Welcome to CS681!

Tue Thu 5:30pm to 6:45pm

S-1-006

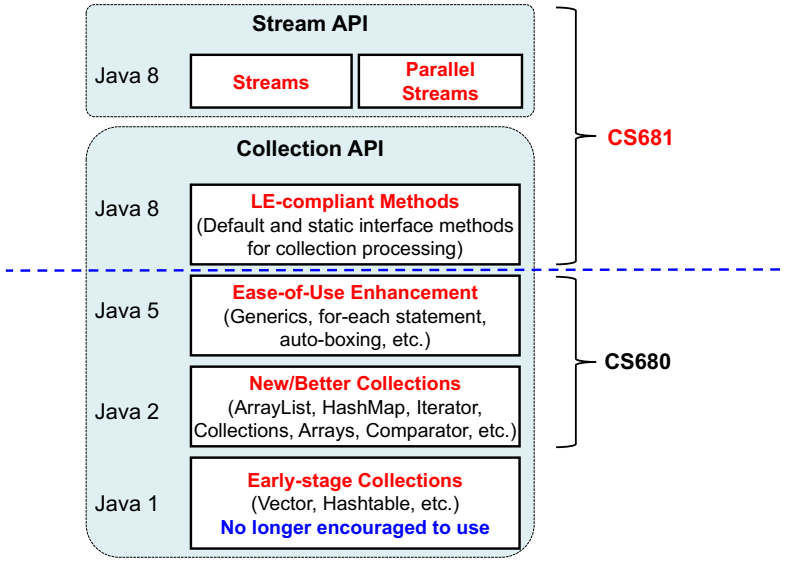
Course Topics: Advanced Software Engineering

- Language/library enhancements in Java 7 and 8
- Concurrent programming (multi-threading) with Java

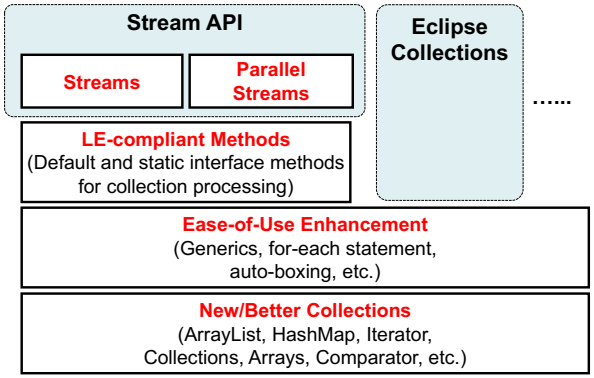
Course Topics # 1

- Language/library enhancements in Java 7 and 8
  - File handling with NIO (New I/O) API, try-with-resources statement, etc.
  - Functional programming with lambda expressions
    - Continuation from CS680
    - Collection processing with LEs
      - Newly-added default and static interface methods to process collections with LEs
      - Stream API, which heavily uses LEs

Collection and Stream APIs in Java



Extra Utilities



- Eclipse Collections
  - https://www.eclipse.org/collections/
  - https://github.com/eclipse/eclipse-collections-kata
  - Used to be Goldman Sachs (GS) Collections
    - https://github.com/goldmansachs/gs-collections

Course Topics # 2

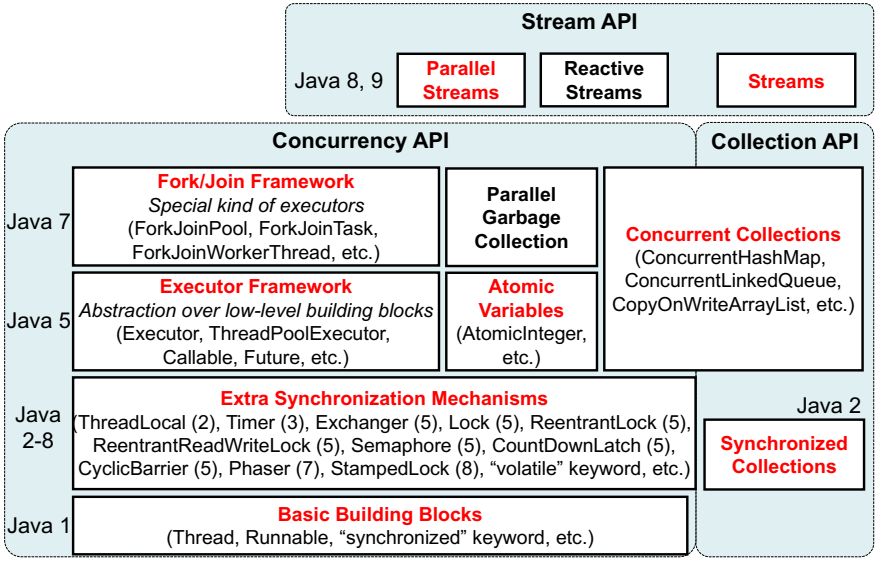
- Concurrent programming (multi-threading) with Java
  - Mechanisms, data structures, libraries and frameworks for concurrency (multi-threading)
  - Concurrent object-oriented design patterns
    - e.g., MapReduce, Producer-Consumer
  - Concurrency with LEs (incl. parallel streams)
  - Network programming
    - Socket programming with TCP/IP, I/O techniques, etc.

Concurrency as a Part of SE? Yes!

- Concurrency was for special kinds of software engineers to develop special kinds of software.
  - Concurrency in programming: Since mid 60s
    - PL/I ("TASK" statement, '65), Concurrent Pascal ('75), Concurrent Smalltalk ('86), Objective-C ('86), Java ('96) , … C# ('02), Erlang ('06), etc.
  - Concurrency in OSes: Since 70s
    - Multix ('70), Mach (cthreads, '85) , NeXTSTEP (cthreads, '86), OS/2 ('87), Solaris libthread ('93), Windows NT ('94), pthreads (POSIX threads '95), Windows ('95), Mac OS X ('01), Linux ('03), iOS, Android, etc.
  - Early applications
    - Super computing and real-time computing in defense, aviation, financial trading (algorithmic and high freq. trades), etc.

- Concurrency is critical/important/useful for many software engineers to develop many kinds of software.
  - Not only "special" applications but also "normal" apps can enjoy, or even require, concurrency.
    - e.g., Web apps, smartphone/tablet apps
  - Goals: ***responsiveness*** and ***performance*** improvement
    - Better I/O handling
    - Multi-core CPUs

Concurrency API in Java



Course Work

- Lectures
  - Sometimes a quiz at the beginning of a lecture
- Homework
  - Reading
  - Programming exercises
    - Submit your HW solutions to **umasscs681@gmail.com**.
  - Individual project
- Send your questions to jxs@cs.umb.edu

Grading

- Grading factors
  - Homework (65-75%)
  - Individual project (25%)
  - Quiz (0-10%)
- No textbooks
  - Look up APIs with Java API documentation as often as possible.
- No exams

Language/Library Enhancements in Java 7

Java Versioning

- Java 7 ~ JDK 1.7
- Java 8 ~ JDK 1.8
  - More precisely, JDK 1.7 and 1.8 implement Java 7's and 8's language specifications, respectively.
    - `> java -version`
      `java version "1.8.0_xx"`
      `Java (TM) SE Runtime Environment (build 1.8.0_xx)`
      `Java HotSpot (TM) 64-Bit Server VM (build…..)`
  - Download and set up JDK 1.8 in your machine.

Notable Enhancements in Java 7

- File handling
  - **`Path`**, **`Paths`** and **`Files`** in the NIO (New I/O) API
- *Try-with-resources* statement and **`AutoCloseable`**
- Null checking with **`java.util.Objects`**
- Type inference with the diamond (<>) operator.

(1) Dealing with File/Directory Paths in NIO

- **`java.nio.Paths`**
  - A utility class (i.e., a set of static methods) to create a path in the file system.
    - Path: A sequence of directory names
      - Optionally with a file name in the end.
  - A path can be *absolute* or *relative*.
    - `Path absolute = Paths.get("/Users/jxs/temp/test.txt");`
    - `Path relative = Paths.get("temp/test.txt");`
- **`java.nio.Path`**
  - Represents a path in the file system.
  - Given a path, *resolve* (or determine) another path.
    - `Path absolute = Paths.get("/Users/jxs/");`
      `Path another = absolute.resolve("temp/test.txt");`
    - `Path relative = Paths.get("src");`
      `Path another = relative.resolveSibling("bin");`

Just in Case: Passing a Variable # of Parameters to a Method

- **`Paths.get()`** can receive a variable number of parameter values (1 to many values)
  - c.f. Java API documentation
  - **`Paths.get(String first, String... more)`**
    - `Paths.get("temp/test.txt");` // relative path
    - `Paths.get("temp", "test.txt");` // relative path
    - `Paths.get("/", "Users", "jxs");` // absolute path
  - **`String... More`** → Can receive zero to many String values.
  - Introduced in Java 5 (JDK 1.5)

- A method handles parameter values with an array.
  - ```
    class Foo{
        public void varParamMethod(String... strings){
            for(int i = 0; i < strings.length; i++){
                System.out.println(strings[i]); } } }
    ```
  - ```
    Foo foo = new Foo();
    foo.varParamMethod("U", "M", "B");
    ```
- **`String... Strings`** is a syntactic sugar for **`String[] strings`**.
  - Your Java compiler transforms the above code to:
    - ```
      class Foo{
          public void varParamMethod(String[] Strings){
              for(int i = 0; i < strings.length; i++){
                  System.out.println(strings[i]); } } }
      ```
    - ```
      Foo foo = new Foo();
      String[] strs = {"U", "M", "B"};
      foo.varParamMethod(strs);
      ```


Reading and Writing into a File w/ NIO

- **`java.nio.file.Files`**
  - A utility class (i.e., a set of static methods) to process a file/directory.
  - Reading a byte sequence and a char sequence from a file
    - ```
      Path path = Paths.get("/Users/jxs/temp/test.txt");
      byte[] bytes = Files.readAllBytes(path);
      String content = new String(bytes);
      ```
    - ```
      List<String> lines = Files.readAllLines(path);
      for(String line: lines){
            System.out.println(line); }
      ```
  - Writing into a file
    - `Files.write(path, bytes);`
    - `Files.write(path, content.getBytes());`
    - `Files.write(path, bytes, StandardOpenOption.CREATE);`
    - `Files.write(path, lines);`
    - `Files.write(path, lines, StandardOpenOption.WRITE);`
    - `StandardOpenOption: CREATE, WRITE, APPEND, DELETE_ON_CLOSE, etc.`

NIO (java.nio) v.s. Traditional I/O (java.io)

- NIO provides simpler APIs.
  - Client code can be more concise and easier to understand.
- NIO:
  - ```
    Path path = Paths.get("/Users/jxs/temp/test.txt");
    byte[] bytes = Files.readAllBytes(path);
    String content = new String(bytes);
    ```
- java.io:
  - ```
    File file = ...;
    FileInputStream fis = new FileInputStream(file);
    int len = (int)file.length();
    byte[] bytes = new byte[len];
    fis.read(bytes);
    fis.close();
    String content = new String(bytes);
    ```


NIO (java.nio) v.s. Traditional I/O (java.io)

- NIO:
  - ```
    Path path = Paths.get("/Users/jxs/temp/test.txt");
    List<String> lines = Files.readAllLines(path);
    ```
- java.io:
  - ```
    int ch=-1, i=0;
    ArrayList<String> contents = new ArrayList<String>();
    StringBuffer strBuff = new StringBuffer();
    File file = ...;
    InputStreamReader reader = new InputStreamReader(
                                        new FileInputStream(file));
    while( (ch=reader.read()) != -1 ){
        if( (char)ch == '\n' ){       //**line break detection
            contents.add(i, strBuff.toString());
            strBuff.delete(0, strBuff.length());
            i++;
            continue;
        }
        strBuff.append((char)ch);
    }
    reader.close();
    ```

** The perfect (platform independent) detection of a line break should be more complex.
Unix: '\n', Mac: '\r', Windows: '\r\n' c.f. BufferedReader.read()

NIO (java.nio) v.s. Traditional I/O (java.io)

- NIO:
  - ```
    Path path = Paths.get("/Users/jxs/temp/test.txt");
    List<String> lines = Files.readAllLines(path);
    ```
- java.io (a bit simplified version):
  - ```
    int ch=-1, i=0;
    ArrayList<String> contents = new ArrayList<String>();
    StringBuffer strBuff = new StringBuffer();
    File file = ...;
    FileReader reader = new FileReader(file); //***
    while( (ch=reader.read()) != -1 ){
        if( (char)ch == '\n' ){  //** Line break detection
            contents.add(i, strBuff.toString());
            strBuff.delete(0, strBuff.length());
            i++;
            continue;
        }
        strBuff.append((char)ch);
    }
    reader.close();
    ```

*** FileReader: A convenience class for reading character files.

Files in Java NIO

- readAllBytes(), readAllLines()
  - Read the whole data from a file without buffering.
- write()
  - Write a set of data to a file without buffering.
- When using a large file, it makes sense to use `BufferedReader` and `BufferedWriter` with `Files`.
  - ```
    Path path = Paths.get("/Users/jxs/temp/test.txt");
    BufferedReader reader = Files.newBufferedReader(path);
    while( (line=reader.readLine()) != null ){
        // do something
    }
    reader.close();
    ```
  - ```
    BufferedWriter writer = Files.newBufferedWriter(path);
    writer.write(...);
    writer.close();
    ```


Just in case: Buffering

- At the lowest level, read/write operations deal with data *byte by byte*, or *char by char*.
  - File access occurs *byte by byte*, or *char by char*.
- Inefficient if you read/write a lot of data.
- Buffering allows read/write operations to deal with data in a coarse-grained manner.
  - Chunk by chunk, not byte by byte or char by char
  - Chunk = a set of bytes or a set of chars
    - The size of a chunk: 512 bytes by default, but configurable

Getting Input/Output Streams from `Files`

- Input and output streams can be obtained from `Files`.
  - ```
    Path path = Paths.get("/Users/jxs/temp/test.txt");
    InputStream is = Files.newInputStream(path);
    ```
    - `is` contains an instance of ChannelInputStream, which is a subclass of InputStream.
    - Make sure to call `is.close()` in the end.
- Can decorate the input/output stream with filters.
  - ```
    ZipInputStream zis = new ZipInputStream(
                                    Files.newInputStream(path) );
    ```
    - Make sure to call `zis.close()` in the end.

Never Forget to Call close()

- Need to call `close()` on each input/output stream (or its filer) in the end.
  - Must-do: Follow the *Before/After* design pattern.
    - In Java, use a *try-catch-finally* or *try-finally* statement.

```
» Open a file here.
  try{
      Do something with the file here.
  }catch(...){
      Handle errors here.
  }finally{
      Close the file here.
  }
```

  - Note: No need to call `close()` when using `readAllBytes()`, `readAllLines()` and `write()` of `Files`.


```
– Path path = Paths.get("/Users/jxs/temp/test.txt");
  BufferedReader reader = Files.newBufferedReader(path);
  try{
    while( (line=reader.readLine()) != null ){
        // do something
    }
  }catch(IOException ex){
    ... // Error handling
  }finally{
    reader.close();
  }
```


(2) Try-with-resources Statement

- Allows you to skip calling close() explicitly in the finally block.
  - *Try-catch-finally*

```
– Open a file here.
  try{
      Do something with the file here.
  }catch(...){
      Handle errors here.
  }finally{
      Close the file here.
  }
```

  - *Try-with-resources*

```
• try ( Open a file here ){
      Do something with the file here.
  }
```


- `close()` is automatically called on a resource used for reading or writing to a file, when exiting a try block.

```
• try( BufferedReader reader =
          Files.newBufferedReader( Paths.get("test.txt")) ){
      while( (line=reader.readLine()) != null ){
          // do something }
  }
```

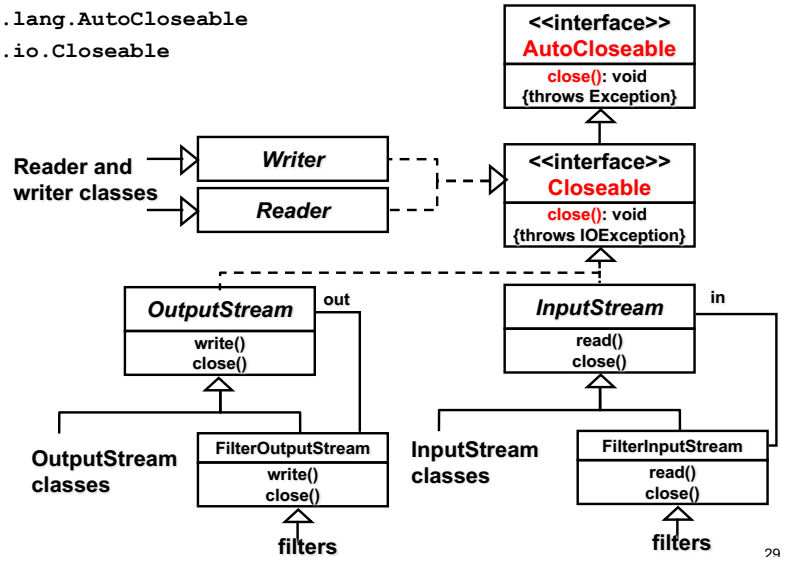
  - No explicit call of close() on `reader` in the finally block. `reader` is expected to implement the `AutoCloseable` interface.

```
• try( BufferedReader reader = Files.newBufferedReader(...);
       PrintWriter writer = new PrintWriter(...) ){
      while( (line=reader.readLine()) != null ){
          // do something
          writer.println(...); }
  }
```

  - Can specify multiple resources in a try block. close() is automatically called on all of them. They all need to implement `AutoCloseable`.

AutoCloseable Interface

- `java.lang.AutoCloseable`
- `java.io.Closeable`



- Recap: No need to call `close()` when using `readAllBytes()`, `readAllLines()` and `write()` of `Files`.
  - Those methods use the try-with-resources statement to read and write to a file.

Try-with-resources-Catch-Finally

- Catch and finally blocks can be attached to a try-with-resources statement.

```
try( BufferedReader reader =
        Files.newBufferedReader( Paths.get("test.txt")) ){
    while( (line=reader.readLine()) != null ){
        // do something. This part may throw an exception.}
}catch(...){
    //This block runs if the try block throws an exception.
}finally{
    ...
    //No need to do reader.close() here.
}
```

  - The catch and finally blocks run (if necessary) AFTER close() is called on `reader`.

(3) Null Checking with Objects

- `java.util.Objects`, extending `java.lang.Object`
  - A utility class (i.e., a set of static methods) for the instances of `java.lang.Object` and its subclasses.

```
class Foo{
    private String str;
    public Foo()(String str){
        this.str = Objects.requireNonNull(str);
    } }
```

  - `requireNonNull()` throws a `NullPointerException` if `str==null`. Otherwise, it simply returns `str`.

```
class Foo{
    private String str;
    public Foo()(String str){
        this.str = Objects.requireNonNull(
                        str, "str must be non-null!!!");
    } }
```

  - `requireNonNull()` can accept an error message, which is to be contained in a `NullPointerException`.

- Traditional null checking
  - ```
    if(str == null)
        throw new NullPointerException();
    this.str = str;
    ```
- With `Objects.requireNonNull()`
  - ```
    this.str = Objects.requireNonNull(str);
    ```
- Can eliminate an explicit conditional statement and make your code simpler.

(4) Type Inference

- Can be used to create a collection object with generics.
- Before Java 7:
  - ```
    ArrayList<String> list = new ArrayList<String>();
    ```
- Since Java 7:
  - ```
    ArrayList<String> list = new ArrayList<>();
    ```
    - No need to repeat "ArrayList<String>"
    - Code looks less redundant/dumb.
    - <>: Diamond operator
  - ```
    HashMap<String, ArrayList<Foo>> map = new HashMap<>();
    ```


- Type inference also works when calling a constructor in another method call.
  - Before Java 7:
    - ```
      private void foo(HashMap<String, String> values);
      ```
    - ```
      ...
      foo( new HashMap<String, String>() );
      ```
  - Since Java 7:
    - ```
      private void foo(HashMap<String, String> values);
      ```
    - ```
      ...
      foo( new HashMap<>() );
      ```
- This looks like a simple coding trick, but it was an important step for further type inference mechanisms introduced in Java 8.

Language/Library Enhancements in Java 8
      ```

Notable Enhancements in Java 8

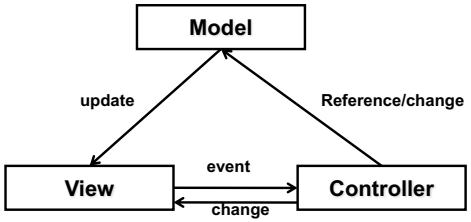
- Lambda expressions
  - Allow you to do *functional programming* in Java
- Static and default methods in interfaces
- See CS680 lecture notes and refresh your memory about LEs.

Benefits of Using Lambda Expressions

- Can make your code more concise (less repetitive)
  - *Callback* functions/methods as lambda expressions
    - e.g., Comparator, GUI, event handling (e.g., Observer)
- Can enjoy the power of functional programming
  - e.g., higher-order functions
- Can gain a new way to access collections
  - Newly-added *default methods* that accept lambda expressions
  - "Internal" iteration as opposed to traditional "external" iteration
  - Collection streams
    - Enables Map-Reduce data processing
- Can simplify concurrent programming (multi-threading)
  - Repeatedly and concurrently executed code (block) as a lambda expression

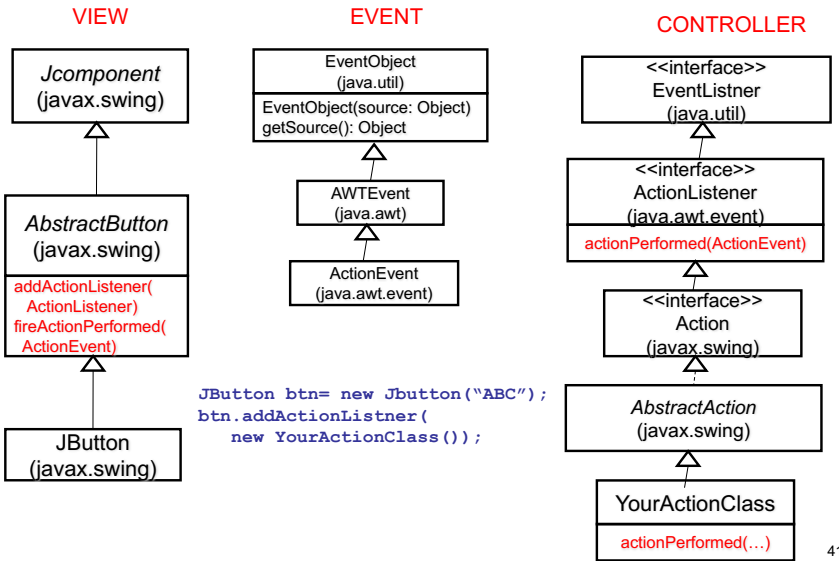
Model-View-Controller (MVC) Architecture

- An architectural design strategy (or architectural pattern) to separate…
  - **Model**
    - Maintains data or primary business logic
  - **View**
    - Deals with UI for a model.
    - Visualization/representation of a model to the user
  - **Controller**
    - Processes inputs/changes from the user to a model.
    - Update the model with the inputs/changes and update its corresponding view.

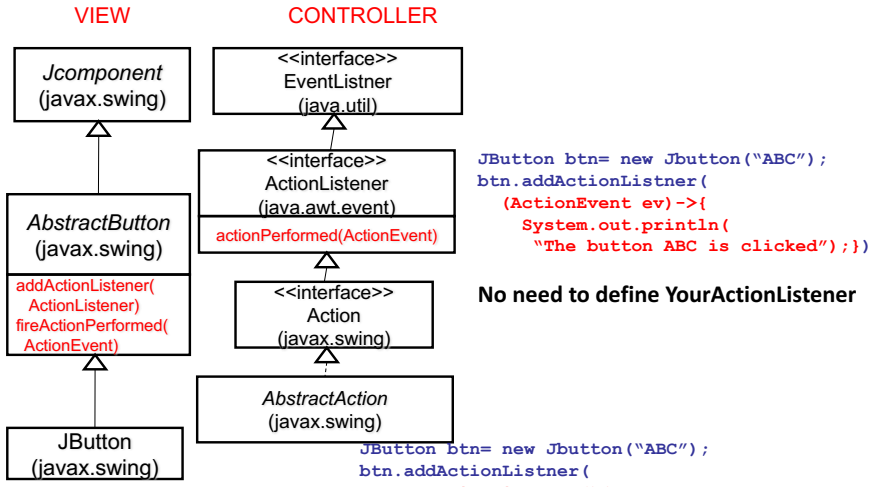


- MVC uses several design patterns internally.
  - Observer, Multicast (typed Observer), etc.
- Many (too many) examples/applications
  - Swing, Struts, Android, Flex, etc. etc., etc.

View and Controller in Swing



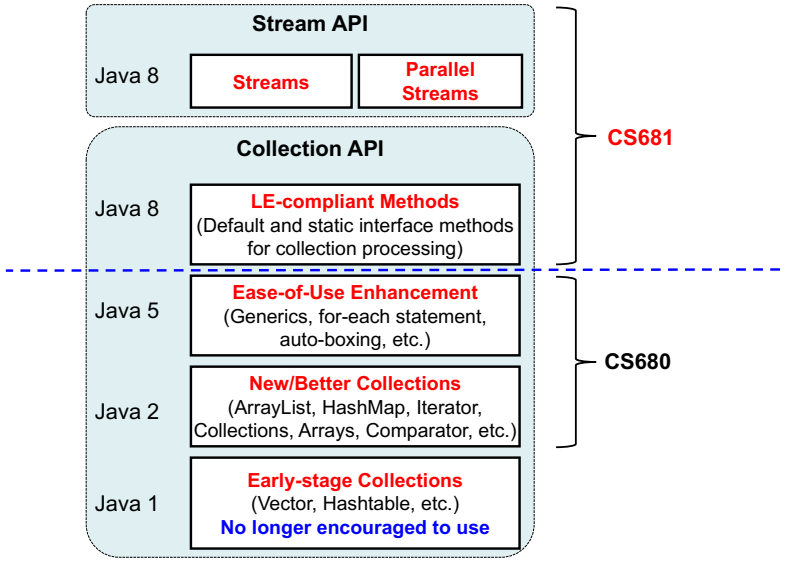
Lambda Expressions in Swing



Benefits of Using Lambda Expressions

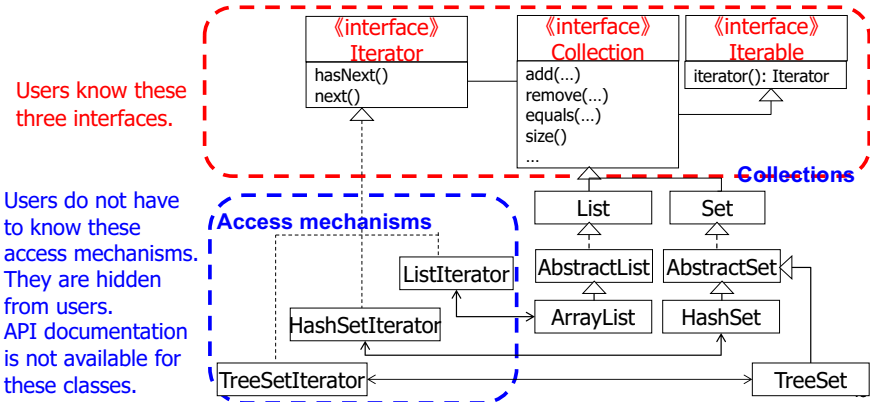
- Can make your code more concise (less repetitive)
  - *Callback* functions/operations as lambda expressions
    - e.g., Comparator, GUI, event handling (e.g., Observer)
- Can enjoy the power of functional programming
  - e.g., higher-order functions
- Can gain a new way to access collections
  - Newly-added *default methods* that accept lambda expressions
  - "Internal" iteration as opposed to traditional "external" iteration
  - Collection streams
    - Enables Map-Reduce data processing
- Can simplify concurrent programming (multi-threading)
  - Repeatedly and concurrently executed code (block) as a lambda expression

Collection and Stream APIs in Java



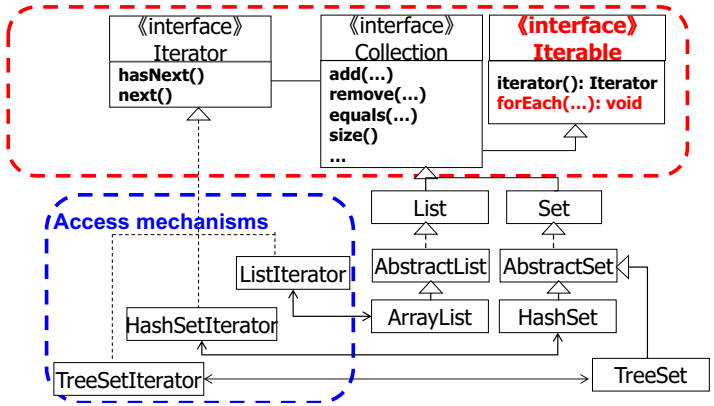
Lambda Expressions for Collections

- **`java.lang.Iterable<T>`**
  - Used to have only one (abstract) method before Java 8
    - **`Iterator<T> iterator()`**
  - c.f. CS680 lecture note



- Before Java 8…
  -
```
ArrayList<String> strList = new ArrayList<>();
strList.add("a"); strList.add("b");
Iterator<ArrayList> iterator = strList.iterator();
while( iterator.hasNext() ) {
  System.out.print(iterator.next()); }
```
  -
```
ArrayList<String> strList = new ArrayList<>();
strList.add("a"); strList.add("b");
for(String str: strList){
    System.out.println(str) }
```
    - Note: "for-each" is a syntactic sugar for iterator-based code.

- **`Iterable<T>`**
  - Java 8 introduced new *default* methods.
    - **`default void forEach(Consumer<T> action)`**
      - Performs a given action for each element of a collection that implements **`Iterable`**.



- **`Iterable<T>`**
  - **`default void forEach(Consumer<T> action)`**

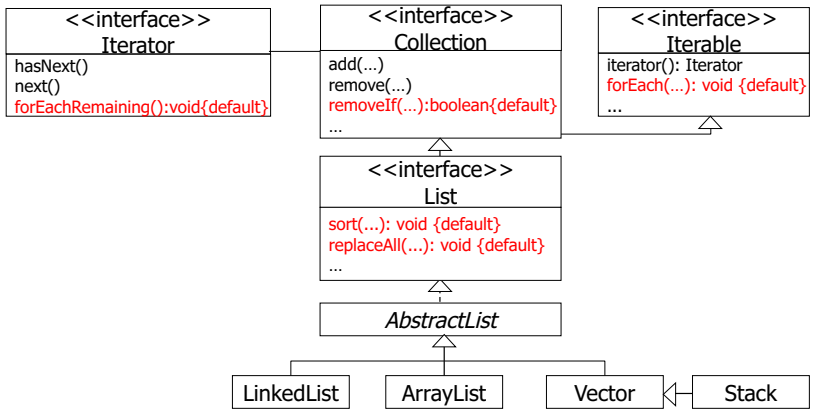
- **`Consumer<T>`**: Functional interface
  - Allows **`forEach()`** to receive a lambda expression.
  - Represents a function (LE) that accepts a parameter (T) and returns no result.
    - When the function is used for **`Iterable.forEach()`**, the LE receives a collection element as a parameter and specifies an action to be applied to the collection element as its code block.
  - Abstract method: **`void accept(T t)`**
    - Performs the function (i.e., the function that **`Consumer<T>`** represents) on a given parameter **`t`**.

-
```
ArrayList<String> strList = new ArrayList<>();
strList.add("a"); strList.add("b");
strList.forEach( (String s)->System.out.println(s) );
```

- Without a lambda expression
  - `Iterator<ArrayList> iterator = strList.iterator();`
    `while( iterator.hasNext() ) {`
    `System.out.print(iterator.next()); }`
  - `for(String str: strList){`
    `System.out.println(str) }`
- With a lambda expression
  - `strList.forEach( (Integer i)->System.out.println(i) )`
    - `strList.forEach( System.out::println )`

New Default Methods for Lists

- Default methods have been added to various interfaces for lists.
  - Accept lambda expressions




```
ArrayList<String> list = new ArrayList(
                                Arrays.asList(
                                  "Yahoo","Yahooo","Yahoooo"));
// Print out each element
list.forEach( (String s)-> System.out.println(s) );
list.forEach(System.out::println);

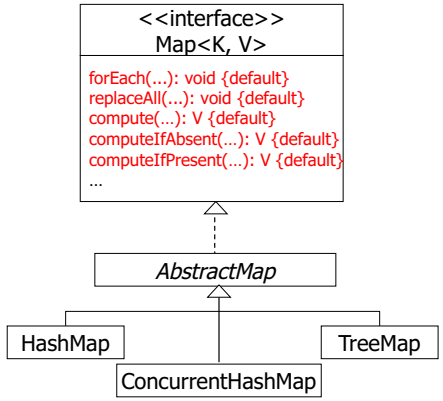
// Sort elements based on the length of each element (descending order)
// Result: Yahoooo, Yahooo, Yahoo
list.sort( (String s1, String s2)-> s2.length()-s1.length() );
list.sort( Comparator.comparing( (String s)-> s.length(),
                                 Comparator.reverseOrder() );
list.sort( Comparator.comparing(String::length).reversed());

// Replace each element with the one returned by a given lambda expression
// Result: YAHOOOO, YAHOOO, YAHOO
list.replaceAll( (String s)-> s.toUpperCase() );
list.replaceAll(String::toUpperCase);

// Remove every element that matches a criterion defined in a given lambda expression
// Result: YAHOOO, YAHOO
list.removeIf( (String s)-> s.endsWith("OOOO") );
```


New Default Methods for Maps

- Default methods have been added to `java.util.Map<K, V>`
  - Accept lambda expressions



- **`forEach(LE)`**
  - Perform an action, which is defined as a given lambda expression, on each element.
  -
```
HashMap<String,Integer> map = new HashMap<>();
map.put("A",1); map.put("B",2); map.put("C",3);

// Print out each element
// Result: A=1, B=2, C=3
map.forEach((String key,Integer val)->
                System.out.println(key + "=" + val));
```

- **`replaceAll(LE)`**
  - Replace each element with the one returned by a given lambda expression
  -
```
// Result: A=10, B=20, C=30
map.replaceAll((String key,Integer val)-> val*10);
```

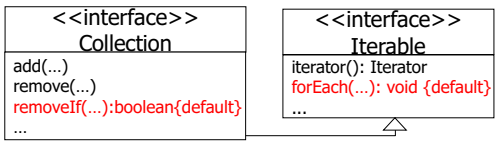
- **`compute(key, LE)`**
  - Pair a key with a value that a given lambda expression returns and add the key-value pair.
  -
```
// Result: A=1, B=20, C=30
map.compute("A", (String key,Integer val)->{
                        if(val==null){returns 0;}
                        else{returns val/10;} });
```


- **`computeIfAbsent(key, LE)`**
  - Pair a key with a value that a given lambda expression returns, ONLY IF the key does not exist, and add the key-value pair.
  -
```
// Result: A=1, B=20, C=30, D=4
map.computeIfAbsent("D", (String key)-> 4);
```

- **`computeIfPresent(key, LE)`**
  - Pair a key with a value that a given lambda expression returns, ONLY IF the key does exist, and replace an existing key-value pair with the new pair.
  -
```
// Result: A=100, B=20, C=30, D=4
map.computeIfPresent("A", (String key,Integer val)->
val*100);
```

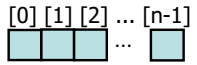

Benefits of Default Methods

- Useful for API designers to add extra methods to existing interfaces
  - Without worrying about backward compatibility.
  - What if Oracle defined **`forEach()`** as an abstract method rather than a default method?

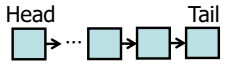


Just in Case: Major Collection Types in Java

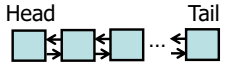
- List
  - Orders elements with their integer index.
  - Offers index-based random access.
  - Can contain duplicate elements.



- Queue
  - Orders elements with links.
  - Offers FIFO (First-In-First-Out) access.
  - Can contain duplicate elements.



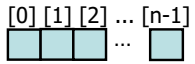
- Dequeue
  - Stands for "Double Ended QUEUE" (pronounced "deck").
  - Orders elements with links.
  - Offers both FIFO and LIFO (Last-In-First-Out) access.
  - Can contain duplicate elements.



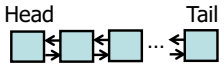
- Set
  - Contains non-duplicate elements without an order.
- Map
  - Contains key-value pairs (n duplicate keys) without an order.

Just in case: ArrayList v.s. LinkedList

- ArrayList
  - Array-based implementation of the List interface
  - Fast indexed access
  - Slow insertion and removal of non-tail elements
    - Fast insertion of the tail element

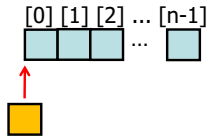


- LinkedList
  - Doubly-linked implementation of the List and Deque interfaces
  - Fast insertion and removal of non-tail elements
  - Slow "indexed" access for "middle" elements.

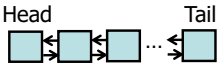


Just in case: ArrayList v.s. LinkedList

- ArrayList
  - Array-based implementation of the List interface
  - Fast indexed access
  - Slow insertion and removal of non-tail elements
    - Fast insertion of the tail element

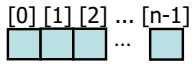


- LinkedList
  - Doubly-linked implementation of the List and Deque interfaces
  - Fast insertion and removal of non-tail elements
  - Slow "indexed" access for "middle" elements.

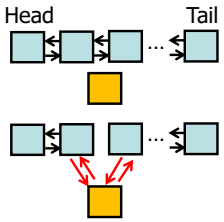


Just in case: ArrayList v.s. LinkedList

- ArrayList
  - Array-based implementation of the List interface
  - Fast indexed access
  - Slow insertion and removal of non-tail elements
    - Fast insertion of the tail element

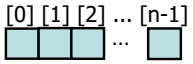


- LinkedList
  - Doubly-linked implementation of the List and Deque interfaces
  - Fast insertion and removal of non-tail elements
  - Slow "indexed" access for "middle" elements.

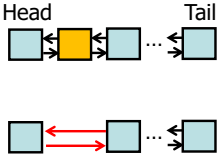


Just in case: ArrayList v.s. LinkedList

- ArrayList
  - Array-based implementation of the List interface
  - Fast indexed access
  - Slow insertion and removal of non-tail elements
    - Fast insertion of the tail element

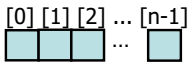


- LinkedList
  - Doubly-linked implementation of the List and Deque interfaces
  - Fast insertion and removal of non-tail elements
  - Slow "indexed" access for "middle" elements.

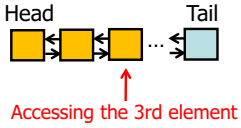


Just in case: ArrayList v.s. LinkedList

- ArrayList
  - Array-based implementation of the List interface
  - Fast indexed access
  - Slow insertion and removal of non-tail elements
    - Fast insertion of the tail element

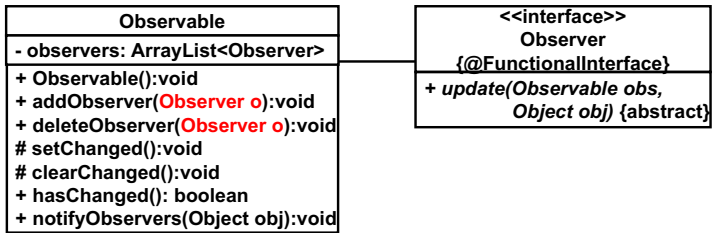


- LinkedList
  - Doubly-linked implementation of the List and Deque interfaces
  - Fast insertion and removal of non-tail elements
  - Slow "indexed" access for "middle" elements.



HW 1: Implement *Observer* with LEs

- Define your own `Observable` (class) and `Observer` (interface)
  - DO NOT reuse `java.util.Observable` and `java.util.Observer`
  - Define `Observer` as a functional interface.
    - `update()` as an abstract method.
  - Implement the listed methods for `Observable`
    - c.f. Java API doc for expected behaviors/responsibilities for the methods
- Use a lambda expression rather than defining a class that implements `Observer` and pass it to `addObserver()` and `deleteObserver()`.



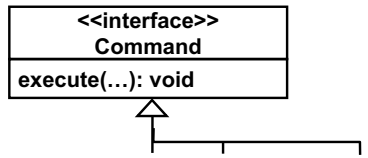
- Example client code

```
Observable observable = new Observable();
observable.addObserver( (Observable o, Object obj)->
                        {System.out.println(obj);} );
observable.setChanged();
observable.notifyObservers("Hello World!");
```

- Submission Requirements
  - Follow the submission requirements in CS680.
    - Never send me binary code. Avoid attaching a .zip file. Use Ant.
  - Send your HW solution to umasscs681@gmail.com
  - No need to do unit testing.
  - Due: March 8 midnight

HW 2: Implement *Command* with LEs [Optional]

- Just like HW 1, you can implement the *Command* design pattern with LEs.
  - Define the `Command` interface as a functional interface.
    - Have `Command` define the abstract method `execute()`.
  - Implement the body of `execute()` as a LE.
    - Do not define classes that implement `Command`.
- Explain how to do that with a specific example.
  - You can use an example in CS680. Your own example is fine too.
  - Submit pseudo code or compilable/runnable code.



- Due: March 8midnight