

Chapter 1

Primitive Java

```
 1 // First program
 2 // MW, 9/1/97
 3
 4 public class FirstProgram
 5 {
 6     public static void main( String [ ] args )
 7     {
 8         System.out.println( "Is there anybody out there?" );
 9     }
10 }
```

A simple first program

Primitive Type	What It Stores	Range
`byte`	8-bit integer	–128 to 127
`short`	16-bit integer	–32,768 to 32,767
`int`	32-bit integer	–2,147,483,648 to 2,147,483,647
`long`	64-bit integer	-2^{63} to $2^{63} - 1$
`float`	32-bit floating-point	6 significant digits, (10^{-46}, 10^{38})
`double`	64-bit floating-point	15 significant digits, (10^{-324}, 10^{308})
`char`	Unicode character	
`boolean`	Boolean variable	`false` and `true`

The eight primitve types in Java

```
 1 public class OperatorTest
 2 {
 3     // Program to illustrate basic operators
 4     // The output is as follows:
 5     // 12 8 6
 6     // 6 8 6
 7     // 6 8 14
 8     // 22 8 14
 9     // 24 10 33
10
11     public static void main( String [ ] args )
12     {
13         int a = 12, b = 8, c = 6;
14
15         System.out.println( a + " " + b + " " + c );
16         a = c;
17         System.out.println( a + " " + b + " " + c );
18         c += b;
19         System.out.println( a + " " + b + " " + c );
20         a = b + c;
21         System.out.println( a + " " + b + " " + c );
22         a++;
23         ++b;
24         c = a++ + ++b;
25         System.out.println( a + " " + b + " " + c );
26     }
27 }
```

Program that illustrates operators

X	Y	X && Y	X || Y	!X
false	false	false	false	true
false	true	false	true	true
true	false	false	true	false
true	true	true	true	false

Result of logical operators

```
if( expression )
    statement
next statement

if( expression )
    statement1
else
    statement2
next statement

while( expression )
    statement
next statement

for( initialization; test; update )
    statement
next statement

do
    statement
while( expression );
next statement

while( ... )
{
    if( something )
        break;
}

  outer:
    while( ... )
    {
        while( ... )
            if( disaster )
                break outer; // Go to after outer
    }
    // Control passes here after outer loop is exited

for( int i = 1; i <= 100; i++ )
{
    if( i % 10 == 0 )
        continue;
    System.out.println( i );
}
```

Examples of conditional and looping constructs

```
switch( someCharacter )
{
  case '(':
  case '[':
  case '{':
    // Code to process opening symbols
    break;

  case ')':
  case ']':
  case '}':
    // Code to process closing symbols
    break;

  case '\n':
    // Code to handle newline character
    break;

  default:
    // Code to handle other cases
    break;
}
```

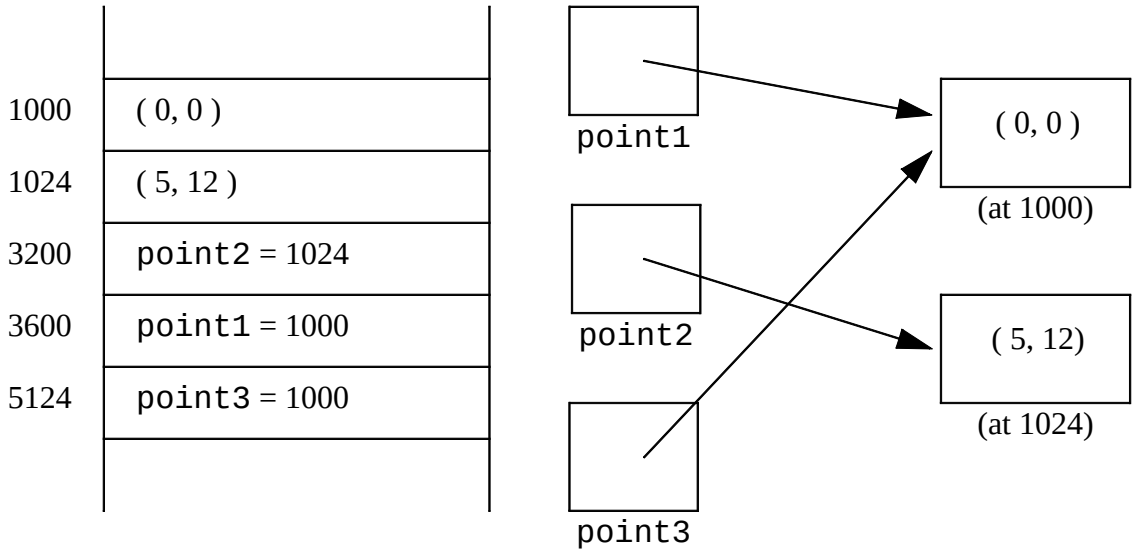
Layout of a `switch` statement

```
 1 public class MinTest
 2 {
 3     public static void main( String [ ] args )
 4     {
 5         int a = 3;
 6         int b = 7;
 7
 8         System.out.println( min( a, b ) );
 9     }
10
11     // Method declaration
12     public static int min( int x, int y )
13     {
14         return x < y ? x : y;
15     }
16 }
```

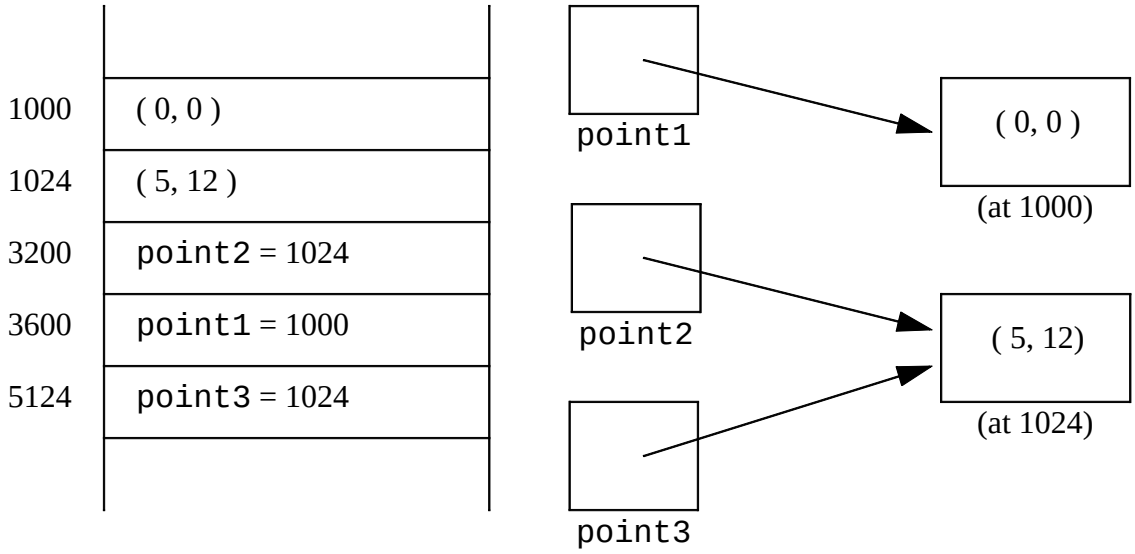
Illustration of method declaration and calls

Chapter 2

References



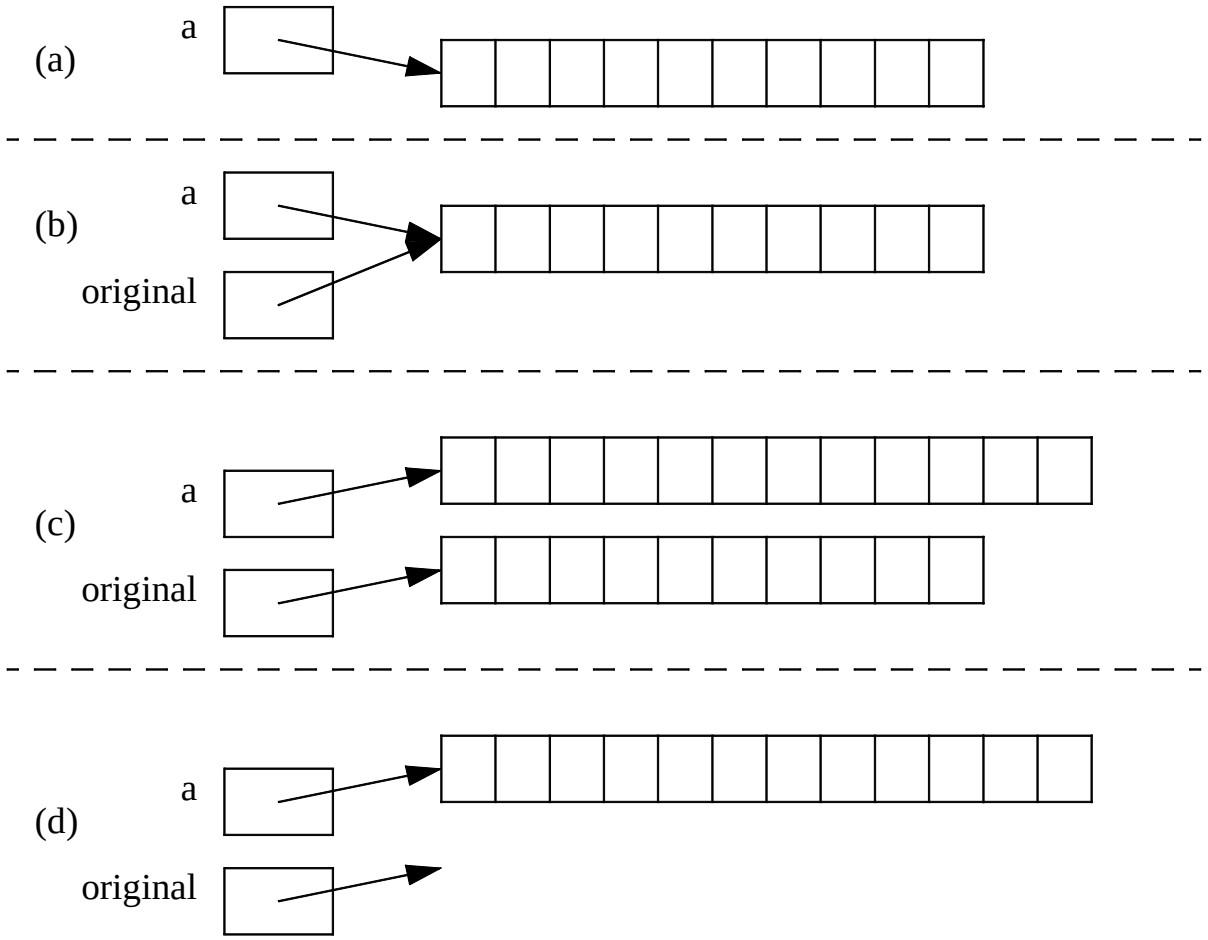
An illustration of a reference: The `Point` object stored at memory location 1000 is referenced by both `point1` and `point3`. The `Point` object stored at memory location 1024 is referenced by `point2`. The memory locations where the variables are stored are arbitrary



The result of `point3=point2`: `point3` now references the same object as `point2`

```
 1 import Supporting.Random;
 2
 3 public class Lottery
 4 {
 5     // Generate lottery numbers (from 1-49)
 6     // Print number of occurrences of each number
 7
 8     public static final int DIFF_NUMBERS     =   49;
 9     public static final int NUMBERS_PER_GAME =    6;
10     public static final int GAMES            = 1000;
11
12     public static void main( String [ ] args )
13     {
14         // Generate the numbers
15         int [ ] numbers = new int [ DIFF_NUMBERS + 1 ];
16         for( int i = 0; i < numbers.length; i++ )
17             numbers[ i ] = 0;
18
19         Random r = new Random( );
20
21         for( int i = 0; i < GAMES; i++ )
22             for( int j = 0; j < NUMBERS_PER_GAME; j++ )
23                 numbers[ r.randomInt( 1, DIFF_NUMBERS ) ]++;
24
25         // Output the summary
26         for( int k = 1; k <= DIFF_NUMBERS; k++ )
27             System.out.println( k + ": " + numbers[ k ] );
28     }
29 }
```

Simple demonstration of arrays



Array expansion: (a) starting point: `a` references 10 integers; (b) after step 1: `original` references the 10 integers; (c) after steps 2 and 3: `a` references 12 integers, the first 10 of which are copied from `original`; (d) after `original` exits scope, the original array is unreferenced and can be reclaimed

Standard Run-time Exception	Meaning
`ArithmeticException`	Overflow or integer division by zero.
`NumberFormatException`	Illegal conversion of `String` to numeric type.
`IndexOutOfBoundsException`	Illegal index into an array or `String`.
`NegativeArraySizeException`	Attempt to create a negative-length array.
`NullPointerException`	Illegal attempt to use a null reference.
`SecurityException`	Run-time security violation.

Common standard run-time exceptions

Standard Checked Exception	Meaning
`java.io.EOFException`	End-of-file before completion of input.
`java.io.FileNotFoundException`	File not found to open.
`java.io.IOException`	Includes most I/O exceptions.
`InterruptedException`	Thrown by the `Thread.sleep` method.

Common standard checked exceptions

```
 1 import java.io.*;
 2
 3 public class DivideByTwo
 4 {
 5     public static void main( String [ ] args )
 6     {
 7
 8         BufferedReader in = new BufferedReader( new
 9                                 InputStreamReader( System.in ) );
10         int x;
11         String oneLine;
12
13         System.out.println( "Enter an integer: " );
14         try
15         {
16             oneLine = in.readLine( );
17             x = Integer.parseInt( oneLine );
18             System.out.println( "Half of x is " + ( x / 2 ) );
19         }
20         catch( Exception e )
21           { System.out.println( e ); }
22     }
23 }
```

Simple program to illustrate exceptions

```
 1 import java.io.*;
 2
 3 public class ThrowDemo
 4 {
 5     public static void processFile( String toFile )
 6                                             throws IOException
 7     {
 8         // Omitted implementation propagates all
 9         // thrown IOException back to the caller
10     }
11
12     public static void main( String [ ] args )
13     {
14         for( int i = 0; i < args.length; i++ )
15         {
16             try
17               { processFile( args[ i ] ); }
18             catch( IOException e )
19               { System.err.println( e ); }
20         }
21     }
22 }
```

Illustration of the `throws` clause

```
 1 import java.io.*;
 2 import java.util.*;
 3
 4 public class MaxTest
 5 {
 6     public static void main( String [ ] args )
 7     {
 8         BufferedReader in = new BufferedReader( new
 9                                 InputStreamReader( System.in ) );
10         String oneLine;
11         StringTokenizer str;
12         int x;
13         int y;
14
15         System.out.println( "Enter 2 ints on one line: " );
16         try
17         {
18             oneLine = in.readLine( );
19             str = new StringTokenizer( oneLine );
20             if( str.countTokens( ) != 2 )
21                 throw new NumberFormatException( );
22             x = Integer.parseInt( str.nextToken( ) );
23             y = Integer.parseInt( str.nextToken( ) );
24             System.out.println( "Max: " + Math.max( x, y ) );
25         }
26         catch( Exception e )
27           { System.err.println( "Error: need two ints" ); }
28     }
29 }
```

Program that demonstrates the string tokenizer

```
 1 public class ListFiles
 2 {
 3     public static void main( String [ ] args )
 4     {
 5         if( args.length == 0 )
 6             System.out.println( "No files specified" );
 7         for( int i = 0; i < args.length; i++ )
 8             listFile( args[ i ] );
 9     }
10
11     public static void listFile( String fileName )
12     {
13         FileReader theFile;
14         BufferedReader fileIn = null;
15         String oneLine;
16
17         System.out.println( "FILE: " + fileName );
18         try
19         {
20             theFile = new FileReader( fileName );
21             fileIn  = new BufferedReader( theFile );
22             while( ( oneLine = fileIn.readLine( ) ) != null )
23                 System.out.println( oneLine );
24         }
25         catch( Exception e )
26           { System.out.println( e ); }
27
28         // Close the stream
29         try
30         {
31             if( fileIn != null )
32                 fileIn.close( );
33         }
34         catch( IOException e ) { }
35     }
36 }
```

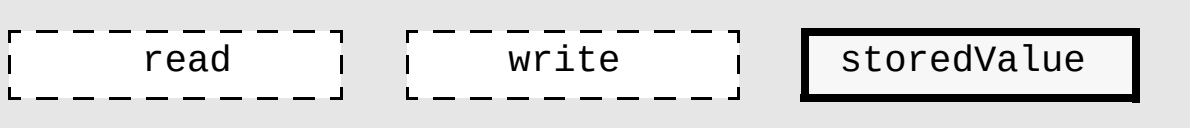
Program to list contents of a file

Chapter 3

Objects and Classes

```
1  // IntCell class
2  //  int read( )         -->  Returns the stored value
3  //  void write( int x ) -->  x is stored
4
5  public class IntCell
6  {
7          // Public methods
8      public int read( )         { return storedValue; }
9      public void write( int x ) { storedValue = x; }
10
11         // Private internal data representation
12     private int storedValue;
13 }
```

A complete declaration of an `IntCell` class



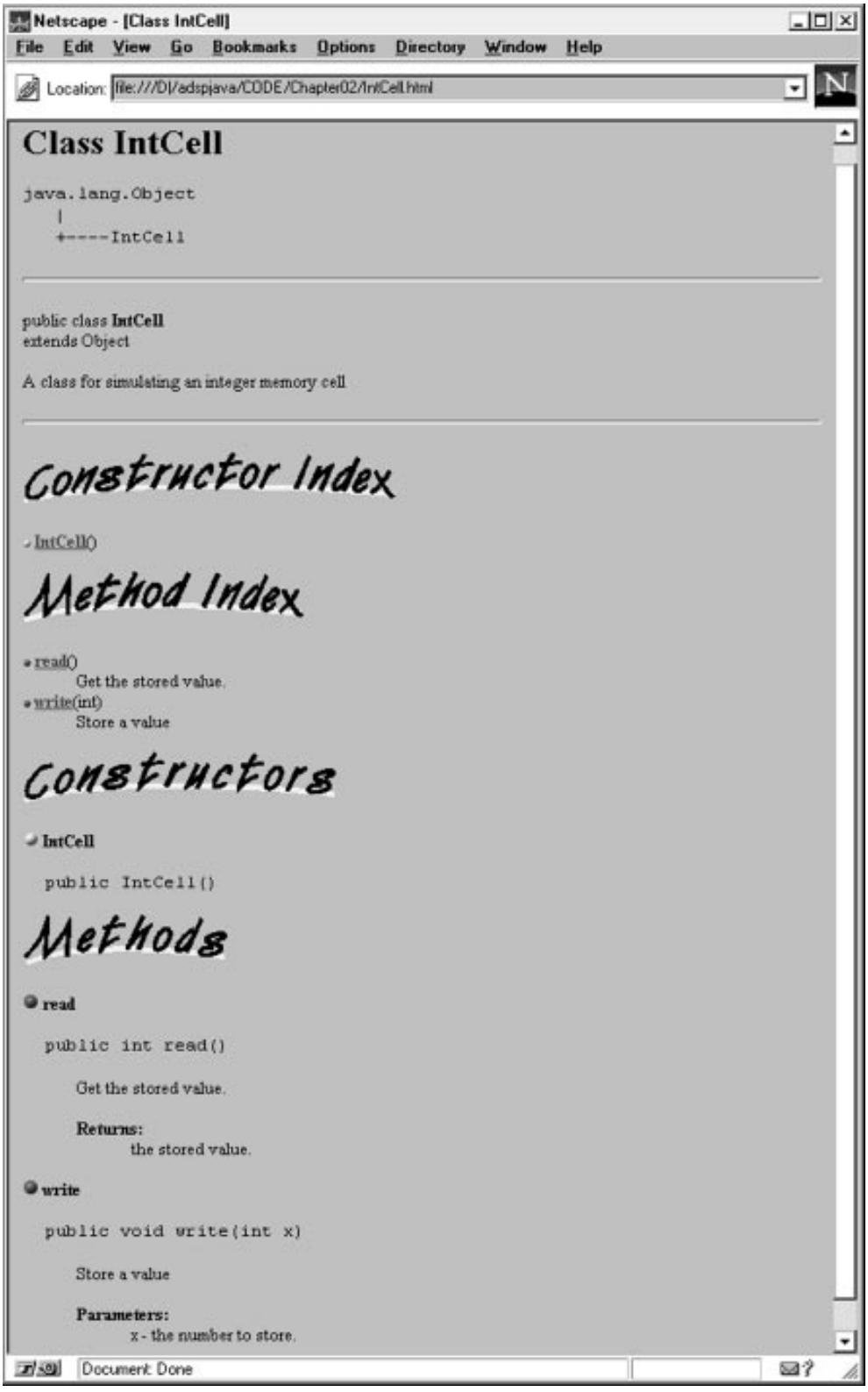
`IntCell` members: `read` and `write` are accessible, but `storedValue` is hidden

```
 1 // Exercise the IntCell class
 2
 3 public class TestIntCell
 4 {
 5     public static void main( String [ ] args )
 6     {
 7         IntCell m = new IntCell( );
 8
 9         m.write( 5 );
10         System.out.println( "Cell contents: " + m.read( ) );
11
12         // The next line would be illegal if uncommented
13         // because storedValue is a private member
14     //  m.storedValue = 0;
15     }
16 }
```

A simple test routine to show how `IntCell` objects are accessed

```
 1 /**
 2  * A class for simulating an integer memory cell
 3  * @author Mark A. Weiss
 4  */
 5
 6 public class IntCell
 7 {
 8     /**
 9      * Get the stored value.
10      * @return the stored value.
11      */
12     public int read( )
13     {
14         return storedValue;
15     }
16
17     /**
18      * Store a value.
19      * @param x the number to store.
20      */
21     public void write( int x )
22     {
23         storedValue = x;
24     }
25
26     private int storedValue;
27 }
```

`IntCell` declaration with *javadoc* comments



javadoc output for `IntCell`

```
 1 // Minimal Date class that illustrates some Java features
 2 // No error checks or javadoc comments
 3
 4 public class Date
 5 {
 6         // Zero-parameter constructor
 7     public Date( )
 8     {
 9         month = 1;
10         day = 1;
11         year = 1998;
12     }
13
14         // Three-parameter constructor
15     public Date( int theMonth, int theDay, int theYear )
16     {
17         month = theMonth;
18         day   = theDay;
19         year  = theYear;
20     }
21
22         // Return true if two equal values
23     public boolean equals( Object rhs )
24     {
25         if( ! ( rhs instanceof Date ) )
26             return false;
27         Date rhDate = ( Date ) rhs;
28         return rhDate.month == month && rhDate.day == day &&
29                rhDate.year == year;
30     }
31
32         // Conversion to String
33     public String toString( )
34     {
35         return month + "/" + day + "/" + year;
36     }
37
38         // Fields
39     private int month;
40     private int day;
41     private int year;
42 }
```

A minimal `Date` class that illustrates constructors and the `equals` and `toString` methods

Package	Use
`DataStructures`	Classes that implement data structures and sorting.
`Exceptions`	Exception classes.
`Supporting`	Various supporting classes and interfaces.

Packages defined in this text

```
 1 package Supporting;
 2
 3 public class Exiting
 4 {
 5     // Suspend current program for a long time
 6     public static void longPause( )
 7     {
 8         try
 9           { Thread.sleep( 1000000000 ); }
10         catch( InterruptedException e ) { }
11     }
12 }
```

A class `Exiting` with a single static method, which is part of the package `Supporting`

```
// Transfer all money from rhs to current account
public void finalTransfer( Account rhs )
{
    dollars += rhs.dollars;
    rhs.dollars = 0;
}

Account account1;
Account account2;
   ...
account2 = account1;
account1.finalTransfer( account2 );
```

Aliasing example

```
// Transfer all money from rhs to current account
public void finalTransfer( Account rhs )
{
    if( this == rhs )    // Alias test
        return;
    dollars += rhs.dollars;
    rhs.dollars = 0;
}
```

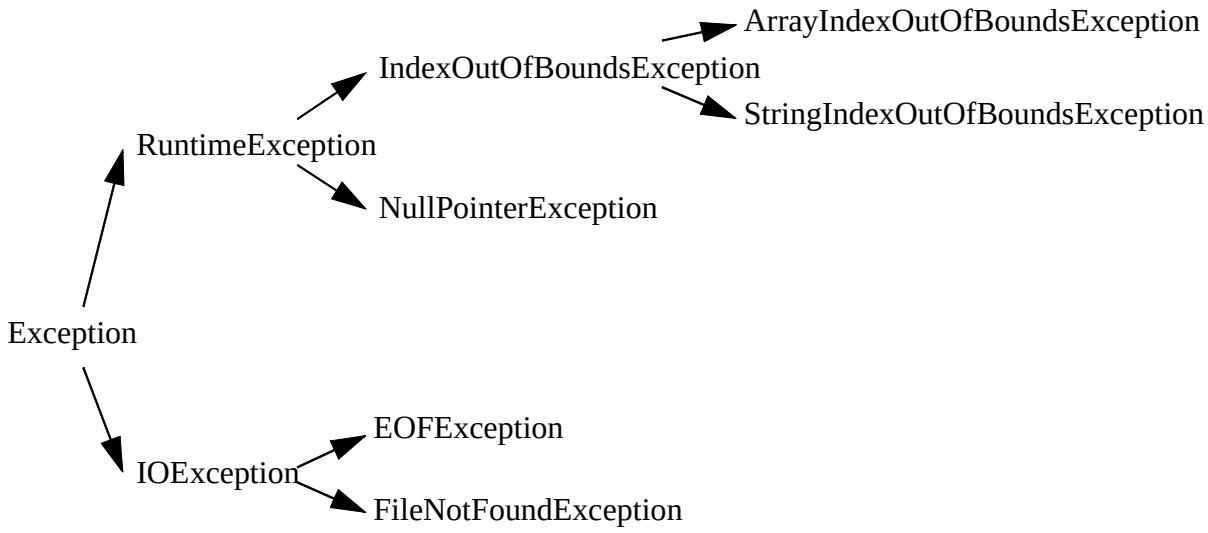
Aliasing fixed

```
 1 public class Squares
 2 {
 3     private static double squareRoots[ ] = new double[ 100 ];
 4
 5     static
 6     {
 7         for( int i = 0; i < squareRoots.length; i++ )
 8             squareRoots[ i ] = Math.sqrt( ( double ) i );
 9     }
10
11     // Rest of class
12 }
```

Example of a static initializer

Chapter 4

Inheritance



Part of the `Exception` hierarchy

```
1  public class Derived extends Base
2  {
3      // Any members that are not listed are inherited unchanged
4      // except for constructor
5
6          // public members
7      // Constructor(s) if default is not acceptable
8      // Base methods whose definitions are to change in Derived
9      // Additional public methods
10
11         // private member
12     // Additional data fields (generally private)
13     // Additional private methods
14 }
```

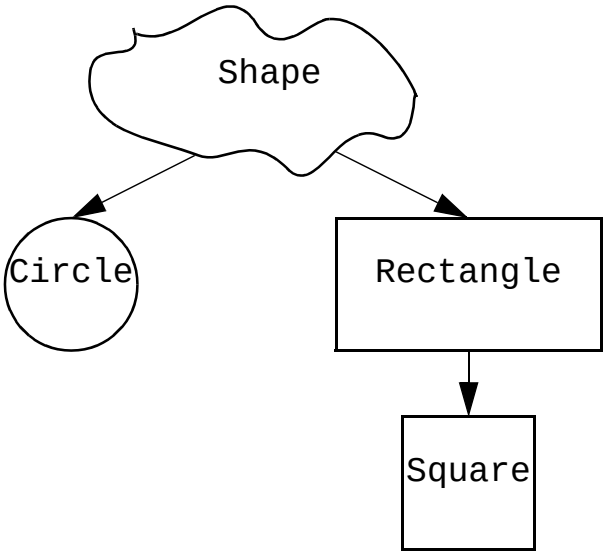
General layout of public inheritance

```
1 package Exceptions;
2
3 public class Underflow extends Exception
4 {
5     public Underflow( String thrower )
6     {
7         super( thrower );
8     }
9 }
```

Constructor for new exception class `Underflow`; uses `super`

```
1 public class Workaholic extends Worker
2 {
3     public doWork( )
4     {
5         super.doWork( );   // Work like a Worker
6         drinkCoffee( );    // Take a break
7         super.doWork( );   // Work like a Worker some more
8     }
9 }
```

Partial overriding



The hierarchy of shapes used in an inheritance example

1. *Final methods*. Overloading is resolved at compile time. We use a final method only when the method is invariant over the inheritance hierarchy (that is, when the method is never redefined).
2. *Abstract methods*. Overloading is resolved at run time. The base class provides no implementation and is abstract. The absence of a default requires either that the derived classes provide an implementation or that the classes themselves be abstract.
3. *Static methods*. Overloading is resolved at compile time because there is no controlling object.
4. *Other methods*. Overloading is resolved at run time. The base class provides a default implementation that may be either overridden by the derived classes or accepted unchanged by the derived classes.

Summary of final, static, abstract, and other methods

1. Provide a new constructor.
2. Examine each method to decide if we are willing to accept its defaults; for each method whose defaults we do not like, we must write a new definition.
3. Write a definition for each abstract method.
4. Write additional methods if appropriate.

Programmer responsibilities for derived class

Array position	0	1	2	3	4	5
Initial State:	8	5	9	2	6	3
After a[0..1] is sorted:	5	8	9	2	6	3
After a[0..2] is sorted:	5	8	9	2	6	3
After a[0..3] is sorted:	2	5	8	9	6	3
After a[0..4] is sorted:	2	5	6	8	9	3
After a[0..5] is sorted:	2	3	5	6	8	9

Basic action of insertion sort (shaded part is sorted)

Array position	0	1	2	3	4	5
Initial State:	8	5				
After a[0..1] is sorted:	5	8	9			
After a[0..2] is sorted:	5	8	9	2		
After a[0..3] is sorted:	2	5	8	9	6	
After a[0..4] is sorted:	2	5	6	8	9	3
After a[0..5] is sorted:	2	3	5	6	8	9

Closer look at action of insertion sort (dark shading indicates sorted area; light shading is where new element was placed)

1. Consists of
   - methods that are `public` and `abstract`
   - fields that are `public`, `static`, and `final`
2. Implemented by a class that
   - declares it `implements` the interface and
   - defines implementations for all the interface methods
3. Multiple interfaces are allowed

Basics of Interfaces

```
1 // MemoryCell class
2 //  Object read( )         -->  Returns the stored value
3 //  void write( Object x ) -->  x is stored
4
5 public class MemoryCell
6 {
7         // Public methods
8     public Object read( )         { return storedValue; }
9     public void write( Object x ) { storedValue = x; }
10
11        // Private internal data representation
12    private Object storedValue;
13 }
```

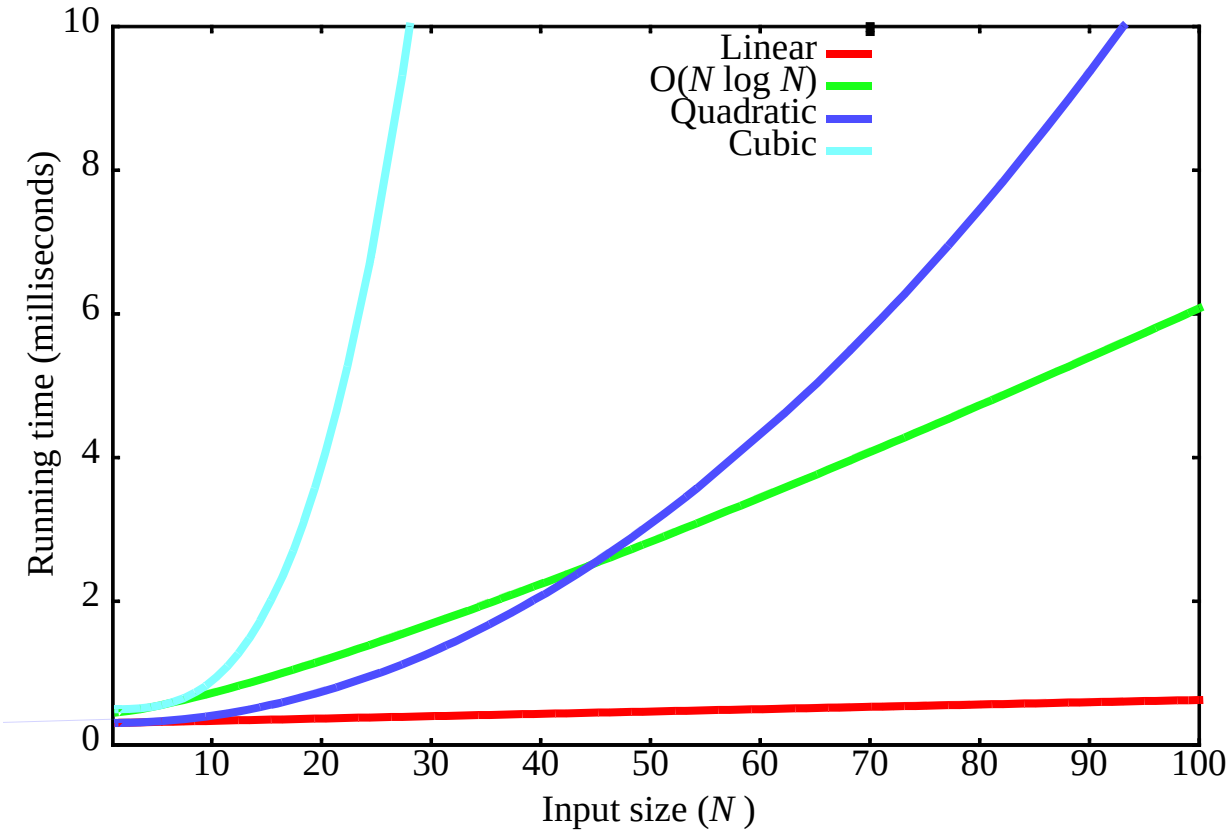
Generic `MemoryCell` class; implemented via inheritance

```
 1 public class TestMemoryCell
 2 {
 3     public static void main( String [ ] args )
 4     {
 5         MemoryCell m = new MemoryCell( );
 6
 7         m.write( new Integer( 5 ) );
 8         System.out.println( "Contents are: " +
 9                     ( (Integer) m.read( ) ).intValue( ) );
10     }
11 }
```

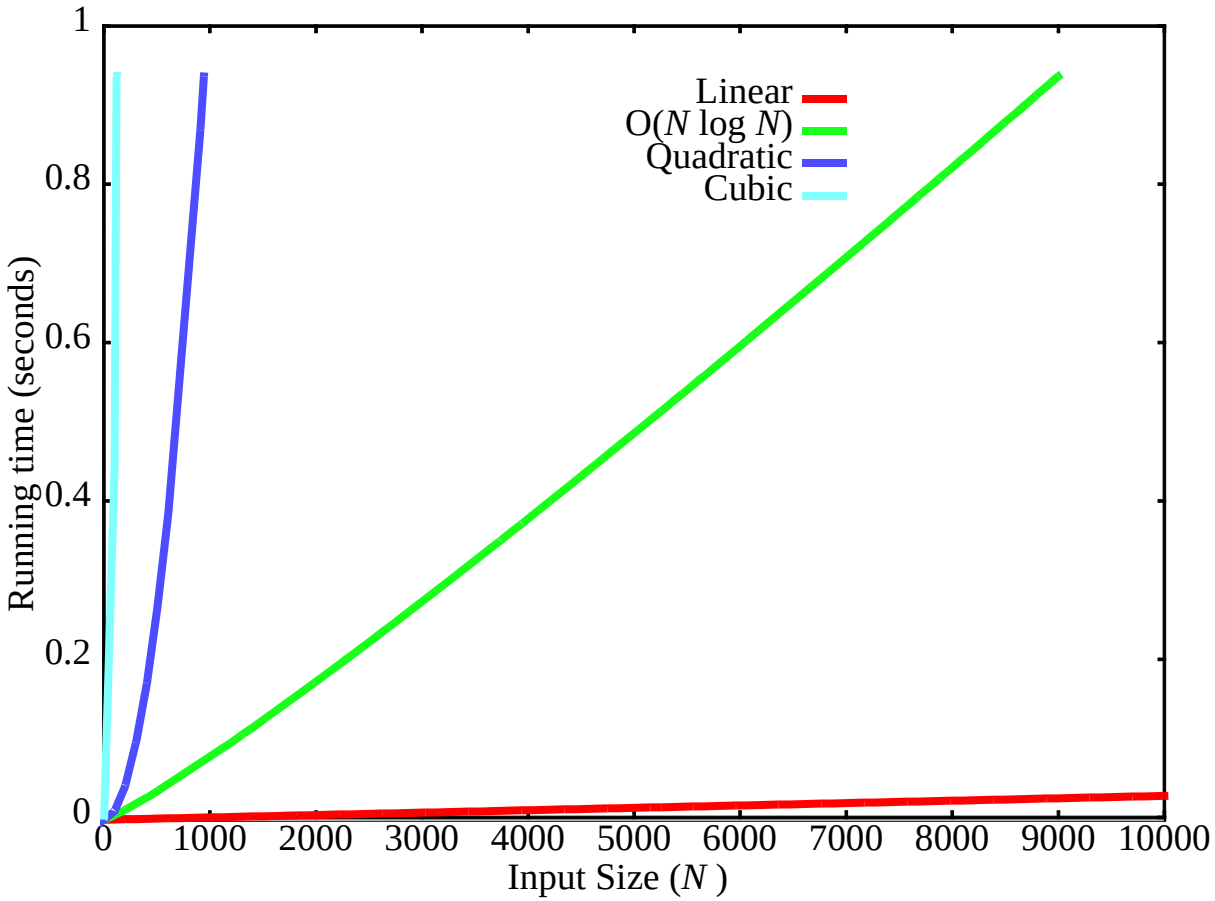
Using the generic `MemoryCell` class

Chapter 5

Algorithm Analysis



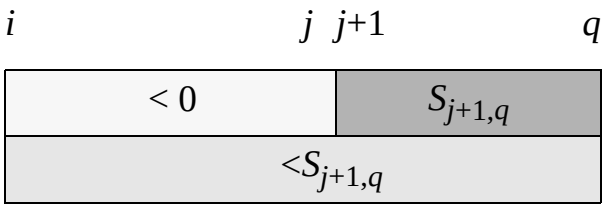
Running times for small inputs



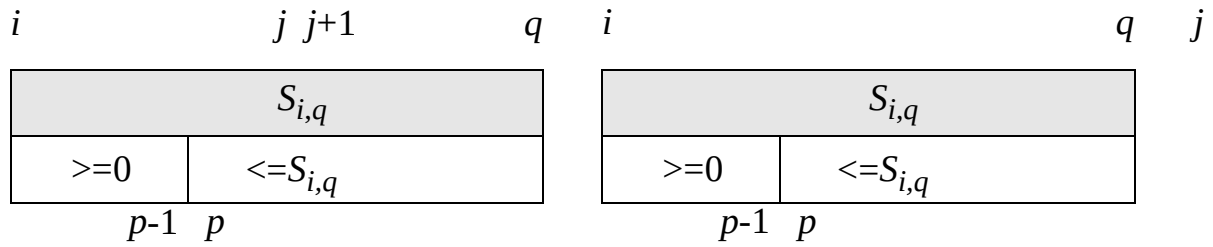
Running time for moderate inputs

Function	Name
c	Constant
$\log N$	Logarithmic
$\log^2 N$	Log-squared
N	Linear
$N \log N$	$N \log N$
N^2	Quadratic
N^3	Cubic
2^N	Exponential

Functions in order of increasing growth rate



The subsequences used in Theorem 5.2



The subsequences used in Theorem 5.3. The sequence from *p* to *q* has sum at most that of the subsequence from *i* to *q*. On the left, the sequence from *i* to *q* is itself not the maximum (by Theorem 5.2). On the right, the sequence from *i* to *q* has already been seen.

DEFINITION: (Big-Oh) $T(N) = O(F(N))$ if there are positive constants c and N_0 such that $T(N) \leq cF(N)$ when $N \geq N_0$.

DEFINITION: (Big-Omega) $T(N) = \Omega(F(N))$ if there are positive constants c and N_0 such that $T(N) \geq cF(N)$ when $N \geq N_0$.

DEFINITION: (Big-Theta) $T(N) = \Theta(F(N))$ if and only if $T(N) = O(F(N))$ and $T(N) = \Omega(F(N))$.

DEFINITION: (Little-Oh) $T(N) = o(F(N))$ if and only if $T(N) = O(F(N))$ and $T(N) \neq \Theta(F(N))$.

Growth rates defined

Mathematical expression	Relative rates of growth
$T(N) = O(F(N))$	Growth of $T(N)$ is $\leq$ growth of $F(N)$
$T(N) = \Omega(F(N))$	Growth of $T(N)$ is $\geq$ growth of $F(N)$
$T(N) = \Theta(F(N))$	Growth of $T(N)$ is = growth of $F(N)$
$T(N) = o(F(N))$	Growth of $T(N)$ is $<$ growth of $F(N)$

Meanings of the various growth functions

N	$O(N^3)$	$O(N^2)$	$O(N \log N)$	$O(N)$
10	0.00103	0.00045	0.00066	0.00034
100	0.47015	0.01112	0.00486	0.00063
1,000	448.77	1.1233	0.05843	0.00333
10,000	NA	111.13	0.68631	0.03042
100,000	NA	NA	8.01130	0.29832

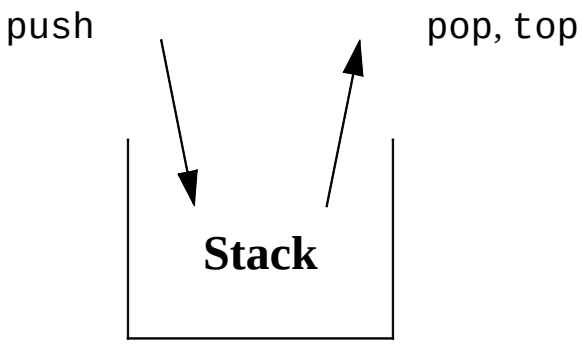
Observed running times (in seconds) for various maximum contiguous subsequence sum algorithms

N	**CPU time T (milliseconds)**	T/N	T/N^2	$T/(N \log N)$
10,000	100	0.01000000	0.00000100	0.00075257
20,000	200	0.01000000	0.00000050	0.00069990
40,000	440	0.01100000	0.00000027	0.00071953
80,000	930	0.01162500	0.00000015	0.00071373
160,000	1960	0.01225000	0.00000008	0.00070860
320,000	4170	0.01303125	0.00000004	0.00071257
640,000	8770	0.01370313	0.00000002	0.00071046

Empirical running time for *N* binary searches in an *N*-item array

Chapter 6

Data Structures

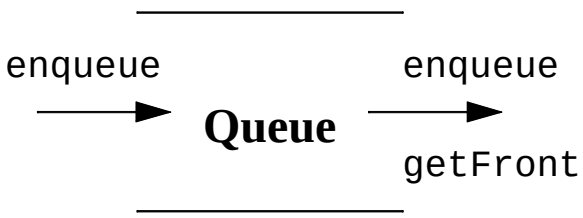


Stack model: input to a stack is by `push`, output is by `top`, deletion is by `pop`

```
 1 import DataStructures.*;
 2 import Exceptions.*;
 3
 4 // Simple test program for stacks
 5
 6 public final class TestStack
 7 {
 8     public static void main( String [ ] args )
 9     {
10         Stack s = new StackAr( );
11
12         for( int i = 0; i < 5; i++ )
13             s.push( new Integer( i ) );
14
15         System.out.print( "Contents:" );
16         try
17         {
18             for( ; ; )
19                 System.out.print( " " + s.topAndPop( ) );
20         }
21         catch( Underflow e ) { }
22
23         System.out.println( );
24     }
25 }
```

Sample stack program; output is

```
Contents: 4 3 2 1 0
```

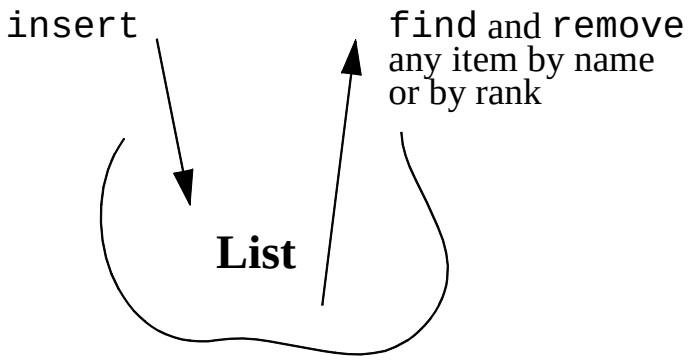



Queue model: input is by `enqueue`, output is by `getFront`, deletion is by `dequeue`

```
 1 import DataStructures.*;
 2 import Exceptions.*;
 3
 4 // Simple test program for queues
 5
 6 public final class TestQueue
 7 {
 8     public static void main( String [ ] args )
 9     {
10         Queue q = new QueueAr( );
11
12         for( int i = 0; i < 5; i++ )
13             q.enqueue( new Integer( i ) );
14
15         System.out.print( "Contents:" );
16         try
17         {
18             for( ; ; )
19                 System.out.print( " " + q.dequeue( ) );
20         }
21         catch( Underflow e ) { }
22
23         System.out.println( );
24     }
25 }
```

Sample queue program; output is

```
Contents:0 1 2 3 4
```

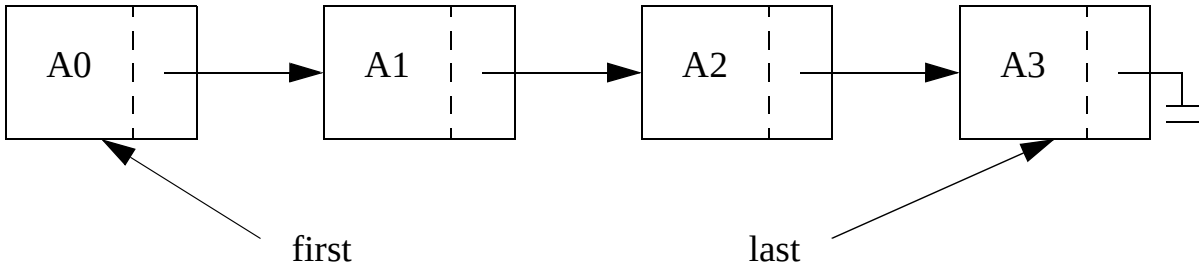



Link list model: inputs are arbitrary and ordered, any item may be output, and iteration is supported, but this data structure is not time-efficient

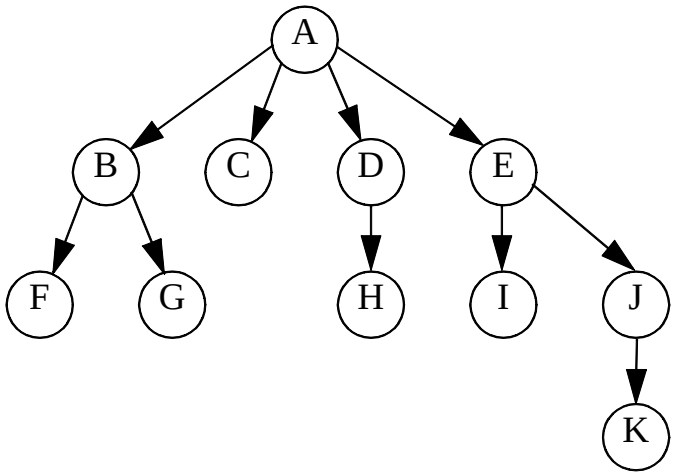
```
 1 import DataStructures.*;
 2 import Exceptions.*;
 3
 4 // Simple test program for lists
 5
 6 public final class TestList
 7 {
 8     public static void main( String [ ] args )
 9     {
10         List    theList   = new LinkedList( );
11         ListItr itr = new LinkedListItr( theList );
12
13             // Repeatedly insert new items as first elements
14         for( int i = 0; i < 5; i++ )
15         {
16             try
17               { itr.insert( new Integer( i ) ); }
18             catch( ItemNotFound e ) { }     // Cannot happen
19             itr.zeroth( );          // Reset itr to the start
20         }
21
22         System.out.print( "Contents:" );
23         for( itr.first( ); itr.isInList( ); itr.advance( ) )
24             System.out.print( " " + itr.retrieve( ) );
25         System.out.println( " end" );
26     }
27 }
```

Sample list program; output is

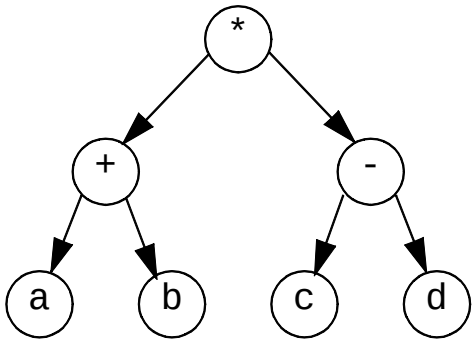
```
Contents: 4 3 2 1 0 end
```

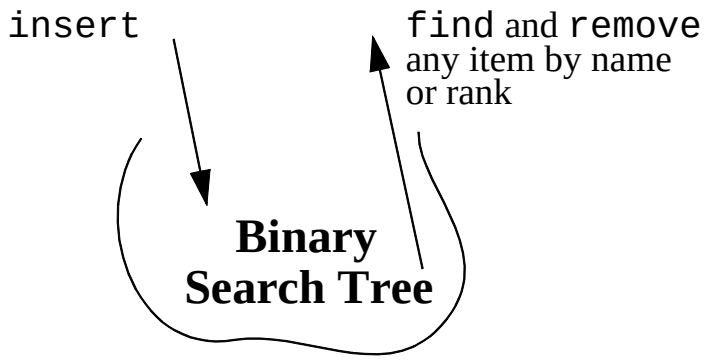
A simple linked list



A tree



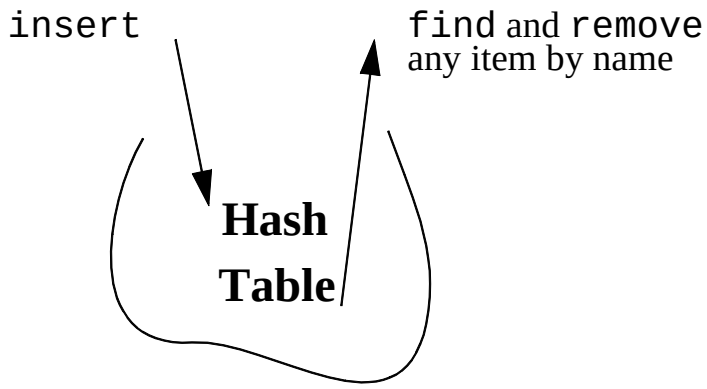
Expression tree for (a+b)*(c-d)



Binary search tree model; the binary search is extended to allow insertions and deletions

```
 1 import DataStructures.*;
 2 import Exceptions.*;
 3
 4 // Simple test program for search trees
 5
 6 public final class TestSearchTree
 7 {
 8     public static void main( String [ ] args )
 9     {
10         SearchTree t = new BinarySearchTree( );
11         MyString result = null;
12
13         try { t.insert( new MyString( "Becky" ) ); }
14         catch( DuplicateItem e ) { }  // Cannot happen
15
16         try
17         {
18             result =
19                  (MyString) t.find( new MyString( "Becky" ) );
20             System.out.print( "Found " + result + ";" );
21         }
22         catch( ItemNotFound e )
23           { System.out.print( "Becky not found;" ); }
24
25         try
26         {
27             result =
28                  (MyString) t.find( new MyString( "Mark" ) );
29             System.out.print( " Found " + result + ";" );
30         }
31         catch( ItemNotFound e )
32           { System.out.print( " Mark not found;" ); }
33
34         System.out.println( );
35     }
36 }
```

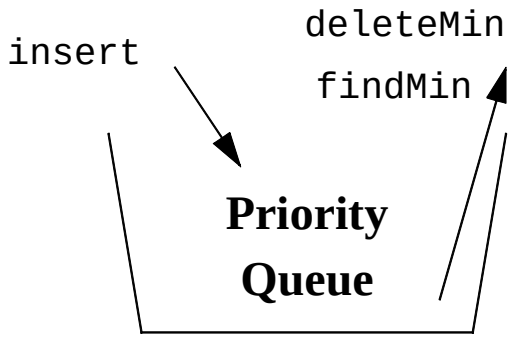
Sample search tree program;
output is `Found Becky; Mark not found;`



The hash table model: any named item can be accessed or deleted in essentially constant time


```
 1 import DataStructures.*;
 2 import Exceptions.*;
 3
 4 // Simple test program for hash tables
 5
 6 public final class TestHashTable
 7 {
 8     public static void main( String [ ] args )
 9     {
10         HashTable h = new QuadraticProbingTable( );
11         MyString result = null;
12
13         h.insert( new MyString( new String( "Becky" ) ) );
14
15         try
16         {
17             result = (MyString)
18                              h.find( new MyString( "Becky" ) );
19             System.out.println( "Found " + result );
20         }
21         catch( ItemNotFound e )
22           { System.out.println( "Becky not found" ); }
23     }
24 }
```

Sample hash table program;
output is `Found Becky;`



Priority queue model: only the minimum element is accessible

```
 1 import DataStructures.*;
 2 import Exceptions.*;
 3 import Supporting.*;
 4
 5 // Simple test program for priority queues
 6 public final class TestPriorityQueue
 7 {
 8     public static void main( String [ ] args )
 9     {
10         PriorityQueue pq = new PairHeap( );
11
12         pq.insert( new MyInteger( 4 ) );
13         pq.insert( new MyInteger( 2 ) );
14         pq.insert( new MyInteger( 1 ) );
15         pq.insert( new MyInteger( 3 ) );
16         pq.insert( new MyInteger( 0 ) );
17
18         System.out.print( "Contents: " );
19         try
20         {
21             for( ; ; )
22                 System.out.print( " " + pq.deleteMin( ) );
23         }
24         catch( Underflow e ) { }
25
26         System.out.println( );
27     }
28 }
```

Sample program for priority queues;
output is `Contents: 0 1 2 3 4`

Data Structure	Access	Comments
Stack	Most recent only, `pop`, $O(1)$	Very very fast
Queue	Least recent only, `dequeue`, $O(1)$	Very very fast
Linked list	Any item	$O(N)$
Search Tree	Any item by name or rank, $O(\log N)$	Average case, can be made worst case
Hash Table	Any named item, $O(1)$	Almost certain
Priority Queue	`findMin`, $O(1)$, `deleteMin`, $O(\log N)$	`insert` is $O(1)$ on average $O(\log N)$ worst case

Summary of some data structures