

A Second Course in Java

Brief Notes for Computer Science 102

©Wayne Goddard (Clemson University) 2006

Java Review

1.1 Primitive Data Types and Operators

In Java, the primitive data types include `int`, `double`, `boolean`, `char`. An `int` variable stores a 32-bit integer which can be positive or negative. This can be initialized at declaration:

```
int count = 0;
```

There are operators for much arithmetic. The operator `%` is the *mod* operator: it calculates the remainder; for example, `11%3` has the value 2.

The increment operator `++` and decrement operator `--` change the value of the variable they are applied to. For example,

```
System.out.println(x++);
```

prints out the value of `x` and THEN increments it.

1.2 Conditionals and Loops

The standard conditional is an *if* construction or an *if-else* construction.

```
if(condition)done-if-true;
```

```
if(condition)done-if-true;
else done-if-false;
```

The standard loops are the *for-loop* and the *while-loop*.

```
for(initialization; condition; post-body action)
  { statements to be repeated }
```

```
initialization
while(condition){
  statements to be repeated
  post-body action
}
```

Nested loops are when there is one loop inside another.

1.3 Arrays

An *array* has a fixed length and the length is a field (instance variable). For example,

```
int[] hourCount = new int[24] ;
```

produces an array of integers with indices from 0 up to 23. Arrays are automatically initialized (e.g. ints to 0).

Example: determine the maximum entry in the above array.

```
int maxSoFar = hourCount[0];
for( int x=1; x<hourCount.length; x++ )
    if( hourCount[x]>maxSoFar )
        maxSoFar = hourCount[x];
```

But what happens if `hourCount` has length 0?

An array can be initialized with a braces construct:

```
double[] data = { 1.414, 3.1415, 1.00, -42 };
```

This can be printed out with the `for-each` construct:

```
for( double D : data)
    System.out.printf("The next value is %d\n", D );
```

1.4 Booleans

A boolean variable stores either `true` or `false`. It is good practice to make its name a verb phrase e.g. “`isPositive`” or “`hasMoreEntries`”. It should NOT be tested against `true` or `false`.

```
while ( hasMoreEntries )
    printOutNext;
if( !isPositive )
    changeSign;
```

Boolean expressions use `&&` for *and*, `||` for *or*, `!` for *not*. Note that `&&` and `||` are *short-circuit* operators: these enable one to check for conditions which might cause exceptions: e.g. an array index out of range, or division by zero:

```
if ( divisor!=0 && val/divisor<=50 )
```

1.5 Strings and Characters

A `String` is an immutable object with `length()` a method. The `+` operator denotes concatenation. Other available methods include `substring`, `charAt`, `trim`, `toLowerCase`. Always look at the help page for the `String` class.

A `char` variable stores a single character. (Conversion from `char` to `int` is automatic.) Character arithmetic is available. The following converts a letter to its position in the alphabet.

```
char lowerCaseLetter;  
...  
int pos = lowerCaseLetter - 'a' + 1;
```

1.6 Program Structure

If a Java class is to be run as a program, then it must have a main method declared a specific way:

```
public static void main(String[] args){
```

In a method, a `return` statement causes execution to halt, and to return a value as required.

Exercise

Write a program that will print out all two-digit numbers which equal twice the sum of the squares of their digits.

Sample Program

```
// ArrayExample.java  
// a silly few array methods for cs102  
// wdg 2006  
  
public class ArrayExample {  
  
    public static void main(String[] args){  
        String[] test = {"aaaa", "abba", "eiEIo"};  
        for(int x=0; x<test.length; x++){  
            System.out.print(test[x]);  
            System.out.print( isAllDiff(test[x]) ? " Different." : " Has repeat." );  
            System.out.print(" Lower vowels are ");
```

```

        printLowerVowels(test[x]);
    }
}

// prints out the lowercase vowels in the string in order
// @post: vowels are printed on standard output with line feed

public static void printLowerVowels(String S){
    for(int x=0; x<S.length(); x++){
        char temp=S.charAt(x);
        if(temp=='a' || temp=='e' || temp=='i' || temp=='o' || temp=='u')
            System.out.print(temp);
    }
    System.out.println();
}

// determines whether there are any duplicate characters in a string
// @parameter: String S - input
// @returns: true if every two chars are different, else false

public static boolean isAllDiff(String S){
    char[] A = S.toCharArray();
    boolean isOkay=true;
    for(int x=0; x<A.length; x++)
        for(int y=x+1; y<A.length; y++)
            if( A[x]==A[y] )
                isOkay=false;
    return isOkay;
}
}

```

Basic Objects

2.1 Terminology

Classes define types. Objects are concrete instances of class and there may be multiple instances of a class. Objects store data in *fields* or *instance variables*, initialize this data with *constructors*, and manipulate this data with *methods*. The current value of the fields is the *state* of the object. A variable of class type stores either a *reference* to an object or the value null.

2.2 Constructors and Methods

A constructor sets up an object at creation. For example:

```
Prof wayne = new Prof();
```

A constructor can have arguments that are used to initialize the instance variables:

```
Prof wayne=new Prof("goddard");
```

A constructor must NOT have a type:

```
public class Prof {
    String name;
    Prof(String n) {
        name = n;
    }
}
```

When a class is created, instance variables are initialized to default values: 0 for int, null for objects etc. Nevertheless, you should explicitly set primitive variables.

A method has a *header* with formal *parameters*, and a *body*. Its *signature* is the types of its parameters and its own type. The return type `void` means that the method returns nothing. A method is called with the `.` notation. For example:

```
object.method();
```

Most objects have *accessor* methods: these allow the user to *get* data from the object. *Mutator* methods allow the user to *set* data in the object.

A method should normally check its arguments. It notifies the caller of a problem by using an *exception* or a special return value. The programmer should try to avoid exceptions: consider error recovery and avoidance.

2.3 Access Modifiers

The fields of an object are accessible by all the methods of that object. A variable has a *scope* (where it is accessible from) and a *lifetime* (when it exists). The variables defined in a method are called *local variables* and are accessible only within the method and exist only while the method is being executed; an exception is *static* variables whose lifetime is the program's execution: they are always available.

A method or field can be made *private*, *public* or *protected*. *private* means it is accessible by no other class, and *public* that it is accessible by all other classes. (We will discuss *protected* later.) Note that within the code for the class, one can access all fields of all objects of that type, even if they are private. For example, if one created one's own version of an integer, one could write:

```
public class MyInteger {
    private int val;
    public boolean isGreaterThan(MyInteger other) {
        return (this.val>other.val);
    }
}
```

The *this* refers to the current object; in this case it could be omitted:

```
return (val>other.val);
```

Constants are usually uppercase: for example:

```
public final static int MAX_SIZE=100;
```

There are also static methods. Consider, for example, a *Coord* class that stores a point. A static method might be used to compute the distance between two points:

```
public static double distance( Coord p1, Coord p2)
```

It is then called as follows

```
Coord X = new Coord(0.0,0.0);
Coord Y = new Coord(3.0,4.0);
System.out.println( Coord.distance(X,Y) );
```

2.4 Objects as Return Values and Parameters

An object (even an array) can be returned from a method (but only one). An object can also be a parameter to a method.

Note that the method receives a *reference* to an object (this is called *pass by reference*). This means that changes to the object so referenced ARE reflected outside the class. On the other hand, changing a primitive-data-type parameter does NOT affect the value outside the class:

```

void clear(int x,int[] A){
    x=0;
    for(int i=0;i<A.length;i++)
        A[i]=0;
}

```

is called from

```

int y = 42 ;
int[] B = {1,2,3,4,5};
clear(y,B);

```

will cause all the values of B to be zero, but will not change y.

2.5 Objects for Numbers

For every primitive data type there is a corresponding object called its wrapper. So there are `Integer`, `Double` and `Character` classes, etc. These have several methods including:

- constructors which take the corresponding base type;
- accessor methods such as `double doubleVal()` which return the base type; and
- static utilities such as `Integer.parseInt(String str)` that translate a `String` into the base type.

Most of the time Java will automatically convert between a primitive data type and its wrapper.

2.6 Exceptions

Java uses `Exceptions` to signal problems with execution, such as an array index out of range or running out of memory. A `RuntimeException` is called an *unchecked exception* since it does not require compiler check. (Other `Exceptions` are checked and require `throws` statements and/or `try...catch` construction, discussed later.) Example usage:

```

if( index<0 )
    throw new RuntimeException("Bad index");

```

Sample Program

Should be re-implemented to use enums!

```
// Card.java
// a silly class for storing a playing card
// wdg 2006

public class Card {

    public final static String suitChars = "SHDC";
    public final static String denomChars = "A23456789TJQK";
    public final static int numSuits = suitChars.length();
    public final static int numDenoms = denomChars.length();

    private int suit; // stores the suit 0=S, 1=H, 2=D, 3=C
    private int denom; // stores the denom 0=A 1=2 ... 12=K

    //constructor
    public Card(String name){
        if(name.length()!=2)
            throw new RuntimeException("Invalid card: "+name);
        setSuit(name.charAt(0));
        setDenom(name.charAt(1));
    }

    //mutator methods
    private void setSuit(char suitChar){
        suit = suitChars.indexOf(suitChar);
        if(suit<0) // no match
            throw new RuntimeException("Invalid suit: "+suitChar);
    }
    private void setDenom(char denomChar){
        denom = denomChars.indexOf(denomChar);
        if(denom<0) // no match
            throw new RuntimeException("Invalid denom: "+denomChar);
    }

    // accessor methods
    public int getSuit(){
        return suit;
    }
}
```

```

}
public int getDenom(){
    return denom;
}

// overrides toString method of Object
public String toString(){
    return suitChars.charAt(suit)+" "+denomChars.charAt(denom);
}

// an example method to use it
// input: array of Cards
// returns: whether all the Cards have the same suit
public static boolean isFlush( Card[] hand ){
    for( int c=1; c<hand.length; c++ )
        if( hand[c].getSuit() != hand[0].getSuit() )
            return false;
    // failing which
    return true;
}
}

```

```

// CardDriver.java
// simple class to exercise Card class
// wdg 2004

public class CardDriver {

    public static void main(String[] args){

        Card A;
        A = new Card("SA");
        System.out.println("A is: " + A);

        Card[] hand;
        hand = new Card[3];
        hand[0] = new Card("SK");
        hand[1] = new Card("S3");
        hand[2] = A;
        if(Card.isFlush(hand))

```

```
        System.out.println("Yes it's a flush");
    else
        System.err.println("OOOPS");

    hand[0] = new Card("D7");
    if(!Card.isFlush(hand))
        System.out.println("No longer a flush");
    else
        System.err.println("OOOPS");

    A = new Card("C4");
    System.out.println("hand[2] is still: " + hand[2]);

    // next should cause exception
    A = new Card("sillyMe");

}
}
```

Program Development

3.1 OOP Design

Object-oriented programming rests on three principles:

- **Abstraction**: ignore the details
- **Modularization**: break into pieces
- **Information hiding**: separate the implementation and the function

We strive for **responsibility-driven design**: each class should be responsible for its own data. We strive for **loose coupling**: each class is largely independent and communicates with other classes via a small well-defined interface. We strive for **cohesion**: each class performs one and only one task (for readability, reuse).

UML is an extensive language for modeling programs especially those for an object-oriented programming language. It is a system of diagrams designed to capture objects, interaction between objects, and organization of objects, and then some.

3.2 Literate Programming

Good programming means extensive comments and documentation. At the very least:

explain the function of each instance variable, and for each method explain its purpose, parameters, returns, where applicable.

You should also strive for a consistent layout and for expressive variable names.

JavaDoc is a program that will automatically extract/generate an HTML help-page from code that is properly commented. In particular, it produces a help file that, for a class, lists the methods, constructors and public fields, and for each method explains what it does together with pre-conditions, post-conditions, the meaning of the parameters, exceptions that may be thrown and other things.

3.3 Testing

One needs to test extensively. Look at the **boundary** values: make sure it handles the smallest or largest value the program must work for, and suitably rejects the value just out of range. Add **watches** or **debug statements** so that you know what is happening at all times. Especially look at the empty case, or the 0 input.

More Java

4.1 ToString

The class `Object`—which every other class extends—has several standard methods. One is the `toString` method: this is automatically called to convert an object to a string.

You should *override* this; that is, replace with code suitable for your class (since the standard method prints out just the type and address in memory). A class `StringBuilder` is available for use when building up a string. For example:

```
public String toString() {
    StringBuilder result = new StringBuilder();
    for(String S : list)
        result.append( S+" " );
    return result.toString();
}
```

4.2 Assignment and Equality Testing

If `A` and `B` are primitive data types, then `A=B` sets `A` to the value of `B`: changing `B` will have no impact on `A`. If `A` and `B` are objects, then `A=B` sets `A` to refer to the same object that `B` references: until either `A` or `B` is re-assigned, they will point to the same object. If you want to make a separate copy, you need to use a suitable version of the `clone` method, discussed later.

The test `A==B`, for objects, only compares whether they reference the SAME object in memory. It is good practice to override the `equals()` method if one will need to test two objects of one's class for equality.

For example, producing an `equals` method for the `Coord` class mentioned earlier:

```
public boolean equals(Object other) {
    if(other instanceof Coord){
        Coord otherPoint = (Coord) other;
        return (this.x==otherPoint.x && this.y==otherPoint.y);
    }
    else return false;
}
```

This uses a cast, which we now discuss.

4.3 Casting

Recall that the *type of a variable* specifies what it can reference: it can reference any object that is of that type or a subtype of that type. It is an informational message to the compiler. On the other hand, an actual *object has a fixed type*. The type of the object that a variable references is sometimes called the *run-time type* of that variable.

A *cast* tells the compiler that it may assume that the object has the type that the cast says. If at run-time the object doesn't have the claimed type, you will get a `ClassCastException`. This occurs in the above code where we say:

```
Coord otherPoint = (Coord) other;
```

Casting is also used for primitive data types, but this is more like a conversion. For example:

```
System.out.print( (char)('A'+2) );
```

prints out "C"; without the `char` cast it prints out 67 (the code for C).

4.4 More on Exceptions

An exception is explicitly thrown using a `throw` statement. All exceptions that are thrown must be eventually caught. A method might not handle an exception but instead propagate it for another method to handle.

A `try` clause is used to delimit a block of code in which a method call or operation might cause an exception. If an exception occurs within a `try` block, then Java aborts the `try` block, executes the corresponding `catch` block and then continues with the statements that follow the `catch` block. If there is no exception, the `catch` block is ignored.

You can define your own exception classes:

```
public class SillyException extends RuntimeException {
    SillyException(String w) {
        super(w);
    }
}
```

4.5 Command-line Arguments

It is common to get the user to specify data on the command line. This is available to the programmer via the `args` array that is the parameter to the `main` method:

```

public static void main( String[] args ) {
    if( args.length!=1 )
        throw new RuntimeException("Usage: java MyProg limit");
    process( Integer.parseInt(args[0]) );
}

```

4.6 Packages and Scanner

Java has an extensive library for doing many tasks. The library is organized as a collection of packages. To use methods/classes from a package you need the `import` command: for example, the following provides access to all classes in `java.util`

```
import java.util.*;
```

The `Scanner` is a simplified reader class which accesses the input or a file, and does limited conversion for the user (almost without `Exceptions`):

```

import java.util.Scanner;
...
Scanner inn = new Scanner(System.in);
System.out.print("Enter grade> ");
int grade = inn.nextInt();

```

4.7 File Input

To read data from a file in Java is relatively complex as compared to printing to the standard output. The package `java.io` has many classes, and the input and file methods tend to throw checked exceptions, so that the user is forced to deal with exception handling. This is somewhat simplified by the `Scanner` class.

```

import java.io.*;
import java.util.*;
....
Scanner in = null;
try {
    in = new Scanner( new FileReader(args[0]) );
} catch(FileNotFoundException e) {
    System.err.println("Could not open file"+args[0]);
    return;
}

```

Yet More Java

We consider some important Java features: interfaces, inheritance and generics.

5.1 Interfaces

An **interface** specifies the methods which a class should contain. A class can then **implement** an interface (actually it can implement more than one). In doing so, it must implement every method in the interface (it can have more). An interface cannot specify the behavior of the constructor. Note that an **interface** is a valid type for a variable. Example: the interface

```
interface Number {  
    void increment();  
    void add(Number other);  
    ETC  
}
```

the implementation:

```
class MyInteger implements Number {  
    private int x;  
    public void increment(){x++;}  
    ETC  
}
```

the calling program (but then one can only execute **Number**'s methods on **ticket**).

```
Number ticket = new MyInteger();  
ticket.increment();
```

There are also **abstract** classes where some of the methods are implemented and some are not.

5.2 Inheritance

In Java you can make one class an *extension* of another. This is called *inheritance*. The classes form an *is-a* hierarchy. The advantage of inheritance is that it avoids code duplication, promotes code reuse, and improves maintenance and extendibility. In fact, every class automatically extends **Object**.

To create a class **Rectangle** which extends a class **Shape**, one starts with

```
class Rectangle extends Shape {
```

The class `Rectangle` then automatically has all the methods of class `Shape`, and you can add some of your own. The default constructor for `Rectangle` is automatically called at the start of the constructor for `Shape`.

The methods and the instance variables of the superclass can be accessed (if not declared as `private`). This can be done with the prefix `super`. The methods and instance variables of a class may be declared as `protected` to allow direct access only to extensions.

Overriding is providing a method with the same signature as the superclass but different body. This method takes precedence on the subclass object.

Substitution/subtyping: A variable may reference objects of its declared type or any subtype of its declared type; subtype objects may be used whenever supertype objects are expected. There are times an objects may need to be **cast** back to original type. Note that the actual type of an object is forever fixed.

Suppose class `Rectangle` extends class `Shape`. Then we could do the following assignments:

```
Shape X;  
Rectangle Y;  
Y = new Rectangle();  
X = Y; // okay  
X = new Rectangle(); // okay  
Y = (Rectangle)X; // cast needed
```

5.3 Using Generics

Java collections used to handle all elements as `Objects`. This avoided writing a new version of the collection for each type of element (`Integer`, `String`, etc.). However, since we can add anything to a collection, when we extracted things we had to promise the compiler, by using a cast, that we knew what we were doing .

Generics let us tell the compiler what kind of thing we want to store in a particular instance of a collection. It can then check that we are adding the right thing (and give us a compile-time warning if we are not). If we want an `ArrayList` that stores `Strings`, we say

```
ArrayList<String> L = new ArrayList<String>();
```

Example Code

```
// NewCard.java  
// extending the silly class for storing a playing card
```

```
// which had private instance variables
// wdg 2005

public class NewCard extends Card {

    //calls constructor of super class
    NewCard(String name){
        super(name);
    }

    // overrides equals method of Object
    public boolean equals(Object other){
        if(!(other instanceof Card))
            return false;
        else {
            Card otherCard = (Card)other;
            boolean sameSuit = this.getSuit()==otherCard.getSuit();
            boolean sameDenom = this.getDenom()==otherCard.getDenom();
            return sameSuit && sameDenom;
        }
    }
}
```

Collections

6.1 Basic Collections

There are three basic collections.

1. The basic collection is often called a *bag*. It stores objects with no ordering of the objects and no restrictions on them.
2. Another unstructured collection is a *set* where repeated objects are not permitted: it holds at most one copy of each item. (A set is often from a predefined universe.)
3. A collection where there is an ordering is often called a *list*. Specific examples include an *array*, a *vector* and a *sequence*. These have the same idea, but vary as to the methods they provide and their efficiency.

6.2 The Bag ADT

An ADT or *abstract data type* defines a way of storing data: it specifies only how the ADT can be used and says nothing about the implementation of the structure. (An ADT is more abstract than a Java specification.)

The Bag ADT has:

- accessors methods such as `size`, `countOccurrence`, possibly an iterator (which steps through all the elements)
- modifier methods such as `add`, `remove`, and `addAll`
- also a `union` method which combines two bags to produce a third.

6.3 The Array Implementation

A common implementation of a collection is a *partially filled array*. This is often expanded every time it needs to be, but rarely shrunk. It has a pointer/counter which keeps track of where the real data ends.

0	1	2	3	4	5	6
Al	Bart	Carl	Don	NULL	NULL	NULL

size=4

6.4 The Java Collection

The `java.util` package provides some ready-made collections. The root interface is `Collection`, and `Bags` should implement this directly. Two specialization interfaces are: `List`, which specifies a list/sequence, and `Set`, which prohibits duplicates.

These have so-called “optional” operations. That is, the user has to provide code for the method, but it might just throw an `UnsupportedOperationException`.

6.5 Issues for Implementing Data Structures

There are many decisions to make when implementing an ADT. Often more than one way will work.

1. *General or specific data types:* Suppose you want to store `ints`. Well, you can choose to write a specific structure in terms of `ints`; or you can choose to implement a structure that stores `Objects` and then wrap this with a structure that converts `ints` to `Integers`.
2. *Exception handling:* The methods should check for anomalous situations. For example, consider adding a value in an invalid position. You might choose to throw an `Exception` (even create your own `Exception` class); or you might simply not perform the addition, either printing an error message or returning a fail value.
3. *Overflowing Java:* You can run out of memory or you can add too many things that can be addressed by an `int`. You might choose to ignore the running out of memory, since that would trigger an `Exception` anyway, or handle it. With the index going past 2^{31} , probably should deal with it.

6.6 Wrappers

A *wrapper class* is a class which simply contains another class but makes for easier/different user access. For example, `Integer` is the wrapper class for `int`: it is an object, whereas `int` is a primitive data type. Wrappers shield the user from details or from dealing with a more general situation.

Suppose we have a generic collection `MyBag` (that stores objects), but we want a collection that stores only nonnegative integers. The class `MyPosIntBag` has as its only instance variable something of type `MyBag`. Any method designed to add something to the collection, simply wraps the `int` and adds it to the inner collection. Any method designed to retrieve something from the collection, simply retrieves the object and unwraps it. Often the wrapper has the same methods as the underlying object.

6.7 Sample Program

```
// MyPosIntBag - wdg 2006
// stores nonnegative integers
// just to show the use of wrappers

public class MyPosIntBag {

    MyCollection<Integer> C;

    // constructor
    MyPosIntBag() {
        C = new MyCollection<Integer>();
    }

    // a typical add method
    void add(int elem) {
        if( elem<0 )
            throw new IllegalArgumentException();
        Integer wrapped = new Integer(elem);
        C.add(wrapped); // could use autobox
    }

    // a typical retrieve method
    int retrieveOne() {
        Integer retrieved = C.retrieveOne();
        return retrieved.intValue(); // could use autounbox
    }

    public static void main(String[] args) {
        MyPosIntBag B = new MyPosIntBag();
        for(int i=0; i<10; i++)
            B.add(i);
        for(int i=0; i<10; i++)
            System.out.println( B.retrieveOne() );
    }
}
```

```
// MyCollection - wdg 2006 - just for show
// stores a collection using partially filled array

public class MyCollection<E> {
    E[] arr;
    int size;    // real data is in positions 0 to size-1

    // constructor
    MyCollection() {
        arr = (E[])new Object[10]; // causes compiler complaint
    }

    // a typical add method
    void add(E item) {
        arr[size] = item;
        size++;
    }

    // a typical retrieve method
    E retrieveOne() {
        size--;
        return arr[size];
    }
}
```

Order Notation

The **order** of a task measures how the time required for a task grows as the input grows. In this notation, the variable n always stands for the size of the input. We write that a task takes time $O(f(n))$ if it is at most proportional to $f(n)$. This is said “O of $f(n)$ ” or “order $f(n)$ ”.

The most common case is that the time taken is proportional to the input. This is written $O(n)$, also called **linear time**. For example, printing out the contents of an array takes time $O(n)$.

The second most common case is that the time taken does not depend at all on the size of the input. This is called **constant time** and is written $O(1)$. For example, accessing the first entry in the array (or the last), takes time $O(1)$.

Sometimes the length of the task might vary depending on the actual data etc. The big-O notation only considers the **worst case**. For example, searching through an array for a particular value is considered $O(n)$. If you’re lucky there will be a match on the first element; but if the value is not there, you will have to look at the entire array.

A useful rule to note is that if you have a loop, then the overall time is (usually) given by the **product rule**: the (worst-case) number of times the loop-body can be performed *times* the (worst-case) time of the loop-body.

Linked Lists

8.1 Links, References and Pointers

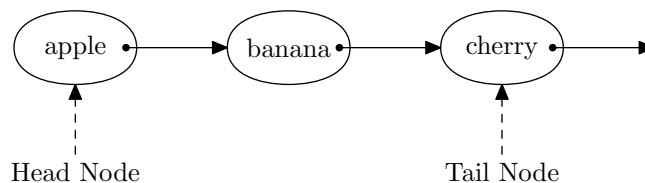
The linked list is not an ADT in its own right; rather it is a way of implementing many data structures. It is designed to *replace* an array.

A linked list is

a sequence of nodes each with a link to the next node.

These links are also called pointers. Both metaphors work. They are links because they go from one node to the next, and because if the link is broken the rest of the list is lost. They are called pointers because this link is (usually) one-directional.

The first node is called the **head node**. The last node is called the **tail node**. The first node has to be pointed to by some external holder; often the tail node is too.



The Java way of having this is:

```
class Node {
  <data>
  Node link;
}
```

(where <data> means any type of data, or multiple types). The class using/creating the linked list then has the declaration:

```
Node head;
```

8.2 Insertion and Traversal

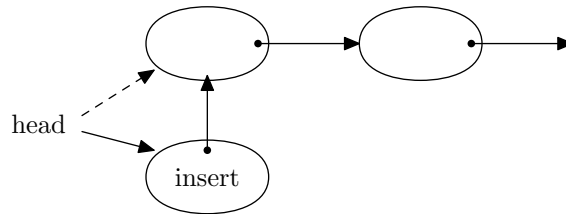
For traversing a list, the idea is to initialize a pointer to the first node (pointed to by head). Then repeatedly advance to the next node. `null` indicates you've reached the end. Such a pointer/reference is called a **cursor**. There is a standard construct for a *for-loop* to traverse the linked list:

```

for( cursor=head; cursor!=null; cursor=cursor.link ){
    <do something with object referenced by cursor>
}

```

For insertion, there are two separate cases to consider: (1) addition at the root, and (2) addition elsewhere. For addition at the root, one creates a new node, changes its pointer to where head currently points, and then gets head to point to it.



In Java:

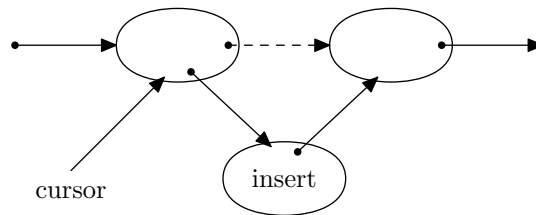
```

head = new Node (<newData>, head);

```

assuming a suitable constructor. This also works if the list is empty.

To insert elsewhere, one needs a reference to the node **BEFORE** where one wants to insert. One creates a new node, changes its pointer to where the node before currently points, and then gets the node before to point to it.



In Java, assuming `cursor` references node before:

```

cursor.link = new Node (<newData>, cursor.link);

```

Traps for Linked Lists:

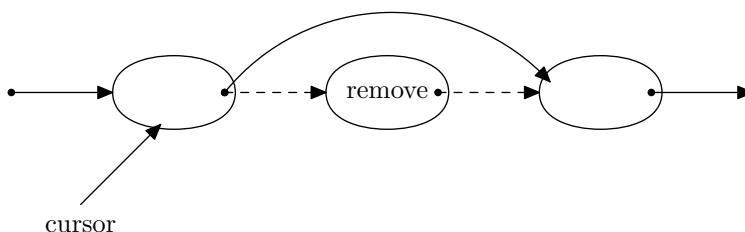
1. You **MUST** think of and test the exceptional cases: The empty list, the beginning of the list, the end of the list.
2. Draw a diagram: you have to get the picture right, and you have to get the order right.

8.3 Removal

The easiest case is removal of the first node. For this, one simply advances the head to point to the next node. (The first node is no longer referenced, and Java garbage collection destroys it.)

```
head = head.link;
```

In general, to remove a node that is elsewhere in the list, one needs a reference to the node **BEFORE** the node one wants to remove. Then one needs only to update the link of the node before to skip that node: that is, point to the node after the one wants to delete.



If the node before is referenced by `cursor`, then `cursor.link` refers to the node to be deleted, and `cursor.link.link` refers to the node after. Hence the Java is:

```
cursor.link = cursor.link.link;
```

The problem is to organize `cursor` to be in the right place. In theory, one would like to traverse the list, find the node to be deleted, and then back up one: but that's not possible. Instead one has to look one node ahead. And then beware null pointers.

8.4 And Beyond

Arrays are better at *random access*: they can provide an element, given a position, in constant time. Linked lists are better at additions/removals at the cursor: done in constant time. Resizing arrays can be inefficient (but is “on average” constant time).

Doubly-linked lists have pointer both forward and backward. These are useful if need to traverse the list in both directions, or to add/remove at both ends.

Dummy header/tail nodes are sometimes used. These allow some of the special cases (e.g. empty list) to be treated the same as a typical case. While searching takes a bit more care, both removal and addition are simplified.

8.5 Direct Access in Nodes

Long before objects, languages such as C and Pascal allowed one to group data (called structs or records). However, only with objects can one group data with its methods.

Some languages, such as C++, allow structs/records to co-exist with objects; but in Java everything is an object.

In principle, instance variables should be private. However, some classes are just helpers for other classes. For example, a linked list class uses a node class, and it is the linked list that the user sees. So, we grant the linked list class direct access to the fields of the node class. This is purely for readability: one can write linked lists without direct access to the instance variables.

Exercise

Develop code for making a copy of a list.

Sample Code

```
// StringNode.java
// a primitive version of node for strings
// wdg 2006

public class StringNode {

    String data;        // stores data
    StringNode next;   // references

    // constructors
    StringNode(){}

    StringNode(String dat,StringNode nxt){
        data=dat;
        next=nxt;
    }

    // toString
    public String toString(){
        return data;
    }
}



---



// StringLinkedBag.java
// uses a linked list to implement a primitive Bag
// wdg 2006
```

```

public class StringLinkedBag {
    StringNode head;
    int size;
    //-----
    // no-argument constructor
    StringLinkedBag(){
        head=null;
        size=0;
    }
    //-----
    // add creates new node and adds at beginning of list
    void add(String newElement){
        head = new StringNode( newElement, head);
        size++;
    }
    //-----
    // tries to delete one node with that value
    // @returns true if target found and false otherwise
    boolean remove(String target) {
        // first special case: list is empty
        if(head==null)
            return false;
        // second special case: first item is deleted
        if(head.data.equals(target)) {
            head=head.next;
            size--;
            return true;
        }
        // so the list is nonempty and newElement does not match first
        StringNode cursor=head;
        while(cursor.next!=null && !cursor.next.data.equals(target))
            cursor=cursor.next;
        if(cursor.next==null)
            return false; // not found
        else {
            cursor.next = cursor.next.next;
            size--;
            return true;
        }
    }
}

```

```

    }
}
//-----
// dump prints out list
void dump(){
    if(head==null)
        System.out.println(">>>Empty");
    else {
        for(StringNode cursor=head; cursor!=null; cursor=cursor.next)
            System.out.println(cursor.data);
    }
    System.out.println(size+" =====");
}
}
}

```

Stacks

A *linear* data structure is one which is ordered. There are two special types with restricted access: a stack and a queue.

9.1 Stacks Basics

A *stack* is a data structure of ordered items such that items can be inserted and removed only at one end (called the *top*). It is also called a LIFO structure: last-in, first-out.

The standard (and usually only) modification operations are:

- **push**: add the element to the top of the stack
- **pop**: remove the top element from the stack and return it

If the stack is empty and one tries to remove an element, this is called *underflow*. Another common operation is called **peek**: this returns a reference to the top element on the stack (leaving the stack unchanged).

A simple stack algorithm could be used to *reverse* a word: push all the characters on the stack, then pop from the stack until it's empty.

$$\text{t h i s} \rightarrow \begin{array}{|c|} \hline \text{s} \\ \hline \text{i} \\ \hline \text{h} \\ \hline \text{t} \\ \hline \end{array} \rightarrow \text{s i h t}$$

9.2 Implementation

A stack is commonly implemented using either an array or a linked list. In the latter case, the head points to the top of the stack: so addition/removal (push/pop) occurs at the head of the linked list.

9.3 Sample Code: ArrayStack

```
// ArrayStack.java
// minimal array implementation of Stack
// no test for overflow
// wdg 2006

public class ArrayStack<E> {

    private E[] arr;    // stores data
    private int size;  // number of elements in stack
    // note that valid data always in 0..size-1

    // constructor
    // arbitrarily sets upper limit
    public ArrayStack(){
        size = 0;
        arr = (E[])new Object[100];
    }

    // pushes object onto stack
    // @post: reference to item placed at right-end of array
    public void push(E item){
        arr[size++] = item;
    }

    // returns whether stack is empty or not
    public boolean isEmpty(){
        return size==0;
    }

    // pops top element from stack
    // @returns: previous top element
    // @post: the top element is removed
    // @throws: EmptyStackException if stack is empty
    public E pop(){
        if(isEmpty())
            throw new java.util.EmptyStackException();
        return arr[--size];
    }
}
```

```

// @returns reference to data on top of stack
//      or null if stack is empty
public E peek(){
    if(isEmpty())
        return null;
    else
        return arr[size-1];
}

// @returns String representation
public String toString(){
    if(isEmpty())
        return "-";
    StringBuffer temp = new StringBuffer();
    for(int x=0; x<size ;x++)
        temp.append( arr[x]+" " );
    return temp.toString();
}
}

```


10.2 Evaluating Arithmetic Expressions

Consider the problem of evaluating the expression: $((3+8)-5)*(8/4)$. We assume for this that the brackets are compulsory: for each operation there is a surrounding bracket. If we do the evaluation by hand, we could:

repeatedly evaluate the first closing bracket and substitute

$$(((3+8)-5)*(8/4)) \rightarrow ((11-5)*(8/4)) \rightarrow (6*(8/4)) \rightarrow (6*2) \rightarrow 12$$

With two stacks, we can evaluate each subexpression when we reach the closing bracket:

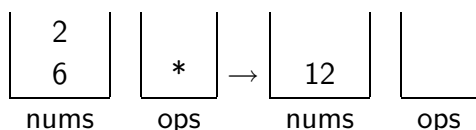
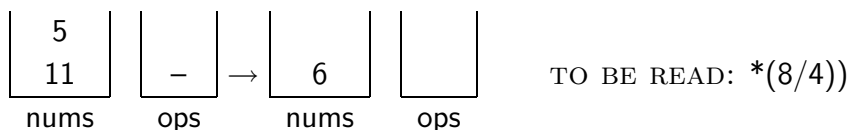
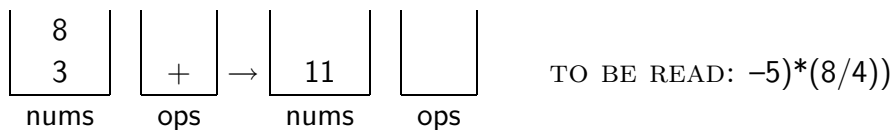
Algorithm (assuming brackets are correct!) is as follows:

Scan the string from left to right and for each char:

1. *If a left bracket, do nothing*
2. *If a number, push onto numberStack*
3. *If an operator, push onto operatorStack*
4. *If a right bracket, do an evaluation:*
 - a.) *pop from the operatorStack*
 - b.) *pop two numbers from the numberStack*
 - c.) *perform the operation on these numbers (in the right order)*
 - d.) *push the result back on the numberStack*

At end of string, the single value on the numberStack is the answer.

The above example $((3+8)-5)*(8/4)$: at the right brackets



Queues

A queue is a *linear* data structure that allows objects to be added only to the rear of the queue and removed only from the front of the queue. Queues are *FIFO* structures: First-in First-out. They are used in operating systems to schedule access to resources such as a printer.

11.1 Queue Methods

The two standard modification methods are:

- void enqueue(Object ob): insert the object to the *rear* of the queue
- Object dequeue(): delete and return the object at the *front* of the queue (sometimes called the first item).

A simple task with a queue is *echoing* the input (in the order it came): repeatedly insert into the queue, and then repeatedly dequeue.

11.2 Queue Implementation as Array

The natural approach to implementing a queue is, of course, an array. This suffers from the problem that as items are enqueued and dequeued, we reach the end of the array but are not using much of the start of the array.

The solution is to allow wrap-around: after filling the array, you start filling it from the front again (assuming these positions have been vacated). Of course, if there really are too many items in the queue, then this approach will also fail. This is sometimes called a *circular array*.

You maintain two markers for the two ends of the queue. The simplest is to maintain instance variables:

- double[] data stores the data
- int size records the number of elements currently in the queue, and int capacity the length of the array
- int front and int rear are such that: if $\text{rear} \leq \text{front}$, then the queue is in positions `data[front] ... data[rear]`; otherwise it is in `data[front] ... data[capacity-1] data[0] ... data[rear]`

For example, the enqueue method is:

```
void enqueue(double elem) {
    if (size == capacity)
        throw new QueueOverflowException() ;
    rear = (rear+1) % capacity ;
    data[rear] = elem ;
    size++;
}
```

As an exercise, provide the dequeue method.

11.3 Queue Implementation as Linked List

A conceptually simpler implementation is a linked list. Since we need to add at one end and remove at the other, we maintain *two* pointers: one to the front and one to the rear. The front will correspond to the head in a normal linked list (doing it the other way round doesn't work: why?).

Queue Applications

12.1 Simulation

There are two very standard uses of queues in programming. The first is in implementing certain searching algorithms. The second is in doing a simulation of a scenario that changes over time. So we examine the CarWash simulation (taken from Main).

The idea: we want to simulate a CarWash to gain some statistics on how service times etc. are affected by changes in customer numbers, etc. In particular, there is a single Washer and a single Line to wait in. We are interested in *how long on average it takes to serve a customer*.

We assume the customers arrive at random intervals but at a known rate. We assume the washer takes a fixed time.

So we create an artificial queue of customers. We don't care about all the details of these simulated customers: just their arrival time is enough:

```
for currentTime running from 0 up to end of simulation:  
1. toss coin to see if new customer arrives at currentTime;  
   if so, enqueue customer  
2. if washer timer expired, then set washer to idle  
3. if washer idle and queue nonempty, then  
   dequeue next customer  
   set washer to busy, and set timer  
   update statistics
```

It is important to note a key approach to such simulations:

wherever possible, you look ahead

Thus when we “move” the Customer to the Washer, we immediately calculate what time the Washer will finish, and then update the statistics. In this case, it actually allows us to discard the Customer: the only pertinent information is that the Washer is busy.

Recursion

Often in solving a problem one breaks up the problem into subtasks. *Recursion* can be used if one of the subtasks is a *simpler version* of the original problem.

13.1 An Example

Suppose we are trying to sort a list of numbers. We could first determine the minimum element; and what remains to be done is to sort the remaining numbers. So the Java code might look something like this:

```
void sort(Collection C){
    min = C.getMinimum();
    System.out.print(min);
    C.remove(min);
    sort(C);
}
```

Every recursive method needs a *stopping case*: otherwise we have an infinite loop or an error. In this case, we have a problem when *C* is empty. So one always checks to see if the problem is simple enough to solve directly.

```
void sort(Collection C){
    if(C.isEmpty())return;
    ... // as before
```

Example. Printing out a decimal number. The idea is to extract one digit and then recursively print out the rest. It's hard to get the most significant digit, but one can obtain the least significant digit (the “ones” column): use `num % 10`. And then `num/10` is the “rest” of the number.

```
void print( int n ) {
    if( n>0 ) {
        print ( n/10 );
        System.out.print( n%10 );
    }
}
```

13.2 Tracing Code

It is important to be able to *trace* recursive calls: step through what is happening in the execution. Consider the following code:

```
void g( int n ) {
    if( n==0 ) return;
    g(n-1);
    System.out.println(n);
}
```

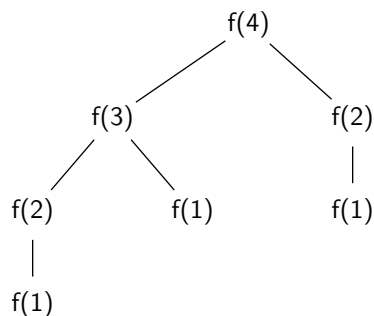
It is not hard to see that, for example, `g(3)` prints out the numbers from 3 down to 1. But, you have to be a bit more careful. The recursive call occurs before the value 3 is printed out. This means that the output is from smallest to biggest.

1
2
3

Here is some more code to trace:

```
void f( int n ) {
    System.out.print(n);
    if(n>1)
        f(n-1);
    if(n>2)
        f(n-2);
}
```

If you call the method `f(4)`, it prints out 4 and then calls `f(3)` and `f(2)` in succession. The call to `f(3)` calls both `f(2)` and `f(1)`, and so on. One can draw a *recursion tree*: this looks like a family tree except that the children are the recursive calls.



Then one can work out that `f(1)` prints 1, that `f(2)` prints 21 and `f(3)` prints 3211. What does `f(4)` print out?

Exercise

Give recursive Java code so that `brackets(5)` prints out `(((((()))))`.

13.3 Methods that Return Values

Some recursive methods return values. For example, the sequence of Fibonacci numbers 1, 1, 2, 3, 5, 8, 13, 21, 34, ... is defined as follows: the first two numbers are 1 and 1, and then each next number is the sum of the two previous numbers. There is obvious recursive code for the Fibonacci numbers:

```
int fib(int n){
    if( n<2 ) return 1;
    else return fib(n-1) + fib(n-2);
}
```

WARNING: Recursion is often easy to write (once you get used to it!). But occasionally it is very inefficient. For example, the code for `fib` above is terrible. (Try to calculate `fib(30)`.)

Recursion Applications

14.1 The Steps problem

THE STEPS PROBLEM. One can use recursion to solve the Steps problem. In this problem, one can take steps of specified lengths and has to travel a certain distance exactly (for example, a cashier making change for a specific amount using coins of various denominations).

The Java/pseudocode is as follows

```
boolean canStep(int required) {
    if( required==0 )
        return true;
    for( each allowed length )
        if( length<=required && canStep(required-length) )
            return true;
    //failing which
    return false;
}
```

The recursive boolean method takes as parameter the remaining distance required, and returns whether this is possible or not. If the remaining distance is 0, it returns true. Else it considers each possible first step in turn. If it is possible to get home after making that first step, it returns true; failing which it returns false. One can adapt this to actually count the minimum number of steps needed. See attached code.

Example Program

```
// StepsByRecursion.java
// a recursive program to solve Steps problem
// wdg 2005

public class StepsByRecursion {

    public static final int IMPOSSIBLE=-1;

    // @pre: required>=0
    // @returns: minimum number of steps; a value of -1 means impossible
    static int minSteps(int required, int[] allowed) {
        if(required==0)
            return 0;
        else {
            int min = required+1; //guaranteed to be too large
            // consider all valid first steps
            for(int poss=0; poss<allowed.length; poss++)
                if( allowed[poss]<=required ) {
                    // call recursively to determine best way to do remainder
                    int temp = minSteps( required-allowed[poss], allowed);
                    if(temp<min && temp!=IMPOSSIBLE)
                        min=temp;
                }
            // return
            if(min==required+1)
                return IMPOSSIBLE;
            else
                return 1+min;
        }//else
    }

    public static void main(String[] args){
        int[] coins = {1,5,10};
        System.out.println(minSteps(17,coins)); // dime, nickel and 2 pennies
        int[] silly = {2,4,6};
        System.out.println(minSteps(17,silly)); // odd is impossible
    }
}
```

14.2 The Queens problem

One can also use recursion to solve the *Queens* problem. The old puzzle is to place 8 queens on a 8×8 chess/checkers board such that no pair of queens attack each other. That is, no pair of queens are in the same row, the same column, or the same diagonal.

The solution uses search and *backtracking*. We know we will have exactly one queen per row. The recursive method tries to place the queens one row at a time. The main method calls `place(0)`. Here is the Java/pseudocode:

```
void place(int row) {
    if(row==8)celebrateAndStop();
    else {
        for( queen[row] = all possible vals ) {
            check if new queen legal;
            record columns and diagonals it attacks;
            // recurse
            place(row+1);
            // if reach here, have failed and need to backtrack
            erase columns and diagonals it attacks;
        }
    }
}
```

One can also use recursion to explore a maze or to draw a snowflake fractal.

Searching and Sorting

15.1 Linear Search and Binary Search

The simplest search is called *linear search*. In this search, the data is stored in an array. The search starts at one end of the array and proceeds to the other. It stops if a match is found. If there is no match, the entire array is accessed. So the running time is $O(n)$.

A classic search is *binary search*. This can be used if the array is *sorted*: say the array is sorted in increasing order. Binary search compares the target value with the middle entry of the array. If there is a match, we are done (not likely!). If the target is bigger than the middle entry, then one can discard the first half of the array. Otherwise, the target is smaller than the middle entry, and so one can discard the second half of the array. Since the array size shrinks by a factor of 2 each time, the running time is $O(\log_2 n)$. The code is provided at the end of the chapter.

For example, searching for 32 in the following array:

7	22	29	32	42	52	59	66	69	76
0				4					9

Start: first=0; last=9;

middle=4;

32 is less than A[4]=42;

Calls: first=0; last=3;

middle=2;

32 is more than A[2]=29;

Calls: first=3; last=3;

middle=3;

a match!

3 is returned.

Example Code

```
public class BinarySearch {

    static final boolean DEBUG=true;
    public static final int UNFOUND=-1;

    // look for TARGET in window positions FIRST, FIRST+1, ..., LAST
    public static int search(int target, int[] A, int first, int last) {
        if(DEBUG)System.out.println("Searching ("+first+": "+last+"");

        if(last<first)
            return UNFOUND;
        else {
            int middle = (first + last)/2;
            if(target == A[middle])
                return middle;
            else if(target < A[middle])
                return search(target, A, first, middle-1);
            else
                return search(target, A, middle+1, last);
        }
    }

    public static int search(int target, int[] A) {
        return search(target, A, 0, A.length-1);
    }

    public static void main(String[] args) {
        int[] test = {7,22,29,32,42,52,59,66,69,76};
        System.out.println( search(32,test) );
    }
}
```

Sorting

16.1 Quadratic Sorting

Say we are trying to sort a list of numbers in increasing order. There are two obvious sorting methods. Both take $O(n^2)$ time.

In *selection sort*, the entire list is scanned to find an extreme element, say the largest element. This is then placed at the end of the list. Then we *recurse*.

We can implement Selection Sort without any (significant) extra storage (it is called an “in situ” sort). The idea is to scan the array for the largest entry, keeping track of its index. Then swap the largest entry with the entry in the last position. And repeat, ignoring the last position of the array. Here is the start array, and the array after one pass.

INITIAL ARRAY:

11	3	14	22	37	45	5	16
----	---	----	----	----	----	---	----

ARRAY AFTER ONE PASS:

11	3	14	22	37	16	5	45
----	---	----	----	----	----	---	----

In *insertion sort*, we build up the sorted list one entry at a time. Each time, we add a new entry. We do binary search to determine where it belongs (quick!). But then we have to move everything down in the array to make space for it (slow). For example, here is the array after six numbers have been added; when the 5 is added, almost all the numbers shift down.

ARRAY AFTER SIX NUMBERS ADDED

3	11	14	22	37	45	0	0
---	----	----	----	----	----	---	---

ARRAY AFTER SEVEN NUMBERS ADDED

3	5	11	14	22	37	45	0
---	---	----	----	----	----	----	---

16.2 Merge Sort

Other famous sorts include Shell Sort, Quick Sort, Heap Sort and Merge Sort. We finish with a discussion of the latter. This is an example of a recursive sort.

```
MergeSort (list A) {  
    Arbitrarily split the list into two halves.  
    Use MergeSort to sort each half.  
    Merge the two sorted halves.  
}
```

One divides the list into two pieces just by slicing in the middle. Then one sorts each piece using recursion. Finally one is left with two sorted lists. And one must now combine them. The process of combining is known as *merging*.

How to merge? Well, think of the two sorted lists as stacks of test papers sitting on the desk with the worst grade on top of each pile. The worst grade in the entire list is either the worst grade in the first pile or the worst grade in the second pile. So compare the two top elements and set the worst aside. The second-worst grade is now found by comparing the top grade on both piles and setting it aside. Etc.

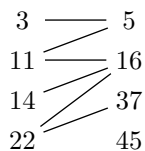
INITIAL ARRAY:

11	3	14	22	37	45	5	16
----	---	----	----	----	----	---	----

AFTER THE TWO RECURSIVE CALLS:

3	11	14	22	5	16	37	45
---	----	----	----	---	----	----	----

MERGING:



It follows that the merge part takes $O(n)$ time. It can be shown (we do not try to prove this) that overall time is $O(n \log_2 n)$. And Merge Sort is provably optimal.

Iterators

A great idea! An *iterator* enables one to step through the collection. Conceptually:

```
forEach object in collection in turn {
    do something with the object
}
```

The `Iterator` interface in Java (in package `java.util`) provides two basic methods:

- `boolean hasNext()`: indicates whether the iterator is finished or not
- `Object next()`: returns the next element in the collection (and adjusts the iterator to the following element); if `hasNext()` is false, then throws an `Exception`.

Further, the constructor should initialize the `Iterator` to the first element of the collection.

Note: You shouldn't modify a collection while an iterator is working, even if the writers of Java provide a `remove()` method.

To print out a list using an `Iterator` for a collection `C`, we might write:

```
Iterator I = C.iterator();
while( I.hasNext() ){
    System.out.println( I.next() );
}
```

A collection `C` that has an iterator also allows one to use the “for each” construct:

```
for(Object ob : C){
    System.out.println( ob );
}
```

For this course, we will implement an iterator with an *external class*. That class will contain a reference to the collection that one is stepping through (passed in by the constructor).

So the code for the Iterator of a Bag might be:

```
import java.util.Iterator;
public class MyBagIterator<E> implements Iterator<E> {
    private Bag<E> B;
    private [[ pointer p ]];
    MyBagIterator(Bag<E> b){
        B=b;
        [[ initialize p ]];
    }
    public boolean hasNext(){
        return [[ some test on p ]] ;
    }
    public E next(){
        if(!hasNext())
            throw new NoSuchElementException();
        E toBeReturned = [[ what p references ]] ;
        [[ advance p ]]
        return toBeReturned;
    }
    public void remove(){
        // not implemented
        throw new UnsupportedOperationException();
    }
}
```

Note that the pointer always points to the next item that will be returned, not the one that was last returned.

More OOP/Java

We discuss some further Java features. These include polymorphism, comparators, and cloning.

18.1 Inheritance Revisited: Polymorphism

Polymorphism (multiple forms) is achieved in Java through over-riding methods in a sub-class. For example, suppose we have a class `Sound` that has a method `play()`. We might have sub-classes `Flute` and `Drum` that extend `Sound` and over-ride this method to provide effects specific to that instrument. Thus objects which are `Sounds` can come in many forms. Java ensures that the appropriate form of the method is used. Hence we can obtain multiple behaviors, but at the same time we can have some general methods which treat all `Sounds` the same.

For example, we could create an array of `Sounds` to represent some music. Then an iterator through the array, calling `play()` at each step, will produce the right sound at the right place. We can also create sub-classes that do not override the method—then the `play()` of `Sound` will be used.

Another example of polymorphism is our writing of a `toString()` method for (most of) our classes.

18.2 Comparable and Comparator

The `Comparable` interface is an in-built interface to enable Java to provide facilities for sorting and searching problems. The idea is that to sort a collection, be it `Strings`, `ints`, or database records, one needs a way to compare two items. Implementing the `Comparable<E>` interface guarantees that there is a method

- `int compareTo(E other)`

which returns one of three values: `-1` if `this` comes before `other`, `+1` if `this` comes after `other`, and `0` if the two objects are to be considered the same. (This method would usually be compatible with the `equals` method: `compareTo==0` if and only if `equals==true`.)

In-built classes such as `String` and `Integer` implement this interface. The `java.util.Array` package has `Sorting` methods which takes arrays whose elements implement `Comparable`.

A more general idea is a `Comparator`. Suppose, for example, one wanted to sort `Strings` in a different way to the standard way. The idea is to define a comparator which specifies how two `Strings` are to be compared (with methods `compare` and `equals`). The comparator is then a parameter to the constructor of the `Collection`.

18.3 Cloneable

This is a kludge. The `Cloneable` interface has NO methods! Its implementation means that the class may be cloned (and should have a suitable `clone` method). It is a *shallow copy*: only the top-level variables are duplicated, not the objects that they reference.