CS 112 – Introduction to Computing II

Wayne Snyder
Computer Science Department
Boston University

Today

- Stacks and Queues;
- Priority Queues;
- Queues implemented by Circular (or Ring) Buffers; [Reading: Wiki "Circular Buffers"]
- Deques
- Exceptions

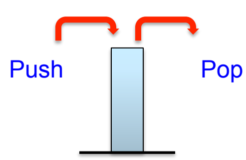
Computer Science

Stack ADT

Computer Science

A **Stack** for integers could be defined by the following interface of public methods:

```
// Stack Interface

  void push(int key);          // push the key onto the top of the stack
  int pop();                   // remove the top key and return it
  int top();                   // examine top element and return it without
                               // removing it from stack
  boolean isEmpty();
  int size();                 // how many integers in the stack
```

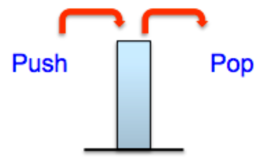


2

Queue ADT

Boston University Computer Science

The **Queue ADT** is a simple variant of a stack which makes a simple change which in fact changes everything: instead of moving items in and out of the same "end" of the list, as in a stack:



Instead you use different ends of the list:

3

Queue ADT

Boston University Computer Science

This means that instead of reversing the order of the items, as with a stack, they remain in the same order; since you have stood in lines many times at Starbucks (or outside my office!), I'll only give a brief example:

4

Queue ADT

This means that instead of reversing the order of the items, as with a stack, they remain in the same order; since you have stood in lines many times at Starbucks (or outside my office), I'll only give a brief example:

enqueue(5);

Queue ADT

This means that instead of reversing the order of the items, as with a stack, they remain in the same order; since you have stood in lines many times at Starbucks (or outside my office), I'll only give a brief example:

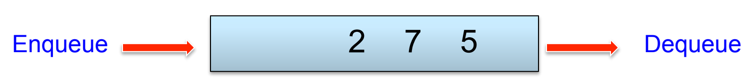
enqueue(5);
enqueue(7);

Queue ADT

Computer Science

This means that instead of reversing the order of the items, as with a stack, they remain in the same order; since you have stood in lines many times at Starbucks (or outside my office), I'll only give a brief example:

```
enqueue(5);
enqueue(7);
enqueue(2);
```



7

Queue ADT

Computer Science

This means that instead of reversing the order of the items, as with a stack, they remain in the same order; since you have stood in lines many times at Starbucks (or outside my office), I'll only give a brief example:

```
enqueue(5);
enqueue(7);
enqueue(2);
int k = dequeue();
```

k = 5

8

Queue ADT

Computer Science

This means that instead of reversing the order of the items, as with a stack, they remain in the same order; since you have stood in lines many times at Starbucks (or outside my office), I'll only give a brief example:

```
enqueue(5);
enqueue(7);
enqueue(2);
int k = dequeue();
enqueue(8);
```

Enqueue → | 8 2 7 | → Dequeue

k = 5

9

Queue ADT

Computer Science

This means that instead of reversing the order of the items, as with a stack, they remain in the same order; since you have stood in lines many times at Starbucks (or outside my office), I'll only give a brief example:

```
enqueue(5);
enqueue(7);
enqueue(2);
int k = dequeue();
enqueue(8);
enqueue( dequeue() )
```

Enqueue → | 7 8 2 | → Dequeue

k = 5

10

Queue ADT

Computer Science

Queues occur all the time, in real life:

And in computer systems:

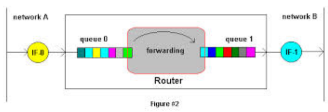


Figure #2

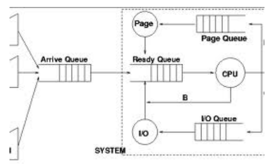


Figure 1: Simple Operating System

In fact, anywhere where one service is desired by many, and must be fairly distributed... there is a whole branch of math called "queueing theory" which you will learn about in CS 237 and CS 350.....

11

Array-based Implementation of Queues

Computer Science

A **Queue** for integers could be defined by the following interface:

```
void enqueue(int key);    // insert the key at the end of the queue
int dequeue();            // remove the key at front of the queue
boolean isEmpty();
int size();               // returns number of integers in queue
```

Enqueue

Dequeue

How to implement this with arrays?

12

Array-based Implementation of Integer Queues

Computer Science

To implement an array-based queue for ints, here is the **first thing** you might think of.....

	9	8	7	6	5	4	3	2	1	0
A:								2	9	5
							↑ next			↑ front

```
void enqueue(int k) {
   A[next] = k;
   ++next;
}

int size() {
   return (next – front);
}
```

```
int dequeue() {
   int temp = A[front];
   ++front;
   return temp;
}

boolean isEmpty() {
   return (size() == 0);
}
```

13

Array-based Implementation of Integer Queues

Computer Science

But there is an obvious problem, and not so trivial..... **running off the end** of the array!

	9	8	7	6	5	4	3	2	1	0
A:		0	1	-3	7	91	25	2	9	5
	↑ next			↑ front						

```
void enqueue(int k) {
  if(size() != A.length) {
    A[next] = k;
    ++next;
  }
}
int size() {
   return (next – front);
}
```

```
int dequeue() {
   int temp = A[front];
   ++front;
   return temp;
}

Boolean isEmpty() {
   return (size() == 0);
}
```

14

Array-based Implementation of Integer Queues

What solutions could we come up with for this problem?

Well, there are several:

Bad: Reallocate a bigger array so you don't run off the end (we'll talk about resizing arrays next week). But then your array grows and grows and grows!

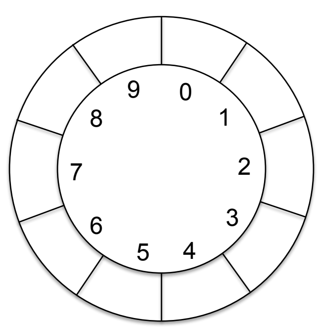
Good: Each time you dequeue, shift all the data over (similarly with how a queue is managed in Starbucks: when the person at the head of the line leaves, everyone moves up!). A natural solution, but if the queue is very large, each dequeue takes a long time, since you have to touch every data item and move it. Enqueue takes $\Theta(1)$ but every dequeue takes $\Theta(N)$ time.

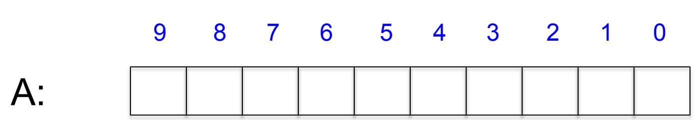
Best: Consider the array to be in a circle, with each end "glued" together, so that you never run off the array..... This will be $\Theta(1)$ for all operations!

15

Array-based Implementation of Queues

In the ring or circular buffer approach, when we reach the end of the array we wrap around to the beginning:





16

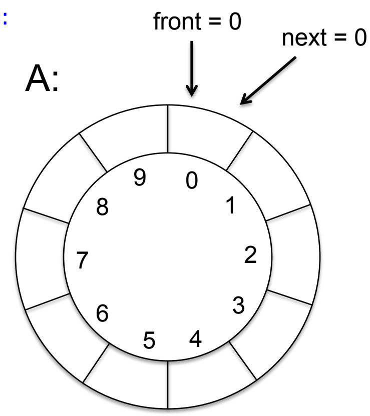
Array-based Implementation of Queues

Computer Science

In the ring or circular buffer approach, when we reach the end of the array we wrap around to the beginning:

```
int size = 0;
int front = 0;
int next = 0;
```

A:

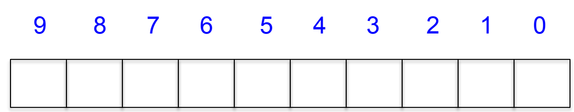


In the **fill count** version of circular buffer, we keep track of the number of elements:

size = 0

How do we move the pointers **front** and **next** around the ring?

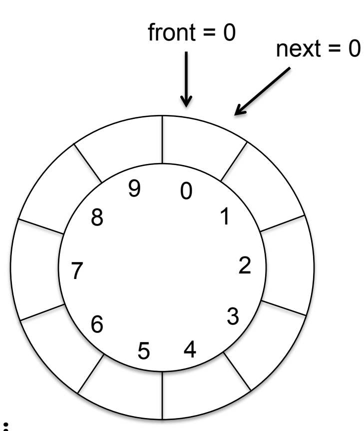
A:



17

Array-based Implementation of Queues

Computer Science



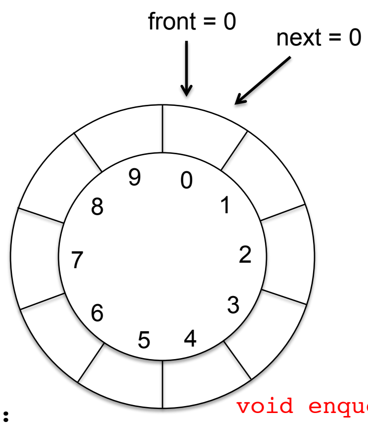

```
int size = 0;
int front = 0;
int next = 0;

// To move a pointer:

int nextSlot(int k) {
    return ((k + 1) % A.length);
}
```

```
next = nextSlot(next);
```

18

Array-based Implementation of Queues

Computer Science




```
int size = 0;
int front = 0;
int next = 0;

// To move a pointer:

int nextSlot(int k) {
    return ((k + 1) % A.length);
}
```

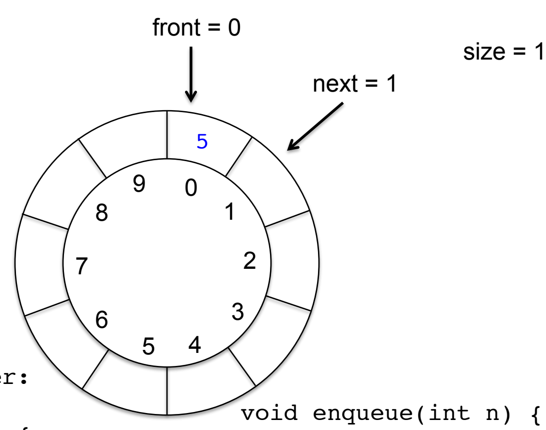
```
void enqueue(int n) {
    A[next] = n;
    next = nextSlot(next);
    ++size;
}
```

19

Array-based Implementation of Queues

Computer Science

```
enqueue(5);
```




```
// To move a pointer:

int nextSlot(int k) {
    return ((k + 1) % A.length);
}
```

```
void enqueue(int n) {
    A[next] = n;
    next = nextSlot(next);
    ++size;
}
```

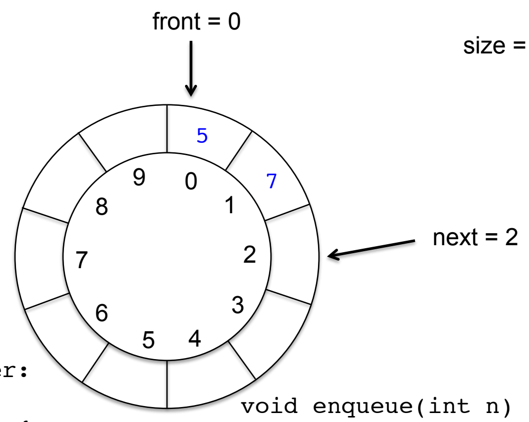
20

Array-based Implementation of Queues

Computer Science

size = 2

```
enqueue(5);
enqueue(7);
```




```
// To move a pointer:

int nextSlot(int k) {
    return ((k + 1) % A.length);
}
```

```
void enqueue(int n) {
    A[next] = n;
    next = nextSlot(next);
    ++size;
}
```

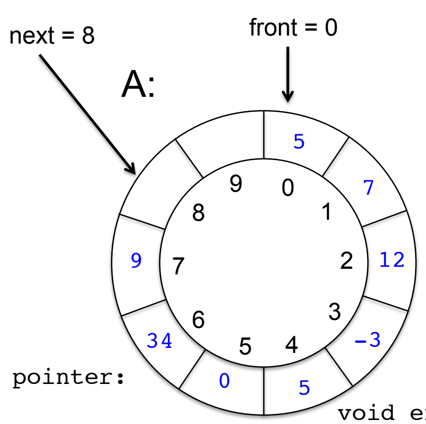
21

Array-based Implementation of Queues

Computer Science

size = 8

```
enqueue(5);
enqueue(7);
enqueue(12);
enqueue(-3);
enqueue(5);
enqueue(0);
enqueue(34);
enqueue(9);
```



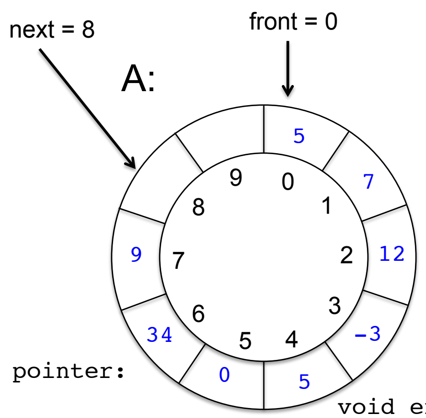

```
// To move a pointer:

int nextSlot(int k) {
    return ((k + 1) % A.length);
}
```

```
void enqueue(int n) {
    A[next] = n;
    next = nextSlot(next);
    ++size;
}
```

22

Array-based Implementation of Queues

Computer Science



size = 8

```
int dequeue() {
  int temp = A[front];
  front = nextSlot(front);
  --size;
  return temp;
}
```

```
// To move a pointer:

int nextSlot(int k) {
    return ((k + 1) % A.length);
}
```

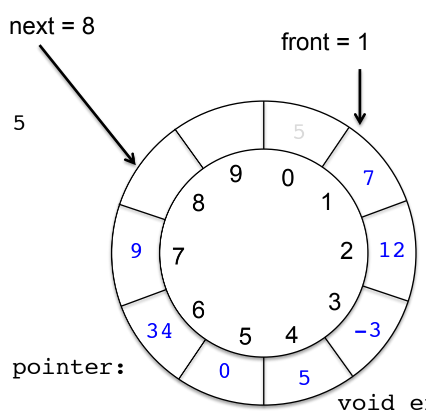
```
void enqueue(int n) {
    A[next] = n;
    next = nextSlot(next);
    ++size;
}
```

23

Array-based Implementation of Queues

Computer Science

```
dequeue() => 5
```



size = 7

```
int dequeue() {
  int temp = A[front];
  front = nextSlot(front);
  --size;
  return temp;
}
```

```
// To move a pointer:

int nextSlot(int k) {
    return ((k + 1) % A.length);
}
```

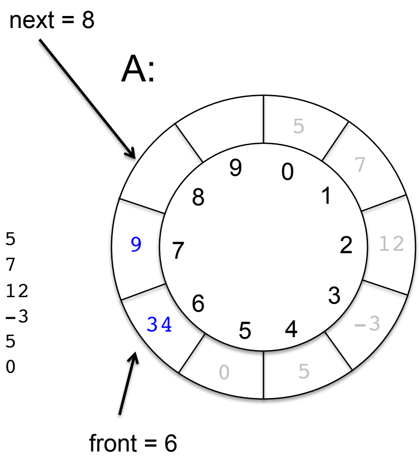
```
void enqueue(int n) {
    A[next] = n;
    next = nextSlot(next);
    ++size;
}
```

24

Array-based Implementation of Queues

Computer Science

size = 2




```
enqueue(5);
enqueue(7);
enqueue(12);
enqueue(-3);
enqueue(5);
enqueue(0);
enqueue(34);
enqueue(9);

dequeue(); => 5
dequeue(); => 7
dequeue(); => 12
dequeue(); => -3
dequeue(); => 5
dequeue(); => 0
```

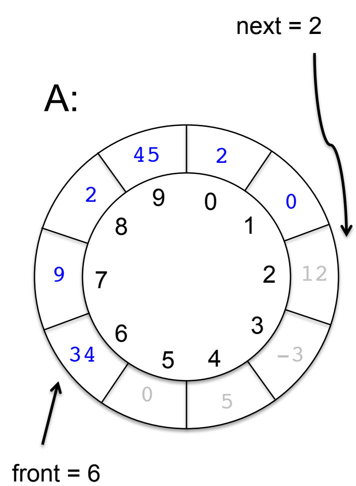
```
int [] A = new int[10];
int size = 0;
int front = 0; int next = 0;

int nextSlot(int k) {
    return ((k + 1) % A.length);
}

void enqueue(int n) {
    A[next] = n;
    next = nextSlot(next);
    ++size;
}

int size() {
   return size;
}

// can still underflow!
int dequeue() {
   int temp = A[front];
   front = nextSlot(front);
   --size;
   return temp;
}

boolean isEmpty() {
   return (size == 0);
}
```

25

Array-based Implementation of Queues

Computer Science

size = 6




```
enqueue(5);
enqueue(7);
enqueue(12);
enqueue(-3);
enqueue(5);
enqueue(0);
enqueue(34);
enqueue(9);
dequeue(); => 5
dequeue(); => 7
dequeue(); => 12
dequeue(); => -3
dequeue(); => 5
dequeue(); => 0

enqueue(2);
enqueue(45);
enqueue(2);
enqueue(0);

Etc....
```

```
int [] A = new int[10];
int size = 0;
int front = 0; int next = 0;

int nextSlot(int k) {
    return ((k + 1) % A.length);
}

void enqueue(int n) {
      A[next] = n;
      next = nextSlot(next);
      ++size;
}

int size() {
   return size;
}

// can still underflow!
int dequeue() {
   int temp = A[front];
   front = nextSlot(front);
   --size;
   return temp;
}

boolean isEmpty() {
   return (size == 0);
}
```

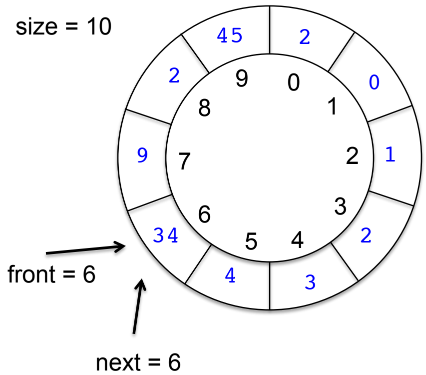
26

Array-based Implementation of Queues

Computer Science

Note: Can't distinguish full or empty from the pointers alone, that is why we keep track of the size!

```
enqueue(5);
enqueue(7);
enqueue(12);
enqueue(-3);
enqueue(5);
enqueue(0);
enqueue(34);
enqueue(9);
dequeue(); => 5
dequeue(); => 7
dequeue(); => 12
dequeue(); => -3
dequeue(); => 5
dequeue(); => 0

enqueue(2);
enqueue(45);
enqueue(2);
enqueue(0);
enqueue(1);
enqueue(2);
enqueue(3);
enqueue(4);
```



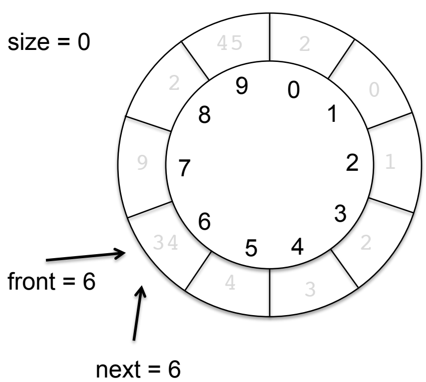

```
int [] A = new int[10];
int size = 0;
int front = 0; int next = 0;

int nextSlot(int k) {
    return ((k + 1) % A.length);
}

void enqueue(int n) {
     A[next] = n;
     next = nextSlot(next);
     ++size;
}

int size() {
   return size;
}

// can still underflow!
int dequeue() {
   int temp = A[front];
   front = nextSlot(front);
   --size;
   return temp;
}

boolean isEmpty() {
   return (size == 0);
}
```


Array-based Implementation of Queues

Computer Science

Note: Can't distinguish full or empty from the pointers alone, that is why we keep track of the size!

```
enqueue(5);
enqueue(7);
enqueue(12);
enqueue(-3);
enqueue(5);
enqueue(0);
enqueue(34);
enqueue(9);
dequeue(); => 5
dequeue(); => 7
dequeue(); => 12
dequeue(); => -3
dequeue(); => 5
dequeue(); => 0

enqueue(2);
enqueue(45);
enqueue(2);
enqueue(0);
enqueue(1);
enqueue(2);
enqueue(3);
enqueue(4);
```

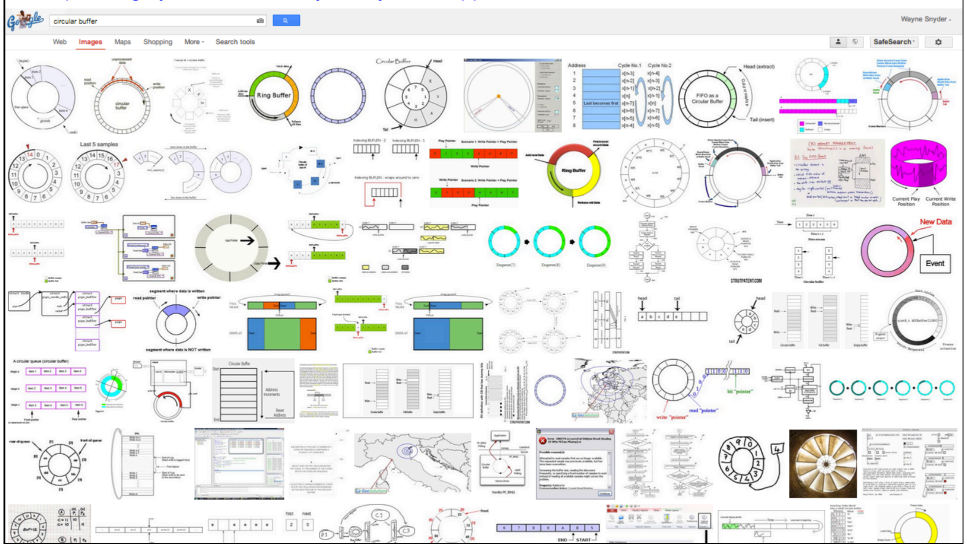



```
int [] A = new int[10];
int size = 0;
int front = 0; int next = 0;

int nextSlot(int k) {
    return ((k + 1) % A.length);
}

void enqueue(int n) {
     A[next] = n;
     next = nextSlot(next);
     ++size;
}

int size() {
   return size;
}

// can still underflow!
int dequeue() {
   int temp = A[front];
   front = nextSlot(front);
   --size;
   return temp;
}

boolean isEmpty() {
   return (size == 0);
}
```

Array-based Implementation of Queues

Computer Science

Circular or ring buffers are the standard technique for implementing queues and buffers in operating systems and many, many other applications!



Queue ADT: Two Important Variations

Computer Science

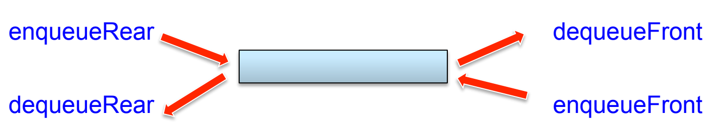
The **Deque ("deck") ADT** is a "double-ended queue" in which you can insert or remove from either end; it is either a queue going in both directions, or two stacks stuck together:

enqueueRear(k): Insert the key k in the rear

dequeueRear(): Remove and return the item from the rear of the list

enqueueFront(k): Insert the key k in the front

dequeueFront(): Remove and return the item from the front of the list



30

Queue ADT: Two Important Variations

Computer Science

The **Priority Queue ADT** is a queue in which the list is always kept ordered; this is useful when elements in the queue have a different need or right for service; the only change is in the enqueue method (and the names change):

There are two flavors: A **MaxQueue** or a **MinQueue**, depending on whether the element removed is the largest or smallest element:

put(k): Insert the key k into the priority queue

getMax(): Remove and return the largest key in the queue

[Or: getMin(): Remove and return the smallest key in the queue.]

put → [ ] → getMax/ getMin

31

Priority Queue ADT

Computer Science

put(5);

put → [5] → getMax

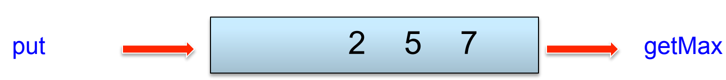
32

Priority Queue ADT

put(5);
put(7);

Priority Queue ADT

put(5);
put(7);
put(2);



Priority Queue ADT

Computer Science

```
put(5);
put(7);
put(2);
int k = geMax();
```

put → | 2 5 | → getMax

k = 7

35

Priority Queue ADT

Computer Science

```
put(5);
put(7);
put(2);
int k = getMax();
put(8);
```

put → | 2 5 8 | → geMax

k = 7

36

Priority Queue ADT

Boston University Computer Science

put(5);

put(7);

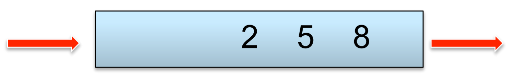
put(2);

int k = getMax();

put(8);

put(getMax())

put



getMax

k = 7

37

Exceptions for Error Handling in Java

Boston University Computer Science

To this point, we have not dealt with how to report and recover from errors in Java; for example, with `IntStack.java`:

```
public class IntStack  {

    private int[] A = new int[20];
    private int next = 0;

    public void push(int n) {
        A[next] = n;
        ++next;
    }

    public int pop() {
        --next;
        return A[next];
    }
}
```

IntStack S = new IntStack();

S.push(3);

3

38

Exceptions for Error Handling in Java

Computer Science

To this point, we have not dealt with how to report and recover from errors in Java; for example, with `IntStack.java`:

```
public class IntStack  {

    private int[] A = new int[20];
    private int next = 0;

    public void push(int n) {
        A[next] = n;
        ++next;
    }

    public int pop() {
        --next;
        return A[next];
    }
}
```

IntStack S = new IntStack();

S.push(3);

S.push(4);

4
3

39

Exceptions for Error Handling in Java

Computer Science

To this point, we have not dealt with how to report and recover from errors in Java; for example, with `IntStack.java`:

```
public class IntStack  {

    private int[] A = new int[20];
    private int next = 0;

    public void push(int n) {
        A[next] = n;
        ++next;
    }

    public int pop() {
        --next;
        return A[next];
    }
}
```

IntStack S = new IntStack();

S.push(3);

S.push(4);

System.out.println(S.pop());

4

3

40

Exceptions for Error Handling in Java

Computer Science

To this point, we have not dealt with how to report and recover from errors in Java; for example, with `IntStack.java`:

```
public class IntStack  {

    private int[] A = new int[20];
    private int next = 0;

    public void push(int n) {
        A[next] = n;
        ++next;
    }

    public int pop() {
        --next;
        return A[next];
    }
}
```

IntStack S = new IntStack();

S.push(3);

S.push(4);

System.out.println(S.pop());

System.out.println(S.pop());

4
3

41

Exceptions for Error Handling in Java

Computer Science

To this point, we have not dealt with how to report and recover from errors in Java; for example, with `IntStack.java`:

```
public class IntStack  {

    private int[] A = new int[20];
    private int next = 0;

    public void push(int n) {
        A[next] = n;
        ++next;
    }

    public int pop() {
        --next;
        return A[next];
    }
}
```

IntStack S = new IntStack();

S.push(3);

S.push(4);

System.out.println(S.pop());

System.out.println(S.pop());

System.out.println(S.pop());

```
java.lang.ArrayIndexOutOfBoundsException: -1
        at IntStack.pop(IntStack.java:61)
        at IntStack.main(IntStack.java:143)
        at sun.reflect.NativeMethodAccessorImpl.in
        at sun.reflect.NativeMethodAccessorImpl.in
        at sun.reflect.DelegatingMethodAccessorImp
```

This is called Stack Underflow.

42

Exceptions for Error Handling in Java

Computer Science

To this point, we have not dealt with how to report and recover from errors in Java; for example, with `IntStack.java`:

```
public class IntStack  {

    private int[] A = new int[20];
    private int next = 0;

    public void push(int n) {
        A[next] = n;
        ++next;
    }

    public int pop() {
        --next;
        return A[next];
    }
}
```

IntStack S = new IntStack();

for(int i = 1; i <= 20; ++i)
    S.push(i);

20
19
.
.
.
3
2
1

43

Exceptions for Error Handling in Java

Computer Science

To this point, we have not dealt with how to report and recover from errors in Java; for example, with `IntStack.java`:

```
public class IntStack  {

    private int[] A = new int[20];
    private int next = 0;

    public void push(int n) {
        A[next] = n;
        ++next;
    }

    public int pop() {
        --next;
        return A[next];
    }
}
```

This is called Stack Overflow.

IntStack S = new IntStack();

for(int i = 1; i <= 20; ++i)
    S.push(i);

S.push(21)

20
19
.
.

```
java.lang.ArrayIndexOutOfBoundsException: 20
        at IntStack.push(IntStack.java:52)
        at IntStack.main(IntStack.java:133)
        at sun.reflect.NativeMethodAccessorImpl.inv
        at sun.reflect.NativeMethodAccessorImpl.inv
        at sun.reflect.DelegatingMethodAccessorImpl
        at java.lang.reflect.Method.invoke(Method.i
```

44

Exceptions for Error Handling in Java

Computer Science

Communicating errors and recovering from them is a big problem, and it is solved in Java by the mechanism of Exceptions. You have seen these already:

```
java.lang.ArrayIndexOutOfBoundsException: 20
        at IntStack.push(IntStack.java:52)
        at IntStack.main(IntStack.java:133)
        at sun.reflect.NativeMethodAccessorImpl.invoke0(Native Me
        at sun.reflect.NativeMethodAccessorImpl.invoke(NativeMeth
        at sun.reflect.DelegatingMethodAccessorImpl.invoke(Delega
        at java.lang.reflect.Method.invoke(Method.java:597)
```

When a piece of code encounters a serious error, it **throws** an **exception**, which is an instance of a class that reports the error and terminates execution of that piece of code. By **catching** an exception, we can handle it and prevent the program itself from terminating.

45

Exceptions for Error Handling in Java

Computer Science

Exceptions are an essential way to deal with errors in Java, most commonly, you only have to deal with the simple case of an ADT throwing some exception that must be caught by the client. You have to remember a couple of things:

```
public class IntStack  {

    private int[] A = new int[20];
    private int next = 0;

    public void push(int n) {
        A[next] = n;
        ++next;
    }

    public int pop() {
        --next;
        return A[next];
    }
}

class StackUnderflowException extends Exception {
      // could have members but usually not
}
```

1. An exception is an **instance of a class**, and can contain members; usually, the exception contains nothing, and the name itself is important.

46

Exceptions for Error Handling in Java

Computer Science

Exceptions are an essential way to deal with errors in Java, most commonly, you only have to deal with the simple case of an ADT throwing some exception that must be caught by the client. You have to remember a couple of things:

1. An exception is an **instance of a class**, and can contain members; usually, the exception contains nothing, and the name itself is important.
2. You **throw** an exception when you encounter the condition/error by calling the constructor for the exception in a **throw** statement.

```
public class IntStack  {

    private int[] A = new int[20];
    private int next = 0;

    public void push(int n) {
        A[next] = n;
        ++next;
    }

    public int pop() {
        if( next == 0 )
          throw new StackUnderflowException();     // default constructor for class
        --next;
        return A[next];
    }
}

class StackUnderflowException extends Exception {
      // could have members but usually not
}
```

47

Exceptions for Error Handling in Java

Computer Science

Exceptions are an essential way to deal with errors in Java, most commonly, you only have to deal with the simple case of an ADT throwing some exception that must be caught by the client. You have to remember a couple of things:

1. An exception is an **instance of a class**, and can contain members; usually, the exception contains nothing, and the name itself is important.
2. You **throw** an exception when you encounter the condition/error by calling the constructor for the exception in a **throw** statement.
3. Any call to that method must be inside a try-catch block which catches that exception (or a superclass).

```
public class IntStack  {

    private int[] A = new int[20];
    private int next = 0;

    public void push(int n) {
        A[next] = n;
        ++next;
    }

    public int pop() {
        if( next == 0 )
          throw new StackUnderflowException();
        --next;
        return A[next];
    }
}

class StackUnderflowException extends Exception {
      // could have members but usually not
}
```

```
try {
    System.out.println( S.pop() );
}
catch (StackUnderflowException e) {
    System.out.println("Q underflew!");
}
```

48

Exceptions for Error Handling in Java

Computer Science

Exceptions are an essential way to deal with errors in Java, most commonly, you only have to deal with the simple case of an ADT throwing some exception that must be caught by the client. You have to remember a couple of things:

1. An exception is an **instance of a class**, and can contain members; usually, the exception contains nothing, and the name itself is important.
2. You **throw** an exception when you encounter the condition/error by calling the constructor for the exception in a **throw** statement.
3. Any call to that method must be inside a try-catch block which catches that exception (or a superclass).
4. The header of the **method** must list all exceptions that it throws.

```
public class IntStack  {

    private int[] A = new int[20];
    private int next = 0;

    public void push(int n) {
        A[next] = n;
        ++next;
    }

    public int pop() throws StackUnderflowException {
        if( next == 0 )
          throw new StackUnderflowException();
        --next;
        return A[next];
    }
}

class StackUnderflowException extends Exception {
      // could have members but usually not
}
```

```
try {
    System.out.println( S.pop() );
}
catch (StackUnderflowException e) {
    System.out.println("Q underflew!");
}
```

49

Exceptions for Error Handling in Java

Computer Science

1. An exception is an **instance of a class**, and can contain members; usually, the exception contains nothing, and the name itself is important.
2. You **throw** an exception when you encounter the condition/error by calling the constructor for the exception in a **throw** statement.
3. Any call to that method must be inside a try-catch block which catches that exception (or a superclass).
4. The header of the **method** must list all exceptions that it throws.

```
public class IntStack  {

    private int[] A = new int[20];
    private int next = 0;

    public void push(int n) throws StackOverflowException {
        if( next == 20)
           throw new StackOverflowException();
        A[next] = n;
        ++next;
    }

    public int pop() throws StackUnderflowException {
        if( next == 0 )
           throw new StackUnderflowException();
        --next;
        return A[next];
    }
}

class StackUnderflowException extends Exception {}

class StackOverflowException extends Exception {}
```

```
try {
    S.push( S.pop() );
}
catch (StackUnderflowException e) {
    System.out.println("Q underflew!");
}
catch (StackOverflowException e) {
    System.out.println("Q overflew!");
}
```

50