Exceptions

This section touches upon the exceptions that can arise while using execution policies.

WE'LL COVER THE FOLLOWING

- Example

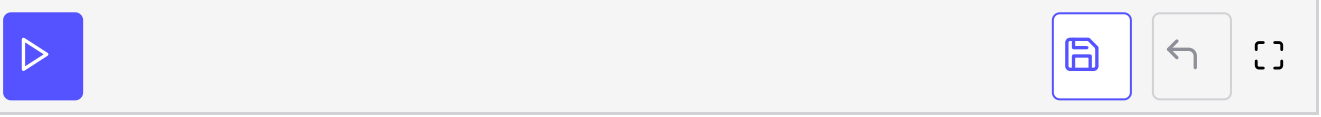
When using execution policies, you need to be prepared for two kinds of situations.

- the scheduler or the implementation fails to allocate resources for the invocation - then `std::bad_alloc` is thrown.
- an exception is thrown from the user code (a functor) - in that case, the exception is not re-thrown, `std::terminate()` is called.

Example

```
#include <algorithm>
#include <execution>
#include <iostream>
#include <vector>

int main() {
    // if you throw exception from the lambda
    // then the std::Terminate will be called
    // exceptions are not re thrown
    try {
        std::vector<int> v{ 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 };

        std::for_each(std::execution::par, v.begin(), v.end(),
            [](int& i) {

                        ++i;

            if (i == 5)
                throw std::runtime_error("something wrong... !");
        });
    }
    catch (const std::bad_alloc& e){
        std::cout << "Error in execution: " << e.what() << '\n';
    }
    catch (const std::exception& e) { // will not happen
```

```
    catch (const std::exception& e) { // will not happen
        std::cout << e.what() << '\n';
    }
    catch (...) {
        std::cout << "error!\n";
    }

    return 0;
}
```



If you run the above code, the `catch` section will only handle `std::bad_alloc`. And if you exit a lambda because of some exception, then the `std::terminate` will be called. The exceptions are not re-thrown.

> When you use parallel algorithms, for better error handling try to make your functors `noexcept`.

The next lesson we will touch upon the new algorithms that support the new parallel execution patterns.