Prime Factorization

- Describing a number as a product of prime factors
  - 6 = 2 * 3 84 = 2 * 2 * 3 * 7,
  - 2489 = 19 * 131 8633 = 89 * 97

- Sample code: `PrimeFactorizer`
  - Performs prime factorization for a given number (`dividend`)

```
class PrimeFactorizer {
  protected long dividend;
  protected LinkedList<Long> factors;
  public PrimeFactorizer(long dividend){ ... }
  public void generatePrimeFactors(){ ... }
  public LinkedList<Long> getPrimeFactors(){ return primes };
```

  - Client code (single-threaded)

```
PrimeFactorizer fac = new PrimeFactorizer(84);
fac.generatePrimeFactors()
fac.getPrimeFactors();        // returns [2, 2, 3, 7]
```


Sample Code: `PrimeFactorizer`

- Performs prime factorization for "`dividend`"

```
class PrimeFactorizer {
  protected long dividend;
  protected LinkedList<Long> factors;

  public PrimeFactorizer(long dividend){ ... }
  public void generatePrimeFactors(){ ... }
  public LinkedList<Long> getPrimeFactors(){ ... }
```

- `generatePrimeFactors()`
  - Identifies each prime number in the range of 2 and `dividend`
    - e.g., When the dividend is 84, the factorization range is [2, 84]
  - Examines if it is a factor of `dividend`

Concurrent Prime Factorization

- Single-threaded
  - Identifies each prime in the range of 2 and `dividend`
    - e.g., When the dividend is 84, the factorization range is [2, 84]
  - Examines if it is a factor of `dividend`
    - e.g., Found prime factors: 2, 2, 3, 7

- Multi-threaded
  - Identifies each prime in the range of 2 and `sqrt(dividend)`
    - e.g., When the dividend is 84, the factorization range is [2, 9]
  - Splits the range and assigns each split range to a thread
    - [2, 9] to be divided to [2, 4] and [5, 9], if # of threads is 2.
  - Allows each thread to find prime factors in its (assigned) range
    - Thread #1 to find 2, 2, 3 in [2, 4]
    - Thread #2 to find 7 in [5, 9]

Sample Code: `RunnablePrimeFactorizer`

- Performs prime factorization for "`dividend`"

```
class RunnablePrimeFactorizer extends PrimeFactorizer
                              runs Runnable {
  protected long dividend, from, to;
  protected LinkedList<Long> factors;

  public PrimeFactorizer(long dividend, long from, long to){ ... }
  public void generatePrimeFactors(){ ... }
  public void run(){ generatePrimeFactors() } }
```

- Client code (multi-threaded)

```
r1 = new RunnablePrimeFactorizer(84, 2, Math.sqrt(84)/2);
r2 = new RunnablePrimeFactorizer(84, 1+Math.sqrt(84)/2,
                                 Math.sqrt(84) );

t1 = new Thread(r1);
t2 = new Thread(r2);
t1.start(); t2.start();
```

Note

- `RunnablePrimeFactorizer` may not return the complete set of prime factors.
  - Generates 19 only (not 131) for 2489
    - even though 2489 = 19 * 131
  - Factorization range: [2, `sqrt(dividend)`]
    - e.g., When the dividend is 2489, the factorization range is [2, sqrt(2489)] = [2, 49]
  - The complete set of prime numbers can be derived from generated factors.
    - e.g., 131 can be derived by 2489/19.

Some Context to Prime factorization

- Prime factorization is a mathematical foundation in RSA encryption algorithm (used in SSL, SSH, etc.)
  - SSL is used to access https://… web sites.
- The security of RSA relies on the fact that…
  - it's easy (quick) to calculate a product of two big primes, but really hard (time consuming) to factorize a big composite number; e.g., 2489 = 19 * 131
- Key generation
  - Prepares big primes, *p* and *q*.
  - Generates a public key, which contains the product of *p* and *q*.
  - Generates a private key, which can also be derived from *p* and *q*.
  - Erases *p* and *q*.
- Encryption and decryption
  - Encrypts a plain message with the public key
  - Decrypts an encrypted message with the private key
  - You need *p* and *q* if you try to convert an encrypted msg back to a plain msg without the private key.

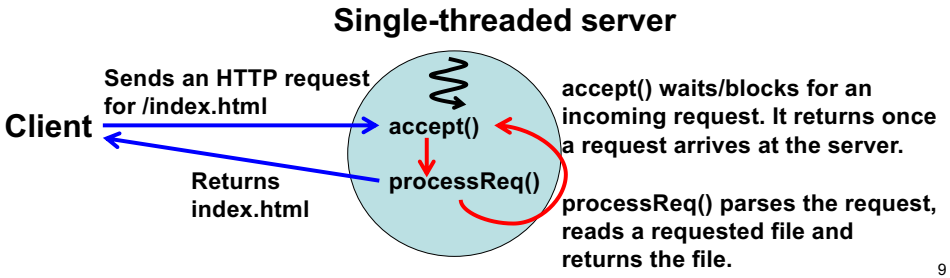
- RSA-2048
  - Uses a 2048-bit binary number for a big composite number (617 decimal digits long).
- Can look into a public key with many/any tools like openssl
  - Composite number in Google's public key (in hex)
    - 9FA1E1B43B3A570ED0CF54BCCD18D8B2121331A44C373D093EEF73DD6423618E951FE46C8 D4052626EDE0E82BF4C2ACF86FD413E81757484F9603150CFF293899FD4786426D6D2C2E71 B01002D82AD220B5BBA9830D71F6B25FCD501E152921ABC8861875154776E6651640079B1C 1C9B1C90B7A050CA45E5EC63647ED88966D55C8BF6513DA06B1679198D909B247F9C6A9C7 4BFD8660532CF5401B2205E53C05D5A95D5D3DFAED2EFA4061A7E949C8D0EE42B9AEC6535 2435666CFBACD248114DACEFC96E20D7B8C616D3494F2E3752A957ED367C07BE8E642B2AA 15C496E5561EC8D160DC0C5C08AD25A250415CF62D39835838F712BC63BB6987CB5BC2FF
- RSA-768 has been broken in 2009
  - Binary 768-bit, 232 decimal digits long
  - $50k awarded.
- $200,000 award if you break RSA-2048 ☺

Recap: Why Threads?

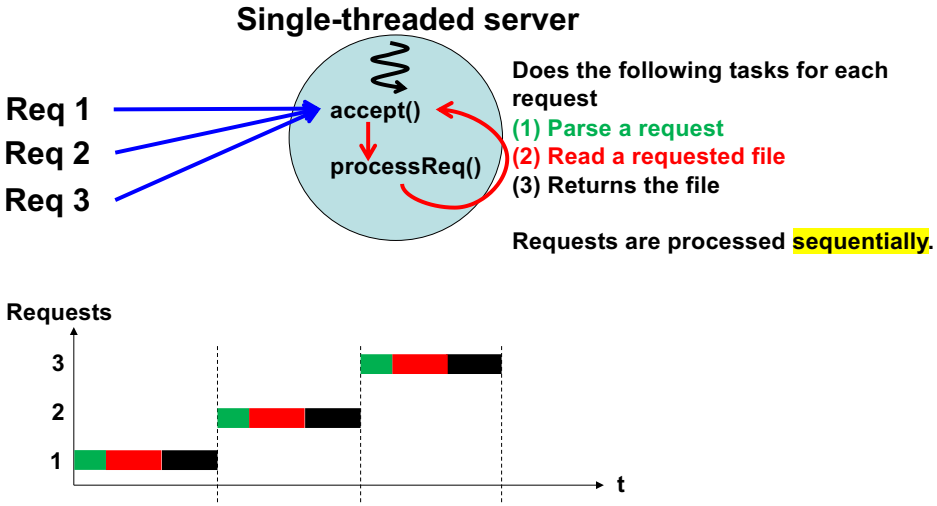
- Improvements in…
  - Performance
    - Division of labor over multiple threads
      - c.f. MCTest, RunnablePrimeGenerator, RunnablePrimeFactorizer
    - Possible on multi-core CPUs
  - Responsiveness/availability
    - Threads can process concurrent tasks in a fair way.
    - Threads allow a program to continue running even if a part of it is blocked for I/O or is performing a long operation.
    - Possible on both single-core and multi-core CPUs

Responsiveness/Availability Improvement

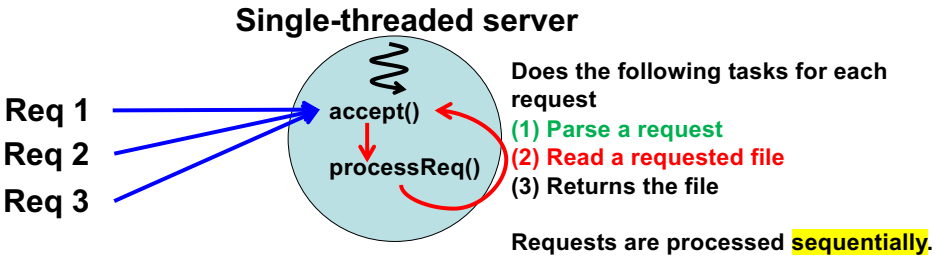
- Suppose you are developing a web server.
  - Receives an HTTP request that a client (browser) transmits to request an HTML file.
  - Returns the requested file to the client.
- What if the server receives multiple requests from multiple clients simultaneously?
  - If the server is single-threaded,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

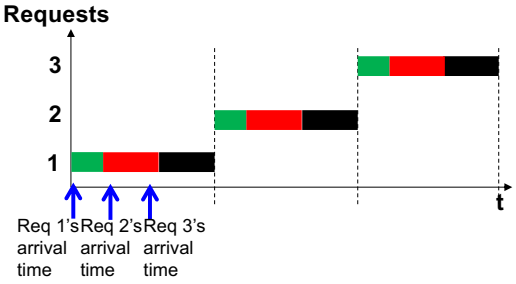


Single-threaded Web Server



Single-threaded Web Server



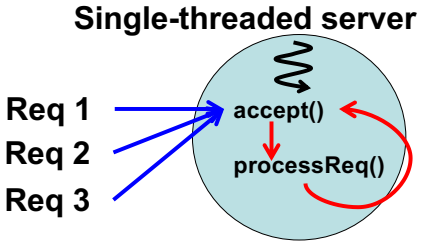


Response time for clients can vary a lot.

Request 3 may be treated unfairly even if its arrival time is very close to Request 1's.

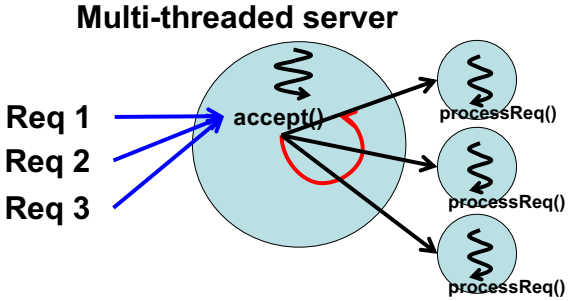
The server should be more responsive (or available sooner) to start processing Request 3 earlier.

Concurrent (Multi-threaded) Web Server



The server does the following tasks for each request
(1) Parse a request
(2) Read a requested file
(3) Returns the file

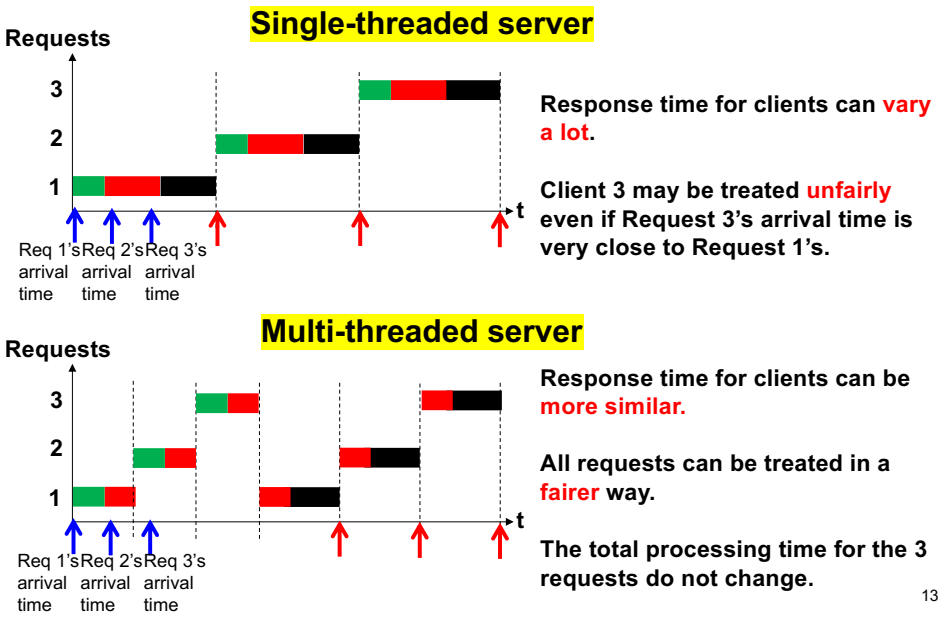
Requests are processed sequentially.



Each thread does the following tasks for a request.
(1) Parse a request
(2) Read a requested file
(3) Returns the file

Requests are processed concurrently.

Single- v.s. Muti-threaded Web Servers

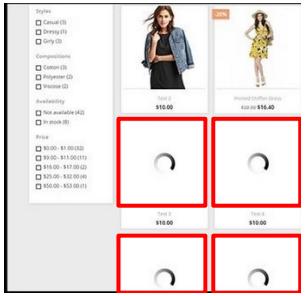


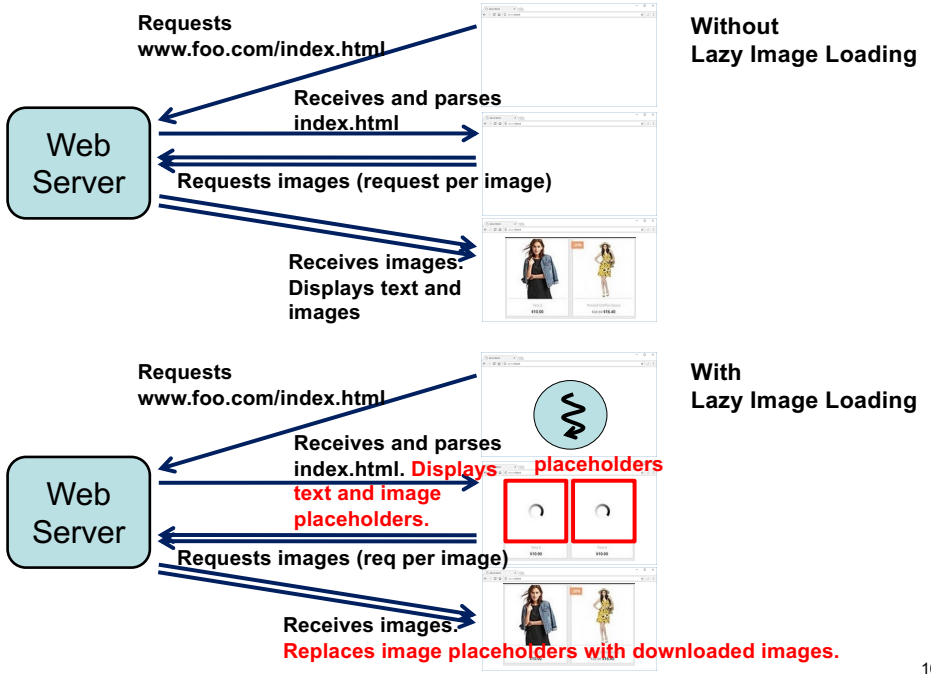
Recap: Why Threads?

- Improvements in…
  - Performance
    - Division of labor over multiple threads
      - c.f. MCTest, RunnablePrimeGenerator (HW 5), RunnablePrimeFactorizer
    - Possible on multi-core CPUs
  - Responsiveness/availability
    - Threads can process concurrent tasks in a fair way.
    - Threads allow a program to continue running even if a part of it is blocked for I/O or is performing a long operation.
    - Possible on both single-core and multi-core CPUs

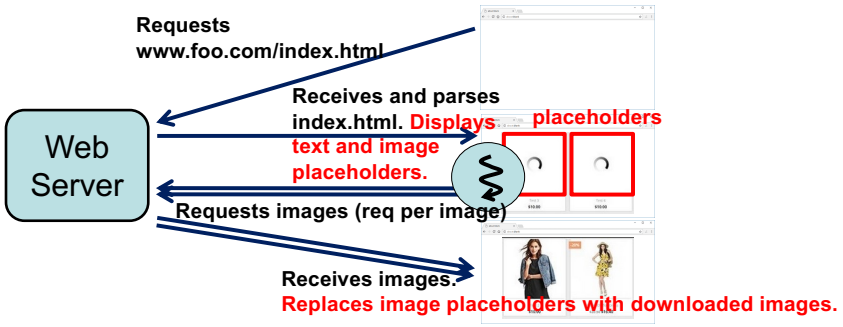
Responsiveness Improvement (Cont'd)

- Suppose you are implementing "lazy image loading" for a web browser.
- When a browser just downloaded a web page (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 a blank browser window until all text and images are downloaded and displayed.
  - Replaces the bounding box with the real (downloaded) image.



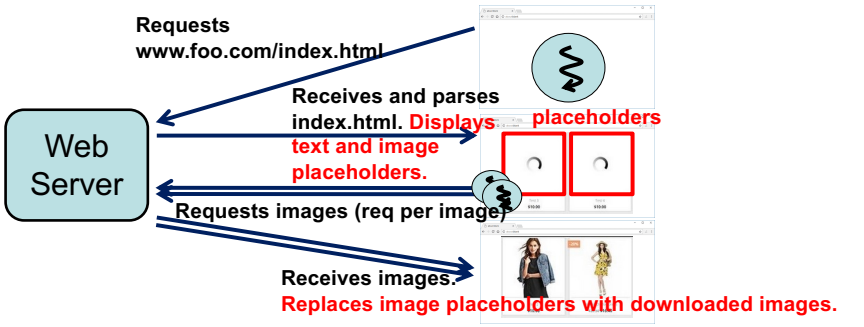


If the Browser is Single-threaded…



- The main thread takes care of
  - Displaying text and image placeholders
  - Downloading and displaying images, and
  - Responding to inputs from the user.
- During the image downloading process, the browser never respond to your inputs
  - Entering a URL
  - Clicking a button
  - Opening a new tab
  - Adjusting the window size

If the Browser is Multi-threaded…



- The main thread takes care of
  - Displaying text and image placeholders and
  - Responding to inputs from the user.
- Extra (download) threads take care of downloading and displaying images.
- During the image downloading process, the browser can still respond to your inputs
  - Entering a URL
  - Clicking a button
  - Opening a new tab
  - Adjusting the window size

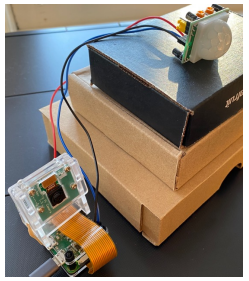
– Responsiveness/availability

- Threads allow a program to continue running even if a part of it is blocked for I/O or is performing a long operation.
  - File downloading: I/O operation, which can be long-running

Another Scenario

- Suppose you are implementing a security camera
  - With a Raspberry Pi, infrared sensor and camera, for example.

```
while true{
  if(irSensor.detectedMotion()){
    vidFile = camera.shootVideo(...);
    mp4File = mediaCoverter(vidFile, ...);
    cloudDB.upload(mp4File, ...);
}
```



- If this logic runs with a single thread…
  - During the file conversion and upload, the IR sensor cannot perform motion detection.

– Responsiveness/availability
  - Threads allow a program to continue running even if a part of it is blocked for I/O or is performing a long operation.
    – File conversion and upload: I/O operations, which can be long-running

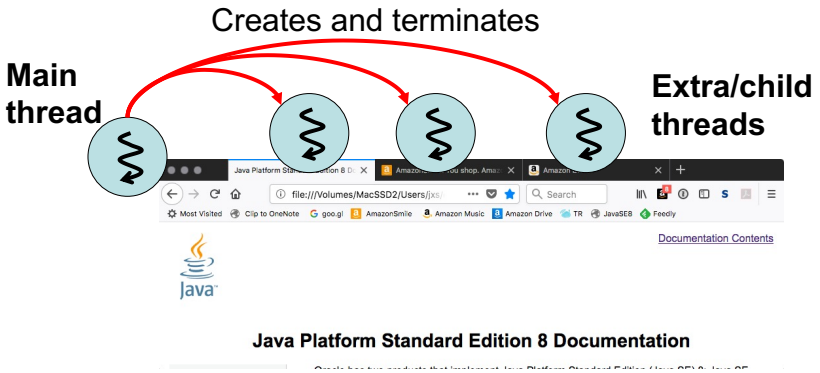
Thread Termination

Thread Termination

- ***Implicit*** termination
  – A thread automatically triggers its own death when run() returns.
    - Once a thread starts executing run(), it will complete a concurrent task and then returns run().
      – c.f. GreetingRunnable, MCTest, RunnablePrimeGenerator, RunnablePrimeFactorization

- ***Explicit*** termination
  – A thread explicitly terminates another thread.
    - when a certain condition is satisfied.
    - even if an on-going concurrent task is not completed.

Examples of Explicit Termination (1)

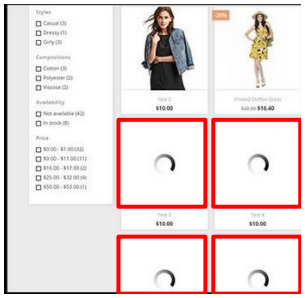


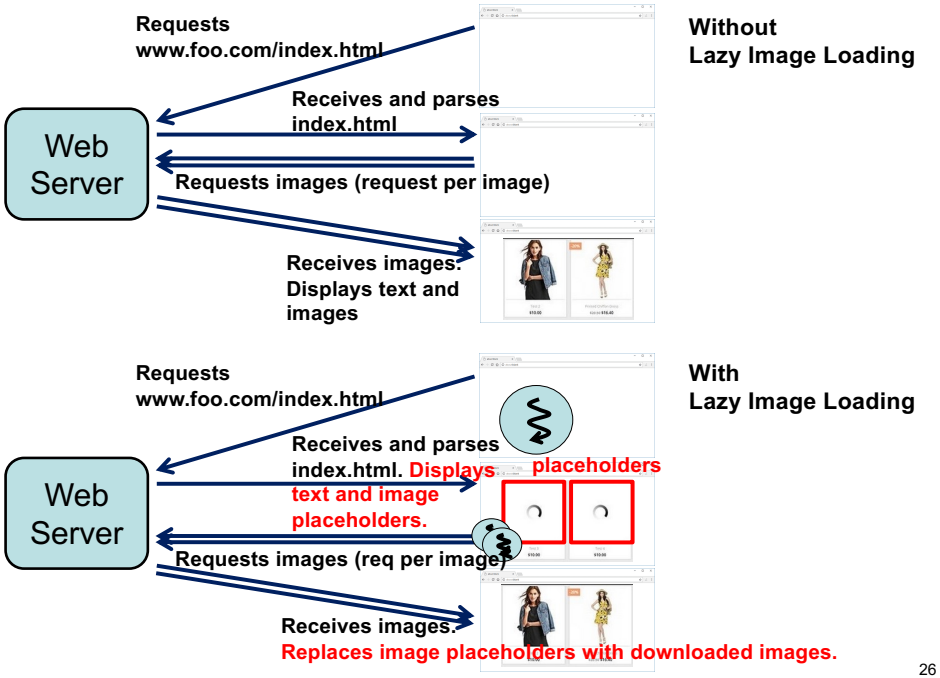
When the user opens a new tab, the main thread creates a new (child) thread and assigns it to the tab.

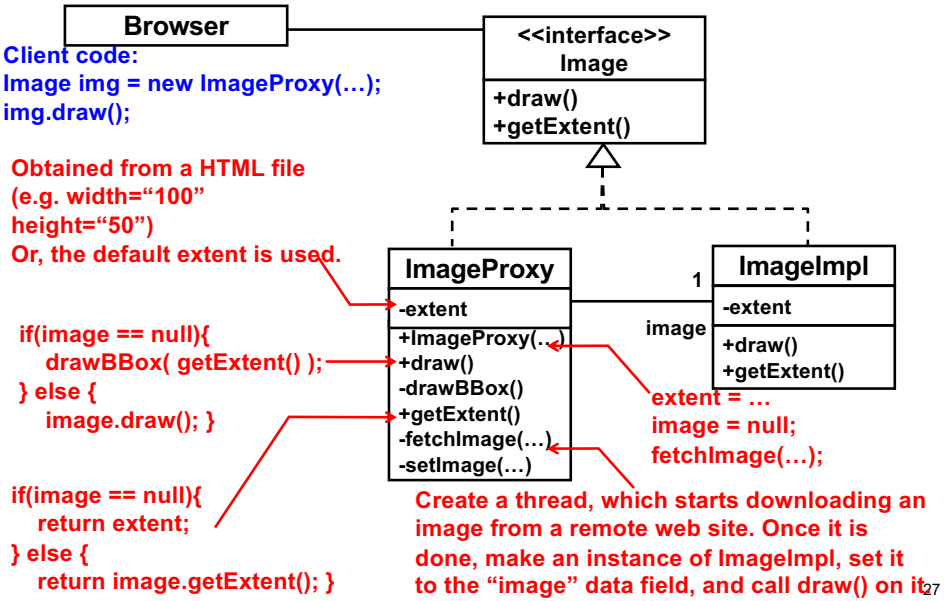
When the user tries to close the browser window, the main thread needs to terminate all other threads (child threads) that are associated with tabs.

Examples of Explicit Termination (2)

- 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 a blank browser window until all text and images are downloaded and displayed.
  - Replaces the bounding box with the real (downloaded) image.

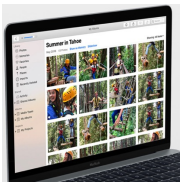






Examples of Explicit Termination (3)

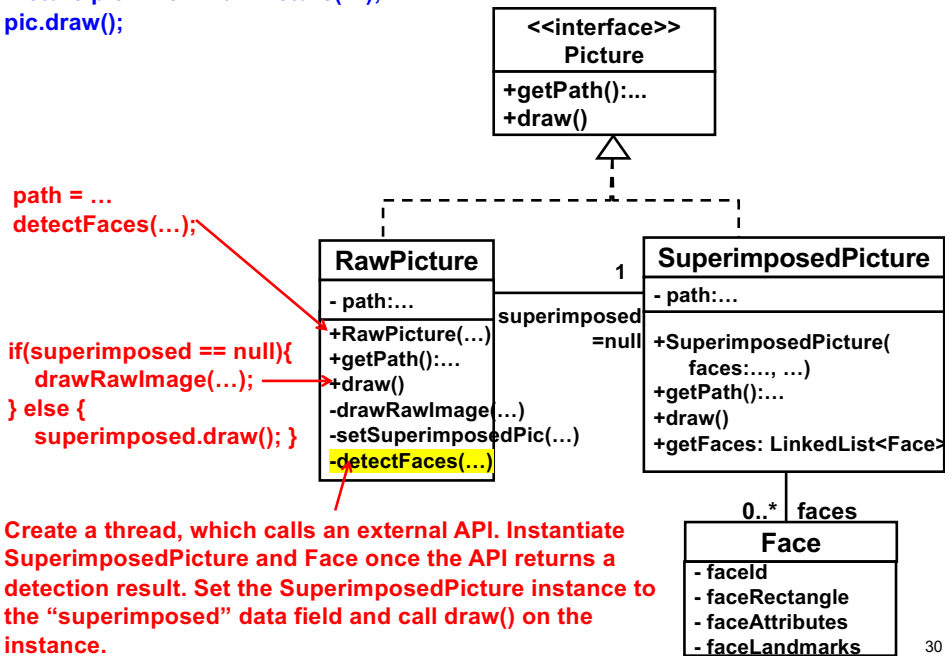
- Suppose you are implementing an app to organize, edit and analyze pictures.
  - When the app loads a raw picture, it superimposes a rectangle on a human face in the picture by (dynamically) calling an external face detection/recognition API.
    - e.g., Microsoft Azure Face API, Google Cloud Vision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

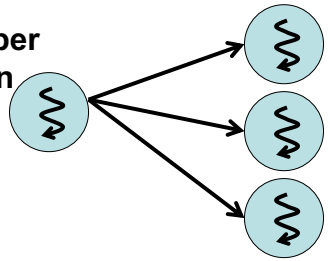
- Lazy loading of detection results
  - Show the user a raw picture first.
  - Call a face detection API.
  - Receive a detection result.
  - Replace the raw picture with a superimposed one, which contains a detection result.

Client code (app):
Picture pic = new RawPicture(…);
pic.draw();



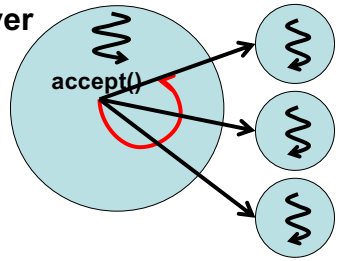
Examples of Explicit Termination (4)

Prime number generation



When the user tries to stop generating primes, the main thread needs to terminate the (worker) threads that are generating primes.

Multi-threaded web server



When a server administrator tries to shut down the server, the main thread needs to terminate the (worker) threads that are processing requests.

Explicit Thread Termination

- ***Explicit*** termination
  - A thread explicitly terminates another thread.
    - when a certain condition is satisfied.
    - even if an on-going concurrent task is not completed.
  - Two approaches
    - With a flag
    - With a thread interruption

Explicit Thread Termination with a Flag

- Define a flag in a Runnable class.

```
public class MyRunnbable implements Runnable{
    private boolean done = false;
    ...
    public void run(){
        while(!done){
            ... // Concurrent task is written here.
        }
    }
    public setDone(){ done=true; }
}
```

- Have a soon-to-be-killed thread periodically check the flag to determine if it should stop/die.
  - Let run() return once "done==true" is detected.
- Stop the thread by flipping the flag to inform that the thread should die.

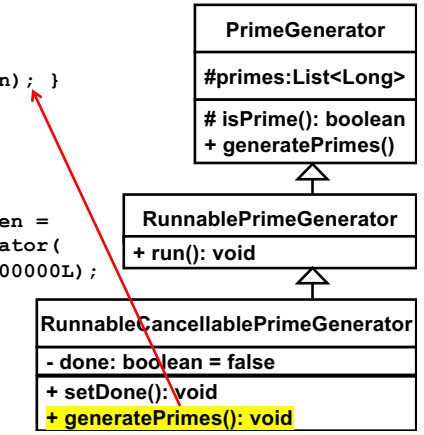
Sample Code: RunnableCancellablePrimeGenerator

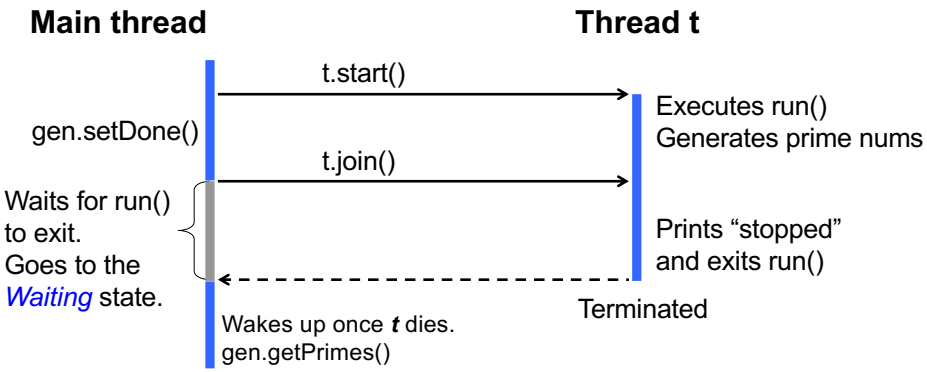
- Define and use a flag to stop generating prime numbers.

```
for (long n = from; n <= to; n++){
  if(done){
      System.out.println("Stopped");
      this.primes.clear();
      break;
  }
  if( isPrime(n) ){ this.primes.add(n); }
}
```

- Client code

```
RunnableCancellablePrimeGenerator gen =
    new RunnableCancellablePrimeGenerator(
                                    1L,1000000L);
Thread t = new Thread(gen);
t.start();
gen.setDone();
t.join();
gen.getPrimes().forEach(...);
```






```
RunnableCancellablePrimeGenerator gen =
       new RunnableCancellablePrimeGenerator(1L,1000000L);
Thread t= new Thread(gen);
t.start();
gen.setDone();
t.join();
gen.getPrimes().forEach(...);
```


Explicit Thread Termination via Interruption

- **Thread.interrupt()**
  - Used to interrupt another thread.

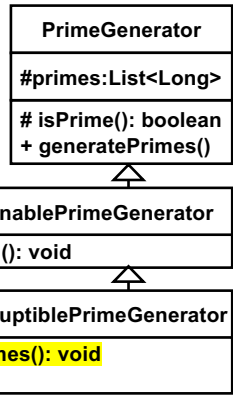
```
Thread thread = new Thread(aRunnable);
thread.start();
thread.interrupt();
```

  - Let that thread know that it is notified via interruption.
- Have a soon-to-be-killed thread periodically detect an interruption to determine if it should stop/die.
  - Let run() return once an interruption is detected.

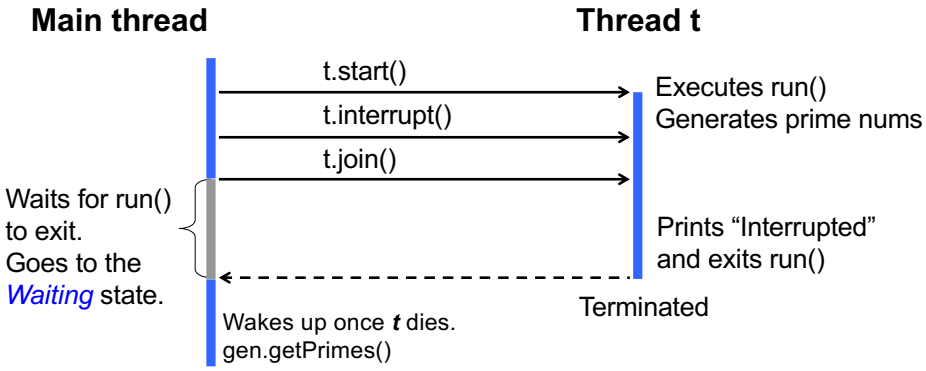
```
public class MyRunnbable implements Runnable{
    ...
    public void run(){
        while(!Thread.interrupted()){
            ...
        }
    } }
```

Sample Code:
RunnableInterruptiblePrimeGenerator

- Detect an interruption from another thread to stop generating prime numbers.
  -
```
for (long n = from; n <= to; n++){
    if(Thread.interrupted()){
        System.out.println("Interrupted");
        this.primes.clear();
        break;
    }
    if( isPrime(n) ){ this.primes.add(n); }
}
```



- Client code
  -
```
RunnableInterruptiblePrimeGenerator gen =
    new InterruptiblePrimeNumberGenerator(1L, 1000000L);
Thread t = new Thread(gen); t.start();
t.interrupt();
t.join();
gen.getPrimes().forEach(...);
```


```
InterruptiblePrimeNumberGenerator gen =
      new InterruptiblePrimeNumberGenerator(1L, 1000000L);
Thread t= new Thread(gen);
t.start();
t.interrupt();
t.join();
gen.getPrimes().forEach(...);
```

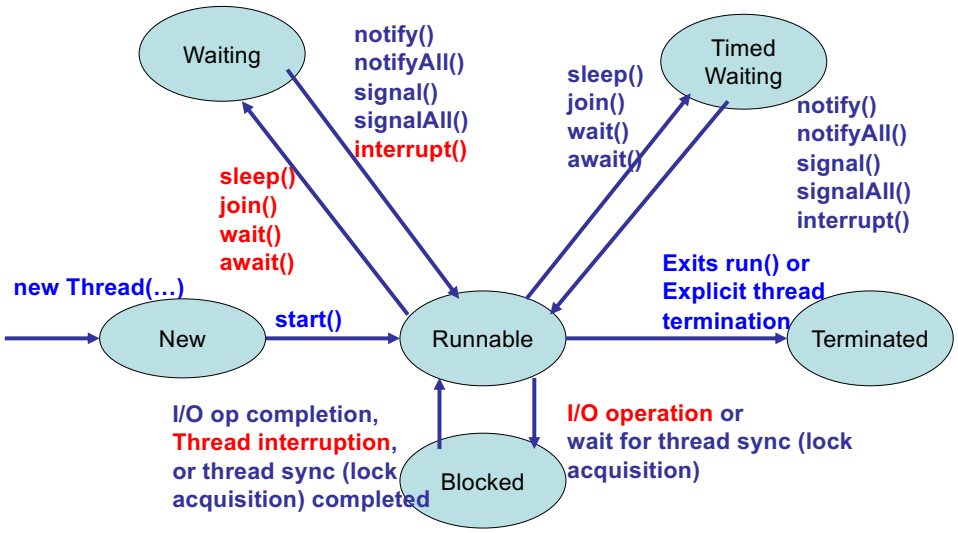

Exercise

- Have the main thread create two or more extra threads (not only one thread)
- Have each extra thread generate primes
- Have the main thread terminate all the extra threads.

Which Termination Scheme to Use?

- Flag-based or interruption-based?
- Both work if `run()` is simple.
- Favor interruption-based scheme if a soon-to-be-killed thread can get into the Waiting or Blocked state.
  - `Thread.sleep()`, `Thread.join()`
    - Change the currently-executed thread's state to Waiting
  - I/O operations
    - Change the currently-executed thread's state to Blocked.
  - These methods can be long-running and interrupted.

States of a Thread



If `Thread.sleep()` is called in `run()`…

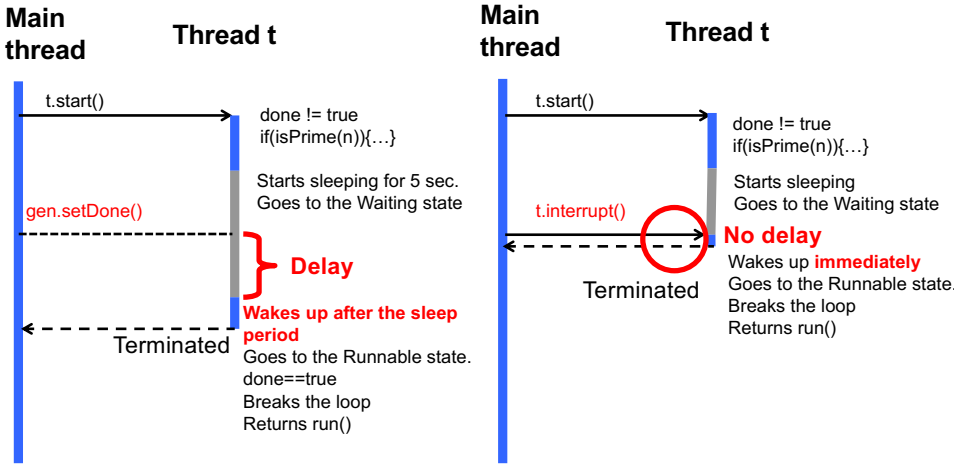
- `RunnableCancellablePrimeGenerator`'s `run()`

```
for (long n = from; n <= to; n++){
  if(done){
      System.out.println("Stopped");
      this.primes.clear();
      break;
  }
  if( isPrime(n) ){ this.primes.add(n); }
  Thread.sleep( Duration.ofSeconds(5) );
```

- `RunnableInterruptiblePrimeGenerator`'s `run()`

```
for (long n = from; n <= to; n++){
  if(Thread.interrupted()){
      System.out.println("Stopped");
      this.primes.clear();
      break;
  }
  if( isPrime(n) ){ this.primes.add(n); }
  Thread.sleep( Duration.ofSeconds(5) );
```


RunnableCancellablePrimeGenerator **RunnableInterruptiblePrimeGenerator**



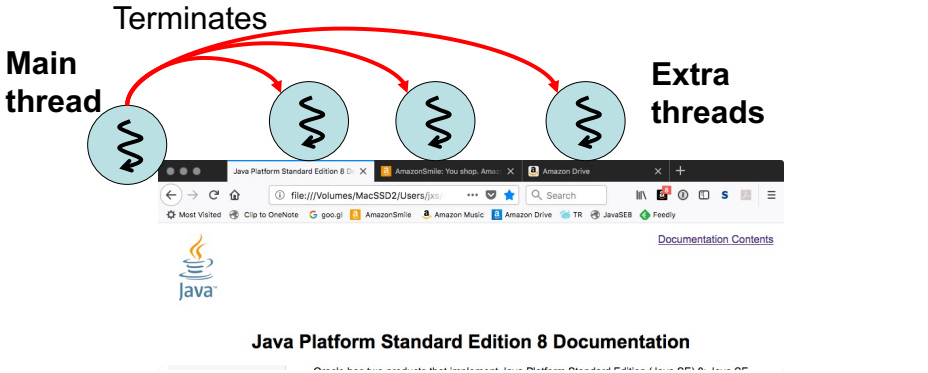
Sleep period: 5 seconds
Less responsive thread termination

Sleep period: Can be less than 5 seconds
More responsive thread termination

If an I/O operation is performed in `run()`…

```
run(){
  for(element: pageElements){
    if(done){ return; }
    download(element);
}
```

```
run(){
  for(element: pageElements){
    if(Thread.interrupted()){return;}
    download(element);
}
```



- **`InputStream`**
  - `public int read() throws IOException`
    - Reads data from a file, a network connection, etc.
    - Gets blocked (i.e., does not return immediately) during this I/O operation.
      - The underlying JVM changes the state of a thread doing this I/O operation (Runnable → Blocked).
      - It changes the thread's state back to Runnable once this I/O operation is completed.
    - Throws an `InterruptedIOException` (subclass of `IOException`) if a thread doing this I/O operation is interrupted.
- **`OutputStream`**
  - `public int write() throws IOException`

- Flag-based approach
  - Cannot trigger thread termination during an I/O operation.
  - Less responsive thread termination
- Interruption-based approach
  - Can trigger thread termination during an I/O operation.
  - More responsive thread termination

Which Termination Scheme to Use?

- Flag-based or interruption-based?
- Both work if `run()` is simple.
- Favor interruption-based scheme if a soon-to-be-killed thread can get into the Waiting or Blocked state.
  - `Thread.sleep()`, `Thread.join()`
    - Change the currently-executed thread's state to Waiting
  - I/O operations
    - Change the currently-executed thread's state to Blocked.
  - These methods can be long-running and interrupted.

Thread Termination Requires Your Attention

- Thread creation is a no brainer.
- (Explicit) Thread termination requires your attention.
  - No methods available in `Thread` to directly terminate threads like `terminate()`.
    - Use:
      - Flag-based OR interruption-based scheme

Deprecated Methods for Thread Termination

- `Thread.stop()` and `Thread.suspend()`
  - NOT thread-safe. **NEVER try to use them.**
  - c.f. "Why Are Thread.stop, Thread.suspend,Thread.resume and Runtime.runFinalizersOnExit Deprecated?"
    - http://docs.oracle.com/javase/1.5.0/docs/guide/misc/threadPrimitiveDeprecation.html