GEORGE MASON UNIVERSITY

CS 310 – Fall 2025

Data Structures

L05- More on ArrayList

Archange G. Destiné
adestine@gmu.edu

Outline Lecture 05

1. **Amortized Analysis**
2. **Recap from last Lecture (Dynamic Array)**
3. **Static Array List Implementation**
4. **Iterators**

5. **Notes:**
   - **Project 1**
   - **Always do the readings**

Amortized Analysis

Not quite the Average time complexity. (probabilistic approach)

What is Average Time Complexity? Let’s see a common example…

search(e) from an array of size n.

- Best case: ?
- Worst case: ?
- Average case: ? = n/2

$= 1/n + 2/n + 3/n + \ldots + n/n = (1/n)\ (1+2+3+\ldots+n)$

$= (1/n) * (n(n+1)/2) \qquad = (n+1)/2$

$\sim n/2$

Amortized Analysis

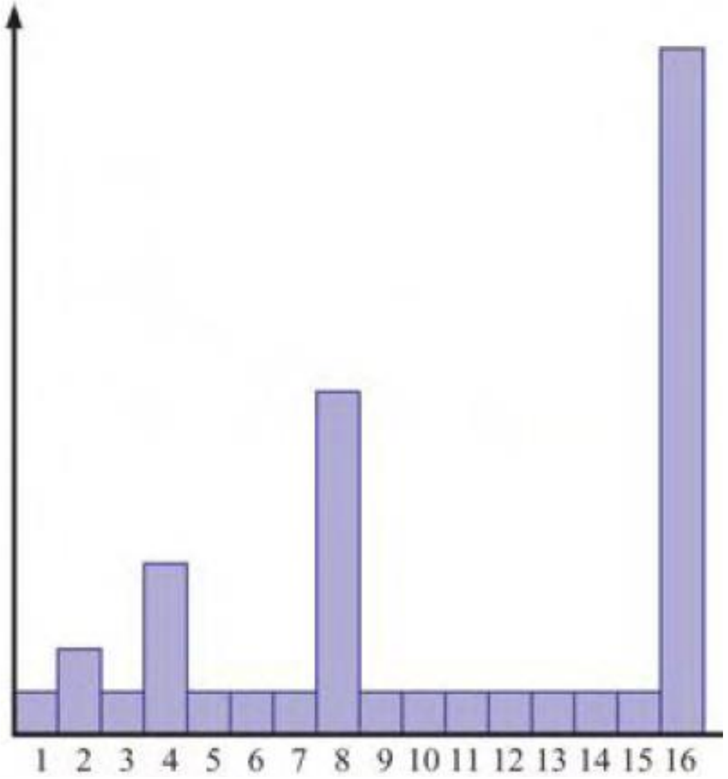
Not quite the Average time complexity. (probabilistic)

It is the total cost per operation over a sequence of operations.

What does Amortized O(1) means? We can use the **accounting** approach to understand the time complexity of add/end for the dynamic array.

Amortized Analysis

We can use the accounting approach to understand the time complexity of add/end for the dynamic array.



0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	…				…	

Amortized Analysis

We can use the accounting approach to understand the time complexity of add/end for the dynamic array.

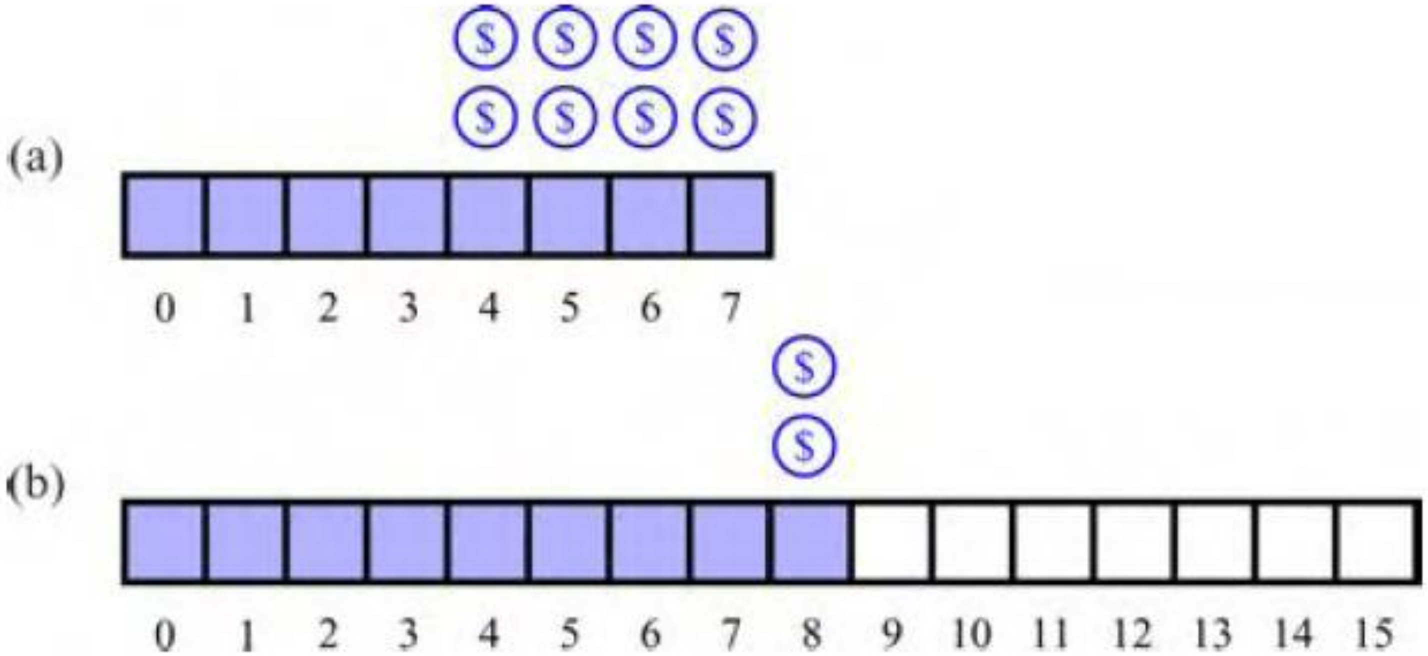
Amortization scheme: "Each operation is charged three cyber-dollars and all the computing time is paid for."

Let imagine this scenario:
-> 1$ for each push (excluding the copying operation)
-> And let's overcharge this push by 2 dollars. (saved for expensive operation)
-> at index 2^i when it is time to double the capacity, we have saved enough
-> 1 push cost 3 units in the long run… → O(1) Amortized complexity

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	…				…	

Amortized Analysis

We can use the accounting approach to understand the time complexity of add/end for the dynamic array.



Questions…

Exception (review from CS211)

Sometimes, exceptional events occur during execution of a program. For example:

- array index is out of bounds
- we divided by zero
- we tried to open a non-existing file
- many others…

Normal sequential control flow is aborted, in search of a way to **handle the exceptional event.**

- We keep escaping code blocks until one is found.
- **If we escape all the way out of main, the program crashes.**

You must handle them (whenever you can)

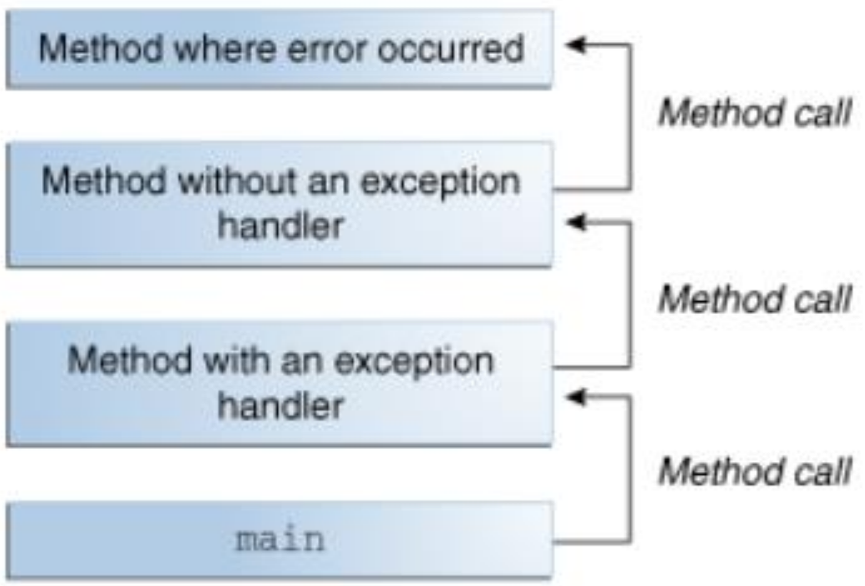
We have two options:

- **Catch It**: wrap the offending code in a **`try-catch`** block that catches the specific type of exception.

- **Defer It**: allow the exception to occur, propagating (*crashing*) its way through your program until it is caught elsewhere.
  - you might have to explicitly list what exceptions are deferred

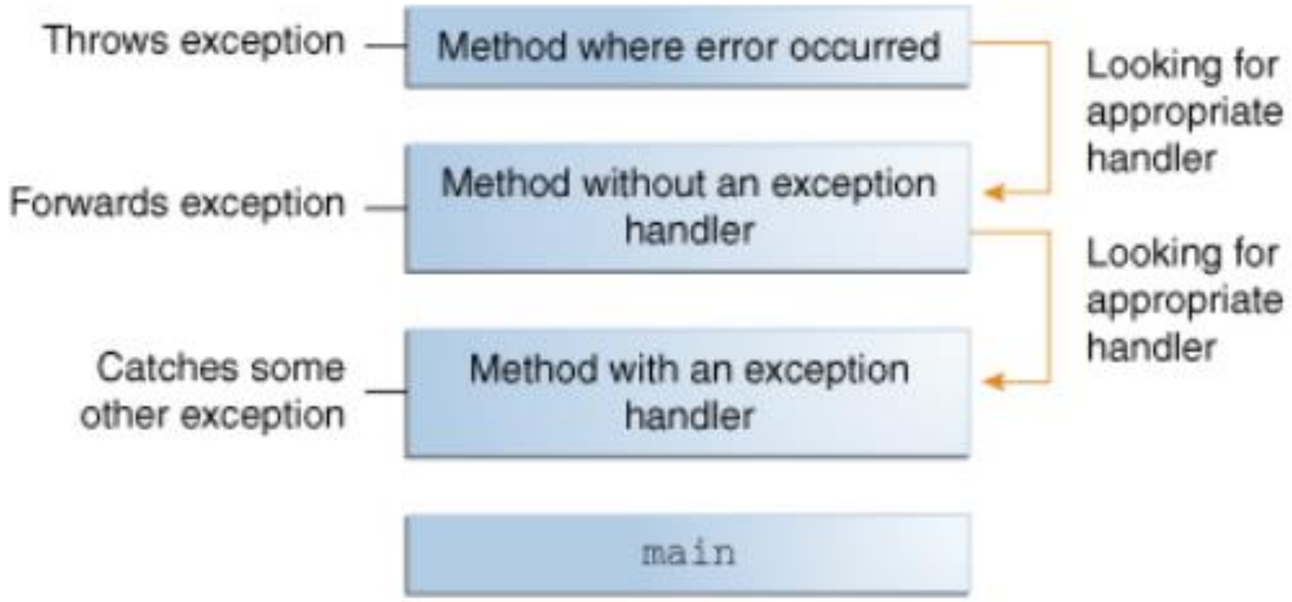
No matter what, the occurring exception immediately starts *crashing* your program by prematurely leaving each code block and method call, until it is caught by a catch-block (or the entire program is *crashed*).

Exception Propagation
What happens in the call stack?



Method where error occurred
Method call
Method without an exception handler
Method call
Method with an exception handler
Method call
main

The call stack.



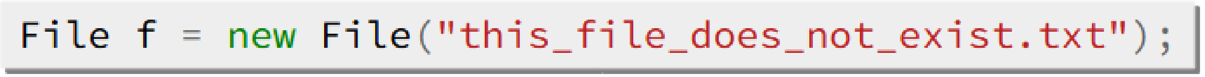
Throws exception
Method where error occurred
Looking for appropriate handler
Forwards exception
Method without an exception handler
Looking for appropriate handler
Catches some other exception
Method with an exception handler
main

How to create Exception?

- **automatic creation**

  Some Exception values are automatically created by an error caused during runtime. Examples:

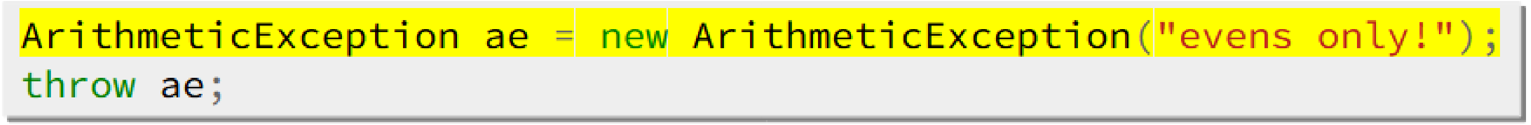
  - **dividing by zero** will cause an `ArithmeticException`
  - using a **wrong index** will cause an `ArrayIndexOutOfBoundsException`
  - opening a **file that doesn't exist** will cause a `FileNotFoundException`



```
File f = new File("this_file_does_not_exist.txt");
```

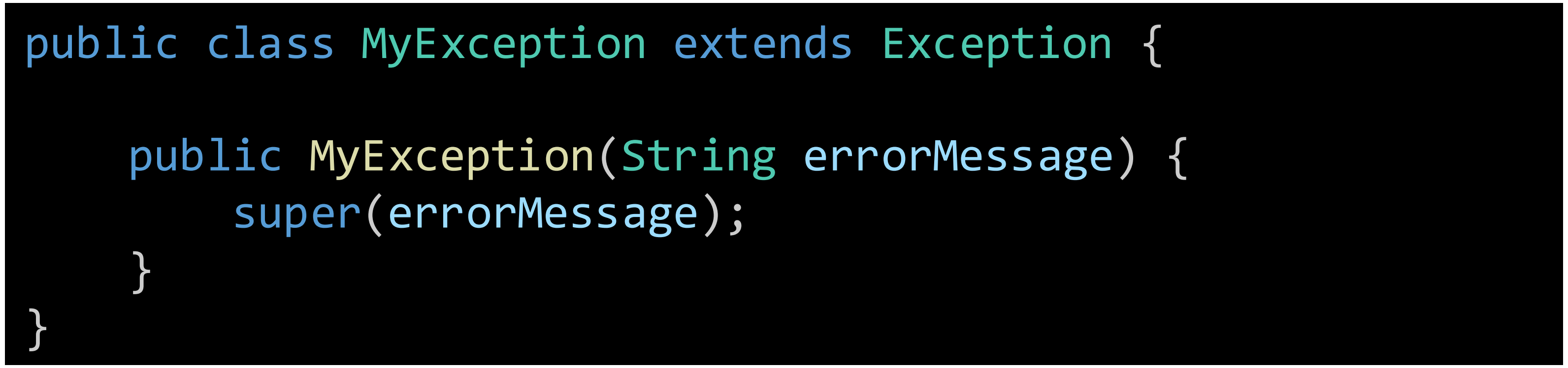
- **intentional creation**

  You can manually create an Exception value by instantiating an object of an Exception class and then 'throw' it:



```
ArithmeticException ae = new ArithmeticException("evens only!");
throw ae;
```

How to create your Own Exception?



```
public class MyException extends Exception {

    public MyException(String errorMessage) {
        super(errorMessage);
    }
}
```

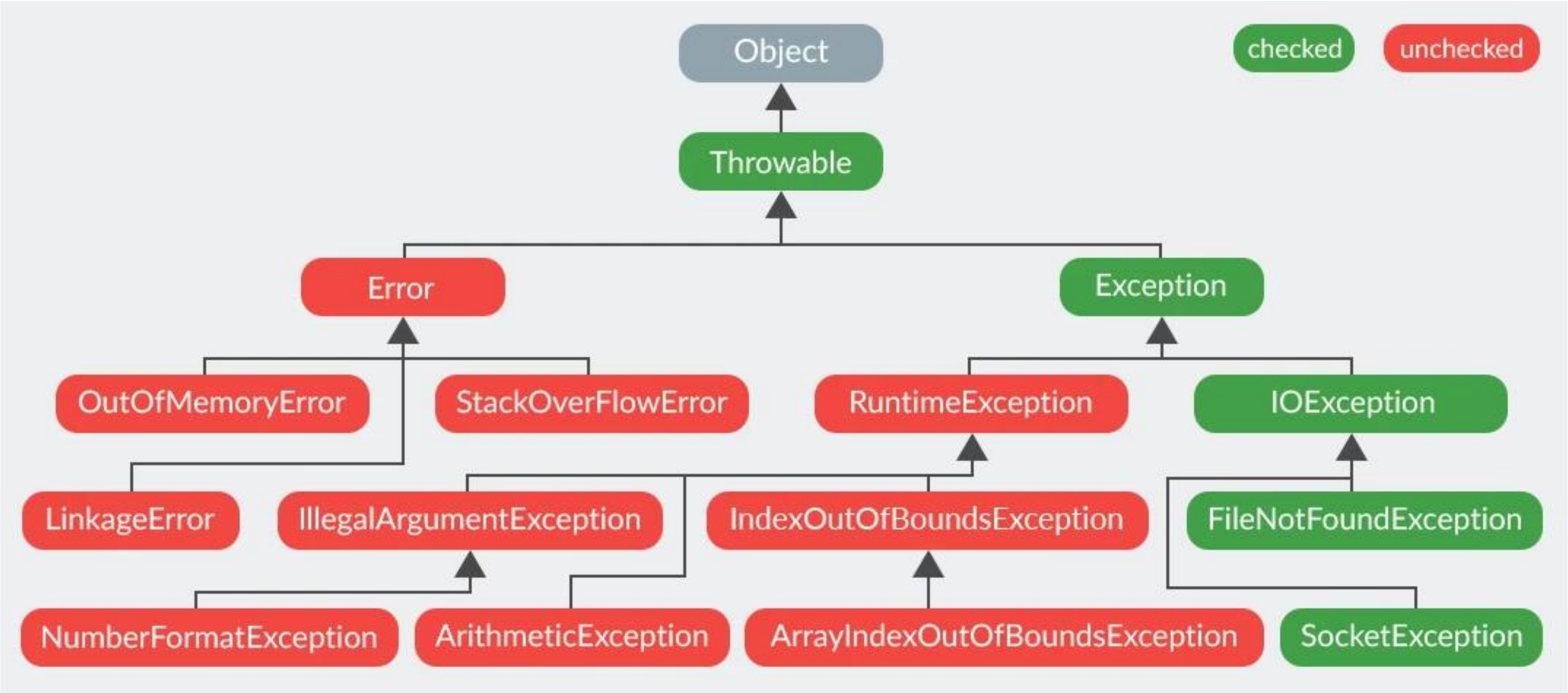
3 types of Exceptions in Java:

Checked exception: A good program ***should allow the user to recover from this****.*
→ Subject to the Catch or Specify Requirement.

Error: due to exceptional conditions ***external to the application****, you do not expect the program to recover from this.*
→ Not subject to the Catch or Specify Requirement.

Runtime exception: Exceptional conditions **internal to the application**, hard to anticipate or recover from.
→ Not subject to the Catch or Specify Requirement.

Checked vs Unchecked Exceptions



Iterators, relation with ADT

Very common operation on any collection of items (array list, list, or sequence) is to **march through its elements in order, one at a time**.

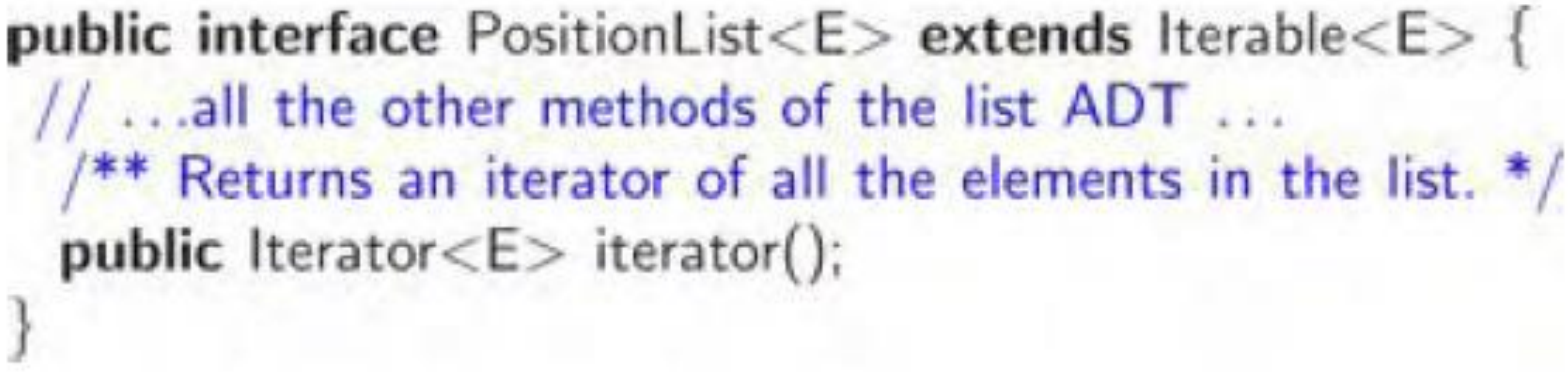
We can define the ***iterator*** ADT as supporting the following two methods:

- `hasNext()`: tell if there is a next element;
- `next()`: returns the next element in the iterator.

Iterators… relation with ADT

Java provides an interface: **`java.util.Iterator`**

ADTs storing **collections** of objects should be able to provide an **iterator** object. One way to enforce this is to have your ADT implementing **Iterable**



```
public interface PositionList<E> extends Iterable<E> {
 // ...all the other methods of the list ADT ...
  /** Returns an iterator of all the elements in the list. */
  public Iterator<E> iterator();
}
```

Iterators… the concept

– The bookmark for data structures!

  – Give access to all the items in a collection in some unspecified order

- Conceptually the iterator has a position between two elements

Operations

– Most important: `next()` and `hasNext()`

– Optional: `previous()`, `hasPrevious()`, `add()`, `remove()`

See it as a finger on the structure

Iterator Basics

Instead of…

```
    int curr= 0
    while(curr < size) {
        value =arr[curr++];
    }
```

We can have this:

```
    //curr is a new iterator //initialization
    while(curr.hasNext()) { //stop condition
        value = curr.next(); //get value AND increment
    }
```

=> The iterator needs to do this:

- data initialization?

- when should we stop?

- how do we get the next object and move over?

Iterator Basics: Dynamic Array

Instead of…

```
int curr= 0
while(curr < size) {
    value =arr[curr++];
}
```

We can have this:

```
//curr is a new iterator //initialization
while(curr.hasNext()) { //stop condition
    value = curr.next(); //get value AND increment
}
```

=> The iterator needs to do this:

- data initialization	→ `curr = 0;`
- when should we stop	→ `curr < size`
- how do we get the next object and move over	→ `arr[curr++]`

Questions?

Java Iterators

Interface Iterable<T>

–**java.lang**

–Iterator<T> iterator()

–http://docs.oracle.com/javase/8/docs/api/java/lang/Iterable.html

Interface Iterator<E>

–**java.util**

–booleanhasNext()

–E next()

–http://docs.oracle.com/javase/8/docs/api/java/util/Iterator.html

Iterators

The iterator needs to have **access to the stored data**…

1) Provide and **use a reference to the underlying structure**…
2) Or, create and **work on a copy** of the underlying data storage

Pros and Cons…

Iterator implemented as a **different class** vs **nested class**?

Nester Review: Advanced Java (from CS211)

Nested classes:

- Static class
- Inner class (non static nested class)
- Local class
- Anonymous class

Review: Advanced Java

- Static class

```
class Outer {
        static class StaticNested{
                /*Code Here*/
        }

        void aMethod() {
                StaticNested obj= new StaticNested();
        }
}

//outside the Outer class...
Outer.StaticNested obj = new Outer.StaticNested();
```

Review: Advanced Java

- Inner class

```
class Outer {
       class Inner {
              /*Code Here*/
       }

       void aMethod() {
              Inner obj= new Inner();
       }
}

//outside the Outer class using an instance of Outer...
Outer.Inner inObj= outObj.new Inner();
```

Review: Advanced Java

- Local class

```
class Outer {
       void aMethod() {
             class Local {
                    /*Code Here*/
             }

             Local loc= new Local();
       }
       /*Can't access Local outside aMethod!*/
}
```

Review: Advanced Java

- Anonymous class

```
interface Exporter {
       public String export();
}

class MyClass{
       public Exporter getExporter() {
             return new Exporter() {
                    public String export() {
                           return "Export";
                    }
             }; //<-very important
       }
}
```

Iterators implementation

```
class MyIterator implements Iterator {

        private MyList data;
        private int current = 0;
        public MyIterator(MyList a) {
                data = a;
        }

        public boolean hasNext() {
                ...
                return ...
        }

        public Object next() {
                if (hasNext())
                        return data.get(current++);
                else throw new NoSuchElementException();
        }

        public void remove() {
                throw new UnsupportedOperationException();
        }
}
```

Iterators implementation (use in our List)

```
class MyList implements Iterable {
        public Object[] data;

        public MyList (Object[] a) {
                data = a;
        }

        public Iterator iterator () {
                return new MyIterator(this);
        }
}
```

Iterators implementation (Simpler as Inner Class)

```
class MyList implements Iterable {
        private Object[] data;

        public MyList(Object[] a) {
                data = a;
        }

        public Iterator iterator () {
                return new MyIterator();
        }

        private class MyIterator implements Iterator {
                private int index = 0;
                public boolean hasNext () {
                        return (index < data.length);
                }
                public Object next () {
                        if (hasNext()) return data[index++];
                        else throw new NoSuchElementException();
                }
                public void remove () {
                        throw new UnsupportedOperationException();}
                }
        }
```

Iterators implementation
(It can be more compact with Anonymous Class)

```
class MyList implements Iterable {
        private Object[] data;

        public MyList(Object[] a) {
                data = a;
        }

        public Iterator iterator () {
                return new MyIterator();
        }

        private class MyIterator implements Iterator {
                private int index = 0;
                public boolean hasNext () {
                        return (index < data.length);
                }
                public Object next () {
                        if (hasNext()) return data[index++];
                        else throw new NoSuchElementException();
                }
                public void remove () {
                        throw new UnsupportedOperationException();}
                }
        }
```

Iterators implementation
(It can be more compact with Anonymous Class)

```
class MyList implements Iterable {
        private Object[] data;

        public MyList(Object[] a) {
                data = a;
        }

        public Iterator iterator () {
                return new Iterator {
                        private int index = 0;
                        public boolean hasNext () {
                                return (index < data.length);
                        }
                        public Object next () {
                                if (hasNext()) return data[index++];
                                else throw new NoSuchElementException();
                        }
                        public void remove () {
                                throw new UnsupportedOperationException();}
                        }
                };
        }
```

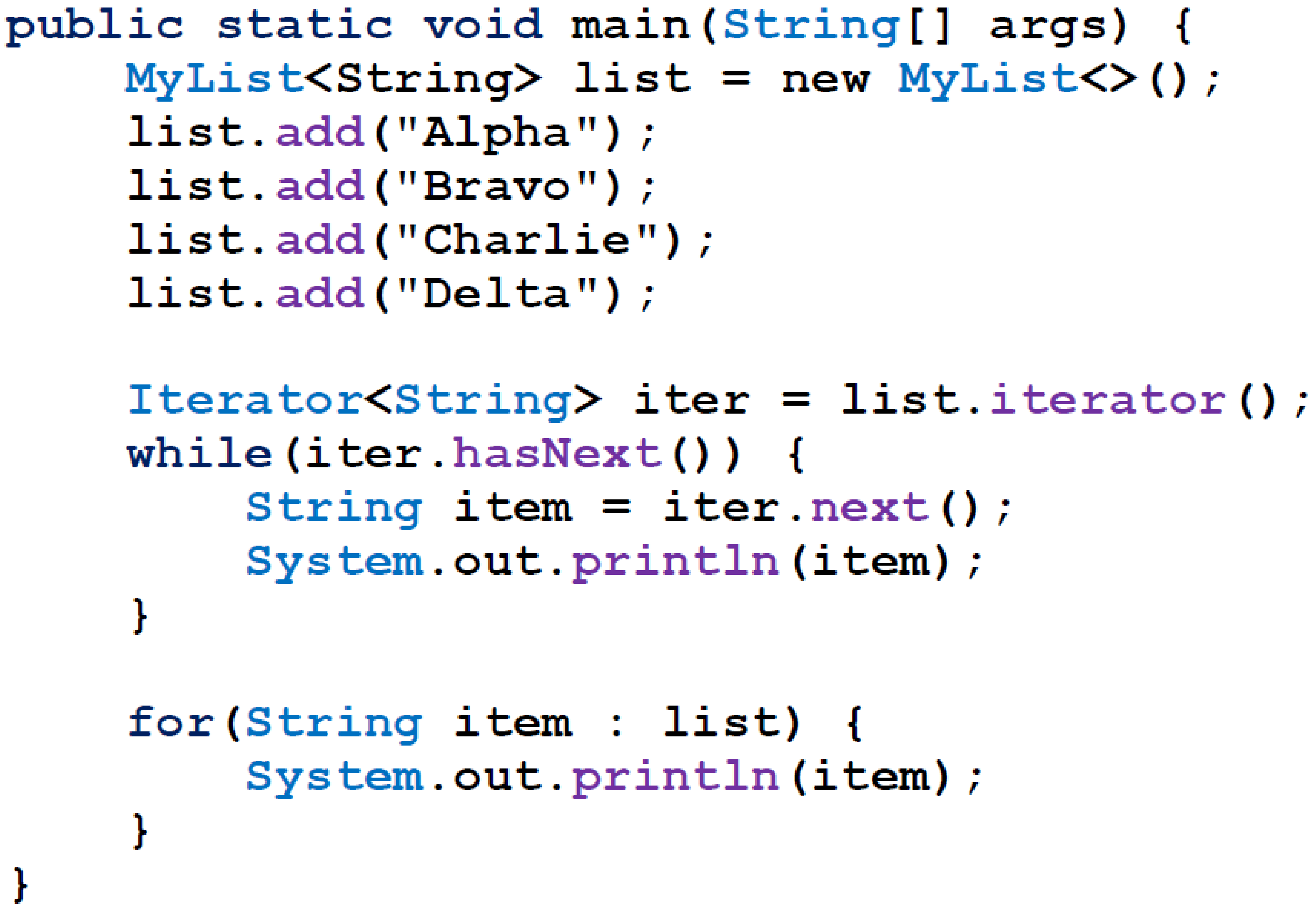
Iterators with Generics

What type of inner class are we using here?

```
import java.util.Iterator;

class MyList<T> implements Iterable<T> {
    public Iterator<T> iterator() {
        return new Iterator<T>() {
            public boolean hasNext() { ... }
            public T next() { ... }
        };
    }
}
```

2 Ways to use Iterator

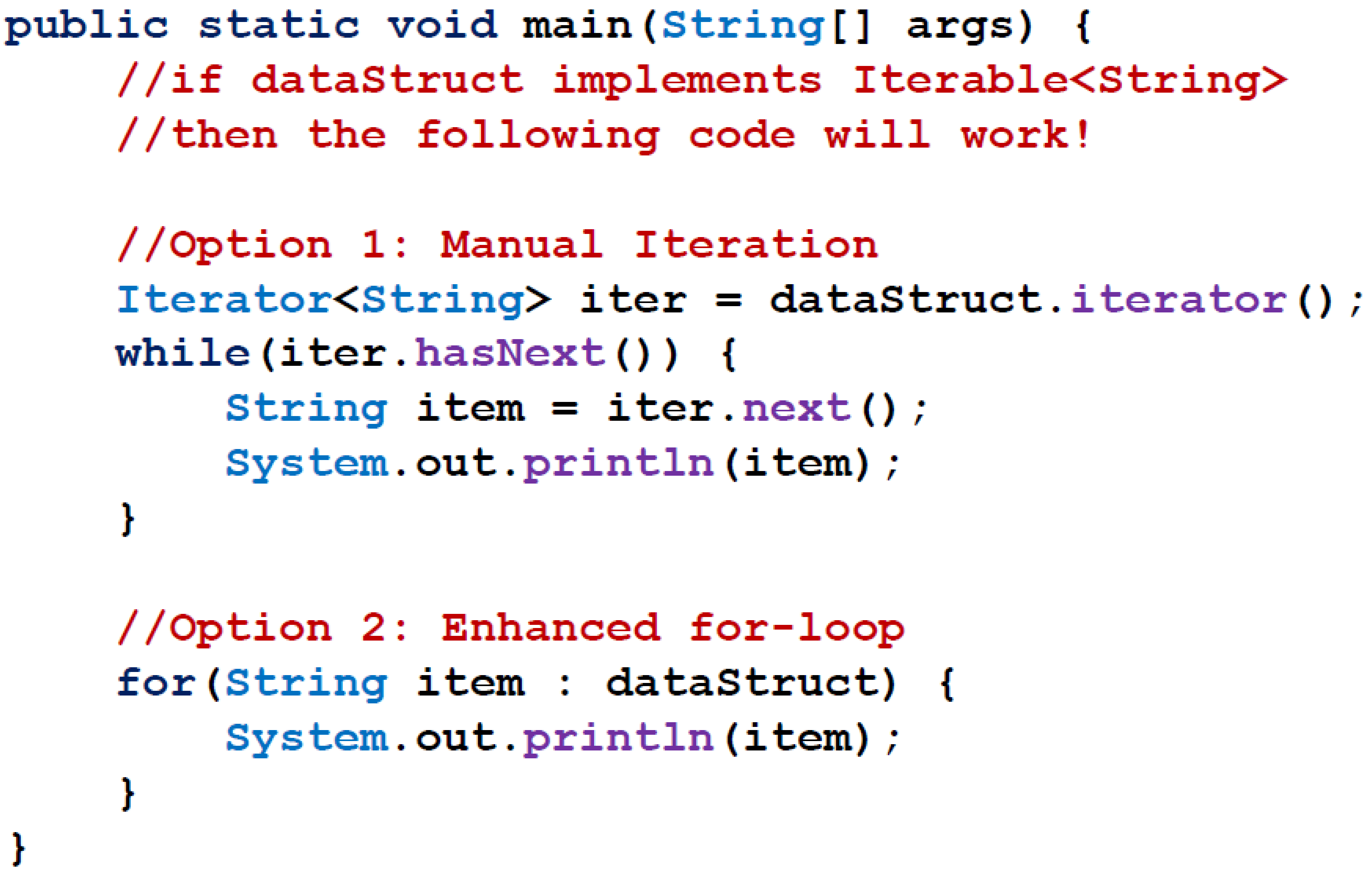


```java
public static void main(String[] args) {
    MyList<String> list = new MyList<>();
    list.add("Alpha");
    list.add("Bravo");
    list.add("Charlie");
    list.add("Delta");

    Iterator<String> iter = list.iterator();
    while(iter.hasNext()) {
        String item = iter.next();
        System.out.println(item);
    }

    for(String item : list) {
        System.out.println(item);
    }
}
```

What about other data structure? Same…



```
public static void main(String[] args) {
    //if dataStruct implements Iterable<String>
    //then the following code will work!

    //Option 1: Manual Iteration
    Iterator<String> iter = dataStruct.iterator();
    while(iter.hasNext()) {
        String item = iter.next();
        System.out.println(item);
    }

    //Option 2: Enhanced for-loop
    for(String item : dataStruct) {
        System.out.println(item);
    }
}
```

Why do we need Iterator/Iterable?

Each data structure is fundamentally different.

Without Iterable/Iterator, clients would have to develop code tailored to each data structure in order to accomplish this.

Using common interface, same code to traverse any data structure:

Lists (Dynamic Arrays, Linked Lists, ...)

Stacks and Queues

Trees and Graphs

More on this later!!!

Questions?

Next Lecture (Lecture 06)

1. **Linked List**
2. **Stacks and Queues**

Reminders:

Do the readings (Textbook)

Work on P1