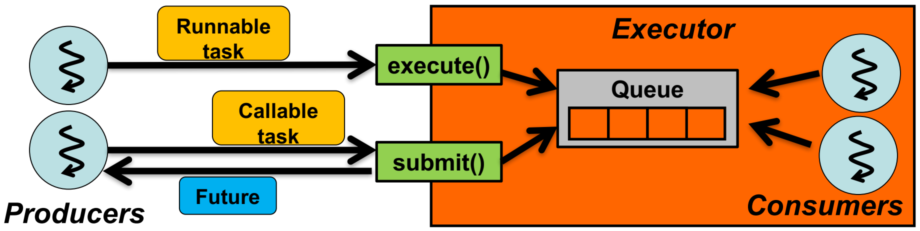
Callable Tasks

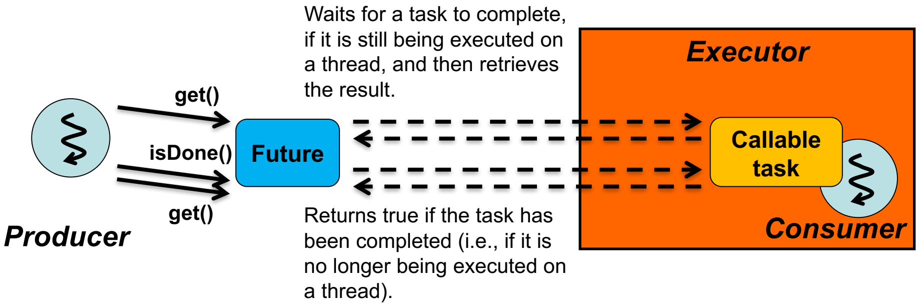
```
CallablePrimeGenerator gen = new CallablePrimeGenerator(...);
ExecutorService executor = Executors.newFixedThreadPool(2);
Future<List<Long>> future = executor.submit(gen);
List<Long> primes = future.get();
```

- `submit()` returns a `Future`, which represents the result of a task.
- An `Executor` can receive `Runnable` and `Callable` tasks simultaneously.
  – Note: A task cannot implement both `Runnable` and `Callable`.



Future

```
public interface Future<T>{
    T get() throws ...;
    T get(long timeout, TimeUnit unit) throws ...;

    boolean cancel(boolean mayInterruptIfRunning);
    boolean isCanceled();
    boolean isDone();   }
```



Waits for a task to complete, if it is still being executed on a thread, and then retrieves the result.

Returns true if the task has been completed (i.e., if it is no longer being executed on a thread).

If You have a Batch of Tasks…

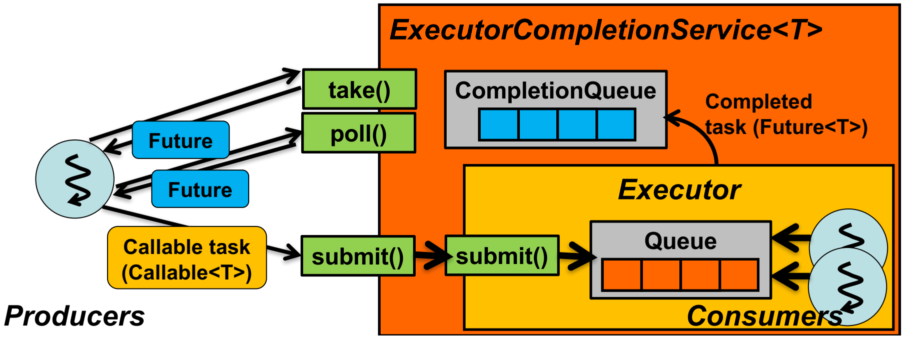
```
ExecutorService executor = Executors.newFixedThreadPool(4);
ArrayList<Future<List<Long>>> futures = new ArrayList<>;

for(int i=0; i<10; i++){
    CallablePrimeGenerator gen = new CallablePrimeGenerator(...);
    futures.add( executor.submit(gen) );
}

for(int i=0; i<10; i++){
    List<Long> primes = futures.get(i).get();
    ... // do something with primes.
}
```

- A `Future's get()` gets blocked (i.e. does not return) if its associated task is not completed yet.
- By default, `Executors` have no mechanisms to return completed tasks as they complete.
  – Need to repeatedly check if each task is completed, if you want to retrieve results as they become available.
  – Call `isDone()` or `get()` with a timeout of zero. A bit tedious.

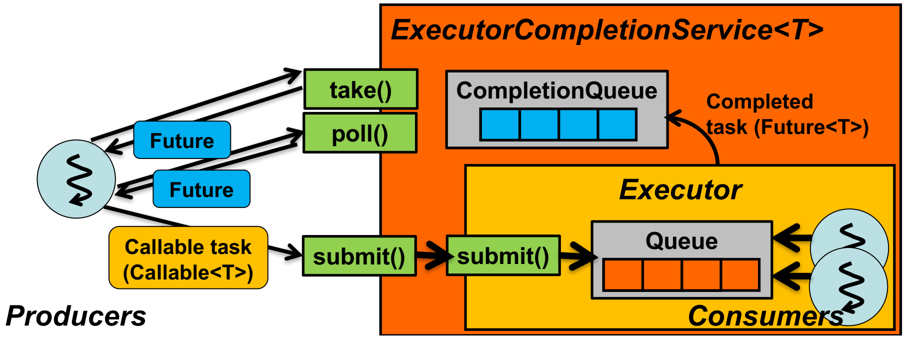
An Extra Type of Executors: `ExecutorCompletionService`

ExecutorCompletionService<T>

- A wrapper of an **Executor**
  - Introduces a *completion queue* atop an **Executor**
    - A queue that contains completed tasks.
- Can return completed tasks as they complete.
- T: Type of a result generated by a task.



- **take()**
  - Retrieves and removes the Future object that represents the next completed task, waiting if none are yet present.
- **poll()**
  - Retrieves and removes the Future object that represents the next completed task, or null if none are present.



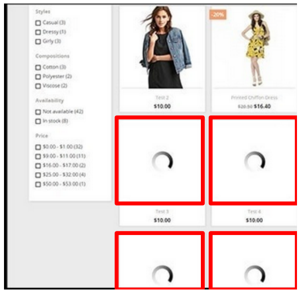
If You have a Batch of Tasks…

```
ExecutorService executor = Executors.newFixedThreadPool(4);
ExecutorCompletionService<List<Long>> completionService
    = new ExecutorCompletionService<>( executor );

for(int i=0; i<10; i++){
    CallablePrimeGenerator gen = new CallablePrimeGenerator(...);
    completionService.submit(gen);
}

for(int compl=0, taskNum=futures.size(); taskNum<compl; compl++){
    Future<List<Long>> future = completionService.take();
    List<Long> primes = future.get();
    ... // do something with primes.
}
```

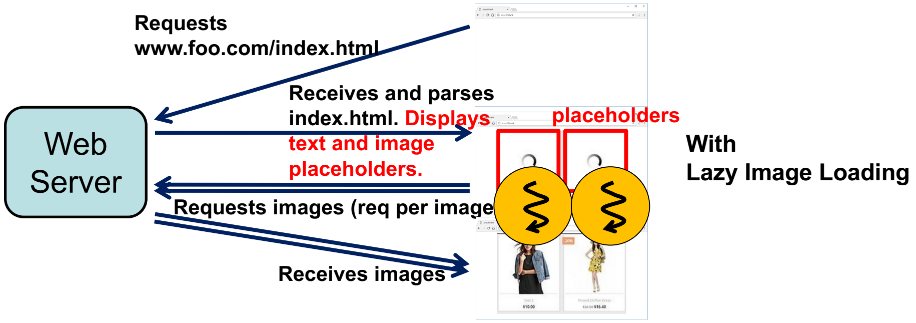
Exercise:
Concurrent Lazy Image Loading

An Example: Lazy Image Loading in a Web Browser

- When an HTML file contains an image(s), a browser
  - Displays a bounding box (placeholder) first for each image
    - Until it fully downloads the image.
      - Most users are not 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.



- Use one thread for each image download
  - One thread for each request-response pair


Requests www.foo.com/index.html
Receives and parses index.html. Displays text and image placeholders.
placeholders
Web Server
Requests images (req per image)
Receives images
With Lazy Image Loading

Recap: *Proxy*

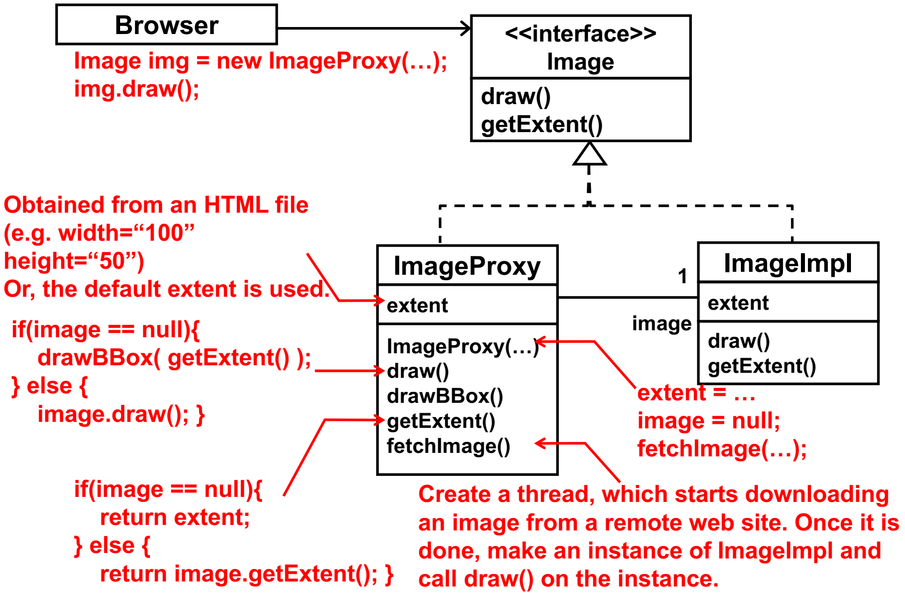


Image img = new ImageProxy(…);
img.draw();

Obtained from an HTML file (e.g. width="100" height="50")
Or, the default extent is used.

if(image == null){
  drawBBox(getExtent());
} else {
  image.draw(); }

if(image == null){
  return extent;
} else {
  return image.getExtent(); }

extent = …
image = null;
fetchImage(…);

Create a thread, which starts downloading an image from a remote web site. Once it is done, make an instance of ImageImpl and call draw() on the instance.

Implementation Strategies (1)

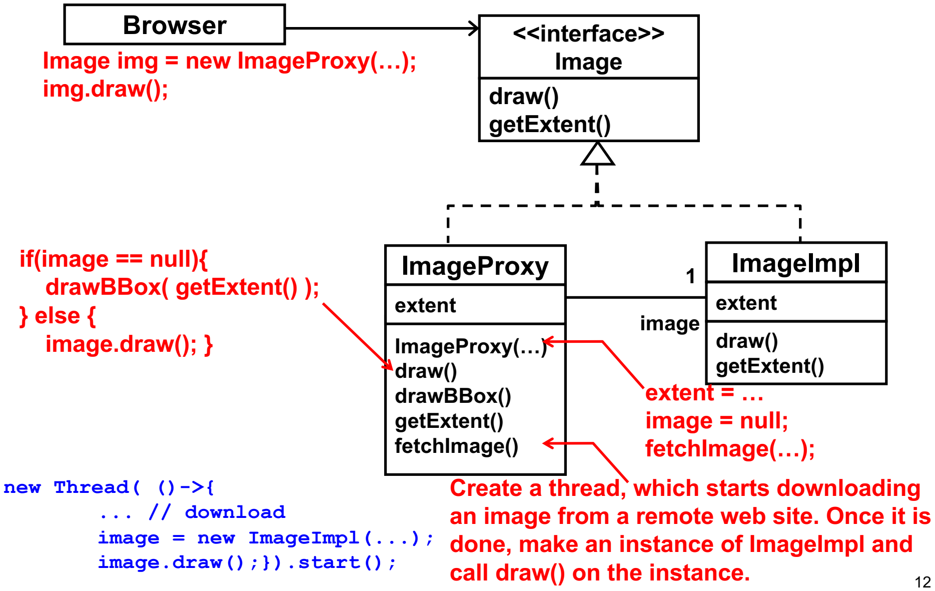


Image img = new ImageProxy(…);
img.draw();

if(image == null){
  drawBBox(getExtent());
} else {
  image.draw(); }

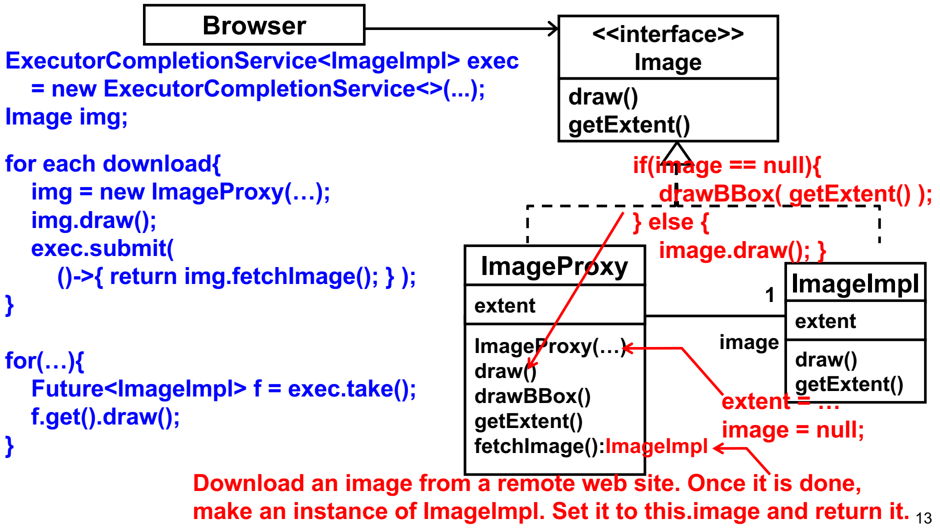
extent = …
image = null;
fetchImage(…);

```
new Thread( ()->{
      ... // download
      image = new ImageImpl(...);
      image.draw();}).start();
```

Create a thread, which starts downloading an image from a remote web site. Once it is done, make an instance of ImageImpl and call draw() on the instance.

Implementation Strategies (2)

- Have Browser initiate downloading each image.



```
ExecutorCompletionService<ImageImpl> exec
   = new ExecutorCompletionService<>(...);
Image img;

for each download{
   img = new ImageProxy(…);
   img.draw();
   exec.submit(
      ()->{ return img.fetchImage(); } );
}

for(…){
   Future<ImageImpl> f = exec.take();
   f.get().draw();
}
```

```
if(image == null){
   drawBBox( getExtent() );
} else {
   image.draw(); }
```

```
extent = …
image = null;
```

Download an image from a remote web site. Once it is done, make an instance of ImageImpl. Set it to this.image and return it.

HW 21

- Pick up your prior HW solution and revise it to use an Executor.
  - You can choose any HW solution.
    - Prime number generation, file caching, access counting, Observer, etc.
  - You can choose any Executor.
    - Replace existing client code like:
      - new Thread(new MyRunnable(...)).start();
    - with new one using an Executor.
  - Make sure to shut down the Executor in the end.