```cpp
int main()
{
    auto tp_start = std::chrono::system_clock::now();

    auto lazy_async = std::async(std::launch::deferred, [] { return std::chrono::system_clock::now(); });
    auto eager_async = std::async(std::launch::async, [] { return std::chrono::system_clock::now(); });

    std::this_thread::sleep_for(std::chrono::seconds(1));

    using dsec = std::chrono::duration<double>;
    auto deferred_sec = static_cast<dsec>(lazy_async.get() - tp_start).count();
    auto eager_sec = static_cast<dsec>(eager_async.get() - tp_start).count();

    std::cout << "duration for deferred in sec : " << deferred_sec << '\n';
    std::cout << "duration for eager in sec    : " << eager_sec << '\n';
}
```

→ async politikasına sahip olan, sleep'e gelmeden zaten çalıştı!

→ get deferred için şimdi hesapladı

→ get sonucu hazır

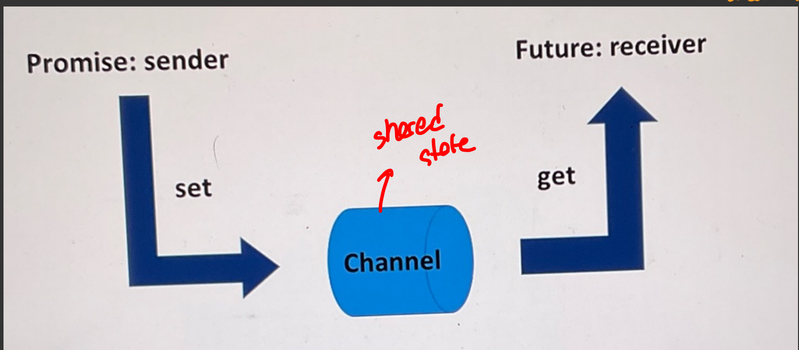
```
Microsoft Visual Studio Debug Console
duration for deferred in sec : 1.0068
duration for eager in sec    : 0.000212

D:\CONCURRENCY\concurrency_1\x64\Release\concurrency_1.exe (process 24740) exited with cod
Press any key to close this window . . .
```

*** Future / Promise:**

- Bize future veren 3 tane yapı var.
  - promise
  - async
  - package task

  bu ikisi de promise nesnesini arkada kullanır.



Tasks as data channels between communication endpoints

→ Promise 1 kez shared state set edebilir.
Future 1 kez " " get edebilir.

→ Promiseler aynı threadler gibi kopyalanamaz ama taşınabilir.

```cpp
int main()
{
    std::promise<int> prom;

}
```

→ gelecekte ya int ya da exception

```cpp
std::promise<int> prom;

std::future<int> ftr = prom.get_future();
```

→ get_future ile, set edilen ya da edilmeyen, future değer çağırılabilir.

```cpp
int main()
{
    std::promise<int> prom;

    auto ftr = prom.get_future();

    prom.set_value(991);

    auto val = ftr.get();

    std::cout << "val = " << val << "\n";

    if (ftr.valid()) {
        std::cout << "future nesnesi gecerli durumda\n";
    }
    else {
        std::cout << "future nesnesi gecersiz durumda\n";
    }

    (void)getchar();
```

future nesnesi get edildiği için artık void değil!

```cpp
int main()
{
    std::promise<int> prom;

    auto ftr = prom.get_future();

    prom.set_value(991);

    auto val = ftr.get();

    std::cout << "val = " << val << "\n";

    try {
        val = ftr.get();
    }
    catch (const std::exception& ex) {
        std::cout << "exception caught: " << ex.what() << '\n';
    }

}
```

2. kez get etmeye çalıştığımız için no state türünden exception gönderir

*Örnek:

```cpp
#include <utility>


void sum_square(std::promise<int>&& prom, int x, int y)
{
    prom.set_value(x * x + y * y);
}


struct Div {
    void operator() (std::promise<int>&& prom, int x, int y) const
    {
        prom.set_value(x / y);
    }
};
```

```cpp
int main()
{
    int x{ 90 }, y{ 15 };

    std::promise<int> sum_square_prom;
    std::promise<int> div_prom;
    //std::future<int> fp_sumsquare = sum_square_prom.get_future();
    auto fp_sumsquare = sum_square_prom.get_future();
    auto fp_div = div_prom.get_future();

    std::thread tss(sum_square, std::move(sum_square_prom), x, y);
    std::thread tdd(Div{}, std::move(div_prom), x, y);
    std::cout << x << " * " << y << " = " << fp_sumsquare.get() << std::endl;
    std::cout << x << " / " << y << " = " << fp_div.get() << std::endl;
    tss.join();
    tdd.join();
}
```

***Hatırlatma:**

```cpp
void func(Myclass &&r)
{
    Myclass m(std::move(r));
}


Myclass m;
func(std::move(m));
```

→ Bu işlem taşıma!

→ eğer move yerine r yazsaydık sadece yine kopyalanacaktı!

→ bu taşıma değil! Cast to r value

```cpp
/*
    bir std::promise nesnesini 2. kez set edersek std::future_error
    sınıfı türünden hata gönderir
*/

#include <future>
#include <iostream>

int main()
{
    std::promise<int> prom;
    prom.set_value(10);

    try {
        prom.set_value(19);
    }
    //catch (const std::exception& ex) {
    catch (const std::logic_error &ex) {
        std::cout << "exception caught: " << ex.what() << '\n';
    }
}
```

→ referans olmasa object slicing yaşanıyor.

***Std:: Shared_future:** – Interface'i future ile aynı!

→ promise aynı promise; fakat farklı threadler, bir çok kez get edebilir.

↔ Shared future nesneleri kopyalanabilir!

```cpp
#include <future>
#include <iostream>
#include <thread>
#include <utility>
#include <syncstream>

struct SumSquare {
    void operator()(std::promise<int>&& prom, int a, int b)const
    {
        prom.set_value(a * a + b * b);
    }
};

void func(std::shared_future<int> sftr)
{
    std::osyncstream os{ std::cout };
    os << "threadId(" << std::this_thread::get_id() << "): ";
    os << "result = " << sftr.get() << std::endl;
```

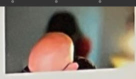
```cpp
int main()
{
    std::promise<int> prom;
    std::shared_future<int> sftr = prom.get_future();
    std::thread th(SumSquare{}, std::move(prom), 5, 9);

    std::thread t1(func, sftr);
    std::thread t2(func, sftr);
    std::thread t3(func, sftr);
    std::thread t4(func, sftr);
    std::thread t5(func, sftr);

    th.join();

    t1.join();
    t2.join();
    t3.join();
    t4.join();
    t5.join();
}
```

└ auto deseydik, shared yerine normal future nesnesi oluşturulurdu!



```cpp
int main()
{
    std::promise<int> prom;
    std::future<int> ftr = prom.get_future();

    std::cout << "ftr is " << (ftr.valid() ? "valid" : "invalid") << '\n';

    std::thread th(SumSquare{}, std::move(prom), 5, 9);
    std::cout << "ftr is " << (ftr.valid() ? "valid" : "invalid") << '\n';
    std::shared_future<int> s_ftr = ftr.share();
    auto s_ftr = ftr.share();
    std::cout << "ftr is " << (ftr.valid() ? "valid" : "invalid") << '\n';
    (void)getchar();
```

→ future nesnesi, share func ile shared future nesnesine çevrildi!

*** Wait For:**
- kendisine verdiğimiz kadar bekler.
- return değeri future_status → enum
  - → deferred
  - → ready (promise is set)
  - → time out (verilen süre geçti ama promise set edilmedi)

```cpp
int main()
{
    std::promise<int> prom;
    auto ft = prom.get_future();
    std::thread th(func, std::move(prom));

    std::future_status status{};
    do {
        status = ft.wait_for(200ms);
        std::cout << "... doing some work here\n" << std::flush;
    } while (status != std::future_status::ready);

    std::cout << "the return value is " << ft.get() << '\n';

    th.join();
```

```cpp
constexpr int x = 50;

long long fib(int n)
{
    return n < 3 ? 1 : fib(n - 1) + fib(n - 2);
}

int main()
{
    using namespace std::literals;

    auto ftr = std::async(fib, x);

    std::cout << "bekle cevap gelecek\n";
    while (ftr.wait_for(10ms) == std::future_status::timeout)
        std::cout << '.' << std::flush;

    auto result = ftr.get();

    std::cout << "fib(" << x << ") is : " << result << '\n';
}
```

→ Bu QT kodlarında ne yazıyabilir!

* Packaged Task:

→ Neco github'ında kurs kodlarında anlatımı var.

std::packaged_task

std::packaged_task sınıfı türünden bir nesne asenkron çağrı yapmak amaçlı bir *callable* sarmalar. *std::packaged_task* nesnesinin *get_future* işlevi ile onunla ilişkilendirilmiş *std::future* nesnesini elde ederiz. Sınıfın fonksiyon çağrı operatör fonksiyonu sarmalanan *callable*'ı çağırır.

std::packaged_task sınıfı çoğunlukla aşağıdaki gibi kullanılır:

- İş yükü olan *callable* bir *std::packaged_task* nesnesi ile sarmalanır:

```cpp
std::packaged_task<int(int, int)> ptask([](int a, int b)
                                        { return a * a + b * b; });
```

- Bir *std::future* nesnesi oluşturulur:

```cpp
std::future<int> ftr = ptask.get_future();
```

- callable çağrılır:

```cpp
ptask(10, 20);
```

- sonuç elde edilir

```cpp
ftr.get();
```