# Object-Oriented Programming

## Abstract Data Type

Understanding data abstraction

Defining ADT with Java Interface

Implementing data structure given a Java Interface

## Outline

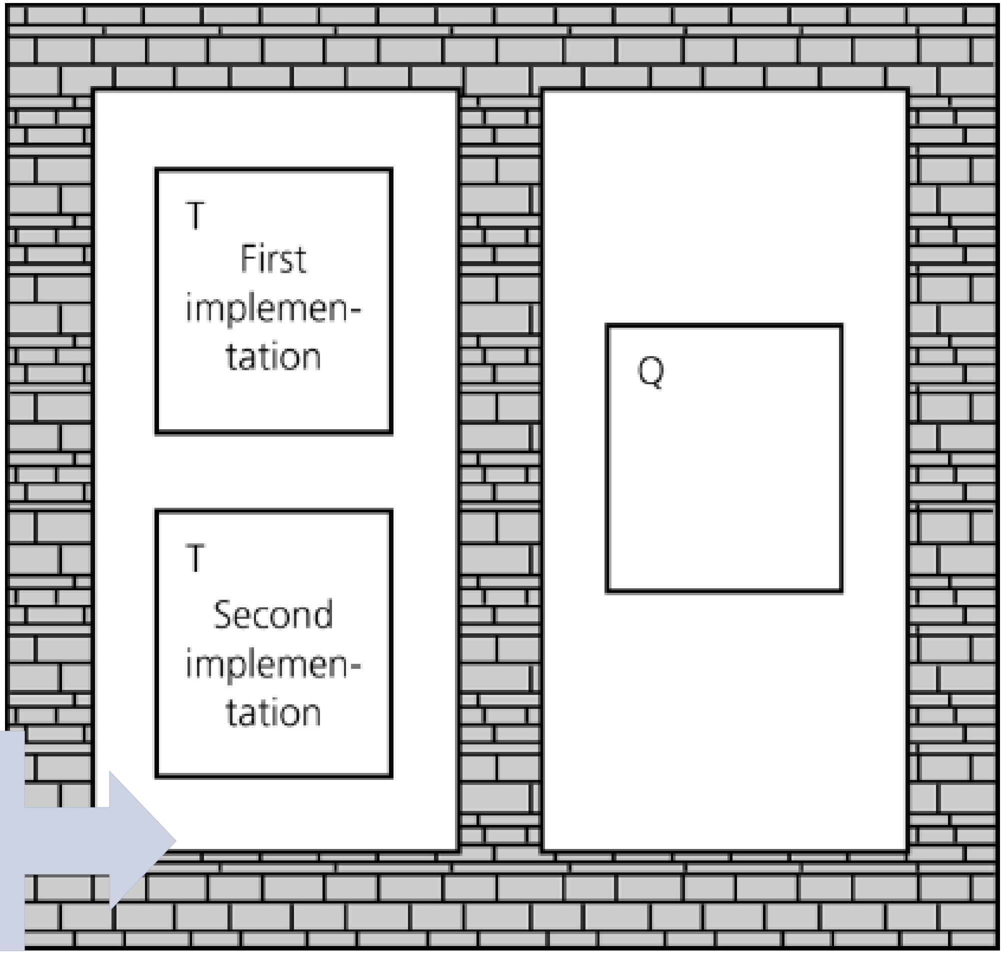
# Motivation

## Program Design Principles

- **Abstraction**
  - Concentrate on what it can do and NOT how it does it
  - Eg: Use of Java Interface
- **Coupling**
  - Restrict interdependent relationship among classes to the minimum
- **Cohesion**
  - A class should be about a single entity only
  - There should be a clear logical grouping of all functionalities
- **Information Hiding**
  - Expose only necessary information to outside

## Information Hiding

- Information hiding is like walls building around the various classes of a program.
- The wall around each class *T* prevents the other classes from seeing how *T* works.
- Thus, if class *Q* uses (depends on) *T*, and if the approach for performing *T* changes, class *Q* will not be affected.




Makes it easy to substitute new, improved versions of how to do a task later

- **Information Hiding** is not complete isolation of the classes
  - Information released is on a **need-to-know** basis
  - Class *Q* does not know how class *T* does the work, but it needs to know how to invoke *T* and what *T* produces
    - E.g: The designers of the methods of **Math** and **Scanner** classes have hidden the details of the implementations of the methods from you, but provide enough information (the method headers and explanation) to allow you to use their methods
  - What goes in and comes out is governed by the terms of the method's specifications
    - If you use this method in this way, this is exactly what it will do for you (pre- and post-conditions)

# Software Engineering Issues (4/5)

- Pre- and post-conditions (for documentation)
  - Pre-conditions
    - Conditions that must be true before a method is called
    - "This is what I expect from you"
    - The programmer is responsible for making sure that the pre-conditions are satisfied when calling the method
  - Post-conditions
    - Conditions that must be true after the method is completed
    - "This is what I promise to do for you"
  - Example

```
// Pre-cond: x >= 0
// Post-cond: Return the square root of x
public static double squareRoot(double x) {
   . . .
}
```

- **Information Hiding** CAN also apply to data

  - **Data abstraction** asks that you think in terms of what you can do to a collection of data independently of how you do it

  - **Data structure** is a construct that can be defined within a programming language to store a collection of data

  - **Abstract data type (ADT)** is a collection of data & a specification on the set of operations/methods on that data
    - Typical operations on data are: *add*, *remove*, and *query* (in general, management of data)
    - Specification indicates what ADT operations do, but not how to implement them

Collection of data + set of operations on the data

- **Data structure** is a construct that can be defined within a programming language to store a collection of data
  - Arrays, which are built into Java, are data structures
  - We can create other data structures. For example, we want a data structure (a collection of data) to store both the names and salaries of a collection of employees

```java
static final int MAX_NUMBER = 500;  // defining a constant
String[] names = new String[MAX_NUMBER];
double[] salaries = new double[MAX_NUMBER];
// employee names[i] has a salary of salaries[i]
```

Or (better choice)

```java
class Employee {
  static final int MAX_NUMBER = 500;
  private String names;
  private double salaries;
}
...
Employee[] workers = new Employee[Employee.MAX_NUMBER];
```

## Abstract Data Type (ADT) (1/4)

- An **ADT** is a collection of data together with a specification of a set of operations on the data
  - Specifications indicate what ADT operations do, *NOT* how to implement them
  - **Data structures** are part of an ADT's implementation


- When a program needs data operations that are not directly supported by a language, you need to create your own ADT
- You should first design the ADT by carefully specifying the operations before implementation

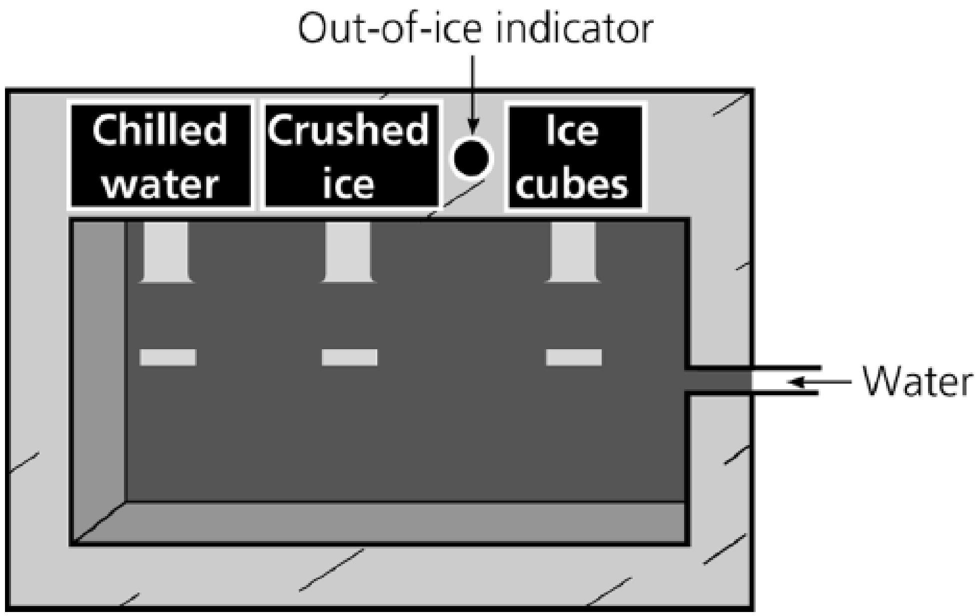
## Abstract Data Type (ADT) (2/4)

- Example: A water dispenser as an ADT
- Data: water
- Operations: *chill*, *crush*, *cube*, and *isEmpty*
- Walls: made of steel

**The only slits in the walls:**

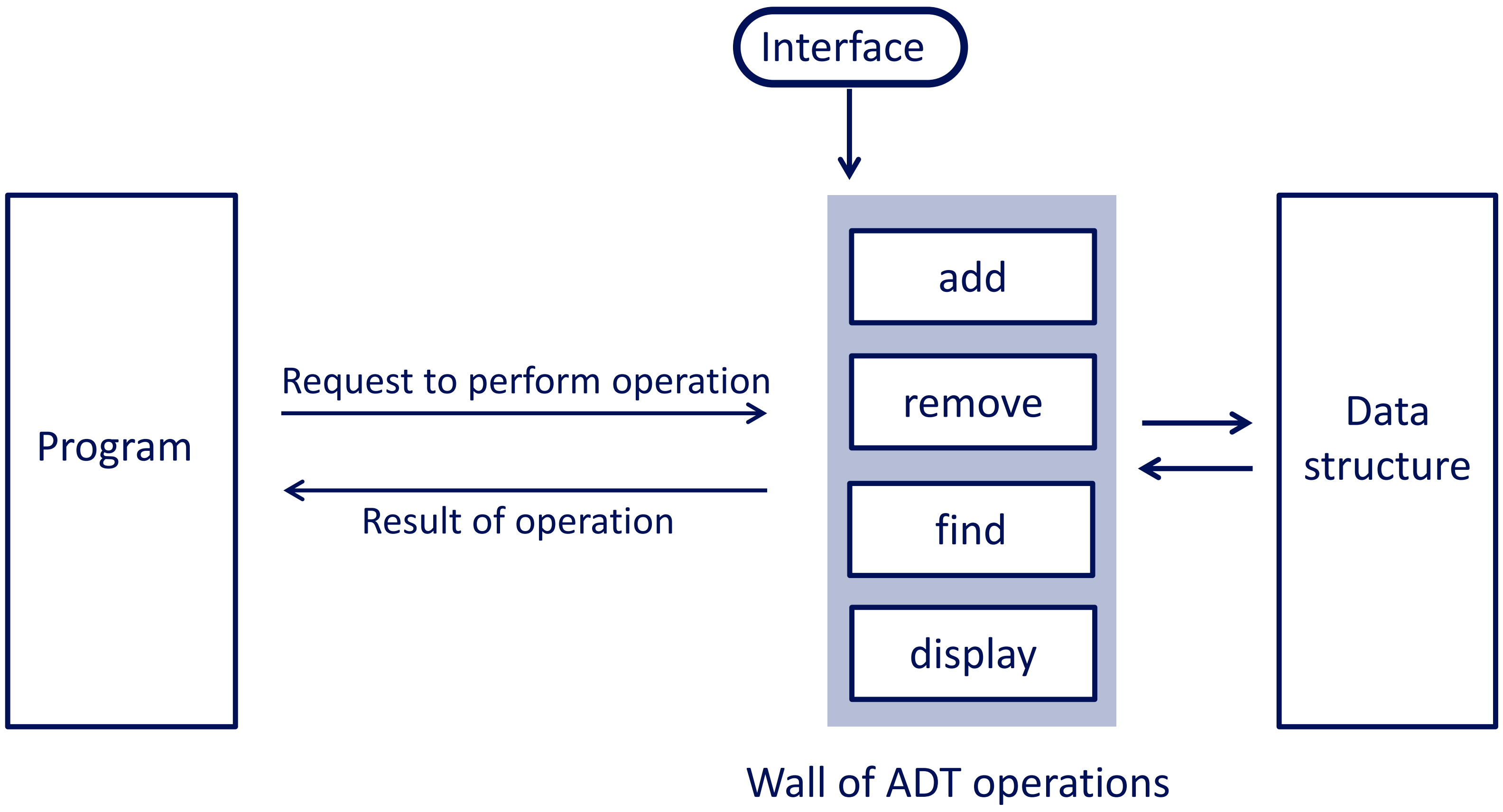
- **Input: water**
- **Output: chilled water, crushed ice, or ice cubes.**

Crushed ice can be made in many ways.
We don't care how it was made

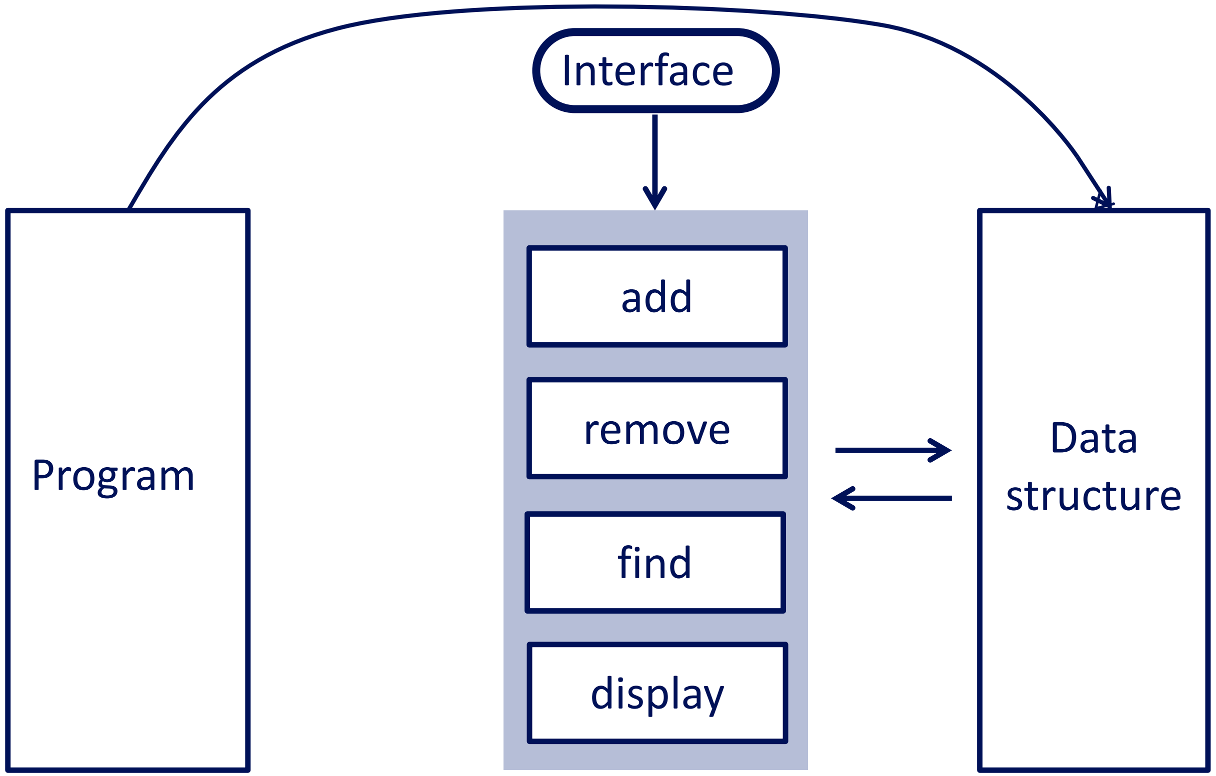



- Using an ADT is like using a vending machine.

- A WALL of ADT operations isolates a data structure from the program that uses it
- An interface is what a program/module/class should understand on using the ADT



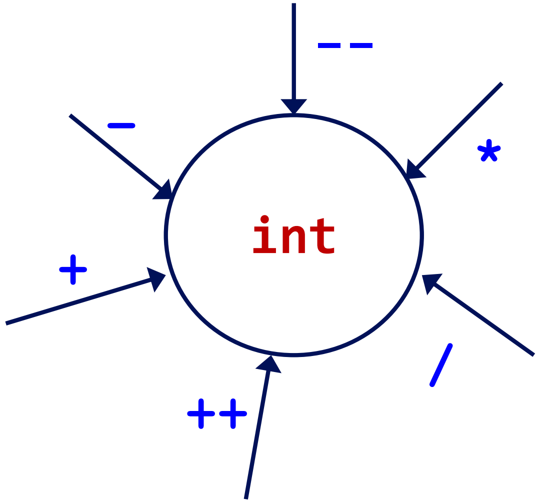
- An interface is what a program/module/class should understand on using the ADT
- The following bypasses the interface to access the data structure. This violates the wall of ADT operations.

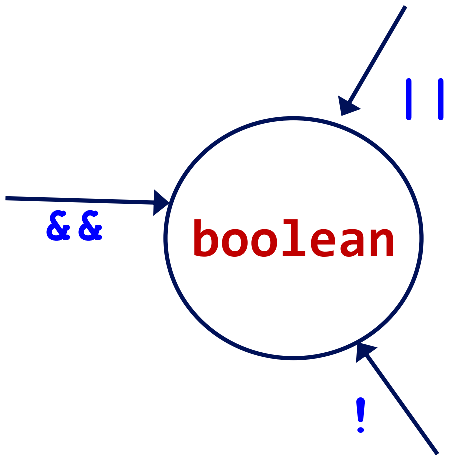



Wall of ADT operations

## Eg: Primitive Types as ADTs (1/2)

- Java's predefined data types are ADTs
- Representation details are hidden which aids portability as well
- Examples: int, boolean, double




`int` type with the operations (e.g.: --, /) defined on it.

`boolean` type with the operations (e.g.: &&) defined on it.

- Broadly classified as:
  (the example here uses the array ADT)

  - Constructors (to add, create data)

    ```
    - int[] z = new int[4];
    - int[] x = {2,4,6,8};
    ```

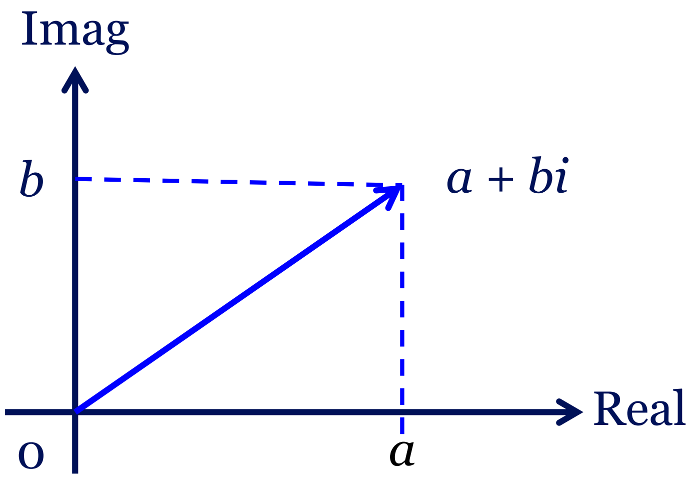
  - Mutators (to modify data)

    ```
    - x[3] = 10;
    ```

  - Accessors (to query about state/value of data)

    ```
    - int y = x[3] + x[2];
    ```

- A **complex number** comprises a real part *a* and an imaginary part *b*, and is written as *a* + *bi*

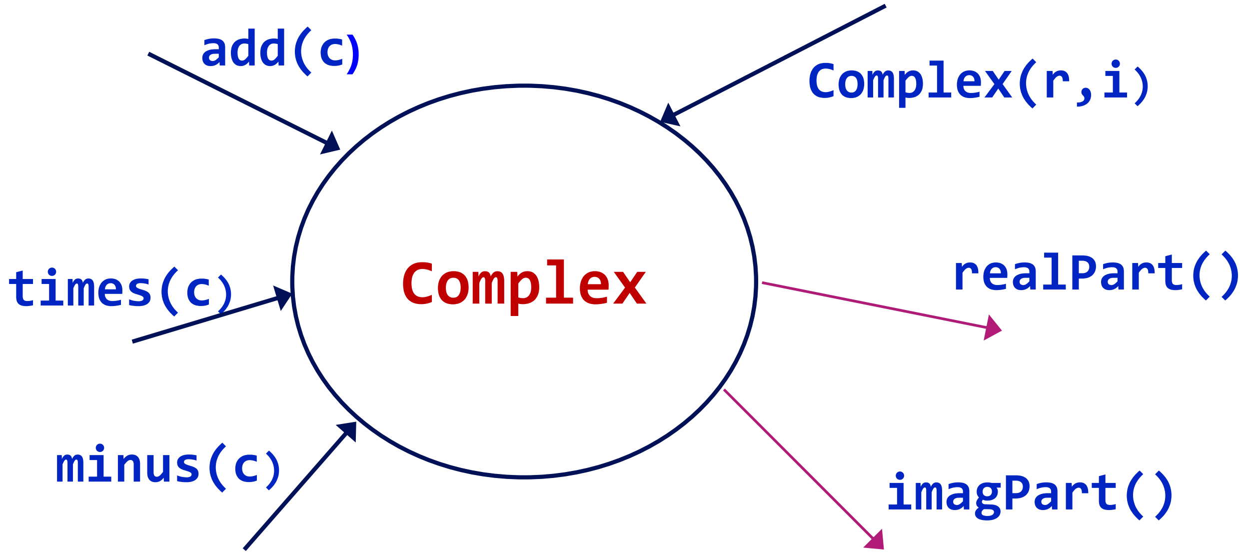



- *i* is a value such that $i^2 = -1$.

- Examples: 12 + 3*i*, 15 – 9*i*, -5 + 4*i*, -23, 18*i*

- A complex number can be visually represented as a pair of numbers (*a*, *b*) representing a vector on the two-dimensional complex plane (horizontal axis for real part, vertical axis for imaginary part)

- User-defined data types can also be organized as ADTs
- Let's create a "Complex" ADT for complex numbers




Note: add(c) means to add complex number object c to "this" object. Likewise for times(c) and minus(c).

- A possible Complex ADT class:

```
class Complex {
  private ...             // data members
  public Complex(double r, double i) { ... }    // create a new object
  public void add(Complex c) { ... }            // this = this + c
  public void minus(Complex c) { ... }          // this = this - c
  public void times(Complex c) { ... }          // this = this * c
  public double realpart() { ... }              // returns this.real
  public double imagpart() { ... }              // returns this.imag
}
```

- Using the Complex ADT:

```
Complex c = new Complex(1,2);     // c = (1,2)
Complex d = new Complex(3,5);     // d = (3,5)
c.add(d);                         // c = c + d
d.minus(new Complex(1,1));        // d = d - (1,1)
c.times(d);                       // c = c * d
```

## Eg: Complex Number as ADT (4/6)

**Complex.java**

```java
class Complex {
  private double real;
  private double imag;

  // CONSTRUCTOR
  public Complex(double r, double i) {
      real = r;
      imag = i;
  }

  // ACCESSORS
  public double realPart() {
      return real;
  }

  public double imagPart() {
      return imag;
  }
```

**Complex.java**

```java
  // MUTATORS
  public void add(Complex c) { // this = this + c
    real += c.realPart();
    imag += c.imagPart();
  }

  public void minus(Complex c) { // this = this - c
    real -= c.realPart();
    imag -= c.imagPart();
  }

  public void times(Complex c) { // this = this * c
    real = real * c.realPart() - imag * c.imagPart();
    imag = real * c.imagPart() + imag * c.realPart();
  }
}
```

One possible implementation: Cartesian

## Eg: Complex Number as ADT (5/6)

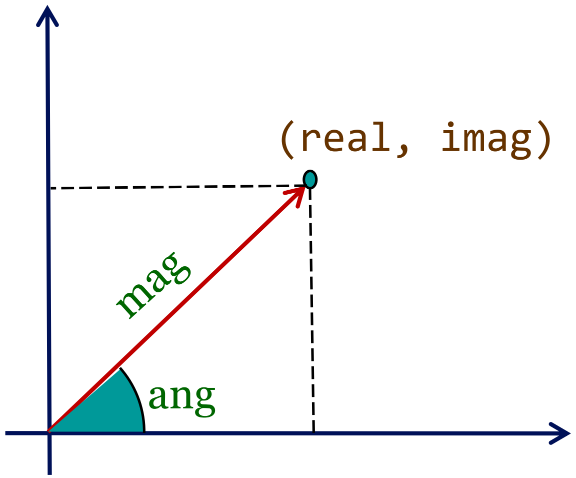
**Complex.java**

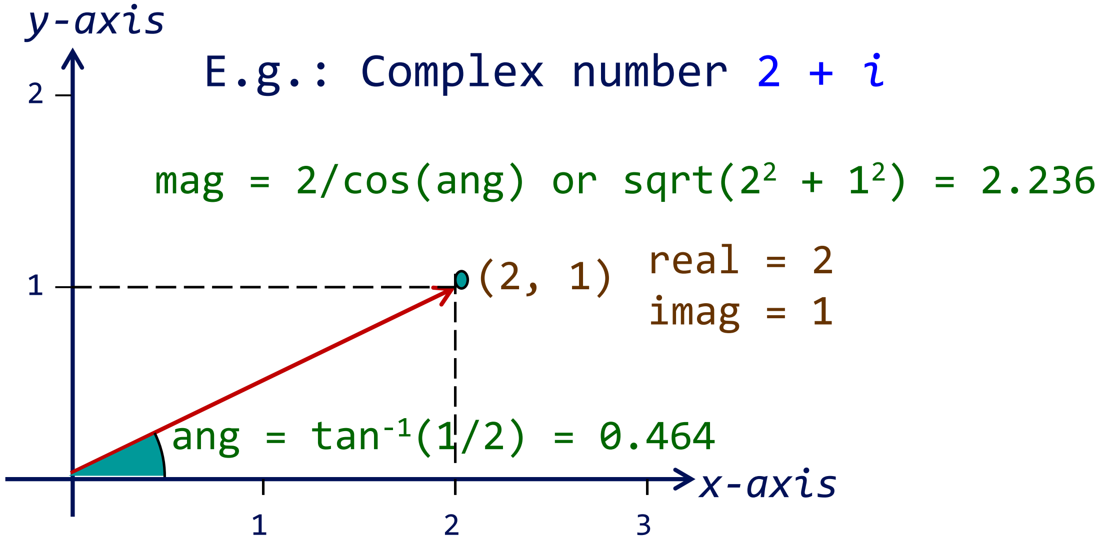
```
class Complex {

  private double ang;   // the angle of the vector
  private double mag;   // the magnitude of the vector
      :
      :
  public times(Complex c)  {  // this = this * c
      ang += c.angle();
      mag *= c.mag();
  }
      :
      :
}
```

One possible implementation: Polar

- "Relationship" between Cartesian and Polar representations

```
From Polar to Cartesian:  real = mag * cos(ang);
                          imag = mag * sin(ang);

From Cartesian to Polar:  ang = tan-1(imag/real);
                          mag = real / cos(ang);
                      or  mag = sqrt(real2 + imag2);
```



# Specifying related methods

- Java interfaces provide a way to specify common behaviour for a set of (possibly unrelated) classes

- Java interface can be used for ADT
  - It allows further abstraction/generalization
  - It uses the keyword **interface**, rather than **class**
  - It specifies methods to be implemented
    - A Java interface is a group of related methods with empty bodies
  - It can have constant definitions (which are implicitly `public static final`)

- A class is said to implement the interface if it provides implementations for **ALL** the methods in the interface

```java
// package in java.lang;
public interface Comparable <T> {
  int compareTo(T other);
}
```

```java
class Shape implements Comparable <Shape> {
  static final double PI = 3.14;

  double area() {...};
  double circumference() { ... };
  int compareTo(Shape x) {
      if (this.area() == x.area()) {
          return 0;
  } else if (this.area() > x.area()) {
      return 1;
  } else {
      return -1;
  }
}
```

Implementation of compareTo()

# E.g. Complex ADT interface

- anticipate both Cartesian and Polar implementations

Complex.java

```
public interface Complex {
  public double realpart();         // returns this.real
  public double imagpart();         // returns this.imag
  public double angle();            // returns this.ang
  public double mag();              // returns this.mag
  public void add(Complex c);       // this = this + c
  public void minus(Complex c);     // this = this - c
  public void times(Complex c);     // this = this * c
}
```

- In Java 7 and earlier, methods in an interface only have **signatures** (headers) but <u>no implementation</u>
- However, Java 8 introduces “default methods” to interfaces. They provide default implementations which can be overridden by the implementing class.

# Example #2: ComplexCart (1/2)

**ComplexCart.java**

```java
class ComplexCart implements Complex {
  private double real;
  private double imag;

  // CONSTRUCTOR
  public ComplexCart(double r, double i) {
    real = r;
    imag = i;
  }
  // ACCESSORS
  public double realPart() {
    return this.real;
  }
  public double imagPart() {
    return this.imag;
  }
  public double imagPart() {
    return this.imag;
  }
```

**ComplexCart.java**

```java
  public double mag() {
    return Math.sqrt(real * real + imag * imag);
  }

  public double angle() {
    if (real != 0) {
      if (real < 0) {
        return (Math.PI + Math.atan(imag/real));
      } else {
        return Math.atan(imag/real);
      }
    } else if (imag == 0) {
      return 0;
    } else if (imag > 0) {
      return Math.PI/2;
    } else {
      return -Math.PI/2;
    }
  }
```

# Example #2: ComplexCart (2/2)

**ComplexCart.java**

```java
    // MUTATORS
    public void add(Complex c) {
      this.real += c.realPart();
      this.imag += c.imagPart();
    }

    public void minus(Complex c) {
      this.real -= c.realPart();
      this.imag -= c.imagPart();
    }
```

**ComplexCart.java**

```java
    // MUTATORS
    public void times(Complex c) {
      double tempReal = real * c.realPart()
                      - imag * c.imagPart();
      imag = real * c.imagPart() + imag * c.realPart();
      real = tempReal;
    }

    public String toString() {
      if (imag == 0)
        return (real + "");

      if (imag < 0) {
        return (real + "" + imag + "i");

      return (real + "+" + imag + "i");
    }
}
```

# Example #2: ComplexPolar (1/3)

**ComplexPolar.java**

```java
class ComplexPolar implements Complex {
  private double mag; // magnitude
  private double ang; // angle

  // CONSTRUCTOR
  public ComplexPolar(double m, double a) {
    mag = m;
    ang = a;
  }
  // ACCESSORS
  public double realPart() {
    return mag * Math.cos(ang);
  }
  public double imagPart() {
    return mag * Math.sin(ang);
  }
  public double mag() {
    return mag;
  }
  public double angle() {
    return ang;
  }
```

**ComplexPolar.java**

```java
  // MUTATORS
  public void add(Complex c) {   // this = this + c
    double real = this.realPart() + c.realPart();
    double imag = this.imagPart() + c.imagPart();

    mag = Math.sqrt(real*real + imag*imag);
    if (real != 0) {
      if (real < 0) {
        ang = (Math.PI + Math.atan(imag/real));
      }
    } else {
      ang = Math.atan(imag/real);
    } else if (imag == 0) {
      ang = 0;
    } else if (imag > 0) {
      ang = Math.PI/2;
    } else {
      ang = -Math.PI/2;
    }
  }
```

## Example #2: ComplexPolar (2/3)

**ComplexPolar.java**

```java
public void minus(Complex c) { // this = this - c
  double real = mag * Math.cos(ang) - c.realPart();
  double imag = mag * Math.sin(ang) - c.imagPart();
  mag = Math.sqrt(real * real + imag * imag);
  if (real != 0) {
    if (real < 0) {
      ang = (Math.PI + Math.atan(imag / real));
    } else {
      ang = Math.atan(imag / real);
    }
  } else if (imag == 0) {
    ang = 0;
  } else if (imag > 0) {
    ang = Math.PI / 2;
  } else {
    ang = -Math.PI / 2;
  }
}
```

# Example #2: ComplexPolar (3/3)

**ComplexPolar.java**

```java
    public void times(Complex c) { // this = this * c
      mag *= c.mag();
      ang += c.angle();
    }

    public String toString() {
      if (imagPart() == 0)
        return (realpart() + "");

      if (imagPart() < 0) {
        return (realPart() + "" + imagPart() + "i");

      return (realPart() + "+" + imagPart() + "i");
    }
  }
```

## Example #2: TestComplex (1/3)

### TestComplex.java

```java
public class TestComplex {
  public static void main(String[] args) {
    // Testing ComplexCart
    Complex a = new ComplexCart(10.0, 12.0);
    Complex b = new ComplexCart(1.0, 2.0);

    System.out.println("Testing ComplexCart:");
    a.add(b);
    System.out.println("a = a + b is " + a);
    a.minus(b);
    System.out.println("a - b (which is the original a) is " + a);
    System.out.println("Angle of a is " + a.angle());
    a.times(b);
    System.out.println("a = a * b is " + a);
```

```
Testing ComplexCart:
a = a + b is 11.0+14.0i
a - b (which is the original a) is 10.0+12.0i
Angle of a is 0.8760580505981934
a = a * b is -14.0+32.0i
```

## Example #2: TestComplex (2/3)

**TestComplex.java**

```java
    // Testing ComplexPolar
    Complex c = new ComplexPolar(10.0, Math.PI/6.0);
    Complex d = new ComplexPolar(1.0, Math.PI/3.0);

    System.out.println("\nTesting ComplexPolar:");
    System.out.println("c is " + c);
    System.out.println("d is " + d);
    c.add(d);
    System.out.println("c = c + d is " + c);
    c.minus(d);
    System.out.println("c - d (which is the original c) is " + c);
    c.times(d);
    System.out.println("c = c * d is " + c);
```

```
Testing ComplexPolar:
c is 8.660254037844387+4.999999999999999i
d is 5.000000000000001+8.660254037844386i
c = c + d is 13.660254037844393+13.660254037844387i
c - d (which is ... c) is 8.660254037844393+5.000000000000002i
c = c * d is 2.83276944823992E-14+100.00000000000007i
```

## Example #2: TestComplex (3/3)

**TestComplex.java**

```java
        // Testing Combined
        System.out.println("\nTesting Combined:");
        System.out.println("a is " + a);
        System.out.println("d is " + d);
        a.minus(d);
        System.out.println("a = a - d is " + a);
        a.times(d);
        System.out.println("a = a*d is " + a);
        d.add(a);
        System.out.println("d = d + a is " + d);
        d.times(a);
        System.out.println("d = d*a is " + d);
    }
}
```

```
Testing Combined:
a is -14.0+32.0i
d is 5.000000000000001+8.660254037844386i
a = a - d is -19.0+23.339745962155614i
a = a * d is -297.1281292110204-47.84609690826524i
d = d + a is -292.1281292110204 5-39.18584287042089i
d = d * a is 84924.59488697552+25620.40696350589i
```

- Each interface is compiled into a separate bytecode file, just like a regular class
  - We cannot create an instance of an interface, but we can use an interface as a data type for a variable, or as a result of casting

```
public boolean equals (Object cl) {
  if (cl instanceof Complex) {
      Complex temp = (Complex) cl;  // result of casting
      return (Math.abs(realPart() - temp.realPart()) < EPSILON
              && Math.abs(imagPart() - temp.imagPart()) < EPSILON);
  }

  return false;
}
```

**Note**: EPSILON is a very small value (actual value up to programmer), defined as a constant at the beginning of the class, e.g.:

```
public static final double EPSILON = 0.0000001;
```

# Practice Exercises

## Fraction as ADT (1/3)

- We are going to view **Fraction** as an ADT, before we proceed to provide two implementations of Fraction
- Qn: What are the data members (attributes) of a fraction object (without going into its implementation)?
- Qn: What are the behaviours (methods) you want to provide for this class (without going into its implementation)?

| Data members |
| --- |
| numerator |
| nenominator |

| Behaviors |
| --- |
| add |
| minus |
| times |
| simplify |

We will leave out divide for the moment

## Fraction as ADT (2/3)

- How do we write an **Interface** for Fraction? Let's call it FractionI
  - You may refer to interface Complex for idea
  - But this time, we wants add(), minus(), times() and simplify() to return a fraction object

**FractionI.java**

```
public interface FractionI {
  public int getNumer();                    // returns numerator part
  public int getDenom();                    // returns denominator part
  public void setNumer(int numer);          // sets new numerator
  public void setDenom(int denom);          // sets new denominator

  public FractionI add(FractionI f);        // returns this + f
  public FractionI minus(FractionI f);      // returns this - f
  public FractionI times(FractionI f);      // returns this * f
  public FractionI simplify();              // returns this simplified
}
```

- Now, to implement this Fraction ADT, we can try 2 approaches
  - **Fraction**: Use 2 integer data members for numerator and denominator
  - **FractionArr**: Use a 2-element integer array for numerator and denominator
  - We want to add a toString() method and an equals() method as well

# PracEx#26: TestFraction (1/2)

**TestFraction.java**

```
import java.util.*;

public class TestFraction {
  public static void main(String[] args) {
      Scanner sc = new Scanner(System.in);

      System.out.print("Enter 1st fraction: ");
      int a = sc.nextInt();
      int b = sc.nextInt();
      FractionI f1 = new Fraction(a, b);

      System.out.print("Enter 2nd fraction: ");
      a = sc.nextInt();
      b = sc.nextInt();
      FractionI f2 = new Fraction(a, b);

      System.out.println("1st fraction is " + f1);
      System.out.println("2nd fraction is " + f2);
```

- To write Fraction.java to implementation the FractionI interface.
- The client program TestFraction.java is given

## PracEx#26: TestFraction (2/2)

**TestFraction.java**

```java
        if (f1.equals(f2)) {
          System.out.println("The fractions are the same.");
        } else {
          System.out.println("The fractions are not the same.");
        }

        FractionI sum = f1.add(f2);
        System.out.println("Sum is " + sum);

        FractionI diff = f1.minus(f2);
        System.out.println("Difference is " + diff);

        FractionI prod = f1.times(f2);
        System.out.println("Product is " + prod);
    }
}
```

- To write Fraction.java, an implementation of FractionI interface.
- The client program TestFraction.java is given

```
Enter 1st fraction: 2 4
Enter 2nd fraction: 2 3
1st fraction is 2/4
2nd fraction is 2/3
The fractions are not the same.
Sum is 7/6
Difference is -1/6
Product is 1/3
```

## PracEx#26: Fraction (1/2)

**Fraction.java**

```java
class Fraction implements FractionI {
  // Data members
  private int numer;
  private int denom;
  // Constructors
  public Fraction() {
    this(1, 1);
  }
  public Fraction(int numer, int denom) {
    setNumer(numer);
    setDenom(denom);
  }
  // Accessors
  public int getNumer() { // fill in the code }
  public int getDenom() { // fill in the code }
  // Mutators
  public void setNumer(int numer) { // fill in the code }
  public void setDenom(int denom) { // fill in the code }
```

- Skeleton program for Fraction.java

## PracEx#26: Fraction (2/2)

**Fraction.java**

```java
    // Returns greatest common divisor of a and b
    // private method as this is not accessible to clients
    private static int gcd(int a, int b) {
      int remainder;
      while (b > 0) {
        remainder = a % b;
        a = b;
        b = remainder;
      }
      return a;
    }

    // Fill in the code for all the methods below
    public FractionI simplify() { // fill in the code }
    public FractionI add(FractionI f) { // fill in the code }
    public FractionI minus(FractionI f) { // fill in the code }
    public FractionI times(FractionI f) { // fill in the code }
    // Overriding methods toString() and equals()
    public String toString() { // fill in the code }
    public boolean equals() { // fill in the code }
  }
```

## PracEx#27: TestFractionArr

### TestFractionArr.java

```java
import java.util.*;

public class TestFractionArr {
  public static void main(String[] args) {

        Scanner sc = new Scanner(System.in);

        System.out.print("Enter 1st fraction: ");
        int a = sc.nextInt();
        int b = sc.nextInt();
        FractionI f1 = new FractionArr(a, b);

        System.out.print("Enter 2nd fraction: ");
        a = sc.nextInt();
        b = sc.nextInt();
        FractionI f2 = new FractionArr(a, b);

        // The rest of the code is the same as TestFraction.java
}
```

- To write FractionArr.java to implementation the FractionI interface.
- The client program TestFractionArr.java is given

## PracEx#27: FractionArr

### FractionArr.java

```java
class FractionArr implements FractionI {
  private int[] members;

  // Constructors
  public FractionArr() { this(1,1); }
  public FractionArr(int numer, int denom) {
      members = new int[2];
      setNumer(numer);
      setDenom(denom);
  }

  // Accessors
  public int getNumer() { // fill in the code }
  public int getDenom() { // fill in the code }

  // Mutators
  public void setNumer(int numer) { // fill in the code }
  public void setDenom(int denom) { // fill in the code }

  // The rest are omitted here
}
```

- Skeleton program for FractionArr.java

## Summary

- We learn about the need of data abstraction
- We learn about using Java Interface to define an ADT
- With this, we will learn and define various kinds of ADTs/data structures in subsequent lectures

# Objectives

1. Able to define a List ADT
2. Able to implement a List ADT with array
3. Able to implement a List ADT with linked list
4. Able to use Java API LinkedList class

## Outline

# Motivation

- **List** is one of the most basic types of data collection
  - For example, list of groceries, list of modules, list of friends, etc.
  - In general, we keep items of the same type (class) in one list

- Typical Operations on a data collection
  - Add data
  - Remove data
  - Query data
  - The details of the operations vary from application to application. The overall theme is the management of data

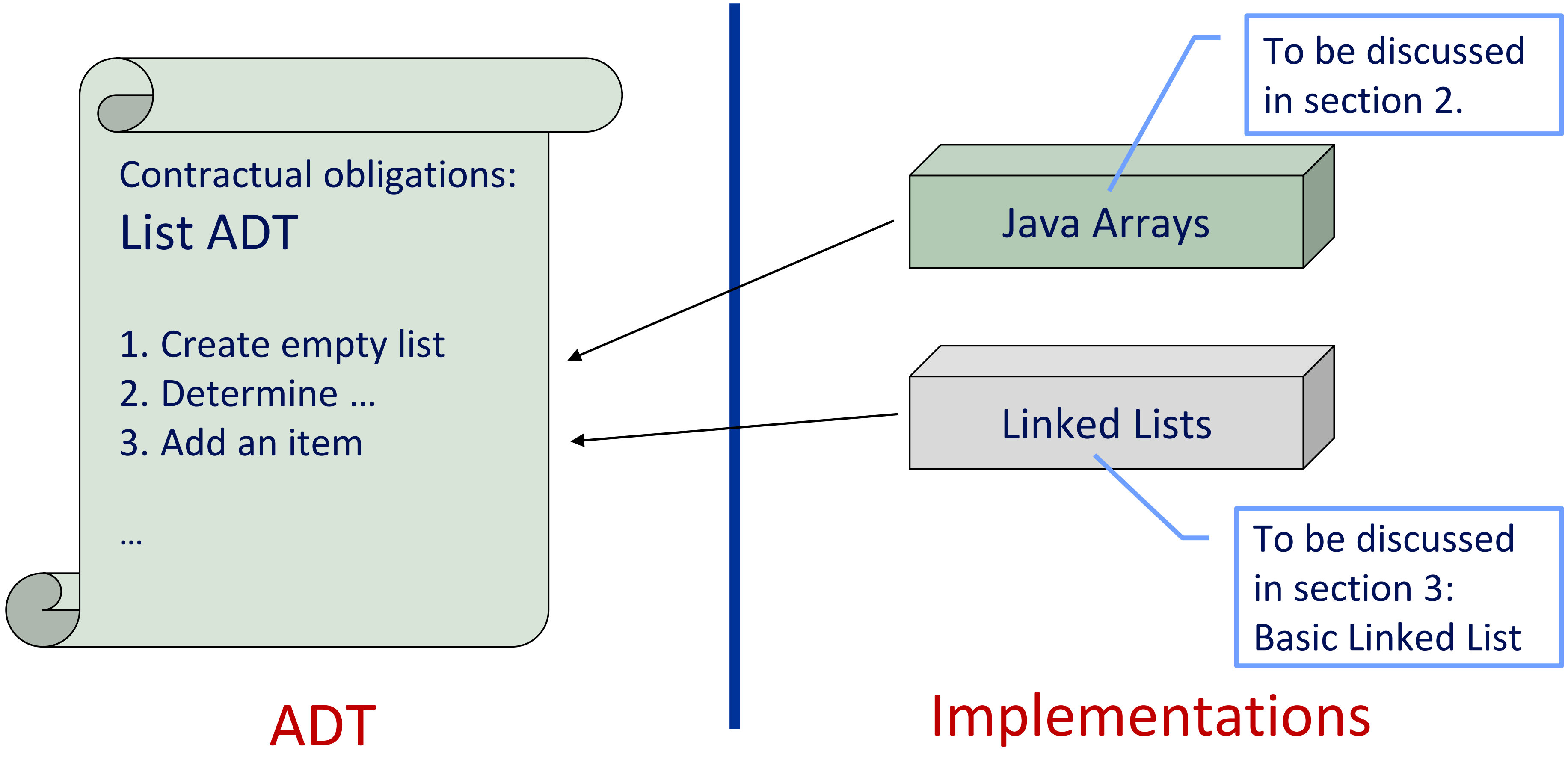
- A list ADT is a dynamic linear data structure
  - A collection of data items, accessible one after another starting from the beginning (head) of the list
- Examples of List ADT operations:
  - Create an empty list
  - Determine whether a list is empty
  - Determine number of items in the list
  - Add an item at a given position
  - Remove an item at a position
  - Remove all items
  - Read an item from the list at a position
- The next slide on the basic list interface does not have all the above operations… we will slowly build up these operations in list beyond the basic list.

ListInterface.java

```
import java.util.*;

public interface ListInterface<E> {
  public boolean isEmpty();
  public int     size();
  public E       getFirst() throws NoSuchElementException;
  public boolean contains(E item);
  public void    addFirst(E item);
  public E       removeFirst() throws NoSuchElementException;
  public void    print();
}
```

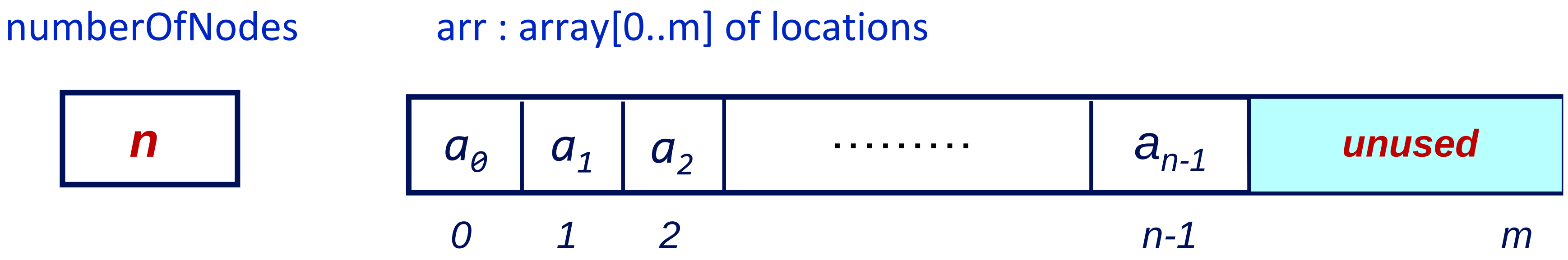
- The **ListInterface** above defines the operations (methods) we would like to have in a List ADT
- The operations shown here are just a small sample. An actual List ADT usually contains more operations.

- We will examine 2 implementations of list ADT, both using the **ListInterface** shown in the previous slide

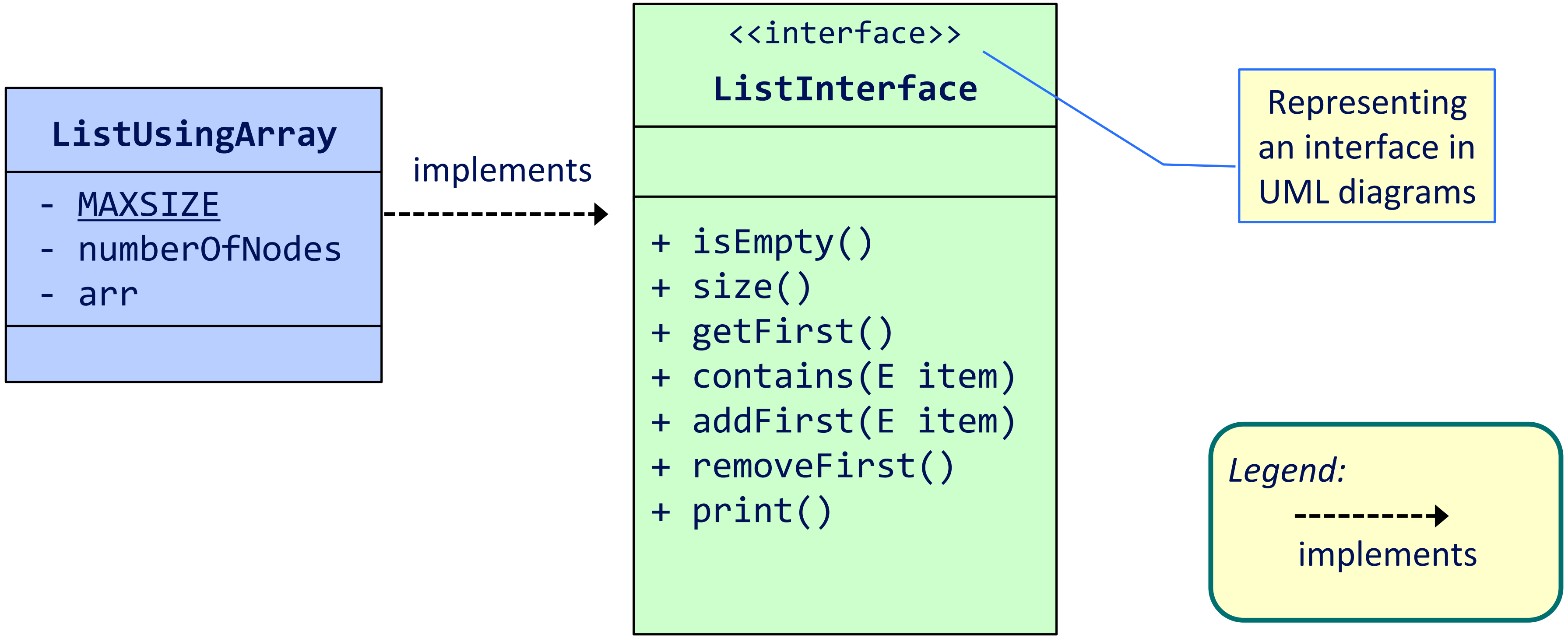


Fixed-size list

- This is a straight-forward approach
  - Use Java array of a sequence of *n* elements



- We now create a class ListUsingArray as an implementation of the interface ListInterface (a user-defined interface)



# 2. List Implementation: Array (3/9)

**ListUsingArray.java**

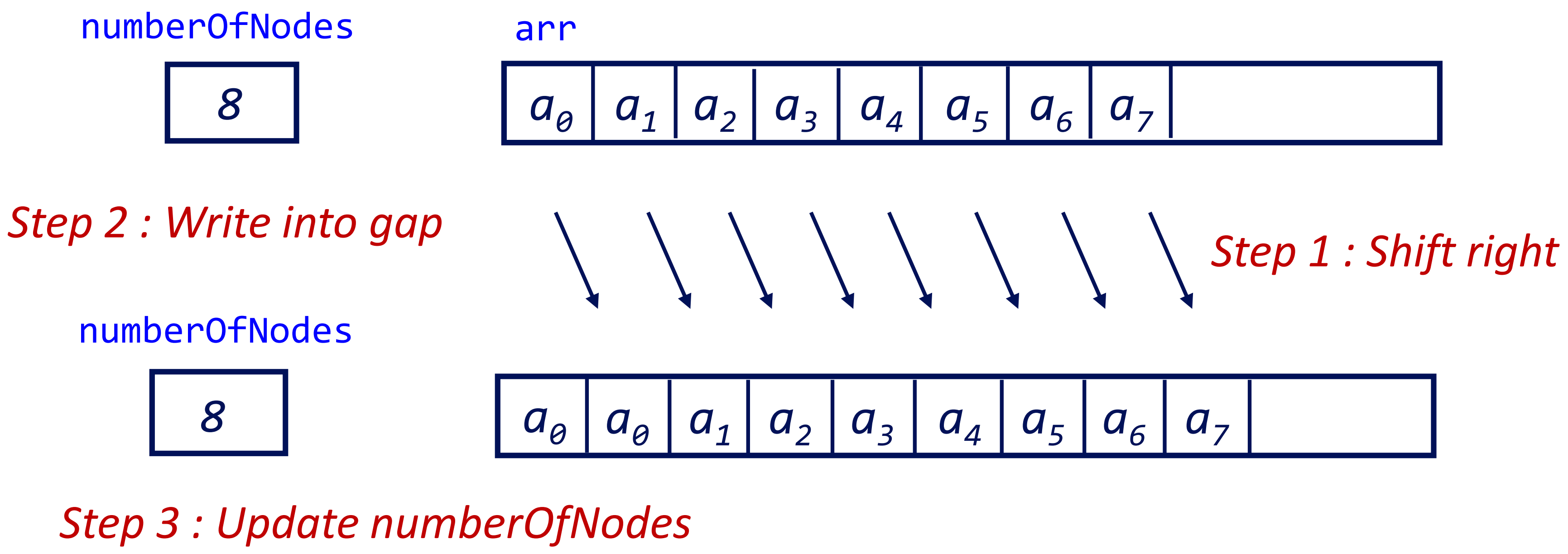
```java
import java.util.*;

class ListUsingArray<E> implements ListInterface<E> {
  private static final int MAXSIZE = 1000;
  private int numberOfNodes = 0;
  private E[] arr = (E[]) new Object[MAXSIZE];

  public boolean isEmpty() {
    return numberOfNodes == 0;
  }
  public int size() {
    return numberOfNodes;
  }
  public E getFirst() throws NoSuchElementException {
      if (numberOfNodes == 0) {
          throw new NoSuchElementException("can't get
                                      from an empty list");
      } else {
          return arr[0];
      }
  }
```

**ListUsingArray.java**

```java
    public boolean contains(E item) {
      for (int i = 0; i < numberOfNodes; i++) {
        if (arr[i].equals(item)) {
          return true;
        }
      }

      return false;
    }
```

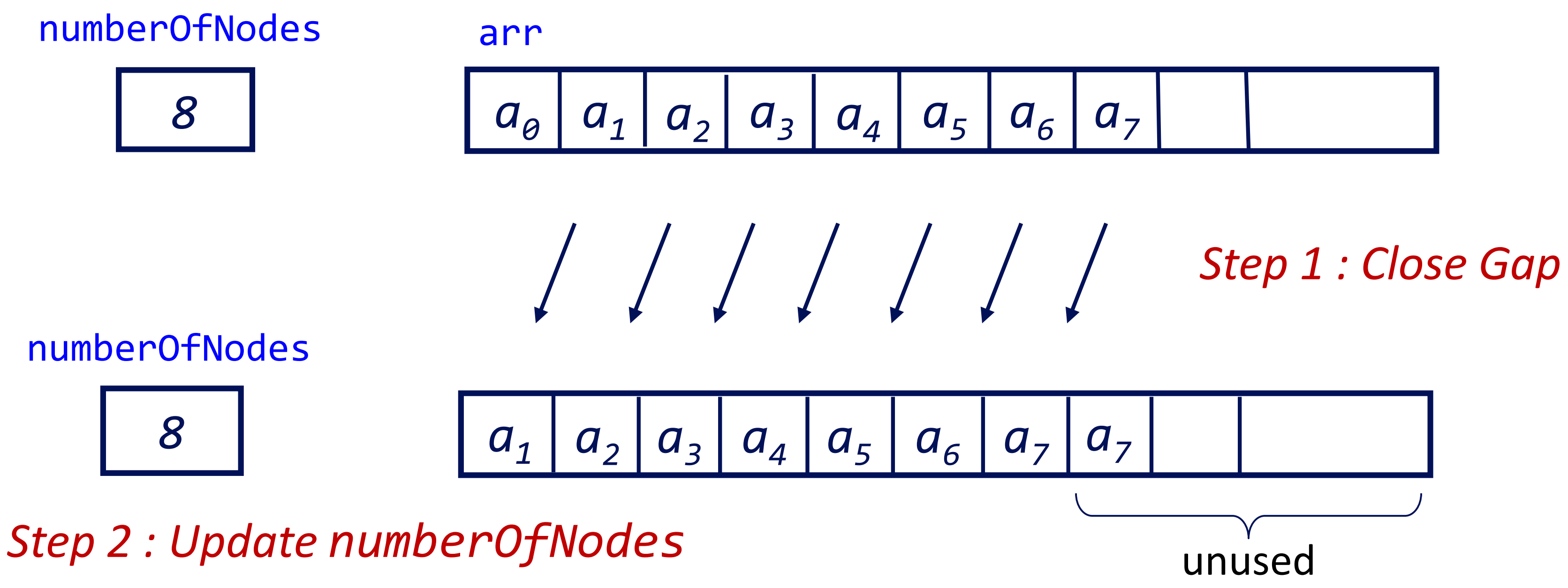
- For insertion into first position, need to shift "right" (starting from the last element) to create room

Example: `addFirst("it")`



- For deletion of first element, need to shift "left" (starting from the first element) to close gap

Example: `removeFirst()`




Need to maintain *numberOfNodes* so that program would not access beyond the valid data.

## 2. List Implementation: Array (6/9)

**ListUsingArray.java**

```java
public void addFirst(E item) throws IndexOutOfBoundsException {
  if (numberOfNodes == MAXSIZE) {
    throw new IndexOutOfBoundsException("insufficient space for add");
  }
  for (int i = numberOfNodes - 1; i >= 0; i--) {
    arr[i + 1] = arr[i]; // to shift elements to the right
  }
  arr[0] = item;
  numberOfNodes++; // update num_nodes
}
public E removeFirst() throws NoSuchElementException {
  if (numberOfNodes == 0) {
    throw new NoSuchElementException("can't remove from an empty list");
  } else {
    E temp = arr[0];
    for (int i = 0; i < numberOfNodes - 1; i++) {
      arr[i] = arr[i + 1]; // to shift elements to the left
    }
    numberOfNodes--; // update num_nodes
    return tmp;
  }
}
```

## 2. Testing Array Implementation (7/9)

**TestListUsingArray.java**

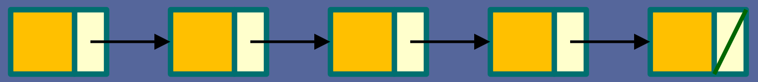
```java
import java.util.*;

public class TestListUsingArray {
  public static void main(String[] args) throws NoSuchElementException {
    ListUsingArray<String> list = new ListUsingArray<String>();
    list.addFirst("aaa");
    list.addFirst("bbb");
    list.addFirst("ccc");
    list.print();

    System.out.println("Testing removal");
    list.removeFirst();
    list.print();

    if (list.contains("aaa")) {
      list.addFirst("xxxx");
    }
    list.print();
  }
}
```

```
List is:
ccc, bbb, aaa.
Testing removal
List is:
bbb, aaa.
List is:
xxxx, bbb, aaa.
```

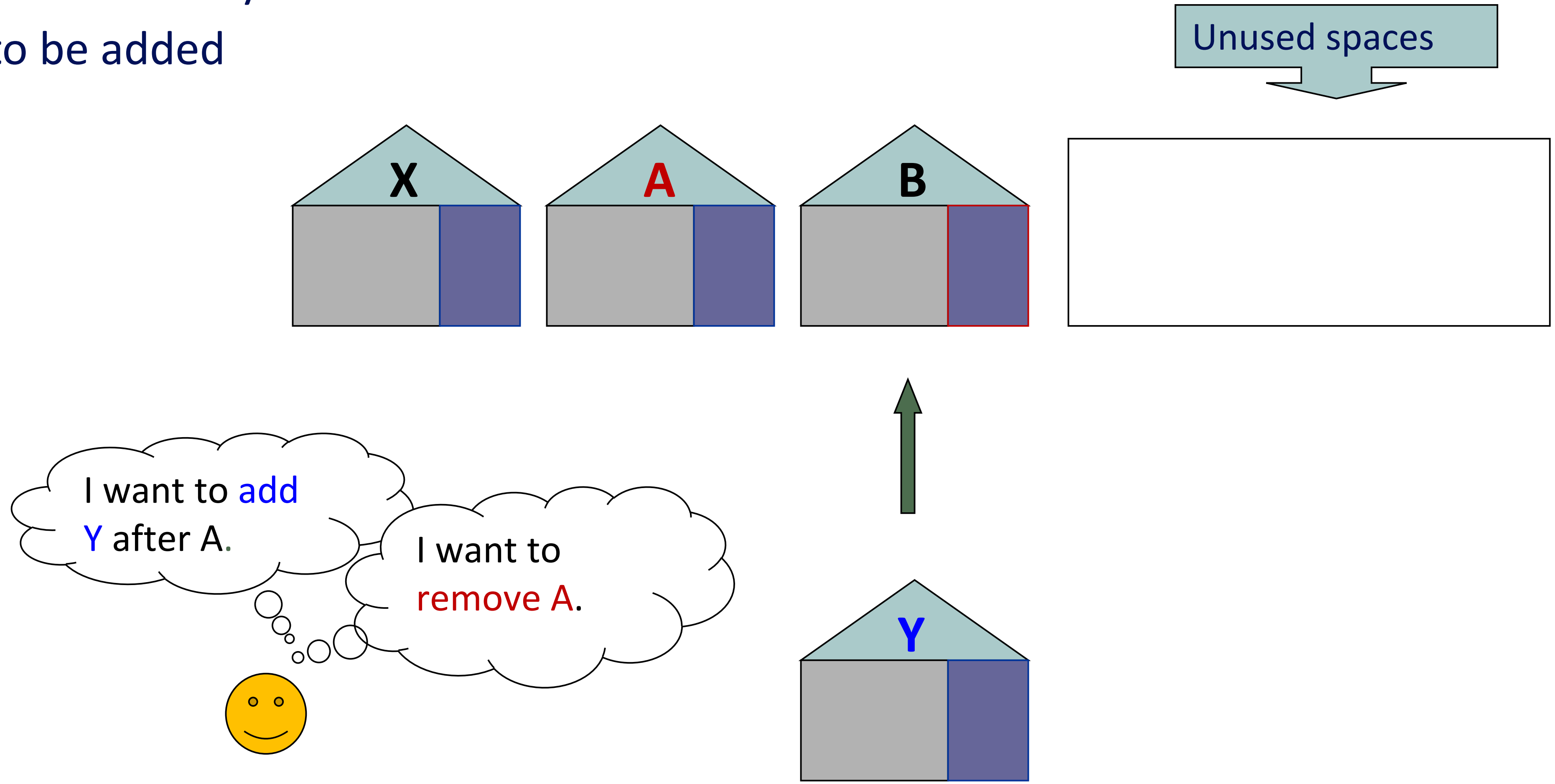
- Question: Time Efficiency?
  - Retrieval: getFirst()
    - Always fast with 1 read operation
  - Insertion: addFirst(E item)
    - Shifting of all *n* items – bad!
  - Insertion: add(int index, E item)
    - Inserting into the specified position (not shown in ListUsingArray.java)
      - Best case: No shifting of items (add to the last place)
      - Worst case: Shifting of all items (add to the first place)
  - Deletion: removeFirst(E item)
    - Shifting of all *n* items – bad!
  - Deletion: remove(int index)
    - Delete the item at the specified position (not shown in ListUsingArray.java)
      - Best case: No shifting of items (delete the last item)
      - Worst case: Shifting of all items (delete the first item)

- Question: What is the Space Efficiency?
  - Size of array collection limited by MAXSIZE
  - Problems
    - We don’t always know the maximum size ahead of time
    - If MAXSIZE is too liberal, unused space is wasted
    - If MAXSIZE is too conservative, easy to run out of space
- Idea: make MAXSIZE a variable, and create/copy to a larger array whenever the array runs out of space
  - No more limits on size
  - But copying overhead is still a problem
- When to use such a list?
  - For a fixed-size list, an array is good enough!
  - For a variable-size list, where dynamic operations such as insertion/deletion are common, an array is a poor choice; better alternative – **Linked List**

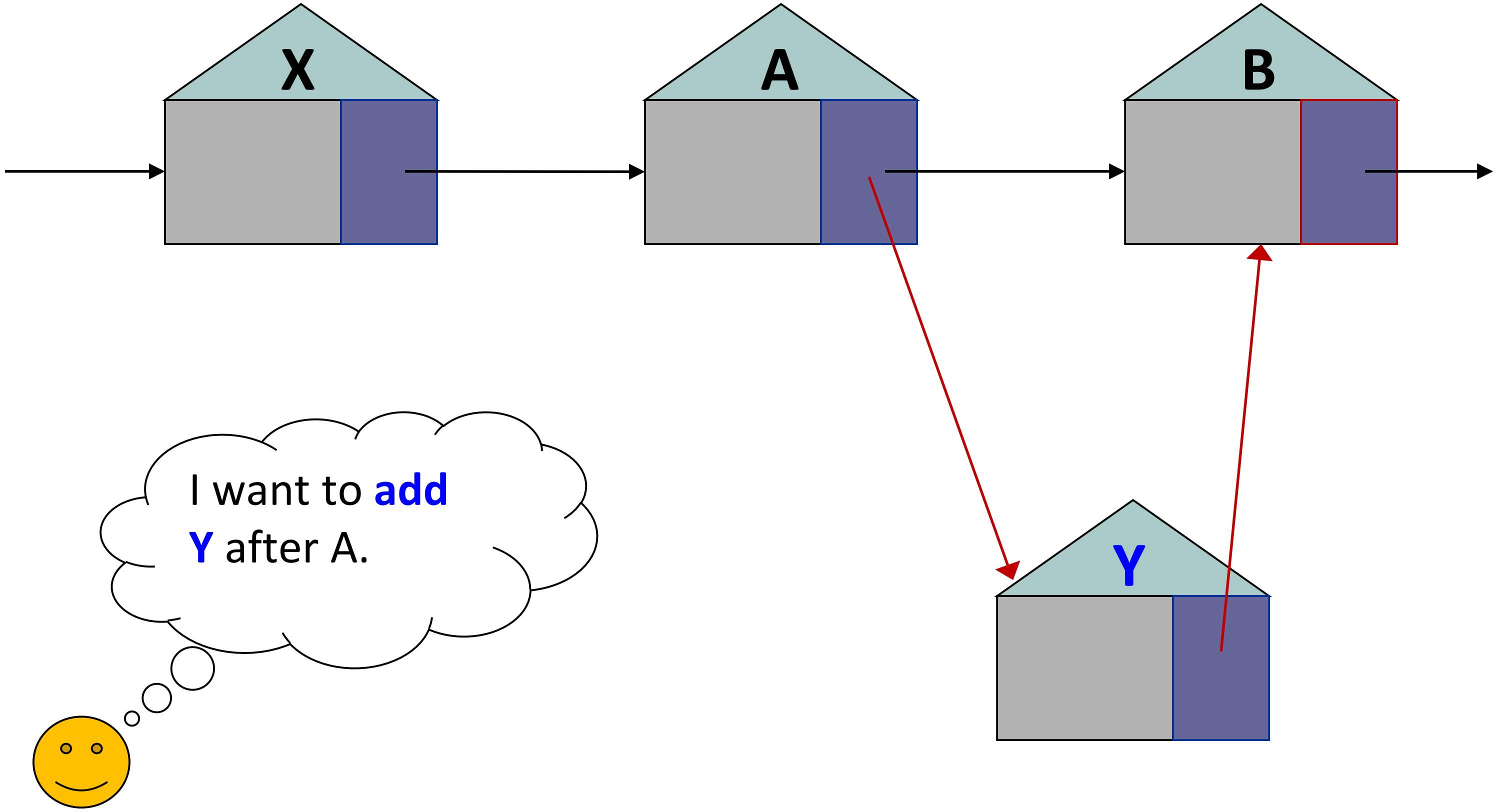
Variable-size list



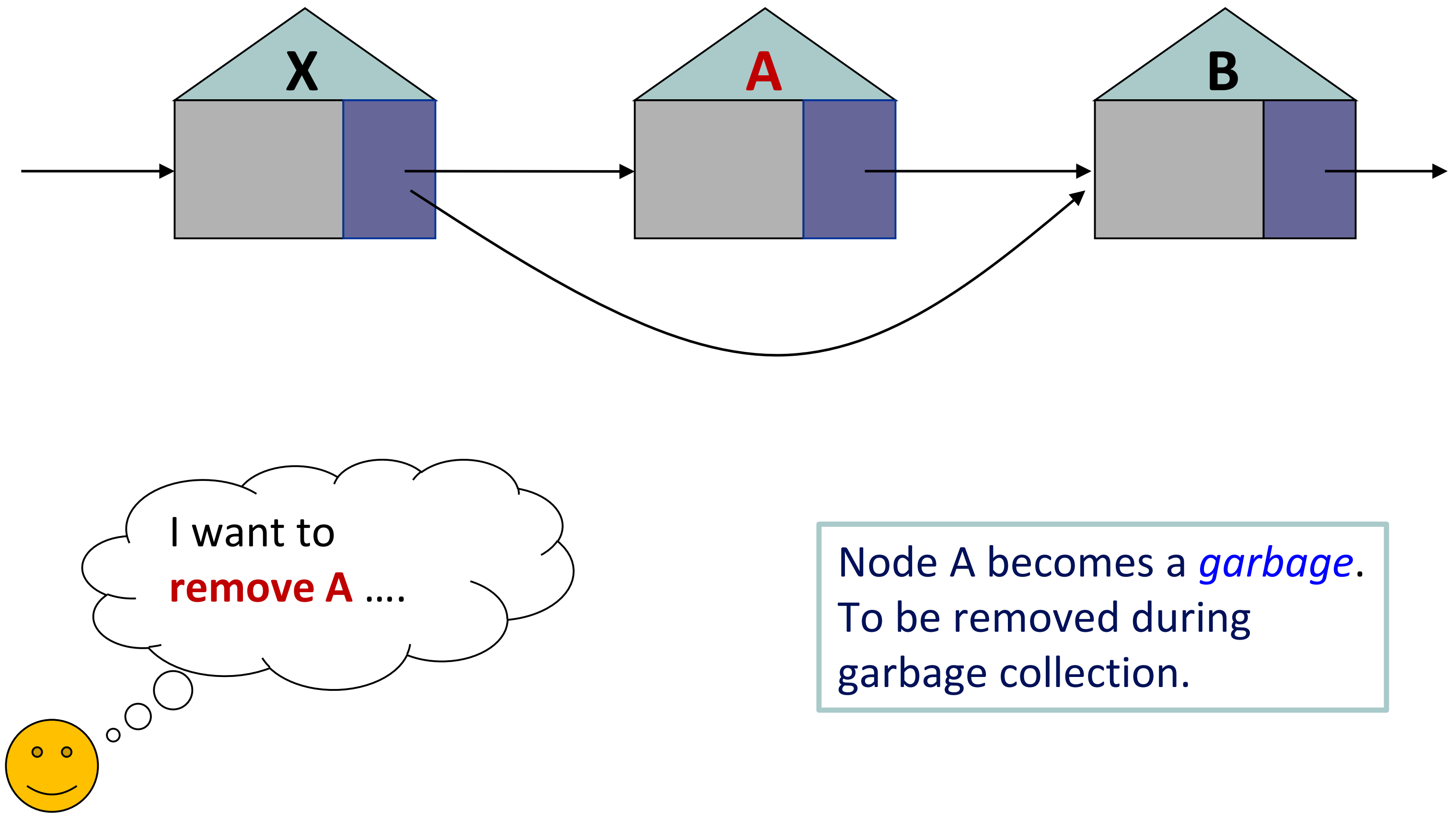
- Recap when using an array...
  - X, A, B are elements of an array
  - Y is new element to be added



- Now, we see the (add) action with linked list…
  - X, A, B are nodes of a linked list
  - Y is new node to be added

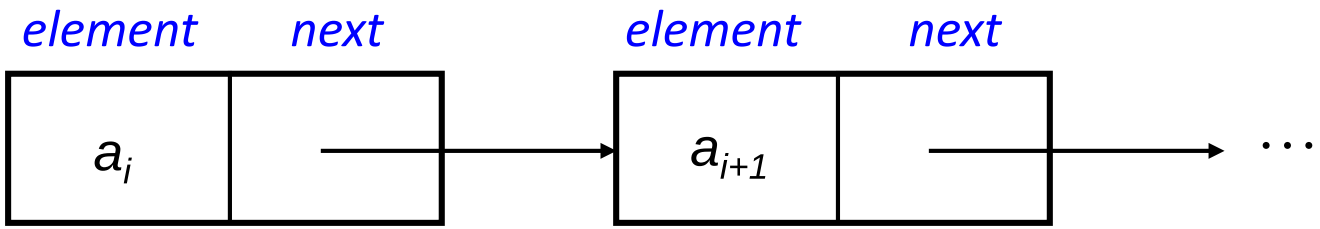


- Now, we see the (remove) action with linked list…



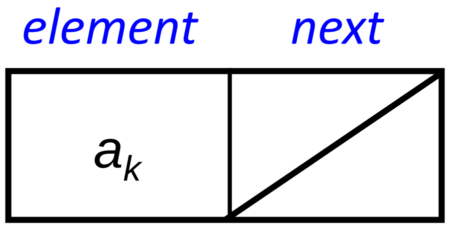

Node A becomes a *garbage*.
To be removed during
garbage collection.

- Idea
  - Each element in the list is stored in a *node*, which also contains a next pointer
  - Allow elements in the list to occupy *non-contiguous* memory
  - Order the nodes by associating each with its neighbour(s)




This is one node of the collection…

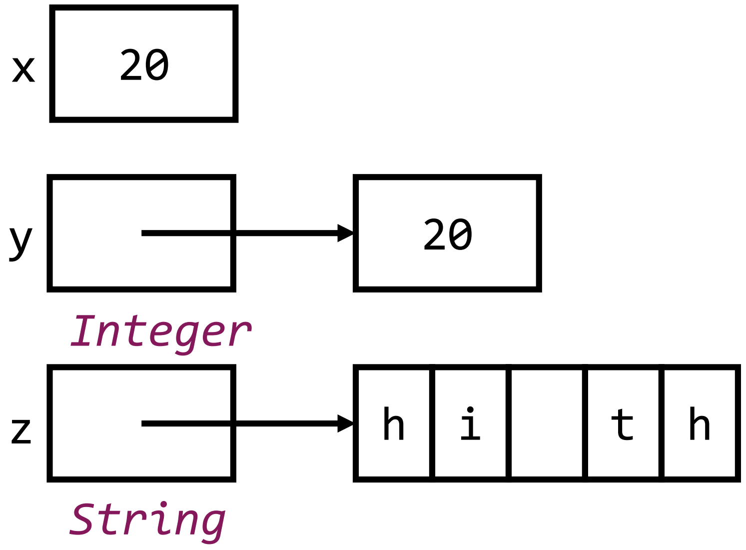
… and this one comes after it in the collection (most likely not occupying contiguous memory that is next to the previous node).




Next pointer of this node is “null”, i.e. it has no next neighbour.

- Recap: Object References (1/2)
  - Note the difference between primitive data types and reference data types

```
int x = 20;

Integer y = new Integer(20);

String z = new String("hi th");
```



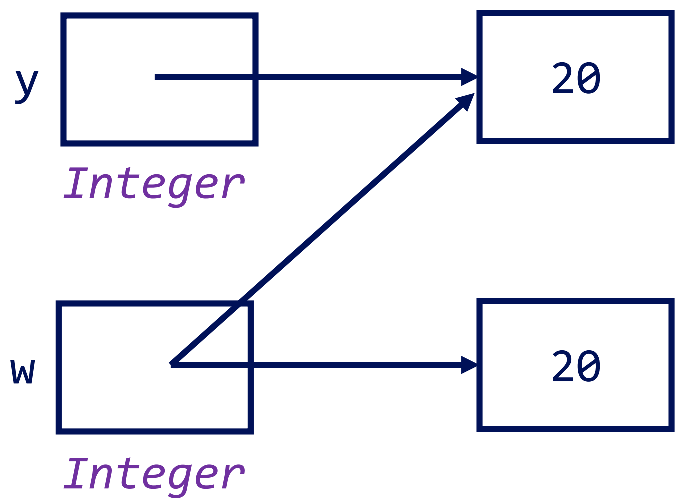

- An instance (object) of a class only comes into existence (constructed) when the new operator is applied
- A reference variable only contains a reference or pointer to an object.

## Recap: Object References (2/2)

- Look at it in more details:

```
Integer y =  new Integer(20);
Integer w;
w = new Integer(20);
if (w == y) {
  System.out.println("1. w == y");
}
w = y;
if (w == y)
  System.out.println("2. w == y");
```

Output: 2. w == y

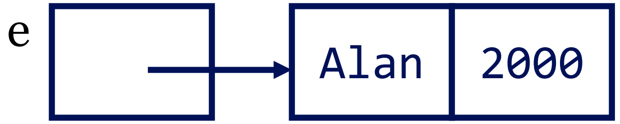


- Quiz: Which is the right representation of e?

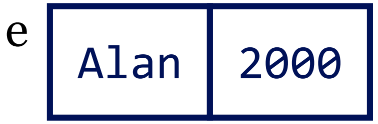
```
class Employee {
  private String name;
  private int salary;
  // etc.
}
```

```
Employee e = new Employee("Alan", 2000);
```

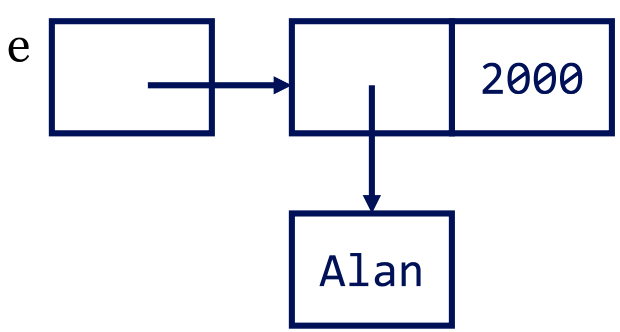
(A)



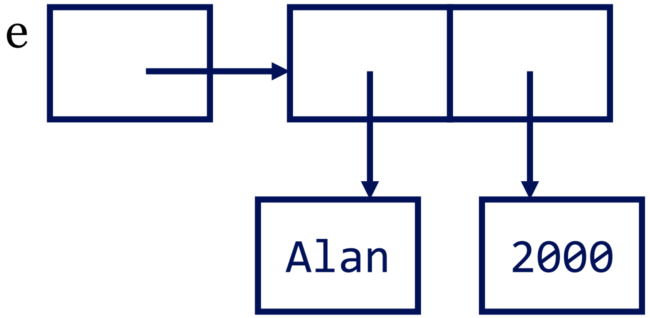

(B)




(C)



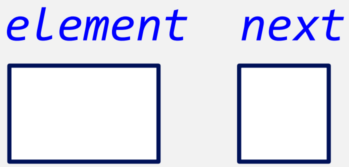

(D)



## 3.3 ListNode (using generic)

**ListNode.java**

```java
class ListNode<E> {

  /* Data attributes */
  private E element;
  private ListNode<E> next;

  /* Constructors */
  public ListNode(E item) {
    this(item, null);
  }

  public ListNode(E item, ListNode<E> node) {
    element = item;
    next = node;
  }
```



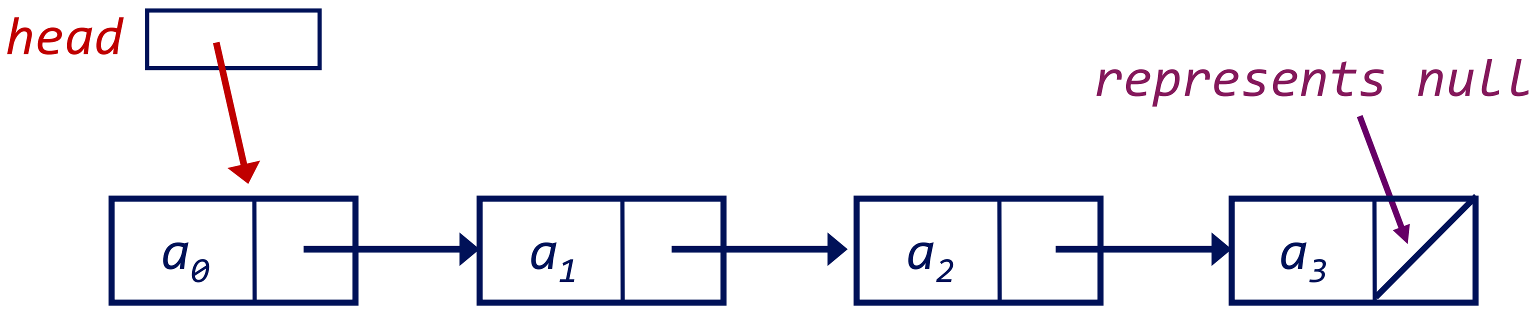

**ListNode.java**

```java
  /* Get the next ListNode */
  public ListNode<E> getNext() {
    return next;
  }

  /* Get the element of the ListNode */
  public E getElement() {
    return element;
  }

  /* Set the next reference */
  public void setNext(ListNode<E> node) {
    next = node;
  }
}
```

**Note** – You may need to refer to it later when we study the different variants of linked list.

- For a sequence of 4 items < $a_0$, $a_1$, $a_2$, $a_3$ >

We need a *head* to indicate where the first node is. From the *head* we can get to the rest.

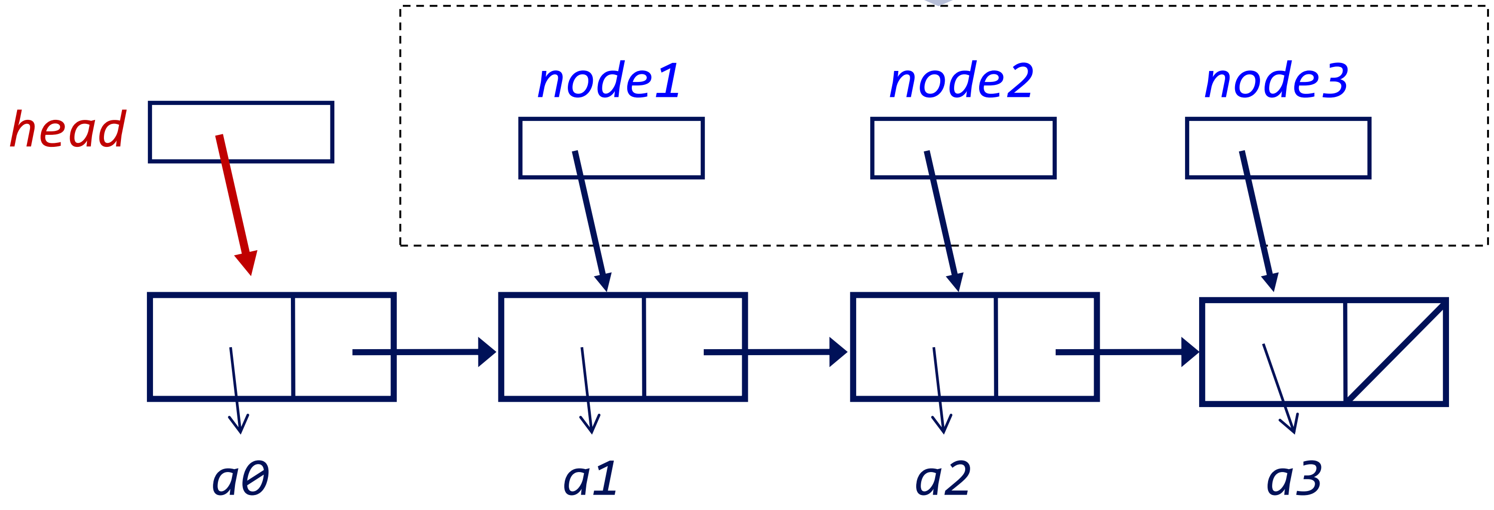


- For a sequence of 4 items $< a_0, a_1, a_2, a_3 >$

```
ListNode<String> node3  = new ListNode<String>("a3", null);
ListNode<String> node2  = new ListNode<String>("a2", node3);
ListNode<String> node1  = new ListNode<String>("a1", node2);
ListNode<String> head   = new ListNode<String>("a0", node1);
```

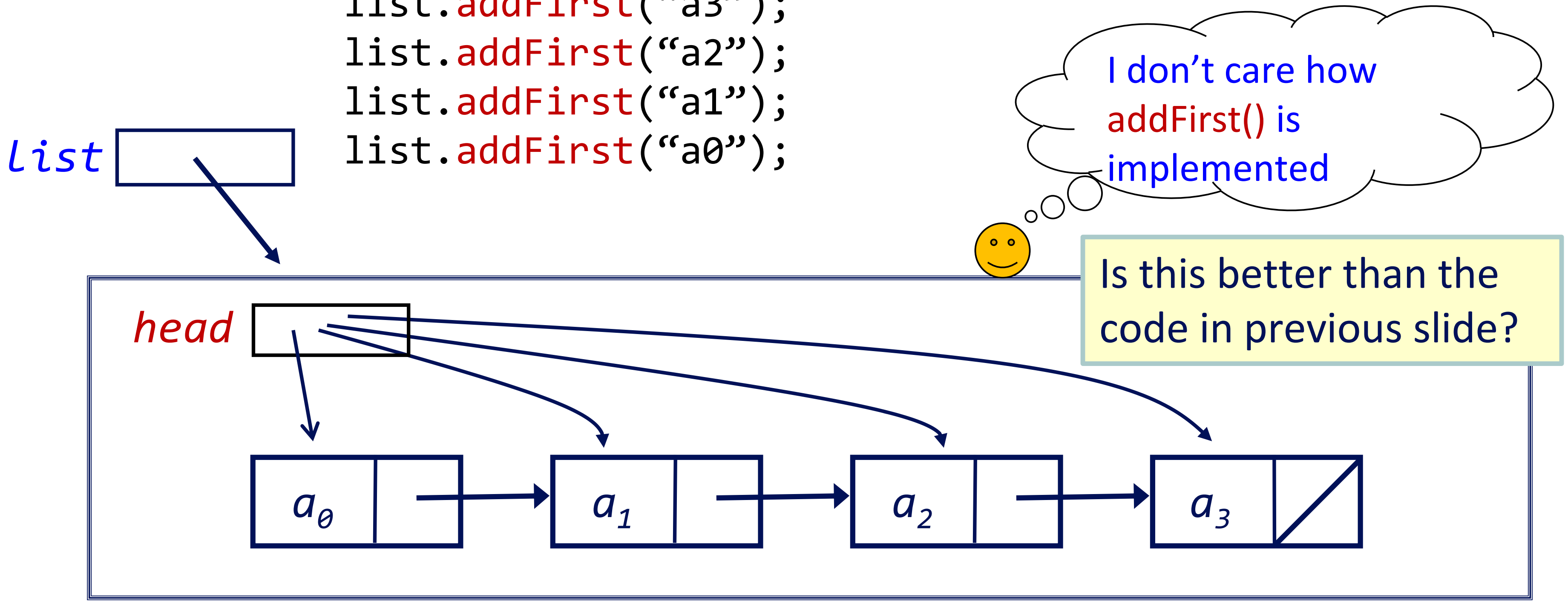
Can the code be rewritten without using these object references node1, node2, node3?

No longer needed after list is built.



- Alternatively we can form the linked list as follows:
  - For a sequence of 4 items < $a_0$, $a_1$, $a_2$, $a_3$ >, we can build as follows:

```
LinkedList<String> list = new LinkedList<String>();
list.addFirst(“a3”);
list.addFirst(“a2”);
list.addFirst(“a1”);
list.addFirst(“a0”);
```




Is this better than the code in previous slide?

**BasicLinkedList.java**

```java
import java.util.*;

class BasicLinkedList<E> implements
ListInterface<E> {

  private ListNode<E> head = null;
  private int numberOfNodes = 0;

  public boolean isEmpty() {
      return (numberOfNodes == 0);
  }

  public int size() {
      return numberOfNodes;
  }
```

**BasicLinkedList.java**

```java
  public E getFirst() throws NoSuchElementException {
    if (head == null) {
      throw new NoSuchElementException("can't get
                                   from an empty list");
    } else {
      return head.getElement();
    }
  }

  public boolean contains(E item) {
    for (ListNode<E> n = head; n != null; n = n.getNext()) {
      if (n.getElement().equals(item)) {
        return true;
      }
    }

   return false;
  }
```

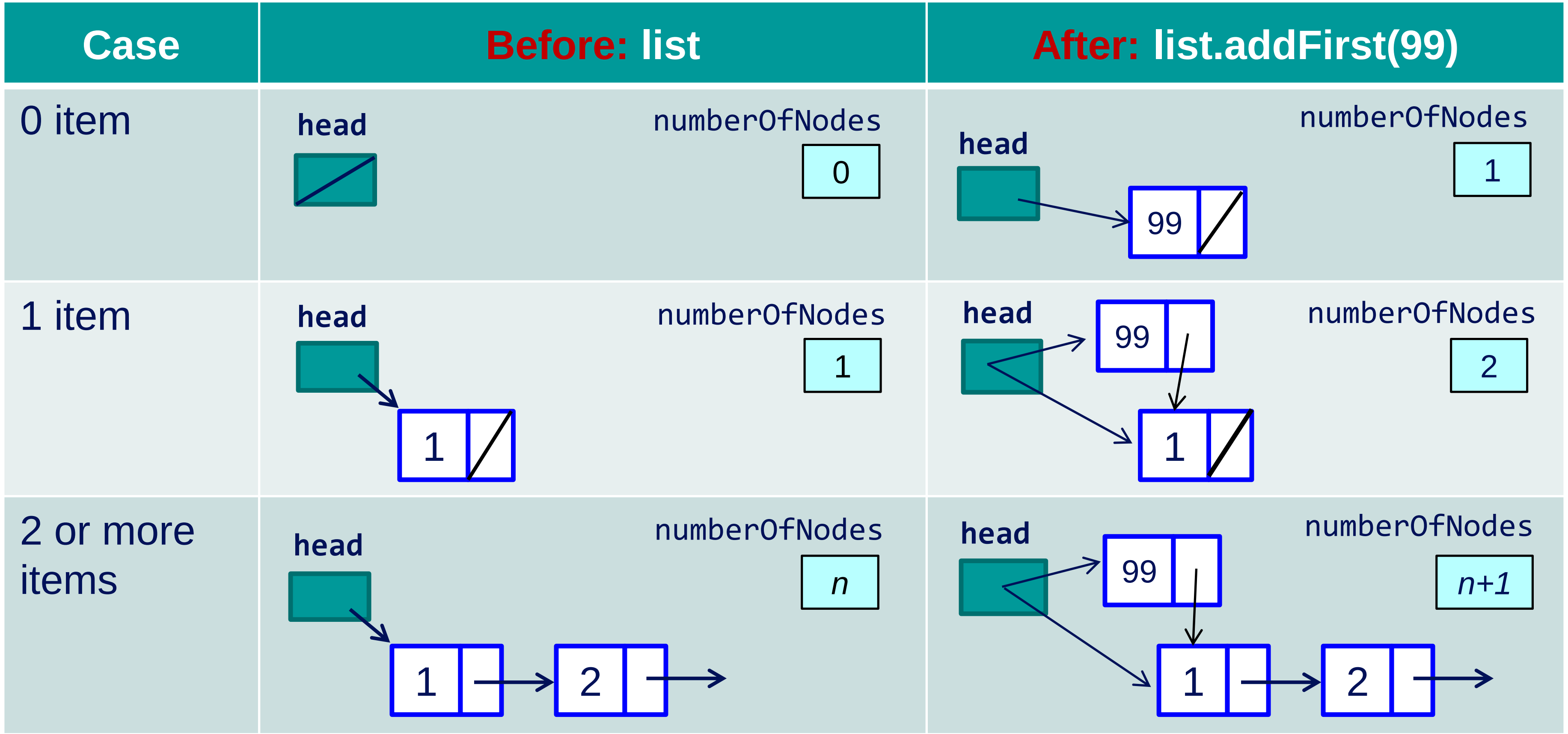
getElement() and getNext() are methods in ListNode class

Using ListNode to define BasicLinkedList

**BasicLinkedList.java**

```java
public void addFirst(E item) {
  head = new ListNode<E>(item, head);
  numberOfNodes++;
}

public E removeFirst() throws NoSuchElementException {
  ListNode<E> node;
  if (head == null)  {
    throw new NoSuchElementException("can't remove from empty list");
  } else {
    node = head;
    head = head.getNext();
    numberOfNodes--;
    return node.getElement();
  }
}
```

The adding and removal of first element




The addFirst() method

```java
public void addFirst(E item) {
  head = new ListNode<E>(item, head);
  numberOfNodes++;
}
```

| Case | Before: list | After: list.addFirst(99) |
|---|---|---|
| 0 item | head; numberOfNodes 0 | Can‘t remove |
| 1 item | head; numberOfNodes 1; 1 | head; ln; numberOfNodes 0; 1 |
| 2 or more items | head; numberOfNodes *n*; 1, 2 | head; ln; numberOfNodes *n-1*; 1, 2 |

The removeFirst() method

```
public E removeFirst() throws NoSuchElementException {
  ListNode<E> ln;
  if (head == null) {
      throw new NoSuchElementException("can't remove");
  } else {
      ln = head;
      head = head.getNext();
      numberOfNodes--;
      return ln.getElement();
  }
}
```

**BasicLinkedList.java**

```java
public void print() throws NoSuchElementException {
  if (head == null) {
    throw new NoSuchElementException("Nothing to print...");
  }

  ListNode<E> node = head;
  System.out.print("List is: " + node.getElement());
  for (int i = 1; i < numberOfNodes; i++) {
    node = node.getNext();
    System.out.print(", " + node.getElement());
  }

  System.out.println(".");
}
```

Printing of the linked list

TestBasicLinkedList1.java

```java
import java.util.*;

public class TestBasicLinkedList1 {
  public static void main(String[] args) throws NoSuchElementException {
    BasicLinkedList<String> list = new BasicLinkedList<String>();
    list.addFirst("aaa");
    list.addFirst("bbb");
    list.addFirst("ccc");
    list.print();

    System.out.println("Testing removal");
    list.removeFirst();
    list.print();

    if (list.contains("aaa")) {
      list.addFirst("xxxx");
    }
    list.print();
  }
}
```

```
List is: ccc, bbb, aaa.
Testing removal
List is: bbb, aaa.
List is: xxxx, bbb, aaa.
```

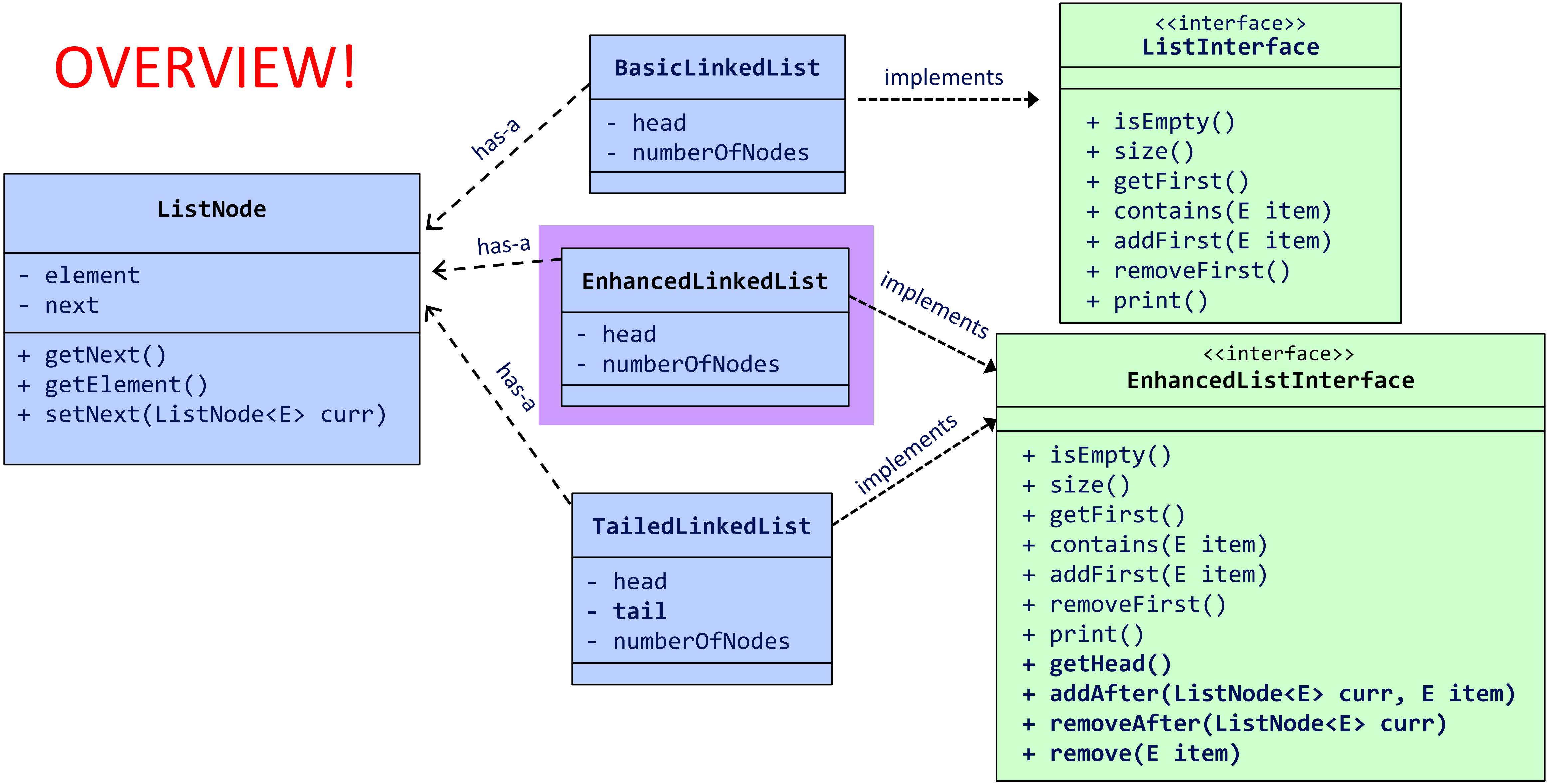
**TestBasicLinkedList2.java**

```java
import java.util.*;

public class TestBasicLinkedList2 {
  public static void main(String[] args) throws NoSuchElementException {
    BasicLinkedList<Integer> list = new BasicLinkedList<Integer>();

    list.addFirst(34);
    list.addFirst(12);
    list.addFirst(9);
    list.print();

    System.out.println("Testing removal");
    list.removeFirst();
    list.print();
  }
}
```

```
List is: 9, 12, 34.
Testing removal
List is: 12, 34.
```

Exploring variants of linked list

OVERVIEW!


BasicLinkedList
- head
- numberOfNodes
implements
<<interface>>
ListInterface
+ isEmpty()
+ size()
+ getFirst()
+ contains(E item)
+ addFirst(E item)
+ removeFirst()
+ print()
has-a
ListNode
- element
- next
+ getNext()
+ getElement()
+ setNext(ListNode<E> curr)
has-a
EnhancedLinkedList
- head
- numberOfNodes
implements
has-a
<<interface>>
EnhancedListInterface
+ isEmpty()
+ size()
+ getFirst()
+ contains(E item)
+ addFirst(E item)
+ removeFirst()
+ print()
+ getHead()
+ addAfter(ListNode<E> curr, E item)
+ removeAfter(ListNode<E> curr)
+ remove(E item)
implements
TailedLinkedList
- head
- tail
- numberOfNodes

- We explore different implementations of Linked List
  - Basic Linked List, Tailed Linked List, Circular Linked List, Doubly Linked List, etc.
- When nodes are to be inserted to the middle of the linked list, BasicLinkedList (BLL) is not good enough.
- For example, BLL offers only insertion at the front of the list. If the items in the list must always be sorted according to some key values, then we must be able to insert at the right place.
- We will enhance BLL to include some additional methods. We shall call this Enhanced Linked List (ELL).
  - (Note: We could have made ELL a subclass of BLL, but here we will create ELL from scratch instead.)

**EnhancedListInterface.java**

```java
import java.util.*;

public interface EnhancedListInterface<E> {

  public boolean isEmpty();
  public int size();
  public E getFirst() throws NoSuchElementException;
  public boolean contains(E item);
  public void addFirst(E item);
  public E removeFirst() throws NoSuchElementException;
  public void print();

  public ListNode<E> getHead();
  public void addAfter(ListNode<E> current, E item);
  public E removeAfter(ListNode<E> current) throws NoSuchElementException;
  public E remove(E item) throws NoSuchElementException;
}
```

We use a new interface

New

**EnhancedLinkedList.java**

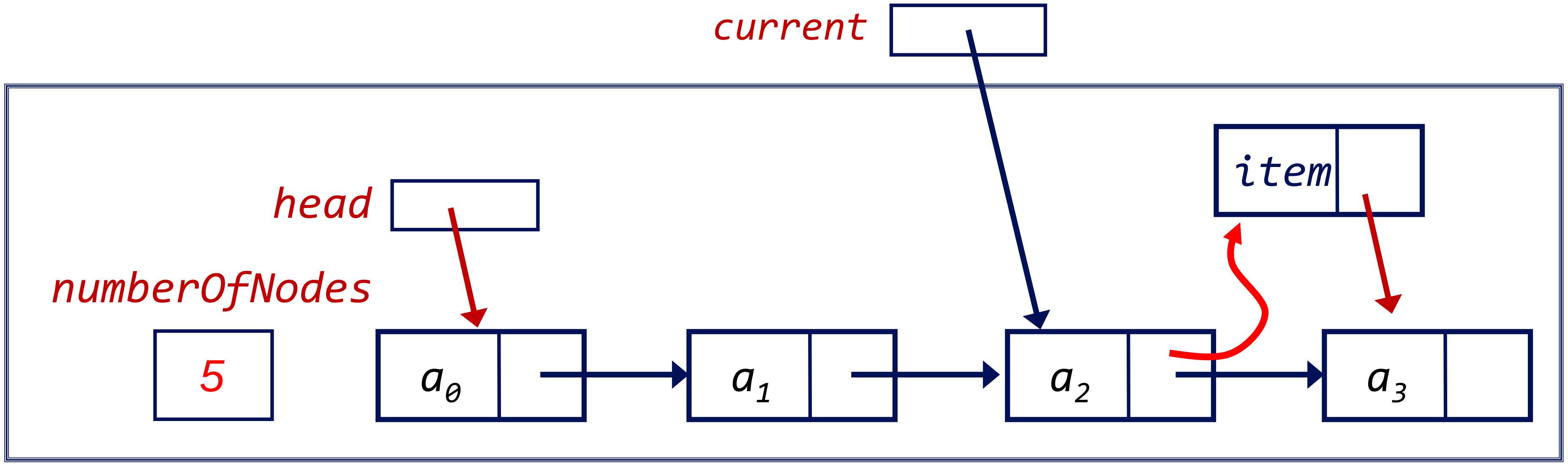
```
import java.util.*;

class EnhancedLinkedList<E> implements EnhancedListInterface<E> {

  private ListNode<E> head = null;
  private int numberOfNodes = 0;

  public boolean isEmpty() { return (numberOfNodes == 0); }
  public int size() { return numberOfNodes; }
  public E getFirst() { ... }
  public boolean contains(E item) { ... }
  public void addFirst(E item) { ... }
  public E removeFirst() throws NoSuchElementException { ... };
  public void print() throws NoSuchElementException { ... };

  public ListNode<E> getHead() { return head; }
```

Same as in BasicLinkedList.java

*To continue on next slide*

EnhancedLinkedList.java

```java
public void addAfter(ListNode<E> current, E item) {
  if (current != null) {
      current.setNext(new ListNode<E>(item,current.getNext()));
  } else { // insert item at front
      head = new ListNode<E> (item, head);
  }
  numberOfNodes++;
}
```

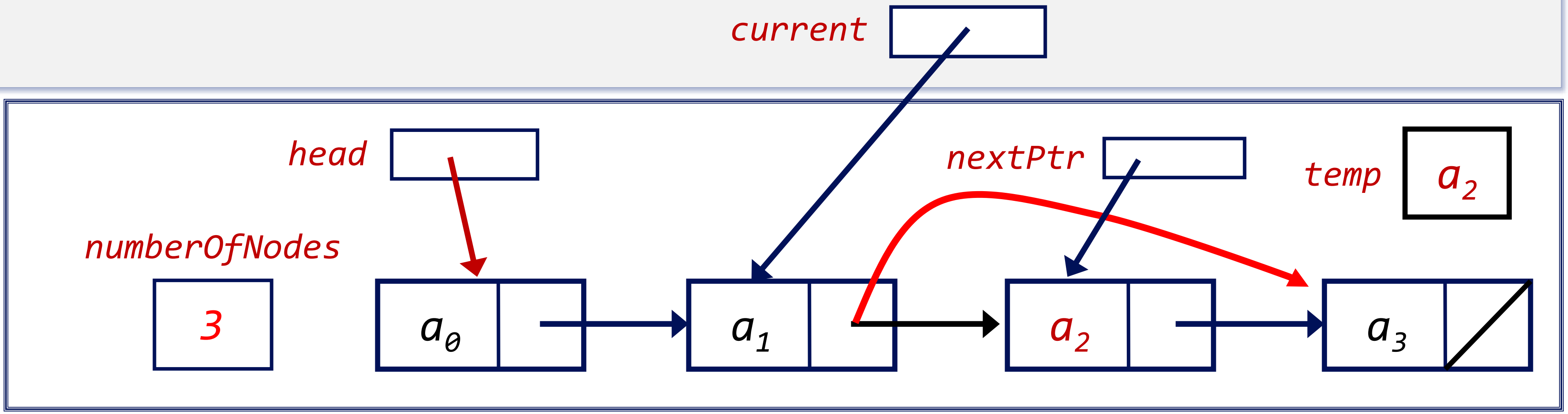


**EnhancedLinkedList.java**

```java
public E removeAfter(ListNode<E> current) throws NoSuchElementException {
  E temp;
  if (current != null) {
    ListNode<E> nextPtr = current.getNext();
    if (nextPtr != null) {
      temp = nextPtr.getElement();
      current.setNext(nextPtr.getNext());
      numberOfNodes--;
      return temp;
    } else {
      throw new NoSuchElementException("No next node to remove");
    }
  } else {  // if current is null, assume we want to remove head
    if (head != null) {
      temp = head.getElement();
      head = head.getNext();
      numberOfNodes--;
      return temp;
    } else {
      throw new NoSuchElementException("No next node to remove");
    }
  }
}
```

**EnhancedLinkedList.java**

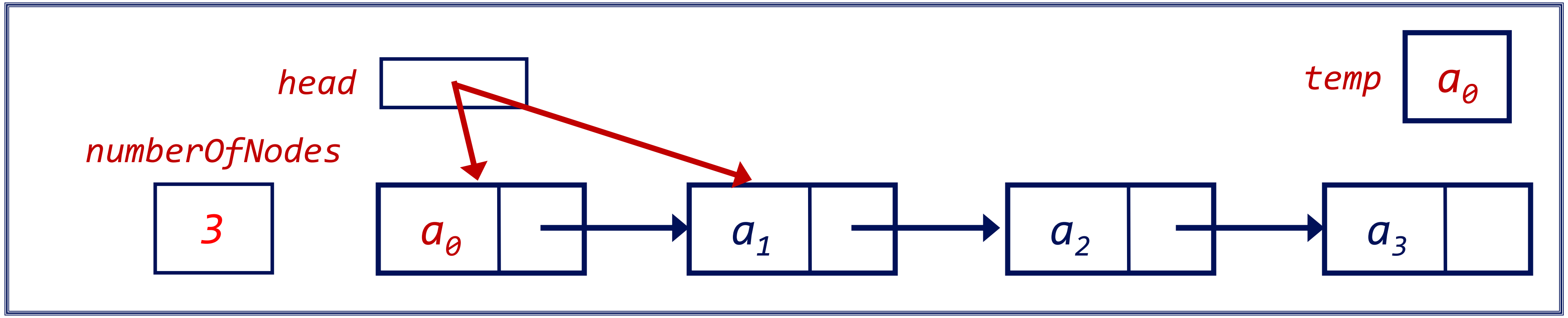
```
public E removeAfter(ListNode<E> current) throws NoSuchElementException {
  E temp;
  if (current != null) {
    ListNode<E> nextPtr = current.getNext();
    if (nextPtr != null) {
      temp = nextPtr.getElement();
      current.setNext(nextPtr.getNext());
      numberOfNodes--;
      return temp;
    } else {
      throw new NoSuchElementException("No next node to remove");
    }
  } else {…}
}
```



## EnhancedLinkedList.java

```
public E removeAfter(ListNode<E> current) throws NoSuchElementException {
  E temp;
  if (current != null) {
    …
  } else {  // if current is null, assume we want to remove head
    if (head != null) {
      temp = head.getElement();
      head = head.getNext();
      numberOfNodes--;
      return temp;
    } else {
      throw new NoSuchElementException("No next node to remove");
    }
  }
}
```

*current* null



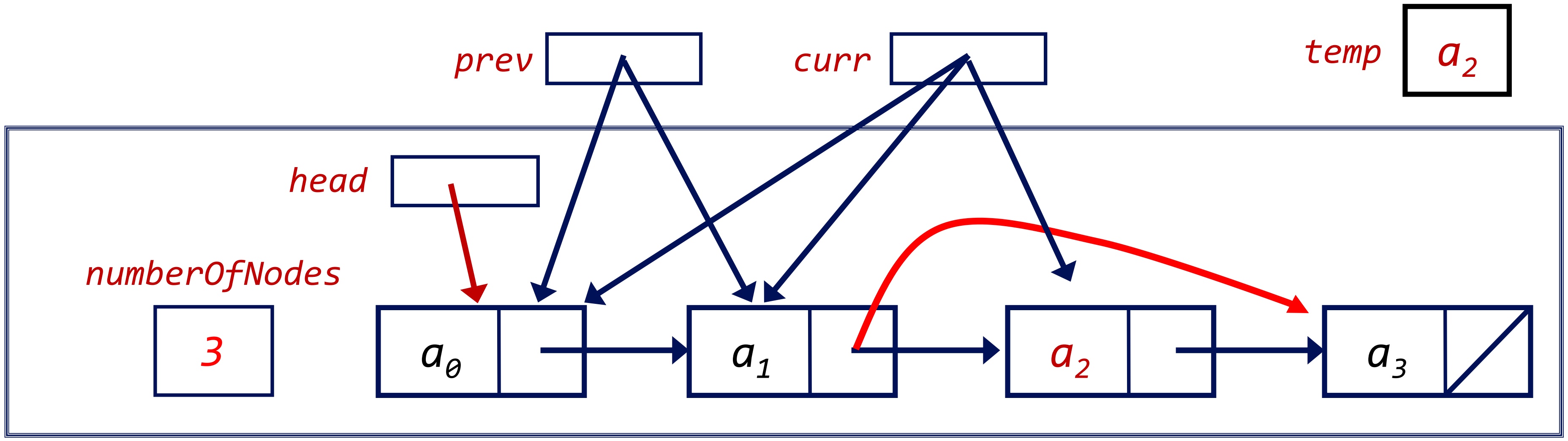
- remove(E item)
  - Search for item in list
  - Re-using removeAfter() method

**EnhancedLinkedList.java**

```
public E removeAfter(E item) throws NoSuchElementException {

    // Write your code below...
    // Should make use of removeAfter() method.




}
```

**EnhancedLinkedList.java**

```
public E removeAfter(E item) throws NoSuchElementException {




}
```



TestEnhancedLinkedList.java

```java
import java.util.*;

public class TestEnhancedLinkedList {
  public static void main(String[] args) throws NoSuchElementException {

      EnhancedLinkedList<String> list = new EnhancedLinkedList<String>();
      System.out.println("Part 1");
      list.addFirst("aaa");
      list.addFirst("bbb");
      list.addFirst("ccc");
      list.print();

      System.out.println();
      System.out.println("Part 2");
      ListNode<String> current = list.getHead();
      list.addAfter(current, "xxx");
      list.addAfter(current, "yyy");
      list.print();
```

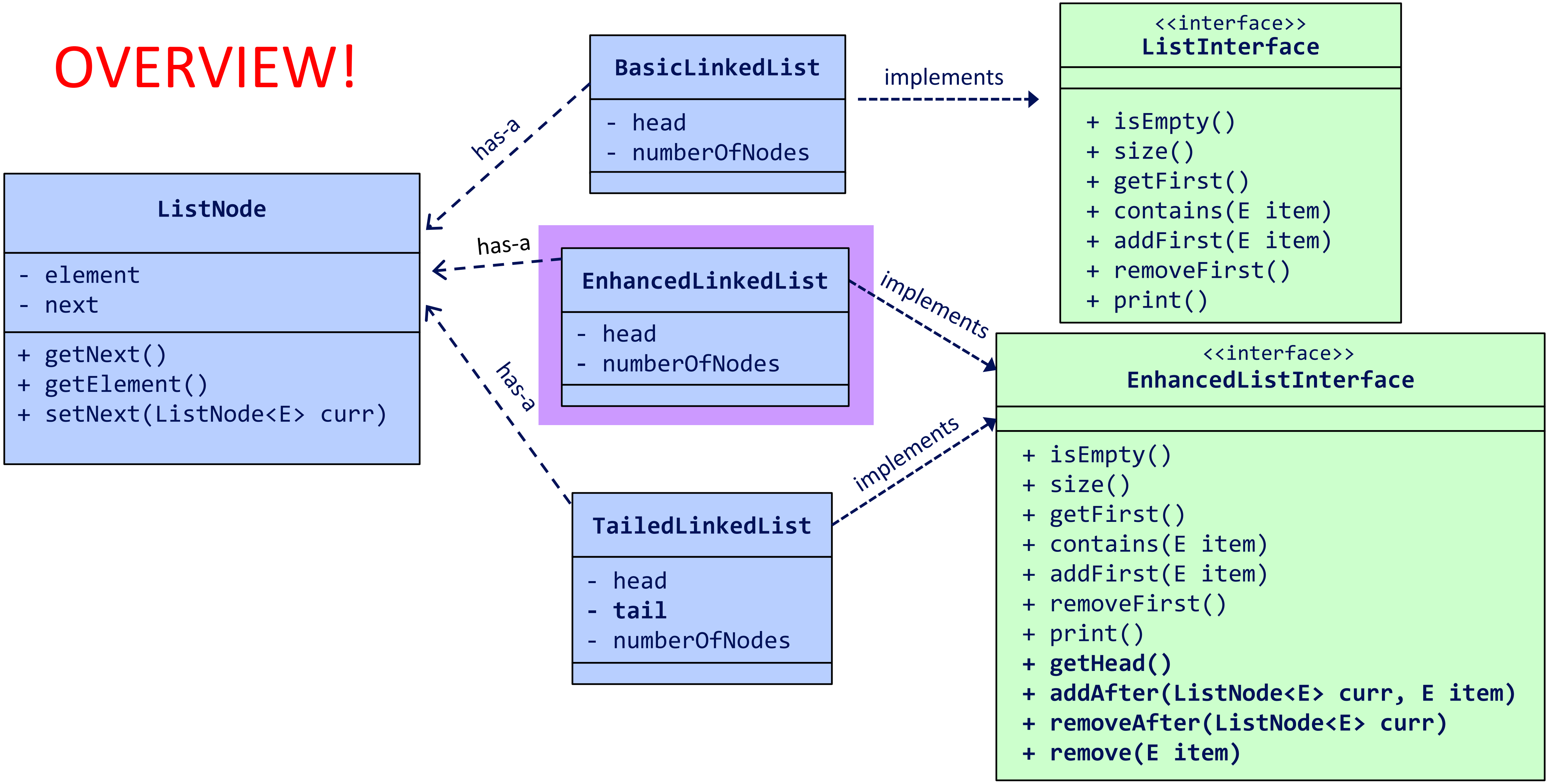
```
Part 1
List is: ccc, bbb, aaa.

Part 2
List is: ccc, yyy, xxx, bbb, aaa.
```

TestEnhancedLinkedList.java

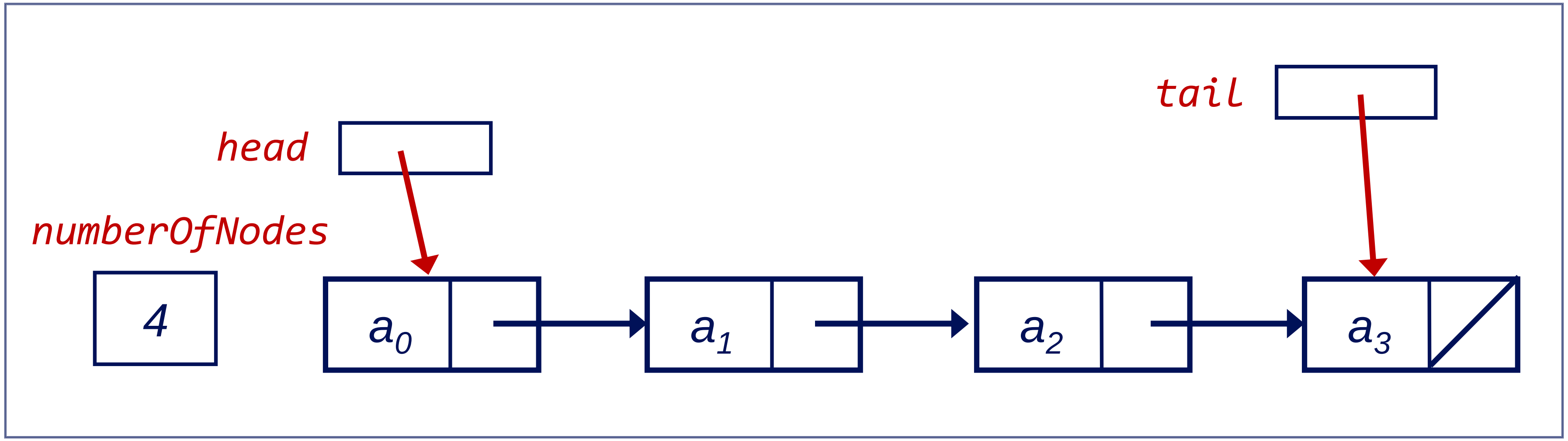
```
        System.out.println();
        System.out.println("Part 3");
        current = list.getHead();
        if (current != null) {
          current = current.getNext();
          list.removeAfter(current);
        }
        list.print();

        System.out.println();
        System.out.println("Part 4");
        list.removeAfter(null);
        list.print();
    }
}
```

```
Part 3
List is: ccc, yyy, bbb, aaa.

Part 4
List is: yyy, bbb, aaa.
```

# OVERVIEW!


BasicLinkedList
- head
- numberOfNodes
implements
<<interface>>
ListInterface
+ isEmpty()
+ size()
+ getFirst()
+ contains(E item)
+ addFirst(E item)
+ removeFirst()
+ print()
has-a
ListNode
- element
- next
+ getNext()
+ getElement()
+ setNext(ListNode<E> curr)
has-a
EnhancedLinkedList
- head
- numberOfNodes
implements
has-a
<<interface>>
EnhancedListInterface
+ isEmpty()
+ size()
+ getFirst()
+ contains(E item)
+ addFirst(E item)
+ removeFirst()
+ print()
+ getHead()
+ addAfter(ListNode<E> curr, E item)
+ removeAfter(ListNode<E> curr)
+ remove(E item)
implements
TailedLinkedList
- head
- tail
- numberOfNodes

- We further improve on Enhanced Linked List
  - To address the issue that adding to the end is slow
  - Add an extra data member called tail
  - Extra data member means extra maintenance too – no free lunch!
  - (Note: We could have created this Tailed Linked List as a subclass of Enhanced Linked List, but here we will create it from scratch.)
- Difficulty: Learn to take care of ALL cases of updating...



**TailedLinkedList.java**

```java
import java.util.*;

class TailedLinkedList<E> implements EnhancedListInterface<E> {
  private ListNode<E> head = null;
  private ListNode<E> tail = null;
  private int numberOfNodes = 0;

  public ListNode<E> getTail() {
      return tail;
  }

  public void addFirst(E item) {
      head = new ListNode<E> (item, head);
      numberOfNodes++;
      if (numberOfNodes == 1) {
          tail = head;
      }
  }
```

New code

- A new data member: tail
- Extra maintenance needed, eg: see addFirst()

TailedLinkedList.java

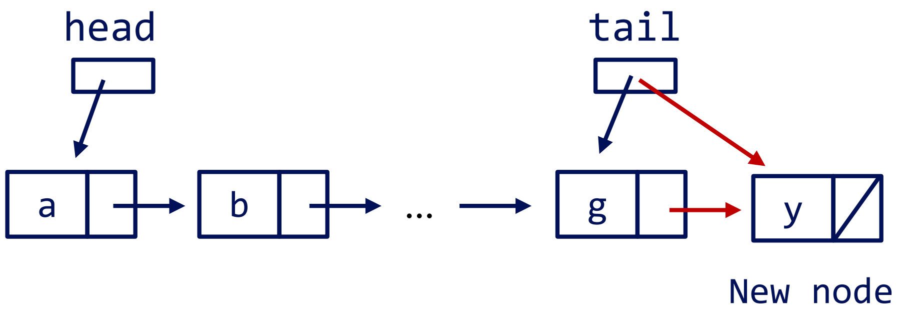
```java
public void addLast(E item) {
  if (head != null) {
    tail.setNext(new ListNode<E>(item));
    tail = tail.getNext();
  } else {
    tail = new ListNode<E>(item);
    head = tail;
  }
  numberOfNodes++;
}
```

- With the new member tail, can add to the end of the list directly by creating a new method addLast()
  - Remember to update tail

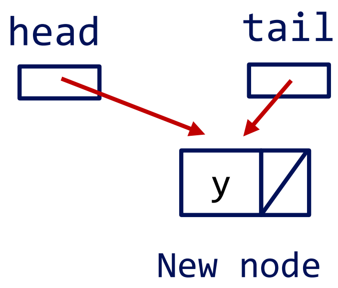
**TailedLinkedList.java**

```java
public void addLast(E item) {
  if (head != null) {
    tail.setNext(new ListNode<E>(item));
    tail = tail.getNext();
  } else {
    tail = new ListNode<E>(item);
    head = tail;
  }
  numberOfNodes++;
}
```

- Case 1: head != null




- Case 2: head == null



**TailedLinkedList.java**

```java
public void addAfter(ListNode<E> current, E item) {
  if (current != null) {
    current.setNext(new ListNode <E> (item, current.getNext()));
    if (current == tail) {
      tail = current.getNext();
    }
  } else {  // add to the front of the list
    head = new ListNode<E>(item, head);
    if (tail == null) {
      tail = head;
    }
  }
  numberOfNodes++;
}
```

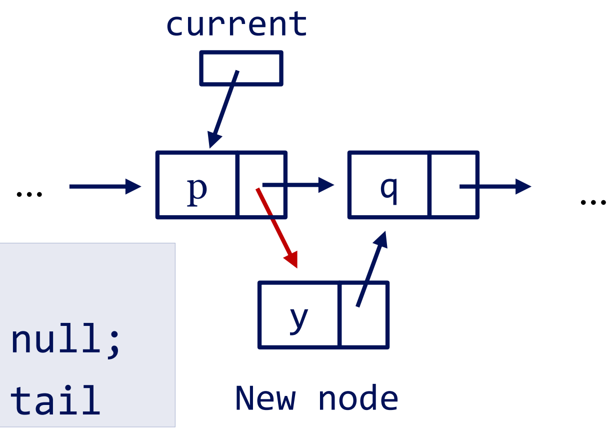
addAfter() method

We may replace our earlier addFirst() method with a simpler one that merely calls addAfter(). How?

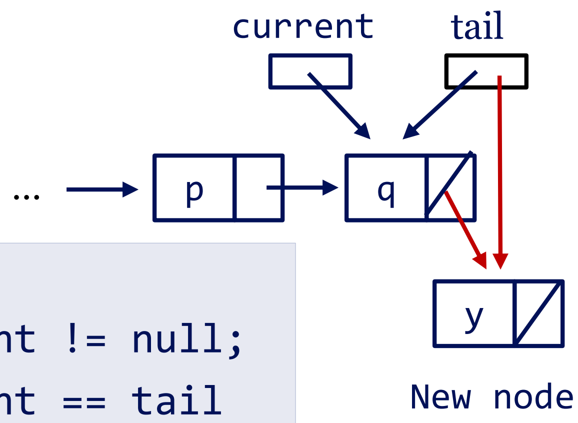
Hint: Study the removeFirst() method.

### TailedLinkedList.java

```java
public void addAfter(ListNode<E> current, E item) {
  if (current != null) {
    current.setNext(new ListNode <E> (item, current.getNext()));
    if (current == tail) {
      tail = current.getNext();
    }
  } else {  // add to the front of the list
    …
  }
  numberOfNodes++;
}
```




- Case 1A
  - current != null;
  - current != tail

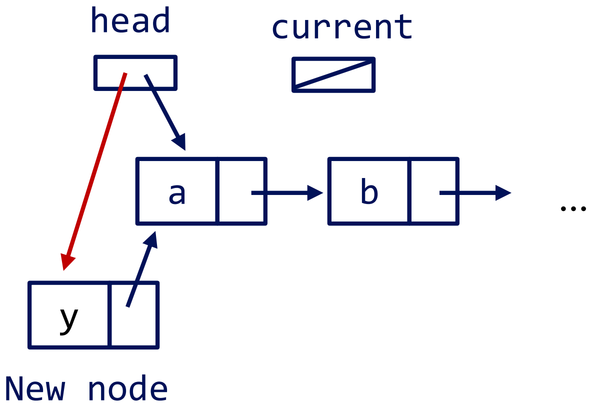



- Case 1B
  - current != null;
  - current == tail

# 4.2 Tailed Linked List (7/10)

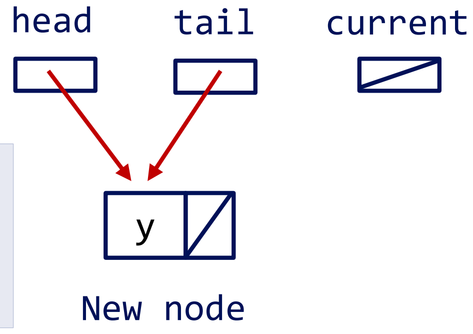
**TailedLinkedList.java**

```java
public void addAfter(ListNode<E> current, E item) {
  if (current != null) {
    …
  } else {  // add to the front of the list
    head = new ListNode<E>(item, head);
    if (tail == null) {
      tail = head;
    }
  }
  numberOfNodes++;
}
```




- Case 2A
  - current == null;
  - tail != nul;

- Case 2B
  - current == null;
  - tail == null;



**TailedLinkedList.java**

```java
public E removeAfter(ListNode<E> current)
  throws NoSuchElementException {
  E temp;
  if (current != null) {
    ListNode<E> nextPtr = current.getNext();
    if (nextPtr != null) {
      temp = nextPtr.getElement();
      current.setNext(nextPtr.getNext());
      numberOfNodes--;
      if (nextPtr.getNext() == null) {
        // last node is removed
        tail = current;
      }
      return temp;
    } else {
      throw new NoSuchElementException("...");
    }
  }
```

**TailedLinkedList.java**

```java
else {
  // if current == null, we want to remove head
  if (head != null) {
    temp = head.getElement();
    head = head.getNext();
    numberOfNodes--;
    if (head == null) {
      tail = null;
    }
    return temp;
  } else {
    throw new NoSuchElementException("...");
  }
}
```

**TailedLinkedList.java**

```
public E removeFirst() throws NoSuchElementException {
    return removeAfter(null);
}
```

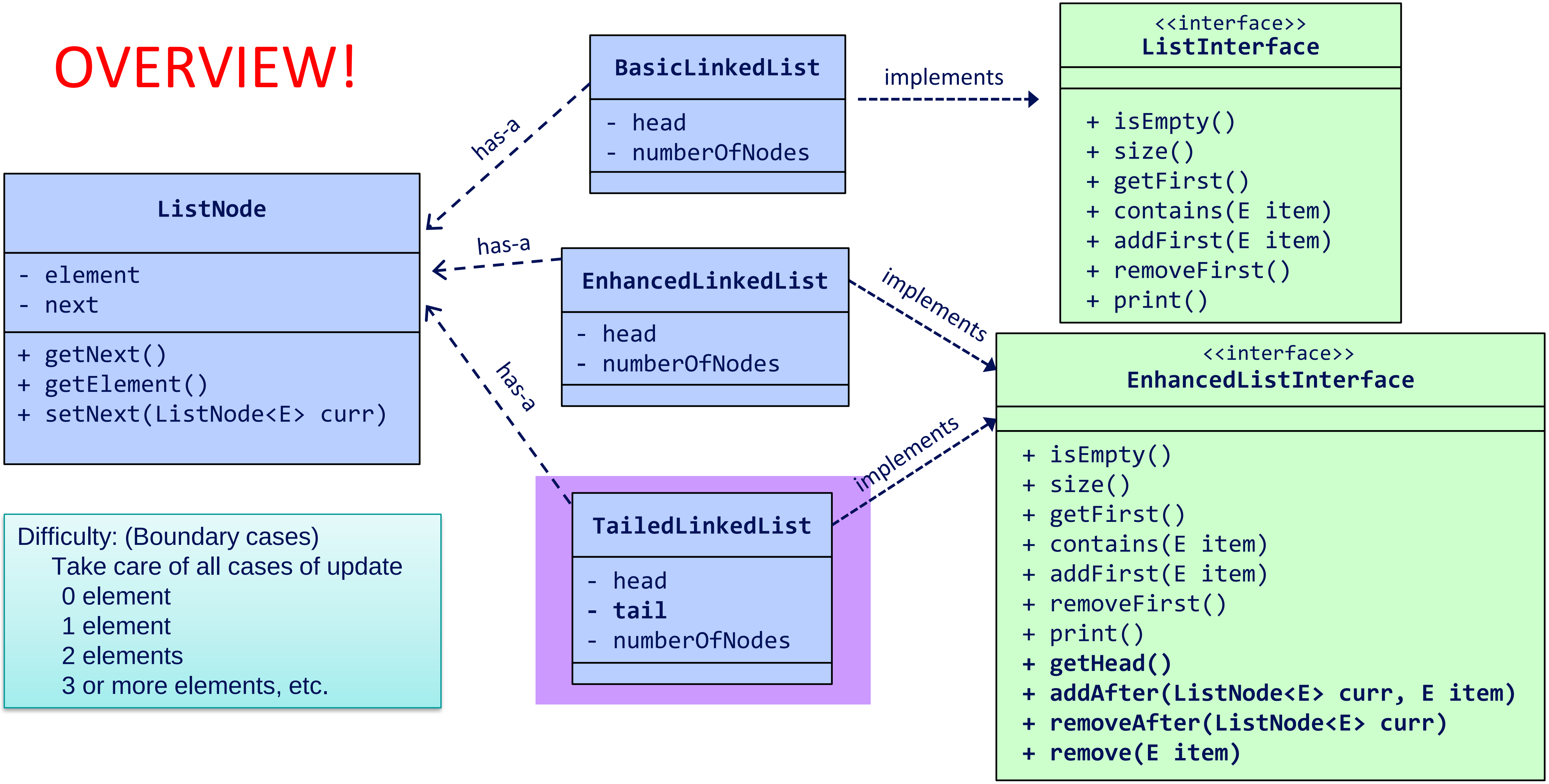
- removeFirst() method
    - removeFirst() is a special case in removeAfter()

TestTailedLinkedList.java

```java
import java.util.*;

public class TestTailedLinkedList {
  public static void main(String[] args) throws NoSuchElementException {
    TailedLinkedList<String> list = new TailedLinkedList<String>();

    System.out.println("Part 1");
    list.addFirst("aaa");
    list.addFirst("bbb");
    list.addFirst("ccc");
    list.print();
    System.out.println("Part 2");
    list.addLast("xxx");
    list.print();
    System.out.println("Part 3");
    list.removeAfter(null);
    list.print();
  }
}
```

```
Part 1
List is: ccc, bbb, aaa.
Part 2
List is: ccc, bbb, aaa, xxx.
Part 3
List is: bbb, aaa, xxx.
```

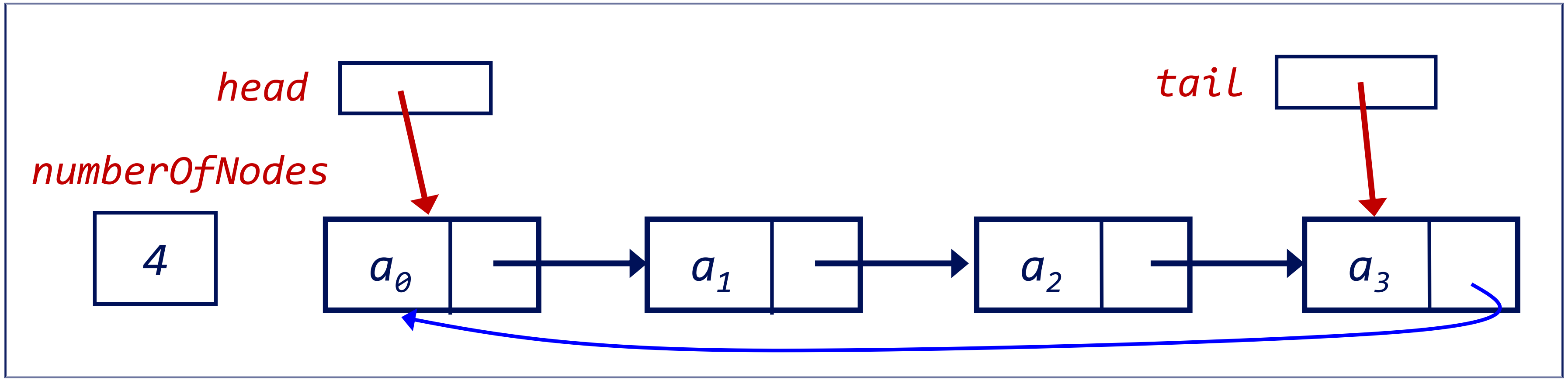
# OVERVIEW!



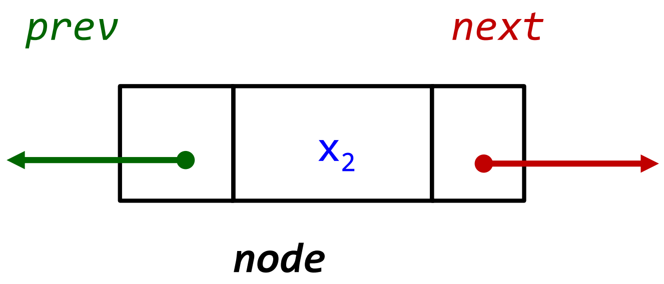

Difficulty: (Boundary cases)
Take care of all cases of update
- 0 element
- 1 element
- 2 elements
- 3 or more elements, etc.

# Other variants of linked lists

- There are many other possible enhancements of linked list
- Example: Circular Linked List
  - To allow cycling through the list repeatedly, e.g. in a **round robin system** to assign shared resource
  - Add a link from tail node of the TailedLinkedList to point back to head node
  - Different in linking need different maintenance – no free lunch!
- Difficulty: Learn to take care of ALL cases of updating, such as inserting/deleting the first/last node in a Circular Linked List
- Explore this on your own; write a class CircularLinkedList



- In the preceding discussion, we have a "**next**" pointer to move forward
- Often, we need to move backward as well
- Use a "**prev**" pointer to allow backward traversal
- Once again, no free lunch – need to maintain "**prev**" in all updating methods
- Instead of ListNode class, need to create a DListNode class that includes the additional "**prev**" pointer



# 5.2 Doubly Linked List: DListNode (2/3)

**TestTailedLinkedList.java**

```java
class DListNode<E> {
  private E element;
  private DListNode<E> prev;
  private DListNode<E> next;

  public DListNode(E item) {
      this(item, null, null);
  }

  public DListNode(E item,
                   DListNode<E> prevNode,
                   DListNode<E> nextNode) {
      element = item;
      prev = prevNode;
      next = nextNode;
  }

  /* get the prev DListNode */
  public DListNode<E> getPrev() {
      return this.prev;
  }
```

**TestTailedLinkedList.java**

```java
  /* get the next DListNode */
  public DListNode<E> getNext() {
    return this.next;
  }

  /* get the element of the ListNode */
  public E getElement() {
    return this.element;
  }

  /* set the prev reference */
  public void setPrev(DListNode<E> prevNode) {
    prev = prevNode;
  }

  /* set the next reference */
  public void setNext(DListNode<E> nextNode) {
    next = nextNode;
  }
}
```

# 5.2 Doubly Linked List: DListNode (2/3)

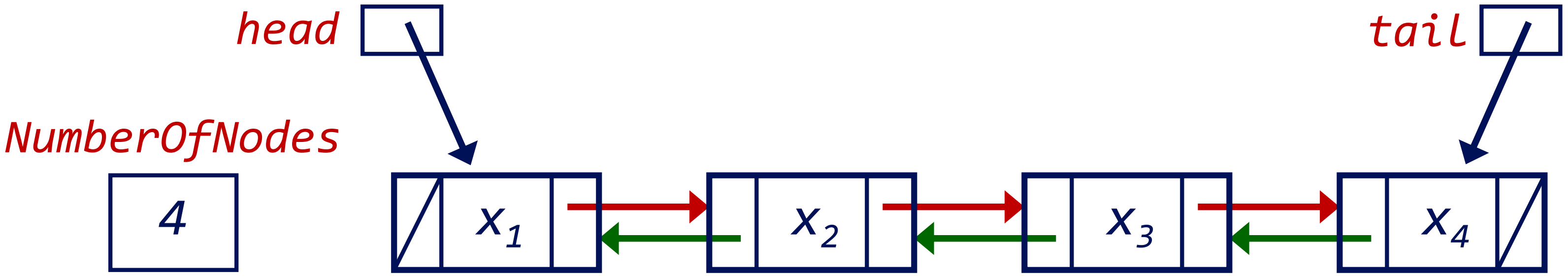
**DListNode.java**

```java
class DListNode<E> {
  private E element;
  private DListNode<E> prev;
  private DListNode<E> next;

  public DListNode(E item) {
      this(item, null, null);
  }

  public DListNode(E item,
                   DListNode<E> prevNode,
                   DListNode<E> nextNode) {
      element = item;
      prev = prevNode;
      next = nextNode;
  }

  /* get the prev DListNode */
  public DListNode<E> getPrev() {
      return this.prev;
  }
```

**DListNode.java**

```java
  /* get the next DListNode */
  public DListNode<E> getNext() {
    return this.next;
  }

  /* get the element of the ListNode */
  public E getElement() {
    return this.element;
  }

  /* set the prev reference */
  public void setPrev(DListNode<E> prevNode) {
    prev = prevNode;
  }

  /* set the next reference */
  public void setNext(DListNode<E> nextNode) {
    next = nextNode;
  }
}
```

- An example of a doubly linked list




- Explore this on your own.
- Write a class DoublyLinkedList to implement the various linked list operations for a doubly linked list.

# Using the LinkedList class

## 6 Java Class: LinkedList<E>

- This is the class provided by Java library
- This is the linked list implementation of the List interface
- It has many more methods than what we have discussed so far of our versions of linked lists. On the other hand, we created some methods not available in the Java library class too.
- Please do not confuse this library class from our class illustrated here. In a way, we open up the Java library to show you the inside working.
- For purposes of sit-in labs or exam, please use whichever one as you are told if stated.

## 6.1 Class LinkedList: API (1/3)

**Constructors**

| Constructor | Description |
|---|---|
| `LinkedList()` | Constructs an empty list. |
| `LinkedList(Collection<? extends E> c)` | Constructs a list containing the elements of the specified collection, in the order they are returned by the colle |

### *Method Summary*

**All Methods** | **Instance Methods** | **Concrete Methods**

| Modifier and Type | Method | Description |
|---|---|---|
| `void` | `add(int index, E element)` | Inserts the specified element at the specified position in this list. |
| `boolean` | `add(E e)` | Appends the specified element to the end of this list. |
| `boolean` | `addAll(int index, Collection<? extends E> c)` | Inserts all of the elements in the specified collection into this list, starting at |
| `boolean` | `addAll(Collection<? extends E> c)` | Appends all of the elements in the specified collection to the end of this list, i |
| `void` | `addFirst(E e)` | Inserts the specified element at the beginning of this list. |
| `void` | `addLast(E e)` | Appends the specified element to the end of this list. |
| `void` | `clear()` | Removes all of the elements from this list. |

**TestLinkedListAPI.java**

```java
import java.util.*;

public class TestLinkedListAPI {

  static void printList(LinkedList <Integer> alist) {
      System.out.print("List is: ");
      for (int i = 0; i < alist.size(); i++) {
          System.out.print(alist.get(i) + "\t");
      }
      System.out.println();
  }

  // Print elements in the list and also delete them
  static void printListv2(LinkedList <Integer> alist) {
      System.out.print("List is: ");
      while (alist.size() != 0) {
          System.out.print(alist.element() + "\t");
          alist.removeFirst();
      }
      System.out.println();
  }
```

**TestLinkedListAPI.java**

```java
  public static void main(String[] args) {
    LinkedList<Integer> alist = new LinkedList<Integer>();
    for (int i = 1; i <= 5; i++) {
      alist.add(new Integer(i));
    }

    printList(alist);

    System.out.println("First element: " + alist.getFirst());
    System.out.println("Last element: " + alist.getLast());

    alist.addFirst(888);
    alist.addLast(999);
    printListv2(alist);
    printList(alist);
  }
}
```

```
List is: 1       2       3       4       5
List is: 1       2       3       4       5
First element: 1
Last element: 5
List is: 888     1       2       3       4       5       999
List is:
```

## Why “reinvent the wheel”?

- In a data structures course, students are often asked to implement well-known data structures.
- A question we sometimes hear: “Since there is the API, why do we need to learn to write our own code to implement a data structure like linked list?”
- Writing the code allows you to gain an indepth understanding of the data structures and their operations
- The understanding will allow you to appreciate their complexity analysis (to be covered later) and use the API effectively

# 7. Summary (1/2)

- We learn to create our own data structure

  - In creating our own data structure, we face 3 difficulties:
    1. Re-use of codes (inheritance confusion)
    2. Manipulation of pointers/references (The sequence of statements is important! With the wrong sequence, the result will be wrong.)
    3. Careful with all the boundary cases

  - Drawings are very helpful in understanding the cases (point 3), which then can help in knowing what can be used/manipulated (points 1 and 2)

## 7. Summary (2/2)

- Once we can get through this lecture, the rest should be smooth sailing as all the rest are similar in nature
  - You should try to add more methods to our versions of LinkedList, or to extend ListNode to other type of node

- Please do not forget that the Java Library class is much more comprehensive than our own – for sit-in labs and exam, please use whichever one as you are told if stated.

# Thank you!

HUS
VNU UNIVERSITY OF SCIENCE