Concurrency and Immutability

Immutable Classes

- Classes that never change the state of each instance
  - State = A set of data field values
  - Getter methods only; no setter methods available.
- No state (value) changes → No race conditions!
  - Race condition = State/value inconsistency on a shared data field(s) among threads
  - c.f. `BankAccount`'s `balance`, `PrimeGenerator`'s `done`
- No methods require thread sync.
- All methods are always thread-safe.

An Example Immutable Class: `java.lang.String`

```
char[] str = {"u", "m", "b"};
String string = new String(str);
```

```
String string = "umb";    // Syntactic sugar for the above code
```

- A series of constructors to initialize string data.
- All non-constructor methods never change the initialized string.
- No setter methods are available.
  - c.f. API doc.

Example Methods in `String`

```
String str = "umb";
System.out.println( str );     // println() calls str.toString()
                               // prints out "umb"
```

```
System.out.println( str.replace("b","l"));// uml
                        // Creates a new String instance that
                        // contains "uml" and returns it.
System.out.println( str );              // umb
```

```
System.out.println( str.toUpperCase() );  // UMB
                        // Creates a new String instance
                        // that contains "UMB" and returns it.
System.out.println( str );              // umb
```

```
System.out.println( str.substring(0,1) ); // um
                        // Creates a new String instance that contains
                        // "um" and returns it.
System.out.println( str );              // umb
```

- Some methods of `String` look like setters, but they are actually NOT.
  - They never change the initialized string data ("umb").

`String`

- Final class, which cannot be extended (sub-classed)
  - `public final class String{…}`
  - Prevents its sub-classes from introducing new methods that try to change the initialized string data (e.g. "umb").
- Maintains the initialized string in a private and final data field.
  -
```
public final class String{
    private final char value[];
    ... }
```
  - Once a value is assigned to a final variable, the value cannot change afterward.
    - No methods of `String` can change the initialized string data.

- Each "setter-like" method of `String` creates another `String` instance that contains another string data.
  -
```
public final class String{
    private final char[] value;            // Immutable
    ...
    public String toUpperCase(){
        int length = value.length;         // Local variable
        char[] result = new char[length];  // Local variable

        for( int i = 0; i < length; i++ ){ // Local variable
            result[i] = ... // Transform value[i] to an upper case
        }
        return new String(result);          // Local variable
    } }
```
  - This is actually NOT a setter method! It is thread-safe.
    - No local variables are shared by threads.
      - Each thread creates a copy of a local variable.
      - Different threads use different copies of it.

Benefits of Immutability

- For API designers
  - An immutable class never require thread sync in its methods.
    - No need to guard its data field (e.g., `value` in `String`) with a lock
      - The data field's value never changes.
      - All threads simply read "fixed" (or "finalized") data from the data field.
  - Its methods are free from race conditions.
    - Code revision and debugging get easier.
- For API users
  - Immutable classes are free from race conditions.
    - Their client code doesn't always need thread synch.
  - They are free from potential performance loss due to thread sync.
    - Thread sync forces every thread to acquire a lock.
      - There is some overhead to acquire a lock.
    - If the lock is not available, the thread needs to get into the Blocked state until it becomes available.
      - It cannot do anything to make progress.

Note That…

- An immutable class's methods are thread-safe, but…
- Client code of those methods may NOT be always thread-safe.
  - The code below is NOT thread-safe; it requires thread sync.

```
public class Person {
 private String firstName, lastName;  // Shared variables

 public void setLastName(String last){
   lastName = last; }                 // 2 Steps; NOT thread-safe

 public String getLastName(){
   return lastName; }                 // 2 Steps; NOT thread-safe

 public String getFullName(){
   return firstName + " " + lastName; // Multi-steps; NOT thread-safe
}}
   // Syntactic sugar for new StringBuilder().append(firstName).append(" ")
   //                                        .append(lastName).toString()
   // Reads on firstName and lastName are NOT thread-safe.
   // An instance of StringBuilder is local for each thread though.
```

- An immutable class's methods are thread-safe, but…
- Client code of those methods may NOT be always thread-safe.
  - The code below is thread-safe.

```
public class ErrorMsgGenerator {
 ...
 public String getFileNotFoundErrorMsg(String path){
   return "The requested file " + path + " is not found.";

     // Syntactic sugar for new StringBuilder().append("The requested...")
     //                                        .append(path)
     //                                        .append(" was not...")
     //                                        .toString();
     // Multiple steps, but thread-safe
     // Local variables are not sharable by threads.
     // Reads and writes on a local variable are thread-safe.
     // "path" is a local variable.
     // An instance of StringBuilder is local for each thread.
 } }
```

Other Immutable Classes

- Wrapper classes for primitive types
  - c.f. CS680
- **`java.nio.file.Path`**
  - c.f. CS680
- **`java.util.regex.Pattern`**
- Some classes in **`java.net`**
  - e.g., **`URL`**, **`URI`**, **`Inet4Address`** and **`Inet6Address`**
- Date and Time API (**`java.time`**)

Primitive type	Wrapper class
boolean	Boolean
byte	Byte
char	Character
float	Float
int	Integer
long	Long
short	Short
double	Double

`Integer`

- Wrapper class of an **`int`** value
  - Final class, which cannot be extended (sub-classed)
  - Maintains the initialized **`int`** data in a private and final data field.

```
- Integer int = Integer.valueOf(10); // Static factory method
- Integer int = 10;    // Auto-boxing; Syntactic sugar for
                       // the above code
```

  - Has no setter methods; no methods change the initialized **`int`** data.
  - All methods are thread-safe.

Note That…

- An immutable class's methods are thread-safe, but…
- Client code of those methods may NOT always be thread-safe.
  - The code below is NOT thread-safe; it requires thread sync.

```
public class Person {
 private Integer age;                  // Shared variable

 public void setAge(Integer age){
   this.age = age; }                   // 2 steps; NOT thread-safe

 public Integer getAge(){
    return this.age; }                 // 2 steps; NOT thread-safe


public boolean isKindergartener(){     // Multi-steps. Not thread-safe
   if(this.age < 6){return true;}      // Equivalent to:
   else{return false;}                 // if(this.age.intValue()<6)...
 }
```


- An immutable class's methods are thread-safe, but…
- Client code of those methods may NOT always be thread-safe.
  - The code below is thread-safe.


```
public class ErrorMsgGenerator {
 private final Integer FILE_NOT_FOUND = Integer.valueOf(404);
 ...

 public String getFileNotFoundErrorMsg(Path path){
   String header = "Error code: " + FILE_NOT_FOUND;
      // Syntactic sugar for
      //      new StringBuilder().append("Error code: ")
      //                         .append(FILE_NOT_FOUND.toString()).toString()
      // Reads on local and final variables are thread-safe.
      // An instance of StringBuilder is local for each thread.

   String body = "The requested file " +
                    path.toString() + " is not found."
      // "path" is a local variable. A read on it is thread-safe.

   return header + " " + body;                 // Thread-safe  }  }
```

Implementing User-Defined (Your Own) Immutable Classes

- Immutable class
  - Defined as a `final` class
  - Has `private final` data fields only.
  - Has no setter methods.

- Clearly state immutability in program comments, API documents, design documents, etc.
  - Java API documentation does so too.
  - Put {immutable} to a class in UML

An Example User-Defined Immutable Class

```
public final class SSN {
  private final int first3Digits, middle2Digits, last4Digits;

  public SSN(int first, int middle, int last){   // Thread-safe
      this.first3Digits = first;
      this.middle2Digits = middle;
      this.last4Digits = last;  }
                                                    // 2 steps but a read
  public int getLast4Digits(){return last4Digits;} // on a final variable
                                                    // is thread-safe.
  public String toString(){
      return first3Digits + "-" + middle2Digits + "-" + last4Digits;
        // Multiple steps, but thread-safe
        // Reads on final variables are thread-safe.  }

  public boolean equals( SSN anotherSSN ){
      if( this.toString().equals(anotherSSN.toString()) ){ return true; }
      else{ return false; }
        // Multiple steps, but thread-safe
        // SSN and String are immutable.  }}
```

```
public final class SSN {
  private final int first3Digits, middle2Digits, last4Digits;

  public SSN(int first, int middle, int last){ // Thread-safe
      this.first3Digits = first;
      this.middle2Digits = middle;
      this.last4Digits = last;  }
```

- A constructor is always executed atomically.
  - Only one thread can run a constructor on a class instance that is being created and initialized.
    - Multiple threads never call a constructor(s) on the same instance concurrently.
  - Until a thread returns/completes a constructor on a class instance, no other threads can call public methods on that instance.

Note That…

- An immutable class's methods are thread-safe, but…
- Client code of those methods may NOT always be thread-safe.
  - The code below is thread-safe.

```
public class Person {
  private final SSN ssn;   // Shared final variable

  public Person(SSN ssn){ this.ssn = ssn; }

  public SSN getSSN(){     // 2 Steps, but thread-safe.
    return ssn;            // "ssn" is final.
  } }

Person person = new Person( new SSN(012, 34, 5678) );
person.getSSN();
```

- An immutable class's methods are thread-safe, but…
- Client code of those methods may not always be thread-safe.
  - The code below is NOT thread-safe; it requires thread sync.

```
public class Person {
  private SSN ssn;         // Shared (non-final) variable

  public Person(SSN ssn){ this.ssn = ssn; }

  public SSN setSSN(SSN ssn){
    this.ssn = ssn;        // 2 steps; not thread-safe
  }

  public SSN getSSN(){     // 2 steps; not thread-safe
    return ssn;            // "ssn" is NOT final.
  } }
```

`Person` requires thread synch to guard `ssn`, although `SSN` does not.

Record Type: Convenience Mechanism to Implement an Immutable Classes

- Available since Java 16

```
public record SSN(int first3Digits,
                  int middle2Digits,
                  int last4Digits){}
```

- Syntactic sugar for:

```
public final class SSN extends java.lang.Record{
  private final int first3Digits;
  private final int middle2Digits;
  private final int last4Digits;

  public SSN(int first3Digits, int middle2Digits, int last4Digits){
    this.first3Digits = first3Digits;
    ... }

  public int first3Digits(){ return first3Digits; }

  // 2 other getters, equals(), toString() and hashCode() are
  // auto-generated }
```

- You can
  - Override the (auto-generated) constructor.
  - Override auto-generated methods (e.g. getters)
  - Add extra methods.

```
public record SSN(int first3Digits,
                  int middle2Digits,
                  int last4Digits){
    public int to9Digits(){
        return ...
    }
}
```

```
public class Aircraft {
  private Position position;      // Shared (non-final) variable

  public Aircraft(Position pos){ this.position = pos; }

  public void setPosition(double newLat,
                          double newLong,
                          double new Alt){
    this.position = this.position.change(newLat, newLong, newAlt);
                                  // 2 steps; NOT thread-safe.  }

  public Position getPosition(){
    return position;              // 2 steps; NOT thread-safe.
  }  }
```

`Aircraft` requires thread sync to guard **`position`**, although **`Position`** does not.

HW 9

- Implement your own immutable:

```
public record Position(double latitude,
                       double longitude,
                       double altitude){} // decimal degrees
```

- Add extra methods:

```
List<Double> coordinate()

Position change(double newLat, double newLon, double newAlt){
  return new Position(newLat, newLon, newAlt);}
```

  - It sounds like a setter, but it is NOT. It creates a new instance and returns it.

```
boolean higherAltThan(Position anotherPosition)
boolean lowerAltThan(Position anotherPosition)
boolean northOf(Position anotherPosition)
boolean southOf(Position anotherPosition)
```

- Turn in
  - Immutable **`Position`**
  - Thread-safe **`Aircraft`**, which…
    - has a **`ReentrantLock`** as a data field and performs thread synch with it, OR
    - uses **`AtomicReferenceType<Position>`** and skips thread sync.
  - **`Runnable`** class whose **`run()`** calls **`Aircraft`**'s **`setPosition()`** and **`getPosition()`**
    - You can replace the **`Runnable`** class with a lambda expression, if you like.
  - Test code to create and run multiple threads

Performance Implications

- An immutable object makes a big difference in performance
  - when threads often read data from the object
  - because they do not perform thread synch.
- If you are interested, compare the performance of
  - Immutable `Position` and
  - Mutable `Position` that performs thread sync in its setters and getters.
  - Immutable `Position` is approx. 25% faster on my machine.
- Setter-like methods of an immutable class
  - e.g., replace() and toUpperCase() of String
  - NOT that efficient
    - because they create new instances of the immutable class.
- Again, the performance benefit of an immutable class stands out when…
  - threads often READ data from an immutable object
- An immutable object never trigger performance loss in single-threaded apps.
  - If a single-threaded app calls a mutable object's method that performs thread synch, the app incurs unnecessary performance loss.
    - The app never need thread synch, but the mutable object's method does it for the app.
- Recent APIs often rely on immutables, whenever/wherever possible, considering both thread safety and performance.
  - e.g., Date and Time API

Well, Not All Classes can be Immutable…

- Immutable classes are good for both API designers and API users.
- However, in practice, many classes need to be mutable…
- Think of separating a class to mutable and immutable parts
  - if read operations are called very often
  - if write operations are occasionally called but they are computationally expensive.

An Example: `String` and `StringBuilder`

- Both represent string data.
- **`String`**
  - Immutable: Its state never change once it is set up.
  - Thread-safe
  - Getter methods (reads): thread-safe.
  - Setter-like methods (writes): thread-safe, relatively slow.
- **`StringBuilder`**
  - Mutable: Its state can change through its methods.
  - NOT thread-safe; its public methods never perform thread synch.
  - Getters (reads): NOT thread-safe.
  - Setters (writes): NOT thread-safe, relatively fast.

- Use **`String`** as far as…
  - you often call getter methods (i.e. read operations)
    - e.g., toString(), equals(), contains() and length()
  - you occasionally call setter-like methods (i.e. write operations)
    - e.g., replace(), toUpperCase() and substring()
    - Here, "occasionally" means calling a setter-like method once or a few times at a time.
- Use **`StringBuilder`** only when…
  - you call write operations more often than usual
    - e.g., hundreds/thousands of times in a loop
    - **`StringBuilder`**'s write operations (setter methods) are faster than **`String`**'s write operations (setter-like methods).

Performance Comparison:
String Concatenation as an Example Write Operation

```
String str = "UMass";
str = str + " Boston";          // "UMass Boston"
    // Syntax sugar for:
    // str = new StringBuilder(str).append(" Boston").toString();

    // Creates 2 instances: StringBuilder and String
    // Calls 3 methods: StringBuilder's constructor and 2 methods
```

```
StringBuilder builder new StringBuilder(str);
builder.append(" Boston");
str = builder.toString();
```

- No difference in performance.

```
String header = "Error code: "+ FILE_NOT_FOUND;
String body = "The requested file " + path.toString()
                                    + " was not found."
return header + " " + body;
   // Syntax sugar for:
   // header = new StringBuilder("...").append(FILE_NOT_FOUND).toString();
   // body = new StringBuilder("...").append(...).append("...").toString();
   // return new StringBuilder(header).append(" ").append(body).toString();
   //
   // Creates 6 instances and calls 11 methods
```

```
StringBuilder builder = new StringBuilder();
builder.append("Error code: ");
builder.append(FILE_NOT_FOUND);
builder.append("The requested file ");
...
builder.append(" was not found.");
return builder.toString();
   // Creates 2 instance and calls 5 methods
```

- More visible difference in performance, if string concatenation is performed with multiple statements.

- `LinkedList<String> emailAddrs = ...;`

- ```
  String commaSeparatedEmailAddrs;
  for(String emailAddr: emailAddrs){
      commaSeparatedEmailAddrs += emailAddr + ", ";  }
  ```

- ```
  StringBuilder commaSeparatedEmailAddrs;
  for(String emailAddr: emailAddrs){
      commaSeparatedEmailAddrs.append(emailAddr).append(", "); }
  ```

- The latter code can run 20-100% faster depending on the number of collection elements (i.e. email addresses).

- DO NOT use `String` (immutable class) to perform a number of write operations.

When to Use Immutable and Mutable Classes?

- Use `String` (immutable class) for read operations

- Use `StringBuilder` (mutable class) for write operations
  - Note that `StringBuilder`'s methods are NOT thread-safe; e.g., `append()`.
    - Use a `StringBuilder` as a local variable
    - Do thread synchronization in multi-threaded apps.

- `String`-to-`StringBuilder` conversion is implemented in a constructor of `StringBuilder`.

- `StringBuilder`-to-`String` conversion is implemented in a constructor of `String`.

`StringBuffer`

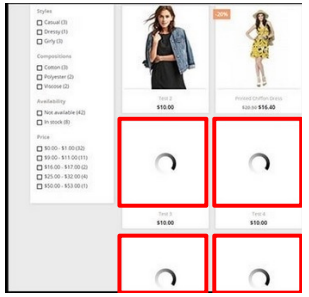
- Provides the same set of public methods as `StringBuilder` does.

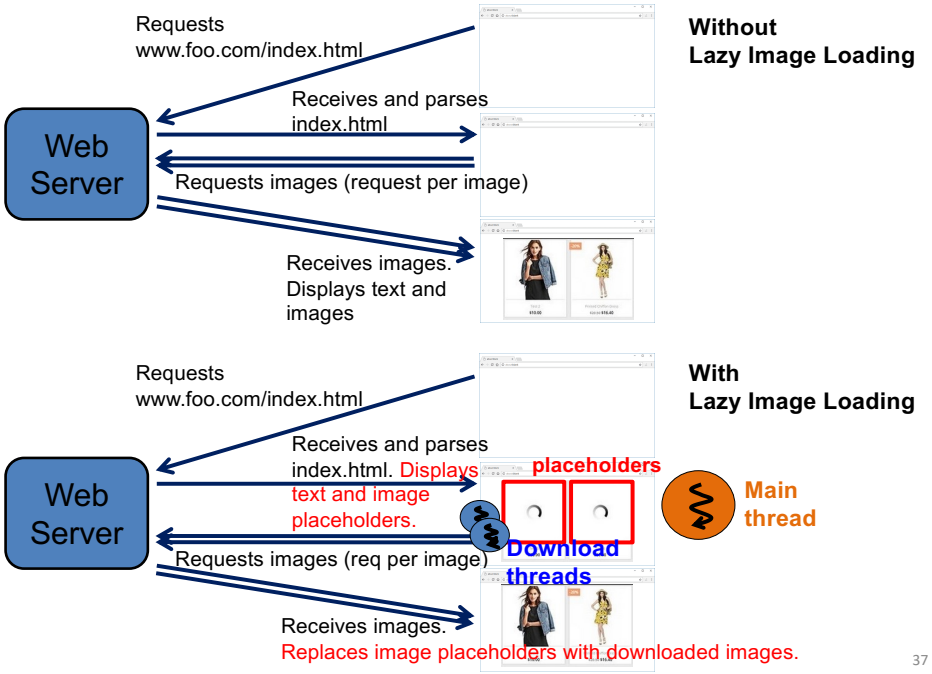
- `StringBuffer` (since Java 1.0)
  - All public methods are thread-safe with thread sync.
  - Client code of `StringBuffer` may still require thread sync.
  - DO NOT use this class.
    - It makes no sense to use it in single-threaded apps.

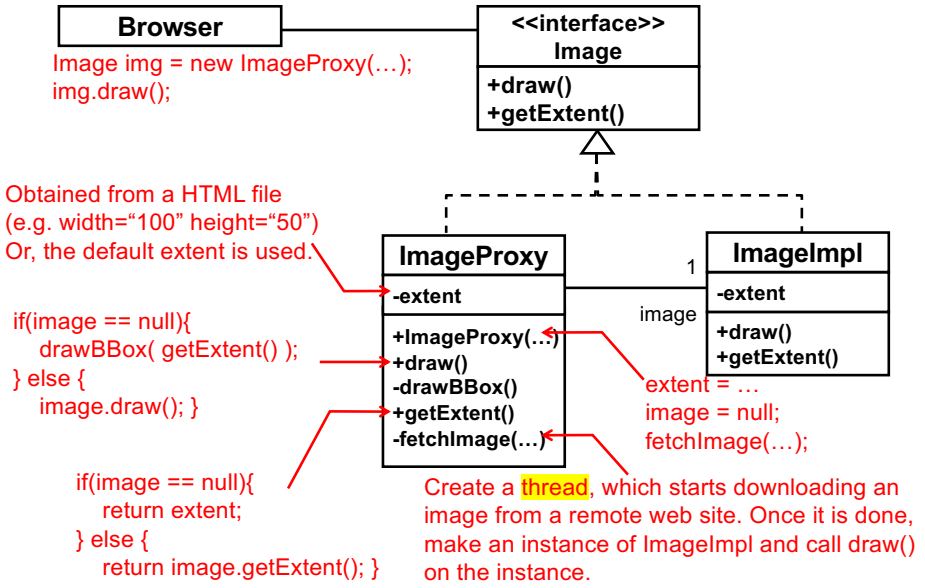
- `StringBuilder` (since Java 5)
  - All public methods are NOT thread-safe.
  - Client code of `StringBuilder` require thread sync.
  - Use this class
    - regardless of single-threaded or multi-threaded apps.

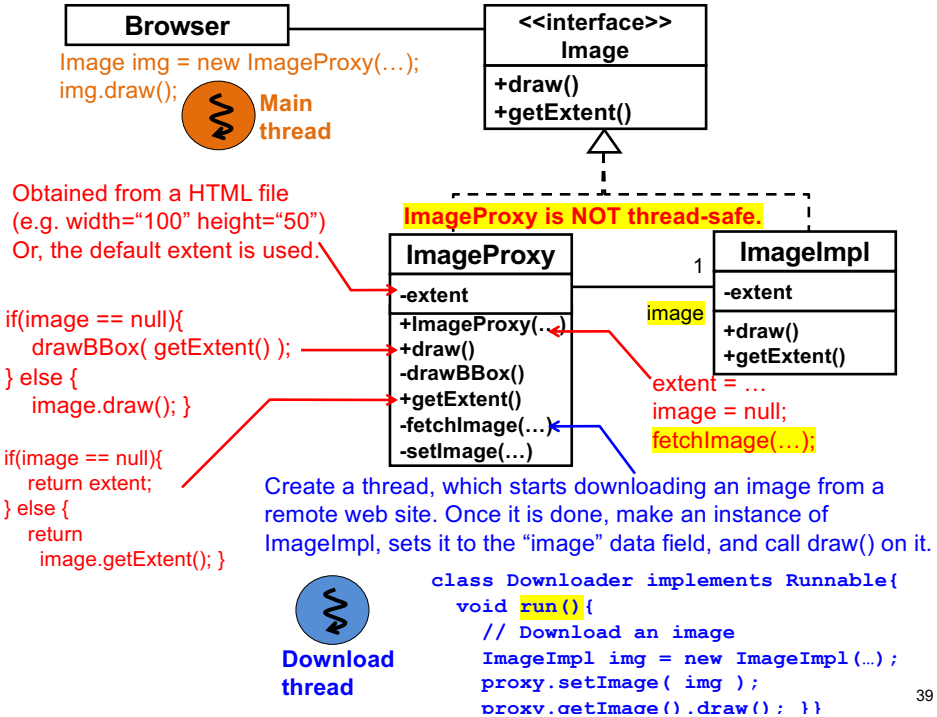
Exercise: Lazy Image Loading

- Suppose a browser just downloaded an HTML file and found it contains images. The browser…
  - Displays a bounding box (placeholder) first for each image
    - Until it fully downloads the image.
      - Most users cannot be patient enough to keep watching blank browser windows until all text and images are downloaded and displayed.
  - Replaces the bounding box with the real image.



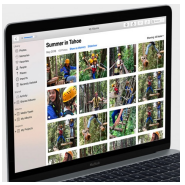






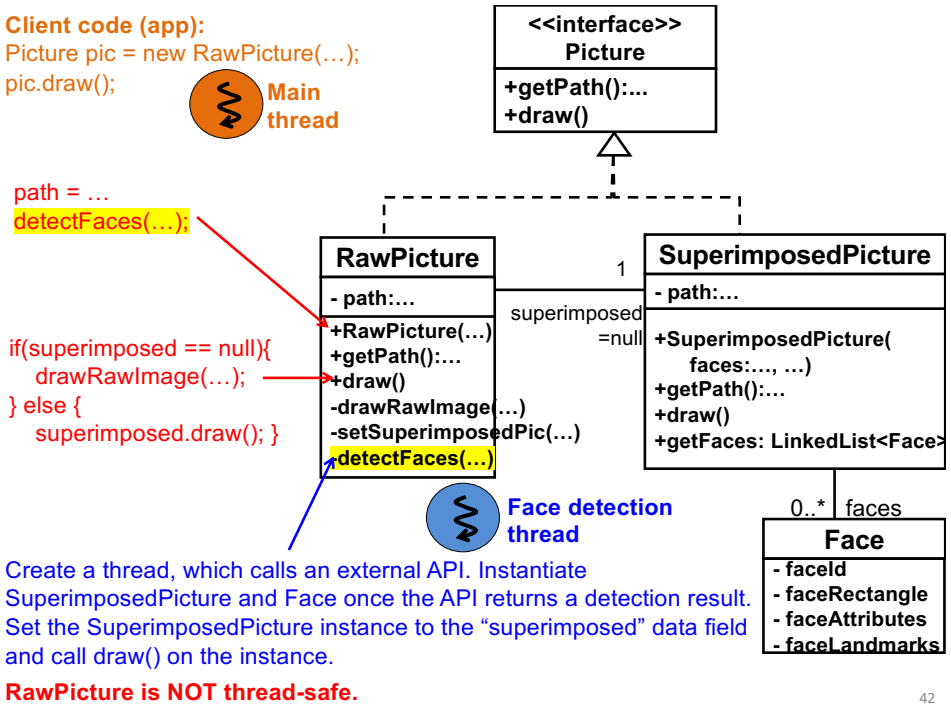
Recap: Face Detection

- Suppose you are implementing an app to organize, edit and analyze pictures.
  - When the app loads a raw picture, it superimpose a rectangle on a human face in the picture by (dynamically) calling an external face detection/recognition API.
    - e.g., Microsoft Azure Face API, Google Cloud Vision API

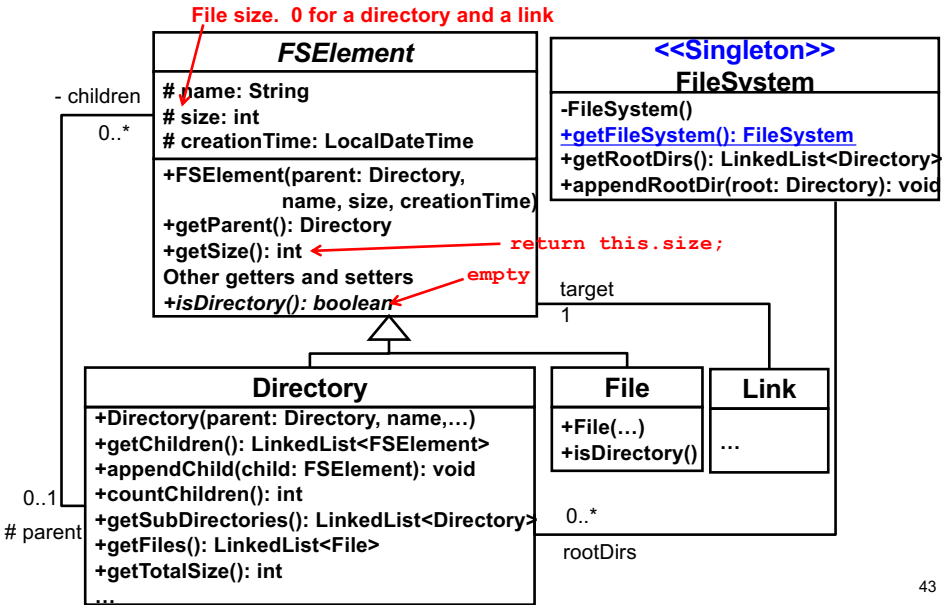


- Some delay is expected to receive a face detection result from an external API.
  - The user is not patient enough to keep watching a blank app window until the picture and face detection result are displayed.

- Lazy loading of detection results
  - Show the user a raw picture first.
  - Call a face detection API.
  - Receive a detection result.
  - Replace the raw picture with a superimposed one, which contains a detection result.



Recall CS680's HW 8



HW 10

- In HW 7, you revised `FileSystem`'s `getInstance()` to be thread-safe.
- In this HW, revise all the methods in `FSElement`, `Directory`, `File`, `Link` and `FileSystem` to be thread-safe.
- Define a `ReentrantLock` in `FSElement` as its data field.
  - Use the lock to guard all the data fields in `FSElement`, `Directory`, `File`, and `Link`.
    - Any methods that access those data fields must perform thread sync with that lock.
  - e.g., `name` in `FSELement`

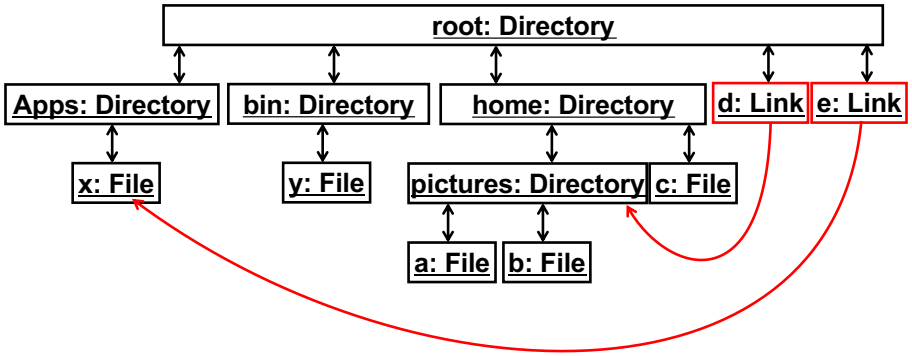
```
getName(){
  this.lock.lock();
  return this.name;
  this.lock.unlock();}
```

```
setName(String newName){
  this.lock.lock();
  this.name = newName;
  this.lock.unlock();}
```

- **`creationTime`** is typed with **`LocalDateTime`**.
  - All the classes in Date and Time API are thread-safe.
    - **`LocalDateTime`** is thread-safe; all of its public methods are thread-safe.
  - When you call a method of **`LocalDateTime`**, you do NOT have to do thread sync.

- **`children`** of **`Directory`** is typed with **`LinkedList`**.
  - All the collection classes in the **`java.util`** package are NOT thread-safe.
    - **`LinkedList`** is NOT thread-safe.
      - All of its public methods are NOT thread-safe; they never perform thread synchronization.
  - Whenever you call a method of **`LinkedList`** on **`children`**, make sure to do thread sync (i.e., surround that method call with **`lock()`** and **`unlock()`** on the lock in **`Directory`**).

- Keep **`FileSystem`** as a thread-safe *Singleton* class.



- Use this tree structure for testing.
  - You used this structure as a test fixture in CS680.
  - Assign values to data fields (size, etc) as you like.
- Have the main thread create and run 10+ extra threads, each of which will call the methods of **`Directory`**, **`File`**, **`Link`** and **`FileSystem`**.
- Have the main thread perform 2-step termination for those extra threads.