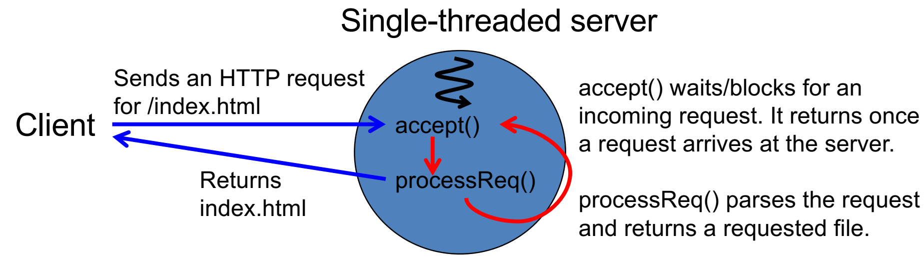
Race Conditions

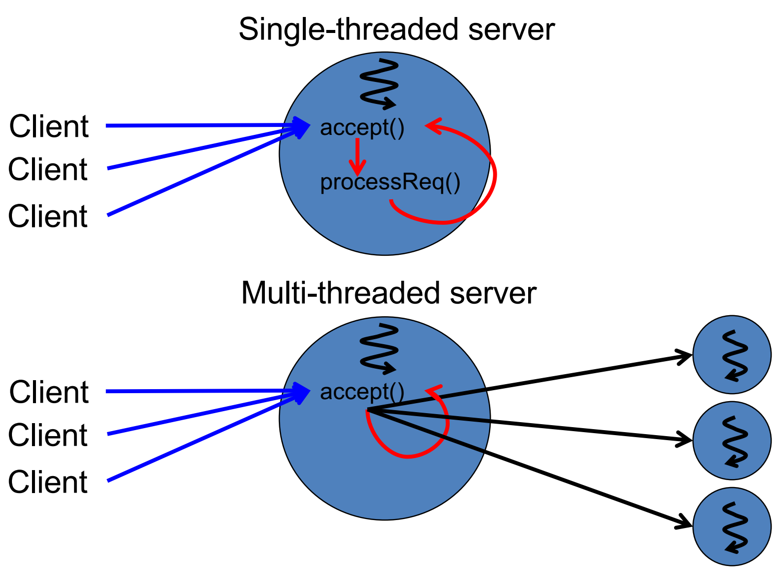
- Race conditions can occur if…
  - Multiple threads share and access a variable (data field).
    - Solution to eliminate race conditions
      - Define a lock in the variable's enclosing class
      - Use the lock to access the variable
        - » Surround every read/write logic on the variable with lock() and unlock()
  - Multiple threads call an API method that is NOT thread-safe.
    - You cannot define a lock in the method's enclosing class (i.e., API class)
    - You need to perform thread synchronization in your client code that uses the API method.

Exercise: Access Counter for a Web Server

- Suppose you are developing a web server.
  - Receives a HTTP request that a client (browser) transmits to request an HTML file.
  - Returns the requested file to the client.
- What if the server receives multiple requests from multiple clients simultaneously?
  - If the server is single-threaded, it processes requests *sequentially*.

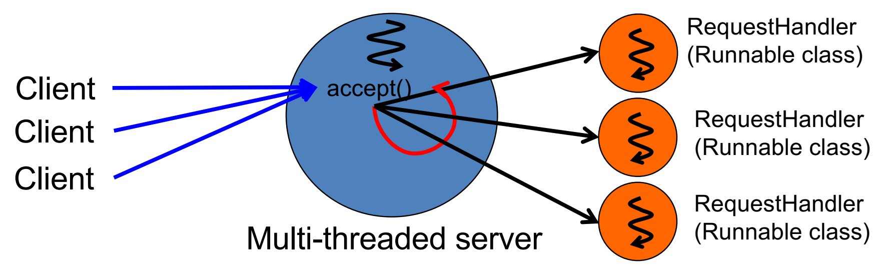


Concurrent (Multi-threaded) Web Server



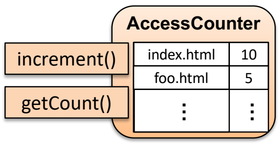
Thread-per-Request Concurrency

- Once the web server receives a request from a client, it creates a new thread.
  - The thread parses the incoming request and returns a requested file.
  - The thread terminates once the requested file is returned to the client.



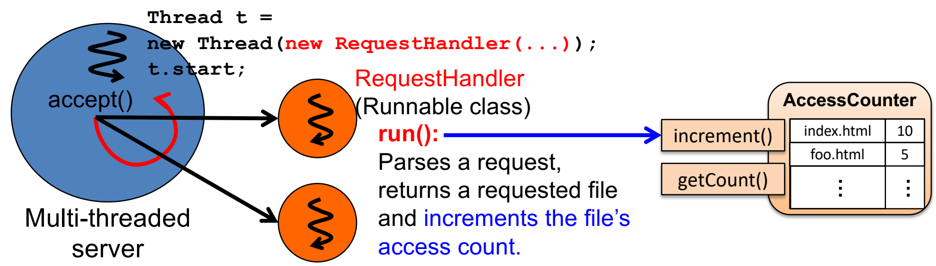
Access Counter in a Concurrent Web Server

- **AccessCounter**
  - Maintains a map that pairs a relative file path and its access count.
    - Assume `java.util.HashMap<Path, Integer>`
  - `void increment(Path path)`
    - accepts a file path and increments its access count.
  - `int getCount(Path path)`
    - accepts a file path and returns its access count.



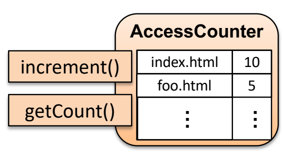
Access Counter in a Concurrent Web Server

- **AccessCounter**
  - Maintains a map that pairs a relative file path and its access count.
    - Assume `java.util.HashMap<Path, Integer>`
  - `void increment(Path path)`
    - accepts a file path and increments its access count.
  - `int getCount(Path path)`
    - accepts a file path and returns its access count.

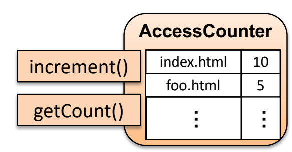


Concurrent Access Counter

- `HashMap` is NOT thread-safe.
  - All of its public methods never perform thread synchronization.
    - containsKey(), put(), get(), putIfAbsent(), replace(), etc.
  - Race conditions can occur in those public methods.
- Client code of those public methods need to perform thread synchronization.
  - `AccessCounter`'s `increment()` and `getCount()`
    - `increment()`
      - if(A requested path is in AC){
        increment the path's access count. }
        else{
        add the path and the access count of 1 to AC. }
    - `getCount()`
      - if(A requested path is in AC){
        get the path's access count and return it. }
        else{
        return 0. }



- Client code of those public methods need to perform thread synchronization.
  - `AccessCounter`'s `increment()` and `getCount()`
    - `increment()`
      - lock.lock();
        if(A requested path is in AC){
        increment the path's access count. }
        else{
        add the path and the access count of 1 to AC. }
        lock.unlock();
    - `getCount()`
      - lock.lock();
        if(A requested path is in AC){
        get the path's access count and return it. }
        else{
        return 0. }
        lock.unlock();



HW 13

- Implement **`AccessCounter`** as a thread-safe *Singleton* class.
  - Define a **`HashMap<java.nio.Path, Integer>`**
  - Define a regular (non-static) lock and use the lock in **`increment()`** and **`getCount()`**
  - Define another (static) lock and use the lock in **`getInstance()`**
- Place some test/dummy files
  - ```
    AccessCounter.java
    RequestHandler.java
    a.html
    b.html
    ...
    ```
- **`RequestHandler`**: A Runnable class
  - **`run()`**: Picks up one of the files at random, calls **`increment()`** and **`getCount()`** for that file, and sleep for a few seconds. Repeat this forever with an infinite loop.
- **`main()`**: Test code
  - Creates and starts 10+ threads that execute **`RequestHandler`**'s **`run()`**.
- Implement 2-step thread termination in **`RequestHandler`**.
  - Have the main thread terminate those 10+ threads in 2 steps.

- Deadline: Nov 1 (Thu) midnight

Concurrency and Immutability

Immutable Classes

- Classes that never change the state of each instance
  - Getter methods only; no setter methods available.
- All public methods are thread-safe because they never need thread synchronization.
  - No need to worry about race conditions.
  - No performance loss.
- An example: **`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 data.
  - No setter methods are available.
    ```

Example Methods in `String`

- ```
  String str = "umb";
  System.out.println( str );               // 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(1,2) ); // mb
                           // Creates a new String instance that contains
                           // "mb" and returns it.
  ```
- ```
  System.out.println( str );               // umb
  ```

- Some methods of `String` look like setter methods, 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 updating the initialized string data.
- Maintains the initialized string data (e.g., "umb") 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 be changed afterward.
    - No methods of `String` can change the value.

- 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);
    } }
    ```
  - This is actually NOT a setter method!

Benefits of Immutability

- For API designers
  - An immutable class never require thread synchronization 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" data from the data field.
  - Its methods are free from race conditions.
    - Makes it easier to do debugging.
    ```

- for API users
  - Immutable classes are free from potential performance loss due to thread synchronization.
    - Thread synchronization 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 be in 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 or may not be thread-safe.
  - The code below is NOT thread-safe; it requires thread synchronization.

```
public class Person {
 private String firstName, lastName; // Shared variables
 ...
 public void setFirstName(String first){
   firstName = first;             // 2 steps. Not thread-safe  }

 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.
   // Syntax sugar for new StringBuilder().append(firstName).append(" ")
   //                                     .append(lastName).toString()
   // append() and toString() are thread-safe though. } }
```

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

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

     // Syntax sugar for new StringBuilder().append("...")
     //                                    .append(path.toString())
     //                                    ....toString();
     // Multiple steps, but thread-safe!
     // append() and toString() are thread-safe.
     // Reads and writes on a local variable are thread-safe.
     // Local variables are not sharable by threads.
     // A copy of it is created for each thread.
     // "path" is also a local variables.
 } }
```

Other Immutable Classes

- Wrapper classes for primitive types

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

- `java.nio.file.Path`
- `java.util.regex.Pattern`
- Some classes in `java.net`
  - e.g., `URL`, `URI`, `Inet4Address` and `Inet6Address`
- Date and Time API (`java.time`)
  - All the classes are immutable and thread-safe.

`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);
– Integer int = 10;   // 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 or may not be thread-safe.
  - The code below is thread-safe.

```
public class ErrorMsgGenerator {
 private final Integer FILE_NOT_FOUND = new Integer.valueOf(404);
      // immutable
 ...
 public String getFileNotFoundErrorMsg(Path path){
   String header = "Error code: " + FILE_NOT_FOUND;
      // Syntax sugar for
      //      new StringBuilder().append("...")
      //                  .append(FILE_NOT_FOUND.toString()).toString()
      // append() and toString() are thread-safe.
      // Reads on local and immutable variables are thread-safe.

   String body = "The requested file " +
                 path.toString() + " was not found."
      // toString() is thread-safe.

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

Date and Time API: History

- `java.util.Date` (since JDK 1.0)
  - Poorly designed: Never try to use this class
    - It still exists only for backward compatibility
- `java.util.Calendar` (since JDK 1.1)
  - Deprecated many methods of `java.util.Date`
  - Limited capability: Try not to use this class
- Date and Time API (`java.time`)
  - Since JDK 1.8
  - Always try to use this API.

Date and Time API: `Instant`

- Represents an instantaneous point on the timeline, which starts at 01/01/1970 (on the prime Greenwich meridian).
  - Can be used as a timestamp.
- **`Duration`**
  - Represents an amount of time in between two `Instant`s

```
– Instant start = Instant.now();
  ...
  Instant end = Instant.now();
  Duration timeElapsed = Duration.between(start, end);
  long timeElapsedMSec = timeElapsed.toMillis();
```

  - This code is thread-safe as far as all the variables are local variables.

Date and Time API: "Local" Classes

- **`LocalDate, LocalTime, LocalDateTime`**
  - Used to represent date and time without a time zone (time difference)
  - Apply leap-year rules automatically.
    - ```
      LocalDate today = LocalDate.now();
      LocalDate birthday = LocalDate.of(2009, 9, 10);
      LocalDate 18thBirthday = birthday.plusYears(18);
      birthday.getDayOfWeek().getValue();
      ```
- **`Period`**
  - Represents an amount of time in between two local date/time.
    - ```
      Period period = today.until( 18thBirthday );
      period.getDays();
      ```
- All these code are thread-safe as far as all the variables are local variables.

Date and Time API: Other Classes

- **`TemporalAdjusters`**
  - Utility class that implements various calendaring operations.
    - e.g., Getting the first Sunday of the month.
- **`ZonedDateTime`**
  - Similar to `LocalDateTime`, but considers time zones (time difference) and time-zone rules such as daylight savings.
- **`DateTimeFormatter`**
  - Useful to parse and print date-time objects.
- All public methods are thread-safe in these classes.

Implementing User-Defined (Your Own) Immutable Classes

- Immutable class
  - Defined as a **`final`** class
  - Has **`private final`** data fields only.
  - Has no setter methods.
  - c.f. A Strategy for Defining Immutable Objects
    - https://docs.oracle.com/javase/tutorial/essential/concurrency/imstrat.html
- Clearly state immutability in program comments, API documents, design documents, etc.
  - Use {frozen} or {immutable} in UML class diagrams
  - Java API documentation does so too.

An Example User-Defined Immutable Class

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

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

  public int getLast4Digits(){ return last4Digits; }

  public String toString(){
      return first3Digits + "-" + middle2Digits + "-" + last4Digits;
      // Multiple steps, but thread-safe
      // cancat() is thread-safe
      // Those 3 data fields are immutable  }

  public Boolean equals( SSN anotherSSN ){
      if( this.toString().equals(anotherSSN.toString()) ){ return true; }
      else{ return false; }
      // Multiple steps, but thread-safe
      // toString() and equals() are thread-safe
      // "this" and "anotherSSN" 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 as an atomic code.
  - 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.

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

```
public class Person {
  private SSN ssn;     // Shared variables

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

  public SSN getSSN(){// 2 Steps, but thread-safe.
    return ssn;       // No race condition occurs in between the
                      // 2 steps even if a context switch happens
                      // there, because:
                      //   * Person has no setters.
                      //   * SSN has no setters. }  }

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 or may not be thread-safe.
  - The code below is NOT thread-safe; it requires thread synchronization.

```
public class Person {
  private SSN ssn;     // Shared variables

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

  public SSN setSSN(SSN ssn){
    this.ssn = ssn;    // 2 steps. NOT thread-safe.
                       // A race condition occurs in between the
                       // 2 steps if setSSN() is called there. }

  public SSN getSSN(){// 2 steps. NOT thread-safe.
    return ssn;        // A race condition occurs in between the
                       // 2 steps if setSSN() is called there. }  }
```

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

HW 14

- Implement your own immutable class:
  - ```
    public final class Address {
      private final String street, city, state;
      private final int zipcode;
      ... }
    ```
  - Define a constructor that takes 4 parameters and sets up an address.
  - Define getters, **`equals()`** and **`toString()`**
  - Define **`change()`** to change the current address
    - ```
      public Address change(String street, String city,
                            String state, int zipcode){
          return new Address(street, city, state, zipcode);  }
      ```
    - It sounds like a setter, but it is NOT. It creates a new instance and returns it.
      ```

```
public class Customer {
  private Address address;    // Shared variable

  public Person(Address addr){ address = addr; }

  public Address setAddress(Address addr){
    address = addr;           // Customer needs a setter.
                              // 2 Steps. NOT thread-safe.
                              // A race condition occurs in between
                              // the 2 steps if setAddress() is
                              // called there. }

  public Address getAddress(){ // 2 Steps. NOT thread-safe.
    return address;           // A race condition occurs in between
                              // the 2 steps if setAddress() is
                              // called there. }  }
```

```
Customer customer = new Customer( new Address( ... ) );
customer.getAddress();
customer.setAddress( new Address ( ...) );
customer.setAddress( customer.getAddress().change( ... ) );
```

`Customer` requires thread synchronization to guard `address`, although `Address` does not.

- Turn in
  - immutable `Address`
  - thread-safe `Customer`
  - `Runnable` class whose `run()` calls `Customer`'s `setAddress()` and `getAddress()`
    - You can replace the `Runnable` class with a lambda expression, if you like
  - Test code to create and run multiple threads
- Deadline: Nov 6 (Tue) midnight

Performance Implication

- An immutable object makes a bigger difference in performance
  - As more threads read data from the object more often.
- If you are interested, compare the performance of
  - Immutable `Address` and
  - Mutable `Address` that performs thread synchronization in its setters and getters.
  - Immutable `Address` is approx. 25% faster on my machine.

Well, Not All Classes can be Immutable…

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

An Example: `String` and `StringBuilder`

- Both represent string data.
- **`String`**
  - Immutable: Its state never change.
  - Thread-safe
  - Faster to run read operations (getters).
- **`StringBuilder`**
  - Mutable: Its state can change through its methods.
  - Not thread-safe
  - A LOT faster to perform write operations (setters).

```
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.

```
ArrayList<String> emailAddrs = ...;
```

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

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

- The latter code can run 20-100% faster depending on the number of collection elements.

- Use `String` (immutable class) for read operations
- Use `StringBuilder` (mutable class) for write operations

- `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 locking.
  - Client code of `StringBuffer` may still require locking.
  - 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 locking.
  - Use this class
    - regardless of single-threaded or multi-threaded apps.

Appendix: NIO-based File/Path Handling and Try-with-resources Statement

(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)

- Parameter values are handled 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 or easier-to-use 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.
            Throw an exception if an error occurs.
        }catch(...){
            Error-handling code 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();
  }
  ```

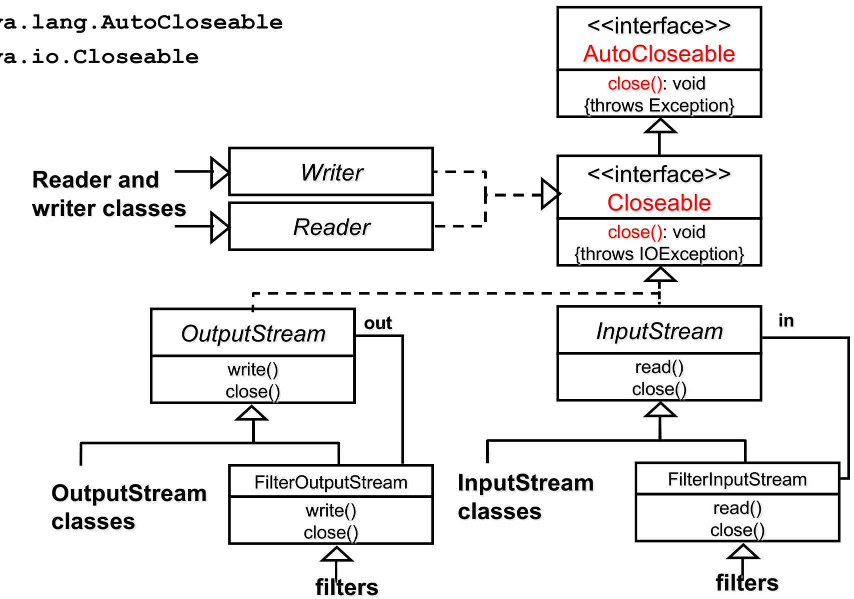

(2) Try-with-resources Statement

- 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 internally 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) <u>AFTER close() is called on **`reader`**.</u>
    ```