1

Function in global scope

```
void counting1(int count){
    for(int i{0} ; i < count ; i ++){
        qDebug() << "Counting : " << i;
    }
}
```

Run function in a separate thread

```
thread = QThread::create(counting1,1000000000);

connect(thread,&QThread::started,[](){
    qDebug() << "Thread started";
});
connect(thread,&QThread::finished,[=](){
    qDebug() << "Thread finished" <<
                " thread :" << QThread::currentThread() << " id : " <<
                QThread::currentThreadId();
});
/*    ...
thread->start();
```

Manage the memory of the thread object

```
thread = QThread::create(counting1,1000000000);

connect(thread,&QThread::started,[](){
    qDebug() << "Thread started";
});
connect(thread,&QThread::finished,[=](){
    qDebug() << "Thread finished" <<
                " thread :" << QThread::currentThread() << " id : " <<
                QThread::currentThreadId();
});

//This is necessary for memory management
connect(thread,&QThread::finished,thread,&QThread::deleteLater);

thread->start();
```

2

Named lambda

```
auto countlambda = [](int count){
    for(int i{0} ; i < count ; i ++){
        qDebug() << "countlambda counting...";
        qDebug() << "Counting : " << i <<
                    " thread :" << QThread::currentThread() << " id : " <<
                    QThread::currentThreadId();
    }
};

thread = QThread::create(countlambda,100000);

connect(thread,&QThread::started,[](){
    qDebug() << "Thread started";
});
connect(thread,&QThread::finished,[=](){
    qDebug() << "Thread finished" <<
                " thread :" << QThread::currentThread() << " id : " <<
                QThread::currentThreadId();
});

//This is necessary for memory management
connect(thread,&QThread::finished,thread,&QThread::deleteLater);

thread->start();
```

3

Non named lambda function

```
thread = QThread::create([](){
    for(int i{0} ; i < 100000 ; i ++){
        qDebug() << "Counting : " << i <<
                    " thread :" << QThread::currentThread() << " id : " <<
                    QThread::currentThreadId();
    }
});

 //Connections
connect(thread,&QThread::started,[](){
    qDebug() << "Thread started";
});
connect(thread,&QThread::finished,[=](){
    qDebug() << "Thread finished" <<
                " thread :" << QThread::currentThread() << " id : " <<
                QThread::currentThreadId();
});

//This is necessary for memory management
connect(thread,&QThread::finished,thread,&QThread::deleteLater);

thread->start();
```

4

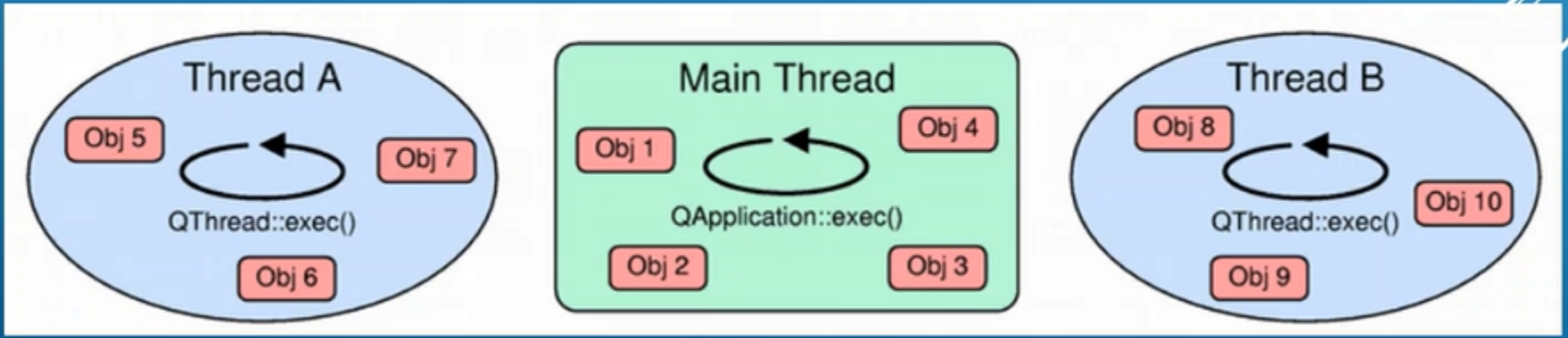
Call member method in lambda function

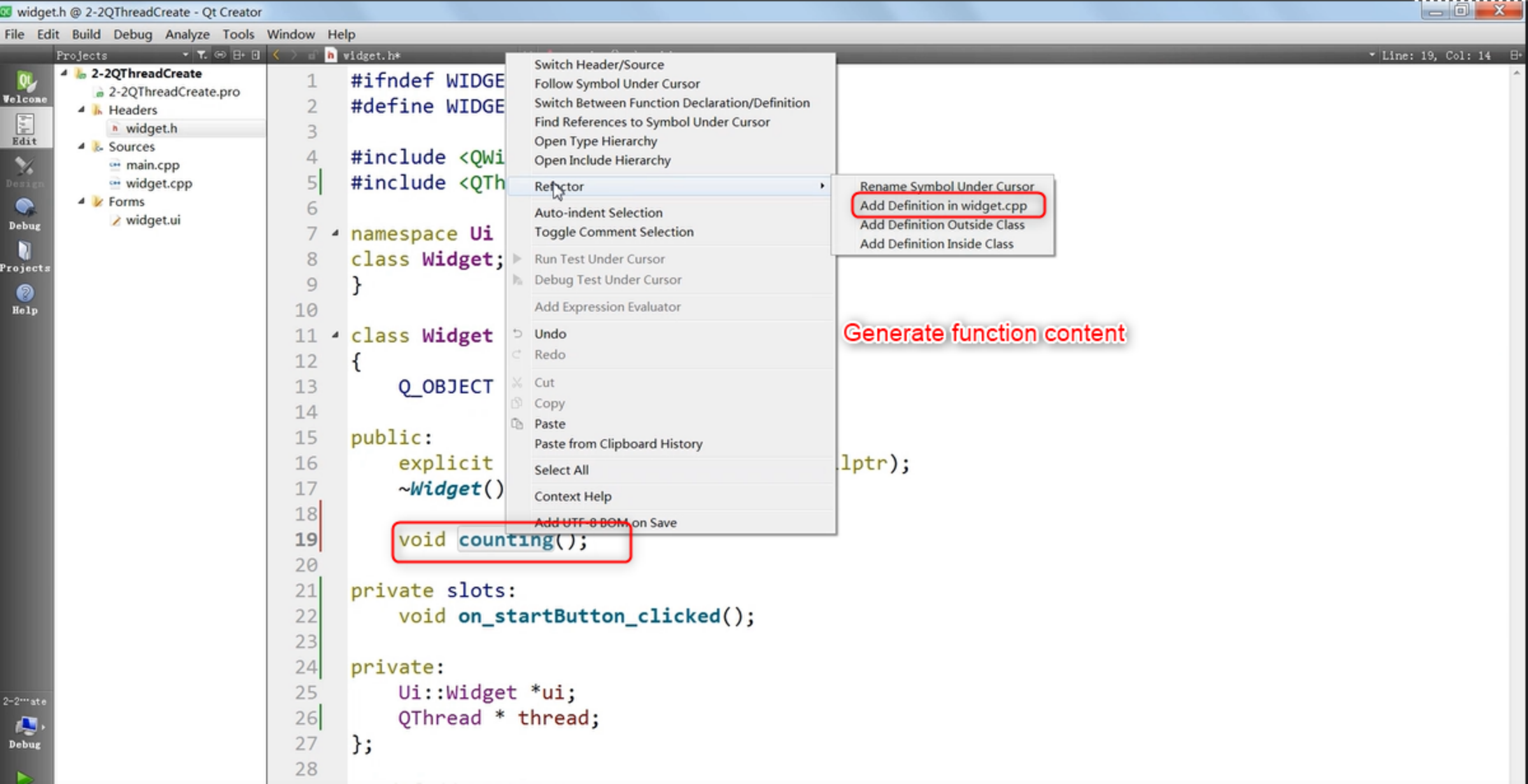
```
thread = QThread::create([=](){
    counting();
});


 //Connections
connect(thread,&QThread::started,[](){
    qDebug() << "Thread started";
});
connect(thread,&QThread::finished,[=](){
    qDebug() << "Thread finished" <<
                " thread :" << QThread::currentThread() << " id : " <<
                QThread::currentThreadId();
});

//This is necessary for memory management
connect(thread,&QThread::finished,thread,&QThread::deleteLater);

thread->start();
```

QThread :: create()

No event loop

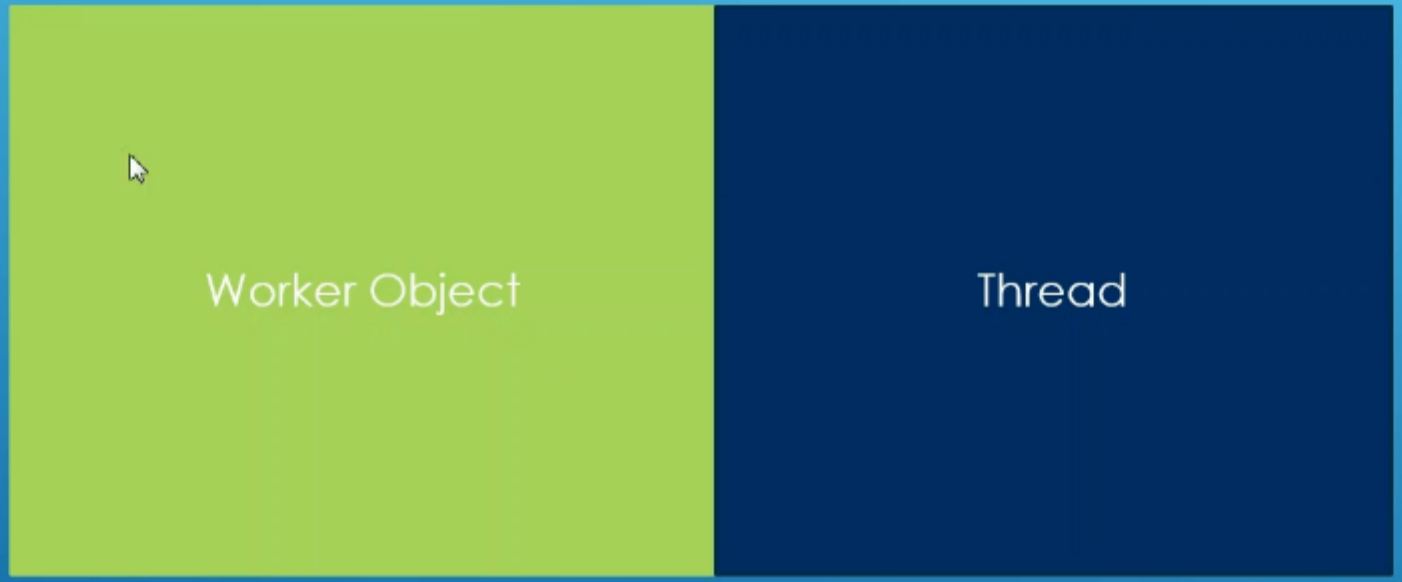
- Finished signal is emitted when method returns
- Any calls to isRunning() after method returns will return false
- Will talk more about event loop later. Just be aware of this




widget.h @ 2-2QThreadCreate - Qt Creator
File Edit Build Debug Analyze Tools Window Help
Projects
2-2QThreadCreate
2-2QThreadCreate.pro
Headers
widget.h
Sources
main.cpp
widget.cpp
Forms
widget.ui
Welcome
Edit
Design
Debug
Projects
Help
widget.h*
Line: 19, Col: 14
#ifndef WIDGE
#define WIDGE
#include <QWi
#include <QTh
namespace Ui
class Widget;
}
class Widget
{
Q_OBJECT
public:
explicit
lptr);
~Widget()
void counting();
private slots:
void on_startButton_clicked();
private:
Ui::Widget *ui;
QThread * thread;
};
Switch Header/Source
Follow Symbol Under Cursor
Switch Between Function Declaration/Definition
Find References to Symbol Under Cursor
Open Type Hierarchy
Open Include Hierarchy
Refactor
Auto-indent Selection
Toggle Comment Selection
Run Test Under Cursor
Debug Test Under Cursor
Add Expression Evaluator
Undo
Redo
Cut
Copy
Paste
Paste from Clipboard History
Select All
Context Help
Add UTF-8 BOM on Save
Rename Symbol Under Cursor
Add Definition in widget.cpp
Add Definition Outside Class
Add Definition Inside Class
Generate function content

QThread :: create()

QObject::moveToThread()


Worker Object
Thread

Worker Object

```
class Worker : public QObject
{
    Q_OBJECT
public:
    explicit Worker(QObject *parent = nullptr);
    ~Worker();

signals:
    void countFinished();
    void currentCount(int count);

public slots:
    void doCounting();

};
```

Worker Object

```
void Worker::doCounting()
{
    qDebug() << "Worker counting method running in thread : " << thread();
    for(int i{0} ; i < 1000000001 ; i++){
        /*
         * Only emit signal to send info to ui at 100000 intervals. UI can handle this.
         * Otherwise it is going to freeze.
         * */
        if((i%100000) == 0){
            double percentage = ((i/1000000000.0)) * 100;
            emit currentCount(QVariant::fromValue(percentage).toInt());
        }
    }
    emit countFinished();
}
```

Worker Object

```
class Worker : public QObject
{
    Q_OBJECT
public:
    explicit Worker(QObject *parent = nullptr);
    ~Worker();

signals:
    void countFinished();
    void currentCount(int count);

public slots:
    void doCounting();

};
```

Create thread object

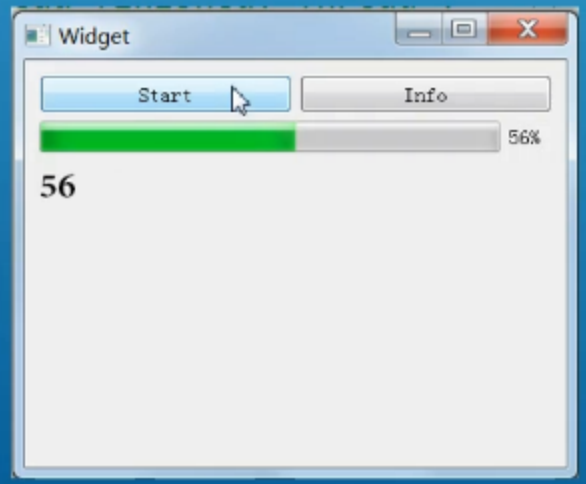
```
workerThread = new QThread(this);
connect(workerThread,&QThread::finished,this,&Widget::threadFinished);
```

Move worker object to other thread

```
void Widget::on_startButton_clicked()
{
    Worker * worker = new Worker;
    worker->moveToThread(workerThread);

    connect(worker,&Worker::countFinished,this,&Widget::countDone);

    connect(workerThread, &QThread::finished, worker, &Worker::deleteLater);

    connect(workerThread,&QThread::started,worker,&Worker::doCounting);

    connect(worker,&Worker::currentCount,this,&Widget::currentCount);

    workerThread->start();
}
```

Use feedback from thread in UI

```
void Widget::currentCount(int value)
{
    if(value == 1000){
        qDebug() << "CurrentCount. Thread : " << thread();
        qDebug() << "CurrentCount.Current Thread : " << QThread::currentThread();
    }

    ui->progressBar->setValue(value);
    ui->label->setText(QString::number(value));

}
```



QObject::moveToThread()

Gives us an event loop by default

- Count finished but thread is still running
- This is because there is an event loop running in the thread
- To force out of the event loop, call the exit method on the thread

Pausing and Resuming ?

Subclass QThread

```
class WorkerThread : public QThread
{
    Q_OBJECT
public:
    explicit WorkerThread(QObject *parent = nullptr);
    ~WorkerThread() override;

signals:
    void currentCount(int value);

public slots:

    // QThread interface
protected:
    void run() override;
};
```

```
void WorkerThread::run()
{
    for(int i{0} ; i < 1000000001 ; i++){
        /*
         * Only emit signal to send info to ui at 100000 intervals. UI can handle this.
         * Otherwise it is going to freeze.
         * */
        if((i%100000) == 0){
            double percentage = ((i/1000000000.0)) * 100;
            emit currentCount(QVariant::fromValue(percentage).toInt());
        }
    }
}
```

Initialize thread

```
workerThread = new WorkerThread(this);
connect(workerThread, &WorkerThread::currentCount, this, &Widget::currentCount);
```

```
void Widget::on_startButton_clicked()
{
    workerThread->start();
}
```

QThread with async code (QTimer)

QThread::create()

Asynchronous code in a secondary thread requires an event loop to work properly.Think Qtimer, QTcpSocket,..

Qthread::create() with a timer

```
thread = QThread::create([=](){
    QTimer * timer = new QTimer();
    connect(timer,&QTimer::timeout,[](){
        qDebug() << "Time out . Running in thread : " << QThread::currentThread();
    });
    timer->setInterval(1000);
    timer->start();
});
```

Allow apps to communicate through Windows Defender Firewall

To add, change, or remove allowed apps and ports, click Change settings.

What are the risks of allowing an app to communicate?

Change settings

Allowed apps and features:

Name	Private	Public
@FirewallAPI.dll,-80201	☑	☑
@FirewallAPI.dll,-80206	☑	☑
{78E1CD88-49E3-476E-B926-580E596AD309}	☑	☑

Wi-Fi)

WPS: Bật vô hiệu hóa

PIN hiện tại: 00856621 Khôi phục PIN Tạo PIN mới

vô hiệu hóa PIN thiết bị

Thêm vào thiết bị mới: Thêm thiết bị

QThread with async code (QTimer)

Qobject::moveToThread

Asynchronous code in a secondary thread requires an event loop to work properly.Think Qtimer, QTcpSocket,..

With moveToThread, the worker method is running in an event loop already

Qobject::moveToThread with asynchronous code(Timer)

```
class Worker : public QObject
{
    Q_OBJECT
public:
    explicit Worker(QObject *parent = nullptr);

signals:

public slots:
    void doWork();
};
```

Qobject::moveToThread with asynchronous code(Timer)

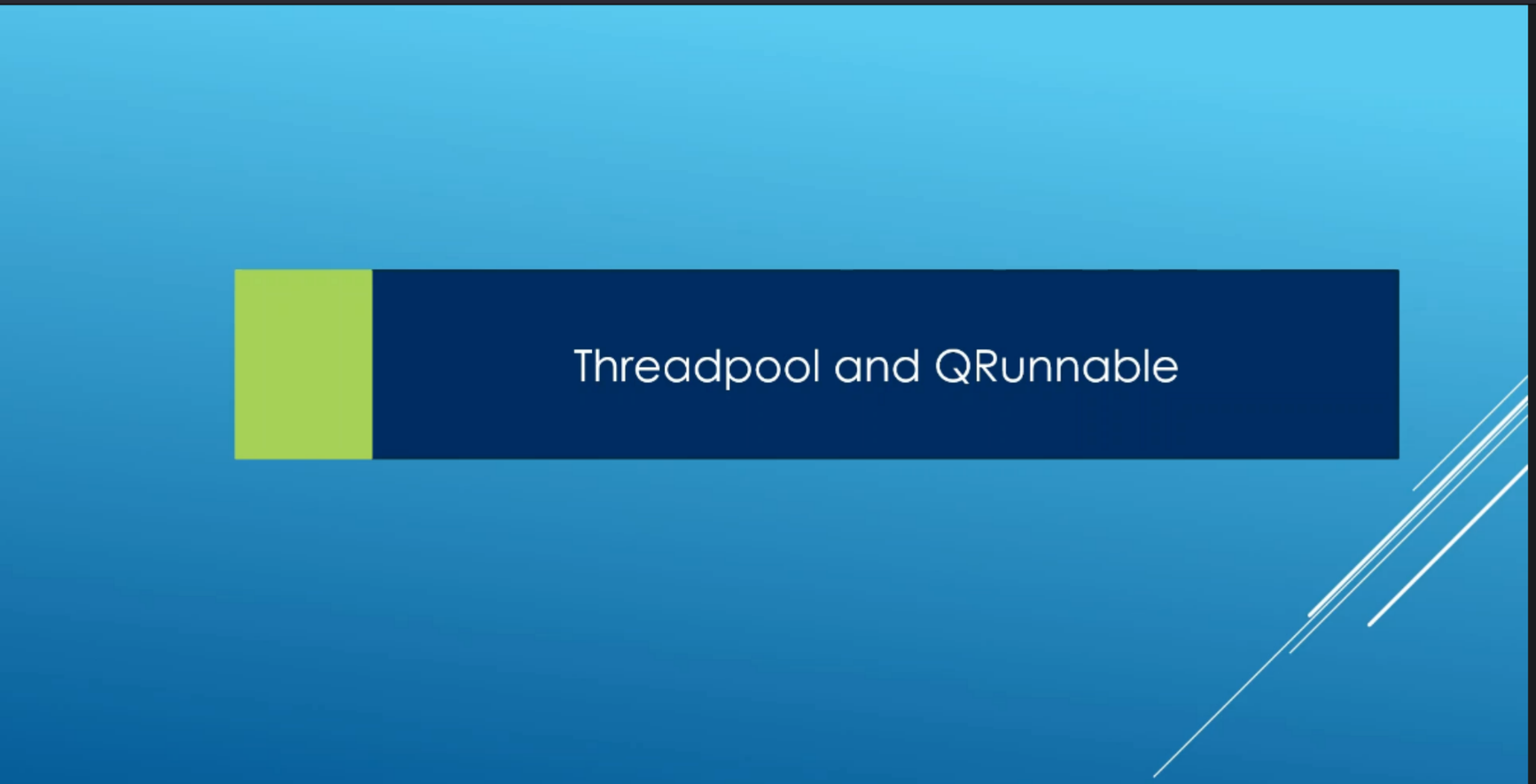
```
void Worker::doWork()
{
    QTimer * timer = new QTimer();
    connect(timer,&QTimer::timeout,[](){
        qDebug() << "Time out . Running in thread : " << QThread::currentThread();
    });
    timer->setInterval(1000);
    timer->start();
}
```

```
thread = new QThread(this);
```

```
Worker * worker = new Worker;

worker->moveToThread(thread);

connect(thread,&QThread::started,worker,&Worker::doWork);
connect(thread,&QThread::finished,[=](){
    qDebug() << "Thread finished";
});

thread->start();
```

QThread with async code (QTimer)

Subclass QThread



Course content

10min Resources

10. ThreadPool and QRunnable
11min Resources

11. ThreadPool and QRunnable - Sending feedback to ui
24min Resources

12. ThreadPool and QRunnable - Async Code
5min Resources

13. Custom Type Signal Parameters
13min Resources

14. Threading Methods Comparison
5min

Section 3: Thread Synchronization
0 / 8 | 2hr 9min

Section 4: Thread Safety and Reentrancy
0 / 6 | 1hr 10min

- A number of threads is reserved in your Qt application, to run long running operations for you
- All you need to do is wrap your long operation in a QRunnable subclass
- And throw your runnable task in a pool of threads to run it

- A number of threads is reserved in your Qt application, to run long running operations for you
- All you need to do is wrap your long operation in a QRunnable subclass
- And throw your runnable task in a pool of threads to run it

Subclass QRunnable

```
class Worker : public QRunnable
{
public:

    explicit Worker(QObject * receiver);
    ~Worker() override;

    // QRunnable interface
    void run() override;

private:
    QObject * m_receiver;
};
```

Worker implementation

```
void Worker::run()
{
    qDebug() << "Worker code running in thread : " << QThread::currentThread();
    for(int i{0} ; i < 1000000001 ; i++){

        if((i%100000) == 0){
            double percentage = ((i/1000000000.0)) * 100;
            qDebug() << "Current percentage : " << percentage;

        }
    }
}
```

Throw runnable in a thread pool

```
void Widget::on_startButton_clicked()
{
    Worker * worker = new Worker(this);
    worker->setAutoDelete(false);

    QThreadPool::globalInstance()->start(worker);
}
```

Threadpool and QRunnable

Sending feedback to the UI


Custom Events
QMetaObject::InvokeMethod()

Custom Event

```
class ProgressEvent : public QEvent
{
public:
    enum {EventId = QEvent::User +1};
    ProgressEvent(int progress = 0 );
    int progress() const;
private:
    int m_progress;
};
```

Send feedback as a custom event

```
void Worker::run()
{
    qDebug() << "Worker code running in thread : " << QThread::currentThread();
    for(int i{0} ; i < 1000000001 ; i++){
        /*  ...
        if((i%100000) == 0){
            double percentage = ((i/1000000000.0)) * 100;
            qDebug() << "Current percentage : " << percentage;
            //2.Posting custom event
            QApplication::postEvent(m_receiver,
                                    new ProgressEvent(QVariant::fromValue(percentage).toInt()));
        }
    }
}
```

Handle event in receiver object

```
bool Widget::event(QEvent *event)
{
    if (event->type() ==
            static_cast<QEvent::Type>(ProgressEvent::EventId))
    {
        ProgressEvent *progressEvent =
                static_cast<ProgressEvent*>(event);
        //qDebug() << "This is Widget. Progress is : " << progressEvent->progress();
        ui->progressBar->setValue(progressEvent->progress());
        return true;
    }
    return QWidget::event(event);
}
```

Custom Events

- QMetaObject::InvokeMethod()

Invoke public slot in receiver object

```
QMetaObject::invokeMethod(m_receiver, "gotUpdate",
                          Qt::QueuedConnection,
                          Q_ARG(int, QVariant::fromValue(percentage).toInt()));
```

Threadpool and QRunnable

Asynchronous code

Async code running in a thread pool

```
void Worker::run()
{
    QTimer * timer = new QTimer();
    QObject::connect(timer,&QTimer::timeout,[](){
        qDebug() << "Time out . Running in thread : " << QThread::currentThread();
    });
    timer->setInterval(1000);
    timer->start();
}
```

Custom Type Signal Parameters

Q_DECLARE_METATYPE()

```
class NumberdString
{
public:
    NumberdString();
    NumberdString(int number, QString string);
    int number() const;
    void setNumber(int number);

    QString string() const;
    void setString(const QString &string);

private :
    int m_number;
    QString m_string;
};
Q_DECLARE_METATYPE(NumberdString);
```

Register class in main()

```
int main(int argc, char *argv[])
{
    QApplication a(argc, argv);

    qRegisterMetaType<NumberdString>("NumberedString");

    Widget w;
    w.show();

    return a.exec();
}
```

Emit custom type signal

```
void Worker::doCounting()
{
    qDebug() << "Worker counting method running in thread : " << thread();
    for(int i{0} ; i < 1000000001 ; i++){
        /*
         * Only emit signal to send info to ui at 100000 intervals. UI can handle this.
         * Otherwise it is going to freeze.
         * */
        if((i%100000) == 0){
            double percentage = ((i/1000000000.0)) * 100;
            //emit currentCount(QVariant::fromValue(percentage).toInt());
            QString txt = "Hello : " + QVariant::fromValue(percentage).toString();
            emit currentNumberString(NumberdString(QVariant::fromValue(percentage).toInt(),
                                                   txt));
        }
    }
    emit countFinished();
}
```

Register class in main()

```
int main(int argc, char *argv[])
{
    QApplication a(argc, argv);

    qRegisterMetaType<NumberdString>("NumberedString");

    Widget w;
    w.show();

    return a.exec();
}
```

Low level threading : A Comparison

QThread::create

QThreadPool and QRunnable

MoveToThread

Subclass QThread

QThread::create

- No event loop
- Can't send feedback using signals and slots
- Can send feedback using custome events and QMetaObject::invoke method

MoveToThread

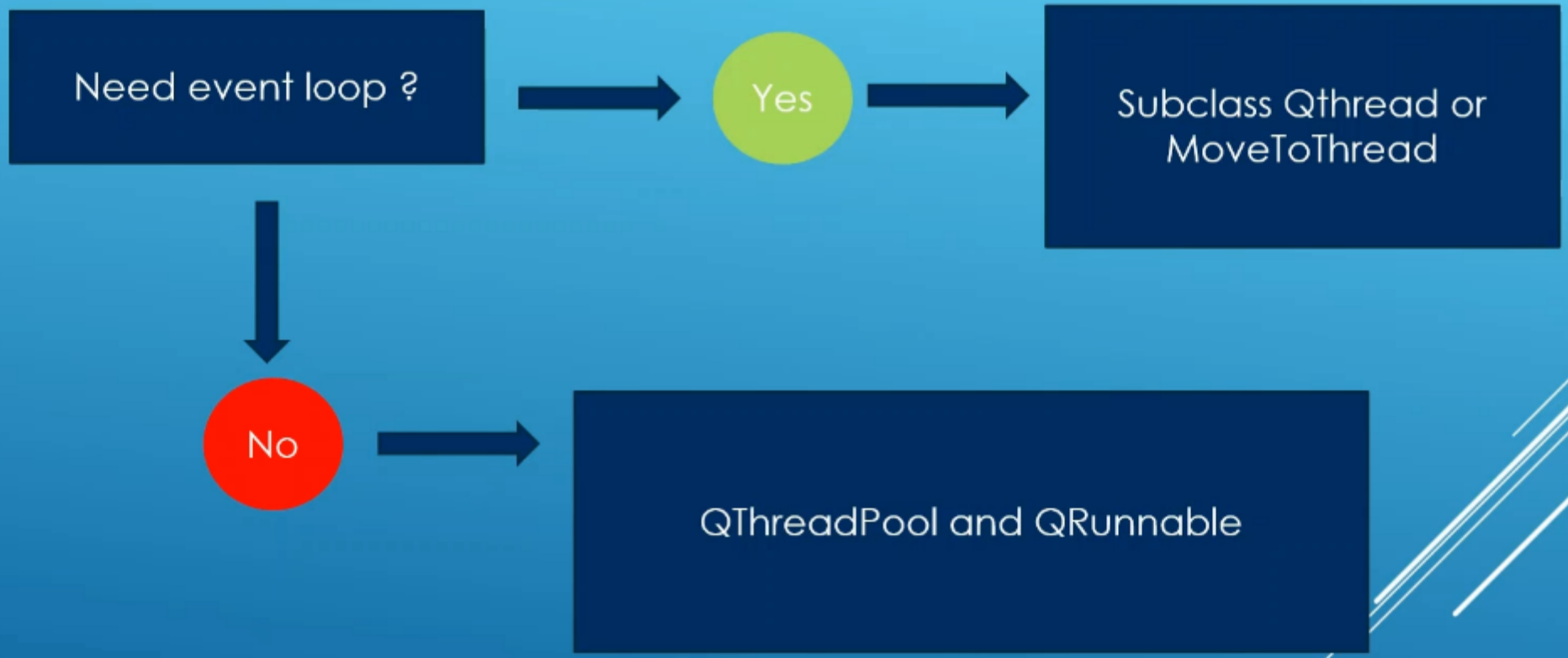
Subclass QThread

QThreadPool and QRunnable

Easy API to just run an operation in background thread

You don't have to maintain the threads

Can send feedback to UI with custom events and QMetaObject::invokeMethod


Need event loop ?
Yes
Subclass Qthread or MoveToThread
No
QThreadPool and QRunnable

Course content

13min Resources

14. Threading Methods Comparison
5min

Section 3: Thread Synchronization
0 / 8 | 2hr 9min

15. Thread Synchronization Overview
4min

16. Thread Synchronization - Mutex
21min Resources

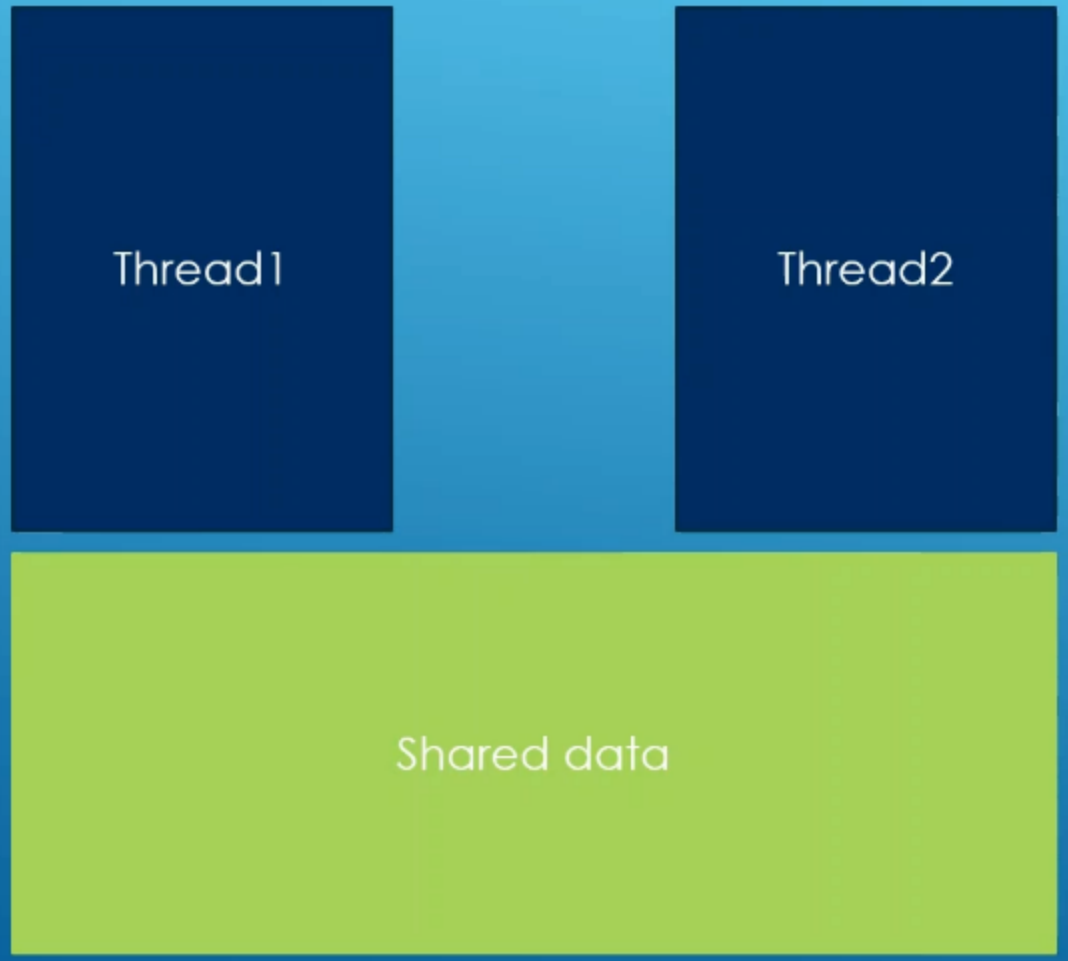
17. Thread Synchronization - Mutex -Shared variable
15min Resources

18. Thread Synchronization - ReadWrite Lock
17min Resources

19. Thread Synchronization - Semaphores
30min Resources

20. Thread Synchronization - WaitConditions
21min Resources

21. Wait Conditions - Pause Resume


Thread1
Thread2
Shared data

```
int number = 6;

void method1()
{
    number *= 5;
    number /= 4;
}

void method2()
{
    number *= 3;
    number /= 2;
}
```

Call in succession

```
// method1()
number *= 5;          // number is now 30
number /= 4;          // number is now 7

// method2()
number *= 3;          // number is now 21
number /= 2;          // number is now 10
```

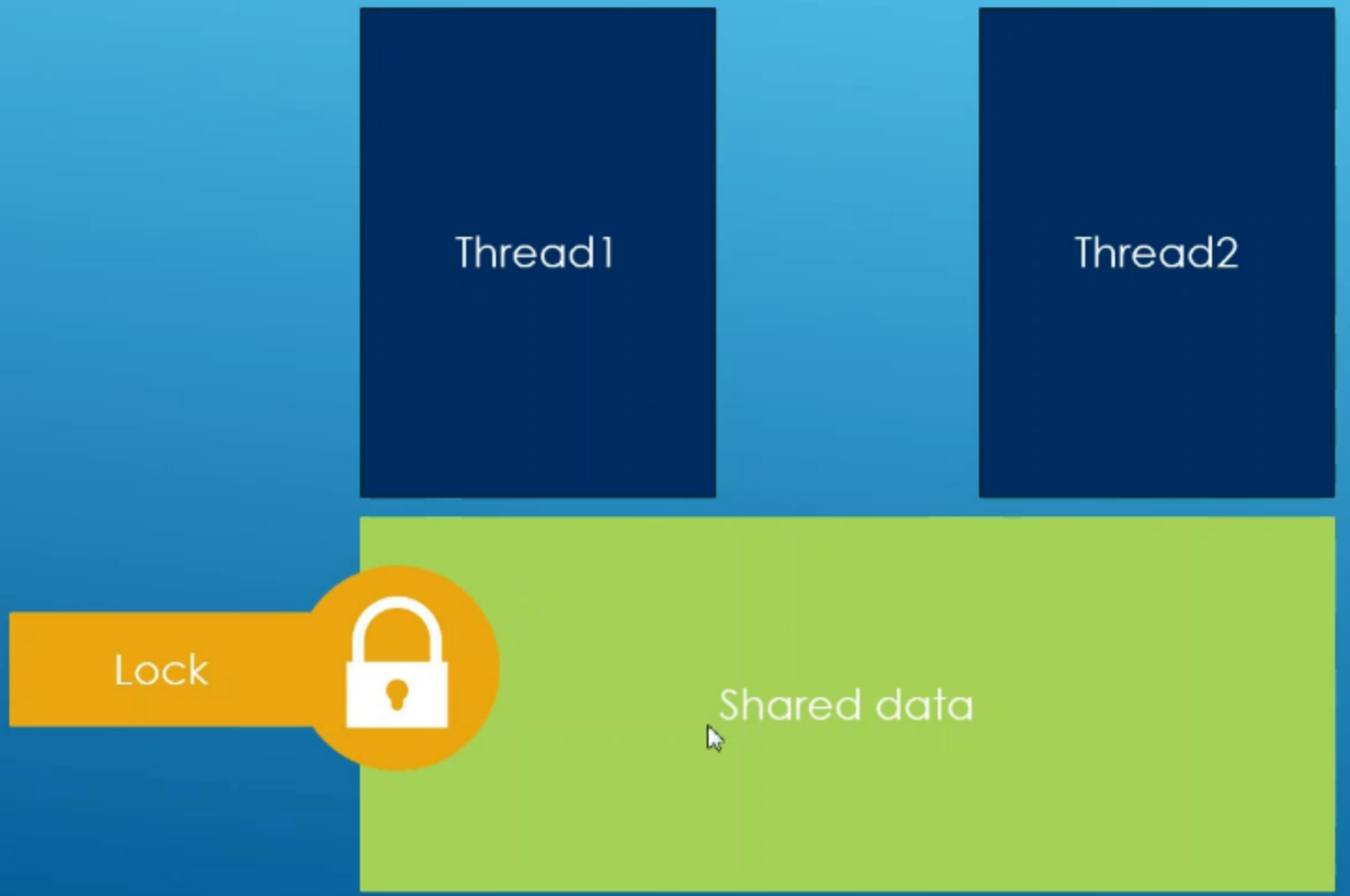
Example from Qt Docs

```
int number = 6;

void method1()
{
    number *= 5;
    number /= 4;
}

void method2()
{
    number *= 3;
    number /= 2;
}
```

Call in succession

```
// Thread 1 calls method1()
number *= 5;          // number is now 30

// Thread 2 calls method2().
//
// Most likely Thread 1 has been put to sleep by the operating
// system to allow Thread 2 to run.
number *= 3;          // number is now 90
number /= 2;          // number is now 45

// Thread 1 finishes executing.
number /= 4;          // number is now 11, instead of 10
```

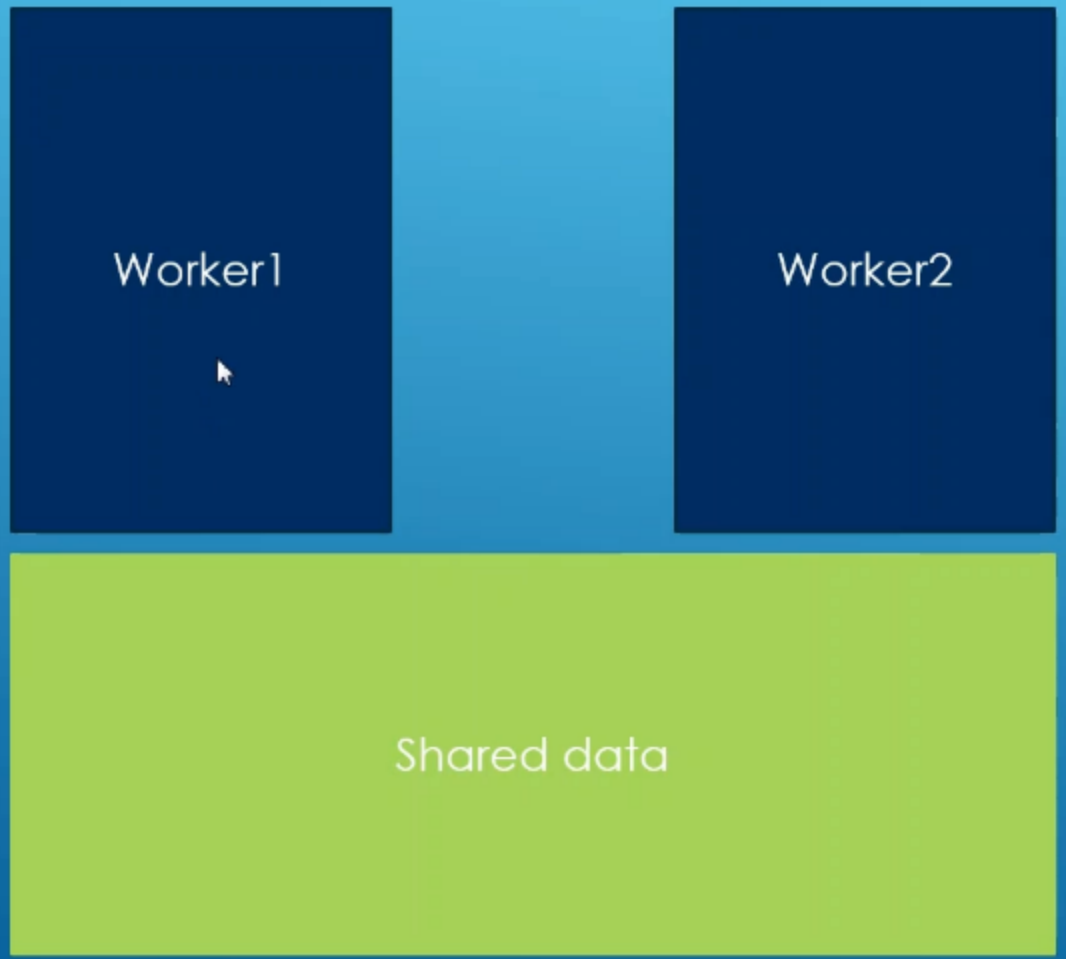
Example from Qt Docs

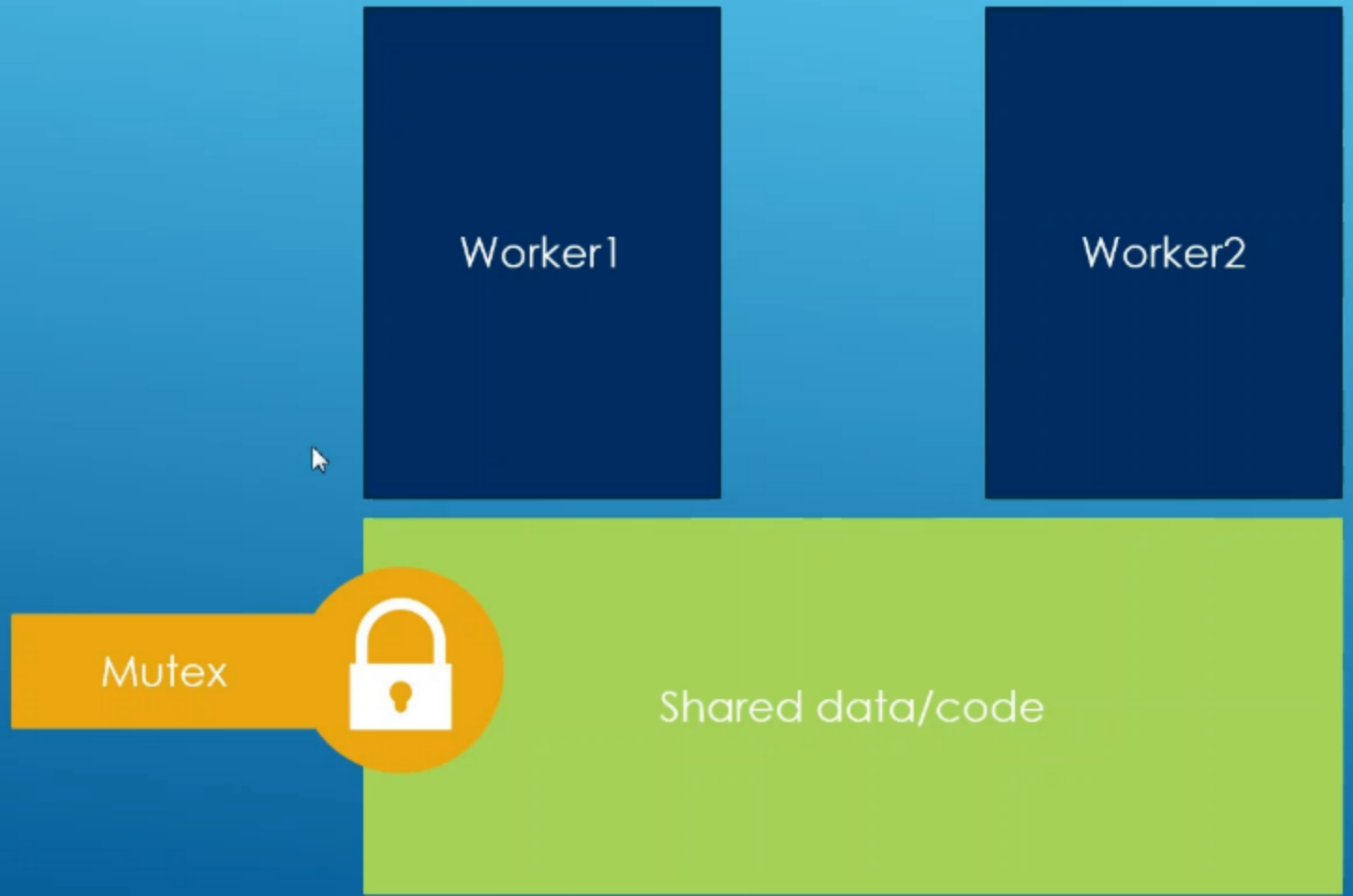

Thread1
Thread2
Lock
Shared data

- Qmutex and QMutexLocker
- ReadWrite Locks
- Semaphores
- Wait Conditions
- Pause / Resume in Secondary Threads

Thread Synchronization

Mutexes


Worker1
Worker2
Shared data


Worker1
Worker2
Mutex
Shared data/code

Mutexes are used to lock up a piece of code for atomic access

The mutex has to be shared across objects.

Mutex example

```
class PrintWorker : public QThread
{
    Q_OBJECT
public:
    explicit PrintWorker(const QString & name,bool * stop,
                         QMutex * mutex,QObject *parent = nullptr);

signals:

public slots:
private:
    QString m_name;
    bool * m_stop ;
    QMutex * m_mutex;

    // QThread interface
protected:
    void run() override;
};
```

```
int main(int argc, char *argv[])
{
    QApplication a(argc, argv);

    bool stopFlag = false;

    QMutex mutex;

    PrintWorker black("Black",&stopFlag,&mutex),
                white("White",&stopFlag,&mutex);

    black.start();
    white.start();

    QMessageBox::information(nullptr,"QMutex",
                             "Thread working. Close Me to stop");

    stopFlag = true;

    black.wait();
    white.wait();

    return 0;
}
```

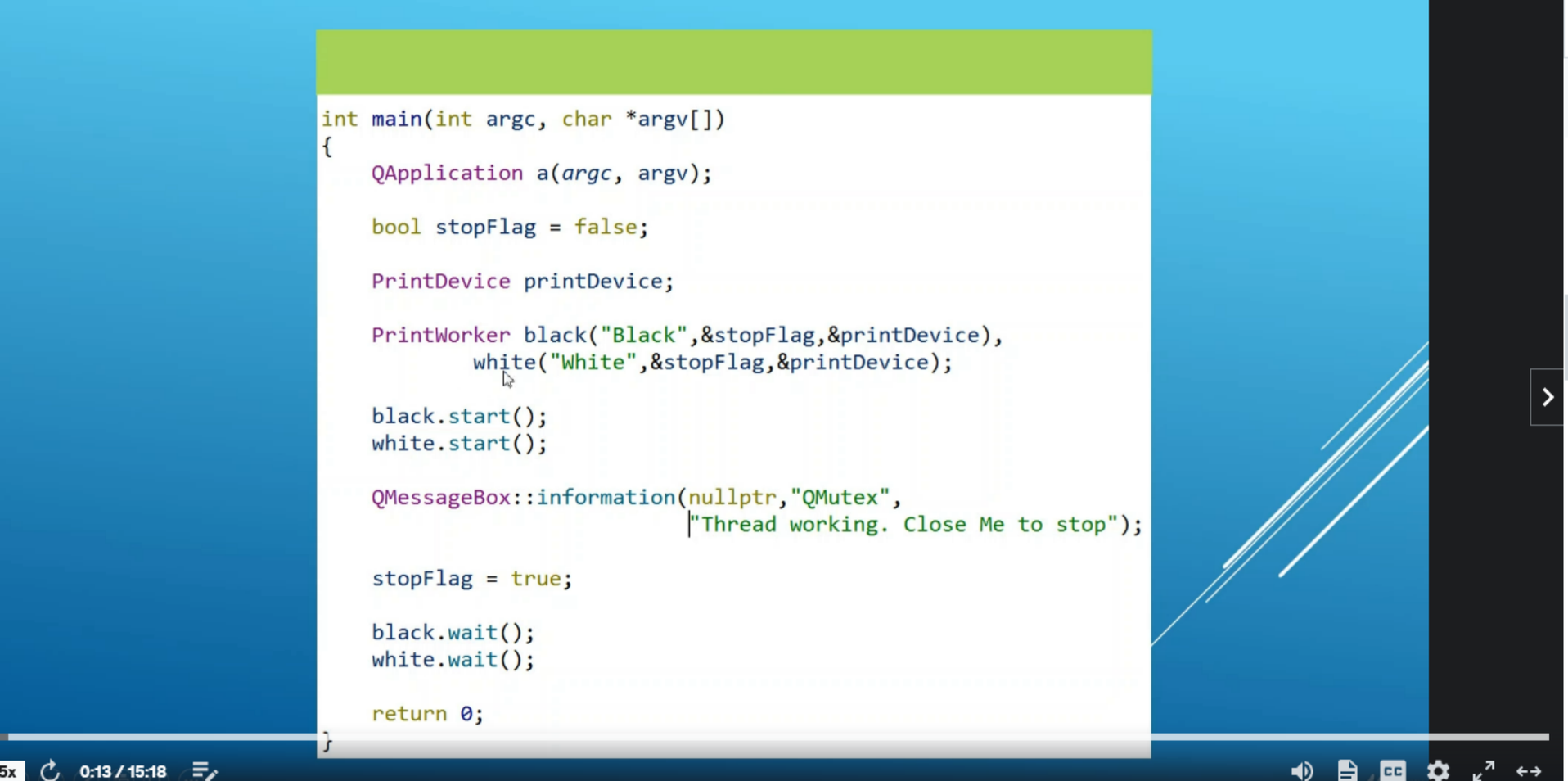
```
int main(int argc, char *argv[])
{
    QApplication a(argc, argv);

    bool stopFlag = false;

    QMutex mutex;

    PrintWorker black("Black",&stopFlag,&mutex),
                white("White",&stopFlag,&mutex);

    black.start();
    white.start();

    QMessageBox::information(nullptr,"QMutex",
                             "Thread working. Close Me to stop");

    stopFlag = true;

    black.wait();
    white.wait();

    return 0;
}
```

Mutex example

```
PrintWorker::PrintWorker(const QString &name,bool* stop,QMutex * mutex, QObject *parent) :
    QThread (parent) , m_name(name),m_stop(stop),
    m_mutex(mutex)
{
}

void PrintWorker::run()
{
    while(!(*m_stop)){
        m_mutex->lock();

        if(*m_stop){

        m_mutex->unlock();
        return;
    }
    qDebug() << m_name ;
    sleep(1);
    m_mutex->unlock();
}
```

Thread Synchronization

Mutex – Shared variable



Course content

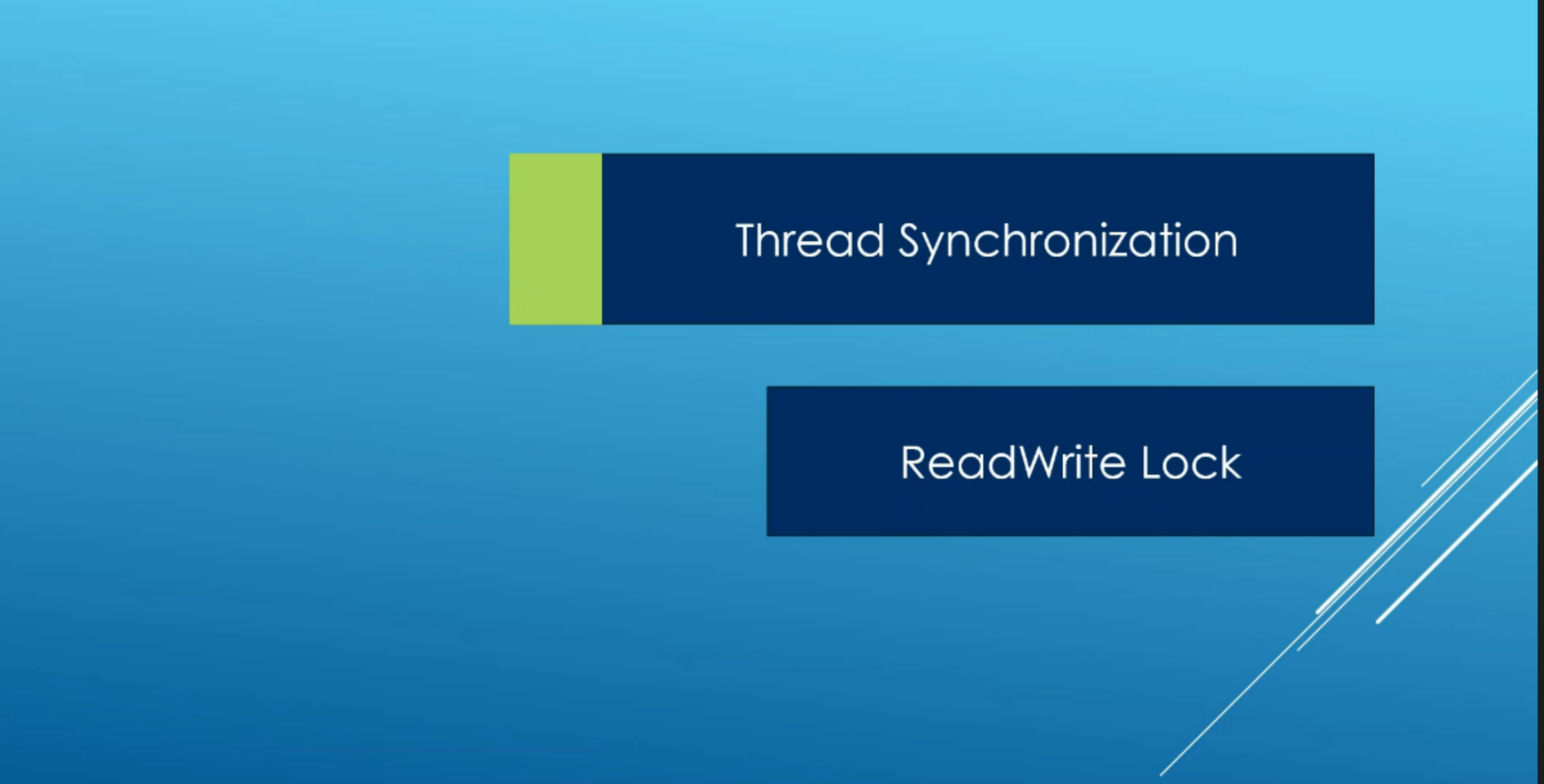
Section 3: Thread Synchronization

2 / 8 | 2hr 9min

- [x] 15. Thread Synchronization Overview — 4min
- [x] 16. Thread Synchronization - Mutex — 21min — Resources
- [ ] 17. Thread Synchronization - Mutex -Shared variable — 15min — Resources
- [ ] 18. Thread Synchronization - ReadWrite Lock — 17min — Resources
- [ ] 19. Thread Synchronization - Semaphores — 30min — Resources
- [ ] 20. Thread Synchronization - WaitConditions — 21min — Resources
- [ ] 21. Wait Conditions - Pause Resume — 19min — Resources
- [ ] 22. Thread Synchronization- Chapter Review — 2min

Print device

```
class PrintDevice
{
public:
    PrintDevice();

    void print(const QString & text);

private:
    int m_count;
    QMutex m_mutex;
};
```



Course content

Detailed Description

The QReadWriteLock class provides read-write locking.

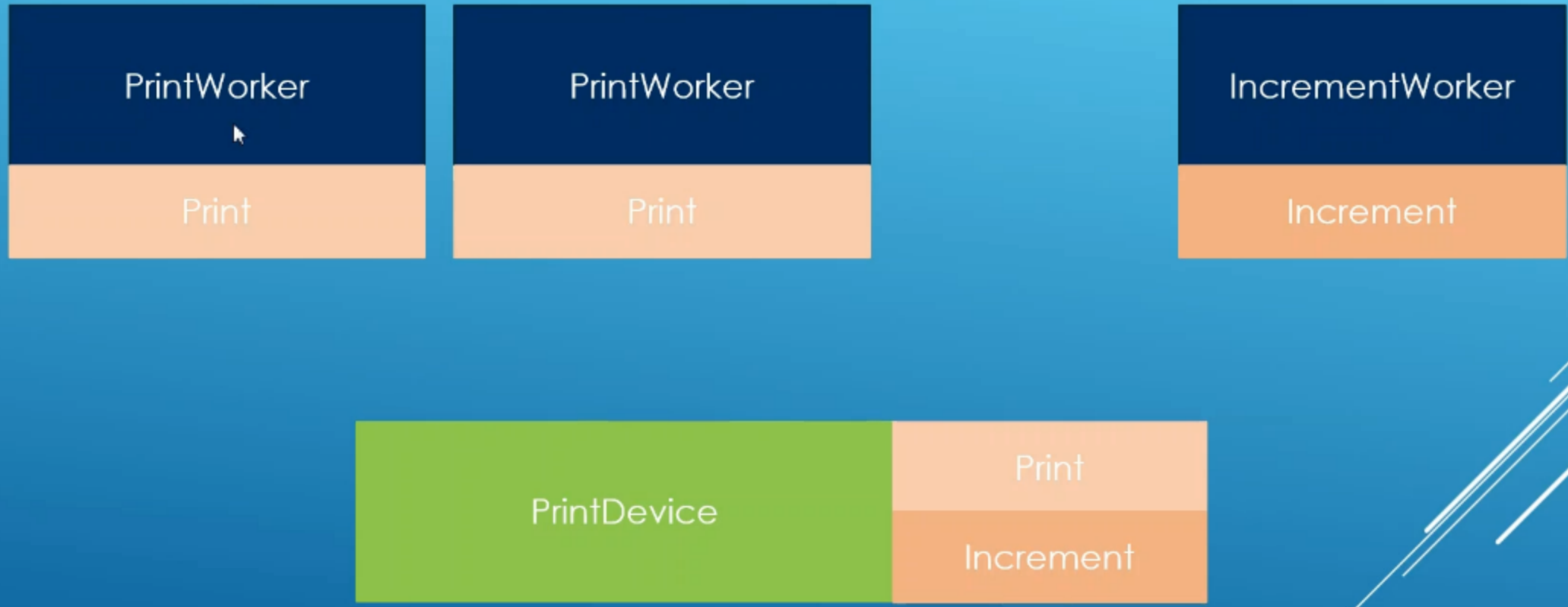
A read-write lock is a synchronization tool for protecting resources that can be accessed for reading and writing. This type of lock is useful if you want to allow multiple threads to have simultaneous read-only access, but as soon as one thread wants to write to the resource, all other threads must be blocked until the writing is complete.

In many cases, QReadWriteLock is a direct competitor to QMutex. QReadWriteLock is a good choice if there are many concurrent reads and writing occurs infrequently.

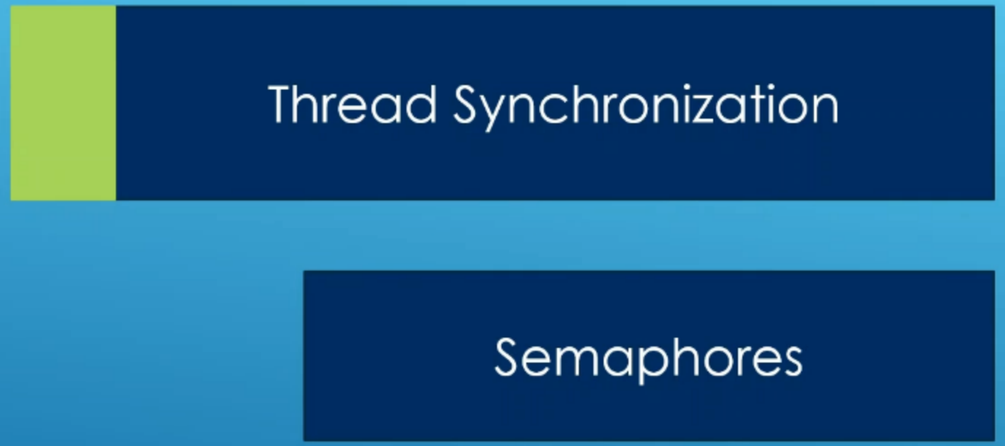
Example:

```
QReadWriteLock lock;

void ReaderThread::run()
{
    ...
    lock.lockForRead();
    read_file();
    lock.unlock();
    ...
}

void WriterThread::run()
{
    ...
    lock.lockForWrite();
    write_file();
    lock.unlock();
    ...
}
```

To ensure that writers aren't blocked forever by readers, readers attempting to obtain a lock will not


PrintWorker
Print
PrintWorker
Print
IncrementWorker
Increment
PrintDevice
Print
Increment

```
int main(int argc, char *argv[])
{
    QApplication a(argc, argv);

    bool stopFlag = false;

    PrintDevice printDevice;

    IncrementWorker incrementWorker(&stopFlag,&printDevice);

    PrintWorker black("Black",&stopFlag,&printDevice),
            white("White",&stopFlag,&printDevice);

    black.start();
    white.start();
    incrementWorker.start();

    QMessageBox::information(nullptr,"QMutex",
                             "Thread working. Close Me to stop");
    stopFlag = true;

    black.wait();
    white.wait();
    incrementWorker.wait();

    return 0;
}
```



Course content

21min Resources

17. Thread Synchronization - Mutex -Shared variable
15min Resources

18. Thread Synchronization - ReadWrite Lock
17min Resources

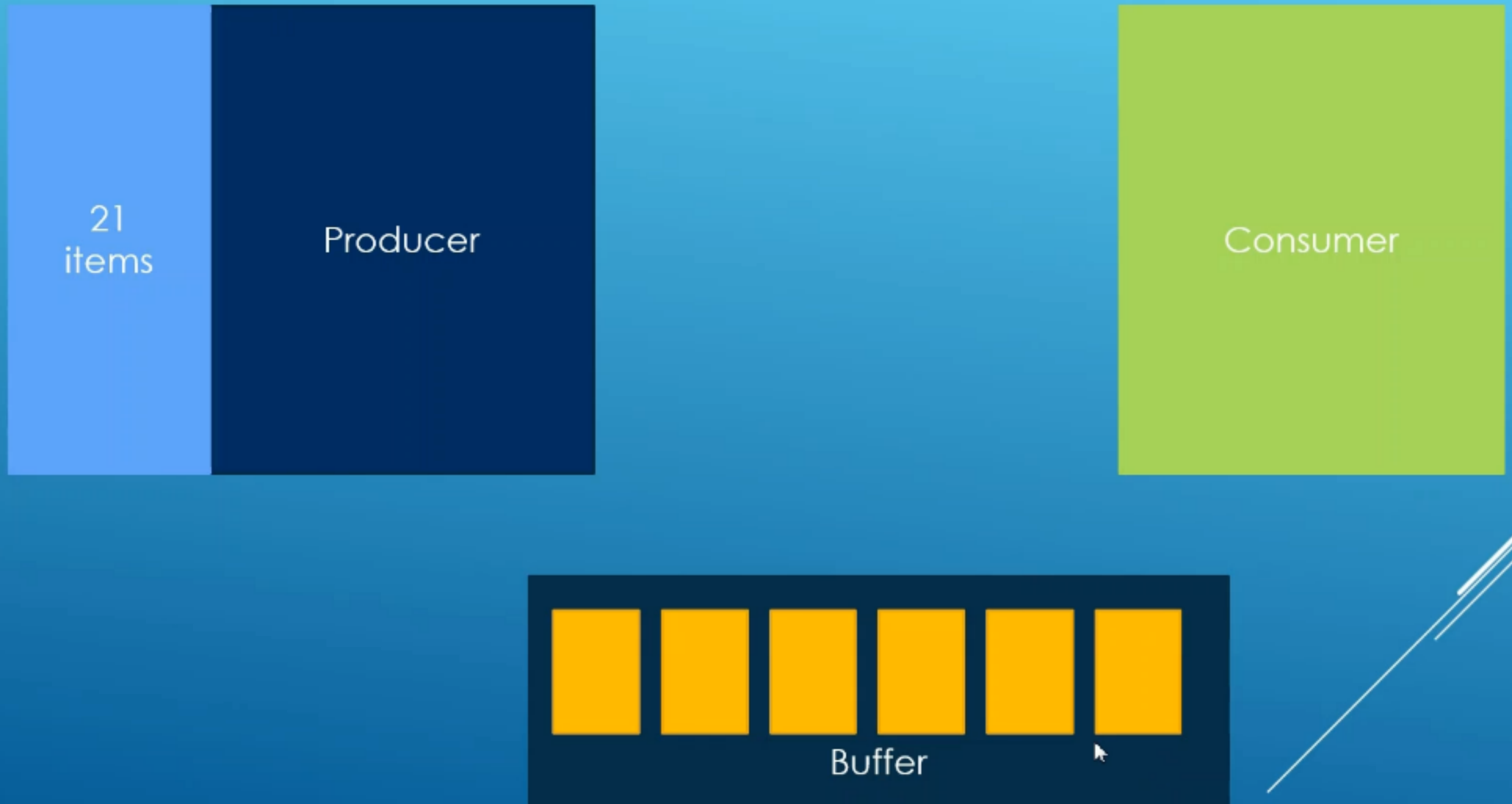
19. Thread Synchronization - Semaphores
30min Resources

20. Thread Synchronization - WaitConditions
21min Resources

21. Wait Conditions - Pause Resume
19min Resources

22. Thread Synchronization- Chapter Review
2min

Section 4: Thread Safety and Reentrancy
0 / 6 | 1hr 10min


21
items
Producer
Consumer
Buffer

Look for: QSemap
QSemaphore
QSemaphoreReleaser
~QSemaphore
~QSemaphoreReleaser

Detailed Description

The QSemaphore class provides a general counting semaphore.

A semaphore is a generalization of a mutex. While a mutex can only be locked once, it's possible to acquire a semaphore multiple times. Semaphores are typically used to protect a certain number of identical resources.

Semaphores support two fundamental operations, acquire() and release():

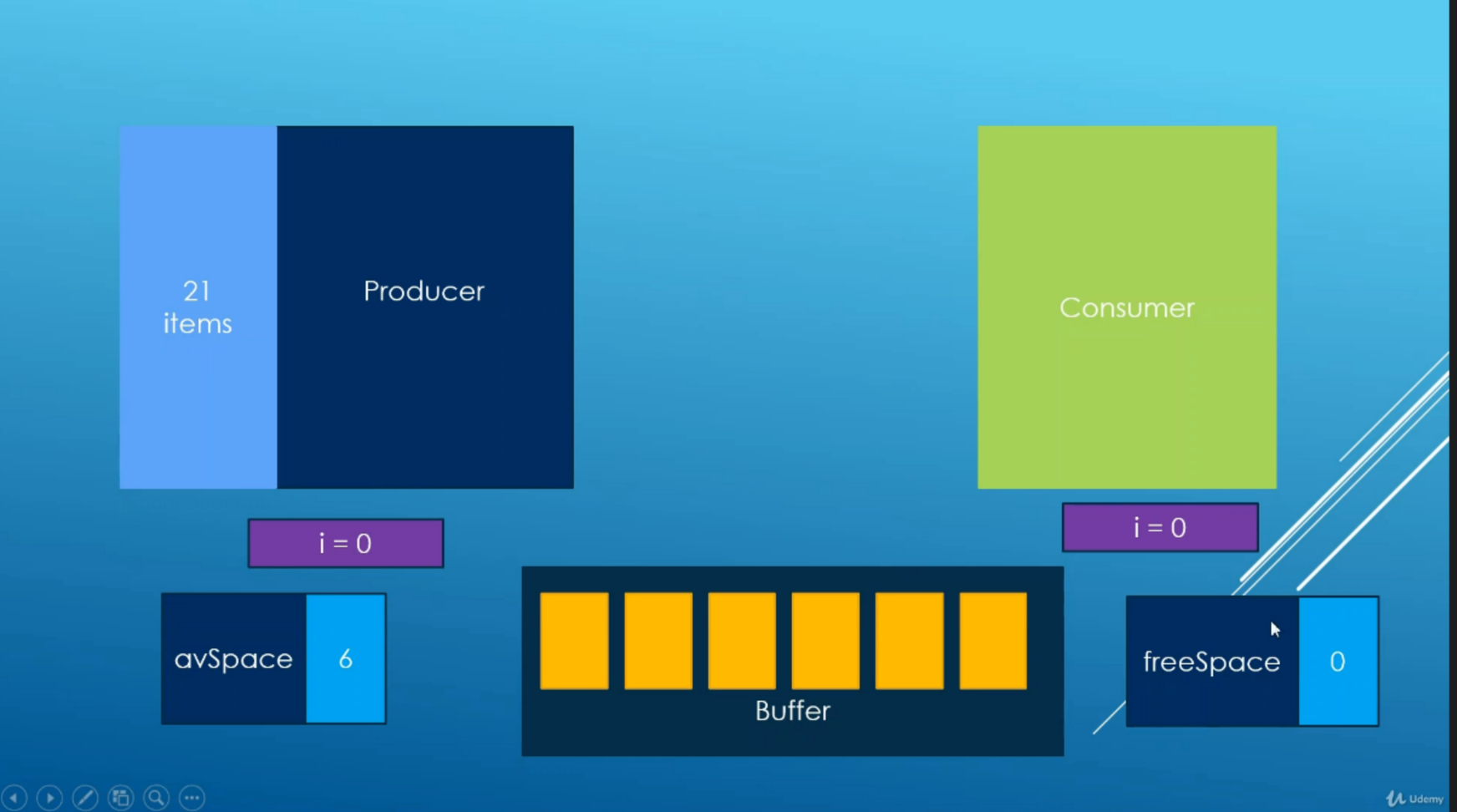
- acquire(*n*) tries to acquire *n* resources. If there aren't that many resources available, the call will block until this is the case.
- release(*n*) releases *n* resources.

There's also a tryAcquire() function that returns immediately if it cannot acquire the resources, and an available() function that returns the number of available resources at any time.

Example:

```
QSemaphore sem(5);      // sem.available() == 5

sem.acquire(3);         // sem.available() == 2
sem.acquire(2);         // sem.available() == 0
sem.release(5);         // sem.available() == 5
sem.release(5);         // sem.available() == 10

sem.tryAcquire(1);      // sem.available() == 9, returns true
sem.tryAcquire(250);    // sem.available() == 9, returns false
```

A typical application of semaphores is for controlling access to a circular buffer shared by a producer thread and a consumer thread. The Semaphores Example shows how to use QSemaphore to solve that problem.



Course content

21min Resources

17. Thread Synchronization - Mutex -Shared variable
15min Resources

18. Thread Synchronization - ReadWrite Lock
17min Resources

19. Thread Synchronization - Semaphores
30min Resources

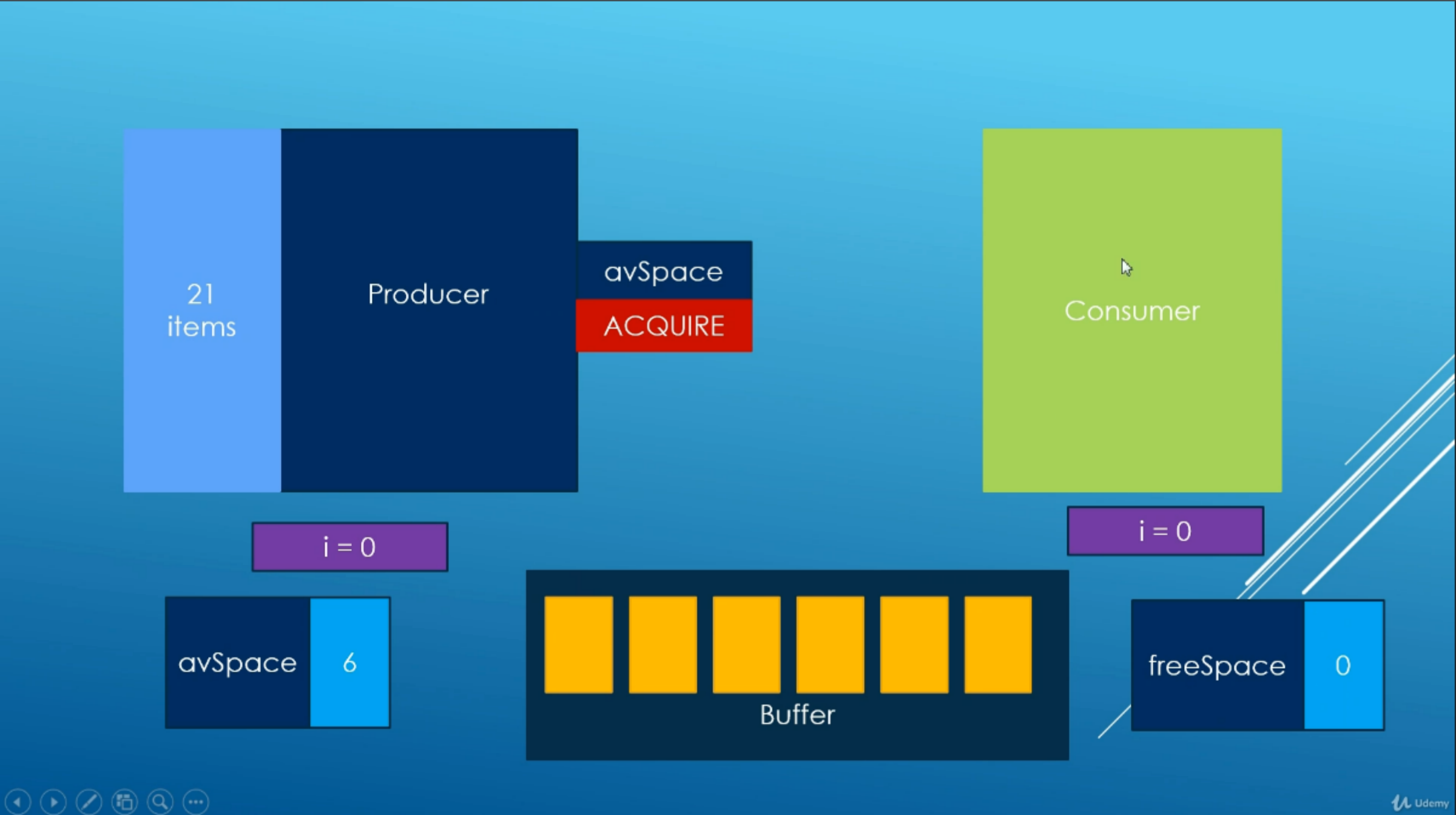
20. Thread Synchronization - WaitConditions
21min Resources

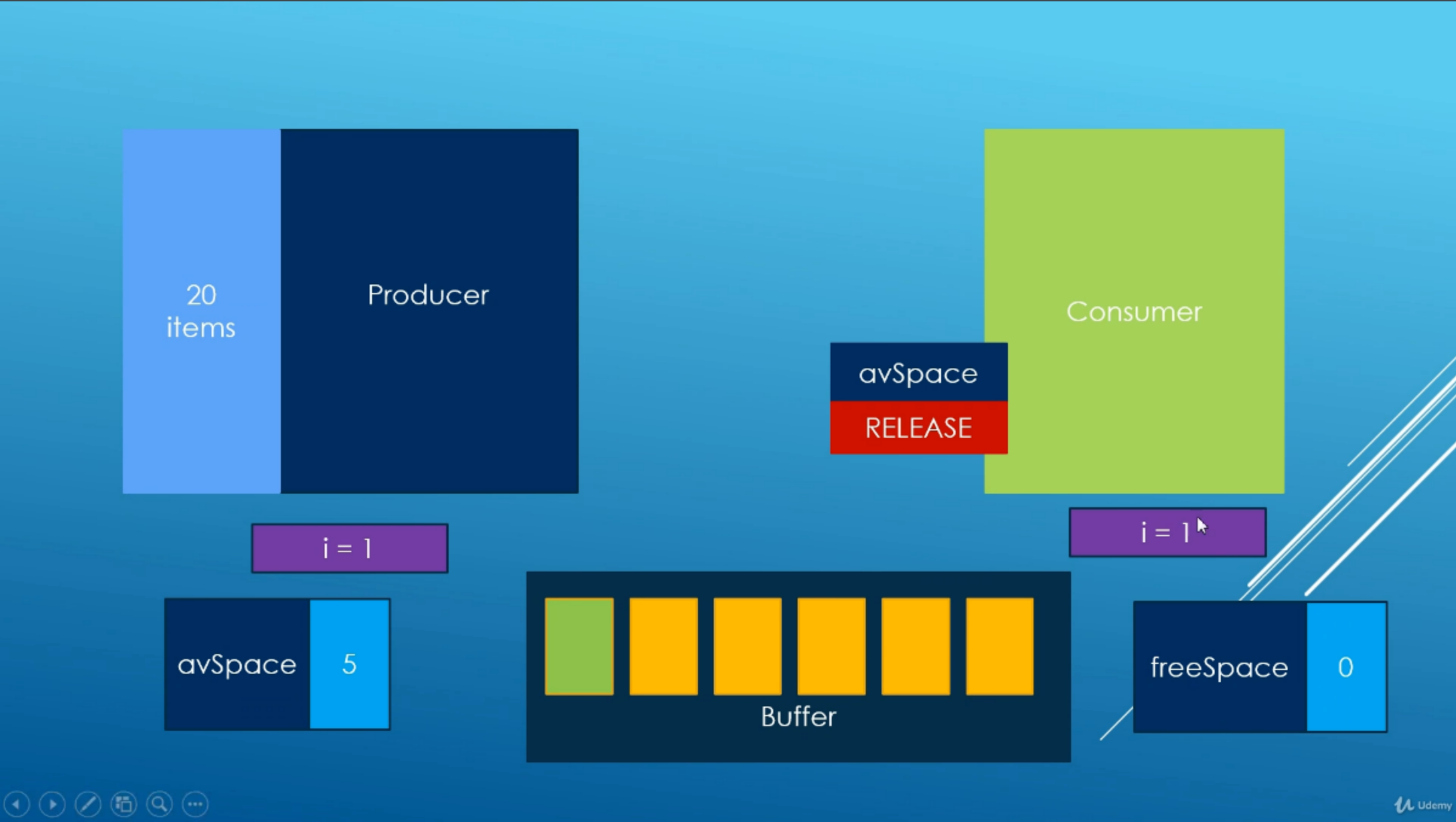
21. Wait Conditions - Pause Resume
19min Resources

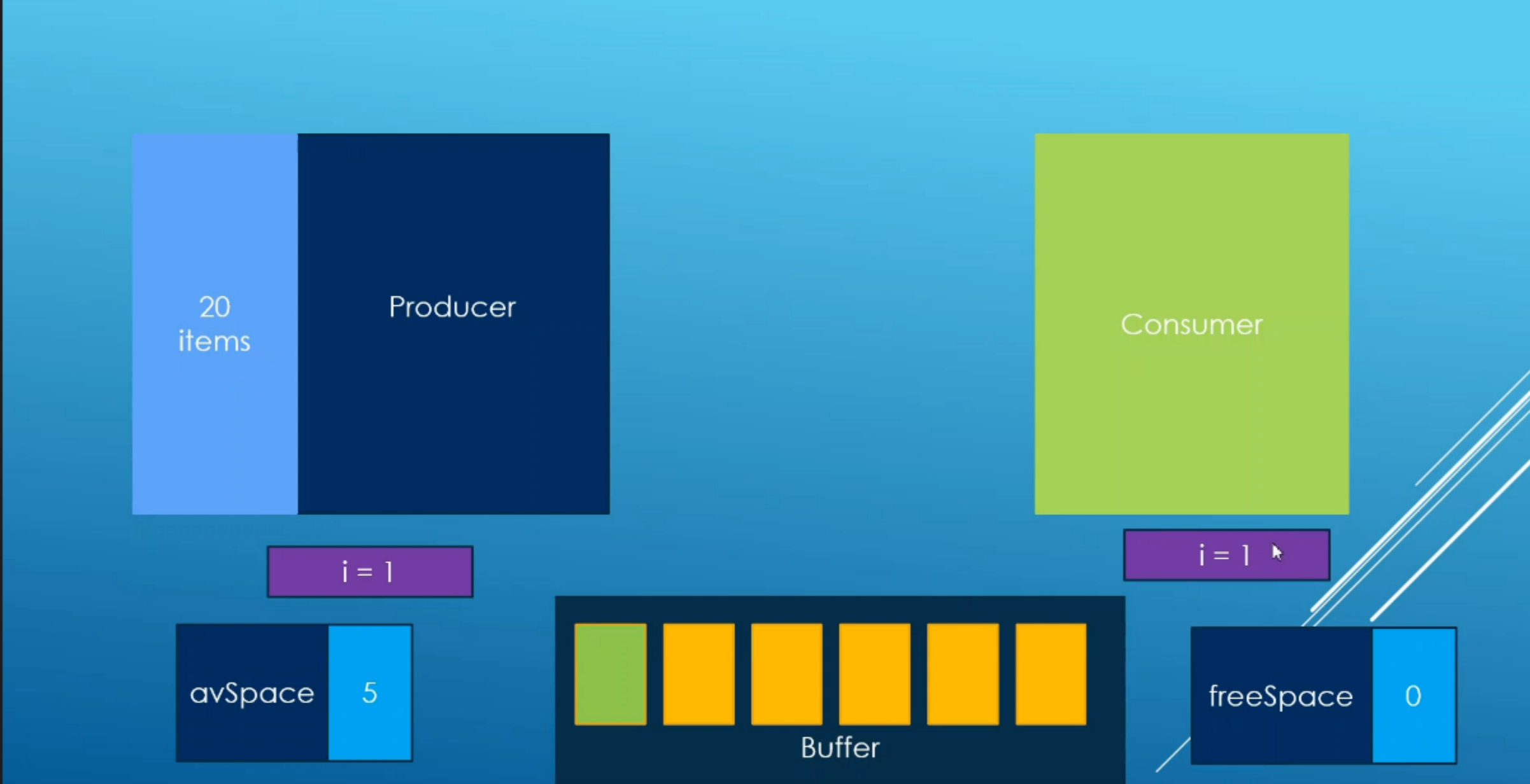
22. Thread Synchronization- Chapter Review
2min

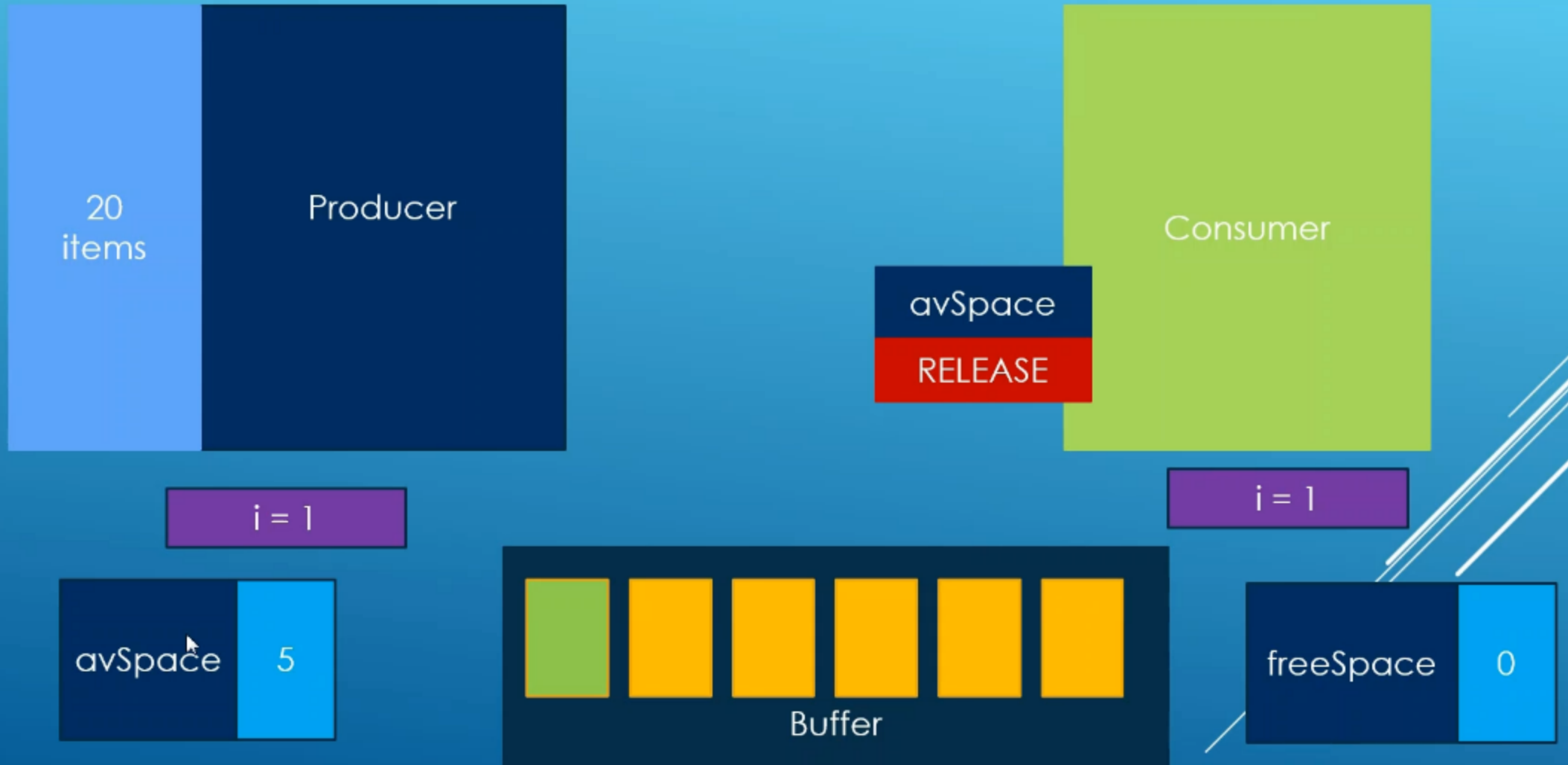
Section 4: Thread Safety and Reentrancy
0 / 6 | 1hr 10min

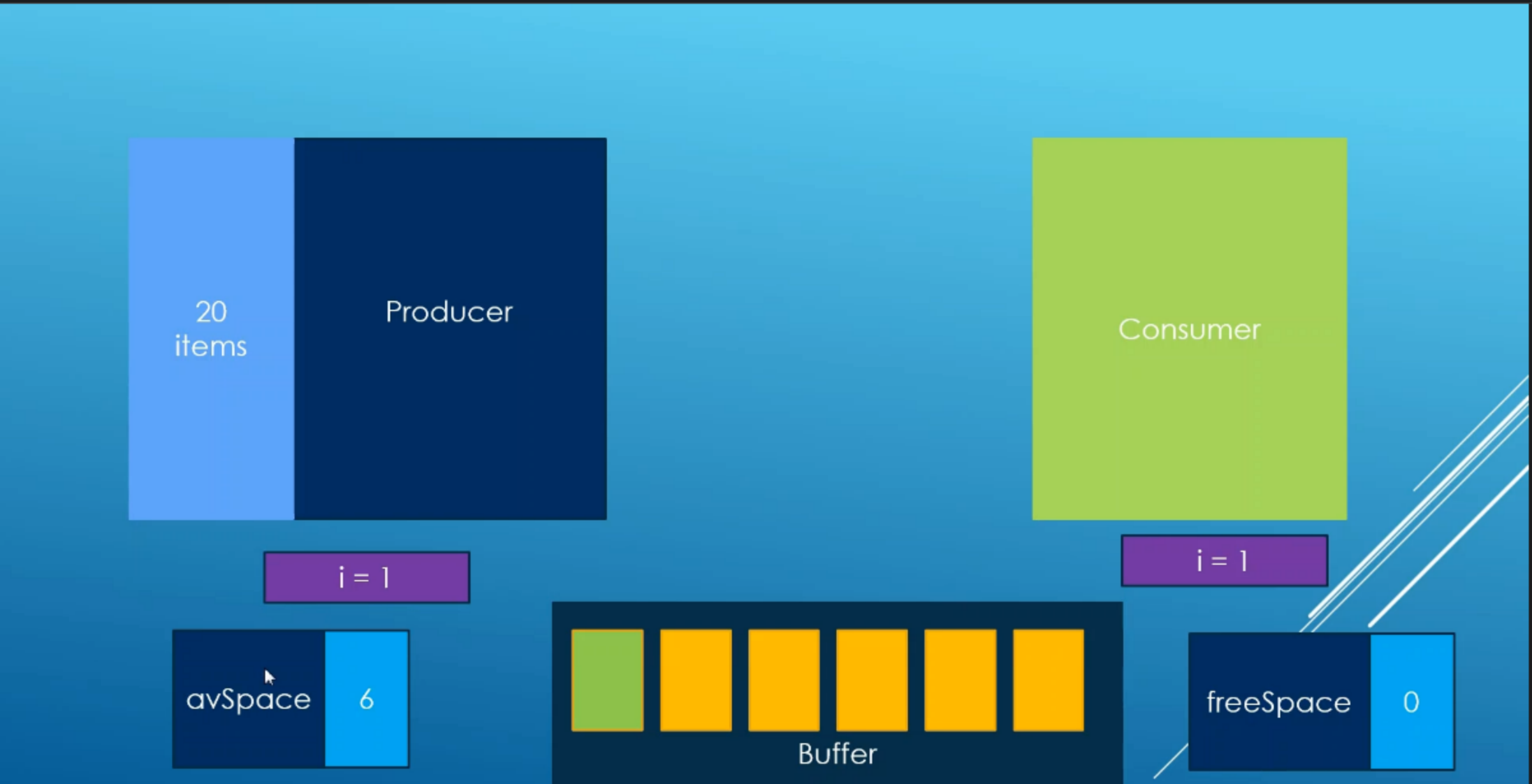
Section 5: Qt Concurrent
0 / 12 | 2hr 37min

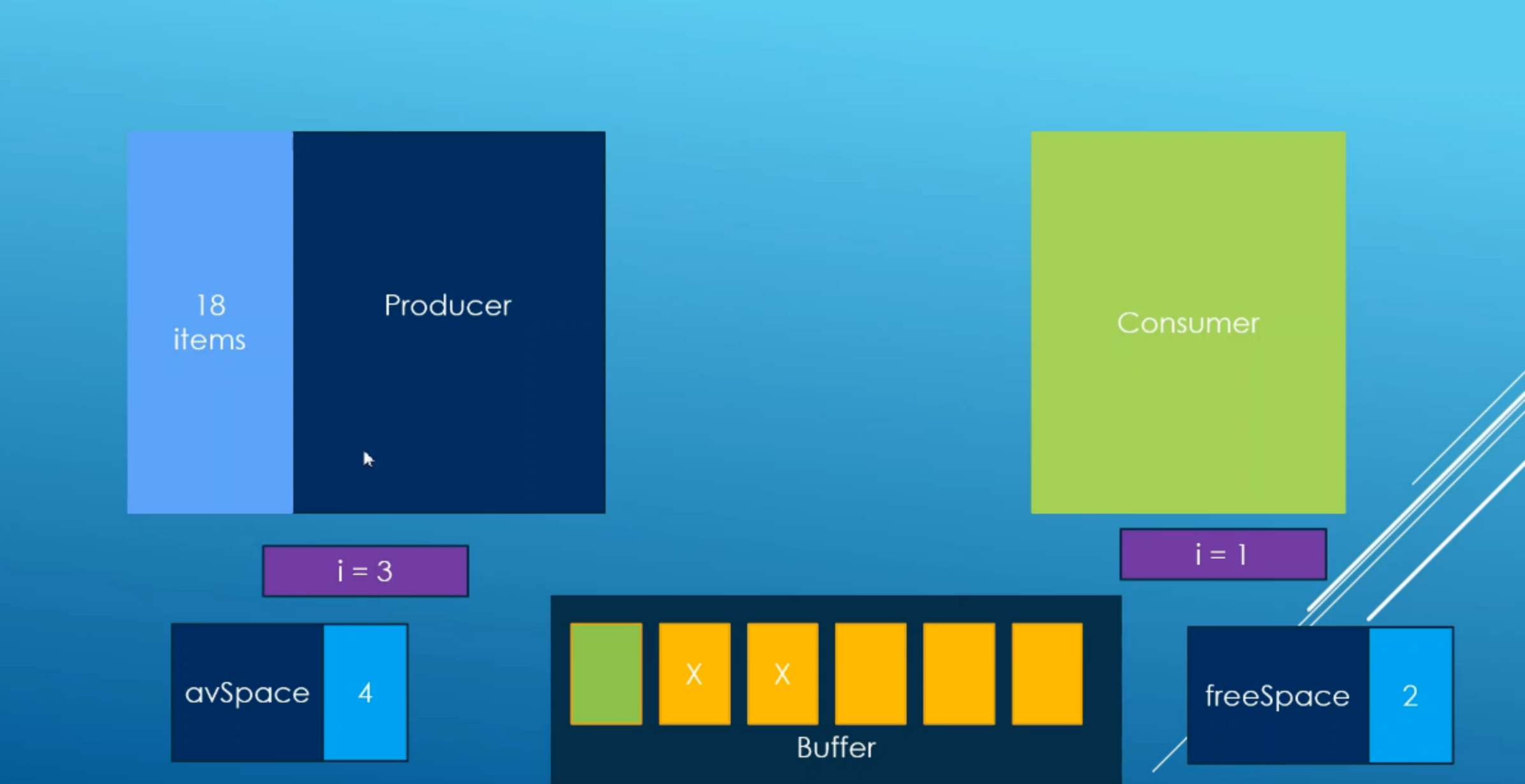

21
items
Producer
avSpace
ACQUIRE
Consumer
i = 0
i = 0
avSpace
6
Buffer
freeSpace
0

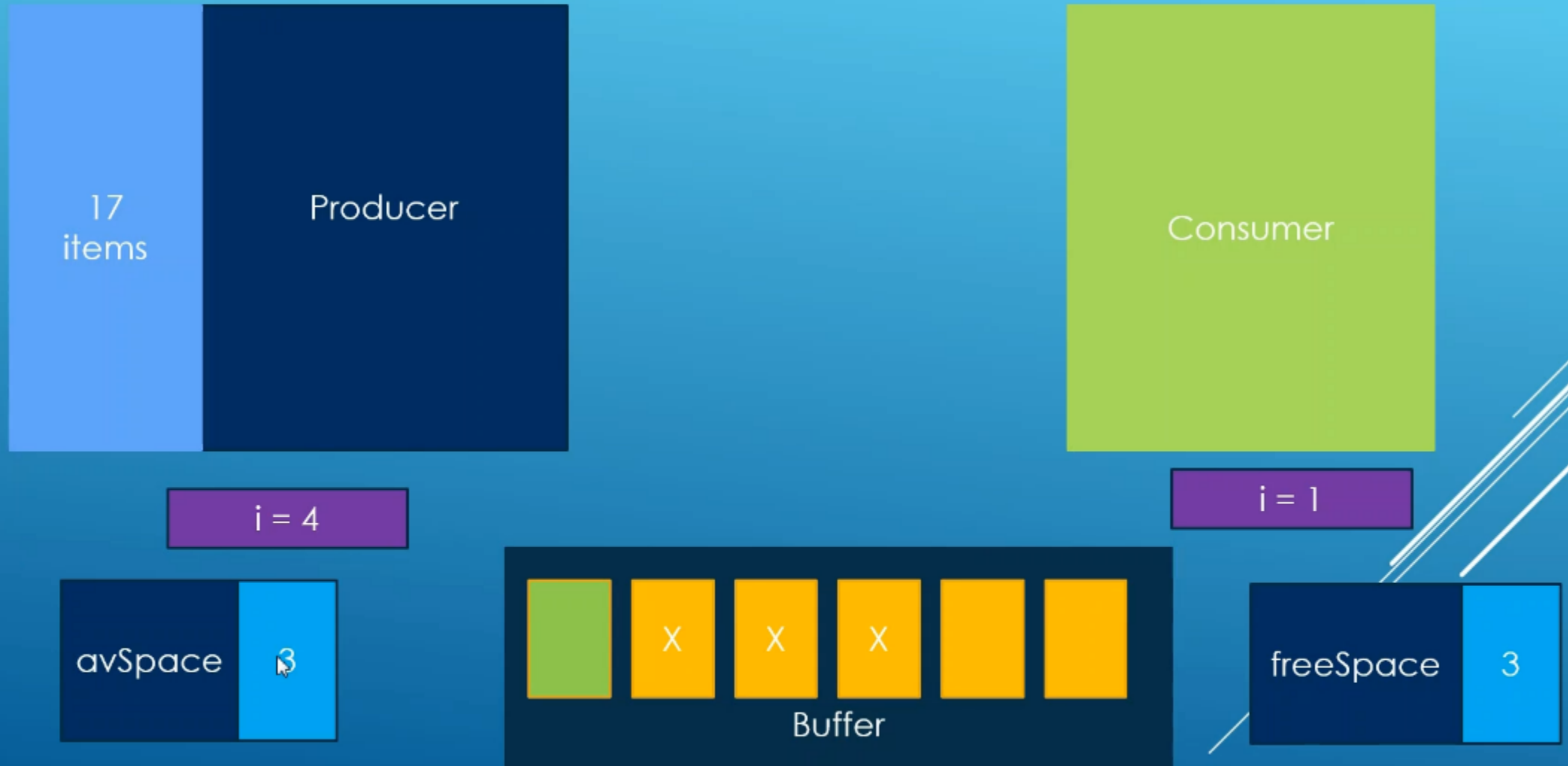

20
items
Producer
Consumer
avSpace
RELEASE
i = 1
i = 1
avSpace
5
Buffer
freeSpace
0

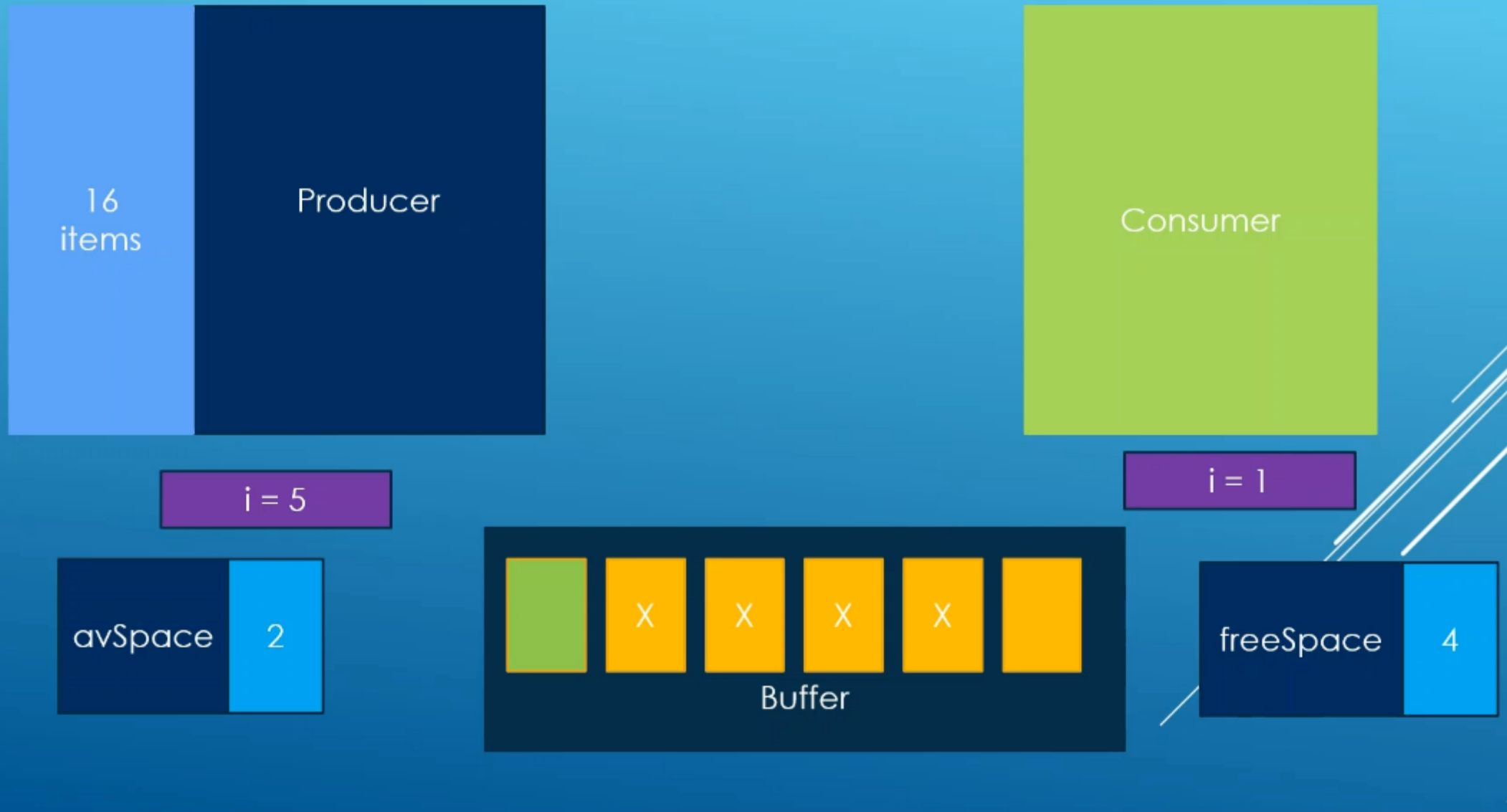

20
items
Producer
Consumer
i = 1
i = 1
avSpace
5
Buffer
freeSpace
0

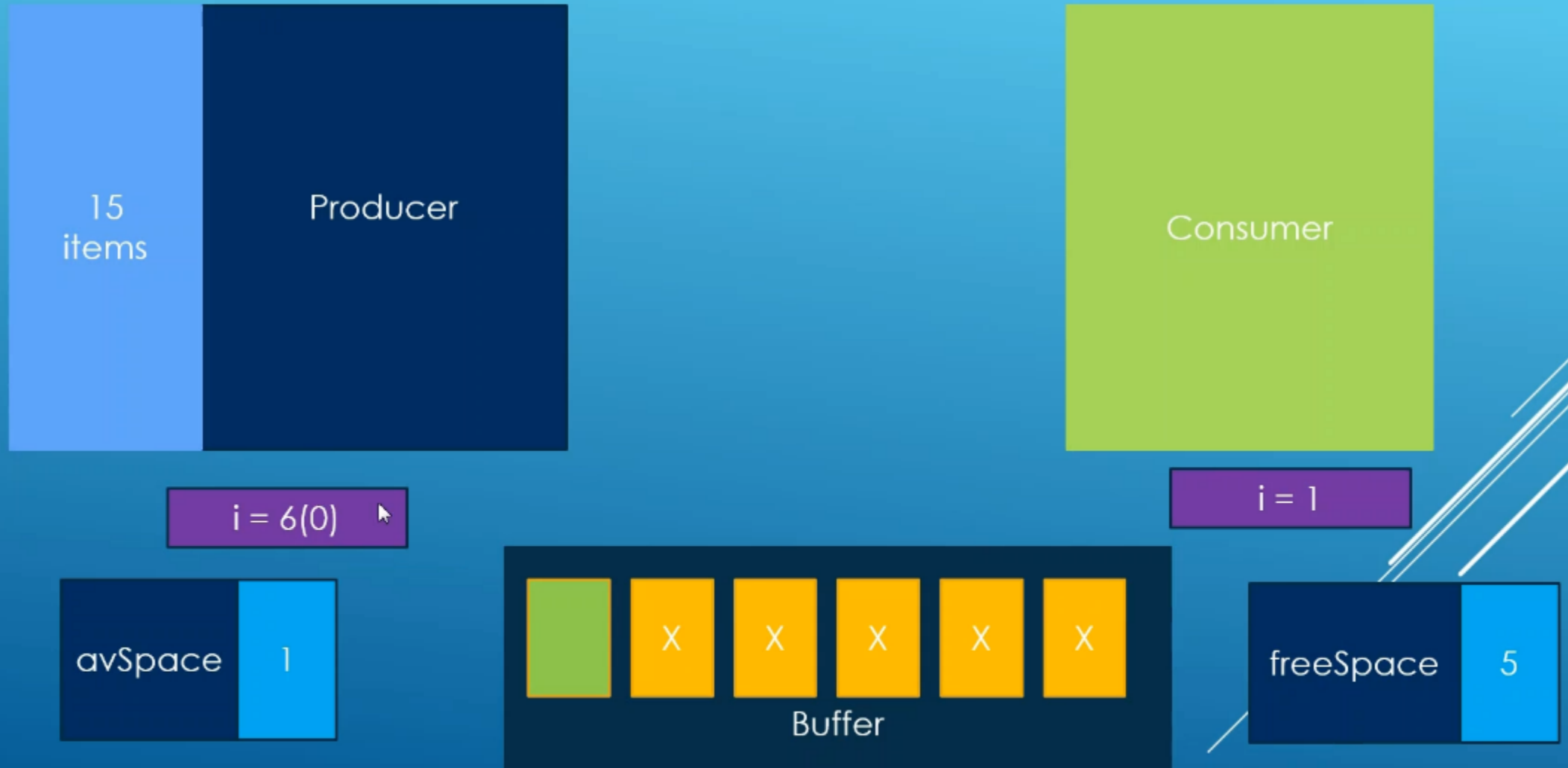

20
items
Producer
Consumer
avSpace
RELEASE
i = 1
i = 1
avSpace
5
Buffer
freeSpace
0

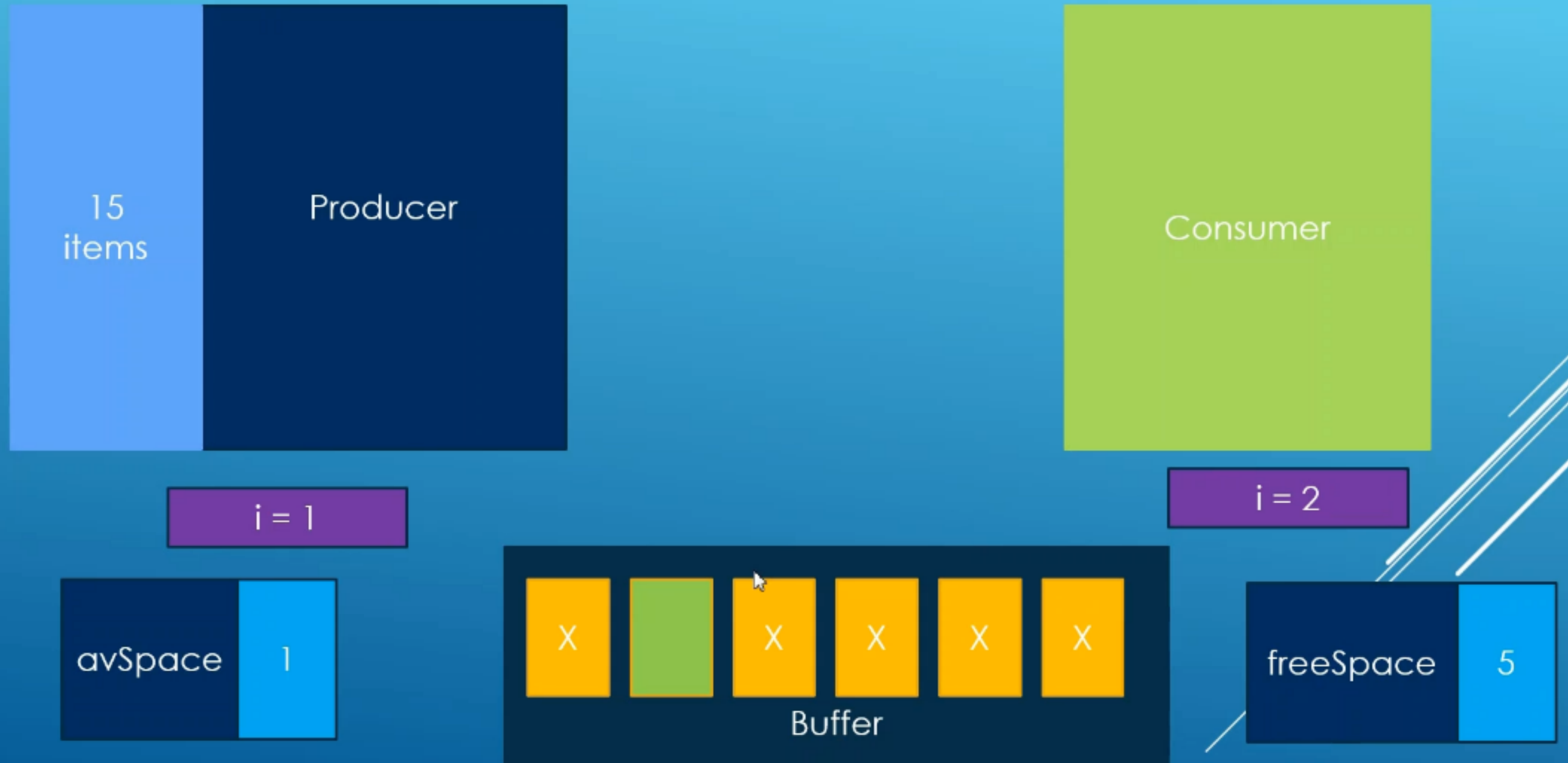

20
items
Producer
Consumer
i = 1
i = 1
avSpace
6
Buffer
freeSpace
0

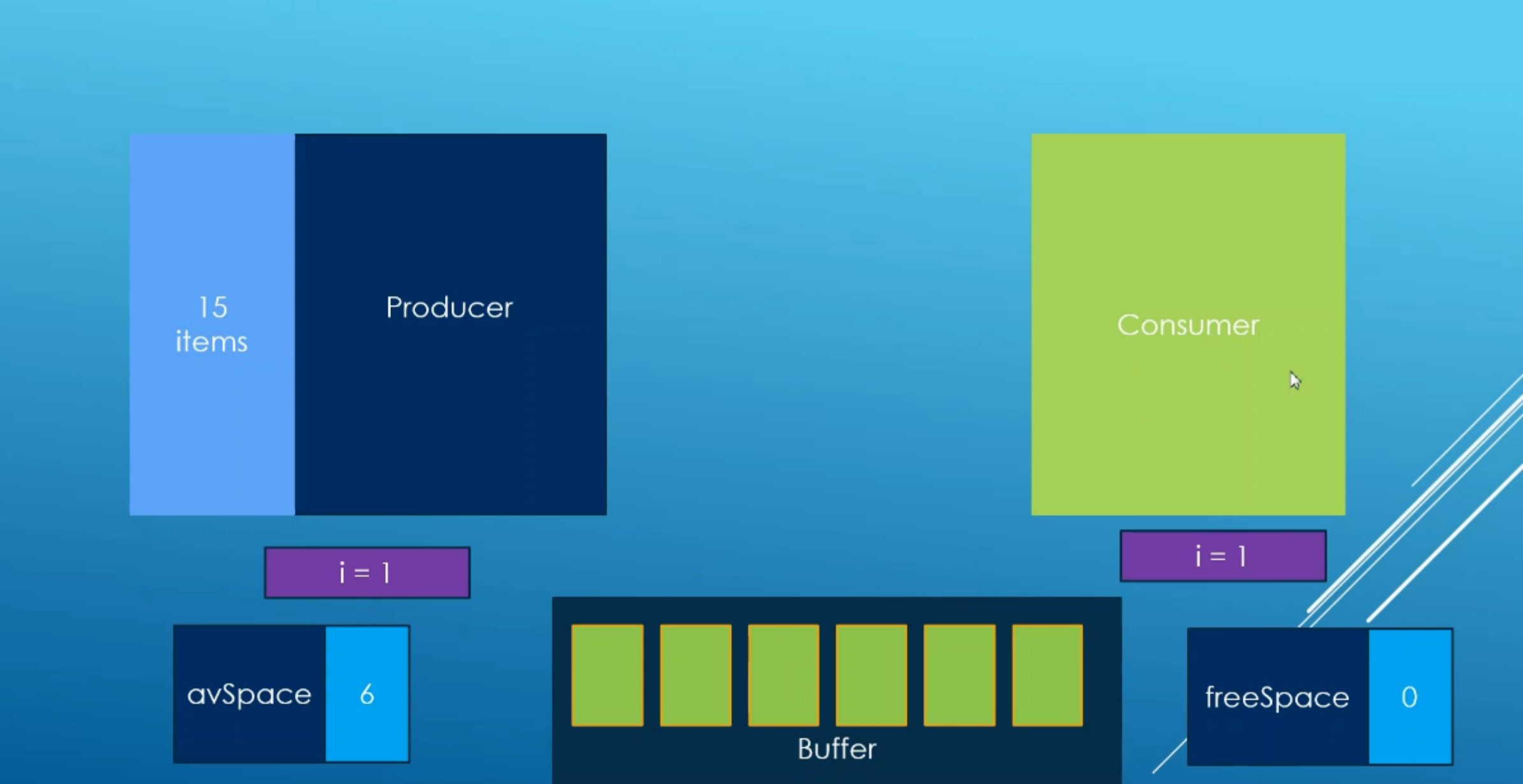

18
items
Producer
Consumer
i = 3
i = 1
avSpace
4
X
X
Buffer
freeSpace
2

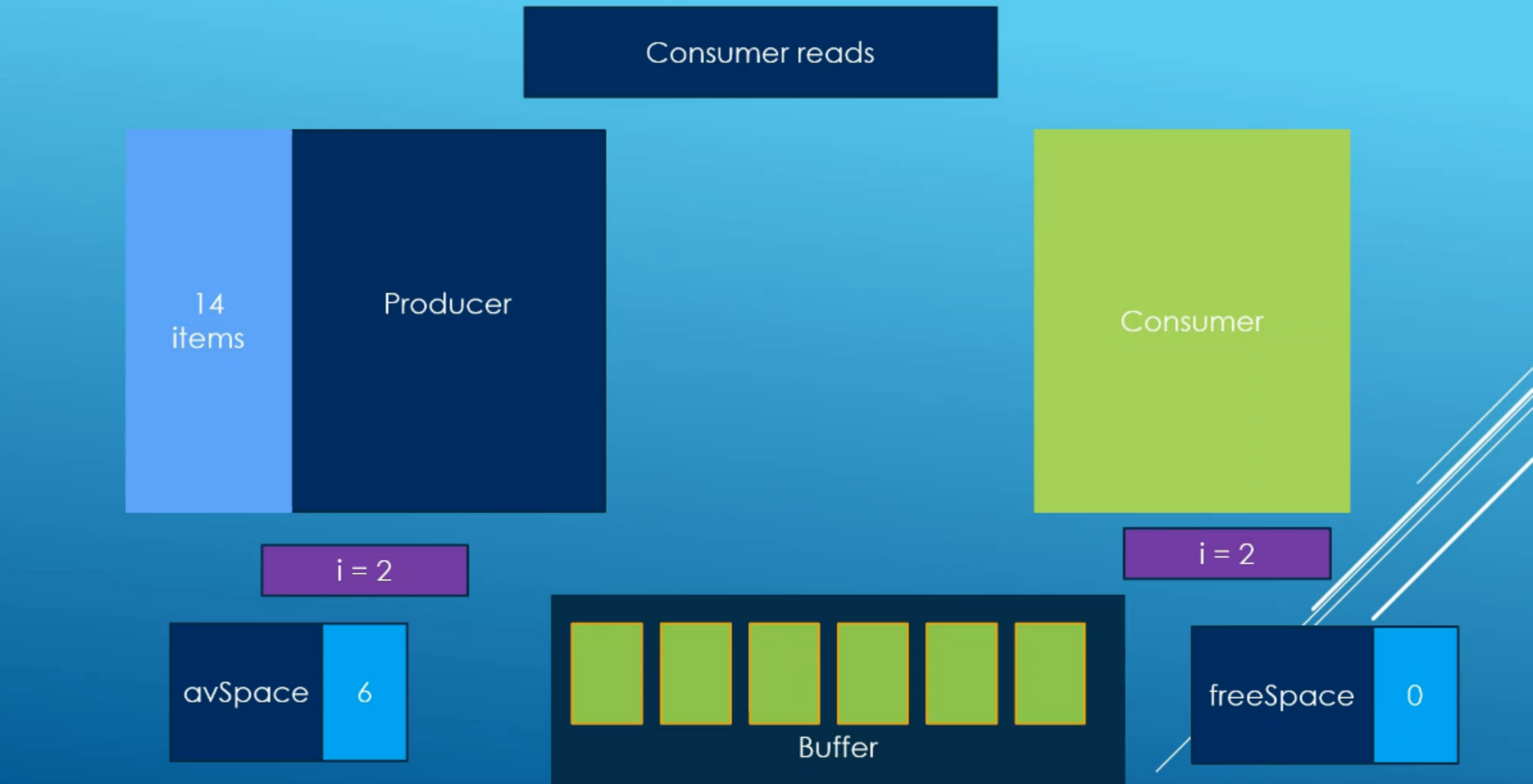

17
items
Producer
Consumer
i = 4
i = 1
avSpace
3
X
X
X
Buffer
freeSpace
3


16
items
Producer
Consumer
i = 5
i = 1
avSpace
2
X
X
X
X
Buffer
freeSpace
4


15
items
Producer
Consumer
i = 6(0)
i = 1
avSpace
1
X
X
X
X
X
Buffer
freeSpace
5


15
items
Producer
Consumer
i = 1
i = 2
avSpace
1
X
X
X
X
X
Buffer
freeSpace
5


15
items
Producer
Consumer
i = 1
i = 1
avSpace
6
Buffer
freeSpace
0


Consumer reads
14
items
Producer
Consumer
i = 2
i = 2
avSpace
6
Buffer
freeSpace
0

This will keep going on until there are no items left for the producer to write in the buffer

Producer

```
class Producer : public QThread
{
    Q_OBJECT
public:
    explicit Producer(const QVector<int> & dataSource,
                      int * buffer,int bufferSize,QSemaphore * freeSpace,
                     QSemaphore * availableSpace,
                     bool * atEnd,QObject *parent = nullptr);


    // QThread interface
protected:
    void run() override;
private:
    QVector<int> m_data_source;
    int * m_buffer;
    int m_BUFFER_SIZE;
    QSemaphore * m_free_space;// Space where the producer can write data
    QSemaphore * m_available_space;//Space where consumer can read data from
    bool * m_at_end;
};
```

Producer

```
Producer::Producer(const QVector<int> & dataSource,  int * buffer,
                   int bufferSize,QSemaphore *freeSpace,
                   QSemaphore *availableSpace, bool *atEnd, QObject *parent) :
    QThread (parent) ,
    m_data_source(dataSource),
    m_buffer(buffer),
    m_BUFFER_SIZE(bufferSize),
    m_free_space(freeSpace),
    m_available_space(availableSpace),
    m_at_end(atEnd)
{
}

void Producer::run()
{
    for(int i{0} ; i < m_data_source.length() ; i++){
        m_free_space->acquire();
        /*    ...
        m_buffer[i % m_BUFFER_SIZE] = m_data_source.at(i);
        //qDebug() <<  m_buffer[i % 6] << " written into buffer";

        if(i == m_data_source.length() -1)
            *m_at_end = true;

        //Signal that there is something to read for the consumer
        m_available_space->release();
    }
}
```