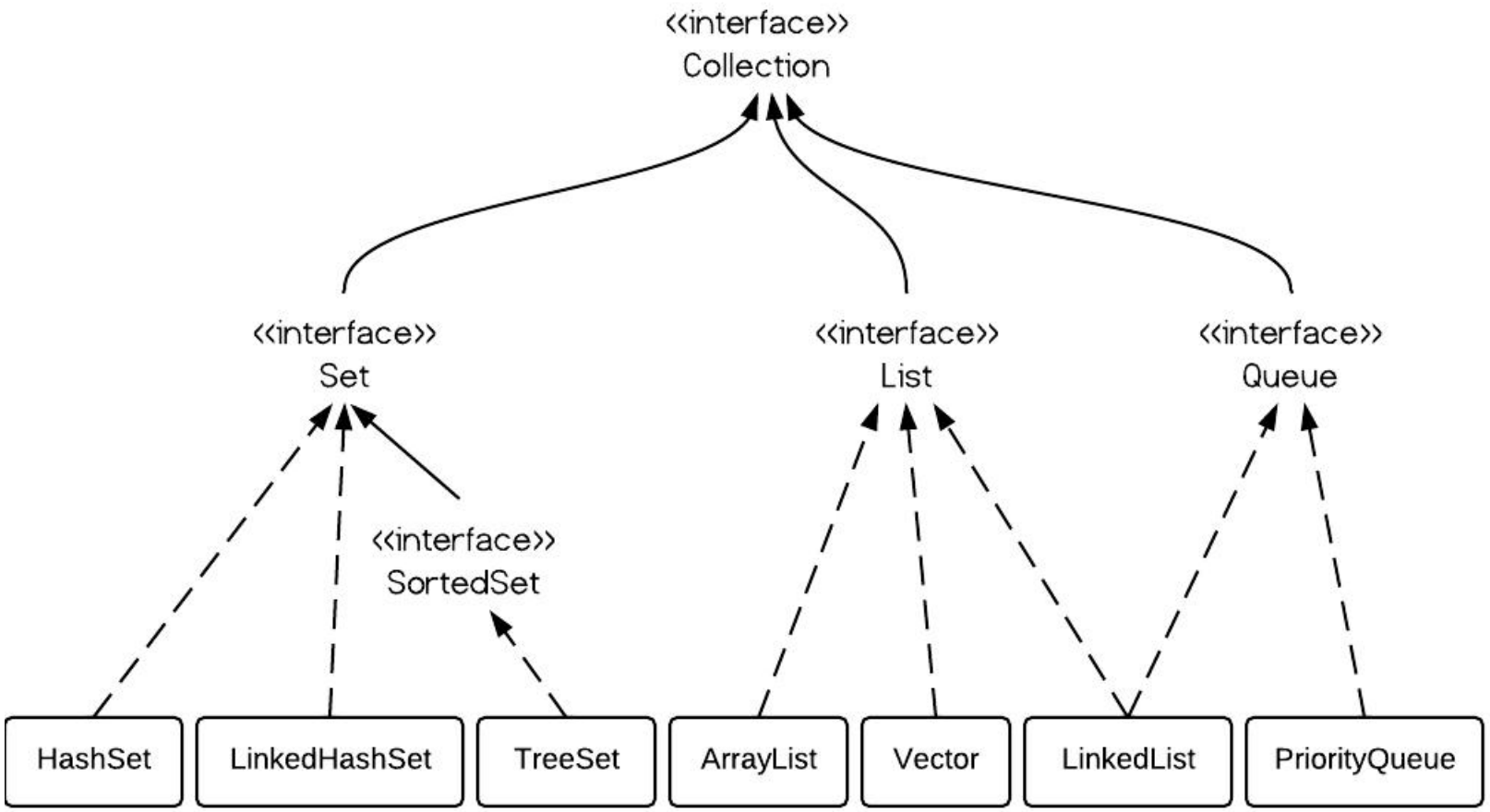
# Giới thiệu về Java (cont.)

**Lecturer**: Dr. Nguyễn Minh Hải
(Ho Chi Minh University of Technology)

# Collections

- Collection/container
  - object that groups multiple elements
  - used to store, retrieve, manipulate, communicate aggregate data
- Iterator - object used for traversing a collection and selectively remove elements
- Generics – implementation is parametric in the type of elements

# General Purpose Implementations




```
List<String> list1 = new ArrayList<String>(c);
List<String> list2 = new LinkedList<String>(c);
```

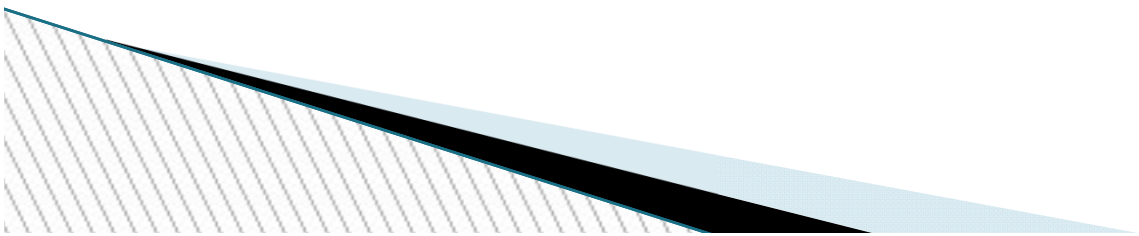
# Java Collection Framework

- Goal: Implement reusable data-structures and functionality

- Collection interfaces - manipulate collections independently of representation details
- Collection implementations - reusable data structures

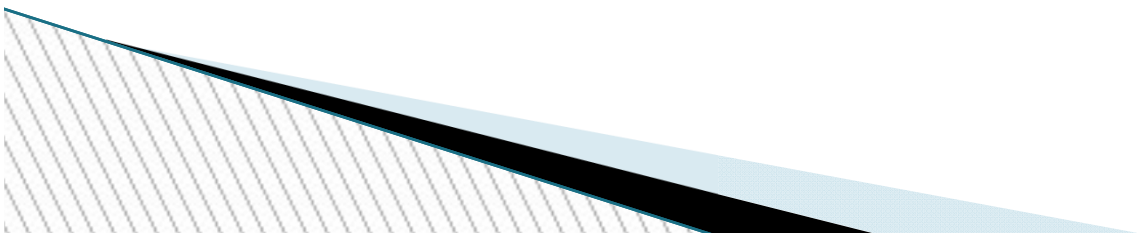
```
List<String> list = new ArrayList<String>(c);
```

- Algorithms - reusable functionality
  - computations on objects that implement collection interfaces
  - e.g., searching, sorting
  - polymorphic: the same method can be used on many different implementations of the appropriate collection interface



# Problem

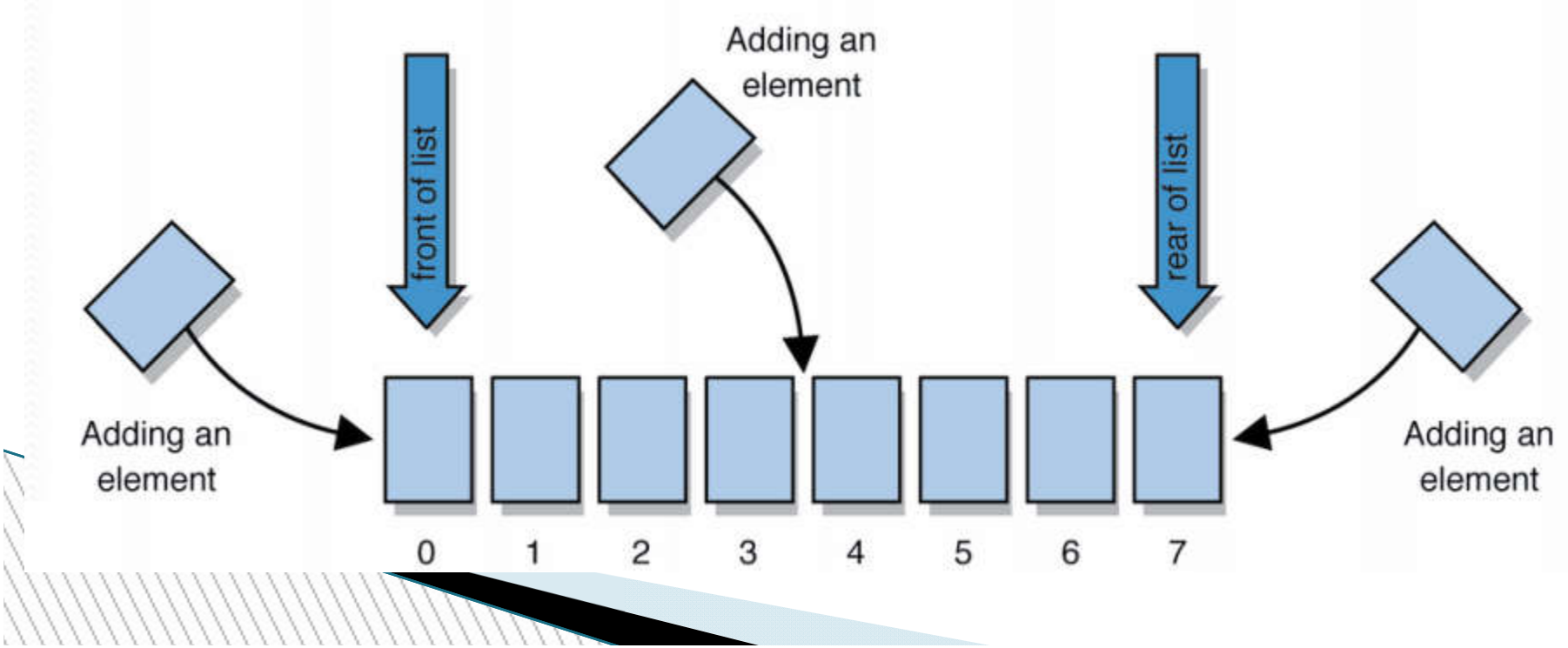
```
String[] allWords = new String[1000];
int wordCount = 0;

Scanner input = new Scanner(new File("data.txt"));
while (input.hasNext()) {
    String word = input.next();
    allWords[wordCount] = word;
    wordCount++;
}
```



# ArrayList

A collection storing an ordered sequence of elements

- Each element is accessible by a 0-based **index**
- a list has a **size** (number of elements that have been added)
- elements can be added to the front, back, or elsewhere



# ArrayList Method

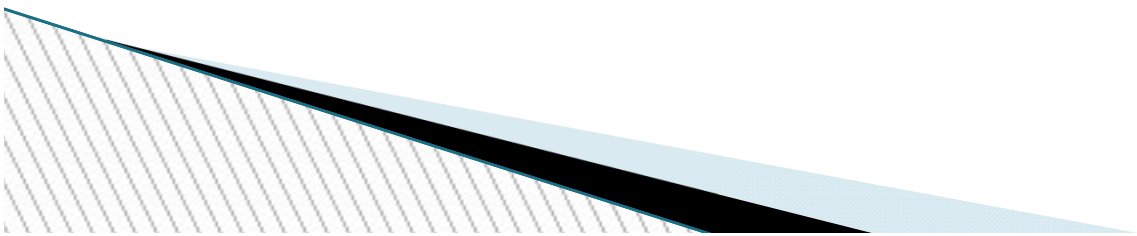
| | |
|---|---|
| `add(`**value**`)` | appends value at end of list |
| `add(`**index**`, `**value**`)` | inserts given value just before the given index, shifting subsequent values to the right |
| `clear()` | removes all elements of the list |
| `indexOf(`**value**`)` | returns first index where given value is found in list (-1 if not found) |
| `get(`**index**`)` | returns the value at given index |
| `remove(`**index**`)` | removes/returns value at given index, shifting subsequent values to the left |
| `set(`**index**`, `**value**`)` | replaces value at given index with given value |
| `size()` | returns the number of elements in list |
| `toString()` | returns a string representation of the list such as `"[3, 42, -7, 15]"` |

# Type Parameters (Generics)

ArrayList<**Type**> **name** = new ArrayList<**Type**>();

This is called a *type parameter* or a *generic* class.
Allows the same ArrayList class to store lists of different types

ArrayList**<String>** names = new ArrayList**<String>**();
names.add("Marty Stepp");
names.add("Stuart Reges");

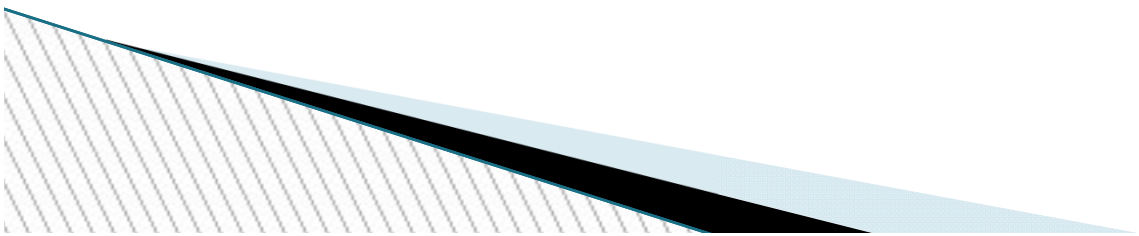


# ArrayList vs. array

- construction
  String[] names = new String[5];
  **ArrayList<String> list = new ArrayList<String>();**

- storing a value
  names[0] = "Jessica";
  **list.add("Jessica");**

- retrieving a value
  String s = names[0];
  **String s = list.get(0);**

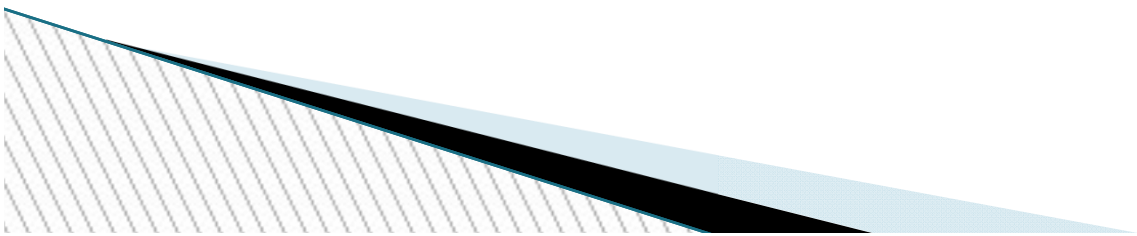


# ArrayList vs. array (cont.)

- construction
  String[] names = new String[5];
  **ArrayList<String> list = new ArrayList<String>();**

- storing a value
  names[0] = "Jessica";
  **list.add("Jessica");**

- retrieving a value
  String s = names[0];
  **String s = list.get(0);**



# ArrayList vs. array

- Doing something to each value that starts with "B"

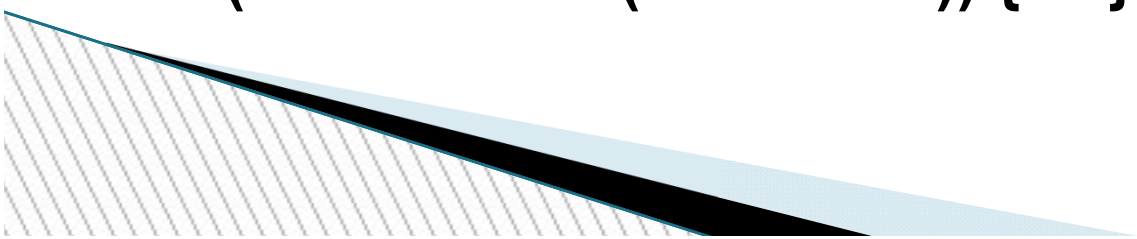
```
for (int i = 0; i < names.length; i++) {
if (names[i].startsWith("B")) { ... }
}
```

```
for (int i = 0; i < list.size(); i++) {
if (list.get(i).startsWith("B")) { ... }
}
```

- Seeing whether the value "Benson" is found

```
for (int i = 0; i < names.length; i++) {
if (names[i].equals("Benson")) { ... }
}
```

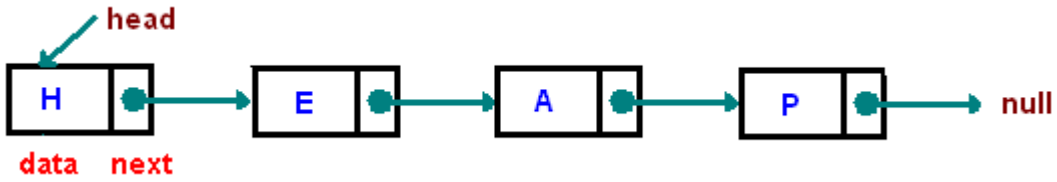
```
if (list.contains("Benson")) { ... }
```



# Linked Lists

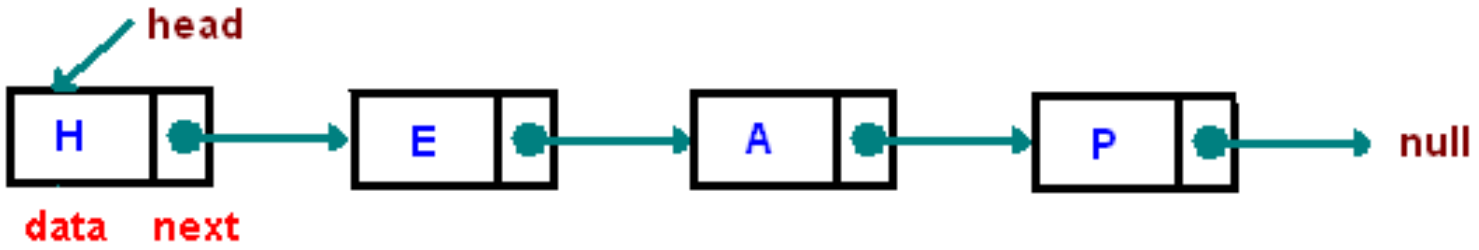
- Arrays are expensive to maintain new insertions and deletions

- A linked list is a linear data structure where each element is a separate object



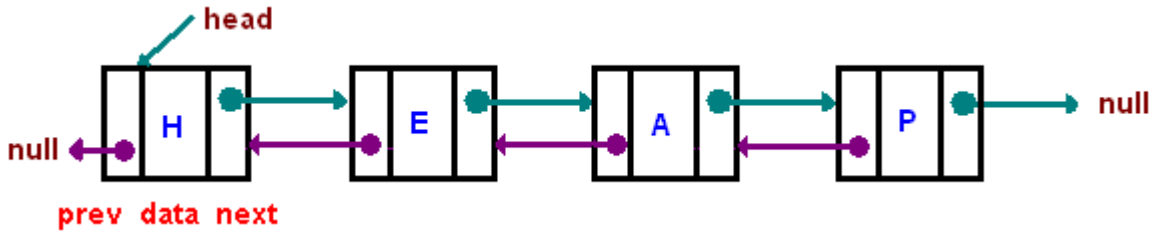
# Linked Lists

- A linked list is a dynamic data structure. The number of nodes in a list is not fixed and can grow and shrink on demand.



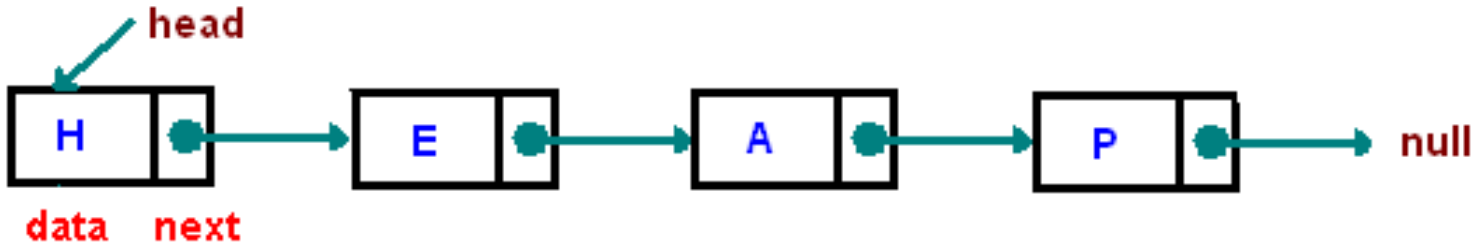
# Linked Lists

- A **singly linked list** is described above
- A **doubly linked list** is a list that has two references, one to the next node and another to previous node.



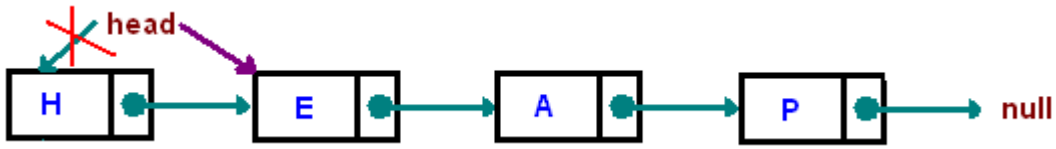
# Linked Lists

```
private static class Node<AnyType>
{
   private AnyType data;
   private Node<AnyType> next;

   public Node(AnyType data, Node<AnyType> next)
   {
      this.data = data;
      this.next = next;
   }
}
```

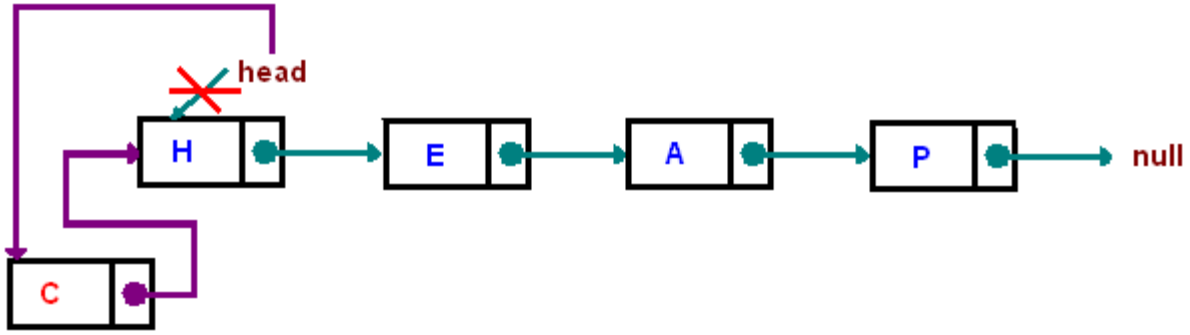


# Linked Lists

```
public class LinkedList<AnyType> implements Iterable<AnyType>
{
   private Node<AnyType> head;

  /**
    *  Constructs an empty list
    */
   public LinkedList()
   {
      head = null;
   }
  /**
    *  Returns true if the list is empty
    *
    */
   public boolean isEmpty()
   {
      return head == null;
   }
```



# Linked Lis



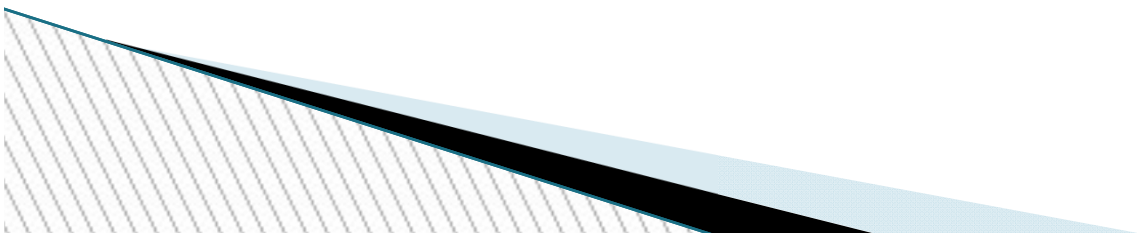

```
/**
  *  Inserts a new node
at the beginning of this list.
  *
  */
  public void addFirst(AnyType item)
  {
     head = new Node<AnyType>(item, head);
  }
/**
  *  Returns the first element in the list.
  *
  */
  public AnyType getFirst()
  {
     if(head == null) throw new NoSuchElementException();

     return head.data;
  }
```

# Linked Lists

```
/**
  *  Removes the first element in the list.
  *
  */
  public AnyType removeFirst()
  {
     AnyType tmp = getFirst();
     head = head.next;
     return tmp;
  }
/**
  *  Inserts a new node to the end of this list.
  *
  */
  public void addLast(AnyType item)
  {
     if( head == null)
        addFirst(item);
     else
     {
        Node<AnyType> tmp = head;
        while(tmp.next != null) tmp = tmp.next;

        tmp.next = new Node<AnyType>(item, null);
     }
  }
```

# Performance of ArrayList vs. LinkedList

| | ArrayList | LinkedList |
|---|---|---|
| get() | O(1) | O(n) |
| add() | O(1) | O(1) amortized |
| remove() | O(n) | O(n) |



# final

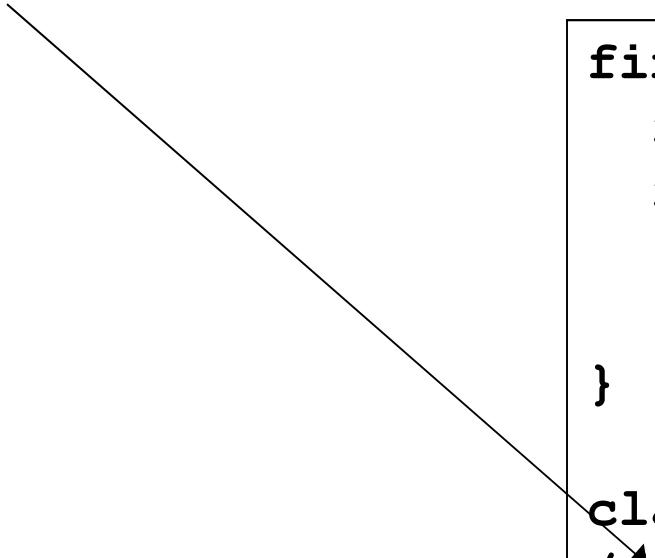
- ***final* member data**
  Constant member
- ***final* member function**
  The method can't be overridden.
- ***final* class**
  'Base' is final, thus it can't be extended

(String class is final)

```
final class Base {
  final int i=5;
  final void foo() {
     i=10;
//what will the compiler say
about this?
   }
}

class Derived extends Base {
// Error
   // another foo ...
   void foo() {

   }
}
```

# final

```
Derived.java:6: Can't subclass final classes: class Base
class class Derived extends Base {
      ^
1 error
```

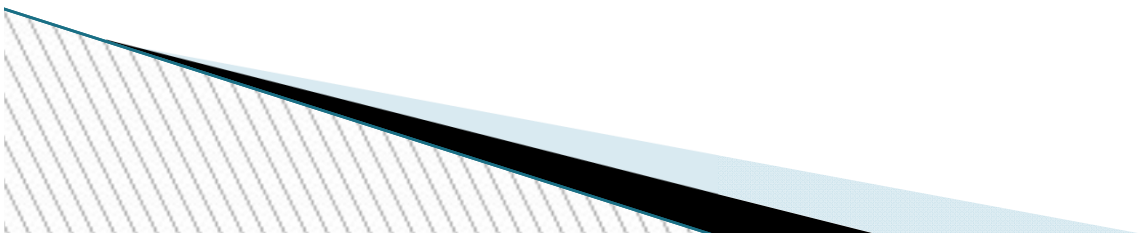


```
final class Base {
  final int i=5;
  final void foo() {
    i=10;
  }
}

class Derived extends Base {
// Error
  // another foo ...
  void foo() {

  }
}
```

# Exception - What is it and why do I care?

**Definition:** An *exception* is an event that occurs during the execution of a program that disrupts the normal flow of instructions.

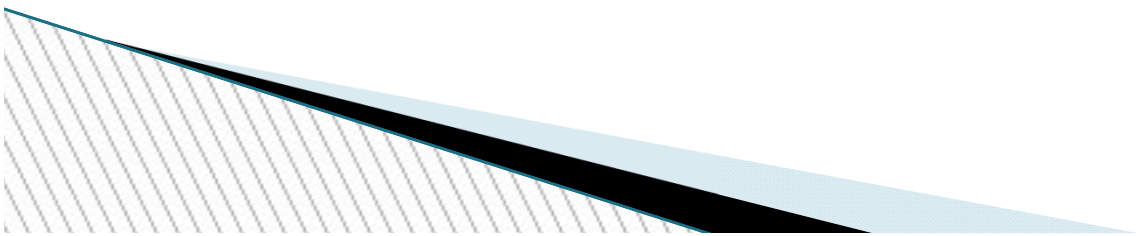
- Exception is an Object
- Exception class must be descendent of Throwable.



# Exception - What is it and why do I care?(2)

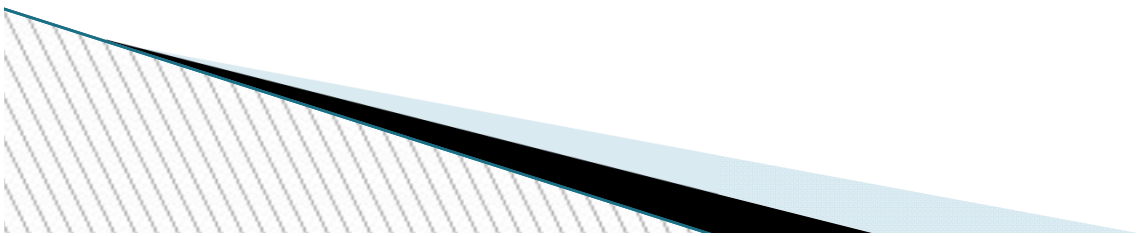
By using exceptions to manage errors, Java programs have the following advantages over traditional error management techniques:

**1**: Separating Error Handling Code from "Regular" Code
**2**: Propagating Errors Up the Call Stack
**3**: Grouping Error Types and Error Differentiation



# 1: Separating Error Handling Code from "Regular" Code (1)

```
readFile {
    open the file;
    determine its size;
    allocate that much memory;
    read the file into memory;
    close the file;
}
```



# 1: Separating Error Handling Code from "Regular" Code (2)

```
errorCodeType readFile {
    initialize errorCode = 0;
    open the file;
    if (theFileIsOpen) {
        determine the length of the file;
        if (gotTheFileLength) {
            allocate that much memory;
            if (gotEnoughMemory) {
                read the file into memory;
                if (readFailed) {
                    errorCode = -1;
                }
            } else {
                errorCode = -2;
            }
        } else {
            errorCode = -3;
        }
        close the file;
        if (theFileDidntClose && errorCode == 0) {
            errorCode = -4;
        } else {
            errorCode = errorCode and -4;
        }
    } else {
        errorCode = -5;
    }
    return errorCode;
}
```

# 1: Separating Error Handling Code from "Regular" Code (3)

```
readFile {
    try {
        open the file;
        determine its size;
        allocate that much memory;
        read the file into memory;
        close the file;
    } catch (fileOpenFailed) {
        doSomething;
    } catch (sizeDeterminationFailed) {
        doSomething;
    } catch (memoryAllocationFailed) {
        doSomething;
    } catch (readFailed) {
        doSomething;
    } catch (fileCloseFailed) {
        doSomething;
    }
}
```

# 2: Propagating Errors Up the Call Stack

```
method1 {
     try {
          call method2;
     } catch (exception) {
          doErrorProcessing;
     }
}
method2 throws exception {
     call method3;
}
method3 throws exception {
     call readFile;
}
```

***THANKS FOR LISTENING***