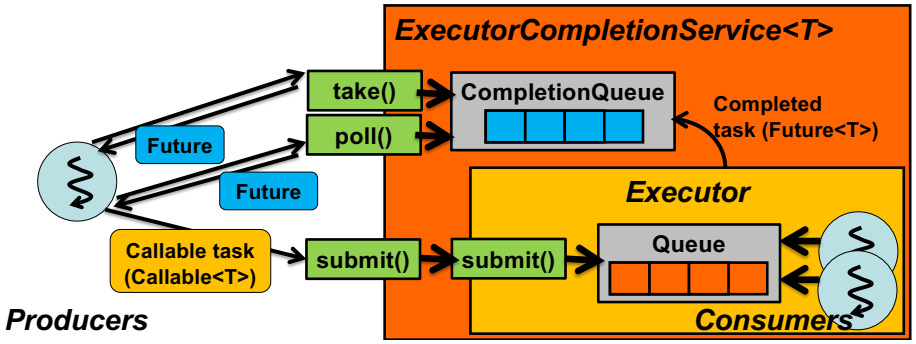
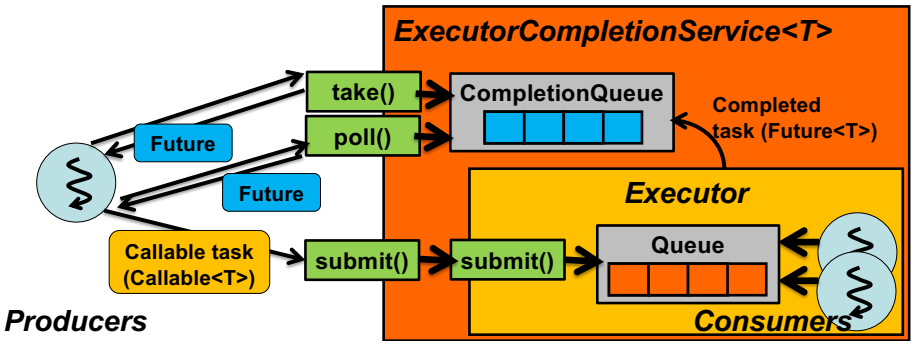
An Extra Type of Executors:
`ExecutorCompletionService`

`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.

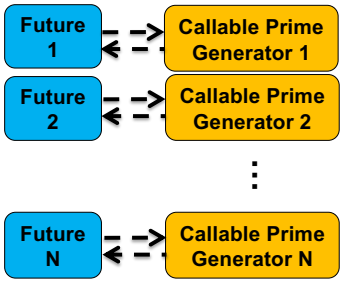


- `take()`
  - Retrieves and removes the `Future` object that represents the next completed task, waiting (i.e., getting blocked) if none are present.
- `poll()`
  - Retrieves and removes the `Future` object that represents the next completed task, or returns null if none are present.



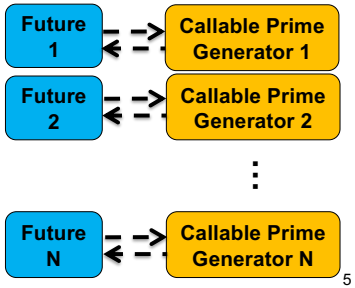
If You have a Batch of Tasks…
CallablePrimeGeneratorBatchTestCompletionService.java

```
ExecutorService executor =
  Executors.newFixedThreadPool(2);

ExecutorCompletionService<List<Long>>
completionService =
  new ExecutorCompletionService<>(executor);

ArrayList<Future<List<Long>>> futures =
  new ArrayList<>;

for(int i=0; i<N; i++){
    futures.add(
        completionService.submit(
          new CallablePrimeGenerator(...)) );
```



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

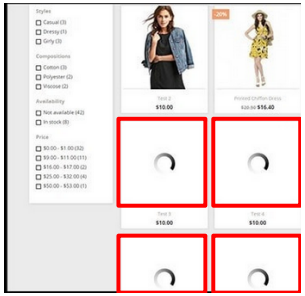
- `ExecutorCompletionService` returns completed tasks as they complete.
  - `take()` returns one of the results that have been generated by Prime Generators.
  - The order of collecting results (from generators) DOES NOT follow the order of generators.

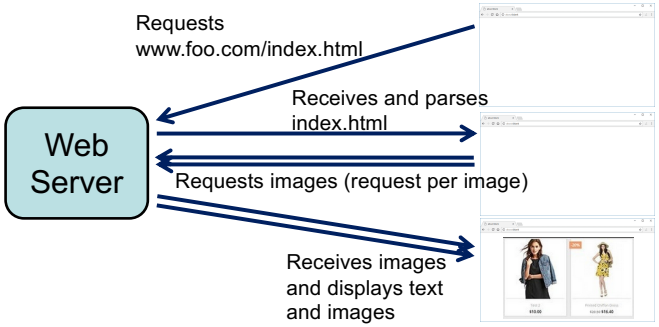


Exercise: Concurrent Lazy Image Loading

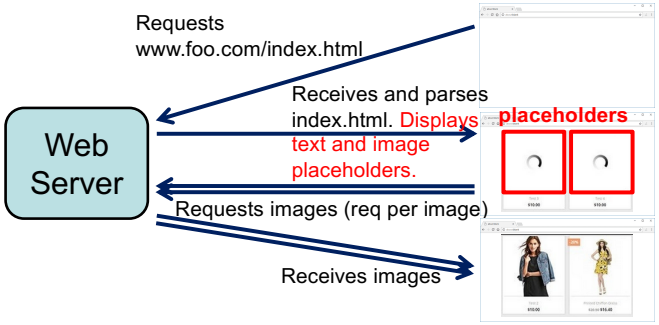
Recap: Lazy Image Loading in a Web Browser

- 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 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.



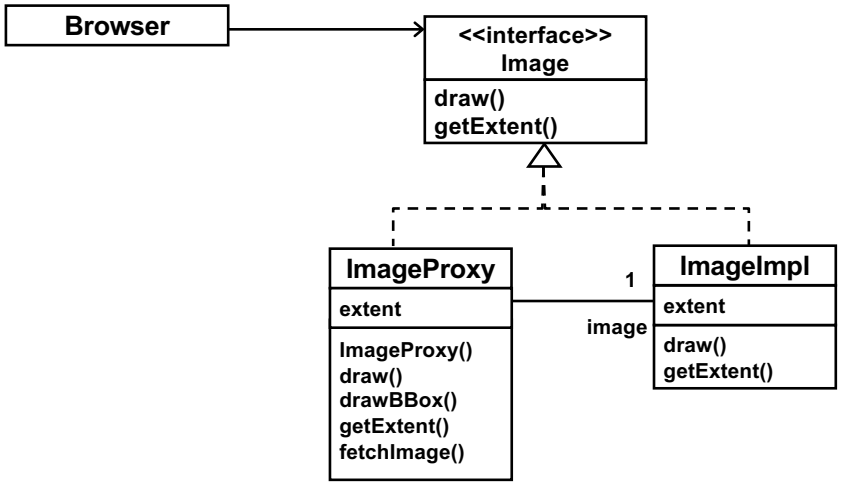


Without Lazy Image Loading



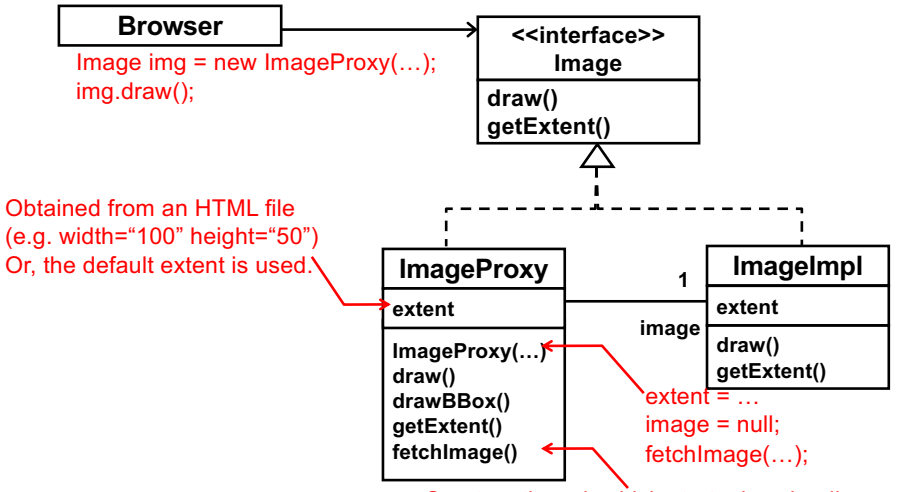
With Lazy Image Loading

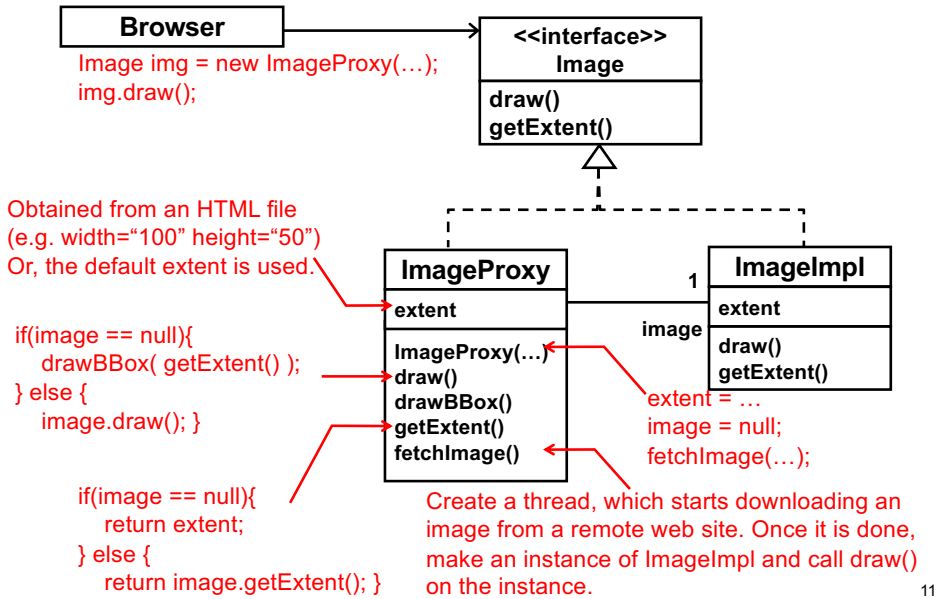
Recap: *Proxy* Design Pattern



Proxy Class

Provides a surrogate or placeholder for another object to control access to it.



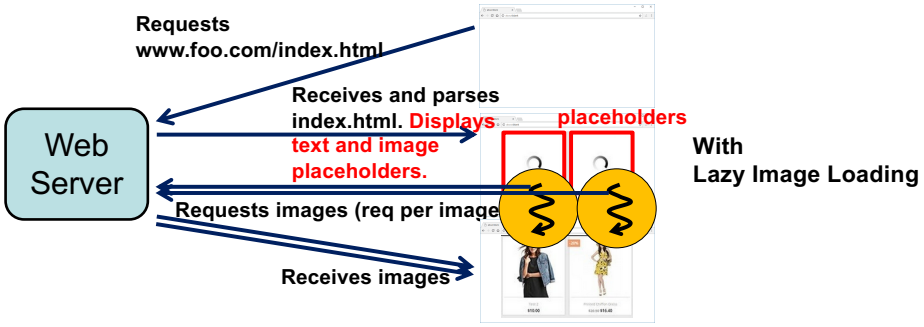


What's the Point?

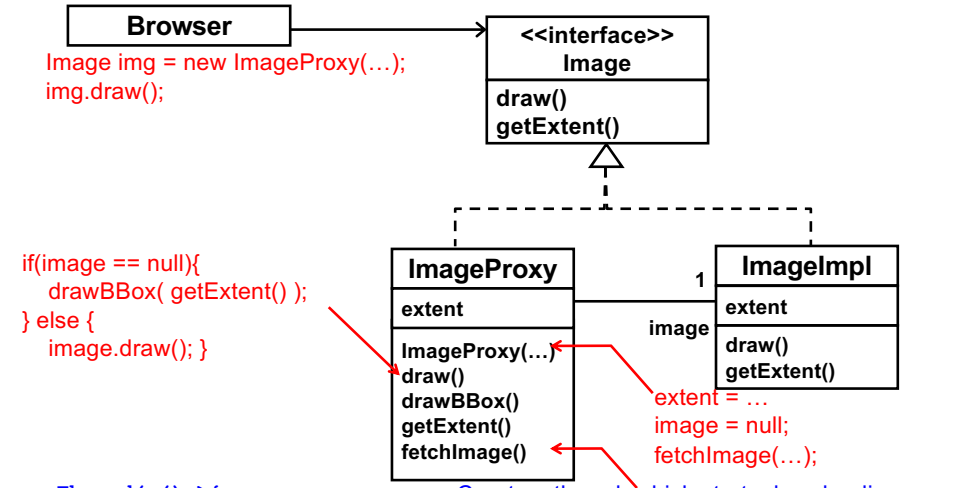
- Decouple (i.e., loosely couple) *bounding box placement (lazy image loading)* and *image rendering*.
- Why is that important?
  - Changes are expected for
    - Image formats that the browser supports.
    - Rendering algorithms
  - Image loading is independent from those changes.
  - Separate *what can change often* from *what wouldn't* to improve maintainability.

Concurrent Lazy Image Loading

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



Implementation Strategies (1)

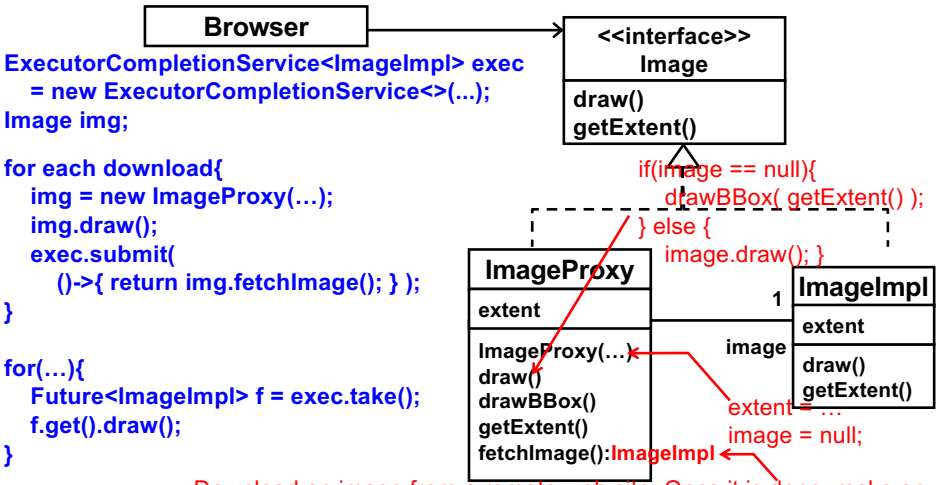



```
new Thread( ()->{
        ... // download
        image = new ImageImpl(...);
        image.draw();}).start();
```

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 each image download.



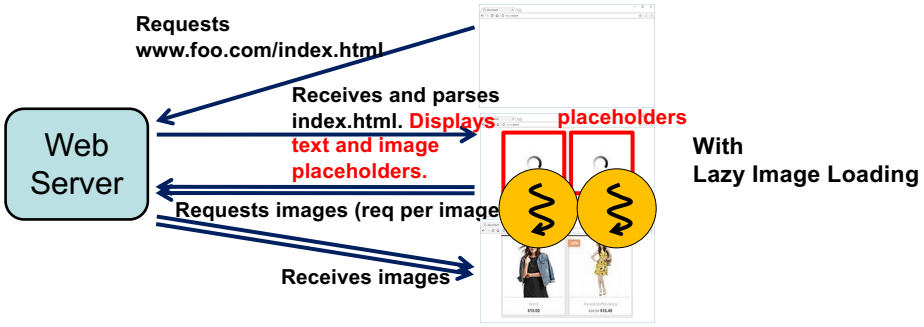
Download an image from a remote web site. Once it is done, make an instance of ImageImpl. Set it to this.image and return it.

An Extra Type of Futures:

`CompletableFuture`

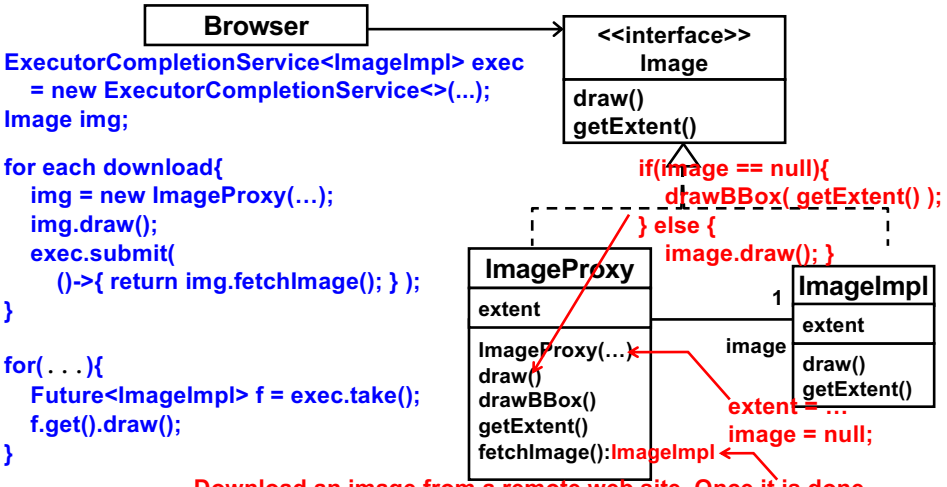
Recap: Lazy Image Loading

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



Recap: Concurrent Image Downloads

- Have Browser initiate downloading each image.

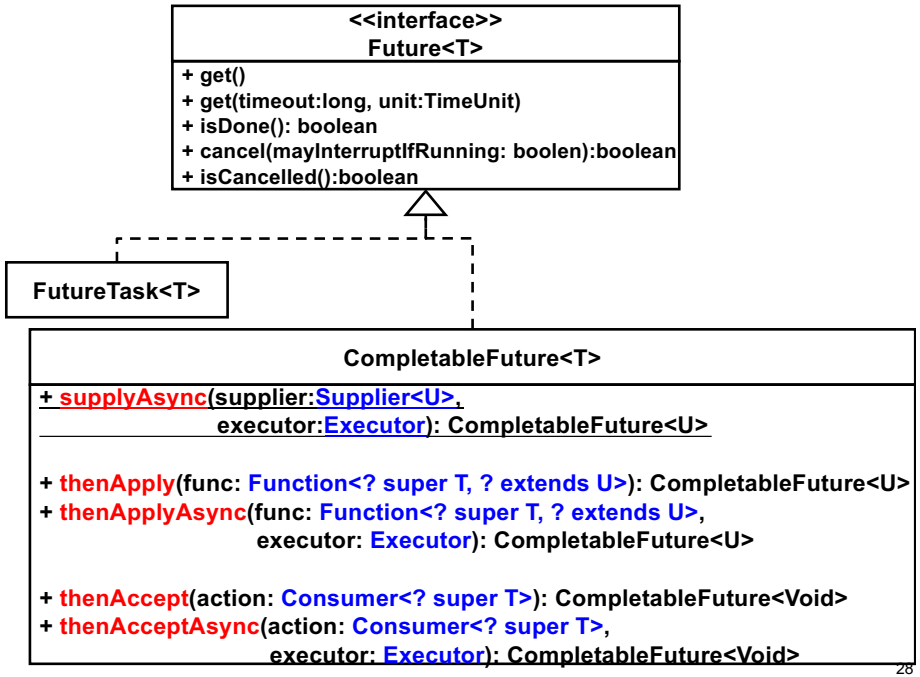



```
ExecutorCompletionService<ImageImpl> exec
    = new ExecutorCompletionService<>(...);
Image img;

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

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

- Can retrieve and process the results of concurrent tasks as those tasks complete.
- Can we state "here is what to do for the result of each task once the task completes" in a declarative manner (not imperative/procedual manner)?

CompletableFuture<T>

- A special type of `Future`.
  - `T`: Type of a task's result
- Allows you to state "here is what to do for the result of each task once the task completes" as a lambda expression
  - in a declarative manner (not imperative/procedual manner)
- Allows you to structure a pipeline of concurrent tasks



Important General-Purpose Functional Interfaces

	Params	Returns	Example use case
Function<T,R>	T	R	Get the price (R) from a Car object (T) Generate a function (R) from another (T)
Consumer<T>	T	void	Print out a collection element (T)
Predicate<T>	T	boolean	Has this car (T) had an accident?
Supplier<T>	NO	T	A factory method. Create a Car object and return it.
UnaryOperator<T>	T	T	Logical NOT (!)
BinaryOperator<T>	T, T	T	Multiplying two numbers (*)
BiFunction<U,T>	U, T	R	Return TRUE (R) if two params (U and T) match.

Important General-Purpose Functional Interfaces

	Params	Returns	Example use case
Function<T,R>	T	R	Get the price (R) from a Car object (T) Generate a function (R) from another (T)
Consumer<T>	T	void	Print out a collection element (T)
Predicate<T>	T	boolean	Has this car (T) had an accident?
Supplier<T>	NO	T	A factory method. Create a Car object and return it.
UnaryOperator<T>	T	T	Logical NOT (!)
BinaryOperator<T>	T, T	T	Multiplying two numbers (*)
BiFunction<U,T>	U, T	R	Return TRUE (R) if two params (U and T) match.

`thenApply(Function<...>)` : receives a function that takes a task's result (T), processes it and returns the processed result (R).

`thenAccept(Consumer<...>)` : receives a function that takes a task's result (T) and processes/consumes it.

Just In Case: Wildcards in Generics

- Function<? super T, ? extends U>
- Consumer<? super T>

- ? super T
  - Any super type (class/interface) of T

- extends U
  - Any sub type (class interface) of U

Concurrent Image Downloads

```
ExecutorService executor = Executors.newFixedThreadPool(4);

Image img;

for each download{
    img = new ImageProxy(...);
    img.draw();
    CompletableFuture<ImageImpl> downloadedImage
      = CompletableFuture<ImageImpl>.supplyAsync(
          ()->{ return img.fetchImage(); },
          executor );
    downloadedImage.thenAccept( (ImageImpl image)-> image.draw() );
}
```


Concurrent Image Downloads

```
ExecutorService executor = Executors.newFixedThreadPool(4);

Image img;

for each download{
    img = new ImageProxy(…);
    img.draw();
    CompletableFuture<ImageImpl> downloadedImage
      = CompletableFuture<ImageImpl>.supplyAsync(
          ()->{ return img.fetchImage(); },
          executor );
    downloadedImage.thenAccept( (ImageImpl image)-> image.draw() );
}
```

States "what to do for the result of each task once the task completes.


```
ExecutorService executor = Executors.newFixedThreadPool(4);
Image img;

for each download{
    img = new ImageProxy(…);
    img.draw();
    CompletableFuture<ImageImpl> downloadedImage
      = CompletableFuture<ImageImpl>.supplyAsync(
          ()->{ return img.fetchImage(); },
          executor )
        .thenAccept( (ImageImpl image)-> image.draw() );
}
```



```
ExecutorService executor = Executors.newFixedThreadPool(4);
Image img;

for each download{
    img = new ImageProxy(…);
    img.draw();
    CompletableFuture<ImageImpl> downloadedImage
      = CompletableFuture<ImageImpl>.supplyAsync(
          ()->{ return img.fetchImage(); },
          executor )
        .thenAccept( (ImageImpl image)-> image.draw() );
}
```

Two tasks are pipelined.

```
ExecutorService executor = Executors.newFixedThreadPool(4);
Image img;

for each download{
    img = new ImageProxy(…);
    img.draw();
    CompletableFuture<ImageImpl> downloadedImage
      = CompletableFuture<ImageImpl>.supplyAsync(
            ()->{ return img.fetchImage(); },
            executor )
        .thenAccept( (ImageImpl image)-> image.draw() );
}
```

- Method with the suffix **`Async`**
  - Takes a task as a LE and an Executor
  - Runs the task with an extra thread in the Executor
- Method without the suffix **`Async`**
  - Takes a task as a LE. Runs it with a thread that executed a previous task.


```
ExecutorService executor = Executors.newFixedThreadPool(4);
Image img;

for each download{
    img = new ImageProxy(…);
    img.draw();
    CompletableFuture<ImageImpl> downloadedImage
      = CompletableFuture<ImageImpl>.supplyAsync(
            ()->{ return img.fetchImage(); },
            executor )
        .thenAccept( (ImageImpl image)-> image.draw() );
}
```

Two tasks (image download and placeholder replacement) are executed on the same thread.

HW 20

- Pick up your prior HW solution and revise it to use an Executor.
  - You can choose any HW solution.
    - Prime generation, access counting, cars, Observer, etc.
  - You can choose any type of 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.
  - If you use prime generation as an application, use **`CompletableFuture`**