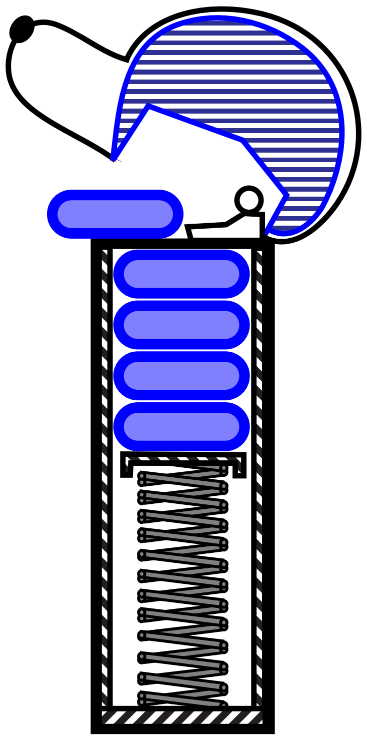
# STACKS

- Abstract Data Types (ADTs)
- Stacks
- Application to the analysis of a time series
- Java implementation of a stack
- Interfaces and exceptions

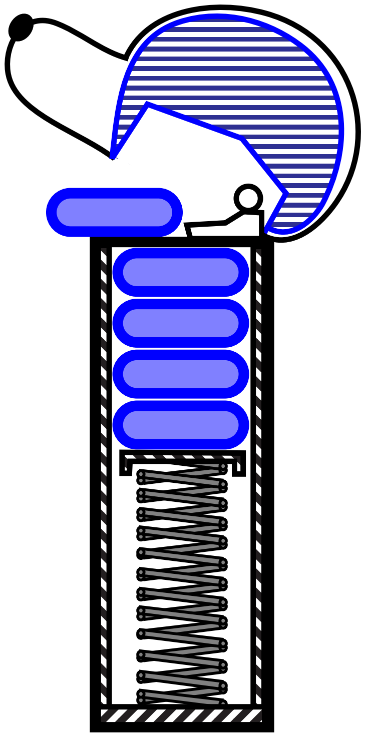


## Abstract Data Types (ADTs)

- An **Abstract Data Type** is an abstraction of a data structure: no coding is involved.
- The ADT specifies:
  - what can be stored in the ADT
  - what operations can be done on/by the ADT
- For example, if we are going to model a bag of marbles as an ADT, we could specify that
  - this ADT stores marbles
  - this ADT supports putting in a marble and getting out a marble.
- There are lots of formalized and standard ADTs. A bag of marbles is not one of them.
- In this course we are going to learn a lot of different standard ADTs. (stacks, queues, trees...)

## Stacks

- A stack is a container of objects that are inserted and removed according to the last-in-first-out (LIFO) principle.
- Objects can be inserted at any time, but only the last (the most-recently inserted) object can be removed.
- Inserting an item is known as “pushing” onto the stack. “Popping” off the stack is synonymous with removing an item.
- A PEZ® dispenser as an analogy:

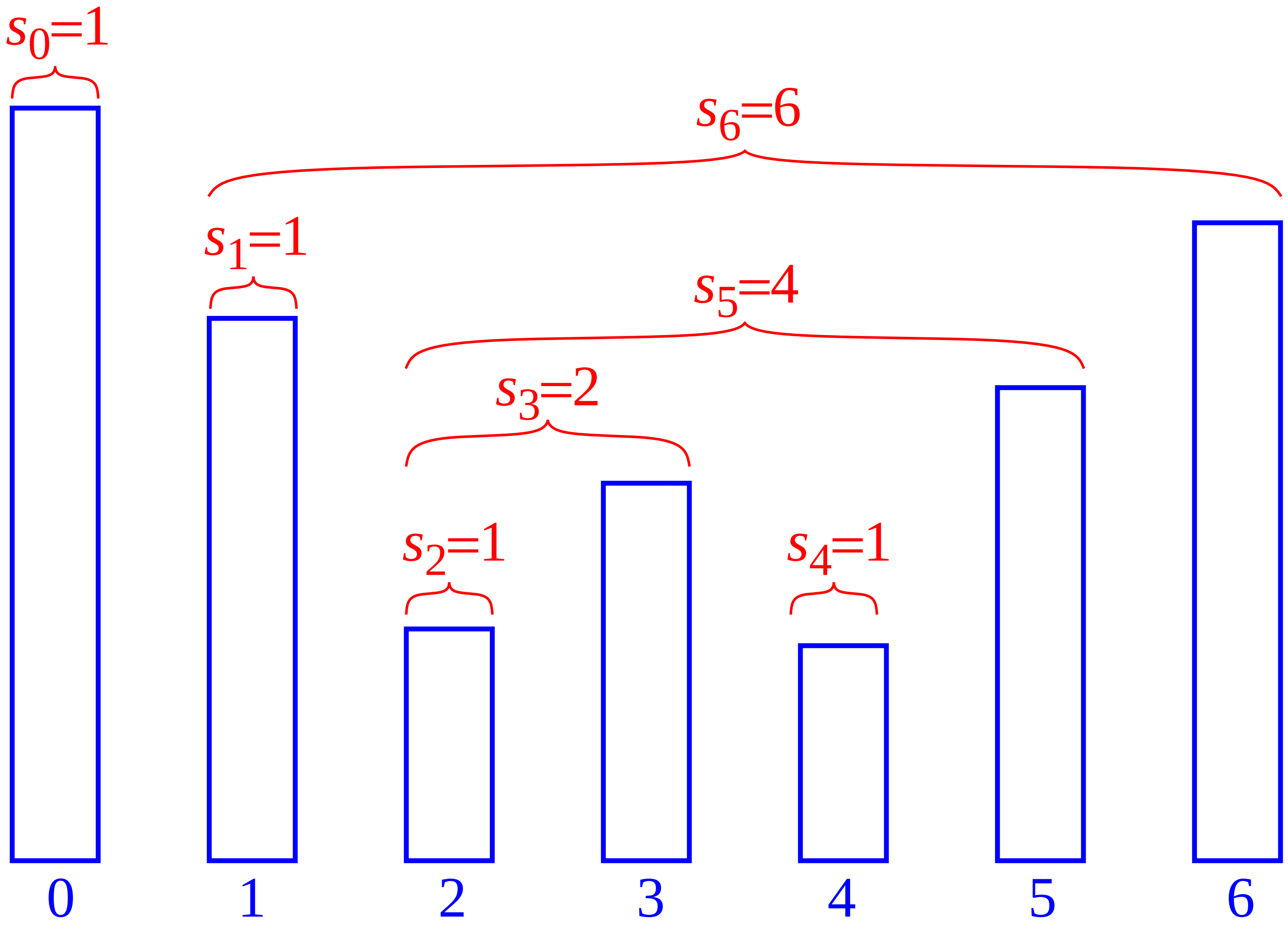


## The Stack Abstract Data Type

- A stack is an abstract data type (ADT) that supports two main methods:
  - push(*o*): Inserts object *o* onto top of stack
  - pop(): Removes the top object of stack and returns it; if the stack is empty, an error occurs
- The following support methods should also be defined:
  - size(): Returns the number of objects in stack
  - isEmpty(): Return a boolean indicating if stack is empty.
  - top(): return the top object of the stack, without removing it; if the stack is empty, an error occurs.

# Application: Time Series

- The ***span*** $s_i$ of a stock's price on a certain day *i* is the maximum number of consecutive days (up to the current day) the price of the stock has been less than or equal to its price on day *i*.



## An Inefficient Algorithm

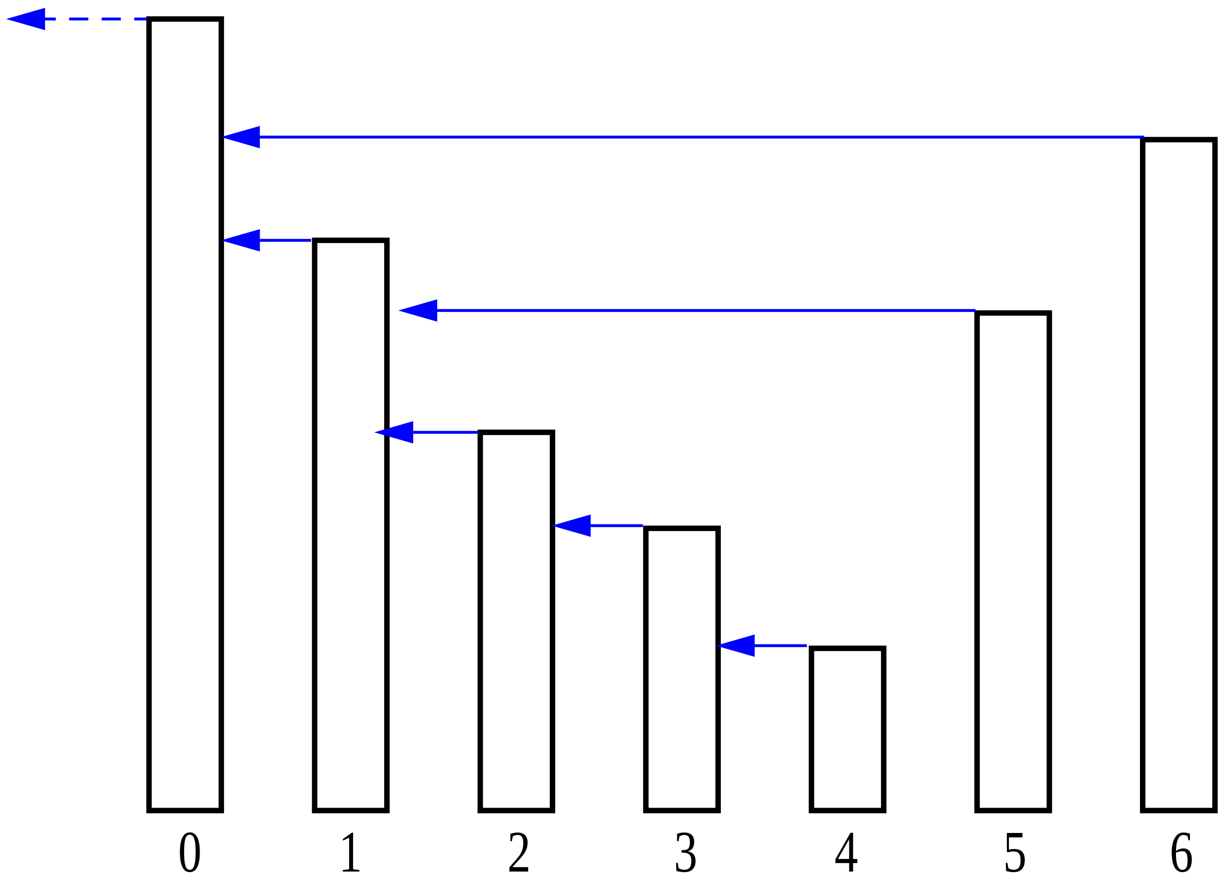
- There is a straightforward way to compute the span of a stock on each of *n* days:

```
Algorithm computeSpans1(P):
  Input: an n-element array P of numbers such that
         P[i] is the price of the stock on day i
  Output: an n-element array S of numbers such that
          S[i] is the span of the stock on day i
  for $i \leftarrow 0$ to $n - 1$ do
    $k \leftarrow 0$
    $done \leftarrow$ false
    repeat
      if $P[i - k] \le P[i]$ then
        $k \leftarrow k + 1$
      else
        $done \leftarrow$ true
    until $(k = i)$ or $done$
    $S[i] \leftarrow k$
  return S
```

- The running time of this algorithm is (ugh!) $O(n^2)$. Why?

## A Stack Can Help

- We see that $s_i$ on day $i$ can be easily computed if we know the closest day preceding $i$, such that the price is greater than on that day than the price on day $i$. If such a day exists, let's call it $h(i)$, otherwise, we conventionally define $h(i) = -1$
- The span is now computed as $s_i = i - h(i)$



We use a ***stack*** to keep track of $h(i)$

## An Efficient Algorithm

- The code for our new algorithm:

```
Algorithm computeSpan2($P$):
  Input: An $n$-element array $P$ of numbers representing
         stock prices
  Output: An $n$-element array $S$ of numbers such that
         $S[i]$ is the span of the stock on day $i$
  Let $D$ be an empty stack
  for $i \leftarrow 0$ to $n - 1$ do
    $done \leftarrow$ false
    while not($D$.isEmpty() or $done$) do
      if $P[i] \geq P[D.\text{top}()]$ then
        $D$.pop()
      else
        $done \leftarrow$ true
      if $D$.isEmpty() then
        $h \leftarrow -1$
      else
        $h \leftarrow D$.top()
      $S[i] \leftarrow i - h$
      $D$.push($i$)
    return $S$
```

- Let's analyize computeSpan2's run time...

# Java Stuff

- Given the stack ADT, we need to code the ADT in order to use it in the programs.
- You need to understand two program constructs: **interfaces** and **exceptions**.
- An interface is a way to declare what a class is to do. It does not mention how to do it.
- For an interface, you just write down the method names and the parameters. When specifying parameters, what really matters is their **types**.
- Later, when you write a class for that interface, you actually code the content of the methods.
- Separating interface and implementation is a useful programming technique.
- Interface example:

```
public interface radio {
   public void play();
   public void stop();
}
```

# A Stack Interface in Java

- While, the stack data structure is a “built-in” class of Java’s java.util package, we define our own stack interface:

```
public interface Stack {

// accessor methods
   public int size();
   public boolean isEmpty();
   public Object top() throws StackEmptyException;

// update methods
   public void push (Object element);
   public Object pop() throws StackEmptyException;
}
```

## Exceptions

- **Exceptions** are yet another useful programming construct.
- This is useful for handling errors. When you find an error (or an ***exceptiona*l** case), you just ***throw*** an exception.
- Example

```
public void eatPizza() throws StomachAcheException
{
   ...

   if (ateTooMuch)
      throw new StomachAcheException("Ouch");

   ...
}
```

- As soon as the exception is thrown, the flow of control exits from the current method.
- So when StomachAcheException is thrown, we exit from method eatPizza() and go to where that method was called from.

## More Exceptions

- Say the following code fragment called the method eatPizza() in the first place.

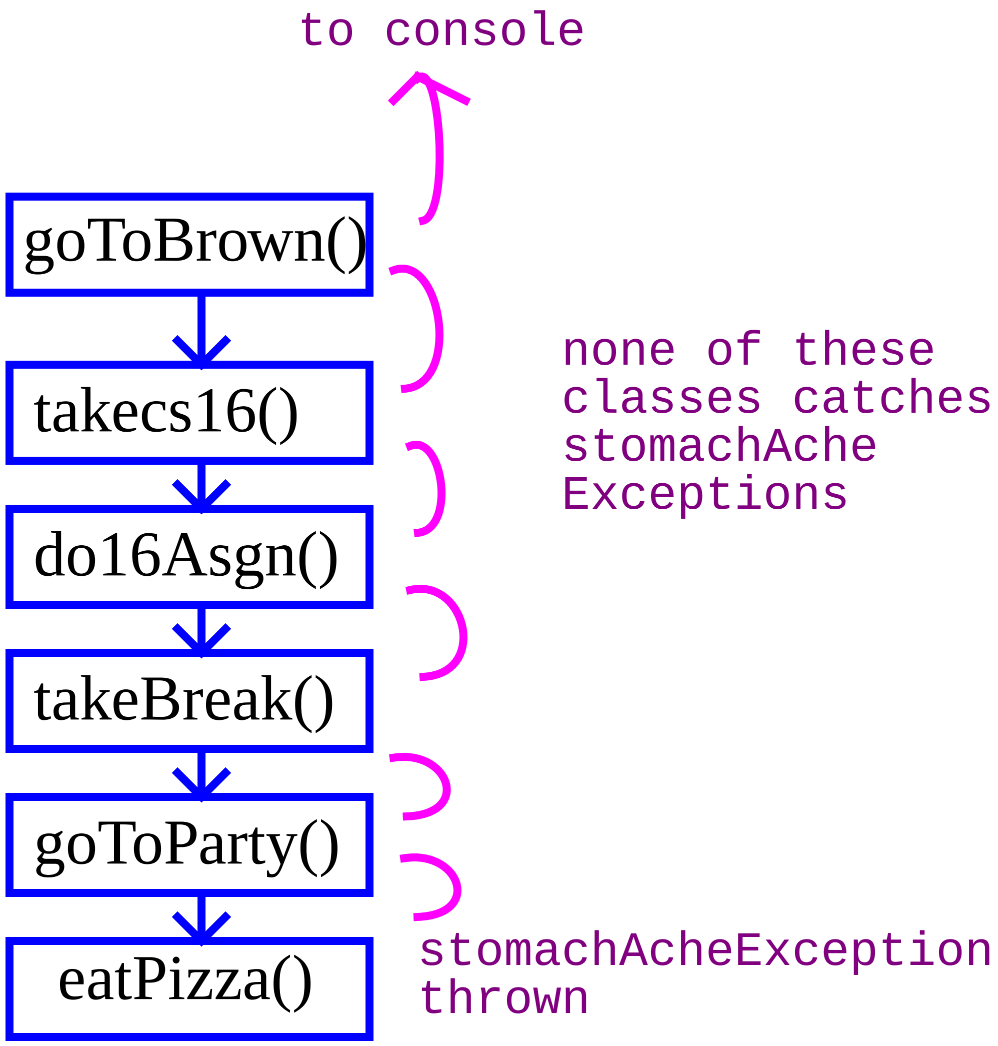
```
private void simulateMeeting()
{
   ...
   try
      {
         aStupidTA.eatPizza();
      }
   catch(StomachAcheException e)
      {
         System.out.println("somebody has a stomach
         ache");
      }
   ...
}
```

## Even More Exceptions

- We will get back to aStupidTA.eatPizza(); because, remember, eatPizza() threw an exception.
- The try block and the catch block means that we are listening for exceptions that are specified in the catch parameter.
- Because catch is listening for StomachAcheException, the flow of control will now go to the catch block. And System.out.println will get executed.
- Note that a catch block can contain anything. It does not have to do only System.out.println. You can handle the caught error in any way you like; you can even throw them again.
- Note that if somewhere in your method, you throw an exception, you need to add a throws clause next to your method name.
- What is the point of using exceptions? You can delegate upwards the responsibility of handling an error. Delegating upwards means letting the code who called the current code deal with the problem.

## Even More Exceptions

- If you never catch an exception, it will propagate upwards and upwards along the chain of method calls until the user sees it.


to console
goToBrown()
takecs16()
do16Asgn()
takeBreak()
goToParty()
eatPizza()
none of these classes catches stomachAche Exceptions
stomachAcheException thrown

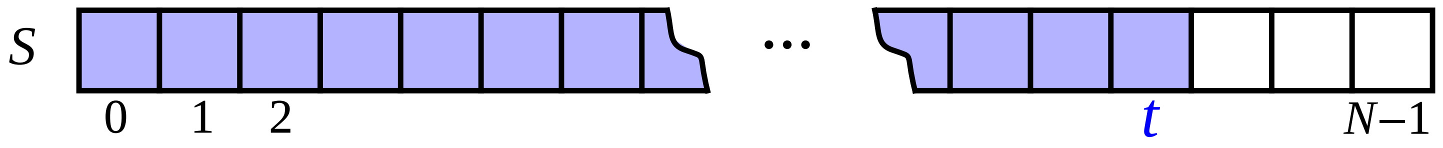
## Final Exceptions

- OK, so we threw and caught exceptions. But what exactly are they in Java? Classes.
- Check out the StomachAcheException.

```
public class StomachAcheException extends
   RuntimeException {
   public StomachAcheException(String err)
      {
         super(err);
      }
}
```

# An Array-Based Stack

- Create a stack using an array by specifying a maximum size *N* for our stack, e.g., *N* = 1,024.
- The stack consists of an *N*-element array *S* and an integer variable *t*, the index of the top element in array *S*.



- Array indices start at 0, so we initialize *t* to -1
- Pseudo-code

```
Algorithm size():
  return $t$ +1

Algorithm isEmpty():
  return ($t < 0$)

Algorithm top():
  if isEmpty() then
    throw a StackEmptyException
  return $S[t]$
…
```

# An Array-Based Stack (contd.)

- Pseudo-Code (contd.)

```
Algorithm push(o):
  if size() = N then
    throw a StackFullException
  t ← t + 1
  S[t] ← o

Algorithm pop():
  if isEmpty() then
    throw a StackEmptyException
  e←S[t]
  S[t]←null
  t←t-1
  return e
```

- Each of the above method runs in $O(1)$ time
- The array implementation is simple and efficient.
- There is a predefined upper bound, $N$, on the size of the stack, which may be too small for a given application, or cause a waste of memory.
- StackEmptyException is required by the interface.
- StackFullException is particular to this implementation.

## Array-Based Stack in Java

```
public class ArrayStack implements Stack {
  // Implementation of the Stack interface
  // using an array.


  public static final int CAPACITY = 1024; // default
                             // capacity of the stack
  private int capacity; // maximum capacity of the
                             // stack.
  private Object S[ ]; // S holds the elements of
                             // the stack
  private int top = –1; // the top element of the
                             // stack.


  public ArrayStack( ) { // Initialize the stack
      this(CAPACITY);// with default capacity
  }


  public ArrayStack(int cap) { // Initialize the
                         // stack with given capacity
      capacity = cap;
      S = new Object[capacity];
  }
```

# Array-Based Stack in Java (contd.)

```
public int size( ) { //Return the current stack size
        return (top + 1);
}
public boolean isEmpty( ) {  // Return true iff
                             // the stack is empty
        return (top < 0);
}
public void push(Object obj)
        throws StackFullException{ // Push a new
                          // element on the stack
        if (size() == capacity) {
           throw new StackFullException("Stack overflow.");
        }
        S[++top] = obj;
}
public Object top( )      // Return the top stack
                          // element
        throws StackEmptyException {
        if (isEmpty( )) {
           throw new StackEmptyException("Stack is
              empty.");
           }
        return S[top];
}
```

# Array-Based Stack in Java (contd.)

```
public Object pop() // Pop off the stack element
      throws StackEmptyException {
    Object elem;
    if (isEmpty( )) {
      throw new StackEmptyException(“Stack is Empty.”);
    elem = S[top];
    S[top––] = null; // Dereference S[top] and
                     // decrement top
    return elem;
    }
}
```