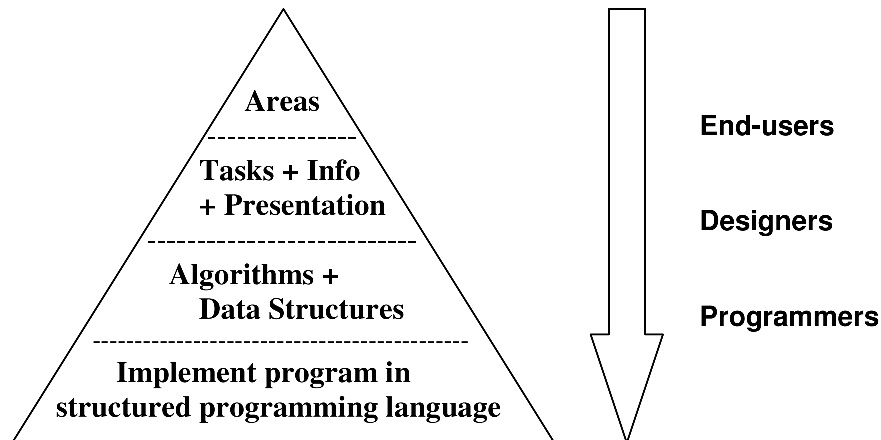
Top-Down vs Object-Oriented Design

Top-Down-Design

The traditional model for software design is called **Top-Down**. It proceeds through these steps:

1) **break down** of the problem into large **areas**
2)**design a solution** by identifying **tasks**, **information**, and **presentation**
3)**describe algorithms** and **data-structures** in **pseudo-code**
4)implement a program in a **standard programming language**



Each level is more detailed than the previous (higher) level. As the design proceeds, it becomes larger and more complex at each level, growing like a pyramid from top (small end) to bottom (big end).

The use of a traditional **structured programming language** like **C** or **Fortran** means that the commands at the bottom level have little or no relationship to the **problem-domain concepts** at the top levels. The **pseudocode** level for algorithms and data-structures provides a transition layer connecting the two, but the connections may be difficult to follow. The **end-user** might be involved in the top two levels, but only programmers can understand the bottom level. This disconnection between descriptions and between people causes lots of problems.

This model was considered "natural" and standard for many years. Unfortunately, it had **several significant shortcomings** :

1) further development becomes more and more difficult, because the complexity grows too rapidly at the bottom, with many small commands

2) revision often requires reworking at all levels (cycling back from bottom to top), which could require throwing away the old system and starting over

3) the modules (pieces) are often **tightly coupled** (interdependent), which means **reuse** is very difficult

4) the **"logical distance"** between the problem domain and programming language is large - that is, the commands in the language are not directly related to the actual problem (see above)

Object Oriented Innovations

In older **high level languages** (like C), data items, data structures, programming constructs, and algorithms were all parts of the language. The **keywords** and **identifiers** formed a long list of unrelated commands. This could be quite confusing, because a procedure name could look like a variable name. From it's beginning as a development language for the Unix operating system in the early 1970's, **C** grew into a confusing collection of data type and function identifiers with very little structure.

A decade later, it was rationalized with the creation of **C++** by Bjarne Stroustrup. C++ extended **C** by adding **classes** and **objects.** But it retained all of the C commands, so programs were still complex and confusing. But the availability of classes and objects in C++ made it the preferred language for developers in the 1980's and enabled long-term development of software projects like Microsoft Windows, MS Office, and other large, complex applications.

Between 1991 and 1995, James Gosling and a group of Sun programmers created **Java**, which is a "pure" **object-oriented** programming language. In Java **everything** (almost) is a **class** or an **object.** Java's big advantages over C++ :

1) Java is **cross-platform**, so developers can write one program and it runs on Windows, Linux, the Mac, and other platforms - this is not true for C++.

2) Java has a very **small set** of standard **keywords** (under 100 compared to many hundreds for C++). Everything else is classes and objects, so programs are easier to read and modules (classes) are widely re-usable

3) Java was created with **Web programming** facilities in mind, and is one of only a few languages with these facilities

4) Java is **"safer"** than C++, because it no longer contains lower level commands (like C). The **memory management** is in Java is **automated** (with automatic Garbage collection), and low level **pointer** commands **(memory allocation and deallocation)** are no longer permitted. Faulty memory-management is responsible for many of the bugs in C++ based software.

Object-Oriented Design

Object-oriented programming (e.g. Java) made only a slight change in the fundamental concept of programming languages, but this change **revolutionized software-design**. The simple change was the creation of **classses** and **objects**, which **encapsulate** both **data** and **algorithms** into easily reusable modules. Object-oriented design means that classes and objects are a fundamental concept at all stages in the design process, so the "top" level of the design is relatively similar to the bottom level. This **reduces the distance** between the problem domain and the programming language - many of the same concepts (classes) appear in both the initial breakdown and the final implementation, thus connecting the levels and making it all more easily understandable. End-users and programmers are using similar terminology and thus can talk to each other more easily. This makes the design and development processes faster, more flexible, and more reliable.

In C-type languages, data-structures are stored in arbitrary places and operated on by algorithms in various modules. In Java, **objects** protect the data-structures so they are only manipulated by algorithms inside the same class. Although it is still possible in Java to allow methods in one class to access data in another class, this is a very **bad idea** and should be avoided whenever possible (almost always). Because the data and algorithms are packaged together, it is more easily possible to **re-use** classes than to re-use modules from C-type languages.

Inheritance further facilitates reuse. Inheritance allows us to start with an existing class and **extend** it by adding a bit more data and adding (or **overriding**) methods, producing exactly the class we need with relatively little work. Thanks to **polymorphism**, common identifiers like "search" or "list" can have different meanings in various classes. This ability to use common words in various ways makes it easier to read and write programs.

Example - Comparison of Top-Down and OO design

Consider the problem of maintaining a **calendar of events**, including a list of participants - this might be used by a school, club, or business. We can quickly list some important parts of the problem:

== Tasks ==

- Save names of all people
- Save list of "real" dates (might not include weekends)
- Input a new event, with title, dates, and participants
  - check that date is valid
  - check that participant names are correct (exist)
  - warn of duplicate entries - e.g. if event with same name already exists, or the same participant is entered twices
- Save events
- Search for an event
- Search for a date
- Find an event by name
- Find events on a specific date
- Print a monthly calendar

Listing the tasks that users wish to perform is an easy way to start. (This list of **tasks** is not complete, but is enough for an initial design). The **tasks** will eventually be refined into **methods** (procedures and functions) - for this example, we can assume that each task above turns into a single method in the program.

Now we need to think about the **information - data items** and **data-structures**.

== Information ==

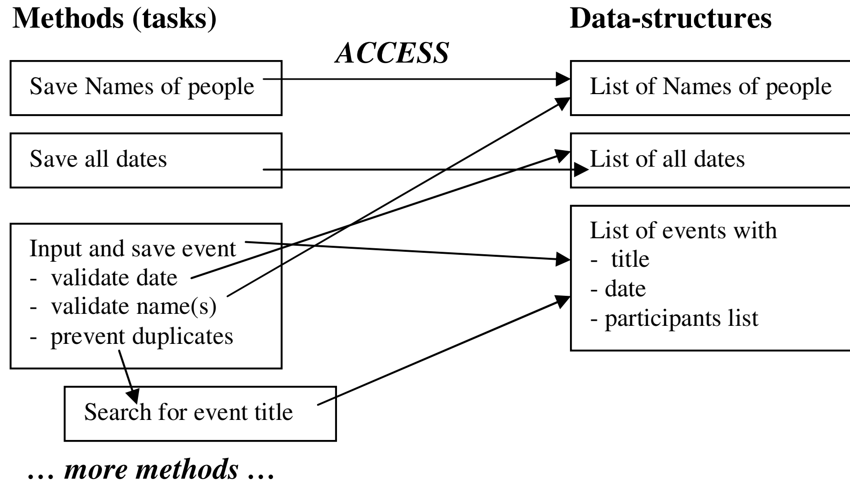
- **Data-Items**
  - **dates**
  - **names** (participants)
  - **titles** (events)
- **Data-Structures**
  - **Event record** = Title/Date/Participant 1/Participant 2/ ...
  - **Events List** = List of all events
  - **Monthly Calendar** = structured table of events on days for one month

More structures may come to mind later.

Top-Down Design and Structured Programming

In traditional programming, all the methods (tasks) are permitted to operate on any of the data items or structures. So it looks something like this:

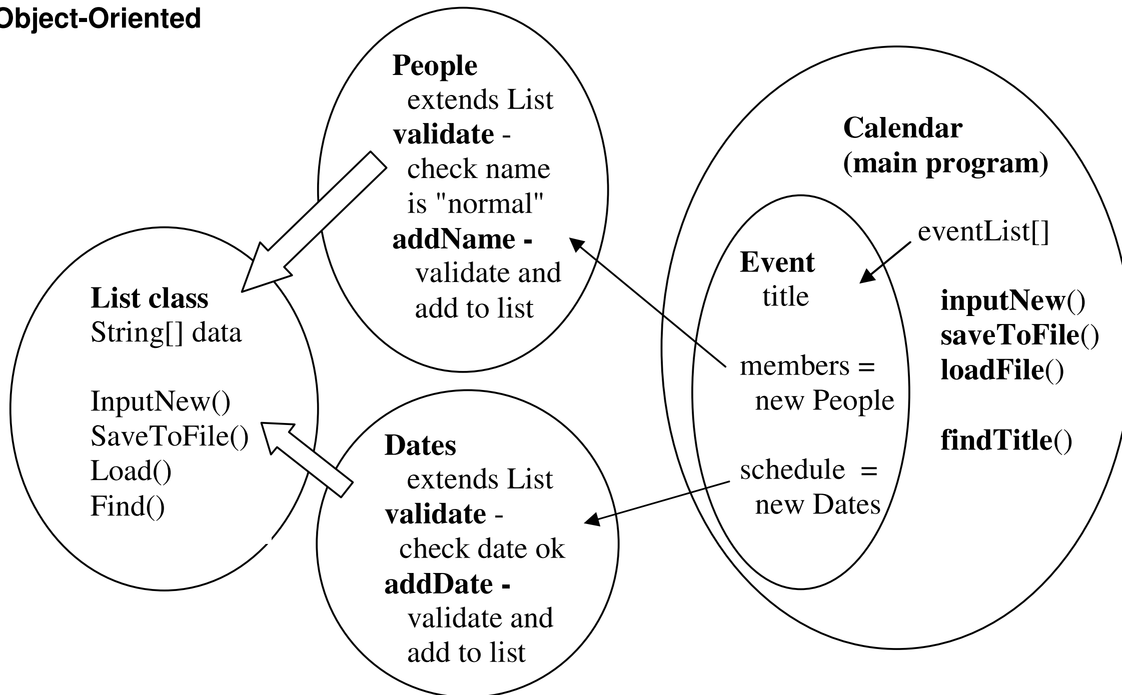
Traditional Top-down design / Structured Programming



… more methods …

We notice that many different methods are accessing the various data-structures. This is not only messy, but also inefficient and dangerous. If there are some **error-trapping** issues (e.g. rejecting bad dates), then each method must implement the error-trapping code. If various programmers write the various methods, there is a danger that one programmer makes assumptions or writes code that conflicts with another programmer's work, thus causing errors. In larger projects, coordinating various programmers' work is a huge issue.

Object-Oriented



In Object-Oriented design and programming, we **encapsulate** (put together and protect) data-structures and related methods in self-contained classes. Then methods only operate on the data in the same class - the data is **protected** from external access. So the design looks more like the diagram above.

The white arrows indicate that one class is using another. The People and Dates classes **inherit** the data and methods from the List class, and extend these with a new **validate()** and **add()** methods, which are different for Dates than for People. The Calendar class (main program) makes an array of Event objects, and takes care of inputting, saving and loading the list of events. The Calendar class is not inheriting (extending) from the Event class - it **instantiates** lots of Event **objects,** and stores **references** in an array (List).

The skinny arrows show that the Event class can ask the People or Dates classes to check a name or date, to make sure it exists. But this should happen through a safe **accessor** method, **NOT** by directly using the array inside the class.

In OO programming, we still have modules accessing other modules, but they do it more "safely" than in traditional structured programming. The "walls" around the classes provide protection through **encapsulation**. Java uses the word **private** to ensure that data members are not accessed directly from outside. Then it must provide **public accessor methods** to change the data. There might be **multiple versions** of similar methods - this is called **polymorphism.** For example, **addDate(String date)** and **addDate(int year, int month, int day)** might be two different ways to add a new Date in a Dates object.

Top-Down vs OO : Which is "better"?

For small applications, **top-down design** and **structured programming** are probably quicker and simpler (and adequate). But **OO design** and **OOP** improve **reusability**, which is more important in larger applications. The many crossing arrows in the structured design will turn into lots and lots of arrows if the application grows - this complexity can grow in a **hyper-linear** fashion. OO designs tend to grow in a more manageable, almost **linear** fashion, as it is common for a class to reference (use) only a few other classes. And the improved **reusability** resulting from **encapsulation and inheritance** means that there is less re-coding from scratch.

In traditional structured programming, we concentrated on **algorithms** (processes), spending time **designing procedures**. In OOP we concentrate on **data structures**, and spend time **discovering classes** that encapsulate (represent) the **state and behavior** of real-world **objects.**

Data-Structure Classes (Abstract Data Types)

All this leads to a discussion of **data-structures**.

In structured programming, the data-structures are separate pieces - for example arrays. The algorithms are added later, wherever they are needed.

In OO programming, a data-structure does not stand alone. Instead, all the **algorithms** (methods) for manipulating the data-structure are created with it inside a **class container.** Then we can **extend** and reuse data-structure classes easily, so it is worthwhile to write a **library** of good, standard data-structure classes. We will spend the next few weeks developing such a library.

Lists, Stacks and Queues – OO Programming

Random Access vs Sequential Access

Random access does not mean unpredictable or without order. Random Access means that the **position** of an item does not effect the speed of access. Hard-disk drives, CD-ROM drives, and RAM (Random Access Memory) or all random-access devices. On a hard-disk, a file might be stored near the center of the disk, or near the outside. But the read/write head moves very quickly to the correct track, so the position is not significant. RAM, any storage location returns data in the same amount of time, regardless of its address.

Sequential access means that items at the beginning of a list are retrieved more quickly than items at the end if the list. Tape-drives are sequential access devices. Text-files are generally sequential access structures – the data items (lines of text) must be written and read in order.

Arrays

Arrays are random-access structures. Arrays are stored in RAM, making it quick and easy to "access" any position in the array. There is no difference in access speed between the following two commands:

```
names[1] = "Alex"

names[1000000] = "Zeke"
```

Printing an array in reverse order is no problem, and runs just as fast as printing forward:

```
for(int x=1000000; x > 0; x=x-1)
   { output(names[x]);}
```

Lists

An array can be used to store a list of words or commands. Normally the entries are copied **sequentially** into the array, like this:

```
count = 0
do
   data = input()
   list[count] = data
   count = count + 1
while !name.equals("xxx")
```

Now the words can be **retrieved** (used) in any order.

FIFO

FIFO stands for **First In, First Out**. This means data is used in the same order that it was saved. This is like a **queue** in a cafeteria. In fact, the term **queue** is a technical computer science term for a list of data used in the same sequential order as it was received.

LIFO

LIFO stands for **Last In, First Out**. This means data is processed in reverse order - the last item received is the first to be processed. This is similar to a **stack** of work, where each new assignment is laid on top of the stack. Then the top item (received last) is taken off the top of the stack and processed first. **Stack** is the technical computer science term for LIFO type processing. This is useful for handling **interrupts**, where the most recent signal demands the immediate attention of the processor.

Examples

- **Cafeteria -** In a narrow line in a cafeteria, customers enter at one end and stand in a queue. They are served when they get to the front of the queue, in **FIFO** order.
- **Bus Passengers -** A bus has a single entrance at the front. Passengers enter and move to the back of the bus. When they exit, the passengers at the front must exit first, in **LIFO** order - Last In, First Out.
- **Restaurants -** Customers enter a restaurant, sit at a table, and wait to be served. The position of the table doesn't affect the service. If all the waiters are efficient, the service should be basically FIFO (e.g. fair), but this isn't necessary, and doesn't always happen. Here the service is **random-access** - e.g. no particular order is required.
- **Print Queue** - A network allows many PCs to use a single printer. Each **print job** (document) arrives at a **server** and is stored temporarily in a **queue.** When the printer finishes with one job, it removes another job from the queue. Using queue-oriented access (FIFO), the first print job entered in the queue is the first to be printed. People consider this "fair".
- **Interrupts** - Computers use **interrupt** signals when a peripheral device (e.g. printer) needs attention from the CPU (PC). A PC might receive a request from a printer to send the next print job. Before it can start, it might receive another interrupt from the modem to receive some data from the interrupt. The PC normally stops working on the print request and services the modem request immediately - e.g. LIFO.

Priorities

Some jobs are more urgent or more important than others. In a hospital, the emergency room patients are more urgent than those who already have a bed. In computer systems, a request from a modem to receive data is more urgent than a request from a printer - the printer can wait, but the modem must keep pace with a web-server somewhere else. To ensure that an urgent job gets attention before a less urgent job, computer systems can assign **priority** codes to devices - e.g. 99 for a high priority, 0 for a low priority. Prioritized data storage can be implemented in a **deque.** A **deque** (double ended queue) works like a queue at the back end, and like a stack at the front end. If a request arrives with a high priority (higher than the item at the front of the queue) it will be "pushed" onto the front, and processed next. This is like permitting a teacher to **push** in at the front of the cafeteria queue, ahead of the students.

Standard Operations

Stack (LIFO)

- **Initialize** - start with the stack empty
- **Push** - add a new item at the **top** of the stack
- **Pop -** remove an item from the **top** of the stack (and process it)

Queue (FIFO)

- **Initialize** - start with the queue empty
- **Enqueue -** enter the queue (at the back)
- **Dequeue** - leave the queue (at the front, after waiting)

Deque (pronounced "Deck")
- implements all the operations from both a queue and stack,
where we recognize that Pop and Dequeue are actually the same

- **Initialize** - start with an empty list
- **Enqueue -** join the **back** of the queue
- **Push -** join at the front of the queue
- **Dequeue -** remove and process the item from the front of the queue (same as Pop)

ADT - Abstract Data Structure

Stacks, Queues, and Deques are examples of Abstract Data Structures - they describe some general concepts which can be applied in a variety of situations. For example, it is possible to **implement** a queue by storing data in a text-file, or by storing it in an array. The file could be stored on a disk-drive, or a tape, or some other storage device. The data might be names, or dates, or e-mail addresses, or anything else. So the requirements for the ADT are **abstract** - that is, general rather than specific.

Programming Stacks

A stack can be stored in an array, with an integer variable TOP which tells where the stack ends. Here is a stack containing Al, Bob, and Carla, where Carla entered last.

0 : position 0 is not used
1 : Al
2 : Bob
3 : Carla TOP = 3

If an item is removed, Carla will be first.

The programming code looks like this:

```
int top = 0;
String[] stack = new String[100];

public void initialize()}
{ top = 0; }

public void push(String newItem)
{ top = top + 1;
  stack[top] = newItem;
}

public String pop()
{ String item = stack[top];
  top = top - 1;
  return item;
}
```

Programming Queues

A queue is a bit more complicated to program than a stack. It needs to keep track of where the **front** of the queue is, as well as the **back.** Here is a queue with Al at the front and Carla at the back.

```
0 :
1 : Al        FRONT = 1
2 : Bob
3 : Carla
4 :           BACK = 4  (next empty position)
```

The programming code looks like this:

```
int front = 0;
int back = 0;
String[] queue = new String[100];

public void initialize()
{ front = 0;
  back = 0;
}

public void enqueue(String newItem)
{ queue[back] = newItem;
  back = back + 1;
}

public String dequeue()
{ String item = queue[front];
  front = front + 1;
  return item;
}
```

Handling Overflow and Underflow Errors

Usually printers are sitting around doing nothing. That means there is no data in the print-queue. After printing the last job in the queue, an attempt to **dequeue** the next job causes an **underflow** - that means there is no more data, so the dequeue operation fails. The command "front=front+1" doesn't cause an arithmetic error, but the data in position [front] is actually meaningless. So the **dequeue** method should not return anything, except possibly an error code. Study the following sequence of queue operations:

Task-> Cell #	#1 Init	#2 EnQ ("Al")	#3 EnQ ("Bob")	#4 DeQ	#5 EnQ ("Cho")	#6 DeQ	#7 DeQ	#8 DeQ
0	front, back	[Al] front	[Al] front					
1		back	[Bob]	[Bob] front	[Bob] front			
2			back	back	[Cho]	[Cho] front		
3					back	back	front, back	back
4								front

After **dequeue** removes "Cho", **front** and **back** are both equal to 3. This means there is actually no data in the queue. Now a **deQueue** command will return the contents of **queue[3]**, and increase front to 4. But this is nonsense.

This **underflow** situation must be **prevented** (handled)**,** by adding an **if...** command to prevent the problem. Something like the following:

```
public String dequeue()
{ if (front<back)
  { String item = queue[front];
    front = front + 1;
    return item;
  }
  else
  { return "***"; }    //  *** represents an error situation //
}
```

As the program continues running, handling more and more data, we notice that the variables **front** and **back** continue to increase. If **queue** is an array of 100 cells, front and back will eventually become larger than 100, causing an **overflow** error. This will actually cause the program to crash, so it must be prevented. It is not quite as simple as handling the underflow error. Once this happens, the program must either shut down, or front and back must be **reset** to start over at 0. But this won't be simple if back reaches 100 when there is still data in the queue.

This program demonstrates the basic functions of a Queue.

```
import java.awt.*;
import java.awt.event.*;

public class QueueTest extends EasyApp
{ public static void main(String[] args)
  { new QueueTest(); }

  //--- Global Variables for the Queue ---
  int front = 0;
  int back = 0;
  String[] queue = new String[100];

  //--- Controls - added to form ------
  Button bEnq = addButton("Enqueue",40,50,70,20,this);
  Button bDeq = addButton("Dequeue",40,80,70,20,this);
  Button bCheck = addButton("Check",40,110,70,20,this);
  Button bQuit = addButton("Quit",40,140,70,20,this);

  public QueueTest()
  //--- Constructor - Adjust controls at start  ---
  { bQuit.setBackground(Color.red);
    setBounds(0,0,150,200);
    setVisible(true);
  }

  public void actions(Object source,String command)
  //--- Perform actions when buttons are clicked ---
  {
    if (source == bQuit)
    { System.exit(0);
    }
    else if (source == bEnq)
    { String name = input("Add a name:");
      enqueue(name);
    }
    else if (source == bDeq)
    { String name = dequeue();
      output(name);
    }
    else if (source == bCheck)
    { output("Current pointers :" +
               "\n  front = " + front +
               "\n  back = " + back);
    }
  }

  public void initialize()
  { front = 0;
    back = 0;
  }

  public void enqueue(String newItem)
  { queue[back] = newItem;
    back = back + 1;
  }

  public String dequeue()
  { String item = queue[front];
    front = front + 1;
    return item;
  }
}
```

---

(1) Random List - Attendance program

(a) Create a **NameList** class containing a String[] array called **names**, for 1000 Strings.
Use the following command:
private String[] items = new String[1000];

(b) Write a **mutator** method for putting data into **items[]** - use the following **method signature:**
public void put(String data, int pos)

The method should copy **data** into position **pos** in the **names[]** array, using this command:
items[pos] = data ;

(c) Write an **accessor** method for getting data from **names[]** - use the following **method signature:**
public String get(int pos)

The method should **return** the data item from position **pos** in the **names[]** array, like this:
return items[pos];

(d) Create a *separate* program (class) called **Attendance** that creates **two NameList** objects - one NameList will contain names of **students** in the course, and the other will be used to contain the names of **absent** students.
NameList students = new NameList();
NameList absent = new NameList();

(e) Create a **user-interface** containing a single button called **[Absent].**
When the user clicks the button, and **input Dialog** inputs the name of an absent student, and this name is **put** into the **absent** NameList. Notice that the user should **not** be required to decide the position for saving the name into the list - this needs to be **automatically** decided by the program.

(f) In the NameList class, create a method called **append** that adds a new String at the end of the list - that is not position 999, but rather the first *blank* position in the list.
Now change the Attendance program so that it uses the **append** method to add a name to the absent[] list.

(g) Put 10 names into the **students** list at the beginning of the program.
Then create a List control and use it to **display** the names of all the students in the course.
DON'T just put the names of the students into the List display. Put the names into the students NameList, then **copy** the names into the List display.

(2) Error trapping

(a) Identify likely **error-conditions** in the Attendance program and/or the List class.

(b) Add **error-trapping** and/or **error-handling** code for some of the problems identified in (a).

(3) Get a copy of the **QueueTest** program : http://ibcomp.fis.edu/design/QueueTest.java

(a) Run the program, and try typing in the sequence of items shown in the table above.
What happens when the Queue "underflows"? Does the program crash?
Now try to Enqueue a new item AFTER the underflow occurs. What does the Dequeue do now?

(b) Add the underflow prevention code mentioned above, and make it work correclty.
This means that an underflow does not cause any problems.

(c) Change the entire program so that it implements a STACK instead of a QUEUE.
Use the same code shown above if you wish. Otherwise, write your own code.

(d) Add error checking code to prevent and underflow in the STACK program.
This is simpler than a queue - here an underflow occurs if (TOP < 1).

(e) Change the STACK program so that it only has 10 cells in the stack, instead of 100.
Add items until an OVERFLOW occurs (9 or 10 or 11 items).

(f) Now add some code to prevent an overflow. This means that new items **cannot** be pushed onto the stack when it already contains 10 items (or is it 9?).