Data Structures and Algorithms

Week 1

Java Basics

Sample Program

```
public class HelloCS526 {
    public static void main (String[ ] args) {
        System.out.println (“Welcome to CS526”);
    }
}
```

Java Basics

Primitive (or Base) Types

- Primitive types:
  - **byte**: 8-bit signed 2’s complement integer; from -128 to 127, inclusive
  - **short**: 16-bit signed 2’c complement integer; from -32768 to 32767, inclusive
  - **int**: 32-bit signed 2’s complement integer; from -2147483648 to 2147483647, inclusive
  - **long**: 64-bit signed 2’s complement integer;

    from -9223372036854775808 to 9223372036854775807, inclusive
  - **char**: 16-bit Unicode character;

    from '\u0000' to '\uffff' inclusive, that is, from 0 to 65535
  - **float**: single-precision, 32-bit floating point number (IEEE 754-1985)
  - **double**: double-precision, 64-bit floating point number (IEEE 754-1985)
  - **boolean**: true of false

Java Basics
Reference Types

- Reference types: class types, interface types, array types.
- Values of a reference type: references to objects
- A reference variable stores the location (i.e., memory address) of an object.

- Example:
  - Counter.java
  - CounterDemo.java

Java Basics

Creating a New Object

- Car c = **new Car(vin, make);**

- Use the *new* operator and the constructor.
- Memory is dynamically allocated.
- Instance variables are initialized .
- The *new* operator returns the *reference* to the new object.
- The reference is assigned to an instance variable (a reference to the object).

Java Basics

Access Control Modifier

- Also called *access level modifier* or *visibility modifier*.
- Declared for classes, variables, and methods.

Modifier	**Access Level**			
	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

Java Basics
Static Modifier

- Specified for variables or methods of a class.
- They belong to the class not to an instance of the class.
- Example:
  - Car.java
  - TestCar.java

Java Basics

Wrapper Class

- Autoboxing and autounboxing

```
public class BoxingTest {
    public static void main(String[] args) {
        Integer a = 1024; // primitive value 1024 is boxed into an object
        System.out.println("a is " + a.intValue());
        int b = a + 10; // object a is unboxed to primitive type
        System.out.println("b is " + b);
    }
}
```

Java Basics

Casting

- Narrowing vs. widening type conversion

```
double x = 3.14
int a = (int)x; // narrowing conversion from
                // double to int
double y = a;   // widening conversion from int
                //to double
```

Java Basics

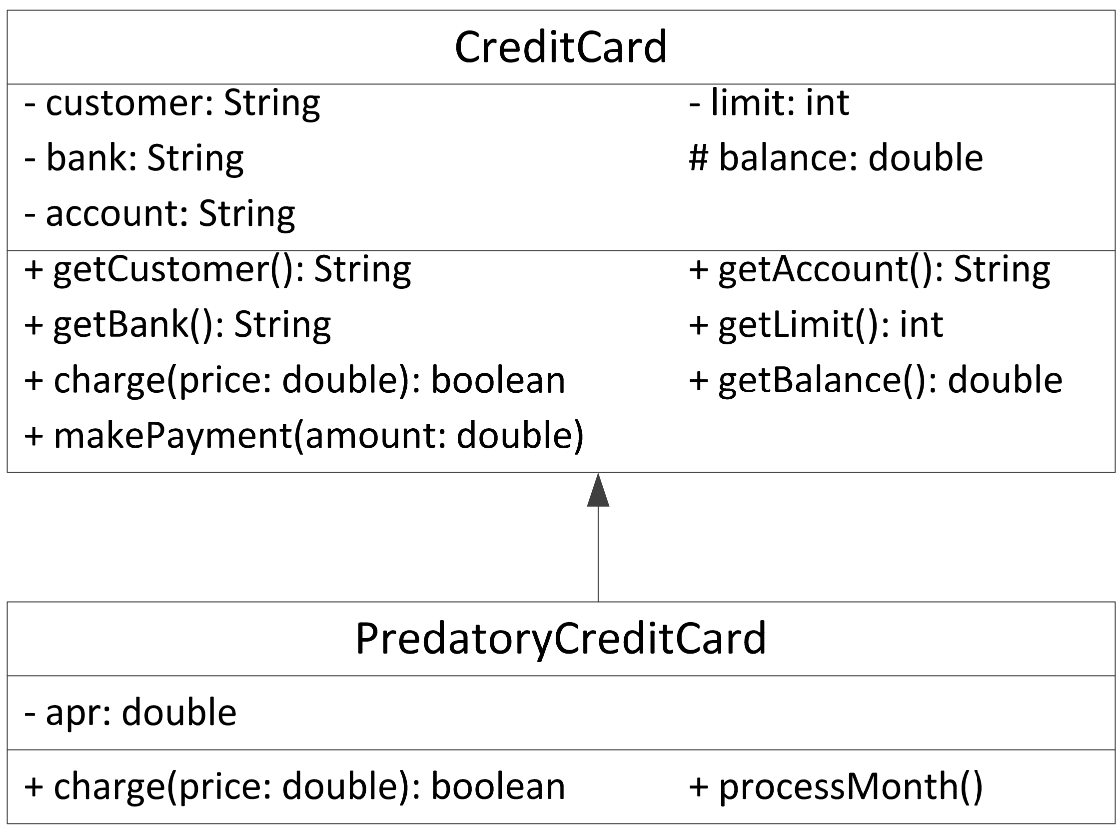
Simple I/O

- Read from standard input and write to standard output example:
  - SimpleIOTest1.java
  - SimpleIOTest2.java

- Read from a text file and write to a text file:
  - SimpleIOTest3.java

Inheritance

- Inheritance hierarchy example



Interface

- Used to specify a “contract” between different programs.
- No data.
- Methods do not have implementation.
- Cannot be instantiated.
- Can be used for multiple inheritance.
- A class implementing an interface must implements all methods.

Interface

```
/** Interface for objects that can be sold. */
public interface Sellable {
  /** Returns a description of the object. */
  public String description();
  /** Returns the list price in cents. */
  public int listPrice();
  /** Returns the lowest price in cents we will accept. */
  public int lowestPrice();
}
```

Interface

```
/** Class for photographs that can be sold. */
public class Photograph implements Sellable {
  private String descript;                        // description of this photo
  private int price;                              // the price we are setting
  private boolean color;                          // true if photo is in color
  public Photograph(String desc, int p, boolean c) {  // constructor
    descript = desc;
    price = p;
    color = c;
  }
  public String description() { return descript; }
  public int listPrice() { return price; }
  public int lowestPrice() { return price/2; }
  public boolean isColor() { return color; }
}
```

Interface

```
/** Interface for objects that can be transported. */
public interface Transportable {
  /** Returns the weight in grams. */
  public int weight();
  /** Returns whether the object is hazardous. */
  public boolean isHazardous();
}
```

Interface

```
public class BoxedItem implements Sellable, Transportable {
  private String descript;      // description of this item
  private int price;                   // list price in cents
  private int weight;                // weight in grams
  private boolean haz;           // true if object is hazardous
  private int height=0;           // box height in centimeters
  private int width=0;             // box width in centimeters
  private int depth=0;             // box depth in centimeters
 public BoxedItem(String desc, int p, int w, boolean h) {
    descript = desc;
    price = p;
    weight = w;
    haz = h;
 }
/* continue to the next slide */
```

Interface

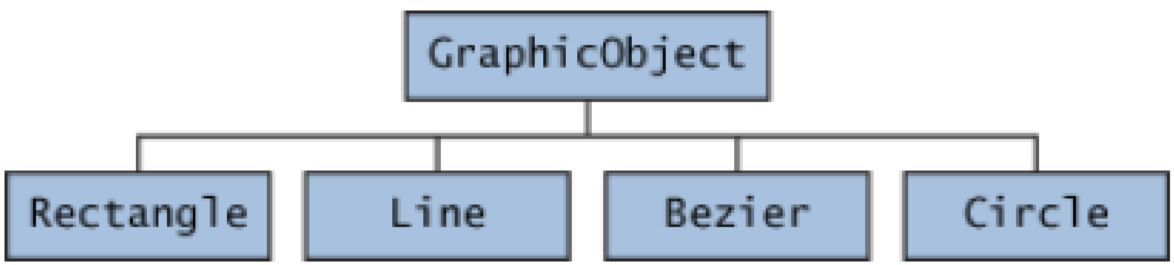
```
  public String description() { return descript; }
  public int listPrice() { return price; }
  public int lowestPrice() { return price/2;  }
  public int weight() { return weight; }
  public boolean isHazardous() { return haz; }
  public int insuredValue() { return price*2; }
  public void setBox(int h, int w, int d) {
    height = h;
    width = w;
    depth = d;
  }
}
```

Abstract Class

- An abstract method: a method without implementation.
- A concrete method: a method with implementation.
- Abstract class:
  - Declared with *abstract* keyword.
  - May or may not have abstract method.
  - A class with an abstract method must be an abstract class.
  - Used when subclasses share many common variables and methods.
  - Cannot be instantiated.

Abstract Class

- An example from Oracle documentation (https://docs.oracle.com/javase/tutorial/java/IandI/abstract.html)



Classes Rectangle, Line, Bezier, and Circle Inherit from GraphicObject

Abstract Class

```
abstract class GraphicObject {
    int x, y;
    . . .
    void moveTo(int newX, int newt) {
        . . .
    }
    abstract void draw();
    abstract void resize();
}
```

Abstract Class

```
class Rectangle extends GraphicObject {
    void draw() {
        // implementation
        . . .
    }
    void resize() {
        // implementation
        . . .
    }
}
```

Interface and Abstract Class

- Consider using interfaces if any of these statements apply to your situation:
  - You expect that unrelated classes would implement your interface. Example: interfaces Comparable and Cloneable in Java
  - You want to specify the behavior of a particular data type, but not concerned about who implements its behavior.
  - You want to take advantage of multiple inheritance of type.

Interface and Abstract Class

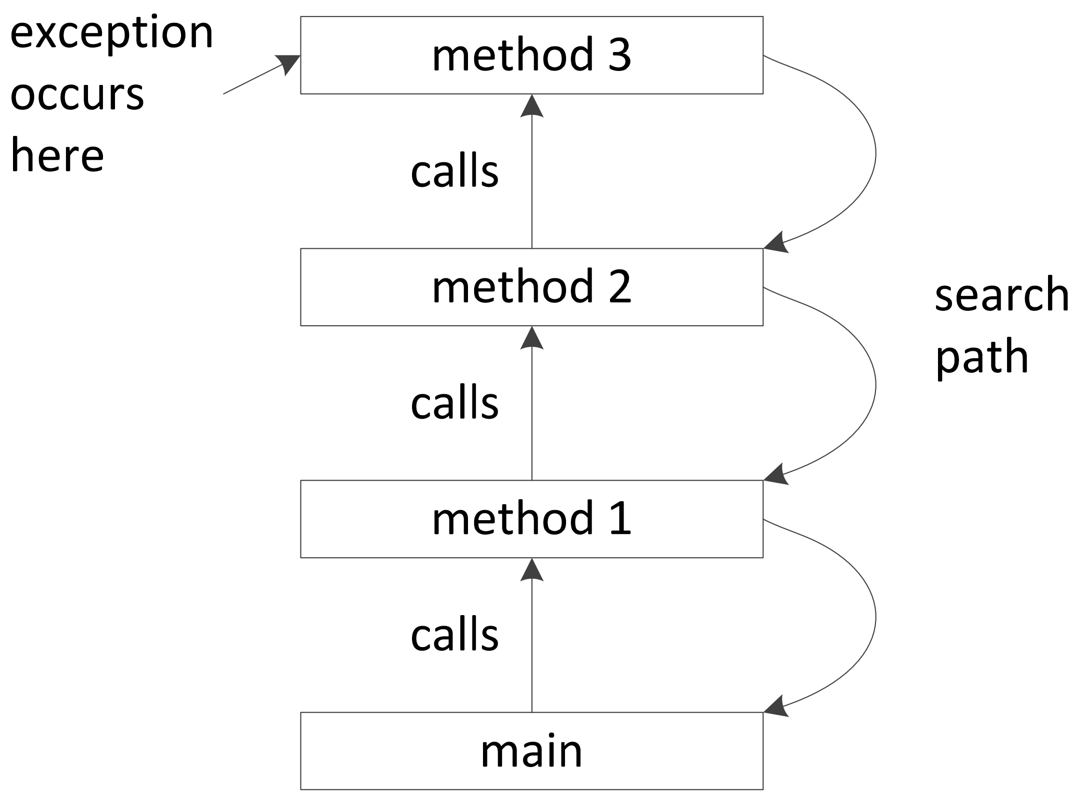
- Consider using abstract classes if any of these statements apply to your situation:
  - You want to share code among several closely related classes.
  - You expect that classes that extend your abstract class have many common methods or fields.

Exceptions

- An *exception*, shorthand for *exceptional event*, is an event that occurs during the execution of a program
- When an exception occurs
  - an exception is *thrown*
  - the runtime system finds an *exception handler*
  - the code in the handler is executed

Exceptions

- Call stack and exception handler search path



Exceptions

- Try-catch statement

try {

guardedBody

} catch (*exceptionType*$_1$ *variable*$_1$) {

remedyBody$_1$

} catch (*exceptionType*$_2$ *variable*$_2$) {

remedyBody$_2$

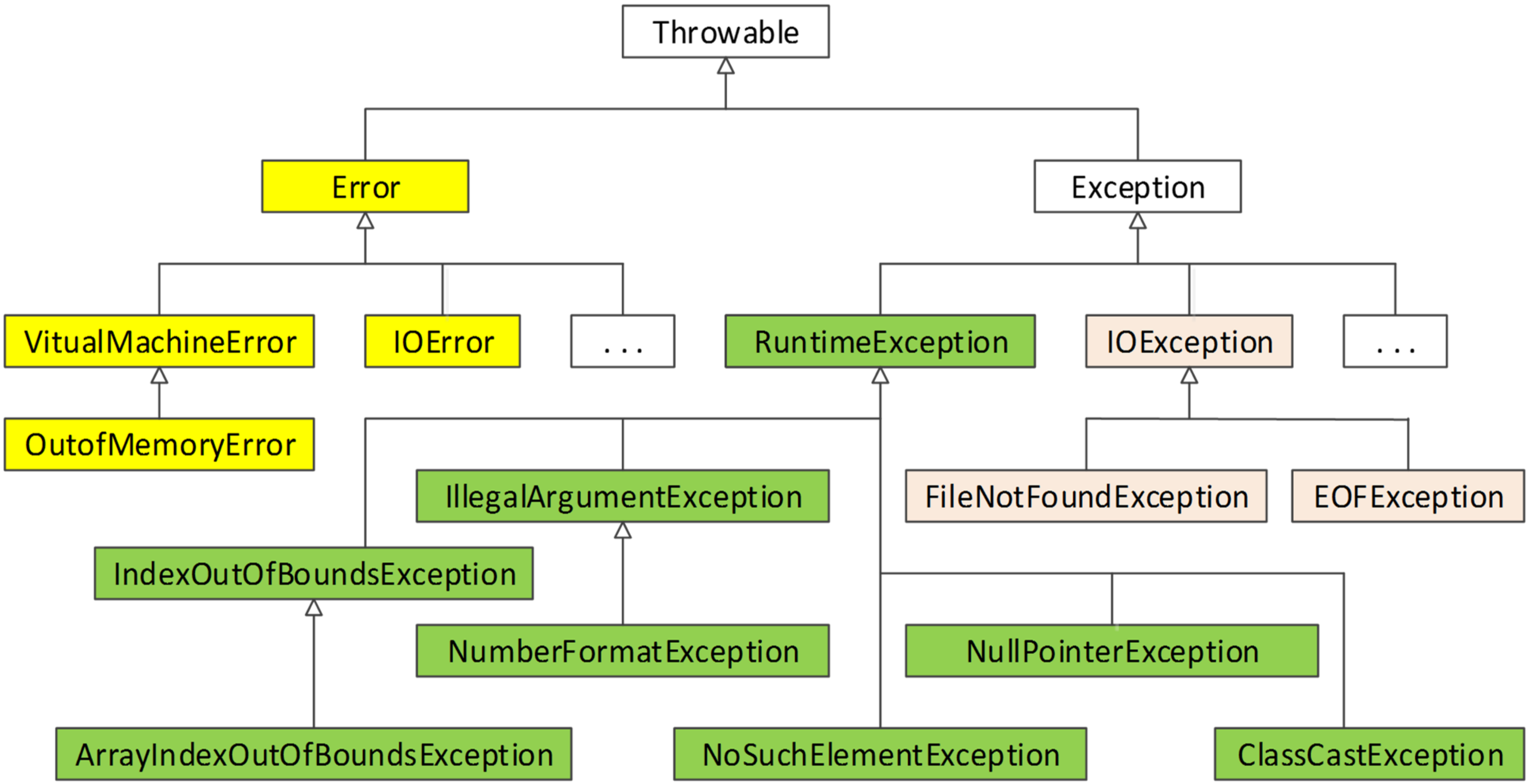
} . . .

. . .

- Example: [ExceptionDemo.java]()

Exceptions

- Java Exception Hierarchy (part)



Exceptions

- *Errors*:
  - exception objects of the *Error* class and all of its subclasses.
  - external to the application and they are thrown by JVM.
- *Runtime exceptions*:
  - exception objects of the *RuntimeException* class and all of its subclasses.
  - exceptional events internal to the application, and that the application usually cannot anticipate or recover from.
- *Checked exceptions*:
  - all other exceptions
  - If a code may throw a *checked exception*, then it must be in a *try-catch* statement or it must be in a method which is declared with a *throws* clause.

Generics

- Types can be declared using generic names:

```
public class Pair<A,B> {
    A first;
    B second;
    public Pair(A a, B b) {              // constructor
        first = a;
        second = b;
    }
    public A getFirst() { return first; }
    public B getSecond() { return second;}
}
```

- They are then instantiated using actual types

```
Pair<String, Double> bid;          // declare
bid = new Pair<>(“pi”, 3.14);    // instantiate
```

Generics

- Generics and arrays

  – Case 1: We have a generic class with parameterized types. We want to declare, outside of the generic class, an array storing objects of the generic class with actual type parameters.

  – Case 2: We have a generic class with parameterized types. We want to declare, as an instance variable of the class, an array storing objects of one of the formal parameter types.

Generics

- Generics and arrays – Case 1

```
1  Pair<String, Double>[ ] holdings; // declaring with actual type
                                     // type parameters are allowed
2  holdings = new Pair<String, Double>[25]; // illegal
3  holdings = new Pair[25];        // this is allowed
4  holdings[0] = new Pair<>("ORCL", 3.14); // this is legal
```

Generics

- Generics and arrays – Case 2

```
public class Portfolio<T> {
    T[ ] data;
    public Portfolio(int capacity) {
       data = new T[capacity];                // illegal; compiler error
       data = (T[ ]) new Object[capacity];    // legal, but compiler
                                              // warning
    }
    public T get(int index) { return data[index]; }
    public void set(int index, T element) { data[index] = element; }
}
```

Generics

- Generic method

```
public class GenericDemo {
    public static <T> void reverse(T[ ] data) {
        int low = 0, high = data.length - 1;
        while (low < high) {          // swap data[low] and data[high]
            T temp = data[low];
            data[low++] = data[high];      // post-increment of low
            data[high--] = temp;           // post-decrement of high
        }
    }
}
```

Generics

- Generic method

```
String[ ] names = new String[ ]{"john", "susan", "molly"};
GenericDemo.reverse(names);

Integer[ ] integers = new Integer[ ]{10, 20, 30, 40, 50};
GenericDemo.reverse(integers);

Character[ ] chars = new Character[ ]{'a', 'b', 'c', 'd', 'e'};
GenericDemo.reverse(chars);
```

Generics

- Demonstration
  - GenericQueue.java
  - GenericDemo1.java
  - GenericDemo2.java
  - GenericDemo3.java

Nested Class

```
class OuterClass {
    ...
    class NestedClass {
        ...
    }
}
```

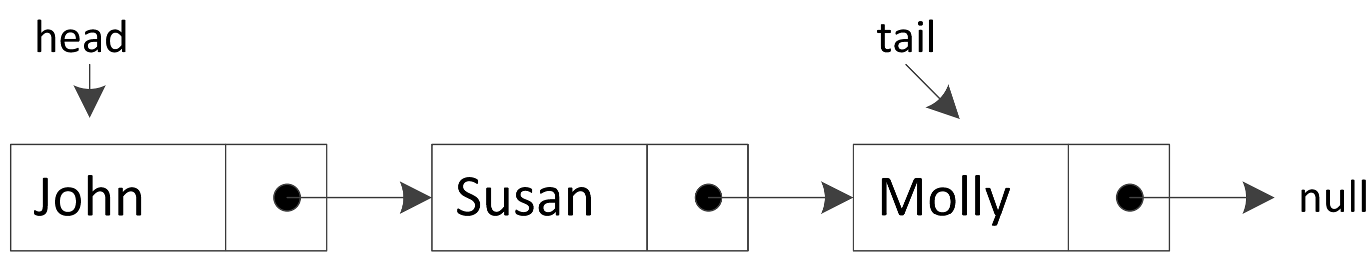
Nested Class

- We use nested classes for the following reasons:
  - *NestedClass* is used only for the *OuterClass*.
  - We want to declare members of the *OuterClass* as private but, at the same time, we want *a* smaller class to be able to access members of the *OuterClass*.
  - We want to implement a data structure which has another smaller data structure as its member.

- The code becomes more readable and it is easy to maintain.
- Nested classes also help reduce name conflict.

Singly Linked Lists

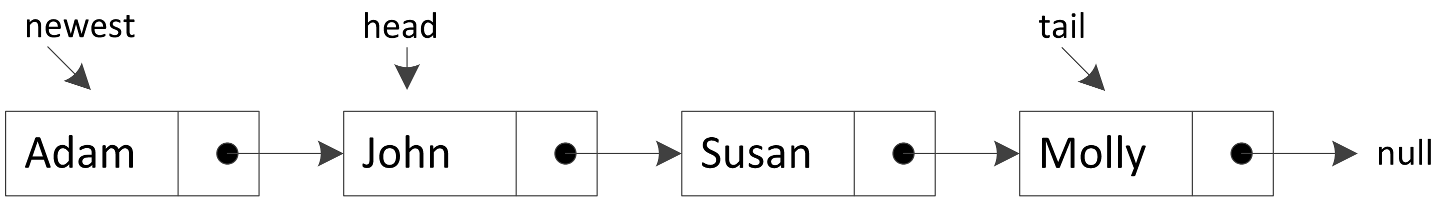
- Nodes are connected by links.
- Singly linked list, circularly linked list, doubly linked list
- A node has *element* and the reference (or pointer) to the next node.
- We usually keep two additional references, *head* and *tail*, for a singly linked list.



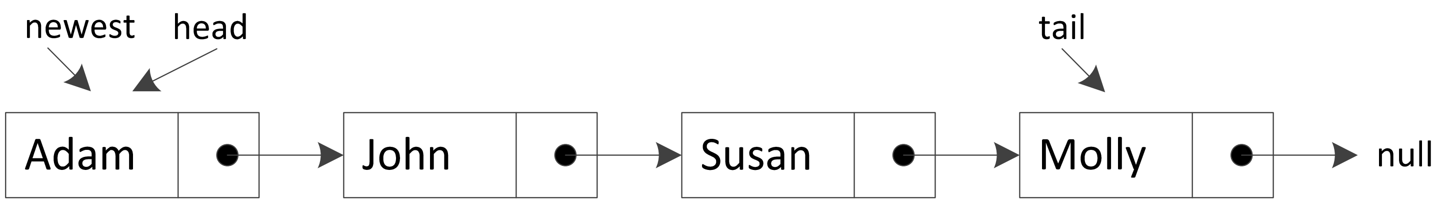
Singly Linked Lists

- Add a node to the head of a list

newest = Node("Adam"); newest.next = head;



head = newest;



Singly Linked Lists

- Add a node to the head of a list

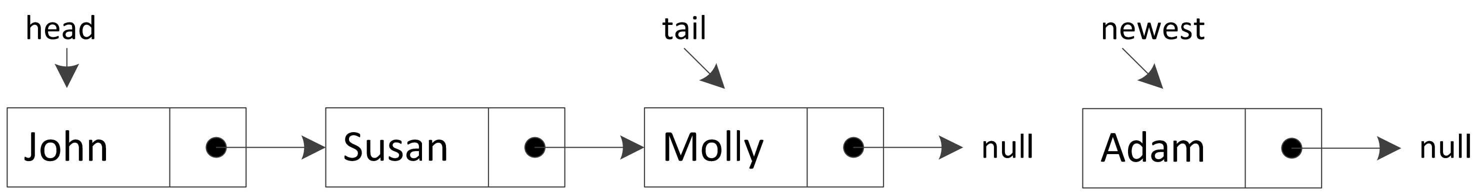
Algorithm addFirst(*e*)

```
newest = Node(e) // new node with element e
newest.next = head // new node's next is set to refer
                   // to current head node
head = newest   // new head refers to new node
size = size + 1    // list size (node count) is
                   // incremented
```

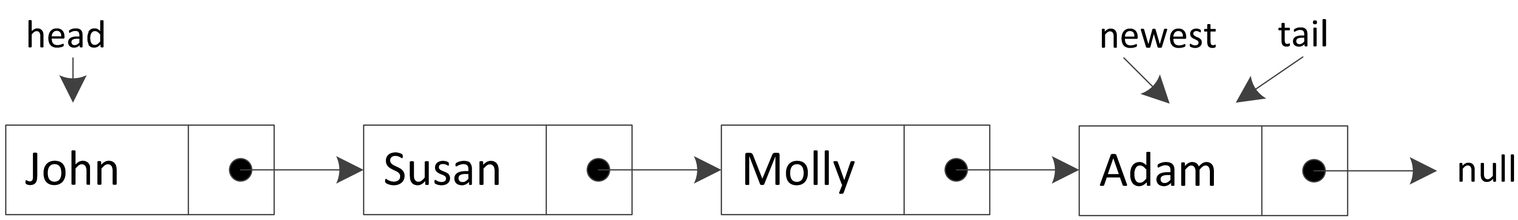
Singly Linked Lists

- Add a node to the tail of a list

newest = Node(“Adam”); newest.next = null ;



tail.next = newest; tail = newest;



Singly Linked Lists

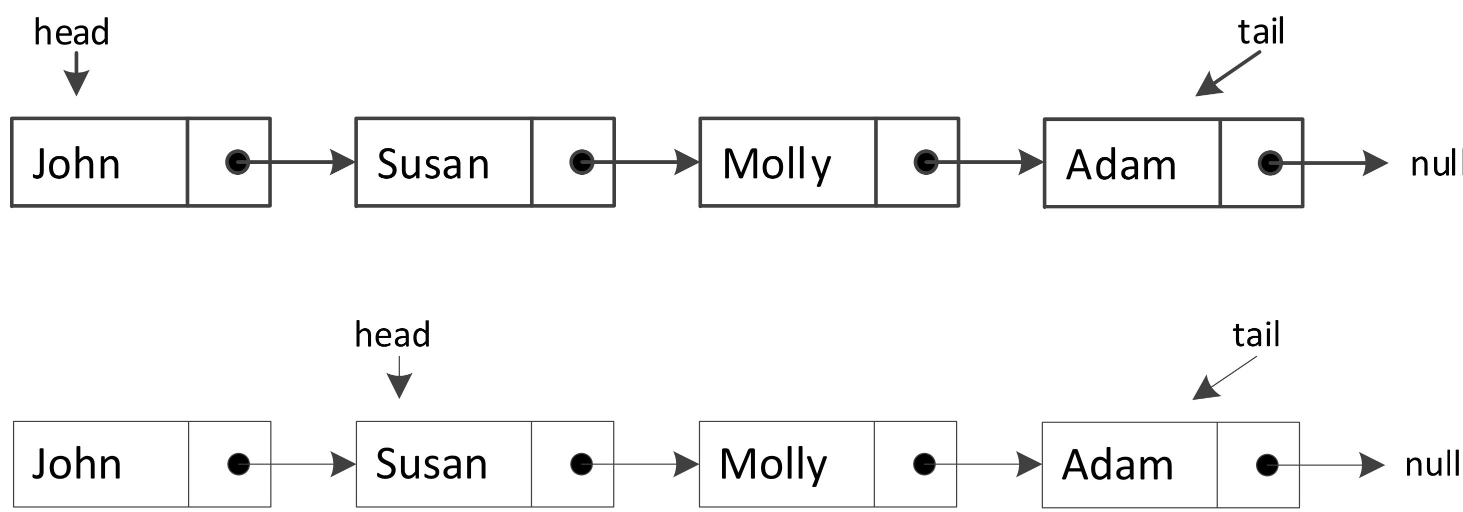
- Add a node to the tail of a list

Algorithm addLast(*e*)

```
newest = Node(e)   // new node with element e
newest.next = null   // new node's next is set to null
tail.next = newest // current tail node's next points to
                   // new node
tail = newest      // new tail points to new node
size = size + 1    // list size (node count) is
                   // incremented
```

Singly Linked Lists

- Removing an arbitrary node: nontrivial and inefficient.
- Removing a node from the head of a list



Singly Linked Lists

```
Algorithm removeFirst( )
    if head == null
        the list is empty
    head = head.next   // new head points to next node
    size = size - 1    // list size (node count) is
                       // decremented
```

Singly Linked Lists

- Generic *Node* class in *SinglyLinkedList* class

```
private static class Node<E> {
    private E element;
    private Node<E> next;
    public Node(E e, Node<E> n) {
        element = e;
        next = n;
    }
    public E getElement() { return element; }
    public Node<E> getNext() { return next; }
    public void setNext(Node<E> n) { next = n; }
}
```

Singly Linked Lists

- Instance variables of *SinglyLinkedList* class

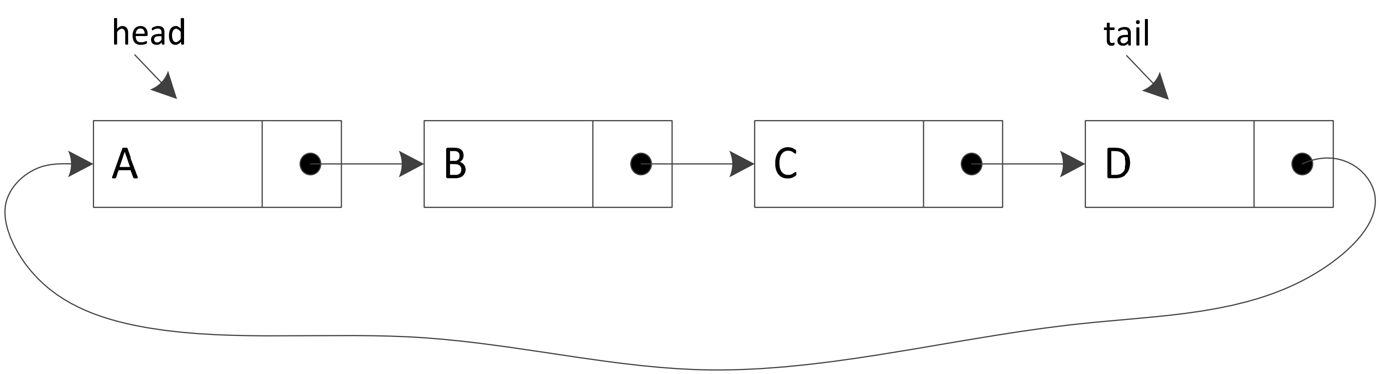
```
public class SinglyLinkedList<E> implements Cloneable
{
    // nested class Node
    protected Node<E> head = null;
    protected Node<E> tail = null;
    protected int size = 0;
    // constructors and methods
```

Singly Linked Lists

- Complete code of *SinglyLinkedList.java*

Circularly Linked Lists

- A singly linked list where the last element is connect to the first element, forming a circle.
- Used in application where objects are manipulated in a round-robin manner, such as process scheduling.



- Don’t need to keep the *head* reference.

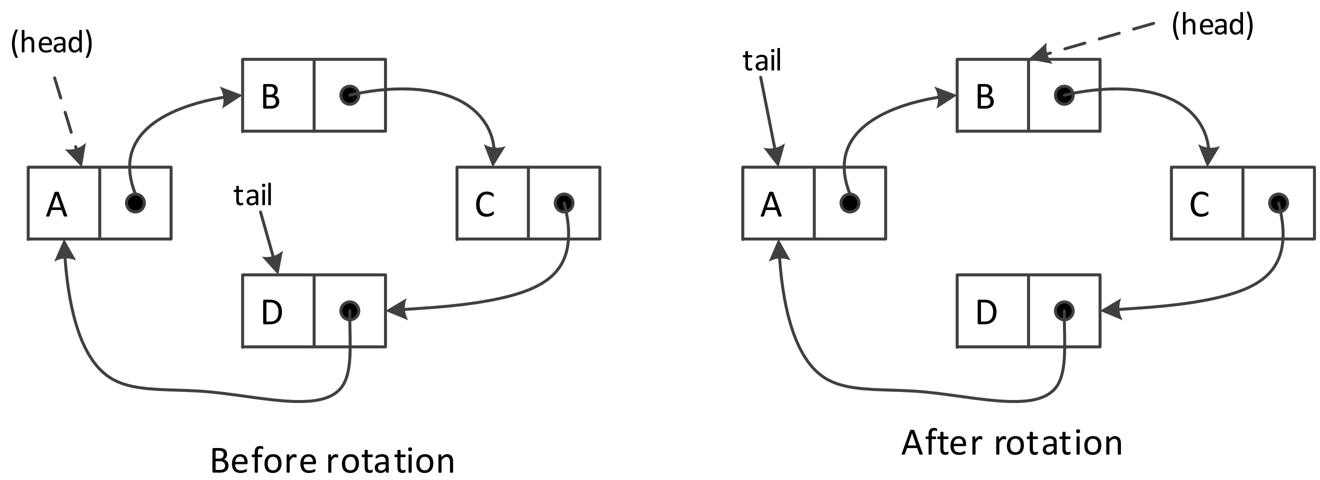
Circularly Linked Lists

- Round-robin process scheduling: Processes are executed by CPU one at a time for a *slice* of time.

```
// C is a circularly linked list
Give a time slice to C.first()
C.rotate()
```

Circularly Linked Lists

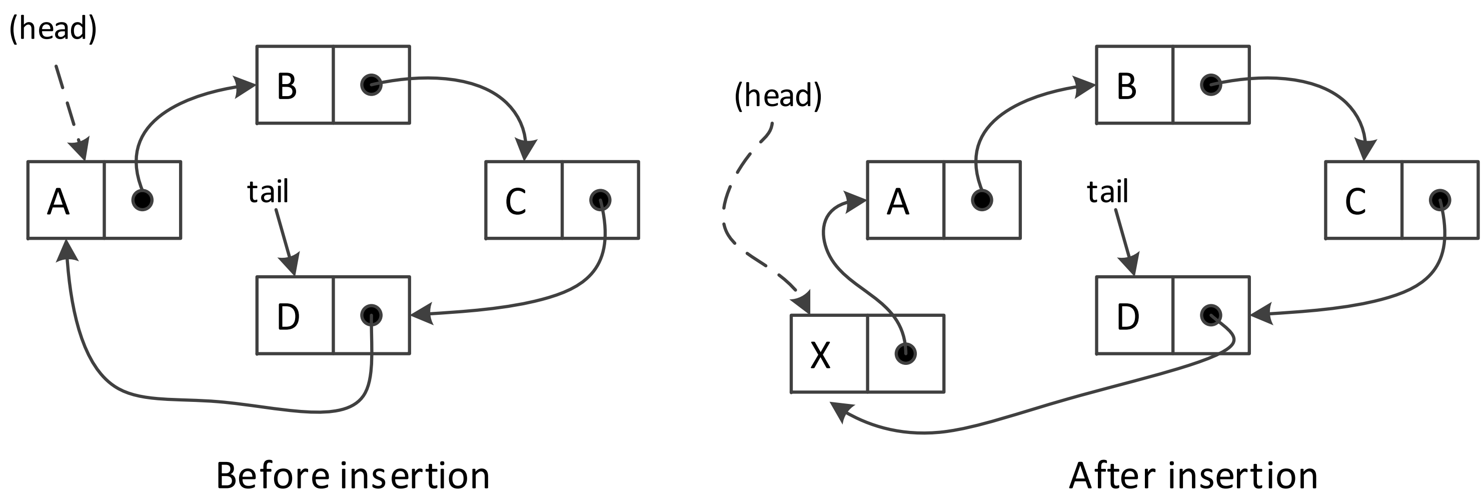
- Rotate operation




```
public void rotate() {        // rotate the first element to the back of the list
    if (tail != null)                // if empty, do nothing
      tail = tail.getNext();     // the old head becomes the new tail
  }
```

Circularly Linked Lists

- Add a node to before head (this is the same as adding a node after tail)



Circularly Linked Lists

- Can reuse most of *SinglyLinkedList* code.
- The *addFirst* method is modified

```
public void addFirst(E e) {     // adds element e to the front of the list
    if (size == 0) {
      tail = new Node<>(e, null);
      tail.setNext(tail);                   // link to itself circularly
    } else {
      Node<E> newest = new Node<>(e, tail.getNext());
      tail.setNext(newest);
    }
    size++;
  }
```

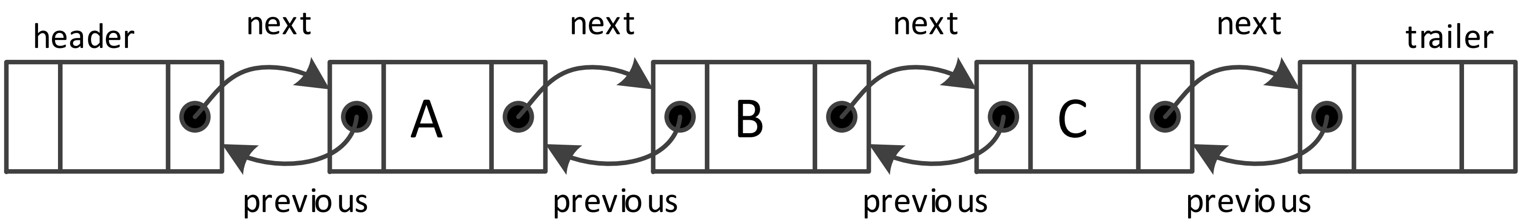
Doubly Linked Lists

- Singly linked list:
  - Not easy to insert a node at an arbitrary position.
  - Nontrivial to delete a node from an arbitrary position.

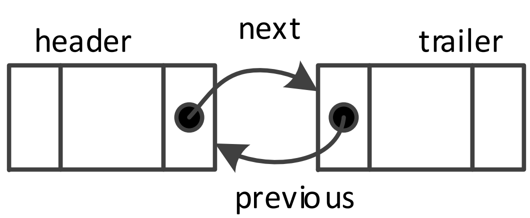
- These operations, however, can be performed relatively efficiently with doubly linked lists.

Doubly Linked Lists

- Doubly linked list example

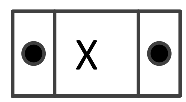


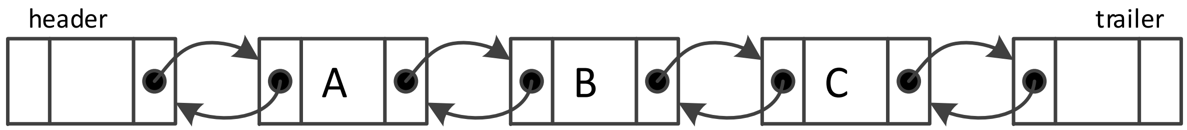
- An empty list



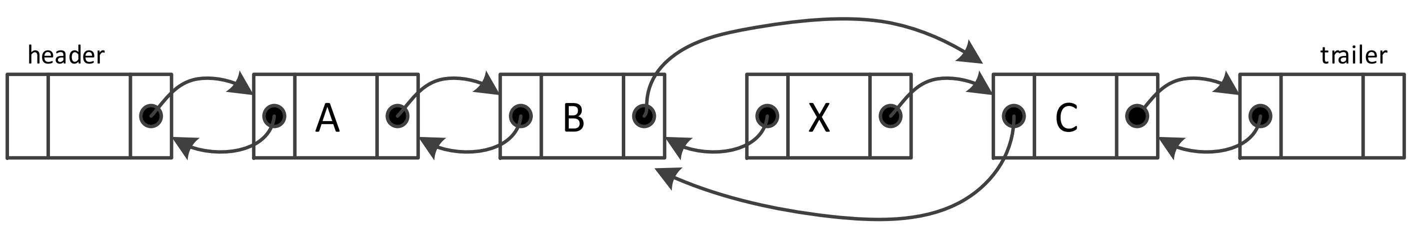
Doubly Linked Lists

- Insert a new node X between B and C



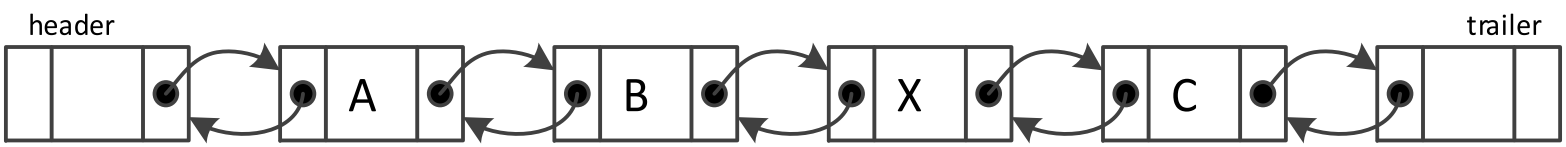


- The previous reference of X are set to point to B.
- The next reference of X are set to point to C.



Doubly Linked Lists

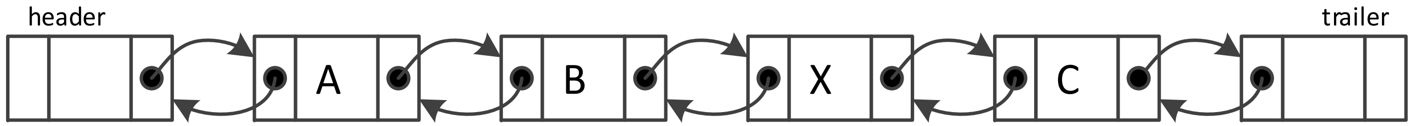
- The next reference of B and the previous reference of C are updated to point to X.



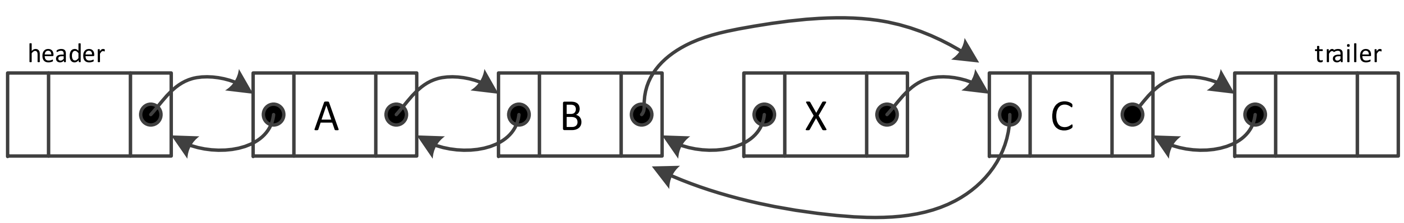

```
private void addBetween(E e, Node<E> predecessor,
                                                  Node<E> successor)  {
Node<E> newest = new Node<>(e, predecessor, successor);
predecessor.setNext(newest);
successor.setPrev(newest);
size++;
}
```

Doubly Linked Lists

- Delete X

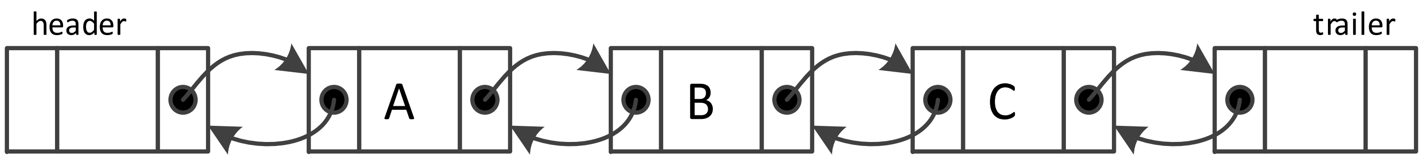


- Set the next reference of B to point to C
- Set the previous reference of C to point to B.



Doubly Linked Lists

- X is not a part of the list any more. The updated list is:




```
private E remove(Node<E> node) {
    Node<E> predecessor = node.getPrev();
    Node<E> successor = node.getNext();
    predecessor.setNext(successor);
    successor.setPrev(predecessor);
    size--;
    return node.getElement();
}
```

Doubly Linked Lists

- A complete code of *DoublyLinkedList.java*

Insertion Sort

- There are different sorting algorithms.
- Will discuss insertion sort algorithm (on an array).
- Pseudocode

Algorithm InsertionSort(*A*)

Input: Array *A* of *n* comparable elements

Output: Array *A* with elements rearranged in dondecreasing order

for *k* from 1 to *n* – 1**do**

Insert *A*[*k*] at its proper location within *A*[0 .. *k*]

Insertion Sort

- Illustration

k = 1

17	6	12	32	24	8	14	11
0	1	2	3	4	5	6	7

k = 2

6	17	12	32	24	8	14	11
0	1	2	3	4	5	6	7

k = 3

6	12	17	32	24	8	14	11
0	1	2	3	4	5	6	7

k = 4

6	12	17	32	24	8	14	11
0	1	2	3	4	5	6	7

k = 5

6	12	17	24	32	8	14	11
0	1	2	3	4	5	6	7

k = 6

6	8	12	17	24	32	14	11
0	1	2	3	4	5	6	7

k = 7

6	8	12	14	17	24	32	11
0	1	2	3	4	5	6	7

6	8	11	12	14	17	24	32
0	1	2	3	4	5	6	7

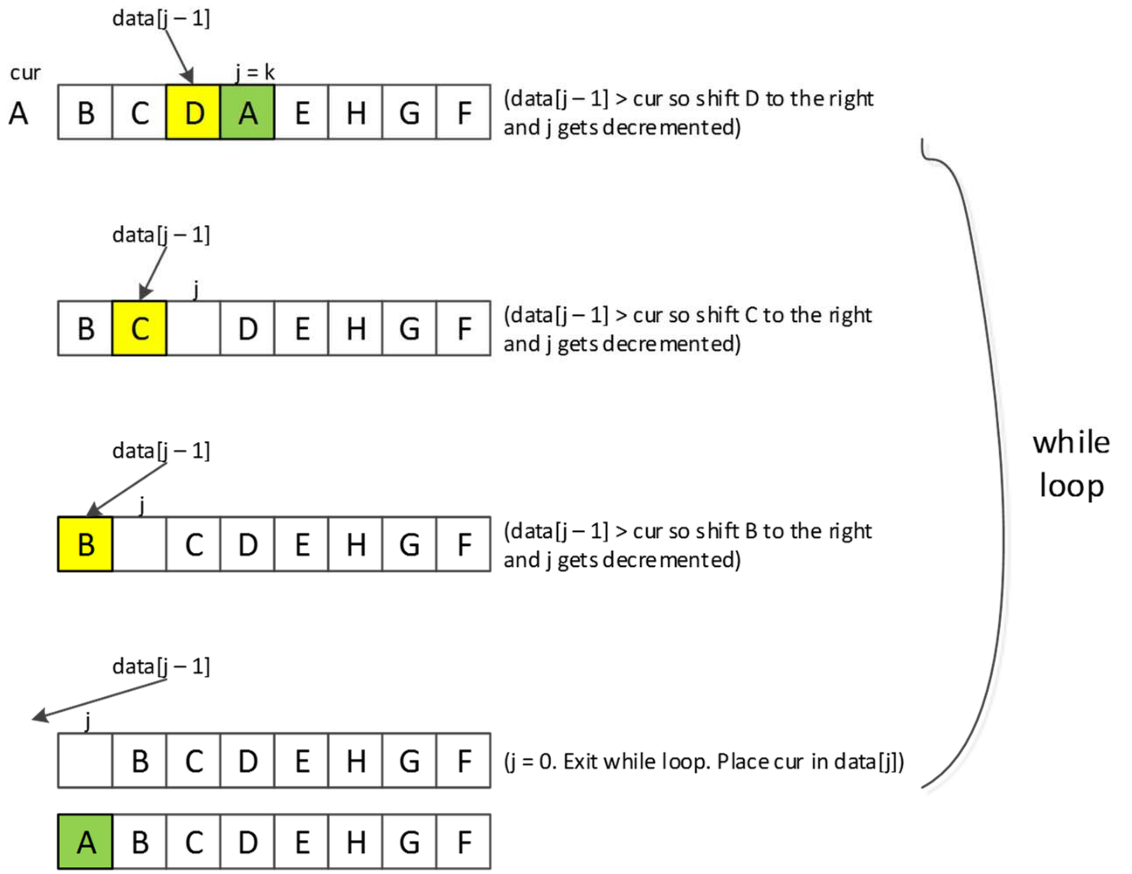
Insertion Sort

- Java implementation

```
public class InsertionSort {
  public static void insertionSort(char[] data) {
    int n = data.length;
    for (int k = 1; k < n; k++) {    // begin with second element
        char cur = data[k];              // save data[k] in cur
        int j = k;                       // find correct index j for cur
        while (j > 0 && data[j-1] > cur) {  // thus, data[j-1] must go after cur
            data[j] = data[j-1];              // shift data[j-1] rightward
            j--;                              // and consider previous j for cur
       } // while
       data[j] = cur;                    // this is the proper place for cur
    } // for
}                                        // running time: O(n^2)
```

Insertion Sort

- Illustration of while loop in Java implementation



Testing Equality

- When comparing two reference variables, there are two notions of equivalence.
- First interpretation: Test whether two reference variables are pointing to the same object.
- Second interpretation: Test whether the contents of the two objects pointed to by the references are the same.

```
String s1 = new String("data structure");
String s2 = new String("data structure");
```

- Is s1 equal to s2?
  - No, by the first interpretation
  - Yes, by the second interpretation

Testing Equality

- In Java, you can compare with “==“ operator or using the *equals* method.
- “==“ compares the values of the reference variables, i.e., it checks whether they refer to the same object.
- The *equals* method is defined in the *Object* class, and, as it is, it is effectively the same as “==“ operator.
- To implement the “second interpretation” for objects of a class, the class must define its own *equals* method tailored for the objects of that class.

Testing Equality

- String class has *equals* method which performs character-by-character, pair-wise comparison.

```
public class StringTest {
    public static void main(String[] args) {
        String s1 = new String("data structure");
        String s2 = s1;
        String s3 = new String("data structure");
        System.out.println("reference s1 equals reference s2: " + (s1 == s2));
        System.out.println("reference s1 equals reference s3: " + (s1 == s3));
        System.out.println("string s1 equals string s3: " + s1.equals(s3));
    }
}
```

- Output: true, false, true

Testing Equality

- Equivalence testing with arrays
  - a == b: Tests if a and b refer to the same array instance.
  - a.equals(b): This is identical to a == b.
  - Arrays.equals(a, b): Returns true if the arrays have the same number of elements and all pairs of corresponding elements are equal to each other. If elements are primitives, == operator is used. If elements are reference types, then *a*[*k*].*equals*(*b*[*k*]) is used.

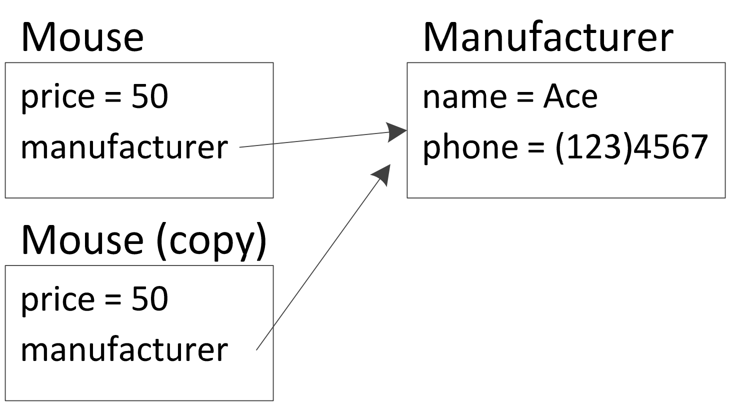
Testing Equality

- Equivalence testing with linked lists
  - Traverse two lists and compare pairs of corresponding elements.
  - Refer to the *equals* method in the *SinglyLinkedList* class.

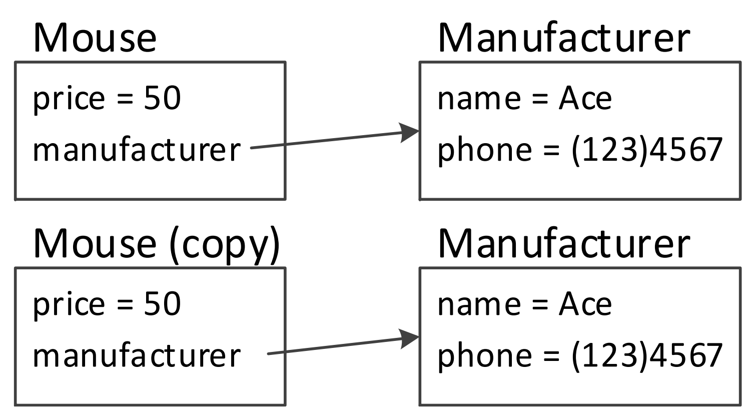
Cloning Data Structures

- Shallow copy vs. deep copy

Shallow copy



Deep copy



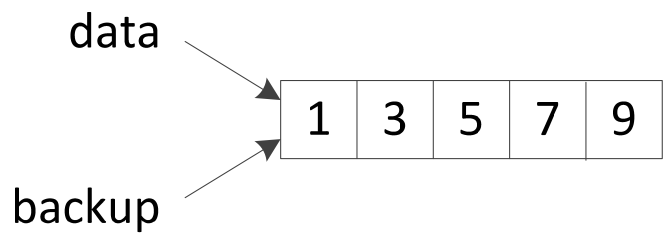
Cloning Data Structures

- Java's Object class has the *clone* method.
- This clone method creates a shallow copy.
- If necessary, each class must define its own clone method.

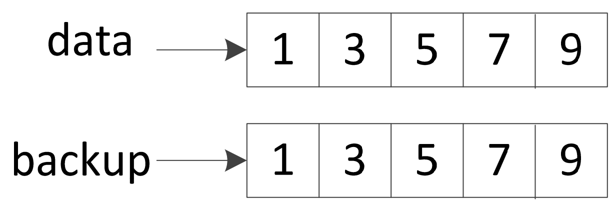
Cloning Data Structures

- Cloning arrays with elements of primitive type

```
int[] data = {1,3,5,7,9};
int[] backup;
backup = data;
```



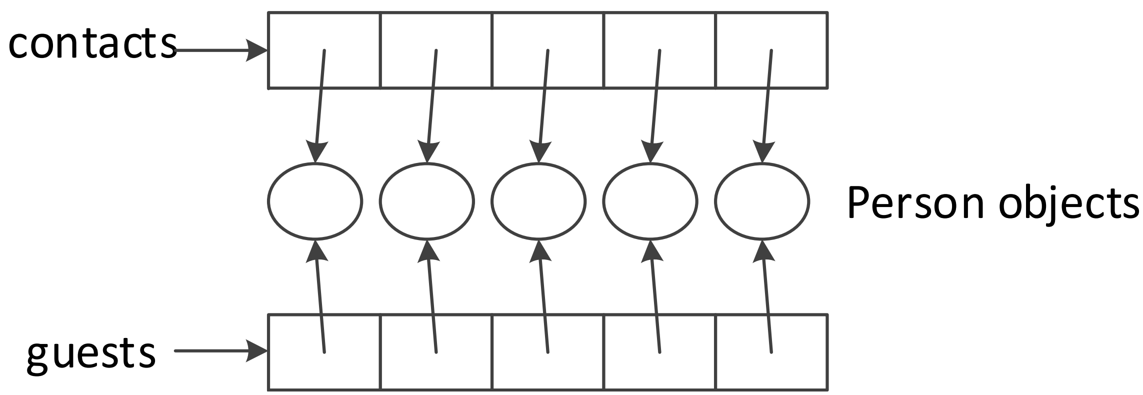

```
backup = data.clone();
```



Cloning Data Structures

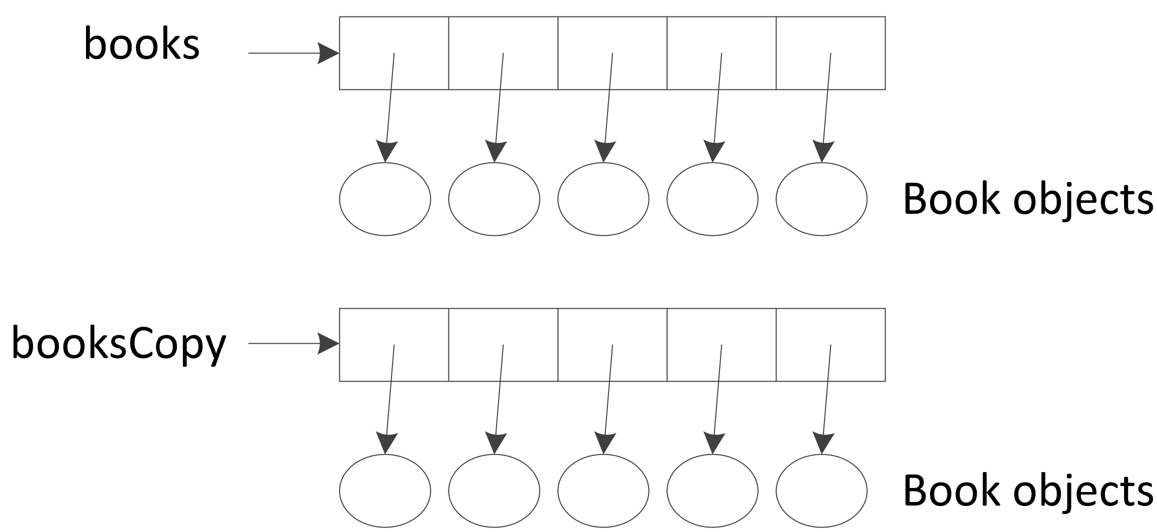
- Cloning arrays with elements of object type

guests = contacts.clone( ); // a shallow copy is created



Cloning Data Structures

- Cloning arrays with elements of object type
  - The following is a deep copy.
  - A separate code must be written.



Cloning Data Structures

- Cloning linked lists:
  - Must copy one node at a time.
  - Refer to the *clone* method in *SinglyLinkedList* class.

References

- M.T. Goodrich, R. Tamassia, and M.H. Goldwasser, “Data Structures and Algorithms in Java,” Sixth Edition, Wiley, 2014.