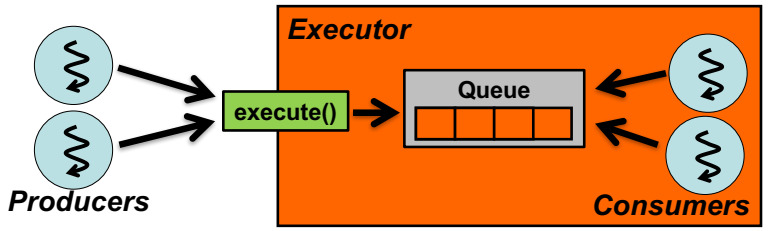
Tasks, Threads and Executor

- Tasks
  - Logical units of work
    - e.g., prime number generation, file caching, file crawling, file indexing, access counting
- Threads
  - Mechanism to run tasks *concurrently*.
- Executor
  - Is the primary abstraction for task execution
    - `Thread` is not anymore.

Executor

- ```
  public interface Executor{
      void execute(Runnable command);
  }
  ```
- `Runnable`'s `run()` implements a task.
- Follows the Producer-Consumer design pattern.
  - Producers: submit tasks
  - Consumers: execute tasks
- Provides the easiest way to implement producers and consumers
- Makes task execution configurable.



Sample Code: PrimeNumberGenExecutorTest.java

- ```
  Runnable r1, r2;
  r1 = new PrimeNumberGenerator(1L, 500000L);
  r2 = new PrimeNumberGenerator(500001L, 1000000L);
  ExecutorService executor = Executors.newFixedThreadPool(2);
  executor.execute(r1);
  executor.execute(r2);
  executor.shutdown();
  executor.awaitTermination(...);
  r1.getPrimes().forEach(...);
  r2.getPrimes().forEach(...);
  ```

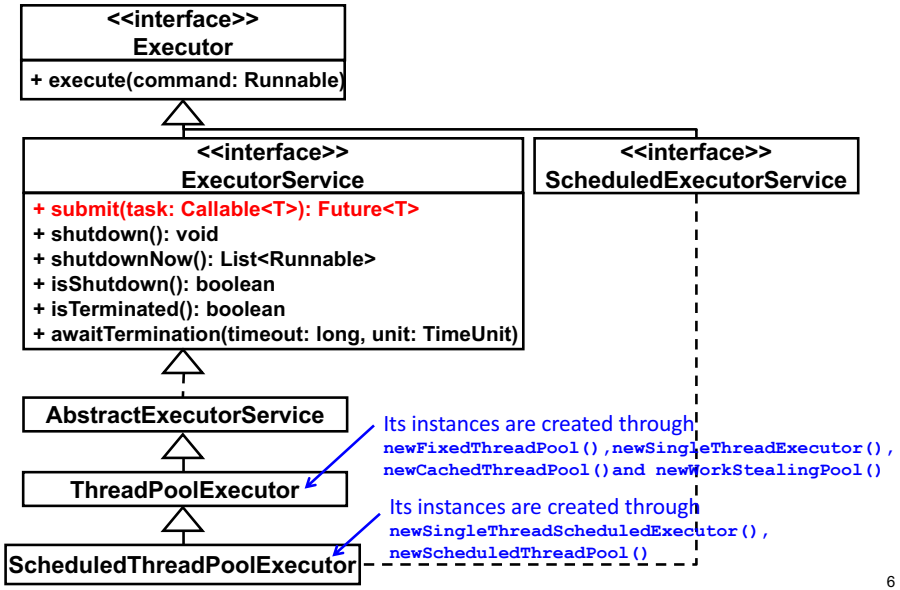

Tasks

- `Runnable` task
  - ```
    public interface Runnable{
        void run();
    }
    ```
  - A `Runnable` class implements a task in `run()`.
    - No parameters. No returned value. No exceptions thrown.
  - Passed to an executor with its `execute()`.
- `Callable` task
  - ```
    public interface Callable<T>{
        T call() throws Exception;
    }
    ```
  - A `Callable` class implements a task in `call()`.
    - No parameters. Can return a value (`T`) and throw an `Exception`.
  - Passed to an executor with its `submit()`.

An Example `Callable` Task

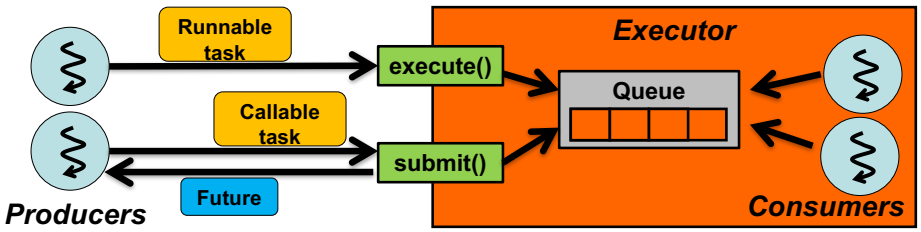
```
class CallablePrimeGenerator implements Callable<List<Long>>{

    private List<Long> primes;

    public CallablePrimeGenerator(long from, long to){
        ...
        this.primes = new ArrayList<List<long>>;  }

    public List<Long> call() throws Exception {
        ...  // generate primes here
        return this.primes;  }  }
```


`ExecutorService`



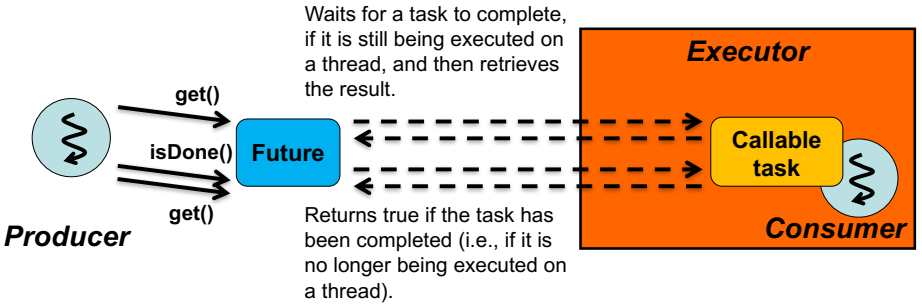

```
CallablePrimeGenerator gen = new CallablePrimeGenerator(...);
ExecutorService executor = Executors.newFixedThreadPool(2);
Future<List<Long>> future = executor.submit(gen);
List<Long> primes = future.get();
```

- `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`.

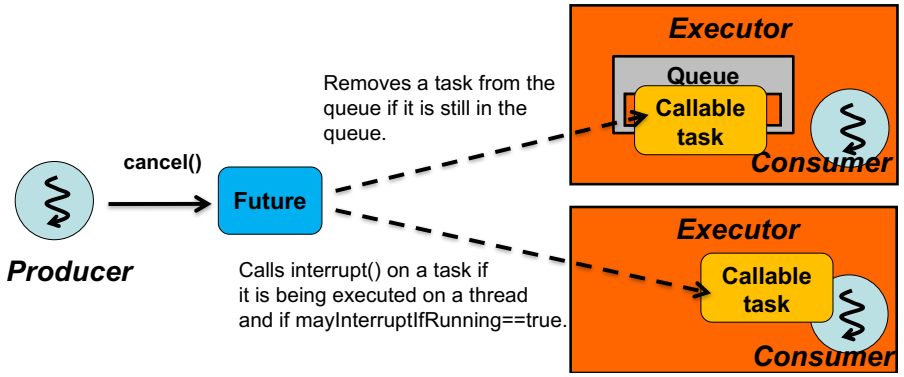


`Future`

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

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



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

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



Runnable and Callable as Functional Interfaces

- `Runnable` (functional interface)
  - `public interface Runnable{ void run(); }`
  - Can implement the body of `run()` as a lambda expression and pass it to an executor's `execute()`.
- `Callable` (functional interface)
  - `public interface Callable<T>{ T call() throws Exception; }`
  - Can implement the body of `call()` as a lambda expression and pass it to an executor's `submit()`.

Future Design Pattern

Futures

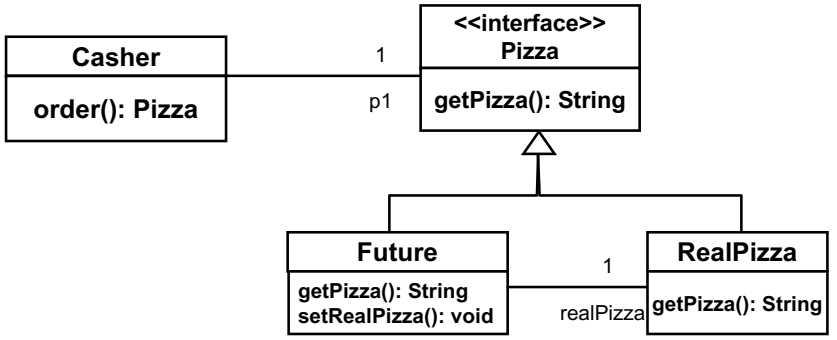
- Futures contract
  - An agreement traded on an organized exchange to buy or sell assets (commodities or shares) at a fixed price but ***to be delivered and paid for later***.

Future Design Pattern

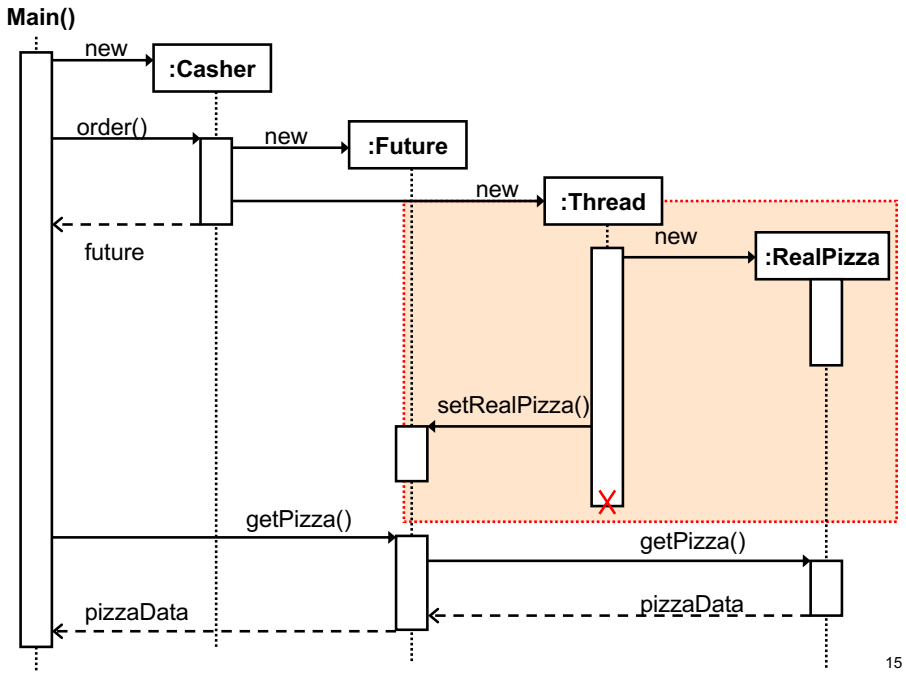
- Intent
  - Perform ***asynchronous task execution*** in a concurrent manner.

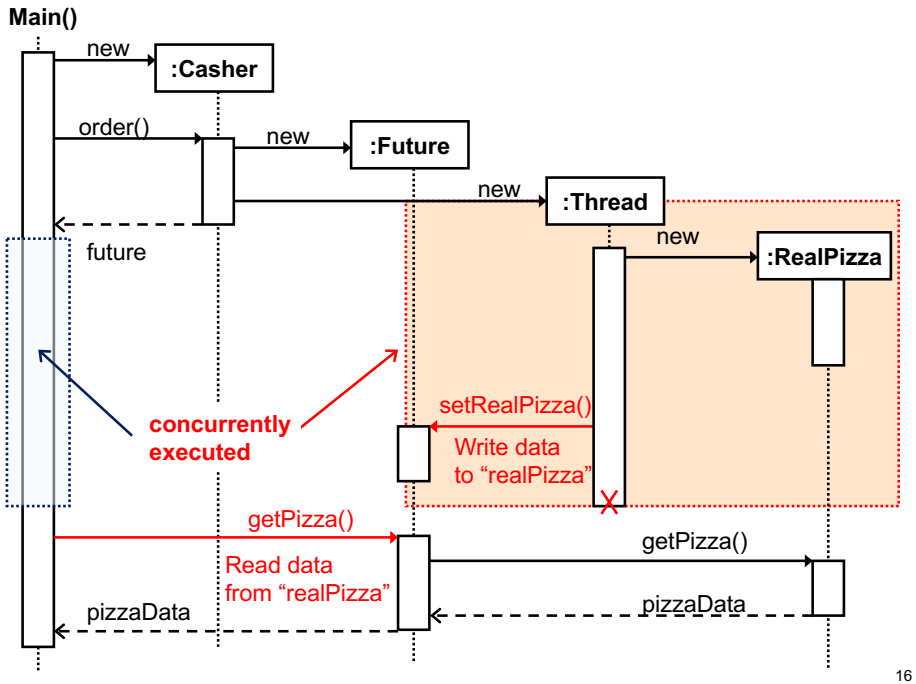
- At a fast food (e.g. pizza) store…
  - You as a customer order a pizza at the casher.
    - The casher asks the kitchen to make a pizza.
    - You pay.
    - The casher gives you a receipt, which shows a purchase confirmation number.
  - You and the kitchen do different things in parallel.
    - The kitchen leaves your pizza at the casher's counter.
  - Sometime later, you check with the casher to see if your pizza is ready to pick up.

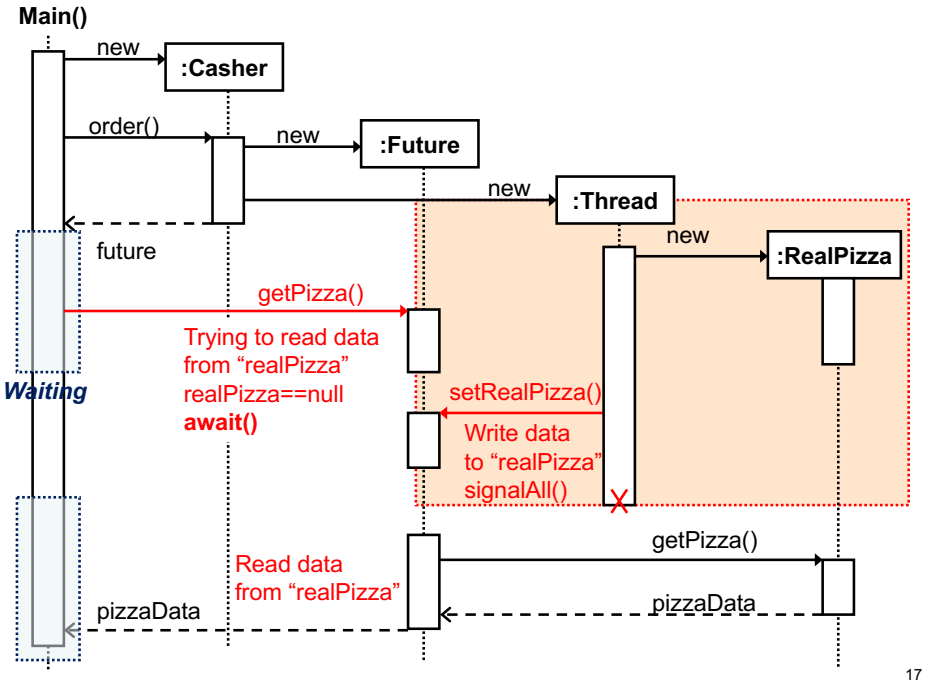
Sample Code using *Future*



Proxy object








```
public class Future implements Pizza{
   private RealPizza realPizza = null;
   private ReentrantLock lock;
   private Condition ready;

   public Future(){
      lock = new ReentrantLock();
      ready = lock.newCondition(); }

   public void setRealPizza( RealPizza real ){
      lock.lock();
      if( realPizza != null ){ return; }
      realPizza = real;
      ready.signalAll();
      lock.unlock(); }

   public String getPizza(){
      String pizzaData = null;
      lock.lock();
      while( realPizza == null ){
         ready.await();
      }
      pizzaData = realPizza.getPizza();
      lock.unlock(); }}
```

(2) Kitchen thread: signalAll().

(1) Customer (main): thread: Goes to "waiting" and temporarily releases the lock if the pizza is not ready.

(3) Customer thread: Goes to "runnable" Acquire the lock again if it is available. If it is not available, goes to "blocked."

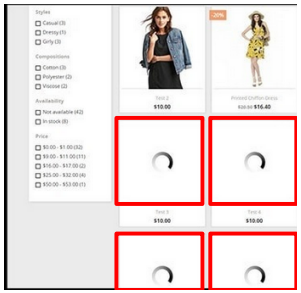
Recap: Proxy Design Pattern

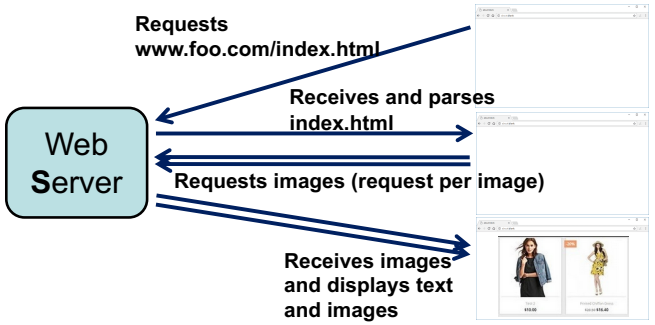
Proxy Design Pattern

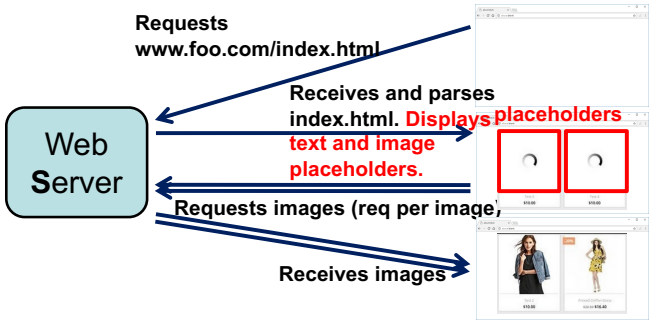
- Intent
  - Provide a surrogate or placeholder for another object to control access to it.

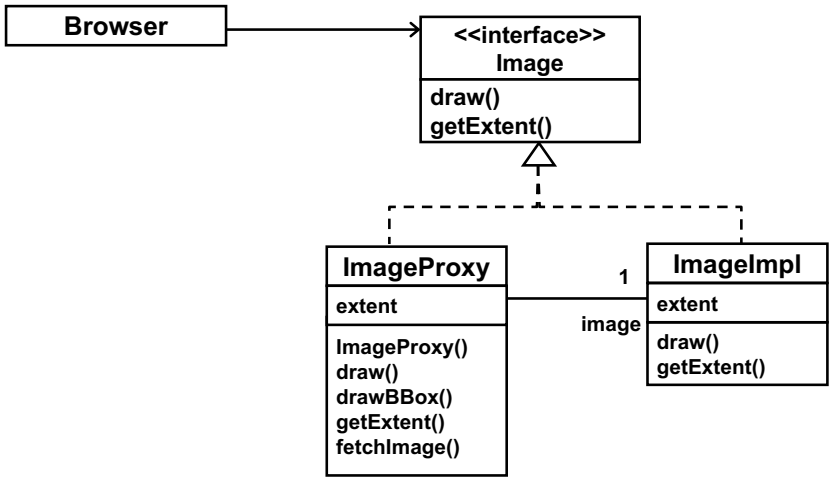
An Example: Lazy Image Loading in a Web Browser

- 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.

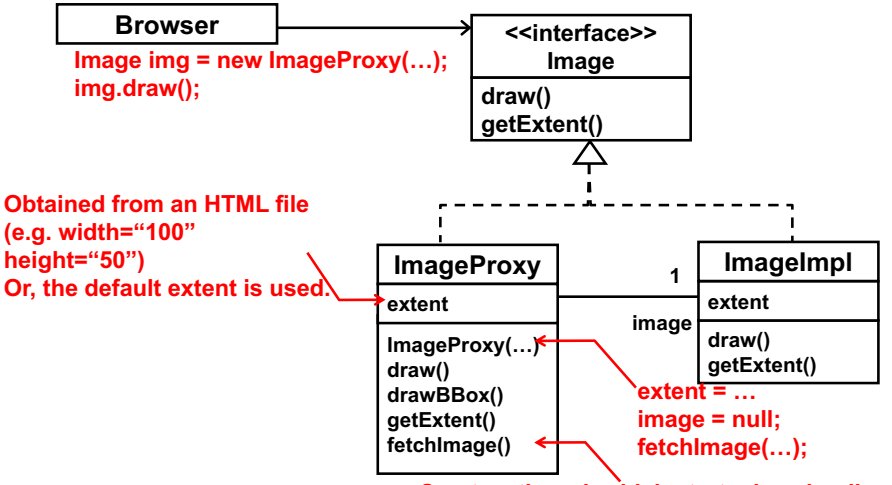


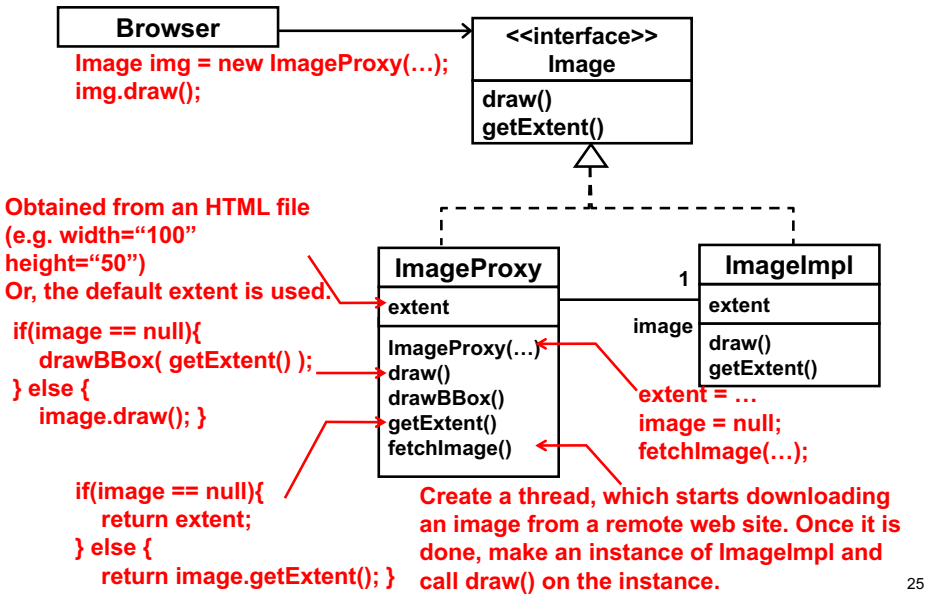






Proxy Class




Browser
Image img = new ImageProxy(…);
img.draw();
<<interface>>
Image
draw()
getExtent()
Obtained from an HTML file
(e.g. width="100"
height="50")
Or, the default extent is used.
ImageProxy
extent
ImageProxy(…)
draw()
drawBBox()
getExtent()
fetchImage()
1
image
ImageImpl
extent
draw()
getExtent()
if(image == null){
drawBBox(getExtent());
} else {
image.draw(); }
extent = …
image = null;
fetchImage(…);
if(image == null){
return extent;
} else {
return image.getExtent(); }
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.