CSYE 6200

Concepts of Object Oriented Design

Functional Programming

Daniel Peters

d.peters@neu.edu

- Lecture
  1. Functional Programming

Functional Programming

- "*Functional programming* is just a a style of programming which focuses more on **functions** (some might like to call it programming with functions…"

-- Mehreen Tahir

https://www.codeproject.com/Articles/1267996/Functional-Programming-in-Cplusplus

Functional Programming

- "…, *Functional programming* is a programming paradigm in which we **code the computer program as** the evaluation of expressions same as mathematical **functions** (**No** changing-state and **mutable data**). "

-- Mehreen Tahir

https://www.codeproject.com/Articles/1267996/Functional-Programming-in-Cplusplus

Functional Programming

- "…, *Functional programming* **minimizes** the **moving parts** of the code (as opposed to the **Object Oriented** paradigm which **encapsulates** the moving parts) and thus makes the code easier to understand, compressed and predictable. "

-- Mehreen Tahir

https://www.codeproject.com/Articles/1267996/Functional-Programming-in-Cplusplus

Functional Programming

- "…, *Functional programming* exposes its true potential when it comes to **parallel processing**. The program will execute safely on multiple CPU(s) and you don't need to worry about excessive locking because it **doesn't use or modify any global resources** "

-- Mehreen Tahir

https://www.codeproject.com/Articles/1267996/Functional-Programming-in-Cplusplus

Functional Programming

- Concepts
  - Immutable data
  - Recursion
  - Lazy Evaluation
  - First Class Functions
  - Pure Functions
  - Higher Order Functions

-- Mehreen Tahir

https://www.codeproject.com/Articles/1267996/Functional-Programming-in-Cplusplus

Functional Programming

- Concepts
  - Immutable data
    - Object state is unchanged after object creation
    - Greatly simplifies concurrent programming
      - Eliminates need for synchronization
        - » No locking of critical code
        - » Allows best system performance

-- Mehreen Tahir

https://www.codeproject.com/Articles/1267996/Functional-Programming-in-Cplusplus

Functional Programming

- Concepts
  - Recursion
    - Recursion employed instead of loops to avoid mutable data

-- Mehreen Tahir

https://www.codeproject.com/Articles/1267996/Functional-Programming-in-Cplusplus

Functional Programming

- Concepts
  - Lazy Evaluation
    - Expressions are evaluated if and only when necessary
    - Optimization

-- Mehreen Tahir

https://www.codeproject.com/Articles/1267996/Functional-Programming-in-Cplusplus

Functional Programming

- Concepts
  - Lambda: a **First Class** Function
    - Can be *passed* as an argument *like data*
    - Can be *stored* in data structure *like data*

https://www.tutorialspoint.com/functional_programming_with_java/functional_programming_with_java_firstclass_function.htm

-- Mehreen Tahir

https://www.codeproject.com/Articles/1267996/Functional-Programming-in-Cplusplus

Functional Programming

- Concepts
  - Arrays.sort(): a **Higher Order** Function
    - **Accepts a function** as an argument (like data)
    - Returns a function like data
    - Arrays.sort()
      - Accepts a Comparator for specification of a sort order

https://jenkov.com/tutorials/java-functional-programming/higher-order-functions.html

-- Mehreen Tahir

https://www.codeproject.com/Articles/1267996/Functional-Programming-in-Cplusplus

Functional Programming

- Concepts
  - Pure Functions
    - ***Always returns the same result*** when supplied with the ***same parameter arguments***
    - Return Value depends only on input
    - No side effects (does not affect state)

https://www.freecodecamp.org/news/pure-function-vs-impure-function/

-- Mehreen Tahir

https://www.codeproject.com/Articles/1267996/Functional-Programming-in-Cplusplus

Functional Programming

- "*Functional programming* is a programming style in which computations are codified as *functional programming functions*. These are mathematical functions-like constructs (e.g., **lambda** functions) that are **evaluated in expression contexts**."

https://www.javaworld.com/article/3314640/java-101-functional-programming-for-java-developers-part-1.html

Functional Programming

- “Functional programming languages are *declarative*, meaning that a computation's logic is **expressed without describing its control flow**. In declarative programming, there are no statements. “

https://www.javaworld.com/article/3314640/java-101-functional-programming-for-java-developers-part-1.html

Functional Programming

- “A *computation* in functional programming is described by **functions that are evaluated** in expression contexts. ”

https://www.javaworld.com/article/3314640/java-101-functional-programming-for-java-developers-part-1.html

Functional Programming

- "Each time a functional programming function is called with the **same arguments**, the **same result** is achieved. Functions in functional programming are said to exhibit *referential transparency*. "

https://www.javaworld.com/article/3314640/java-101-functional-programming-for-java-developers-part-1.html

Functional Programming

- "This means you could **replace a function call with its resulting value** without changing the computation's meaning. "

https://www.javaworld.com/article/3314640/java-101-functional-programming-for-java-developers-part-1.html

Functional Programming

- "Functional programming favors *immutability*, which means the **state cannot change**. This is typically not the case in imperative programming, where an imperative function might be associated with state (such as a Java instance variable)."

https://www.javaworld.com/article/3314640/java-101-functional-programming-for-java-developers-part-1.html

Functional Programming

- Calling this function at different times with the same arguments might result in different return values because in this case state is *mutable*, meaning it changes. ”

https://www.javaworld.com/article/3314640/java-101-functional-programming-for-java-developers-part-1.html

Functional Programming

- **Referential transparency** is an oft-touted **property of (pure) functional languages**, which makes it easier to reason about the behavior of programs. I don't think there is any formal definition, but it usually means that an **expression always evaluates to the same result in any context**.

https://wiki.haskell.org/Referential_transparency

Functional Programming

- **Referential transparency**. ... An expression is said to be referentially transparent if **it can be replaced with its corresponding value** without changing the program's behavior. As a result, evaluating a referentially transparent function gives the **same value for same arguments**. Such functions are **called pure functions**.

https://en.wikipedia.org/wiki/Referential_transparency

Functional Programming

- A **pure function** is a function where the **return value is only determined by its input values**, without observable side effects.

https://www.sitepoint.com/functional-programming-pure-functions/

Functional Programming

- In computer science, **functional programming** is a programming paradigm—a style of building the structure and elements of computer programs—that treats computation as the **evaluation of mathematical functions and avoids changing-state and mutable data**.

https://en.wikipedia.org/wiki/Functional_programming

Functional Programming

- In programming language theory, **lazy evaluation**, or call-by-need is an **evaluation** strategy which delays the **evaluation** of an expression until its value is needed (non-strict **evaluation**) and which also avoids repeated **evaluations** (sharing).

https://en.wikipedia.org/wiki/Lazy_evaluation

Functional Programming

- Not Imperative Programming Style
  - Low Level
  - Step by step instruction
  - Define “What” by implementing “**How**”

- Is Declarative Programming Style
  - High Level
  - Implementation is covered (under abstraction)
  - Declare ‘**What**’ without so much ‘How’

CSYE 6200

Concepts of Object-Oriented Design

Java Stream API

Daniel Peters

d.peters@neu.edu

- Lecture
  1. Functional Style Programming
  2. Stream API
  3. Functional Interface
  4. Generic
  5. Streams
  6. Predicate

Functional Programming

- Concepts
  - Immutable data
  - Recursion
  - Lazy Evaluation
  - First Class Functions
  - Pure Functions
  - Higher Order Functions

-- Mehreen Tahir

https://www.codeproject.com/Articles/1267996/Functional-Programming-in-Cplusplus

Declarative Programming

- Not Imperative Programming Style
  - Low Level
  - Step by step instruction
  - Define “What” by implementing “**How**”

- Is Declarative Programming Style
  - High Level
  - Implementation is covered (under abstraction)
  - Declare ‘**What**’ without so much ‘How’

Find Dan: Imperative

```
final List<String> names = Arrays.asList("jim", "sue", "dan",
"len", "zac");

    boolean found = false;
    for (String name : names) {
            if (name.equals("dan")) {
                    found = true;
                    break;    // exit loop
            }
    }
    System.out.println("You found dan! " + found);
```

OUTPUT: *You found dan!* true

Find Dan: Declarative

final List<String> names = Arrays.*asList("jim", "sue", "dan", "len", "zac");*

System.*out.println(“You found dan! "*

+ *names.**contains**("dan"*));

OUTPUT: *You found dan!* true

Find Dan: Declarative

```
List<String> names = new ArrayList<String>(
Arrays.asList("jim", "sue", "dan", "len", "zac"));

System.out.print("You found dan! ");
names.stream()
        .filter(s -> s == "dan")
        .forEach(System.out::print);
```

OUTPUT: *You found dan!* dan

Find Number 7: Declarative

```
Integer[] a = {0,1,2,3,4,5,6,7,8,9};
List<Integer> mutableNumbers = new
ArrayList<Integer>(Arrays.asList(a));

System.out.print("\n Filter 7 from numbers 0123456789: ");
mutableNumbers.stream()
        .filter(n -> n == 7)
        .forEach(System.out::print);
System.out.println();
```

OUTPUT: Filter 7 from numbers 0123456789: 7

Streams

Package java.util.stream

"Classes to support ***functional-style operations*** on streams of elements, such as map-reduce transformations on collections."

https://docs.oracle.com/javase/8/docs/api/java/util/stream/package-summary.html

Streams

- No storage
  - Streams carry values from a finite source (e.g. *collection*) or an infinite source (e.g. *I/O*)
- Functional
  - Declarative and produces results without modifying underlying data source.
- Lazy
  - Many intermediate operations are **lazy** effecting ***efficient*** pipeline processing.

Stream processing

- Source
  - array, a collection, a generator function, an I/O channel
- Optional ***Intermediate*** operations
  - transform a stream into another stream
    - Followed by another operation
      - Intermediate or Terminal
- ***Terminal*** operation
  - which produces a result
    - End of Stream

Stream processing

- "Streams are ***lazy***;"
- " ***computation*** on the source data is ***only performed*** when the ***terminal operation*** is initiated,"
- " and ***source elements*** are ***consumed only as needed***."

https://docs.oracle.com/javase/8/docs/api/

Stream processing

- Lazy
  - *Intermediate* operations are generally ***lazy*** (e.g., when returning a stream), *postponing* processing until a *terminal* operation.
  - Allows for **efficient operation** over infinite sources, i.e., terminating upon locating element.
- Eager
  - *Terminal* operations (a value producing result or action which does **NOT** return a stream) are generally eager.

Example Stream API

```
public void simpleStream() {
        List<Integer> list = Arrays.asList(5,2,4,1,3);
        list.forEach(n -> System.out.print(n + " "));
        System.out.println("reduce to sorted odd subset");
        list.stream()
        .filter(n -> n % 2 == 1)          // odd ONLY
        .sorted()          // ascending
        .map(n -> 100*n)       // scale by 100
        .forEach(n -> System.out.print(n + ", "));     // output
        System.out.println();
}
```

OUTPUT: 100, 300, 500,

stream.filter()

java.util.stream

filter(Predicate<T> predicate)

Intermediate stream operation

•Returns a stream consisting of the elements of this stream that match the given predicate.

```
int sum = widgets.stream()
        .filter(w -> w.getColor() == RED)
        .mapToInt(w -> w.getWeight())
        .sum();
```

Predicate

java.util.function

Interface Predicate<T>

@**FunctionalInterface**

public interface Predicate<T>

•Represents a function which returns a **boolean** value (i.e. a boolean-valued function, a predicate) that accepts a **single** argument.

•Method

boolean **test**(T t)

stream.sorted()

java.util.stream

sorted(Comparator<? Super T> comparator)

Intermediate (stateful) stream operation

•Returns a stream consisting of the elements of this stream, sorted according to the provided Comparator.

```
int sum = widgets.stream()
        .filter(w -> w.getColor() == RED)
        .sorted(Widgets::sortByColor)
        .sum();
```

Comparator

java.util.function

Interface Comparator<T> comparator>

@**FunctionalInterface**

public interface Comparator<T>

- A comparison function, which can be passed to a sort method (such as Collections.sort).

- Method

int **compare**(T t1, T t2)

stream.map()

java.util.stream

map (Function<? super T, ? extends R> mapper)

Intermediate stream operation

•Returns a stream consisting of the results of applying the given function to the elements of this stream.

```
int sum = widgets.stream()
        .filter(w -> w.getColor() == RED)
        .mapToInt(w -> w.getWeight())
        .sum();
```

Function

java.util.function

Interface Function<? super T, ? extends R>

@**FunctionalInterface**

public interface Function<? super T, ? extends R>

- Represents an operation that accepts a single input argument and returns no result. Unlike most other functional interfaces, Consumer is expected to operate via side-effects.

- Method

R **apply**(T t)

stream.map()

java.util.stream

map (Function<? super T,? extends R> **mapper)**

Intermediate stream operation

•Returns a stream consisting of the results of applying the given function to the elements of this stream.

```
String[] a = {“1,0.49,Candy”, “2,1.49,Bread” }
List<Item> list = Arrays.asList(“a”).stream()
        .map(Item::new)
        .collect(Collectors.toList());
```

stream.foreach()

java.util.stream

foreach (Consumer<? super T**> action)**

Terminal stream operation

•Performs an action for each element of this stream.

```
int sum = widgets.stream()
        .filter(w -> w.getColor() == RED)
        .sorted(Widget::sortByColor)
        .foreach(System.out::println);
```

Consumer

java.util.function

Interface Consumer<T>

@**FunctionalInterface**

public interface Consumer<T>

- Represents an operation that accepts a single input argument and returns no result. Unlike most other functional interfaces, Consumer is expected to operate via side-effects.

- Method

void **accept**(T t)

Predicate Example 1a

```
public void useStreamFilter() {
        List<Integer> numbers = Arrays.asList(1,2,3,4,5);
        numbers
        .stream()
        .filter( n -> n > 3 )
        .forEach(System.out::print);
   }
}
```

Predicate Example 1b

```
public class DemoStreams {
   public boolean greaterThanThree(Integer n) {
      return n > 3;
      }
   public void useStreamFilter() {
            List<Integer> numbers = Arrays.asList(1,2,3,4,5);
            numbers
            .stream()
            .filter( DemoStreams::greaterThanThree )
            .forEach(System.out::print);
   }
}
```

Predicate Example 1c

```
public class DemoStreams {
   public void useStreamFilter() {
          Predicate<Integer> greaterThan3 = (Integer n) ->
{return n>3;};
          List<Integer> numbers = Arrays.asList(1,2,3,4,5);

          numbers.stream()
          .filter( greaterThan3 )
          .forEach(System.out::print);
   } // end useStreamFilter()
} // end class DemoStreams
```

Predicate Example 1

- OUTPUT:

4 5

Predicate Example

```
public  void simplePredicate () {
    List<Integer> numbers = Arrays.asList(1,2,3,4,5,6,7,8,9);
    Predicate<Integer> numOver5 = n -> { return n > 5; };

    for (Integer n : numbers) {
            if ( numOver5.test(n) ) {
            System.out.print(n + " <** ");
            } else {
            System.out.print(n + " ");
            } // end if
    } // end for
} // end simplePredicate
```

OUTPUT: 1 2 3 4 5 6 <** 7 <** 8 <** 9 <**

Using a Predicate for Stream Filter

- Given the Collection ‘states’

```
List<String> states =
    Arrays.asList("ma","ny","ct","vt","ri","nh","nv","nc","nd","wa","wv",
    "ut","ca","az","al","ak","ok","pa","me","ms","il","id","mn","wy","mt",
    "wi","ia","ar","hi","sd","sc","md","nj","de","ga","fl","mi","oh","in","o
    r","ky","tn","va","mo","ks","co","la","tx","nm","ne");
```

- And the Predicate ,‘uStates‘

```
Predicate<String> uStates= s -> { return s.startsWith(“u"); };
```

Example for Stream Filter

- The following Stream Processing sourced by the ‘states’ String Collection filters using a lambda:

```
states.stream()
        .filter(s -> s.startsWith(“u"))
        .map(String::toUpperCase)
        .sorted()
        .forEach(s -> System.out.print(s + ", "));
```

•Produces the (reduced) OUTPUT:

UT,

Using a Predicate for Stream Filter

- The following Stream Processing sourced by the 'states' String Collection filters using the uStates predicate:

```
states.stream()
        .filter(uStates)
        .map(String::toUpperCase)
        .sorted()
        .forEach(s -> System.out.print(s + ", "));
```

•Produces the (reduced) OUTPUT:

UT,

Example for Sequential Stream Filter

- Filter the subset greater than the number 5

```
Integer[] a = {0,1,2,3,4,5,6,7,8,9};
List<Integer> numbers = new ArrayList<Integer>(Arrays.asList(a));

Stream<Integer> sequentialStream = numbers.stream();
Stream<Integer> highNumsSeq = sequentialStream
    .filter(n -> n > 5);
highNumsSeq
    .forEach(n -> System.out.println("High Nums
    sequential="+ n));
```

Example for Stream Filter

- Filter the subset greater than the number 5

OUTPUT:

High Nums sequential=6
High Nums sequential=7
High Nums sequential=8
High Nums sequential=9

Example for Parallel Stream Filter

- Filter the subset greater than the number 5

```
Integer[] a = {0,1,2,3,4,5,6,7,8,9};
List<Integer> numbers = new ArrayList<Integer>(Arrays.asList(a));

Stream<Integer> parallelStream = numbers.parallelStream();
Stream<Integer> highNums = parallelStream
    .filter(n -> n > 5);
//using lambda in forEach
highNums.forEach(n -> System.out.println("High Nums
    parallel="+n));
```

Example for Stream Filter

- Filter the subset greater than the number 5

OUTPUT:

High Nums parallel=8
High Nums parallel=9
High Nums parallel=7
High Nums parallel=6

Online Links

- Oracle:
  - http://www.oracle.com/technetwork/articles/java/ma14-java-se-8-streams-2177646.html

CSYE 6200

Concepts of Object-Oriented Design

Java Subtypes

Daniel Peters

d.peters@neu.edu

- Lecture:
  - Interfaces as Subtypes

Interface

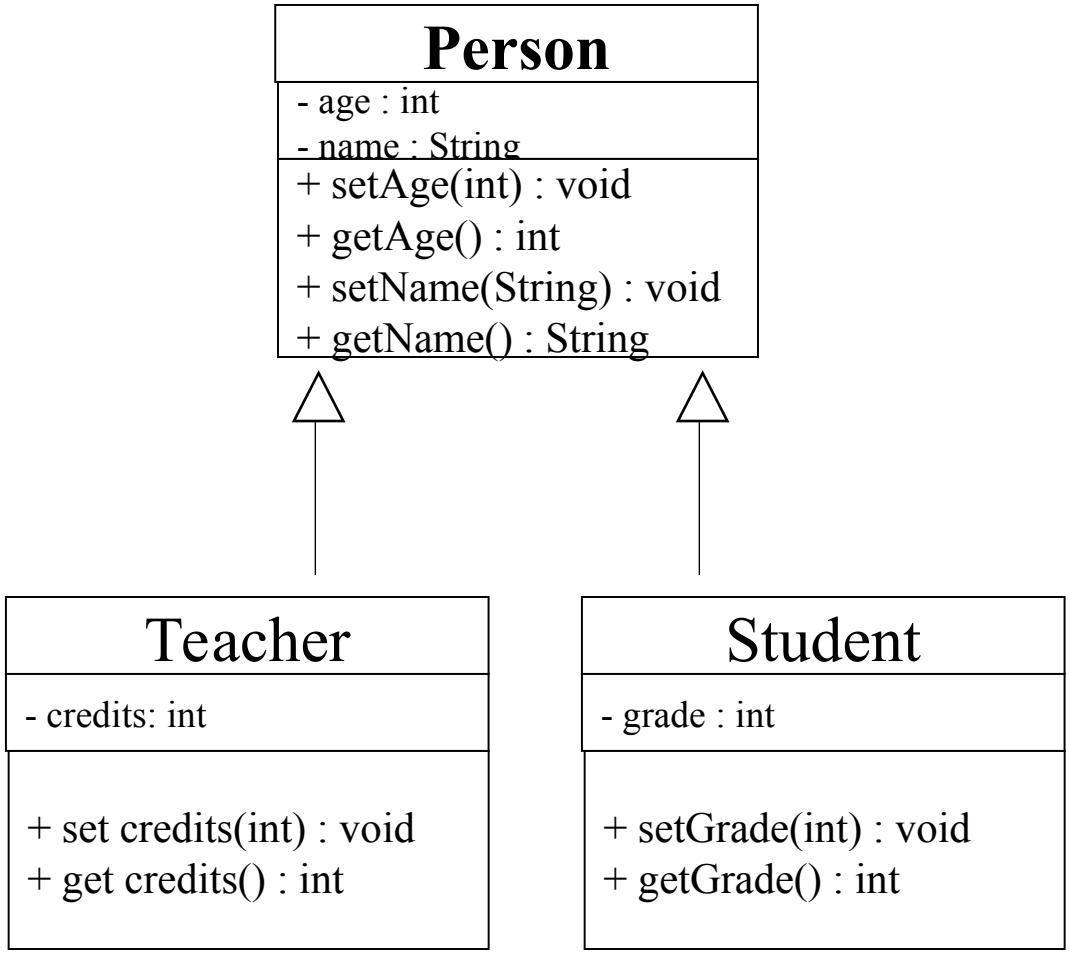
- In its most common form, an interface is a **group of related methods** (signatures) with no method bodies.
  - Fully implemented Exceptions:
    - Default methods
    - Static methods
- https://docs.oracle.com/javase/tutorial/java/concepts/interface.html

Interface Inheritance

- Multiple inheritance supported ONLY for interfaces
  - Interface ***extends*** one or more ***parent*** interfaces;

https://www.tutorialspoint.com/java/java_interfaces.htm

Person Class Diagram

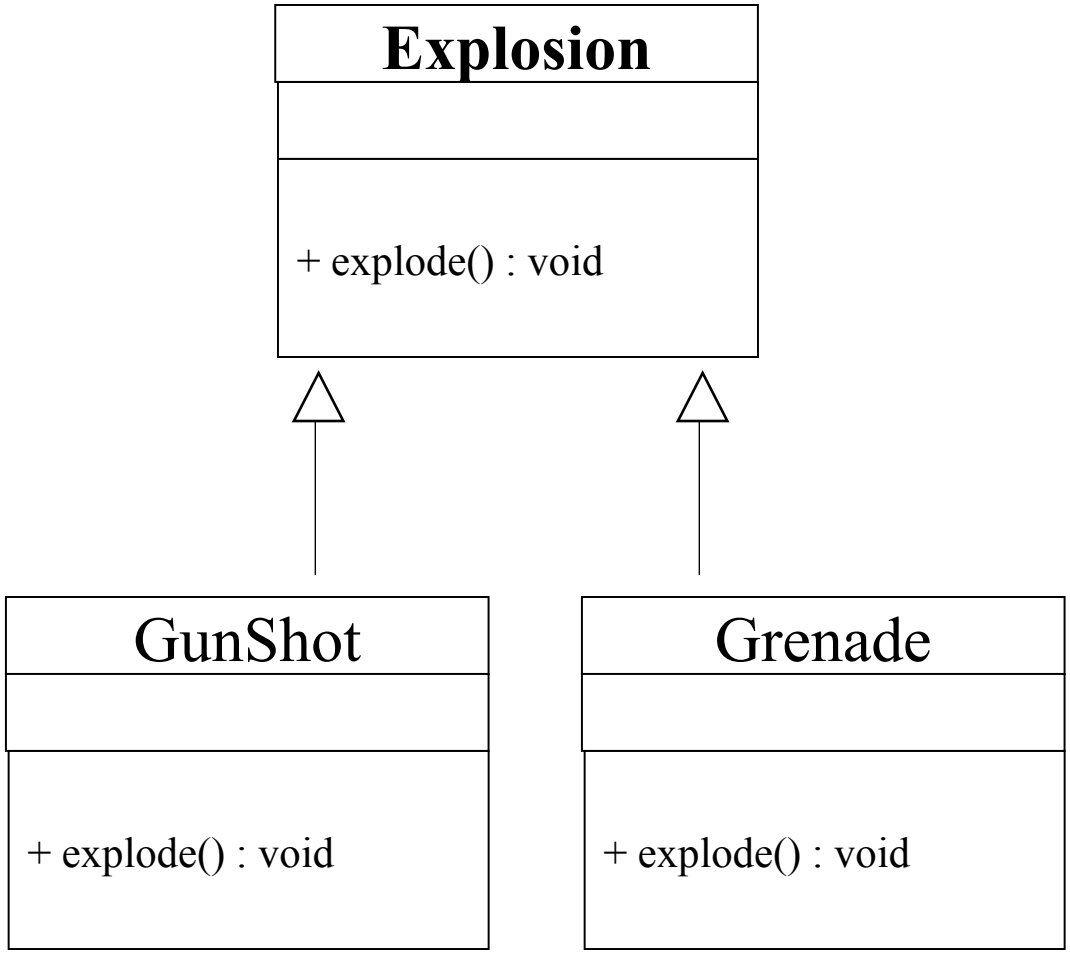


Inheritance: Is-A

- Person p = new Person();
- Person t = new Teacher();
- Person s = new Student();
- Object o = new Person();

- Inheritance relationship allows for subtype to be treated as if it were the super type
  - Super type/subtype "**Is-A**" Relationship
    - Child subtype Is-A Parent super type

Explosion Class Diagram



Inheritance: Is-A

- Explosion gunshot = new GunShot();
- Explosion grenade = new Grenade();

- Inheritance relationship allows for subtype to be treated as if it were the super type
  - Super type/subtype "**Is-A**" Relationship
    - Child subtype Is-A Parent super type

Reference Types

- Class
  - String
  - Is-A Inherited Type
    - Single Inheritance
      - Student Is-A Person
      - GunShot Is-A Explosion
- Interface
  - Multiple implementations
  - Runnable
  - Comparable

instanceof

```
if (new String() instanceof Comparable) {
  System.out.println("String is Comparable!");
}
```

CONSOLE OUTPUT

String is Comparable!

instanceof

```
String name = “Dan”; // Yes!!!
if (name instanceof Comparable) {
        System.out.println(name
                + ” is Comparable!");
}
```

CONSOLE OUTPUT

Dan is Comparable!

instanceof

```
String name = new String("Dan"); // NO
if (name instanceof Comparable) {
        System.out.println(name
                + " is Comparable!");
}
```

CONSOLE OUTPUT

Dan is Comparable!

Interface Inheritance

```
/**
 * An API for obtaining a metric for comparison and quantification.
 */
public interface Measurable {
        /**
         * the measure for quantification.
         * @return    metric value.
         */
        double getMetric();
        /**
         * the description of the measure for quantification.
         * @return    String description of metric value.
         */
        String getMetricDescription();
}
```

Interface Inheritance

```
public interface GunShotMeasurable extends Measurable {
        /**
         * the measure for quantification: muzzle velocity.
         * @return metric value.
         */
        double getMetric();
        /**
         * the description of the measure for quantification: muzzle velocity.
         * @return metric value.
         */
        String getMetricDescription();
        int getCaliber();        // get caliber of gun
}
```

SubType

```
public class GunShot extends Explosion
        implements GunShotMeasurable,
                Comparable<GunShotMeasurable> {
  . . .
}
```

- GunShot **Is-A**:
  - 6 Types
    - Object > Explosion > GunShot
    - GunShotMeasurable > Measurable
    - Comparable

Serializable Interface

"The serialization interface has **no methods or fields** and serves only to identify the semantics of being serializable."

https://docs.oracle.com/javase/8/docs/api/index.html

CSYE 6200

Concepts of Object-Oriented Design

Intro to Data Structures: Stack

Daniel Peters

d.peters@neu.edu

Intro to Data Structures

- Stack
  - LIFO: Last In, First Out
- Borrows from real life
  - Pile of books on a table
- Useful for modeling triage:
  - highest priority item tops everything else

Stack Design Approach in Java

- Fixed-Size stack
  - Stack top index determines full or empty
- Top index used to traverse up and down stack
- Top index used for push and pop
  - Stack Operation: pre-inc Push, post-dec Pop
    - Check for full BEFORE push
      - Pre-Increment BEFORE push (init -1)
    - Check for Empty BEFORE pop
      - Post-Decrement AFTER pop (init 0)

Stack Design Approach in Java

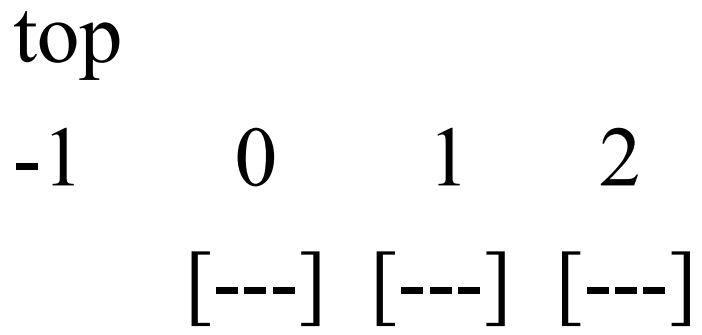
- Stack Errors:
  1. Push attempt on FULL stack
  2. Pop attempt on EMPTY stack

Stack Implementation in Java

- Stack Attributes:
  - Array arr: contiguous queue element storage
  - Top Index: used to traverse up and down stack
    - Increment Top to push onto stack
    - Decrement Top to pop off of stack
  - Capacity: maximum count of stack

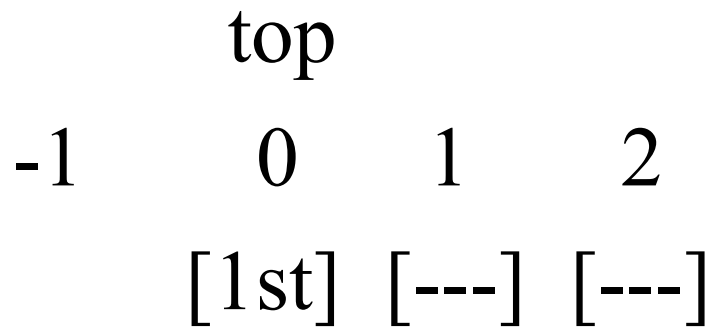
Stack Design Approach in Java

- Empty 3 element Stack:



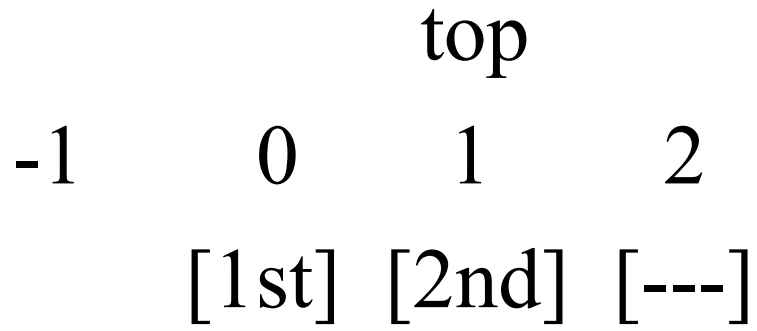
Stack Design Approach in Java

- 3 element Stack after adding 1st element:



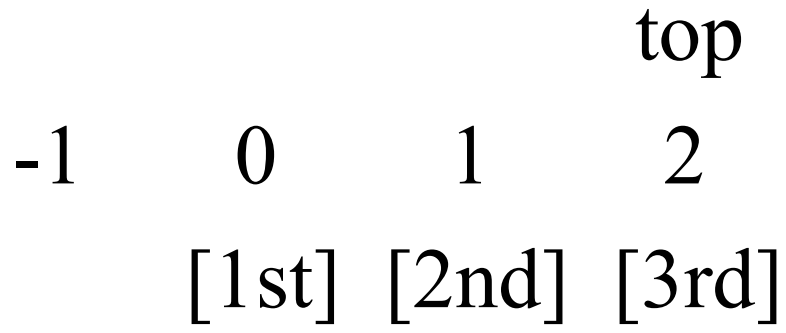
Stack Design Approach in Java

- 3 element Stack after adding 2nd element:



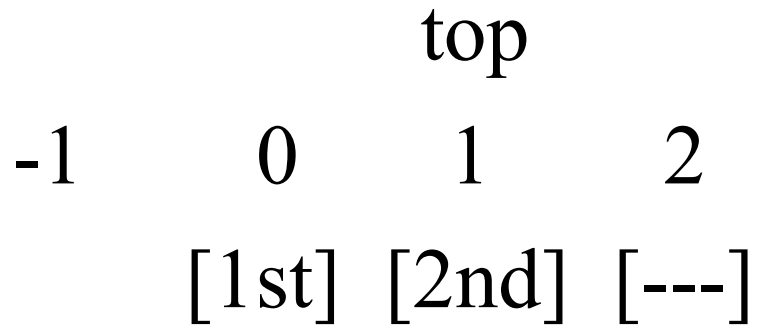
Stack Design Approach in Java

- 3 element Stack after adding 3rd element:



Stack Design Approach in Java

- 3 element Stack after removing 3rd element:



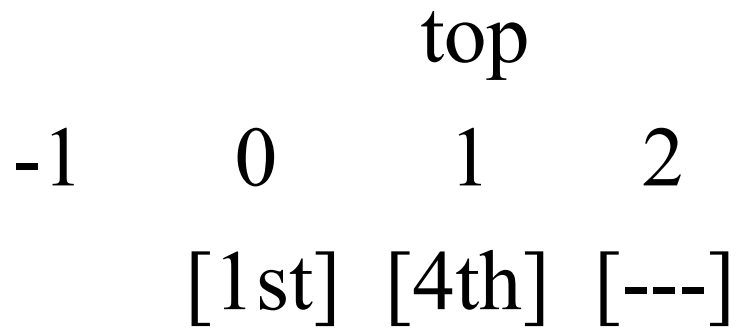
Stack Design Approach in Java

- 3 element Stack after removing 2nd element:

```
        top
-1       0       1       2
        [1st]  [---]  [---]
```

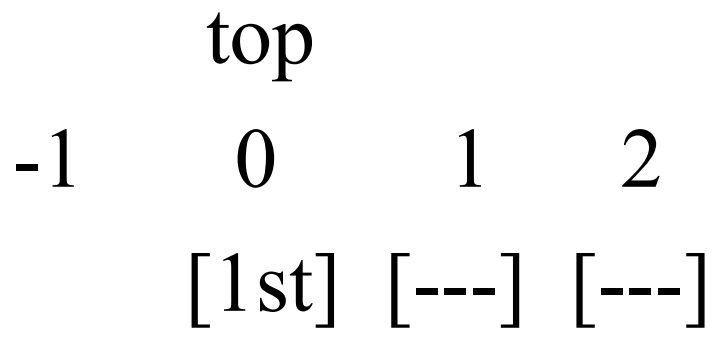
Stack Design Approach in Java

- 3 element Stack after adding 4th element:



Stack Design Approach in Java

- 3 element Stack after removing 4th element:



Stack Design Approach in Java

- 3 element Stack after removing 1st element:

top

-1 0 1 2

[---] [---] [---]

Stack Implementation in Java

- Stack API:
  - Size
  - isEmpty
  - isFull
  - Push
  - Pop
  - Peek

Stack Class Constructor

```
public Stack(int size) {
        arr = new int[size];
        capacity = size;
        top = -1;
}
```

- Queue attribute initialization

Stack: int size()

```
public int size() {
        return top + 1;
}
```

- Utility for determining count of elements on stack

Stack: boolean isEmpty()

```
public boolean isEmpty() {
        return top == -1;
}
```

Stack: boolean isFull()

```
public boolean isFull() {
        return (top == capacity - 1);
}
```

Stack: void push(int item)

```
public void push(int item) {
        if (isFull()) {
                System.exit(1); // ERROR
        }
        System.out.println(”Inserting " + item);
        arr[++top] = item;
}
```

Stack: void pop()

```
public int pop() {
        if (isEmpty()) {
                System.exit(1); // ERROR
        }
        System.out.println("Removing " + peak());
        return arr[top--];
}
```

Stack: void peek()

```
public int peek() {
        if (!isEmpty()) {
                return arr[top];
        } else {
                System.exit(1); // ERROR
        }
        return -1;
}
```

Stack: void demo()

```
public void demo() {
        Stack q = new Stack(5);
        q.push(1);    // 1st in
        q.push(2);    // 2nd in
        q.pop();      // 2nd out is 2
        q.pop();      // 1st out is 1

        q.push(2);    // 3rd in
        System.out.println("Top element is: " + q.peek());
        System.out.println("Stack size is " + q.size());

        q.pop(3);     // 3rd out is 3

        if ( q.isEmpty() )
            System.out.println("Stack Is Empty");
        else
            System.out.println("Stack Is Not Empty");
}
```

Stack: void demo()

```
public void demo() {
        Stack q = new Stack(5);
        q.push(1);        // 1st in
        q.push(2);        // 2nd in
        q.pop();          // 2nd out is 2
        q.pop();          // 1st out is 1

        q.push(2);        // 3rd in
        System.out.println(”Top element is: " + q.peek());
        System.out.println(”Stack size is " + q.size());

        q.pop(3);         // 3rd out is 3

        if ( q.isEmpty() )
            System.out.println(”Stack Is Empty");
        else
            System.out.println(”Stack Is Not Empty");
}
```

Stack: void demo()

```
public void demo() {
        Stack q = new Stack(5);
        q.push(1);      // 1st in
        q.push(2);      // 2nd in
        q.pop();        // 2nd out is 2
        q.pop();        // 1st out is 1

        q.push(2);      // 3rd in
        System.out.println("Top element is: " + q.peek());
        System.out.println("Stack size is " + q.size());

        q.pop(3);       // 3rd out is 3

        if ( q.isEmpty() )
            System.out.println("Stack Is Empty");
        else
            System.out.println("Stack Is Not Empty");
}
```

Stack: void demo()

```
public void demo() {
        Stack q = new Stack(5);
        q.push(1);      // 1st in
        q.push(2);      // 2nd in
        q.pop();        // 2nd out is 2
        q.pop();        // 1st out is 1
        q.push(2);      // 3rd in
        System.out.println("Top element is: " + q.peek());
        System.out.println("Stack size is " + q.size());

        q.pop(3);       // 3rd out is 3

        if ( q.isEmpty() )
            System.out.println("Stack Is Empty");
        else
            System.out.println("Stack Is Not Empty");
}
```

Stack: void demo()

```
public void demo() {
        Stack q = new Stack(5);
        q.push(1);    // 1st in
        q.push(2);    // 2nd in
        q.pop();      // 2nd out is 2
        q.pop();      // 1st out is 1

        q.push(2);    // 3rd in
        System.out.println(”Top element is: " + q.peek());
        System.out.println(”Stack size is " + q.size());

        q.pop(3);         // 3rd out is 3

        if ( q.isEmpty() )
            System.out.println(”Stack Is Empty");
        else
            System.out.println(”Stack Is Not Empty");
}
```

Stack: void demo()

```
public void demo() {
        Stack q = new Stack(5);
        q.push(1);      // 1st in
        q.push(2);      // 2nd in
        q.pop();        // 2nd out is 2
        q.pop();        // 1st out is 1

        q.push(2);      // 3rd in
        System.out.println("Top element is: " + q.peek());
        System.out.println("Stack size is " + q.size());

        q.pop(3);     // 3rd out is 3

        if ( q.isEmpty() )
            System.out.println("Stack Is Empty");
        else
            System.out.println("Stack Is Not Empty");
}
```

CSYE 6200

Concepts of Object-Oriented Design

Intro to Data Structures: Queue

Daniel Peters

d.peters@neu.edu

Intro to Data Structures

- Queue
  - FIFO: First In, First Out
- Borrows from real life
  - Customers waiting in line to make purchase from a store cashier
  - Cycling oldest inventory to front of shelf
- Useful for modeling first come, first served

Queue Design Approach in Java

1. Fixed-Size queue
   - Queue count determines full or empty
2. Circular indices wrap back to zero upon reaching maximum queue size
   1. Rear index used for enqueue
      - Check for full BEFORE enqueue
      - Increment BEFORE enqueue (init -1)
   2. Front index used for dequeue
      - Check for Empty BEFORE dequeue
      - Increment AFTER dequeue (init 0)

Queue Design Approach in Java

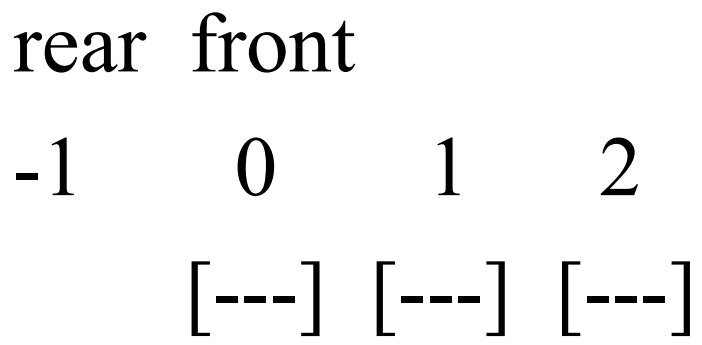
- Queue Errors:
  1. Enqueue attempt on FULL queue
  2. Dequeue attempt on EMPTY queue

Queue Implementation in Java

- Queue Attributes:
  - Array arr: contiguous queue element storage
  - Front Index: circular dequeue index
  - Rear Index: circular enqueue index
  - Capacity: maximum count of queue
  - Count: current contents count in queue

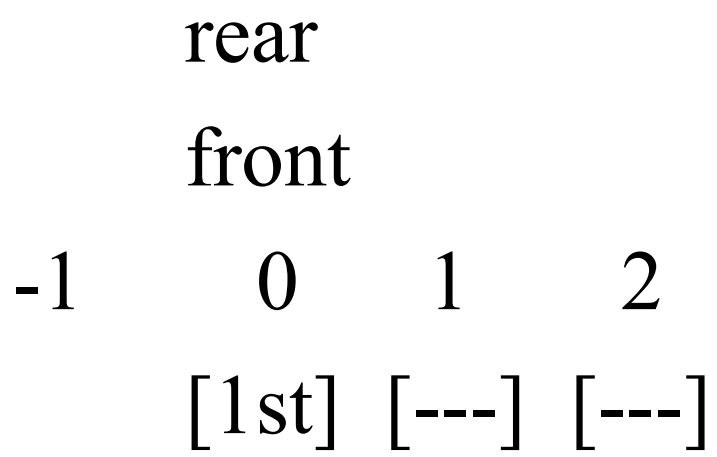
Queue Design Approach in Java

- Empty 3 element Queue:



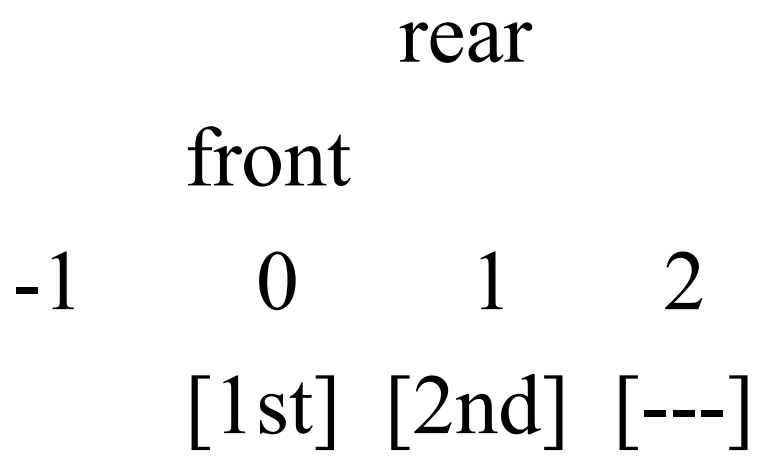
Queue Design Approach in Java

- 3 element Queue after adding 1st element:



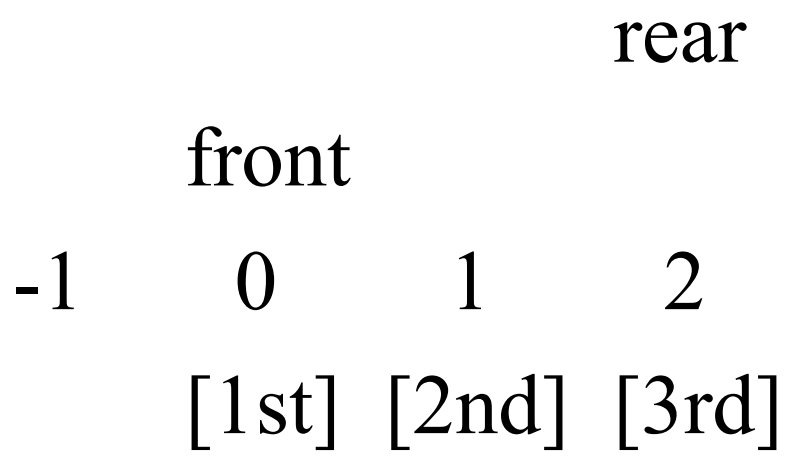
Queue Design Approach in Java

- 3 element Queue after adding 2nd element:



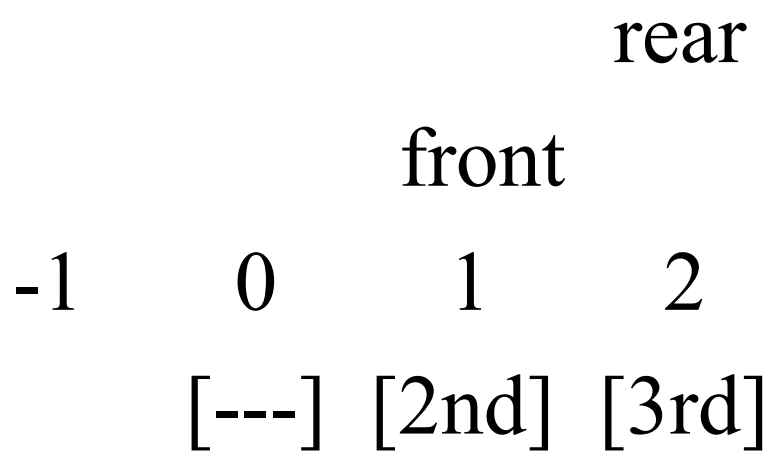
Queue Design Approach in Java

- 3 element Queue after adding 3rd element:



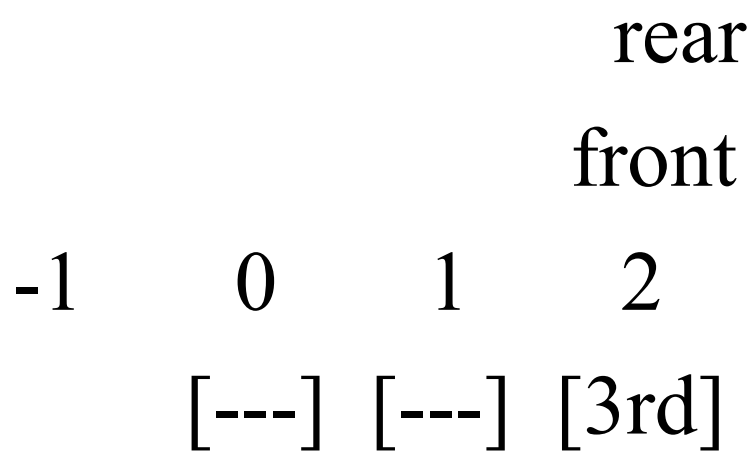
Queue Design Approach in Java

- 3 element Queue after removing 1st element:



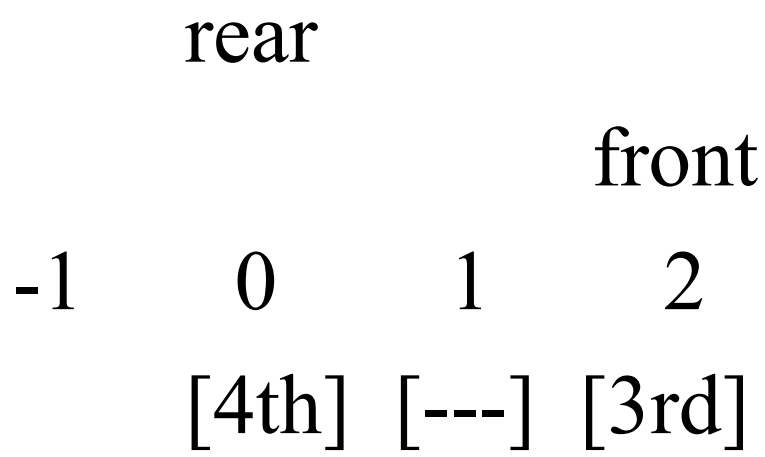
Queue Design Approach in Java

- 3 element Queue after removing 2nd element:



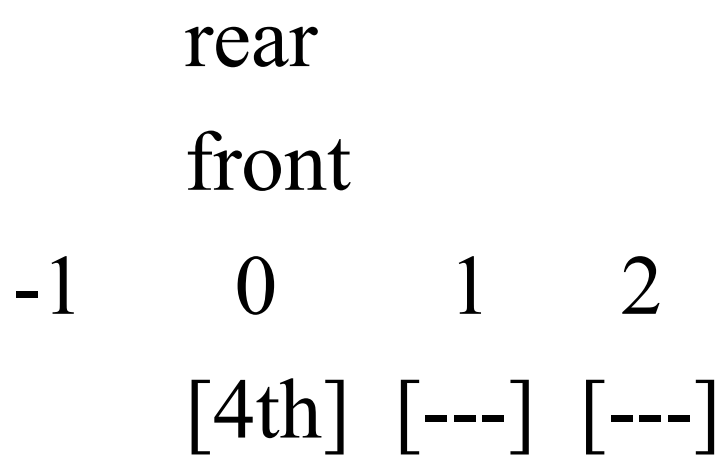
Queue Design Approach in Java

- 3 element Queue after adding 4th element:



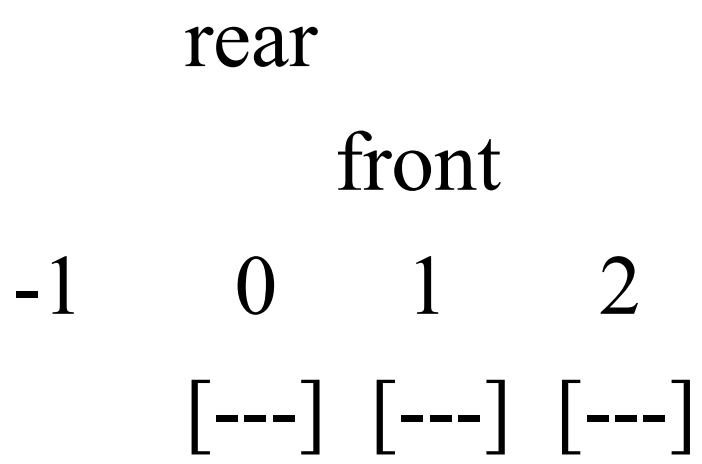
Queue Design Approach in Java

- 3 element Queue after removing 3rd element:



Queue Design Approach in Java

- 3 element Queue after removing 4th element:



Queue Implementation in Java

- Queue API:
  - Size
  - isEmpty
  - isFull
  - Enqueue
  - Dequeue
  - Peek

Queue Class Constructor

```
public Queue(int size) {
        arr = new int[size];
        capacity = size; // Max size
        front = 0;
        rear = -1;
        count = 0; // elements in Queue
}
```

- Queue attribute initialization

Queue: int size()

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

- Utility for determining queue full or empty

Queue: boolean isEmpty()

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

Queue: boolean isFull()

```
public boolean isFull() {
        return (size() == capacity);
}
```

Queue: void enqueue(int item)

```
public void enqueue(int item) {
        if (isFull()) {
                System.exit(1); // ERROR
        }
        System.out.println(”Adding " + item);
        rear = (rear + 1) % capacity; // circular index
        arr[rear] = item;
        count++;
}
```

Queue: void dequeue()

```
public void dequeue() {
        if (isEmpty()) {
                System.exit(1); // ERROR
        }
        System.out.println("Removing " + arr[front]);
        front = (front + 1) % capacity; // circular index
        count--;
}
```

Queue: int dequeue()

```
public int dequeue() {
        if (isEmpty()) {
                System.exit(1); // ERROR
        }
        int e = arr[front];
        System.out.println("Removing " + e);
        front = (front + 1) % capacity; // circular index
        count--;
        return e;
}
```

Queue: void peek()

```
public int peek() {
        if (isEmpty()) {
                System.exit(1); // ERROR
        }
        return arr[front]);
}
```

Queue: void demo()

```
public void demo() {
        Queue q = new Queue(5);
        q.enqueue(1);               // 1st in
        q.enqueue(2);               // 2nd in
        q.enqueue(3);               // 3rd in
        System.out.println("Front element is: " + q.peek());
        q.dequeue();// 1st out is 1
        System.out.println("Front element is: " + q.peek());
        System.out.println("Queue size is " + q.size());

        q.dequeue();// 2nd out is 2
        q.dequeue();// 3rd out is 3

        if ( q.isEmpty() )
            System.out.println("Queue Is Empty");
        else
            System.out.println("Queue Is Not Empty");
}
```

Queue: void demo()

```
public void demo() {
        Queue q = new Queue(5);
        q.enqueue(1);       // 1st in
        q.enqueue(2);       // 2nd in
        q.enqueue(3);       // 3rd in
        System.out.println("Front element is: " + q.peek());
        q.dequeue();    // 1st out is 1
        System.out.println("Front element is: " + q.peek());
        System.out.println("Queue size is " + q.size());

        q.dequeue();    // 2nd out is 2
        q.dequeue();    // 3rd out is 3

        if ( q.isEmpty() )
            System.out.println("Queue Is Empty");
        else
            System.out.println("Queue Is Not Empty");
}
```

Queue: void demo()

```
public void demo() {
        Queue q = new Queue(5);
        q.enqueue(1);  // 1st in
        q.enqueue(2);  // 2nd in
        q.enqueue(3);  // 3rd in
        System.out.println("Front element is: " + q.peek());
        q.dequeue();    // 1st out is 1
        System.out.println("Front element is: " + q.peek());
        System.out.println("Queue size is " + q.size());

        q.dequeue();    // 2nd out is 2
        q.dequeue();    // 3rd out is 3

        if ( q.isEmpty() )
            System.out.println("Queue Is Empty");
        else
            System.out.println("Queue Is Not Empty");
}
```

Queue: void demo()

```
public void demo() {
        Queue q = new Queue(5);
        q.enqueue(1);  // 1st in
        q.enqueue(2);  // 2nd in
        q.enqueue(3);  // 3rd in
        System.out.println("Front element is: " + q.peek());
        q.dequeue();// 1st out is 1
        System.out.println("Front element is: " + q.peek());
        System.out.println("Queue size is " + q.size());

        q.dequeue();    // 2nd out is 2
        q.dequeue();    // 3rd out is 3

        if ( q.isEmpty() )
            System.out.println("Queue Is Empty");
        else
            System.out.println("Queue Is Not Empty");
}
```

Queue: void demo()

```
public void demo() {
        Queue q = new Queue(5);
        q.enqueue(1);  // 1st in
        q.enqueue(2);  // 2nd in
        q.enqueue(3);  // 3rd in
        System.out.println("Front element is: " + q.peek());
        q.dequeue();// 1st out is 1
        System.out.println("Front element is: " + q.peek());
        System.out.println("Queue size is " + q.size());

        q.dequeue();// 2nd out is 2
        q.dequeue();// 3rd out is 3

        if ( q.isEmpty() )
            System.out.println("Queue Is Empty");
        else
            System.out.println("Queue Is Not Empty");
}
```

CSYE 6200

Concepts of Object Oriented Design

Java Swing

Daniel Peters

d.peters@neu.edu

- Lecture
  1. JFC
  2. Swing
  3. Top Level Containers
  4. JTable Example

Java Foundation Classes

- Swing GUI Components
- Pluggable Look-and-Feel Support
  - Choice of look
- Accessibility API
  - Support for assistive technologies
- Java 2D API
  - 2 Dimensional Graphics
- Internationalization
  - Multilanguage support

Swing History

- Abstract Window Toolkit (AWT)
  - Heavyweight components heavy on CPU
  - Leverages native system GUI code (peers)
  - Cross-Platform: Attempts to be portable with platform specific differences and dependencies
- Swing
  - Lightweight (peerless) components light on CPU
  - Leverages AWT minimally, then all Java GUI
  - Portable
- Java 6 : AWT/Swing interoperability

Java Heavyweight Components

- "A *heavyweight* component is associated with its own native screen resource (commonly known as a *peer*). Components from the java.awt package, such as Button and Label, are heavyweight components."
- http://www.oracle.com/technetwork/articles/java/mixing-components-433992.html

Java Lightweight Components

- "A *lightweight* component has no native screen resource of its own, so it is "lighter." A lightweight component relies on the screen resource from an ancestor in the containment hierarchy, possibly the underlying Frame object. Components from the javax.swing package, such as JButton and JLabel, are lightweight components."
- http://www.oracle.com/technetwork/articles/java/mixing-components-433992.html

Swing Hierarchy

java.lang.Object

java.awt.Component

java.awt.Container

JComponent

- JComponent
  - –Base class for all J* classes
    - JButton
    - JPanel
    - JTable
  - –EXCEPT JFrame, JDialog, JApplet

obsolete (no more in use)

Swing

- Not Thread Safe

  SWING is single threaded

  – "All Swing components and related classes, unless otherwise documented, must be accessed on the event dispatching thread (**EDT**)."

  – http://docs.oracle.com/javase/7/docs/api/index.html?javax/swing/JComponent.html

- Content Pane

  – "Each program that uses Swing components has at least one top-level container."

Top Level Containers

- javax.swing.JFrame
- javax.swing.JDialog

Dialog box

- javax.swing.JApplet

Used for execution within a browser, no browser uses it now, it's a security risk.

Top Level Containers

java.lang.Object

java.awt.Component

java.awt.Container

java.awt.Window

java.awt.Frame

- javax.swing.JFrame

Top Level Containers

java.lang.Object

java.awt.Component

java.awt.Container

java.awt.Window

java.awt.Dialog

- javax.swing.JDialog

Top Level Containers

java.lang.Object

java.awt.Component

java.awt.Container

java.awt.Panel

java.applet.Applet

- javax.swing.JApplet

Simple JTable example

```
public void simpleJTableExample {
  // create table model (DefaultTableModel) for
containing table data
   String [] colTitles = {"A","B", "C", "E"};
   DefaultTableModel myTM =
         new DefaultTableModel();
   myTM.setColumnCount(colTitles.length);
   myTM.setColumnIdentifiers(colTitles);
 . . .
}
```

Simple JTable example

```
public void simpleJTableExample {
…
   // add data to table model
   for (int ix=0; ix < 8; ++ix) {
        int i1 = 0 + 10*ix;
        int i2 = 3 + 10*ix;
        char c3 = (char) (0x41 + ix);
        double d4 = 1.0 + 10*ix;
        myTM.addRow(new Object[]{ i1, i2, c3, d4 });
  }
…
```

Simple JTable example

```
public void simpleJTableExample {
 . . .
   // create table (JTable)
   JTable table = new JTable();
   table.setModel(myTM);
   // make table sortable and resizable
   table.setAutoCreateRowSorter(true);

 table.setAutoResizeMode(JTable.AUTO_RESIZE_ALL_COLUMNS);
 . . .
}
```

Simple JTable example

```
public void simpleJTableExample {
 . . .
   // FINALLY create a pop-up window (JFrame)
   // and a ScrollPane for table
   JFrame frame = new JFrame("Pop-up Table Example");
   frame.add(new JScrollPane(table));
   frame.setSize(300, 100);
   frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
   frame.setVisible(true);
}
```

prepareGUI

```
private void prepareGUI() {
    JFrame frame = new JFrame("Java Swing GUI in JFrame");
    frame.setSize(600, 400);
    frame.setLayout(new GridLayout(3, 1));
    frame.addWindowListener(new WindowAdapter() {
            public void windowClosing(WindowEvent windowEvent) {
            System.exit(0);
            }
            });
    JPanel panel = new JPanel();
    panel.setLayout(new FlowLayout());
    frame.add(panel); // add to top-level container
    frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
    frame.setVisible(true);
}
```

CSYE 6200

Concepts of Object Oriented Design

Java Swing Controls

Daniel Peters

d.peters@neu.edu

GUI Controls

- Lecture
  1. JTextField
  2. JTextArea
  3. JPasswordField
  4. JRadio Button
  5. JComboBox
  6. JList
  7. JSpinner
  8. JSlider

JTextField

Class javax.swing.JTextField

- public class JTextField
- extends JTextComponent
- implements SwingConstants

- JTextField jTField = new JTextField ();
- String data = jTField.getText();
- jTField.setText("Dan");

JTextArea

Class javax.swing.JTextArea

- public class JTextArea
- extends JTextComponent

- JTextArea jTArea = new JTextArea ();
- String data = jTArea.getText();
- jTArea.setText("Dan");

JPasswordField

Class javax.swing. JPasswordField

- public class JPasswordField
- extends JTextField
- implements SwingConstants

- JPasswordField pw = new JPasswordField();
- String data = pw.getText();

JRadioButton

Class javax.swing.JRadioButton

- public class JRadioButton
- extends JToggleButton
- implements SwingConstants, Accessible

- JRadioButton rb1 = new JRadioButton (“One”);
- JRadioButton rb2 = new JRadioButton (“Two”);

Listeners

- Class java.util.EventListener
- public interface EventListener
- Handles events
  - **ActionListener**: ***button*** click action event
  - **ItemListener**: ***radio box*** state change item event
    - ItemEvent.SELECTED
    - ItemEvent.DESELECTED
  - **ChangeListener**: ***spinner*** model change event
    - ChangeEvent is the item detecting change

JRadioButton

```
final JRadioButton rb1 =
 new JRadioButton(”RB1", true);
rb1.setMnemonic(KeyEvent.VK_A);

final JRadioButton rb2 =
 new JRadioButton(”RB2");
rb2.setMnemonic(KeyEvent.VK_B);
```

JRadioButton ItemListener

```
rb1.addItemListener(new ItemListener() {
    public void itemStateChanged(ItemEvent e) {

      if (e.getStateChange() == 1) {
            System.out.println("RB1 checked");
      } else {
            System.out.println("RB1 unchecked");
      }
    }
});
```

JRadioButton ItemListener

```
rb2.addItemListener(new ItemListener() {
    public void itemStateChanged(ItemEvent e) {

      if (e.getStateChange() == 1) {
            System.out.println("RB2 checked");
      } else {
            System.out.println("RB2 unchecked");
      }
    }
});
```

ButtonGroup

- To apply a ***mutual-exclusion*** scope
  - Only 1 button on at a time
  - 1 on, all others off

```
//Group the radio buttons.
    ButtonGroup group =
  new ButtonGroup();
    group.add(rb1);
    group.add(rb2);
```

JComboBox

Class javax.swing.JComboBox

- public class JComboBox
- extends JComponent
- implements ItemSelectable, ListDataListener,
- **ActionListener**, Accessible

- **JComboBox** cb = new **JComboBox** ();

JComboBox ActionListener

```
patternList.addActionListener(new ActionListener() {
        @Override
        public void actionPerformed(ActionEvent e) {
                JComboBox<String> cb =
(JComboBox<String>) e.getSource();
                String newSelection = (String)
cb.getSelectedItem();
                currentPattern = newSelection;
                reformat(currentPattern);
        }
});
```

JSpinner

Javax.swing.JSpinner

- public class JSpinner
- extends JComponent
- implements Accessible

```
SpinnerModel spinnerModel =
        new SpinnerNumberModel(10, //initial value
          0, //min
          100, //max
          1);//step
    JSpinner spinner = new JSpinner(spinnerModel);
```

Spinner Listener

```
spinner.addChangeListener(new
ChangeListener() {
        public void stateChanged(ChangeEvent e) {
          statusLabel.setText("Value : "
          + ((JSpinner)e.getSource()).getValue());
        }
    });
```

JSlider

Javax.swing.JSlider

- public class JSlider
- extends JComponent
- implements SwingConstants, Accessible

JSlider ChangeListener

```
slider.addChangeListener(new ChangeListener() {
        public void stateChanged(ChangeEvent e) {
           statusLabel.setText("Value : "
           + ((JSlider)e.getSource()).getValue());
        }
     });
```

CSYE 6200

Concepts of Object Oriented Design

Java Threads

Daniel Peters

d.peters@neu.edu

- Lecture:
  1. Java Thread class
  2. Java Runnable interface
  3. Java 8 Executor Pools
  4. synchronized

Thread Class

https://docs.oracle.com/javase/8/docs/api/

java.lang.Thread

- Implements the Runnable interface
  - **Thread** has a NOP '**run()**' method
- Use **Thread** by creating a subclass
  - Derived subclass *overrides* '**run()**' method to provide it's own implementation for **Thread** superclass.

Thread Class example

```
public class MyThread extends Thread {
  public void run() {
    System.out.println("Hello from thread!");
  }
  public static void main(String args[]) {
    (new MyThread()).start(); }
}
```

Runnable Interface

https://docs.oracle.com/javase/8/docs/api/

java.lang.Runnable

Threads, Lambda, Anonymous Inner class can implement Runnable Interface
Better practice: Implementing Runnable (multiple implementation)
than extending Runnable (java does not support multiple inheritance

- Class which implements **Runnable** must implement the '**run()**' method
- Execution of the '**run()**' method occurs in a new thread
- "… the method run may take any action whatsoever."

Runnable Interface example

```
public class MyTask implements Runnable {
  public void run() {
   System.out.println("Hello from thread!");
  }
  public static void main(String args[]) {
   (new Thread(new MyTask())).start();
  }
}
```

Java Multithreading

- Best Practice
  - Implement Runnable interface
    - A Runnable class task is more general
      - Does NOT have to inherit from Thread
      - Can inherit from another Class
      - Java DOES NOT support multiple inheritance
        - » Definition: Directly deriving from two or more classes
        - » Java supports multiple interface implementing
    - All Runnable tasks are executed by a Thread object
      - Thread class 'start()' method "… causes the 'run()' method to be called…" in the newly created and separately executing thread.

Concurrency

- Thread:
  - Fundamental to concurrency
  - Also Known As Lightweight process
  - Every Java application has a main thread

    program execution starts in main thread
  - A Java application executes in a JVM (with all its threads)

    All threads execute in a process - JVM

    Contents of Process:
    Threads
    File Descriptors
    Network Descriptors

Concurrency

- Thread: we don't always have to use Thread Object, we can use thread pool
  - Best Practice: Use executor thread pools
    - List of created thread ready to be borrowed, used, and returned upon completion.
    - Best for short-lived threads
    - Executor façade has factory methods to manage thread pools

Concurrency: Executor

static factory methods

- **Executors.newCachedThreadPool**
  - Creates a thread pool that creates new threads as needed, but will reuse previously constructed threads when they are available.
- **Executors.newFixedThreadPool**
  - Creates a thread pool that reuses a fixed number of threads operating off a shared unbounded queue.

Concurrency: Executor

- **Executors.newScheduledThreadPool**
  - Creates a thread pool that can schedule commands to run after a given delay, or to execute periodically.
- **Executors.newSingleThreadExecutor**
  - Creates an Executor that uses a single worker thread operating off an unbounded queue.

Concurrency: Executor

- **Executors.newSingleThreadScheduledExecutor**
  - Creates a single-threaded executor that can schedule commands to run after a given delay, or to execute periodically.

Concurrency: Executor facade

- Create pool of ***10*** threads:

ExecutorService executor =
Executors.newFixedThreadPool(***10***); // factory

- Use a thread from pool:

executor.execute(() -> System.*out.println("Execute lambda, execute!")); // Lambda Runnable*

executor.execute(System.*out::println("Execute lambda, execute!")); // Lambda method reference*

- Wait for ALL Thread completions:

executor.awaitTermination(2L,TimeUnit.Hours);

Critical Race Condition

Execution Order	Executed Code	Shared Data
1	**Read counter**	0
2	**Increment counter**	
3	**Write counter**	1
4	*Read counter*	1
5	*Increment counter*	
6	*Write counter*	2

- Thread A and *Thread B* **synchronized by chance** results in correct data state of **2**

Critical Race Condition

Execution Order	Executed Code	Shared Data
1	**Read counter**	0
2	*Read counter*	0
3	*Increment counter*	
4	**Increment counter**	
5	**Write counter**	1
6	*Write counter*	1

- **Thread A** and *Thread B* **unsynchronized by chance** results in wrong data state of **1**

Critical Race Condition

Execution Order	Executed Code	Shared Data
1	**Read counter**	0
2	**Increment counter**	
3	*Read counter*	0
4	**Write counter**	1
5	*Increment counter*	
6	*Write counter*	1

- **Thread A** and *Thread B* **unsynchronized by chance** results in wrong data state of **1**

Critical Race Condition

Execution Order	Executed Code	Shared Data
1	*Read counter*	0
2	**Read counter**	0
3	**Increment counter**	
4	**Write counter**	1
5	*Increment counter*	
6	*Write counter*	1

- **Thread A** and *Thread B* **unsynchronized by chance** results in wrong data state of **1**

Critical Race Condition

Execution Order	Executed Code	Shared Data
1	**Read counter**	0
2	*Read counter*	0
3	*Increment counter*	
4	*Write counter*	1
5	**Increment counter**	
6	**Write counter**	1

- **Thread A** and *Thread B* **unsynchronized by chance** results in wrong data state of **1**

Critical Race Condition

Execution Order	Executed Code	Shared Data
1	*Read counter*	0
2	*Increment counter*	
3	*Write counter*	1
4	**Read counter**	1
5	**Increment counter**	
6	**Write counter**	2

- Thread A and *Thread B* **synchronized by design** results in correct data state of **2**

Critical Race Condition

Execution Order	Executed Code	Shared Data
1	**Read counter**	0
2	**Increment counter**	
3	**Write counter**	1
4	*Read counter*	1
5	*Increment counter*	
6	*Write counter*	2

synchronization done using semaphores https://www.geeksforgeeks.org/semaphore-in-java/
Semaphore is used to enforce when one thread starts to execute a block of code
no other thread can execute that same synchronized block of code i.e. every other thread stops & waits until first thread finishes

- Thread A and *Thread B* **synchronized by design** results in correct data state of **2**

Anatomy: Critical Race Condition

need all the conditions for critical race condition

- Concurrent Execution of multiple threads
- Shared unsynchronized mutable data
- Interruptible sequence read then write data
  - Whenever concurrent thread execution order allows for multiple threads to read data BEFORE writing updated data, result will be inconsistent with correct results
  - System loading means unpredictable thread execution order (cannot predict the order of execution)

Solutions: Critical Race Condition

- Don't share mutable data — have separate data classes
  - Object Oriented Design
- Use immutable data — for sharing
  - Functional Programming — it leverages immutable data
- Use atomic (uninterruptable) operations
- Synchronize code execution to simulate atomic operations — since it inherently uses single thread (lowers optimization)
  try minimize code for synchronization - block then function

Use atomic operations: instead of having 3 different steps read, inc, write which is unpredictable -> use one operation to read, inc & write

Concurrency: synchronized block

```
public void performActionBlock() {
    synchronized( this ) {
        // Some implementation in here
        // will be synchronized
    }
}
```

this -> the object that has access & capability to share

Concurrency: synchronized method

```
public synchronized void anyMethod() {
        // All implementation in this method
        // will be synchronized
}
```

- Oracle Java Tutorial
  - Lesson: Concurrency

http://docs.oracle.com/javase/tutorial/essential/concurrency/

CSYE 6200

Concepts of Object Oriented Design

Networking Concepts

Daniel Peters

d.peters@neu.edu

- Lecture
  1. OSI Model
  2. Modes of Communication
  3. UDP - User Datagram Protocol
  4. TCP - Transmission Control Protocol
  5. Application
     1. Distributed Functional Flexibility
     2. Performance Scalability
     3. Design Extensibility

OSI Reference Model

layer 7 to 1 wraps (encapsulates) data from above layer and sends to below layer

- OSI: Open Systems Interconnect
  - Layer 7 Application Layer protocol — only knows the data to send - HTTP, SMTP, File and will send to next layer
  - Layer 6 Presentation Layer protocol — conversion of data into required format
  - Layer 5 Session Layer protocol — sets up authentication - wraps data into session requirements
  - Layer 4 Transport Layer protocol — end-to-end communications
TCP - all packets are sent, received, in correct order
UDP - all packets are sent only, doesn't care about if it has reached or it received in correct order
  - Layer 3: Network Layer protocol — how to connect between 2 nodes (traverses routes)
  - Layer 2: Link Layer protocol — software - link between 2 connections
  - Layer 1: Physical Layer protocol — physical medium electronics: wire, Wi-Fi, cables, rf transmitter & receiver

https://www.cisco.com/c/en/us/support/docs/ip/routing-information-protocol-rip/13769-5.html

OSI Reference Model

- Each layer has a specialized function in the overall network model
- Each layer use the layer below it as a service provider
- Each layer on a local machine (host) virtually communicates with its peer layer on a remote machine (host)

https://study-ccna.com/osi-tcp-ip-models/

OSI Reference Model

- Each layer specialized function in the overall networking
  1. **Physical**: bits are transmitted and received through hardware interfaces, connectors and cables.
  2. **Link**: Frames containing data are sent to the next (intermediate) destination.
  3. **Network**: path of data packets are routed (directed) through network to ultimate destination.
  4. **Transport**: manages overall communication between host and ultimate destination.
  5. **Session**: handles how two entities begin and conclude communications.
  6. **Presentation**: Data formats including compression and encoding (encryption)
  7. **Application**: supports user application to application communications via networking

Network Protocols and OSI Model

- Protocols and their positioning in the OSI Model
  1. **Physical**: IEEE 802 Ethernet
  2. **Link**: Point to Point Protocol (PPP), High-Level Data Link Control (HDLC), Synchronous Optical Network (SONET).
  3. **Network**: Internet Protocol (IP), Internet Control Message Protocol (ICMP)
  4. **Transport**: Transmission Control Protocol (TCP) User Datagram Protocol (UDP).
  5. **Session**: Point-to-Point Tunneling Protocol (PPTP), Layer 2 Tunneling Protocol (L2TP), Remote Procedure Call (RPC).
  6. **Presentation**: Multi-purpose Internet Mail Extension (MIME), eXternal Data Representation (XDR) , Secure Socket Layer (SSL)
  7. **Application**: Hyper Text Transmission Protocol (HTTP)

Physical Communication Channel

- Radio frequency carrier, wires or cabling
- Limited bandwith or wires is shared
  - 2-wire (Signal and ground)
  - Shared bus technology
    - Legacy Ethernet Carrier Sense Media Access with Collision Detection (CSMA/CD)
  - Time Division Multiplexing (TDM)
    - Frame Relay, ATM
  - Frequency Division Multiplexing (FDM)
    - Broadband (cable TV, cable modem)

Modes of Communications

- Simplex communications
- Half Duplex communications
- Full Duplex communications

Modes of Communications

- Simplex
  - One Way (single direction)
  - Single channel (2-wire)
  - Fixed roles:
    - Transmitter always send data
    - Receiver always receive data

Modes of Communications

- Half Duplex
  - Two-Way (both directions)
  - Single channel (2-wire)
  - Dual roles
    - alternate reversable roles over shared channel

Modes of Communications

- Full Duplex
  - Two-Way Simultaneously
  - Dual channel (4-wire)

Simplex, Half-Duplex, Full-Duplex

- Simplex only recive
  - Broadcast Radio Frequency (RF)
    - AM, FM Radio
- Half Duplex send wait receive send wait receive - takes turns
  - Legacy Ethernet (10Base-2, 10Base-5)
  - Two-way Radio
    - HAM Radio
    - Single antenna WiFi (802.11)
- Full Duplex two way send & receive simultaneously
  - 10 Gigabit Ethernet

TCP/IP Protocol suite

- Layer 7-5
  - TCP/IP Applications typically also encompasses Presentation & Session layers:
    - FTP: File Transfer Protocol
    - SMTP: Simple Mail Transport Protocol
    - SNMP: Simple Network Management Protocol
    - Telnet

TCP/IP Protocol suite

- TCP/IP Protocol suite
  - Layer 4 Transport protocols
    - TCP: Transmission Control Protocol
    - UDP: User Datagram Protocol
  - Layer 3: Network protocol
    - IP: Internetwork Protocol

cares about end-to-end communications, checks whether its has received & in correct order - this is the overhead, ex: online banking

just sends data (no overhead), ex: sensor data, voice over ip : skype

TCP/IP Protocol suite

- TCP/IP Protocol suite over LAN
  - Layer 2: Link protocol
    - LLC: Logical Link Control/MAC (Media Access Control)
  - Layer 1: Physical protocol
    - IEEE 802.11 (WLAN PHY)
    - IEEE 802.3 (Ethernet PHY)
      - 10BASE-T, 100BASE-TX, 1000BASE-T
    - IEEE 802.5 MAC (Token Ring PHY)

TCP/IP Protocol suite

- Addressing
  - Host name or IP address
    - localhost
    - 127.0.0.1
  - Protocol each protocol can have same port
    - UDP
    - TCP
  - Protocol Port number
    - Port numbers for UDP protocol
    - Port numbers for TCP protocol
    - UDP port 4045 IS NOT TCP port 4045

So TCP may use port 1,000 & UDP could also use port 1,000. They are different ports because they're on different protocols.

We need all 3 to address in TCP. It's a triple, the host address the protocol being used and the port on that protocol why? - multiple software entities, using the same protocol, they reside and utilize that protocol with a unique port number.

TCP

- Transmission Control Protocol (Transport Layer 4)
  - Client Server
    - Server started first
    - Client initiates connection to Server
  - Connection-Oriented Transport protocol
    - Connect first, then transfer data over connection
  - Reliable Ordered Stream delivery
    - Error detection, retransmission as needed
    - Guaranteed ordered packet delivery in stream of data

Server is started first.
When client initiates server must be ready to receive.

TCP

- 24 Byte TCP Packet Header (32-bit words)
  - Source Port:16, Destination Port:16
  - Sequence number:32
  - Acknowledment number:32
  - Header Length:16, reserved:16
  - Flags:16, Window:16
  - Checksum:16 (header and data), Urgent:16
  - Options

UDP

- User Datagram Protocol (Transport Layer 4)
  - Client Server
    - Server started first
    - Client sends data to Server
  - Connection-Less Transport protocol
    - Fast, no connection overhead
  - Un-ordered delivery
    - Not a stream, must implement own record boundaries across datagrams
    - Each packet individually addressed and routed
  - Used by Voice over IP (VoIP)

UDP

- 8 Byte UDP Datagram Header (16-bit words)
  - Source Port:16
  - Destination Port:16
  - Length:16
  - Checksum:16 (header and data)

IP

- Internetwork Protocol (Network Layer 3)
- IPv4 addressing: 32-bit address
  - Class A: Network:8, Host:24
  - Class B: Network:16, Host:16
  - Class C: Network:24, Host:8
  - Local Host address: 127.0.0.1
  - Broadcast address: 255.255.255.255
  - Multicast address: 254.x.x.x
- IPv6 addressing: 128-bit address

IP

- Internetwork Protocol (Network Layer 3)
- 24 Byte IPv4 Packet Header (32-bit words)
  - Version:4, Header Length :4, Priority ToS:8, Total Packet Length:16
  - ID:16, Flags:3, Fragment Offset:13
  - TTL:8, Protocol:8, Header Checksum:16
  - Source IP Address:32
  - Destination IP Address:32
  - Options:32

CSYE 6200

Concepts of Object Oriented Design

Java Network Programming

Daniel Peters

d.peters@neu.edu

- Lecture
  1. Java UDP - User Datagram Protocol
  2. Java TCP - Transmission Control Protocol

Java UDP

- java.net
  - **DatagramSocket**
    - Used to obtain UDP socket endpoint
    - Example
- **DatagramSocket** socket = **new DatagramSocket**();

Java UDP

- java.net
  - **DatagramSocket**
    - Datagram socket endpoint
    - Counterpart to Socket for UDP Datagram packets
    - Example:

```
int port = 4545;
DatagramSocket socket = new
  DatagramSocket(port);
```

Java UDP

- java.net
  - **InetAddress**
    - Used to obtain IP address of host
    - Example

ipAddress = **InetAddress**.*getByName*(host);

Java UDP

- java.net
  - **DatagramPacket**
    - Create Datagram for UDP data
    - Example

byte[] *buf* = messageString.getBytes();

sendPacket = new **DatagramPacket**(*buf*, buf.length, ipAddress, port);

socket.***send***(this.sendPacket);

Java UDP

- java.net
  - **DatagramPacket**
    - Used to send and receive Datagrams
    - Example

```
byte[] buf = new byte[DEFAULT_BUF_SIZE];
DatagramPacket rPacket = new
  DatagramPacket(buf, buf.length);
socket.receive(rPacket);
```

UDP Client

```
import java.io.BufferedReader;
import java.io.InputStreamReader;
import java.net.DatagramPacket;
import java.net.DatagramSocket;
import java.net.InetAddress;

class UDPClient {
        public static void main(String args[]) throws Exception {
                BufferedReader inFromUser = new BufferedReader(new InputStreamReader(System.in));
                DatagramSocket clientSocket = new DatagramSocket();
                InetAddress IPAddress = InetAddress.getByName("localhost");
                byte[] sendData = new byte[1024];
                byte[] receiveData = new byte[1024];
                String sentence = inFromUser.readLine();
                sendData = sentence.getBytes();
                DatagramPacket sendPacket = new DatagramPacket(sendData, sendData.length, IPAddress, 9876);
                clientSocket.send(sendPacket);
                DatagramPacket receivePacket = new DatagramPacket(receiveData, receiveData.length);
                clientSocket.receive(receivePacket);
                String modifiedSentence = new String(receivePacket.getData());
                System.out.println("FROM SERVER:" + modifiedSentence);
                clientSocket.close();
        }
}
```

Read User Data from Standard Input

```
import java.io.BufferedReader;
import java.io.InputStreamReader;
import java.net.DatagramPacket;
import java.net.DatagramSocket;
import java.net.InetAddress;
class UDPClient {
        public static void main(String args[]) throws Exception {
        BufferedReader inFromUser = new BufferedReader(new
InputStreamReader(System.in));
         . . .
        }
}
```

Datagram socket endpoint

```
class UDPClient {
        public static void main(String args[]) throws
Exception {
 . . .
                DatagramSocket clientSocket = new
DatagramSocket();
 . . .
                clientSocket.close();
        }
}
```

Datagram Send

```
class UDPClient {
        public static void main(String args[]) throws Exception {
 . . .
        InetAddress IPAddress =
InetAddress.getByName("localhost");
        byte[] sendData = new byte[1024];
        byte[] receiveData = new byte[1024];
        String sentence = inFromUser.readLine();
        sendData = sentence.getBytes();
        DatagramPacket sendPacket = new
DatagramPacket(sendData, sendData.length, IPAddress,
9876);
        clientSocket.send(sendPacket);
```

Datagram Receive

```
class UDPClient {
        public static void main(String args[]) throws Exception {
. . .
        DatagramPacket receivePacket = new
DatagramPacket(receiveData, receiveData.length);
        clientSocket.receive(receivePacket);
        String modifiedSentence = new
String(receivePacket.getData());
        System.out.println("FROM SERVER:" +
modifiedSentence);
        clientSocket.close();
        }
}
```

UDP Server

```
import java.net.DatagramPacket;
import java.net.DatagramSocket;
import java.net.InetAddress;

class UDPServer {
        public static void main(String args[]) throws Exception {
        DatagramSocket serverSocket = new DatagramSocket(9876);
        byte[] receiveData = new byte[1024];
        byte[] sendData = new byte[1024];
        while (true) {
        DatagramPacket receivePacket = new DatagramPacket(receiveData, receiveData.length);
        serverSocket.receive(receivePacket);
        String sentence = new String(receivePacket.getData());
        System.out.println("RECEIVED: " + sentence);
        InetAddress IPAddress = receivePacket.getAddress();
        int port = receivePacket.getPort();
        String capitalizedSentence = sentence.toUpperCase();
        sendData = capitalizedSentence.getBytes();
        DatagramPacket sendPacket = new DatagramPacket(sendData, sendData.length, IPAddress, port);
        serverSocket.send(sendPacket);
        }
        }
}
```

UDP Server

```
import java.net.DatagramPacket;
import java.net.DatagramSocket;
import java.net.InetAddress;
class UDPServer {
        public static void main(String args[]) throws Exception {
        DatagramSocket serverSocket = new
DatagramSocket(9876);
 . . .
        serverSocket.send(sendPacket);
        }
        }
}
```

UDP Server

```
import java.net.DatagramPacket;
import java.net.DatagramSocket;
import java.net.InetAddress;

class UDPServer {
  public static void main(String args[]) throws Exception {

 . . .
     byte[] receiveData = new byte[1024];
     byte[] sendData = new byte[1024];
     while (true) {
        DatagramPacket receivePacket = new
DatagramPacket(receiveData, receiveData.length);
        serverSocket.receive(receivePacket);
```

UDP Server

```
        while (true) {
           DatagramPacket receivePacket = new
              DatagramPacket(receiveData, receiveData.length);
           serverSocket.receive(receivePacket);
           String sentence = new String(receivePacket.getData());
           System.out.println("RECEIVED: " + sentence);
. . .
           }
        }
}
```

UDP Server

```
        while (true) {
. . .
            InetAddress iPAddress = receivePacket.getAddress();
            int port = receivePacket.getPort();
            String capitalizedSentence = sentence.toUpperCase();
            sendData = capitalizedSentence.getBytes();
            DatagramPacket sendPacket = new
               DatagramPacket(sendData, sendData.length,
                  iPAddress, port);
            serverSocket.send(sendPacket);
        }
    }
}
```

Java TCP

- java.net
  - **Socket**
    - socket endpoint
    - Used to connect and transfer data packets
    - Example:

```
String serverName = “localhost”;
int port = 6066;
Socket client = new Socket(serverName, port); // connect
System.out.println("Just connected to ”
    + client.getRemoteSocketAddress());
```

Java TCP

- Stream based data transfer

```
OutputStream outToServer = client.getOutputStream();
DataOutputStream out =
   new DataOutputStream(outToServer);
out.writeUTF("Hello from ”
     + client.getLocalSocketAddress() + data);

InputStream inFromServer = client.getInputStream();
DataInputStream in = new DataInputStream(inFromServer);
System.out.println("Server says " + in.readUTF());
client.close();
```

Java TCP

- java.net
  - **ServerSocket**
    - Used to accept TCP connections
    - Example

```
int port = 6066;
ServerSocket serverSocket = new ServerSocket(port);
serverSocket.setSoTimeout(10000);
Socket server = serverSocket.accept();  // socket endpoint
System.out.println("Just connected to "
      + server.getRemoteSocketAddress());
```

Java TCP

- Stream based data transfer

```
DataInputStream in =
   new DataInputStream(server.getInputStream());
System.out.println(in.readUTF());

DataOutputStream out =
   new DataOutputStream(server.getOutputStream());
out.writeUTF("Thank you for connecting to "
   + server.getLocalSocketAddress() + "\nBye!");
server.close();
```