Welcome to CS681!

Tue Thu 5:30pm to 6:45

Healey H04-0031

Course Topics: Advanced SE

- Language/library enhancements in Java 7 and 8
  - File handling with NIO, try-with-resources statement, etc.
  - Functional programming with lambda expressions
- Concurrent programming
  - Java threads
  - Threading data structures, libraries and frameworks
  - Concurrent object-oriented design patterns
- Network programming
  - Socket programming with TCP/IP
  - I/O techniques
- Middleware and clouds
  - REST
  - MapReduce

Course Work

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

Grading

- Grading factors
  - Homework (65-70%)
  - Project deliverables (25%)
  - Quiz (5-10%)
- No textbooks
  - It might be helpful to have books on Java 8 and Java threading
  - Java API documentation
- No exams

Language/Library Enhancements in Java 7

Elementary Basics in 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 directories
      - 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 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`.
- A 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_NEW);`
    - `Files.write(path, lines);`
    - `Files.write(path, lines, StandardOpenOption.APPEND);`
    - `StandardOpenOption: APPEND, CREATE, CREATE_NEW, 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.
C.f. CS680 lecture note

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 read/write 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 read/write 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

- 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.
      - ```
        openFile();
        try{
            doSomethingWithFile();
        }catch(...){
            // Exception handling
        }finally{
            closeFile();
        }
        ```
  - Note: No need to call **`close()`** when using **`readAllBytes()`**, **`readAllLines()`** and **`write()`** of **`Files`**.

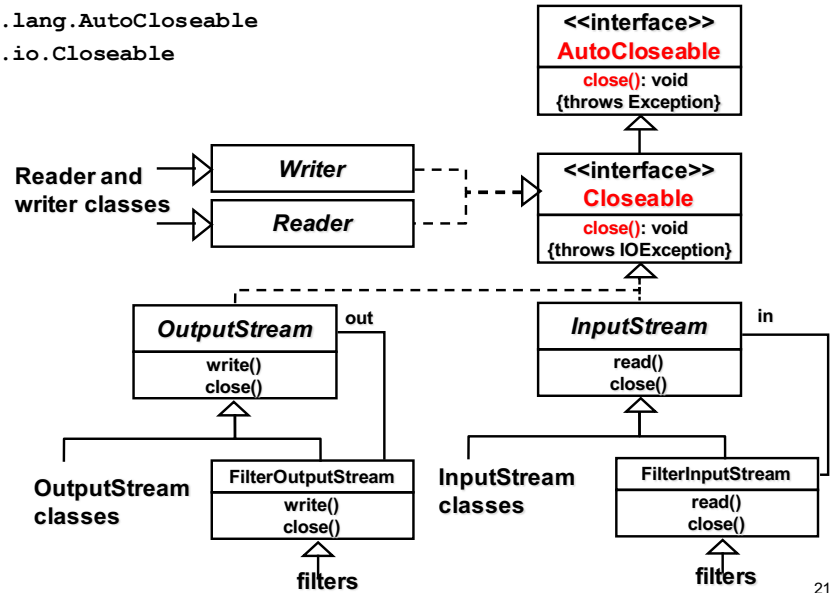
(2) Try-with-resources Statement

- Allows you to skip calling close() explicitly in the finally block.
  - *Try-finally*
    - ```
      openFile();
      try{
          doSomethingWithFile();
      }catch(...){
          // Exception handling
      }finally{
          closeFile();
      }
      ```
  - *Try-with-resources*
    - ```
      try ( openFile() ){
          doSomethingWithFile();
      }
      ```


- **`close()`** is automatically called on a resource used for reading or writing to a file, when existing 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 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 code simpler.

(4) Type Inference

- Can be used to define a variable 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<>();
    ```
- 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
    ```

Lambda Expressions in Java

- Lambda expression
  - A block of code (or a function) that you can pass to a method.
- Before Java 8, methods can receive primitive values and objects only.
  - `public void example(int i, String s, ArrayList<…> list)`
  - Methods receive nothing else.
    - You cannot do like this:
      - `foo.example( [if(Math.random()>0.5){ System.out.println(...);} else{ System.out.println(...);} ] )`

How to Define a Lambda Expression?

- A lambda expression consists of
  - A code block
  - A set of parameters to be passed to the code block

```
- (String str) -> str.toUpperCase()
- (StringBuffer first, StringBuffer second)
      -> first.append(second)
- (int first, int second)
      -> second - first

- (double threshold) -> {
      if(Math.random() > threshold) return true;
      else return false;
  }
- () -> {
      if(Math.random) > 0.5) return true;
      else return false;
  }
```


- No need to specify the name of a function.
  - Lambda expression ~ anonymous function/method that is not bound to a class/interface
- No need to explicitly specify the return value's type.
  - It is automatically inferred.
- Single-expression code block
  - Does not use the "return" keyword.
- Multi-expression code block
  - Surrounds expressions with { and }.
  - Needs to use the "return" keyword.
    - Every conditional branch must return a value.

```
• () -> {
      if(Math.random) > 0.5) return true;
//    else return false; ← A compilation error occurs here if this
                           line is commented out.
  }
```


How to Pass a Lambda Expression?

- A method can receive a lambda expression(s).
  - `foo.example( (int first, int second) -> second-first )`
  - The method receives a lambda expression as a parameter.
  - What is the type of that parameter?
    - *Functional interface*!

Functional Interface

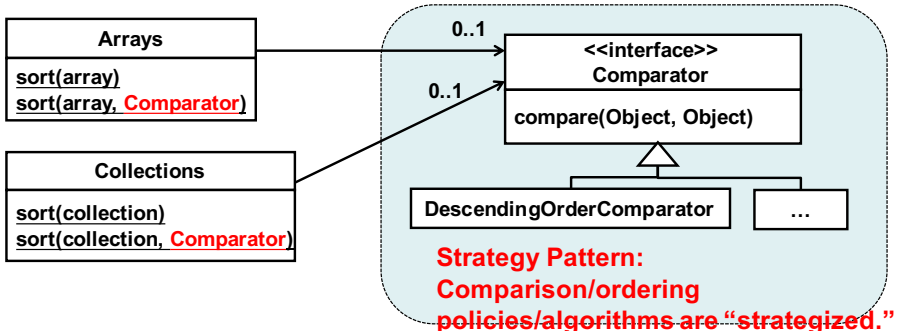
- A special type of interface
  - An interface that has a single abstract (or empty) method.
- An example functional interface: `java.util.Comparator`
  - Defines `compare()`, which is the only abstract/empty method.
    - A new annotation is available:
      - `@FunctionalInterface`
        `public interface Comparator<T>`
      - All functional interfaces in Java API have this annotation.
        - » The API documentation says "This is a functional interface and can therefore be used as the assignment target for a lambda expression…"
- `Collections.sort(List, Comparator<T>)`
  - The second parameter can accept a lambda expression.
  - ```
    Collections.sort( aList,
                      (Integer first, Integer second)->
                          second.intValue()-first.intValue() );
    ```


Recap: Comparators

- Sorting collection elements:
  - ```
    ArrayList<Integer> years2 = new ArrayList<Integer>();
    years2.add(new Integer(2010));
    years2.add(new Integer(2000));
    years2.add(new Integer(1997));
    years2.add(new Integer(2006));
    Collections.sort(years2);
    for(Integer y: years2)
        System.out.println(y);
    ```
  - java.util.Collections: a utility class (i.e., a set of static methods) to process collections and collection elements
  - sort() orders array elements in an ascending order.
    - 1997 -> 2000 -> 2006 -> 2010

Comparison/Ordering Policies

- What if you want to sort array/collection elements in a descending order or any specialized (user-defined) order?
  - Arrays.sort() and Collections.sort() implement ascending ordering only.
    - They do not implement any other policies.
- Define a custom comparator by implementing java.util.Comparator



- Arrays.sort() and Collections.sort() are defined to sort array/collection elements from "smaller" to "bigger" elements.
  - By default, "smaller" elements mean the elements that have *lower* numbers.
- A descending ordering can be implemented by treating "smaller" elements as the elements that have *higher* numbers.
- compare() in comparator classes can define (or re-define) what "small" means and what's "big" means.
  - Returns a negative integer, zero, or a positive integer as the first argument is "smaller" than, "equal to," or "bigger" than the second.
- ```
  public class DescendingOrderComparator implements Comparator{
    public int compare(Object o1, Object o2){
      return ((Integer)o2).intValue()-((Integer) o1).intValue();
    }
  }
  ```
  ```

Sorting Collection Elements with a Custom Comparator

```
ArrayList<Integer> years = new ArrayList<Integer>();
years.add(new Integer(2010)); years.add(new Integer(2000));
years.add(new Integer(1997)); years.add(new Integer(2006));
Collections.sort(years);
for(Integer y: years)
    System.out.println(y);
Collections.sort(years, new DescendingOrderComparator());
for(Integer y: years)
    System.out.println(y);
```

– 1997 -> 2000 -> 2006 -> 2010

– 2010 -> 2006 -> 2000 -> 1997


```
public class DescendingOrderComparator implements Comparator{
  public int compare(Object o1, Object o2){
    return ((Integer)o2).intValue()-((Integer) o1).intValue();
  }
}
```

- A more type-safe option is available/recommended:

```
public class DescendingOrderComparator<Integer>{
  implements Comparator<Integer>{
    public int compare(Integer o1, Integer o2){
      return o2.intValue()-o1.intValue();
    }
}
```


Okay, so What's the Point?

- Without a lambda expression

```
public class DescendingOrderComparator<Integer>{
  implements Comparator<Integer>{
    public int compare(Integer o1, Integer o2){
      return o2.intValue()-o1.intValue();
    }
}
Collections.sort(years, new DescendingOrderComparator());
```

- With a lambda expression

```
Collections.sort(years,(Integer o1, Integer o2)->
                        o2.intValue()-o1.intValue());
```

- Code gets more concise (shorter and simpler).
  - The lambda expression defines `DescendingOrderComparator`'s `compare()` in a concise way.
  - More readable and less ugly than the code based on an anonymous class.
- The LE version is a *syntactic sugar* for the non-LE version.
  - Your compiler does program transformation at compilation time.

FYI: Anonymous Class

- The most expressive (default) version

```
public class DescendingOrderComparator<Integer>{
  implements Comparator<Integer>{
    public int compare(Integer o1, Integer o2){
      return o2.intValue()-o1.intValue();
    }
}
Collections.sort(years, new DescendingOrderComparator());
```

- With an anonymous class

```
Collections.sort(years,
                 new Comparator<Integer>(){
                   @Override
                   public int compare(Integer o1, Integer o2){
                     return o2.intValue()-o1.intValue();
                   }
                 } );
```

- With a lambda expression

```
Collections.sort(years,(Integer o1, Integer o2)->
                        o2.intValue()-o1.intValue());
```

How Do You Know Where You can Use a Lambda Expression?

- You are trying to use `Collections.sort(List, Comparator<T>)`
- Check out `Comparator` in the API doc.
- Notice `Comparator` is a functional interface.
  - `@FunctionalInterface`
    `public interface Comparator<T>`
    - The API doc says "This is a functional interface and can therefore be used as the assignment target for a lambda expression…"
  - This means you can pass a lambda expression to `sort()`.
- Find out which method is the only abstract/empty (i.e., non-static, non-default) method.
  - `public int compare(T o1, T o2)`
- Define a lambda expression to represent the method body of `compare()` and pass it to `sort()`.
  - ```
    Collections.sort( aList,
                      (Integer first, Integer second)->
                          second.intValue()-first.intValue())
    ```


Assignment of a LE to a Functional Interface

- `Comparator` is a functional interface.
  - `@FunctionalInterface`
    `public interface Comparator<T>`
    - The API doc says "This is a functional interface and can therefore be used as the assignment target for a lambda expression…"
- A lambda expression can be assigned to a functional interface.
  - ```
    Comparator<Integer> comparator =
        (Integer o1, Integer o2)-> o2.intValue()-o1.intValue();
    Collections.sort(years, comparator);
    ```
- Parameter types can be omitted thru type inference.
  - ```
    Comparator<Integer> comparator =
        (o1, o2)-> o2.intValue()-o1.intValue()
    ```
  - C.f. Type inference with the diamond operator (introduced in Java 7).

What does Collections.sort() do?

- ```
  class Collections
      static ... sort(List<T> list, Comparator<T> c){
          for each pair (o1 and o2) of elements in list{
              int result = c.compare(o1, o2);
              if(result < 0){
                  ...
              }else if(result > 0){
                  ...
              }else if(result==0){
                  ...
        } } }
  ```
- C.f. Run this two-line code.
  - ```
    Comparator<Integer> comparator =
        (Integer o1, Integer o2)-> o2.intValue()-o1.intValue();
    comparator.compare(1, 10);
    ```
  - compare() returns 9 (10 - 1).

Some Notes

- A lambda expression can be assigned to a functional interface.
  - ```
    public interface Comparator<T>{
        public int compare(T o1, T o2)
    }
    Comparator<Integer> comparator =
        (Integer o1, Integer o2)-> o2.intValue()-o1.intValue()
    ```
  - `Collections.sort(years, comparator);`
- It cannot be assigned to `Object`.
  - ```
    Object comparator =
        (Integer o1, Integer o2)-> o2.intValue()-o1.intValue()
    ```
    ```

- Without a lambda expression
  - ```
    public class DescendingOrderComparator<Integer>{
      implements Comparator<Integer>{
        public int compare(Integer o1, Integer o2){
          return o2.intValue()-o1.intValue();
        }
    }
    Collections.sort(years, new DescendingOrderComparator());
    ```
- With a lambda expression
  - ```
    Collections.sort(years,(Integer o1, Integer o2)->
                           o2.intValue()-o1.intValue());
    ```
- A type mismatch results in a compilation error.
  - ```
    Collections.sort(years,(Integer o1, Integer o2)->
                          o2.floatValue()-o1.floatValue());
    ```
  - The return value type must be int, not float.
    - compare() is expected to return an int value.

- A lambda expression cannot throw an exception
  - if its corresponding functional interface does not specify that for the abstract/empty method.
- Not good (fails compilation)
  - ```
    public interface Comparator<T>{
        public int compare(T o1, T o2)
    }
    ```
  - ```
    Collections.sort(years,(Integer o1, Integer o2)->{
                            if(...) throw new XYZException;
                           else return ... );
    ```
- Good
  - ```
    public interface Comparator<T>{
        public int compare(T o1, T o2) throws ZYZException
    }
    ```
  - ```
    Collections.sort(years,(Integer o1, Integer o2)->{
                            if(...) throw new XYZException;
                           else return ... );
    ```


LEs make Your Code Concise, but…

- You still need to clearly understand
  - the Strategy design pattern
    - Comparator and its implementation classes
    - What compare() is meant to do
  - How Collection.sort() calls compare().
- Using or not using LEs just impact how to *express* your code.
  - This does not impact how to *design* your code.

A Benefit of Using Lambda Expressions

- Your code gets more concise.
  - This may or may not mean "easier to understand" depending on how much you are used to lambda expressions.
    ```