Data Structures and Object-Oriented Design IV

Spring 2014

Carola Wenk

Array-Based Stack vs. DynamicStack

```
public class ArrayStack {
  final static int DEFAULT_CAPACITY=50;
  private int[] S;
  private int top;

  public ArrayStack(){
    this(DEFAULT_CAPACITY);
  }

  public ArrayStack(int capacity){
    S = new int[capacity];
    top=-1;
  }

  public void push(int x){
    //Needs to throw exception when stack is full
    S[++top]=x;
  }

  public int pop(){
    if(top>=0)
      return S[top--];
    else
      throw new RuntimeException("Stack is empty.");
  }
}
```

```
public class Tester{
  public static void main(String[] args) {
    ArrayStack stack = new ArrayStack();
    stack.push(5);
    System.out.println("popped: "+stack.pop());
  }
}
```

```
public class DynamicStack implements Stack{

  private class StackNode {
    private int data;
    private StackNode next;

    public StackNode(int d){
      data = d;
      next=null;
    }
  }

  private StackNode top = null;

  public void push(int x) {
    StackNode temp = new StackNode(x);
    temp.next = top;
    top = temp;
  }

  public int pop(){
    if (top == null)
      throw new RuntimeException("Stack empty!");

    int x = top.data;
    top = top.next;
    return x;
  }
}
```

If we change `ArrayStack` to `DynamicStack`, the code still works.

Java Interfaces

- We can specify that a Java class implements a particular kind of functionality defined as an `interface`.

```java
public interface Stack {
  public int pop();
  public void push(int x);
}
```

```java
public class ArrayStack implements Stack {

  ...

}
```

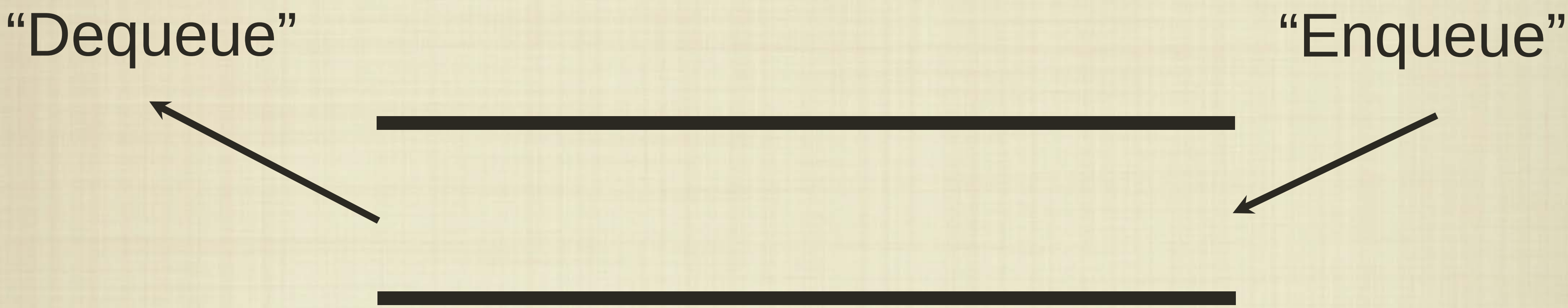
```java
public class DynamicStack implements Stack {

  ...

}
```

```java
public class Tester {
  public static void main(String[] args) {
    Stack stack = new ArrayStack();
    stack.push(5);
    System.out.println("popped: "+stack.pop());
  }
}
```

If we change `ArrayStack` to `DynamicStack`, the code still works.

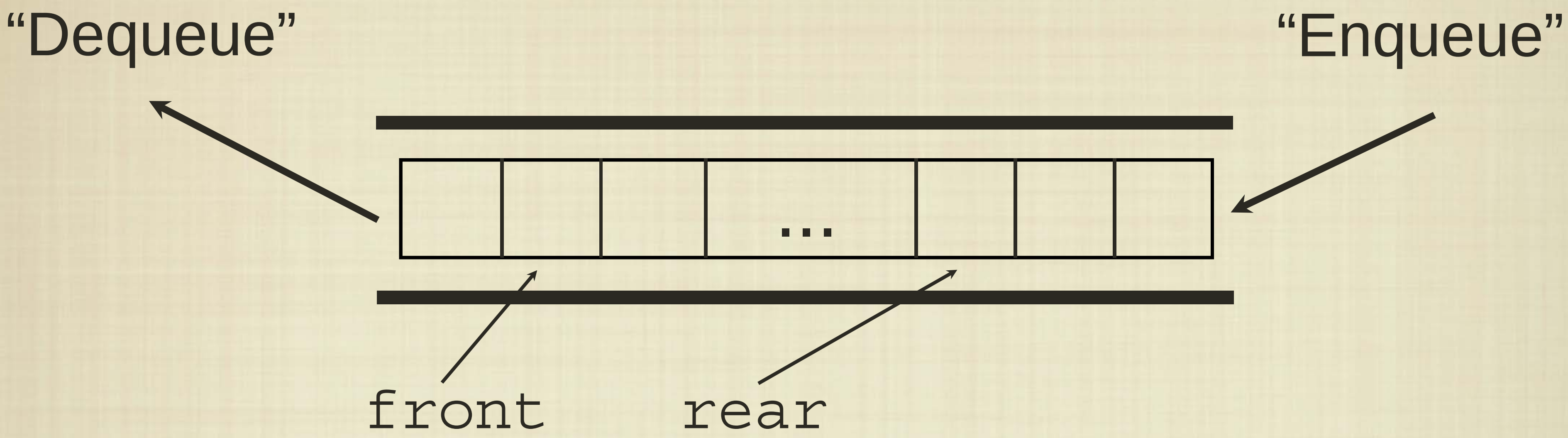
Queues

- A queue is a “first-in, first-out” data structure. How do we implement it?




```
class Queue {

public Queue() { ... }

public int dequeue() { ... }

public void enqueue(int x) { ... }

}
```

Static Implementation



- We need two indices to keep track of the front and rear of the queue. What do we know about the relationship between these two indices?

Queues

A queue can be implemented easily with an array. What limit does the size of the array impose on queue operations?

```
public class Queue {

  private int[] Q = null;
  private int front, rear;

  public Queue(int capacity) {
     Q = new int[capacity];
     front = 0; rear = -1;
  }

  public void enqueue(int x) {
     ...
  }

  public int dequeue() {
     ...
  }

  public int size() { ... }

}
```

Queues

A queue can be implemented easily with an array. What limit does the size of the array impose on queue operations?

```
public class Queue {

  private int[] Q = null;
  private int front, rear;

  public Queue(int capacity) {
     Q = new int[capacity];
     front = 0; rear = -1;
  }

  public void enqueue(int x) {
     Q[++rear] = x;
  }

  public int dequeue() {
     ...
  }

  public int size() { ... }

}
```

Queues

A queue can be implemented easily with an array. What limit does the size of the array impose on queue operations?

```
public class Queue {

  private int[] Q = null;
  private int front, rear;

  public Queue(int capacity) {
     Q = new int[capacity];
     front = 0; rear = -1;
  }

  public void enqueue(int x) {
     Q[++rear] = x;
  }

  public int dequeue() {
     return Q[front++];
  }

  public int size() { ... }

}
```

Queues

A queue can be implemented easily with an array. What limit does the size of the array impose on queue operations?

```
public class Queue {

  private int[] Q = null;
  private int front, rear;

  public Queue(int capacity) {
     Q = new int[capacity];
     front = 0; rear = -1;
  }

  public void enqueue(int x) {
     Q[++rear] = x;
  }

  public int dequeue() {
     return Q[front++];
  }

  public int size() { return rear-front+1; }

}
```

Queues

In what cases do the public methods cause runtime errors? In what cases do they simply return incorrect results?

For a queue of capacity n, how many enqueue operations will succeed before causing a runtime error? Is this reasonable?

```
public class Queue {

  private int[] Q = null;
  private int front, rear;

  public Queue(int capacity) {
     Q = new int[capacity];
     front = 0; rear = -1;
  }

  public void enqueue(int x) {
     Q[++rear] = x;
  }

  public int dequeue() {
     return Q[front++];
  }

  public int size() { return rear-front+1; }

}
```

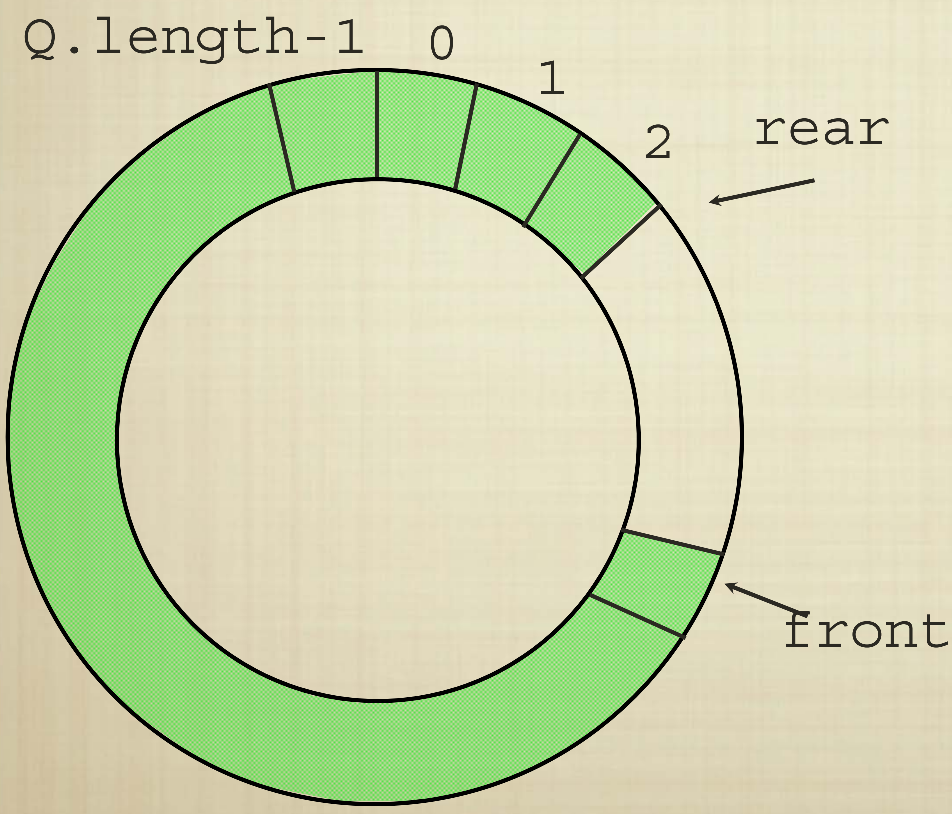
```
public class Queue {

  private int[] Q = null;
  private int front, rear;

  public Queue(int capacity) {
     Q = new int[capacity];
     front = 0; rear = -1;
  }

  public void enqueue(int x) {
     if (size() < Q.length)
        Q[++rear] = x;
     else throw new RuntimeException("Queue full!");
  }

  public int dequeue() {
     if (size() == 1) {
        int returnValue = Q[front];
        front = 0; rear = -1;
        return returnValue;
     } else if (size() > 0)
        return Q[front++];
     else // size() <= 0
       throw new RuntimeException("Queue empty!");
  }

  public int size() { return rear-front+1; }

}
```

What happens if we repeatedly enqueue/dequeue a single element?

Circular Queues

The capacity of the queue should allow a certain number of items that are in the queue at any one time - not the number of queue operations.

We can do this using modulo.

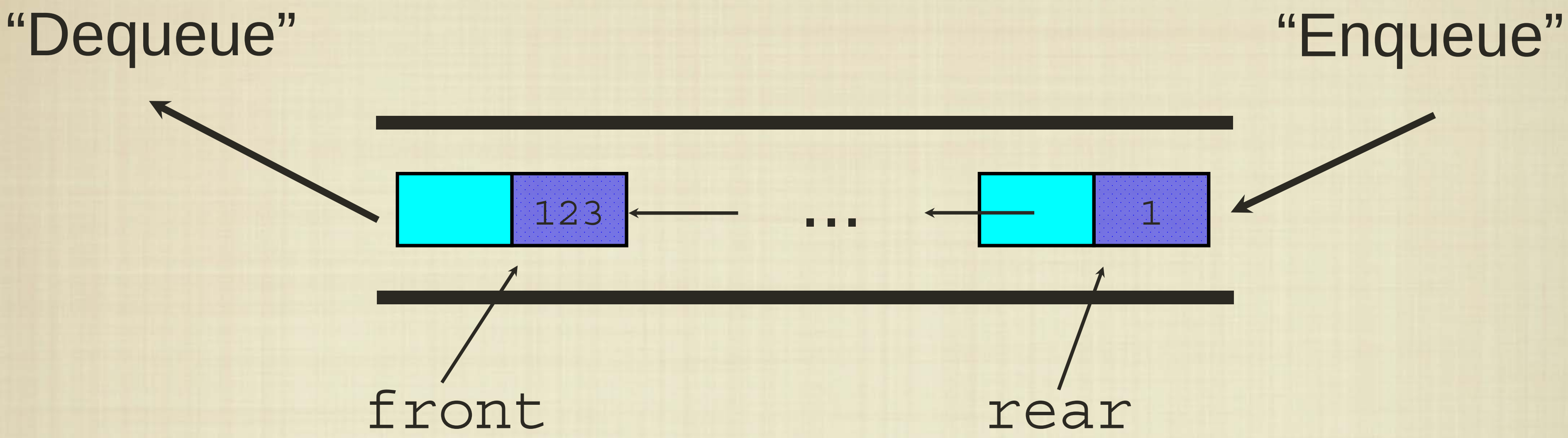



```
public class Queue {
  private int[] Q = null;
  private int front, rear, size;

  public Queue(int capacity) {
      Q = new int[capacity];
      front = 0; rear = -1; size = 0;
  }

  public void enqueue(int x) {
      if (size < Q.length) {
         rear = (rear + 1) % Q.length;
         Q[rear] = x;
         size ++;
      } else throw new RuntimeException("Queue full!");
  }

  public int dequeue() {
      if (size() > 0) {
         int returnValue = Q[front];
         front = (front + 1) % Q.length;
         size--;
         return returnValue;
      } else throw new RuntimeException("Queue empty!");
  }

  public int size() { return size; }

}
```

Instead of `Q[++rear]=x`

Instead of `return Q[front++]`

Dynamic Implementation



- We can use a linked structure to maintain the Queue elements, with a reference to the front and rear of the structure.

What correctness properties do we want from this structure?
Which special cases do we need to handle?

Dynamic Queue

```java
public class DynamicQueue {











  public DynamicQueue(){


  }

  public void enqueue(int x) {









  }
```

```java
  public int dequeue() {








  }

  public int size() {

  }
}
```

Dynamic Queue

```java
public class DynamicQueue {

  private class QueueNode {
    public int data;
    public QueueNode next;
    public QueueNode(int d) {
      data = d;
      next = null;

    }
  }

  private QueueNode front, rear;
  private int size;

  public DynamicQueue(){
    front=rear=null;
    size=0;
  }

  public void enqueue(int x) {
    if (size == 0) {
      front = new QueueNode(x);
      rear = front;
    } else {
      rear.next = new QueueNode(x);
      rear = rear.next;
    }
    size ++;
  }
```

```java
  public int dequeue() {
    if (front == null)
      throw new RuntimeException("Queue empty!");
    else{
      int returnValue = front.data;
      front = front.next;
      size--;
      return returnValue;
    }
  }

  public int size() {
    return size;
  }
}
```

What about Type Compatibility?

- So far, our class definitions have been defined to manipulate a single type (usually `int`).

- Do we really have to define a different class for a stack of strings? Can we define a general-purpose stack?

```
class intStack {

    private int[] S = null;
    private int top;

    public Stack(int capacity) {
     S = new int[capacity];
     top = capacity;
    }

    public int pop() {
     return S[top++];
    }

    public void push(int x) {
     S[--top] = x;
    }
```

```
class StringStack {

    private String[] S = null;
    private int top;

    public Stack(int capacity) {
     S = new String[capacity];
     top = capacity;
    }

    public String pop() {
     return S[top++];
    }

    public void push(String x) {
     S[--top] = x;
    }
```

Using Inheritance

- Note that references are essentially “unidirectional.”

- How general-purpose can we make types using Java’s object model?

```
class Stack {

    private Object[] S = null;
    private int top;

    public Stack(int capacity) {
     S = new Object[capacity];
     top = capacity;
    }

    public Object pop() {
     return S[top++];
    }

    public void push(Object x) {
     S[--top] = x;
    }
```

Limitations

- Inheritance is useful for extending functionality, but it can’t do everything.
- By defining `Stack` to hold `Object`s, we “lose” functionality when we remove things from the stack:

```
...

Stack S = new Stack(10);
S.push(new Integer(15));
S.push(new String(“foo”));


// this is the only legal way to
// retrieve items - why?
Object a = S.pop();
Object b = S.pop();

// what are the types of a and b?
```

Type Casting

- Java actually allows us to regain functionality by “casting” the returned `Object` into the “correct” type.

- This helps us use one class declaration to create different kinds of `Stack`s, but does not allow a heterogeneous `Stack`.

```
...

Stack S = new Stack(10);
S.push(new Integer(15));
S.push(new String(“foo”));


// this is the only legal way to
// retrieve items - why?
Integer a = (Integer) S.pop();
String b = (String) S.pop();

// what are the types of a and b?
```

Java Generics

- Java also provides a mechanism to make classes generic, which avoids the need for casting:

```
class MyClass<T> {

    private T member_variable;

    public T foo(T x) {
     ...
    }

}
```

- This way, we can use the same class definition for multiple types (without losing functionality), and errors in type usage can still be caught at compile-time.

Type Parameters

- Type parameter example: Java's ArrayList (e.g. ArrayList<String>)

- Generic class: declared with one or more type parameters

- A type parameter for ArrayList denotes the element type:

```
public class ArrayList<E>
{
   public ArrayList() { . . . }
   public void add(E element) { . . . }
   . . .
}
```

Type Parameters

- Can be instantiated with class or interface type:

```
ArrayList<DynamicStack>
ArrayList<Stack>
```

- Cannot use a primitive type as a type variable:

```
ArrayList<double> // Wrong!
```

- Use corresponding wrapper class instead:

```
ArrayList<Double>
```

Implementing Generic Classes

- Example: simple generic class that stores pairs of objects such as:

```
Pair<String, Integer> one =
   new Pair<String, Integer>(“one", 1);
```

- Methods `getFirst` and `getSecond` retrieve first and second values of pair:

```
String name = one.getFirst();
Int num = one.getSecond();
```

- Generic `Pair` class requires two type parameters, one for each element type enclosed in angle brackets:

```
public class Pair<T, S>
```

Class **Pair**

```
public class Pair<T, S> {
        private T first;
        private S second;

        public Pair(T firstElement, S
secondElement) {
        first = firstElement;
        second = secondElement;
        }

        public T getFirst() {
        return first;
        }

        public S getSecond() {
        return second;
        }

        public String toString() {
        return "(" + first + ", " + second +
")";
        }
}
```

Good Type Variable Names

Type Variable	Name Meaning
`E`	Element type in a collection
`K`	Key type in a map
`V`	Value type in a map
`T`	General type
`S, U`	Additional general types