4CCS1DST – Data Structures

Lecture 2:

Review of Java (Ch. 2)

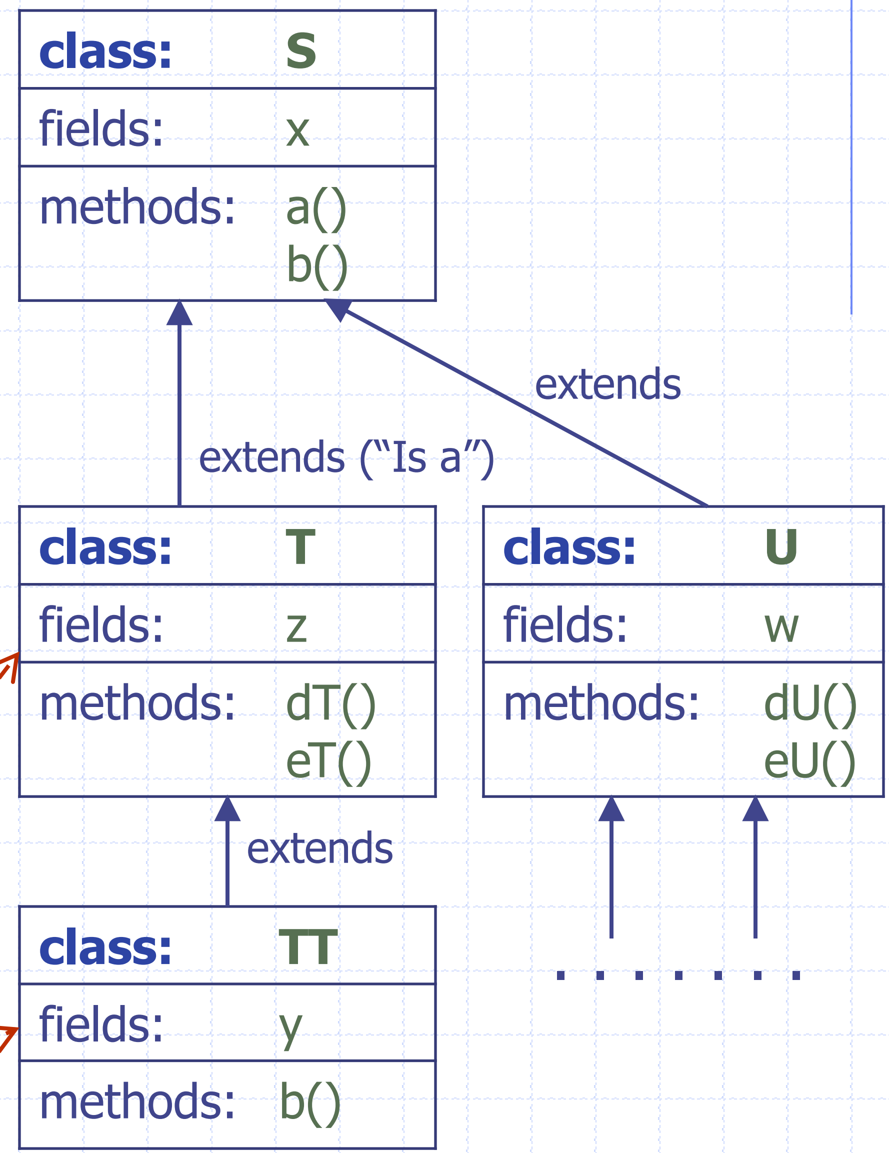
inheritance, polymorphism, abstract classes, interfaces, casting, generic types

Linked Lists (§ 3.2)

© 2010 Goodrich, Tamassia

Inheritance

- Inheritance: general classes can be specialized to more particular classes.
- Main purpose: **reusing code.**
- General class: **base class, superclass.**
- Specialized class: **subclass.**
- We get: **hierarchy of classes**



Class T inherits x, a(), b() from S;
extends S by adding z, dT(), eT().

Class TT inherits x, a(), z, dT(), eT();
overrides (specializes) b() from S.

Inheritance: example

- **Numeric progressions**:
  design classes for stepping through and printing out numeric progressions.
- **Arithmetic progression**; defined by the first number (we assume it's 0) and the common difference; example:
  0, 3, 6, 9, 12, … (difference = 3)
- **Geometric progression**; defined by the first number (we assume it's 1) and the common ratio; example:
  1, 4, 16, 64, … (ratio = 4)
- **Fibonacci progression**; defined by the first number (we assume it's 0) and the second number; each subsequent number is the sum of the previous two numbers; example:
  0, 3, 3, 6, 9, 15, … (second number = 3)

© 2010 Goodrich, Tamassia

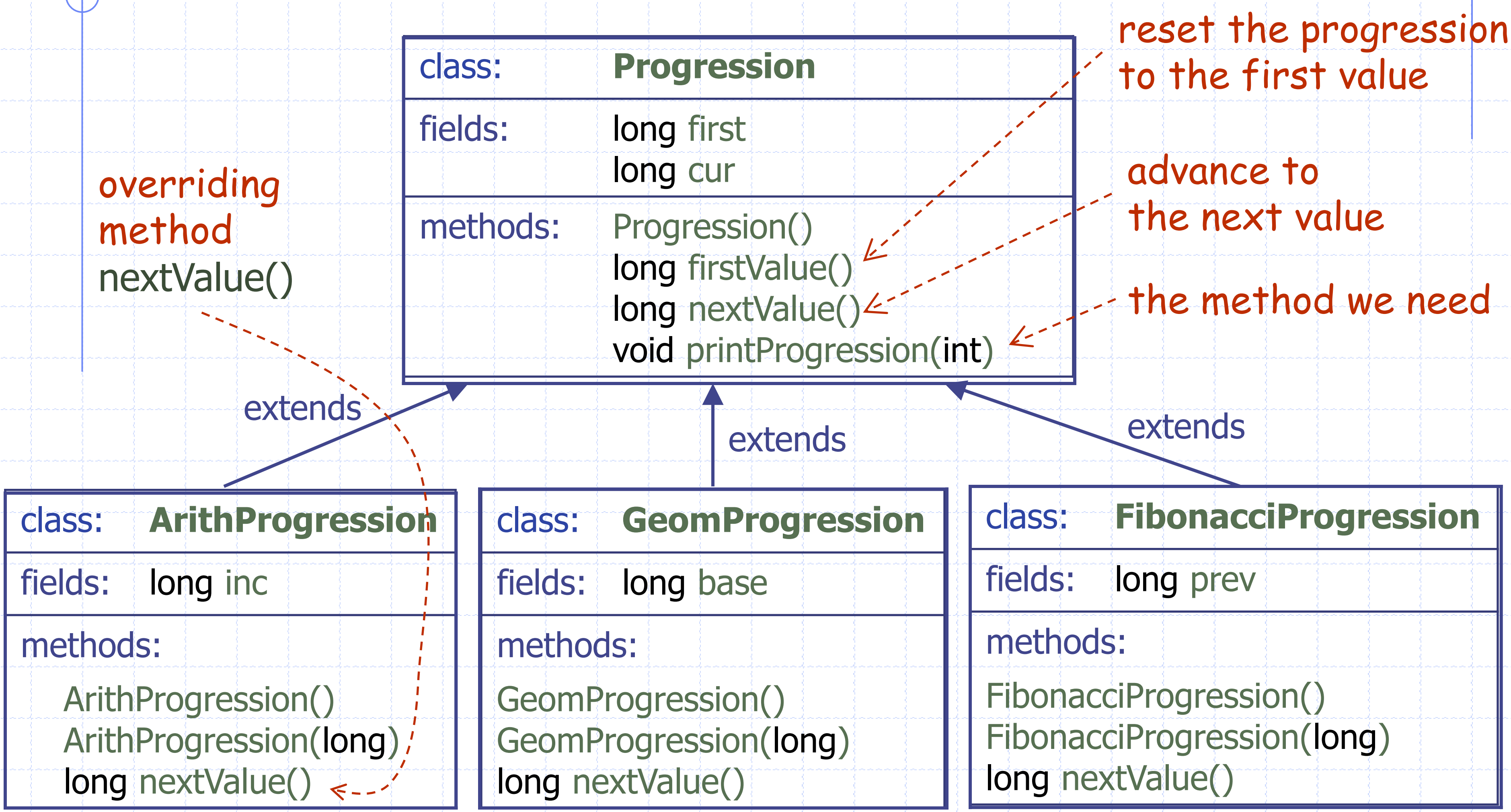
Example (cont.)

- **Intended usage**:

```
...
progA = new ArithProgression(5);
          // create an object representing
          // arithmetic progression with difference 5
progA.printProgression(7);
          // prints:    0 5 10 15 20 25 30
...
progG = new GeomProgression(3);
          // create an object representing
          // geometric progression with ratio 3
progG.printProgression(6);
          // prints:    1 3 9 27 81 243
```

© 2010 Goodrich, Tamassia

"Progression" class hierarchy



Class Progression (1)

Modifier	Class	Package	Subclass	World
public	Y	Y	Y	Y
protected	Y	Y	Y	N
no modifier	Y	Y	N	N
private	Y	N	N	N

```
package progress;

public class Progression {
    protected long first;   // first value of the progression
    protected long cur;     // current value of the progression

    protected Progression() {         // default constructor
        cur = first = 0;
    }

    protected  long  firstValue() {  // resets the progression to the first value
        cur = first;    return  cur;
    }

    protected  long  nextValue() {  // advances the progression to the next value
        return  ++cur;     // default next value
    }
    // (cont.)
```

Class Progression (2)

```
public void printProgression( int n ) {     // prints the first n values
        System.out.print( firstValue() );       // reset to 1-st value and print it
        for (int i = 2; i <= n; i++) {
                System.out.print( " " + nextValue() ); // print the i-th value
        }
        System.out.println();   // ends the line
}

public static void main( String[] args ) {    // test Progression
        Progression prog = new Progression();
        System.out.println( "Default progression:" );
        prog.printProgression(10);
}
}
```

>java Progression	Default progression: 0 1 2 3 4 5 6 7 8 9

Class ArithProgression

overloading

```
class ArithProgression extends Progression {
    // inherits fields:  first,  cur
    protected long  inc;    // increment

    ArithProgression(long increment) {  // constructor: set the given increment
        inc = increment;
    }

    ArithProgression() {     // default constructor setting a unit increment
        this(1);
    }

    //  inherits methods firstValue() and printProgression(int)
    protected long nextValue() {     // specialize nextValue() from Progression
        cur += inc;  return cur;
    }

    public static void main(String[ ] args) { ...  }      // test ArithProgression
}
```

Class GeomProgression

```
class GeomProgression extends Progression {
    // inherits fields:  first,  cur
    protected long  base;    // base (ratio) of the progression

    GeomProgression(long  b) {    // constructor: set the given base
        base = b;    first = 1;   cur = first;
    }

    GeomProgression() {       // default constructor setting base 2
        this(2);
    }

    //  inherits methods firstValue() and printProgression(int)
    protected long nextValue() {     // specialize nextValue() from Progression
        cur *= base;   return cur;
    }

    public static void main(String[ ] args) { …  }      // test GeomProgression
}
```

Class FibonacciProgression

$\boxed{?, F_0,}\ F_1, F_2, \boxed{F_3, F_4,}\ F_5, \ldots$

since we want:
second = first + prev

```
class FibonacciProgression extends Progression {
    // inherits fields:  first,  cur
    protected long  prev;  // the previous value

    FibonacciProgression(long  second) { // the second value given
        prev = second – first;              // default first value = 0
    }

    FibonacciProgression() {   // default constructor setting the second value to 1
        this(1);
    }

    //  inherits methods firstValue() and printProgression(int)
    protected long nextValue() {     // specialize nextValue() from Progression
        long  next = cur + prev;
        prev = cur;   cur = next;   return cur;
    }

    public static void main(String[] args) { …  }    // test FibonacciProgression
}
```

Test the progression classes

```
class TestProgression {
    public static void main(String[] args) {
        Progression prog;

        System.out.println("Arithmetic progression with default increment:");
        prog = new ArithProgression();
        prog.printProgression(10);

        System.out.println("Arithmetic progression with increment 5:");
        prog = new ArithProgression(5);
        prog.printProgression(10);

        System.out.println("Geometric progression with base 3:");
        prog = new GeomProgression(3);
        prog.printProgression(10);

        System.out.println("Fibonacci progression with start values 0 and 4:");
        prog = new FibonacciProgression(4);
        prog.printProgression(10);
    }
}
```

Test the progression classes (cont.)

>java TestProgression

Arithmetic progression with default increment:
0 1 2 3 4 5 6 7 8 9

Arithmetic progression with increment 5:
0 5 10 15 20 25 30 35 40 45

Geometric progression with base 3:
1 3 9 27 81 243 729 2187 6561 19683

Fibonacci progression with start values 0 and 4:
0 4 4 8 12 20 32 52 84 136

© 2010 Goodrich, Tamassia

Polymorphism

- Polymorphism: objects referenced by variables of the same refdrence type can have different forms.
- Objects referenced by variable

  Progression prog;

  can have different forms:
  Progression, ArithProgression, GeomProgression, FibonacciProgression

- Consider:

```
prog  = new ... ;
prog.printProgression(10);
```

from class Progression

- from the class of the current object

```
public void printProgression(int n) {
    System.out.print( firstValue() );
    for (int  i = 2;  i <= n;  i++)
        { System.out.print( " " + nextValue() ); }
    System.out.println();
}
```

Abstract classes

- An **abstract method –** a method which has no definition (a method without body).
- An **abstract class** – a class which may have abstract methods.
- Abstract classes are used only as superclasses in inheritance hierarchies.

  These classes cannot be used to instantiate objects, because they are incomplete.
- Subclasses of an abstract class must declare the "missing pieces".
- Classes that can be used to instantiate objects are called **concrete classes.**

  Such classes provide implementation for every method they declare, and for all abstract methods in superclasses.

© 2010 Goodrich, Tamassia

Example: Progression as an abstract class

Class Progression was "forced" to be a concrete class.

It should be an abstract class, as each progression is of a specific type.

```
public  abstract  class  ProgressionAbstract {
    protected long first, cur;        // the first and the current values
    protected long firstValue() { // resets the progression to the first value
        cur = first;    return  cur;
    }

    protected abstract long nextValue();
    // advances the progression to the next value

    public void printProgression(int n) { ...   // prints the first n values
        for (int  i = 2;  i <= n;  i++)  {
            System.out.print( " " + nextValue() );
        }  ...
    }
}
```

abstract method; without a body; must be defined in concrete subclasses

definition will be taken from the class of the current object

© 2010 Goodrich, Tamassia

Extending abstract classes

```
class ArithProgression2 extends ProgressionAbstract
    { ... } // in this case, everything else as before

class GeomProgression2 extends ProgressionAbstract { ... }

class FibonacciProgression2 extends ProgressionAbstract { ... }

class TestProgressionAbstract {
    public static void main(String[] args) {
        ProgressionAbstract  prog;
        prog = new ProgressionAbstract(5); // cannot instantiate the type
                                           // ProgressionAbstract
        prog = new ArithProgression2();
        System.out.println("Arithmetic progression with default increment:");
        prog.printProgression(10);
        ...
    }
}
```

© 2010 Goodrich, Tamassia

Interfaces

- Java **interface** is an ultimate abstract class: it may have only public abstract methods

  Also static final fields (constants) and, from Java 8, static methods and default methods, all public; but we won't use these possibilities.

- A class is said to **implement the interface** if it provides definitions for all of the abstract methods in the interface.

© 2010 Goodrich, Tamassia

Example: Progression as an interface (1)

```
public  interface  ProgressionInterface {
   public long firstValue();  // resets the progression to the first value
   public long nextValue(); // advances the progression to the next value
   public void printProgression(int n); // prints the first n values of the
                                        // progression
}
```

Subclasses must declare all required fields and must provide definitions (implementations) of all methods specified in the interface.

© 2010 Goodrich, Tamassia

Example: Progression as an interface (2)

```
public class ArithProgressionIntF implements ProgressionInterface {
    protected long first, cur, inc; // all required fields

    ArithProgressionIntF( long increment ) { inc = increment; }
    ArithProgressionIntF() { this(1); }

    public long firstValue() { … }                      // provide full code
    public long nextValue() { … }                       // provide full code
    public void printProgression( int n ) { … }         // provide full code

    public static void main( String[] args ) { // test
        ProgressionInterface prog;
        prog = new ArithProgressionIntF(5);
        prog.printProgression(10);
    }
}
```

Example: Progression as interface, Java 8

```
public  interface  ProgressionInterfaceJava8 {
   int MAX_LENGTH = 100;          // static and final -> constant

   public static void printMaxLength() {
          {  System.out.print("Maximum length: " + MAX_LENGTH);  }

   public long firstValue();  // resets the progression to the first value

   public long nextValue(); // advances the progression to the next value

   public  default  void printProgression(int n) { // prints first n values
      System.out.print(firstValue());
      if ( n > MAX_LENGTH )  { n = MAX_LENGTH; }
      for (int i = 2; i <= n; i++)  { System.out.print(" "+ nextValue()); }
   }
}      // no need for printProgression in classes implementing this interface
```

Interfaces and multiple inheritance

- In Java: a class may extend only one other class.

  For example, this is not possible in Java:

  public class Student { ... }
  public class Employee { ... }
  public class StudentEmployee **extends** Student, Employee {...}

- In Java: a class may implement more than one interface.

  For example, we could have in Java:

  public interface Student { ... }
  public interface Employee { ... }
  public class StudentEmployee **implements** Student, Employee {...}

Type conversions; Casting

- Widening (type) conversions: converting to a "wider" type.
  No problem.
- Narrowing conversions: converting to a "narrower" type.
  "Casting" necessary.
- Example:

Package **java.lang** provides classes that are fundamental to the design of the Java programming language.

This package includes:

public abstract class Number **implements** ...
public final class Integer **extends** Number **implements** ...
public final class Double **extends** Number **implements** ...
..... // there are other classes which also extend class Number

© 2010 Goodrich, Tamassia

Casting: example

```
public class TestCasting {
    public static void main(String[] args) {
        Number  n1, n2;
        Integer  i = new Integer(3);
        Double  d = new Double(3.1415);
        n1 = i;     // OK; widening conversion from Integer to Number
        n2 = d;     // OK; widening conversion from Double to Number
        Integer  j = n1; // compilation error: narrowing conversion, cast required
        Integer  k = (Integer) n1;   // OK; casting to a narrower type
        Integer  p = (Integer) n2;   // compilation OK; runtime casting error
        System.out.println(n1 + " " + n2 + " " + ..... + p);
    }
}
```

Generics

- Starting with version 5.0, Java includes a **generics framework** for using formal type parameters.
- Examples:

In standard Java library: **public class** ArrayList<E>
Use: ArrayList<String> a = **new** ArrayList<String>(10);

```
public class Pair<K, V> {
        K key;
        V value;
        public void set(K k, V v) { key = k; value = v; }   // modifier
        public K getKey() { return key; }                   // accessor
        public V getValue() { return value; }               // accessor
        public String toString() {
                return "[" + getKey() + ", " + getValue() + "]"; }
}
```

formal type parameters

actual type parameters

```
Pair<String, Integer> pair1 = new Pair<String, Integer>();
pair1.set( new String("height"), new Integer(36) );
```

Generic versions of methods

- The generics framework allows us to define generic versions of methods.
- Example:

key value

```
public static <T> void reverse( Pair<T, T> p) {
      p.set( p.getValue(), p.getKey() );
}
```

```
Pair<String, String>  pair2 = new Pair<String, String>();
pair2.set( new String("KCL"), new String("King's") );
System.out.println( pair2 );                    // prints: [KCL, King's]
reverse(pair2);
System.out.println( pair2 );                    // prints: [King's, KCL]
```

© 2010 Goodrich, Tamassia

Specifying and implementing a data structure

- **Data structure** – a systematic way of organizing, accessing and updating data (maintained in the main computer memory during the execution of a broader computing task).
- **Algorithm** – a step-by-step procedure for performing some tasks.
- Algorithms use data structures. Data structures need algorithms for performing the tasks of accessing and updating data.
- **Abstract data type** (**ADT**) – a model of a data structure that specifies the type of data stored, the operations supported on them, and the type of parameters of the operations.

  An ADT specifies what each operation does, but not how it does it.
- In Java, an ADT can be **expressed by an interface** (and that's the approach which we will use).
- An ADT is realized by a concrete data structure, which is implemented in Java by a (concrete) class.

© 2010 Goodrich, Tamassia

Example: Stack data structure – ADT

- An instance of the stack data structure is a sequence of elements (objects) with one end designated as the top of the stack:

  (E1, E2, E3, … , En)

  top of the stack

- Main stack operations:
  - push(e): insert element e at the top of the stack
  - pop(): remove and return the element at the top of the stack
- Additional stack operations:
  - top(): return the top element in the stack (without removing)
  - size(): returns the number of elements stored
  - isEmpty(): indicates if the stack is empty

Stack Interface in Java

(generic) type of elements

```
public interface Stack<E> {
    public void  push( E element );
    public  E  pop()  throws  EmptyStackException;
    public  E  top()   throws  EmptyStackException;
    public  int  size();
    public  boolean  isEmpty();
}
```

© 2010 Goodrich, Tamassia

Implementations of Stack Interface

public class ArrayStack<E> **implements** Stack<E> { … }

public class NodeStack<E> **implements** Stack<E> { … }

© 2010 Goodrich, Tamassia

Implementations of Stack Interface

- An array-based stack implementation:

```
public class ArrayStack<E> implements Stack<E> { … }
```

- Implementation of a stack based on a singly linked list:

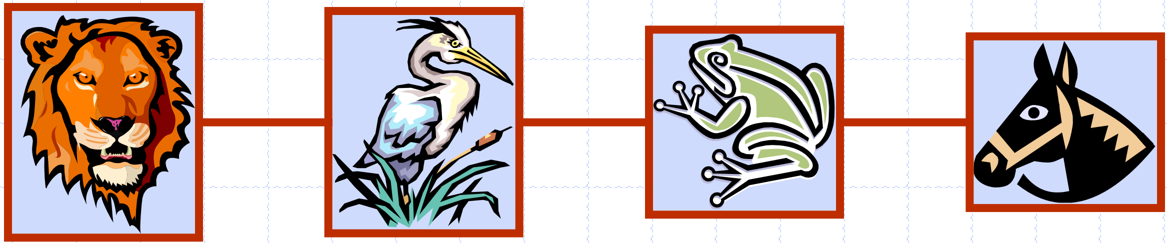
```
public class NodeStack<E> implements Stack<E> { … }
```

- Using an implementation of stack:

```
Stack<String> S = new NodeStack<String>();
S.push( "begin" );
S.push( "if" );
…
```

© 2010 Goodrich, Tamassia

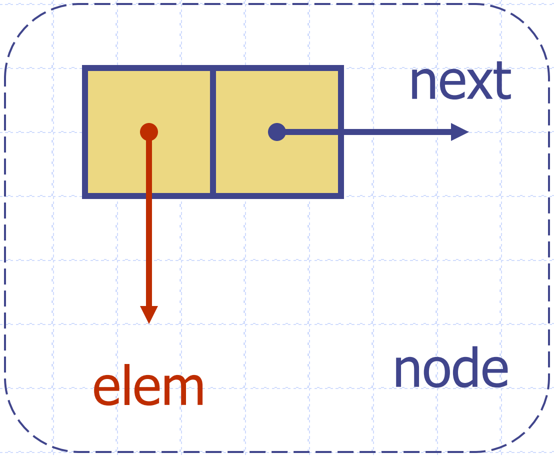
Linked Lists

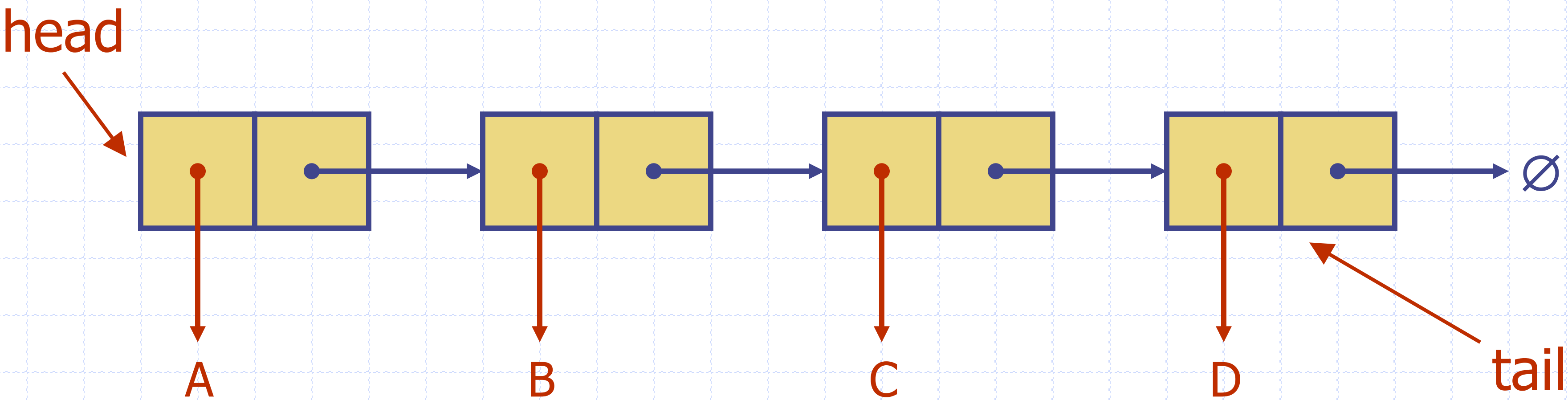


© 2010 Goodrich, Tamassia

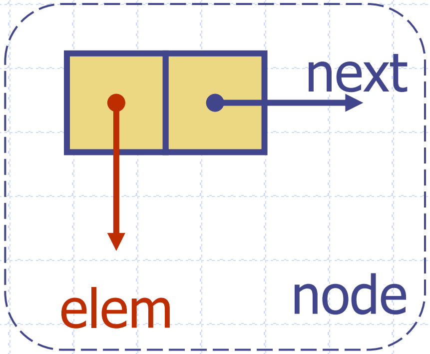
Singly Linked List (§ 3.2)

- **Linked list** – an alternative to array for storing a sequence of objects
- A singly linked list is a sequence of **nodes**
- Each node stores
  - element
  - link to the next node



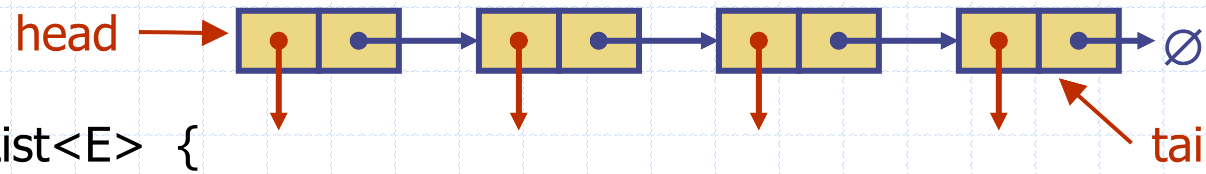


The Node class for list nodes




```
public class   Node<E>  {
    // instance variables:
    private   E  element;
    private   Node<E>  next;
    // creates a node with the given element and next node
    public  Node( E  e,  Node<E>  n )  {  element  =  e;   next  =  n; }
    // creates a node with null references to its element and next node
    public  Node()   {  this(null,  null); }

    // accessor methods:
    public   E  getElement()  {   return  element;  }
    public   Node<E>  getNext()  {  return  next;  }

    // modifier methods:
    public void  setElement( E  newElem )   {  element  =  newElem; }
    public void  setNext( Node<E>  newNext )  {  next  =  newNext;  }
}
```

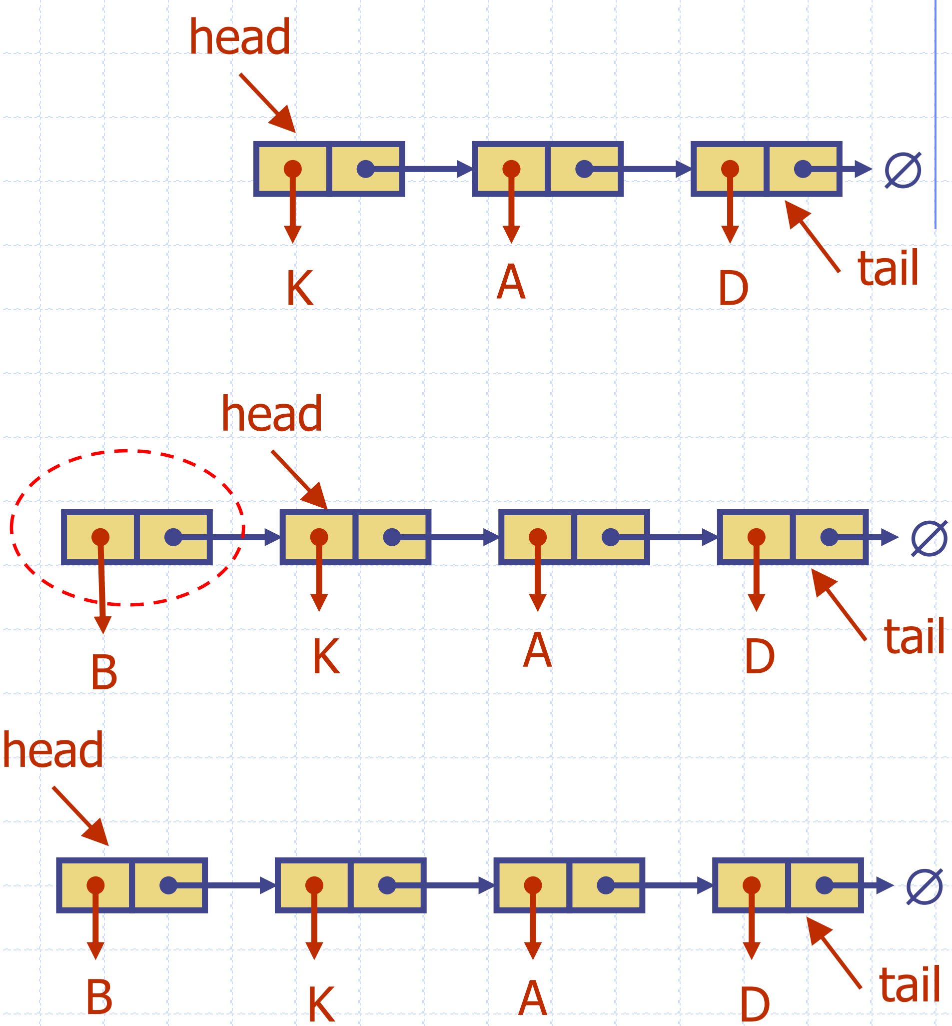
The SLinkedList class defining linked list




```
public class SLinkedList<E>  {
    protected  Node<E>  head;     // head node of the list (always needed)
    protected  Node<E>  tail;     // tail node of the list (if needed)
    protected  long  size;        // number of nodes in the list (if needed)

    // default constructor that creates an empty list
    public  SLinkedList() {  head = null;  tail = null;  size = 0;  }

    // update and search methods
    public  void  insertAtHead( E  newElem)  { … }
    …

    public static void  main( String[]  args ) {  // test
        SLinkedList<String>  list  =  new SLinkedList<String>();
        list.insertAtHead( "word" );
        …
    }
}
```

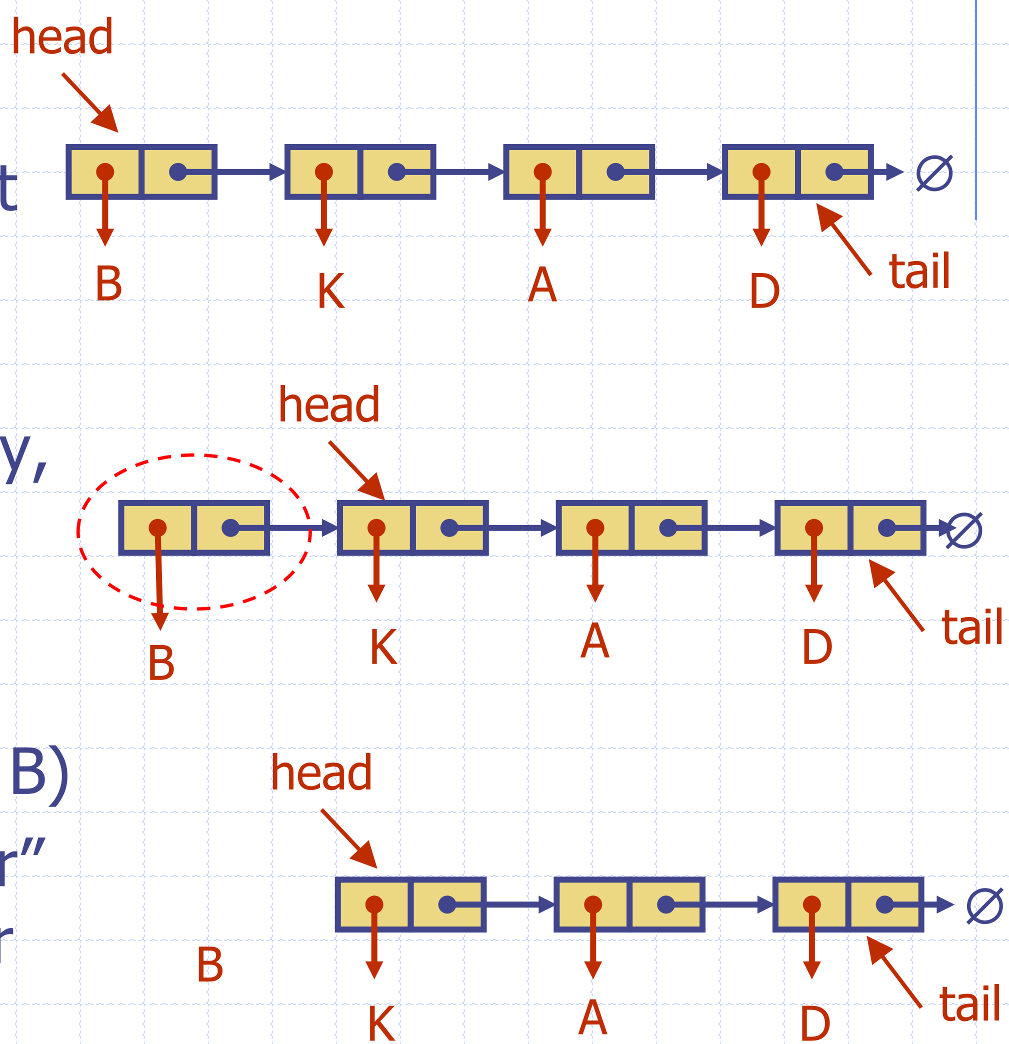
Inserting at the Head

1. Allocate a new node
2. Insert new element
3. Have new node point to old head
4. Update head to point to new node
5. Update "size", if maintained
6. If inserting to empty list, update tail, if maintained



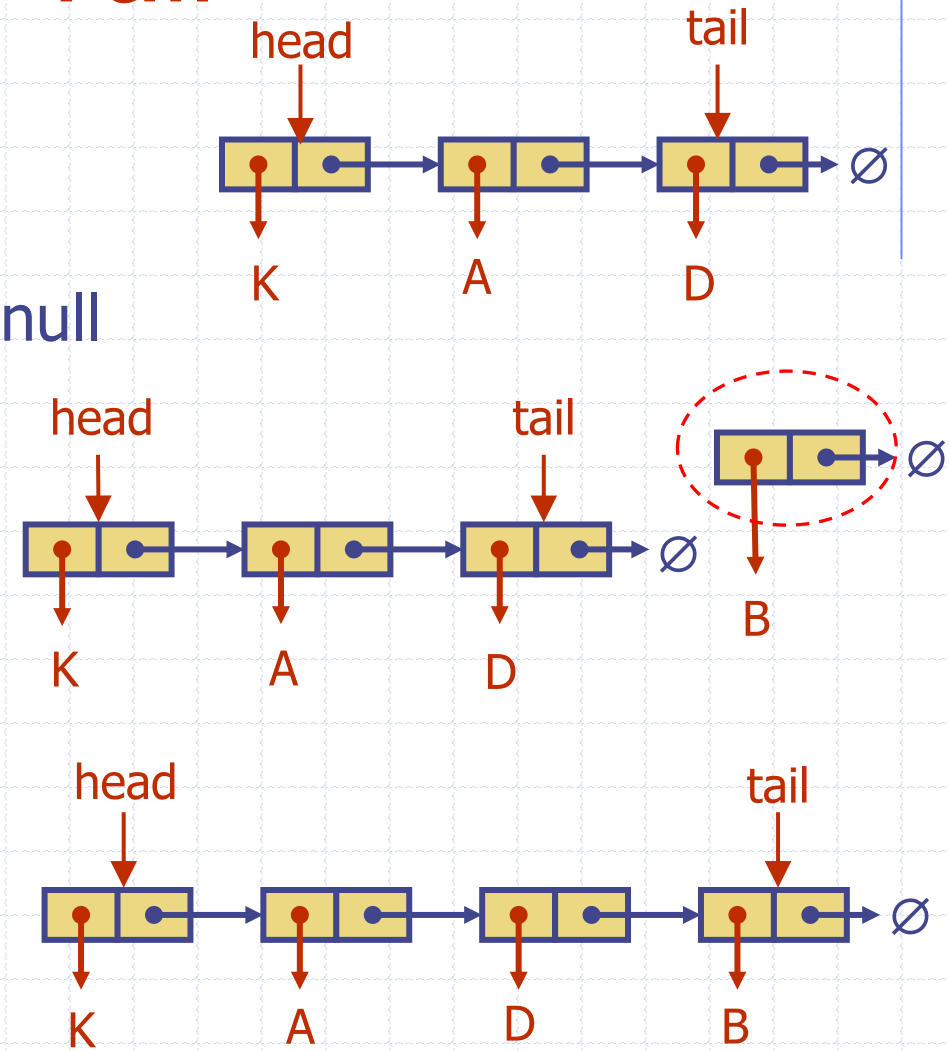
Removing at the Head

1. Update head to point to next node in the list
2. Update "size", if maintained
3. If the list is now empty, update tail, if maintained
4. Return the removed element (here, object B)
5. The "garbage collector" will reclaim the former first node



Inserting at the Tail

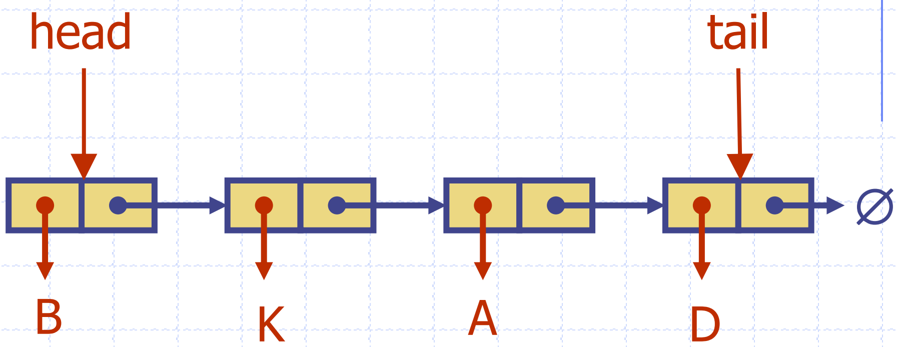
1. Allocate a new node
2. Insert new element
3. Have new node point to null
4. Have old last node point to new node (if the list wasn't empty)
5. Update tail to point to new node
6. If inserting to empty list, update head
7. Update "size", if maintained



© 2010 Goodrich, Tamassia

Removing at the Tail

- Removing at the tail of a singly linked list is not efficient
- There is no "constant-time" way to update the tail to point to the previous node



© 2010 Goodrich, Tamassia

Exercises

© 2010 Goodrich, Tamassia

Exercise 1

```
class TestProgression2 {
    public static void main(String[] args) {
        Progression prog;

        prog = new ArithProgression(5);
        prog.printProgression(5);
        prog.printProgression(7);

        prog = new GeomProgression(2);
        prog.printProgression(5);
        prog.printProgression(7);

        prog = new FibonacciProgression(3);
        prog.printProgression(5);
        prog.printProgression(7);
    }
}
```

```
>java TestProgression2
0 5 10 15 20
0 5 10 15 20 25 30
1 2 4 8 16
1 2 4 8 16 32 64
0 3 3 6 9
0 6 6 12 18 30 48
```

?

Explain why FibonacciProgression behaves differently than other subclasses.

Modify this class to achieve the expected behaviour.

Exercise 2

- In class SLinkedList<E>, show Java code for methods:

```
// return the first element, but don't remove it from the list
public  E  elementAtHead() { ... }

public  void  insertAtHead( E newElem )  { ... }
public  void  insertAtTail( E newElem ) { ... }
public  E  removeAtHead() { ... }
```

Exercise 3

Give code for method "contains" in this class:

```
public class SLinkedListExtended<E> extends SLinkedList<E> {
    // returns true if and only if, "element" is in the list
    public boolean contains(E element) { ... }

    public static void main(String[ ] args) {
        SLinkedListExtended<Integer> list =
                new SLinkedListExtended<Integer>();
        list.insertAtHead(2);   list.insertAtHead(4);  list.insertAtHead(6);
        System.out.println( "the list contains 4: " + list.contains(4));
        // prints: "the list contains 4: true "
}
```