program ***good***? It works as specified!).easy to understand and modify, reasonably efficient.

The idea of an ADT is to separate the notions of **specification** (what kind of thing we're working with and what operations can be performed on it) and **implementation** (how the thing and its operations are actually implemented). (1)Code is easier to understand (e.g., it is easier to see "high-level" steps being performed, not obscured by low-level code).(2)Implementations of ADTs can be changed (e.g., for efficiency) without requiring changes to the program that uses the ADTs.(3)ADTs can be reused in future programs.

each ADT corresponds to a **class** (or **Java interface** - more on this later) and the operations on the ADT are the class/interface's **public methods**. The user, or client, of the ADT only needs to know about the method **interfaces**(the names of the methods, the types of the parameters, what the methods do, and what, if any, values they return), not the actual implementation (how the methods are implemented, the private data members, private methods, etc.).

There are two parts to each ADT:

1. The **public** or **external** part, which consists of:(1)the conceptual picture (the user's view of what the object looks like, how the structure is organized)(2)the conceptual operations (what the user can do to the ADT)
2. The **private** or **internal** part, which consists of:(1)the representation (how the structure is actually stored) (2)the implementation of the operations (the actual code)

1. Initialize 2. add data 3. access data 4.remove data

A Java interface is a reference type (like a class) that contains only (public) method signatures and class constants. Everything contained in a Java interface is public so you do not need to include the public keyword in the method signatures (although you can if you want). (1)include "implements InterfaceName" after the name of the class at the beginning of the class's declaration, and (2) for each method signature given in the interface InterfaceName, define a public method with the exact same signature

an Iterator, which is an interface defined in java.util. Every Java class that implements the Iterable interface (which includes classes that implement the subinterface Collection) provides an iterator method that returns an Iterator for that collection **import java.util.*; // or import java.util.Iterator;**

For black-box testing, the testers know nothing about how the code is actually implemented (or if you're doing the testing yourself, you pretend that you know nothing about it). Tests are written based on what the code is supposed to do and the tester thinks about issues like:

- testing all of the operations in the interface
- testing a wide range of input values, especially including "boundary" cases
- testing both "legal" (expected) inputs as well as unexpected ones

For white-box testing, the testers have access to the code and the usual goal is to make sure that every line of code executes at least once. So for example, if you were to do white-box testing of the ArrayList class, you should be sure to write a test that causes the array to become full so that the expandArray method is tested (something a black-box tester may not test, since they don't even know that ArrayLists are implemented using arrays).

Every exception is either a **checked** exception or an **unchecked** exception. If a method includes code that could cause a ***checked*** exception to be thrown, then:

- the exception must be declared in the method header using a **throws clause**, or
- the code that might cause the exception to be thrown must be inside a try block with a catch clause for that exception.

in general, you must always include some code that acknowledges the possibility of a checked exception being thrown. If you don't, you will get an error when you try to compile your code. (The compiler does not check to see if you have included code that acknowledges the possibility of an ***unchecked*** exception being thrown.)
The answer lies in the exception hierarchy:

- If an exception is a subclass of RuntimeException, then it is ***unchecked***.
- If an exception is a subclass of Exception but not a subclass of RuntimeException, then it is ***checked***.

Exceptions can be built-in (actually, defined in one of Java's standard libraries) or user-defined. Here are some examples of built-in exceptions with links to their documentation:

- ArithmeticException (e.g., divide by zero)
- ClassCastException (e.g., attempt to cast a String object to Integer)
- IndexOutOfBoundsException
- NullPointerException
- NoSuchElementException (e.g., there are no more elements in collection or enumeration)
- FileNotFoundException (e.g., attempt to open a non-existent file for reading)
- StringIndexOutOfBoundsException

Part B: Function f doesn't need to have a throws clause that lists the uncaught exceptions that it might throw because only uncaught ***CHECKED*** exceptions need to be listed in a method's throws clause. The uncaught exceptions that f might throw are all ***UNCHECKED*** exceptions.

1. **Performance**: how much time/memory/disk/... is actually used when a program is run. This depends on the machine, compiler, etc. as well as the code.
2. **Complexity**: how do the resource requirements of a program or algorithm scale, i.e., what happens as the size of the problem being solved gets larger.

- **constructor**: This method allocates the initial array, sets current to -1 and sets numItems to 0. This has nothing to do with the sequence size and is constant-time.
- **add** (to the end of the list): In the worst case, the array was full and you have to allocate a new, larger array, and copy all items. In this case the number of operations is proportional to the size of the list. If the array is not full, this is a constant-time operation (because all you have to do is copy one value into the array and increment numItems).

Formal definition:

A function T(N) is O(F(N)) if for some constant c and for all values of N greater than some value n_0:
T(N) <= c * F(N)
The idea is that T(N) is the ***exact*** complexity of a method or algorithm as a function of the problem size N and that F(N) is an ***upper-bound*** on that complexity (i.e., the actual time/space or whatever for a problem of size N will be no worse than F(N)). In practice, we want the smallest F(N) -- the ***least*** upper bound on the actual complexity.

The only item that can be taken out (or even seen) is the ***most recently added*** (or **top**) item; a Stack is a **Last-In-First-Out** (**LIFO**) abstract data type.

A Queue is a **First-In-First-Out** (**FIFO**) abstract data type. Items can only be added at the **rear** of the queue and the only item that can be removed is the one at the **front** of the queue.

E[] tmp = (E[])(new Object[items.length*2]);

There is no reason to force the front of the queue always to be in items[0], we can let it "move up" as items are dequeued. To do this, we need to keep track of the indexes of the items at the front and rear of the queue (so we need to add two new fields to theArrayQueue class, frontIndex and rearIndex, both of type int).

Note that instead of using incrementIndex we could use the mod operator (%), and write: rearIndex = (rearIndex + 1) % items.length. However, the mod operator is quite slow and it is easy to get that expression wrong, so we will use the auxiliary method (with a check for the "wrap-around" case) instead.

```
private int incrementIndex(int index) {
   if (index == items.length-1)
      return 0;
   else
      return index + 1;
}
```

Stacks are used to manage methods at runtime (when a method is called, its parameters and local variables are pushed onto a stack; when the method returns, the values are popped from the stack). Many parsing algorithms (used by compilers to determine whether a program is syntactically correct) involve the use of stacks. Stacks can be used to evaluate arithmetic expressions (e.g., by a simple calculator program) and they are also useful for some operations on **graphs**, a data structure we will learn about later in the semester.
Queues are useful for many simulations and are also used for some operations on graphs and trees.

A method is **recursive** if it can call itself, either directly or indirectly: Q: Does using recursion usually make your code ***faster***? A: No. Q: Does using recursion usually use ***less memory***? A: No.Q: Then ***why*** use recursion? A: It sometimes makes your code much ***simpler***!

***** RECURSION RULE #1 *****

Every recursive method must have a **base case** -- a condition under which no recursive call is made -- to prevent infinite recursion.

***** RECURSION RULE #2 *****

Every recursive method must **make progress** toward the base case to prevent infinite recursion.

Example 1: Use a String parameter

```
printStr( String S ) {
   if (S.equals("")) {
      return;
   }
   System.out.print( S.substring(0, 1) + " " );
   printStr( S.substring(1) );
}
```

Example 2: Use a linked list parameter

```
printList( Listnode<E> l ) {
   if (l == null) {
      return;
   }
   System.out.println(l.getData());
   printList(l.getNext());
}
```

In general, the recurrence equation for the recursive case will be of the following form:

T(N) = _____ + _____ * T(_____)

The first blank is the **amount of time outside the recursive call(s)**. In the printInt method above, that is constant, so we fill in the first blank with a 1.

The second blank is the **number of recursive calls**. Not the total number of recursive calls that will be made, just the number made here. In the printInt method above, there is just one recursive call, so we fill in the second blank with a 1.

The third blank is the **problem size passed to the recursive call**. It better be less than the current problem size, or the code will not make progress toward the base case! In our example, the problem size (the value of k) decreases by one, so we fill in the third blank with N-1.

The final equation for the recursive case is

T(N) = 1 + 1 * T(N-1)

Which is the same as the equation given above (just with an explicit "times 1"). Now consider a different version of printInt:

```
public static void printInt(int k) {
  if (k == 0) {
     return;
  }
  printInt(k/2);
  for (int j = 0; j < k; j++) {
     System.out.println(j);
  }
  printInt(k/2);
}
```

For this version, the amount of time outside of the recursive calls is proportional to the problem size, there are ***two***recursive calls, and the problem size passed to each of those calls is half the original problem size. Therefore, the recurrence equation for this new version of printInt is
T(N) = N + 2 * T(N/2)

non-linear structure: (1)More than one item can follow another. (2) The number of items that follow can vary from one item to another.

- each letter represents one **node**
- the arrows from one node to another are called **edges**
- the topmost node (with no incoming edges) is the **root** (node A)
- the bottom nodes (with no outgoing edges) are the **leaves** (nodes D, I, G & J)

A **path** in a tree is a sequence of (zero or more) connected nodes;
The **length** of a path is the number of nodes in the path
The **height** of a tree is the length of the longest path from the root to a leaf
The **depth** of a node is the length of the path from the root to that node
Node A is called the **parent**, and node B is called the **child**.
A **subtree** of a given node includes one of its children and all of that child's **descendants**. The descendants of a node N are all nodes reachable from N (N's children, its children's children, etc.).
In a binary tree: (1) Each node has 0, 1, or 2 children. (2)Each child is either a left child or a right child.