CORRECTNESS CRITERIA FOR CONCURRENCY & PARALLELISM

Contracts – Correctness for Sequential Code

Contracts – Correctness for Sequential Code

- Assertions
  - A predicate expected to hold at a particular program point
- Precondition
  - A predicate expected to hold at a function call
  - A failure can be blamed on the caller
- Postcondition
  - A predicate expected to hold at a function return
  - A failure can be blamed on the callee

Code Contracts for .Net

```
int Divide(int n, int d) {
   return n/d;
}
```

Preconditions using Requires

```
int Divide(int n, int d) {
  Contract.Requires( 0 != d );
  return n/d;
}
```

Preconditions using Ensures

```
int Divide(int n, int d) {
  Contract.Requires( 0 != d );
  Contract.Ensures(
     Contract.Result<int>() * d <= n &&
     Contract.Result<int>() * d > n-d
  );

  return n/d;
}
```

Example: Library APIs

vector::push_back

```
void push_back ( const T& x );
```

Add element at the end

Adds a new element at the end of the vector, after its current last element. The c
initialized to a copy of *x*.

This effectively increases the vector size by one, which causes a reallocation of th
vector size was equal to the vector capacity before the call. Reallocations invalida
references and pointers.

Parameters

x

Value to be copied to the new element.
`T` is the first template parameter (the type of the elements stored in the ve

Return value

none

If a reallocation happens, it is performed using `Allocator::allocate()`, which ma
`allocator`, `bad_alloc` is thrown if the allocation request does not succeed).

Example: System Call API

NAME

connect -- initiate a connection on a socket

SYNOPSIS

```
#include <sys/types.h>
#include <sys/socket.h>

int
connect(int socket, const struct sockaddr *address,
    socklen_t address_len);
```

DESCRIPTION

The parameter *socket* is a socket. If it is of type SOCK_DGRAM, this call specifies the peer with which the socket is to be associated; this address is that to which datagrams are to be sent, and the only address from which datagrams are to be received. If the socket is of type SOCK_STREAM, this call attempts to make a connection to another socket. The other socket is specified by *address*, which is an address in the communications space of the socket.

Each communications space interprets the *address* parameter in its own way. Generally, stream sockets may successfully **connect**() only once; datagram sockets may use **connect**() multiple times to change their association. Datagram sockets may dissolve the association by connecting to an invalid address, such as a null address or an address with the address family set to AF_UNSPEC (the error EAFNOSUPPORT will be harmlessly returned).

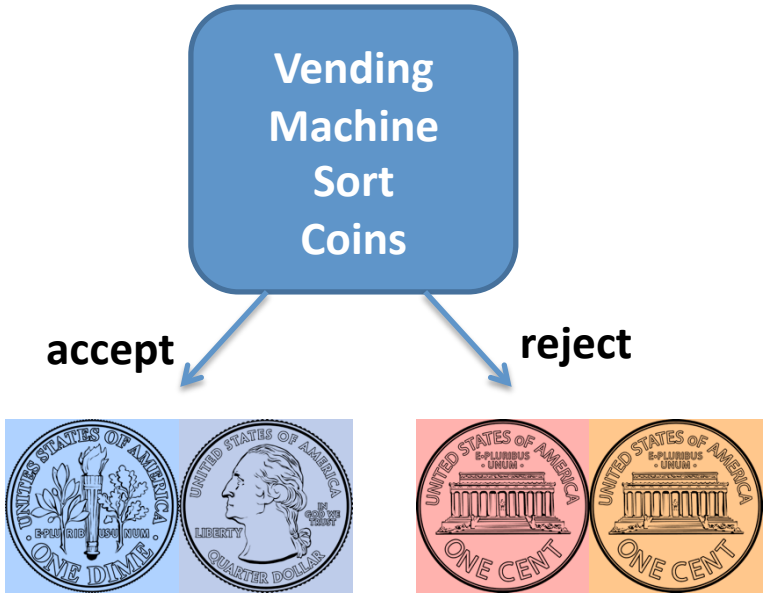
RETURN VALUES

Upon successful completion, a value of 0 is returned. Otherwise, a value of -1 is returned and the global integer variable *errno* is set to indicate the error.

Correctness for Concurrency & Parallelism

- Reuse contracts written for sequential code
- Relate correctness of concurrent/parallel executions to correctness of appropriate sequential executions

Coin Sorting Example



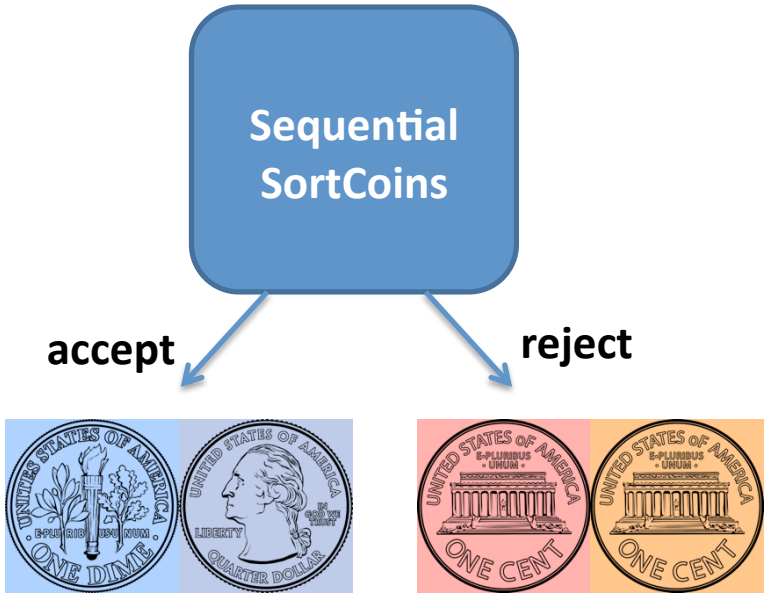
Use Contracts for Correctness Criteria

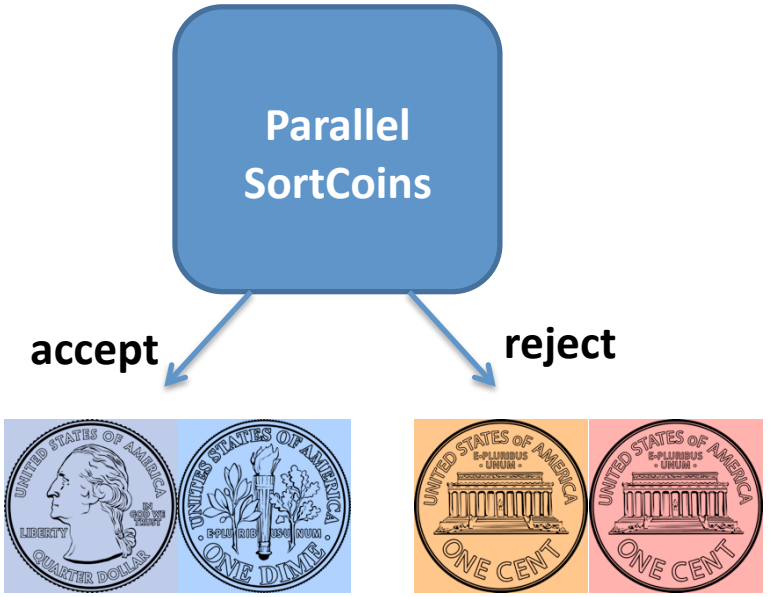
- SortCoins accepts a set of coins and returns a set of bad ones
- Parallelizing SortCoins should not change the contract

```
SortCoins(…) {
    Contract.Requires(…);
    Contract.Ensures(…);
    Sequential Implementation
}
```

```
SortCoins(…) {
    Contract.Requires(…);
    Contract.Ensures(…);
    Parallel Implementation
}
```

Parallelizing can sometimes produce correct but different outputs





The order of coins can change

Restrictive Contracts Can Limit Parallelism

- The order of coins returned by SortCoins might be different from the input order
- Do care about the total amount returned
- If the contracts enforce the ordering, resulting parallelization might be unacceptably slow
- Design interfaces in such a way that contracts are not restrictive

Strategies For Reasoning About Parallel Code

- How do we know if a parallel loop satisfies its contract?
  - Reasoning about parallel executions is hard

General Strategy:

1. Make sure that every parallel behavior is equal to some sequential behavior
2. Convince that the sequential behavior satisfies the contract

Independent Loops

- Let m(1) … m(n) be the loop iterations
- Two iterations m(j) and m(k) (j ≠ k) are *dependent* if they access the same memory location and at least one of them is a write.
- Loop iterations are *independent* if no two of them are dependent

Dependent Loops

- Dependencies need to be made explicit
  - No data races
- Reason that order of dependent operations don't matter
  - e.g. These operations are commutative and associative
  - Recall: reduce/scan

Determinism:
A New Correctness Criteria

- Pre and post conditions do two things
  - Specify how the function behaves sequentially
  - Enforce the same behavior when parallelized

```
int ComputeSum ( IEnumerable<int> input)
{
  Contract.Requires ( input != null);

  Contract.Ensures (
      Contract.Result<int>() == input.Sum(i => i)
    ));

   //implementation
}
```

Determinism Contract

- Allows you to check parallel correctness
- Without having to specify the sequential contract
  - The output of the function does not depend on task interleavings for a given input

```
int ComputeSum ( IEnumerable<int> input)
{
  Contract.Requires ( input != null);

  IsDeterministic (
           Contract.Result<int>(), input );

   //implementation
}
```

Determinism Checking

```
Contract.IsDeterministic (
    output, {input1, input2, … })
```

Is same as saying

```
Contract.Ensures (
     output == F ( input1, input2, … ) )
```

For some deterministic function F

Very useful when specifying F is tedious

Determinism Checking

```
Contract.IsDeterministic (
    output, {input1, input2, … }, comp)
```

Is same as saying

```
Contract.Ensures (
   comp(output, F ( input1, input2,
… ) )
```

For some deterministic function F

Strategies for Checking Determinism

Concurrent Objects

- Can be called concurrently by many threads
- Examples
  - Work Stealing Queue

Concurrent Objects

- Can be called concurrently by many entities
- Examples
  - Work Stealing Queue
  - C Runtime library
  - Operating System
  - Data bases

Correctness Criteria

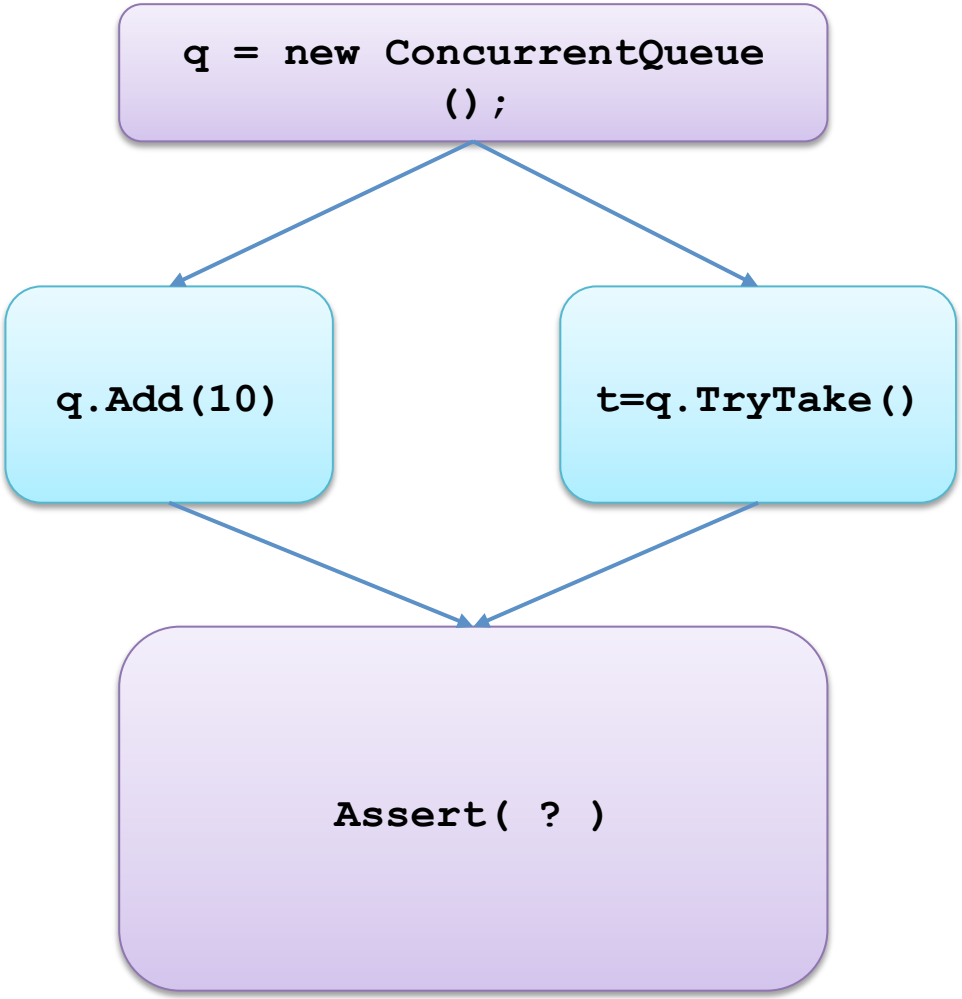
- Informally called “thread safety”
- What does “thread safety” mean to you?

A Simple Concurrent Object

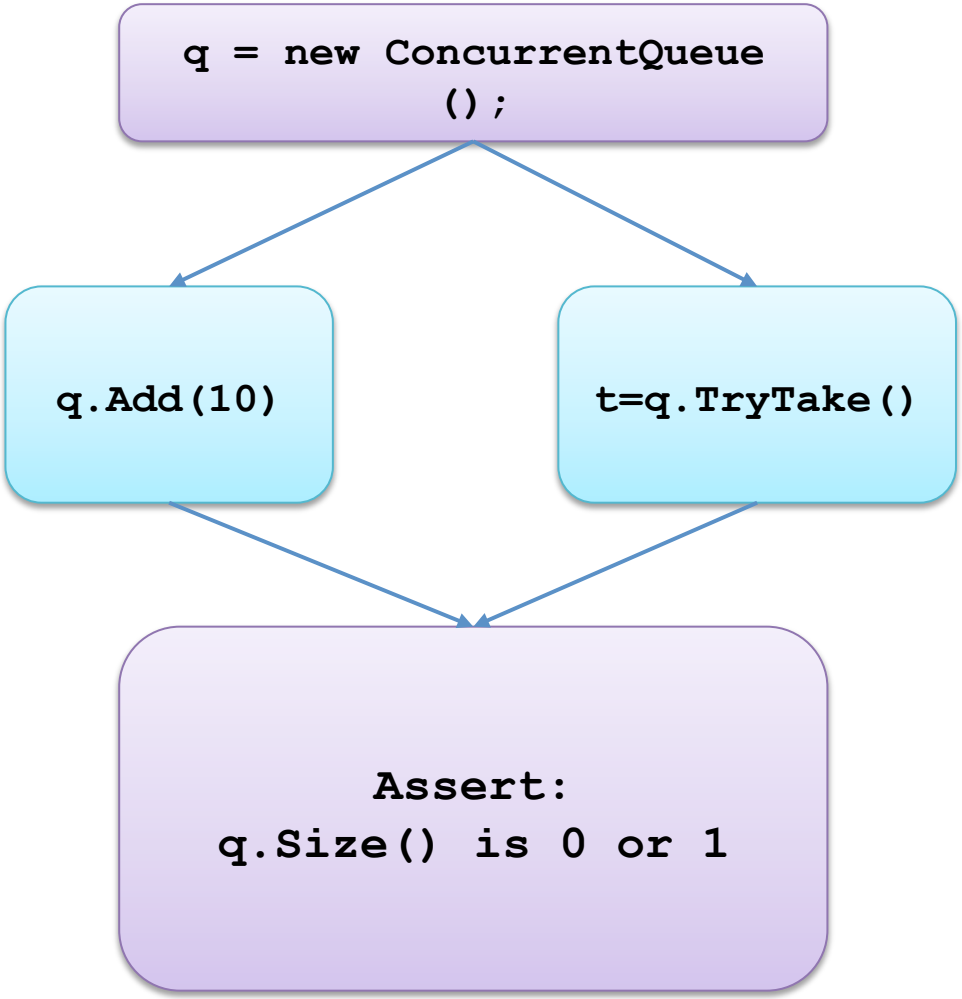
- Sequential Queue
  - Add(item)
  - TryTake() returns an item or “empty”
  - Size() returns # of items in queue

- Consider ConcurrentQueue and its relationship to Queue

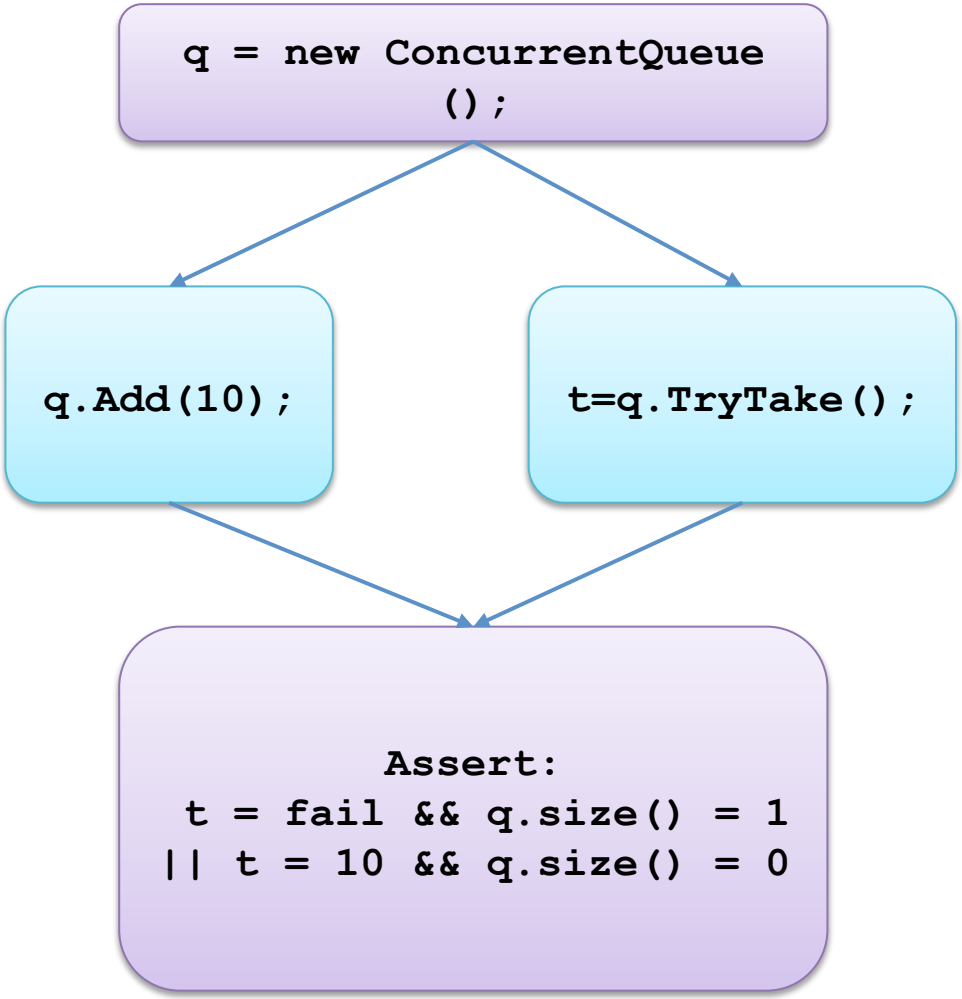
Let's Write a Test



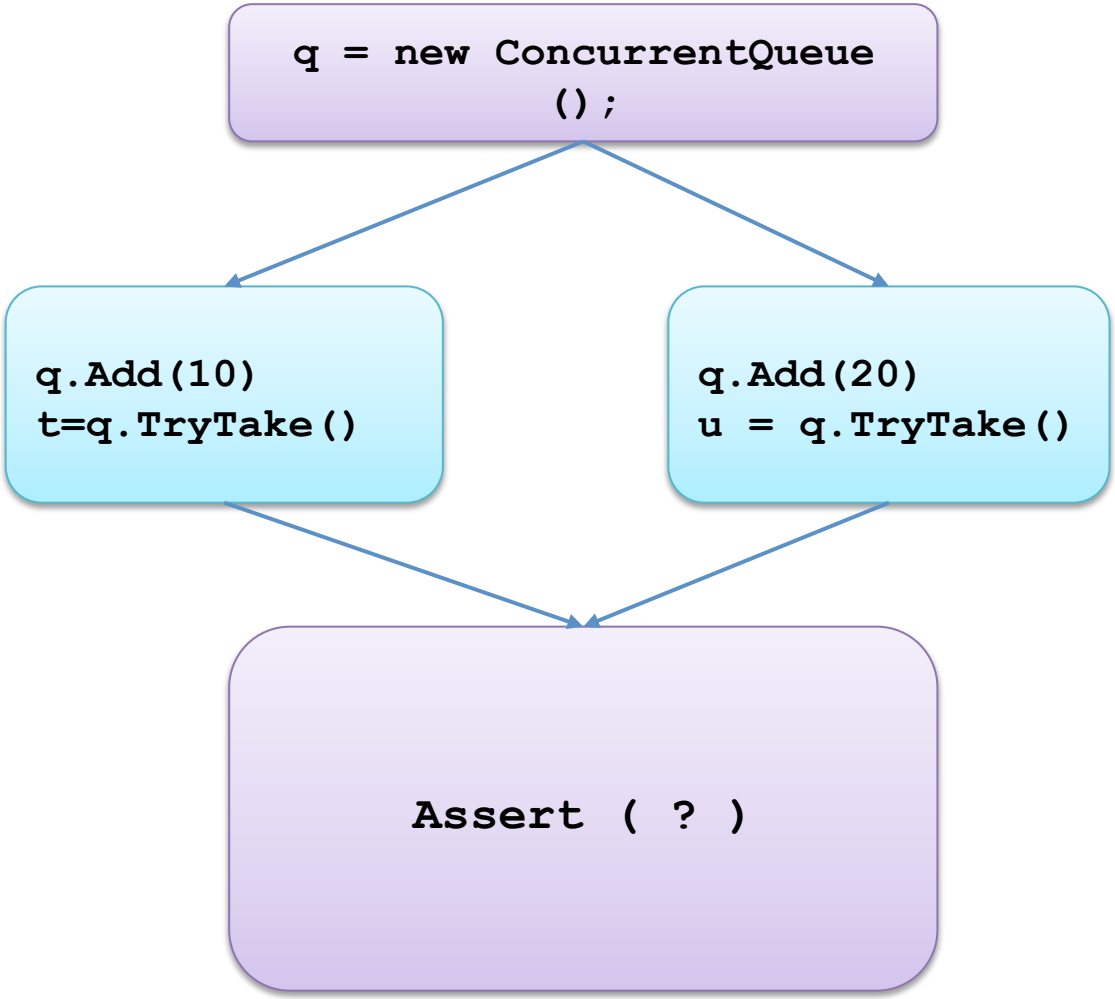
Let's Write a Test



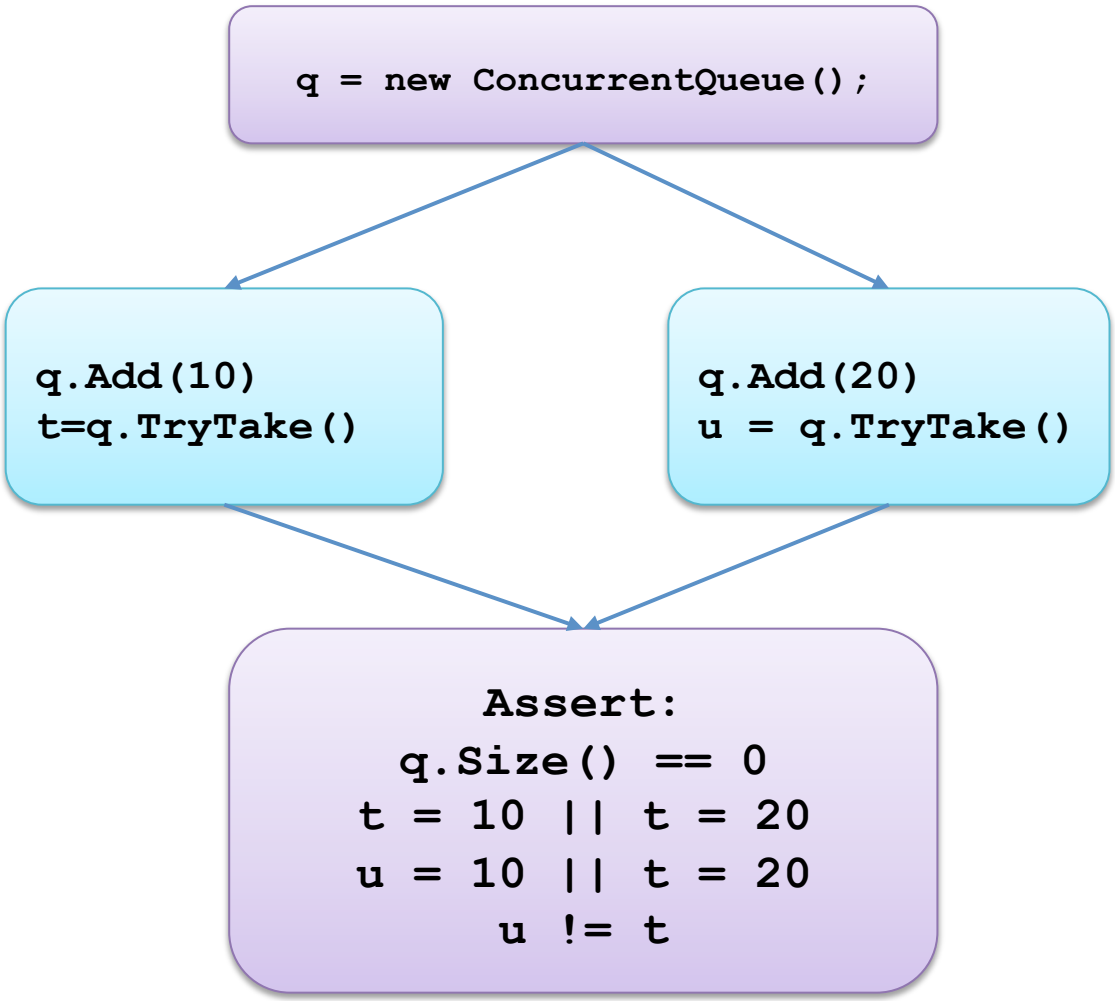
Let's Write a Test



Let's Write a Test



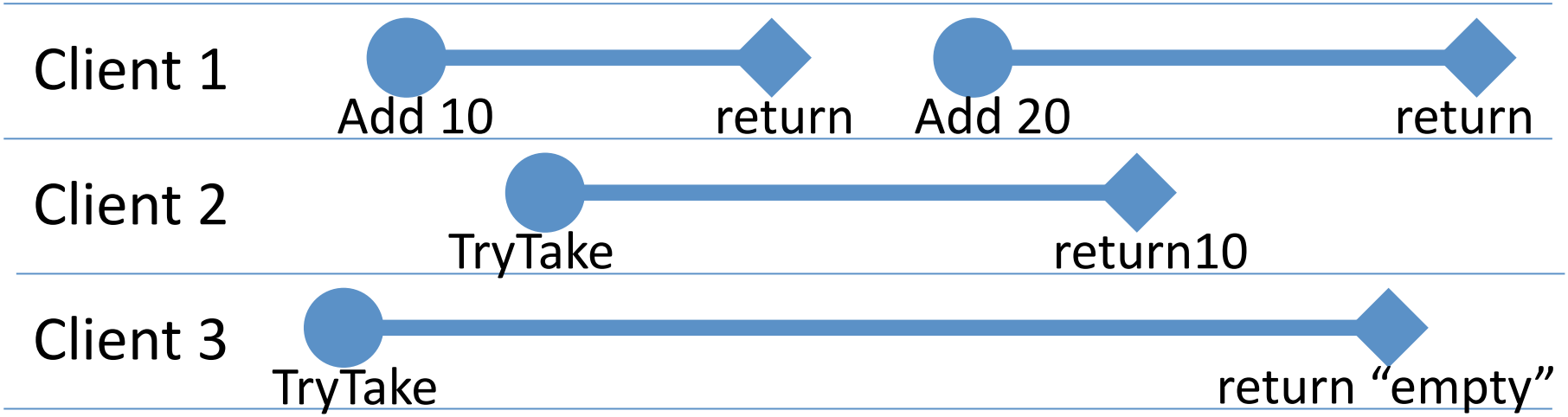
Let's Write a Test



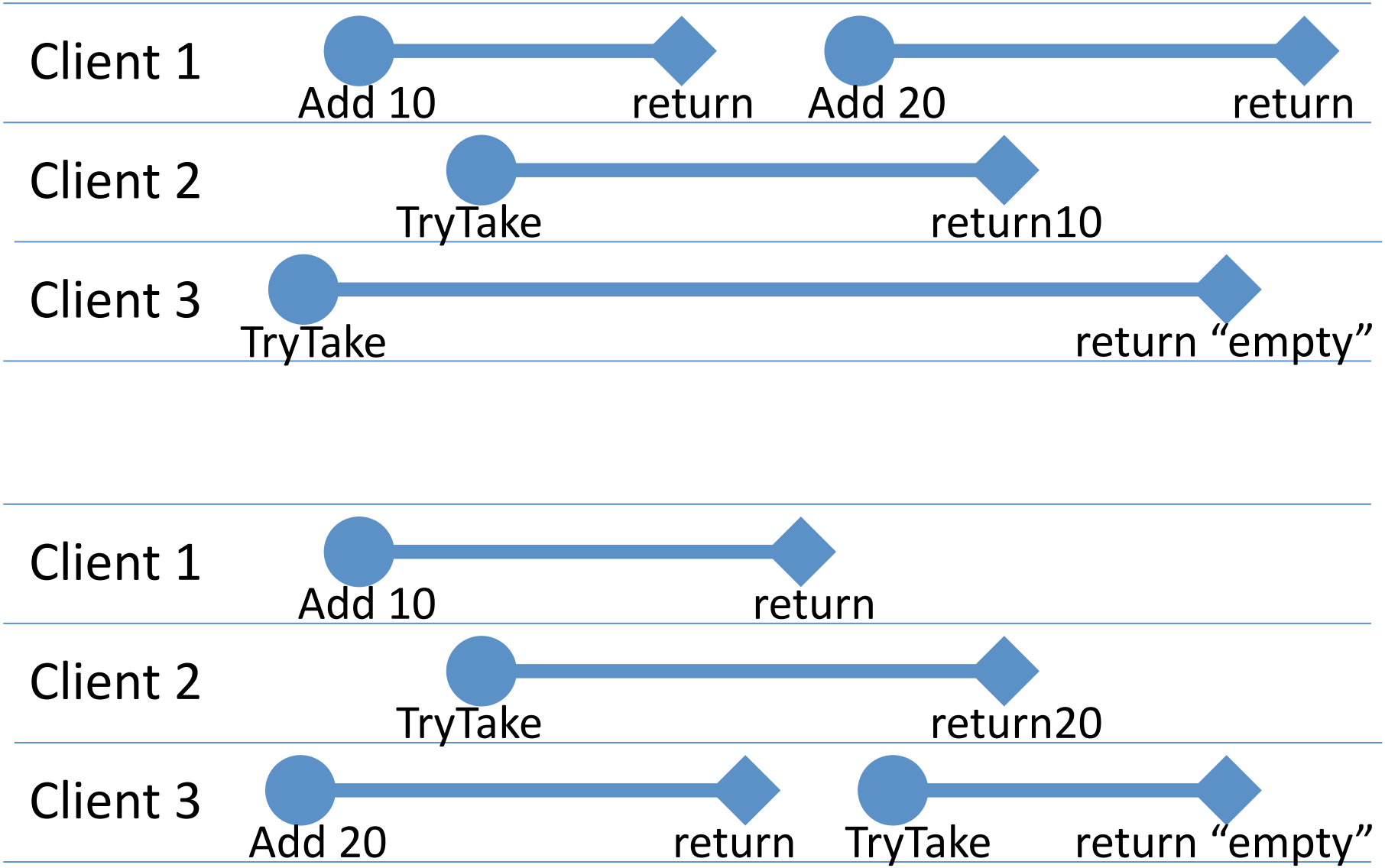
Linearizability

- The correctness notion closest to "thread safety"
- A concurrent component behaves ***as if*** only one thread can enter the component at a time

"Expected" Behavior?



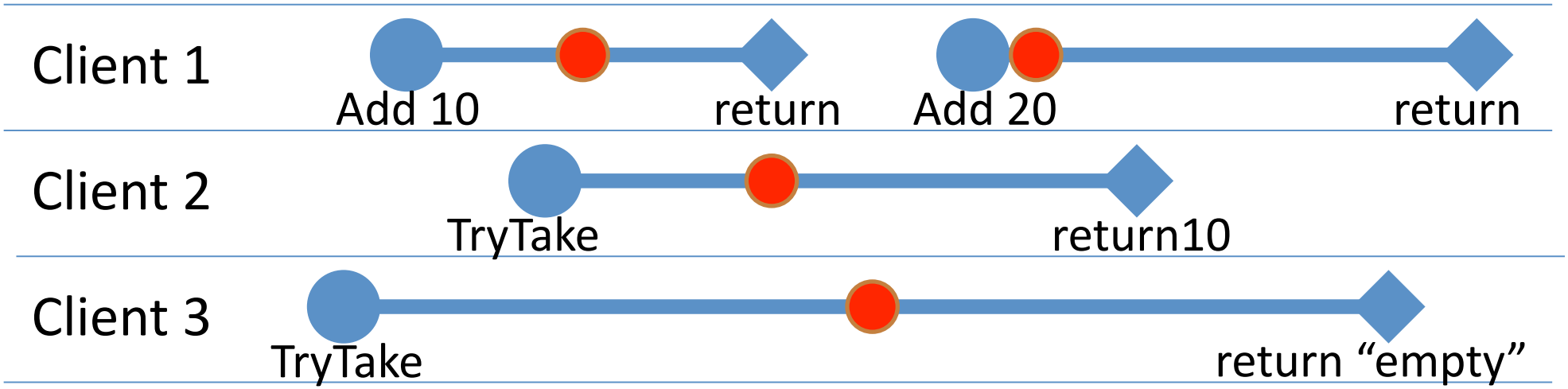
"Expected" Behavior?



Linearizability

- Component is *linearizable* if all operations
  - Appear to take effect atomically at a single temporal point
  - And that point is between the call and the return

- "As if the requests went to the queue one at a time"



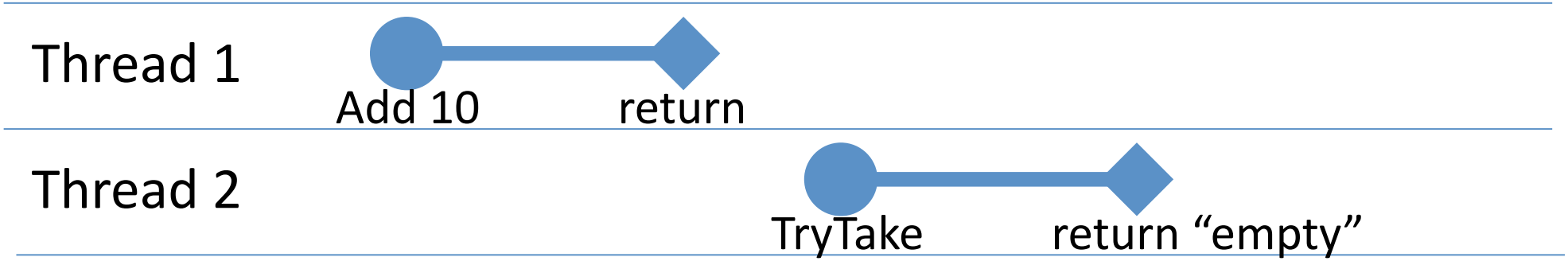
Linearizability vs Seriazliability?

- Serializability
  - All operations (transactions) appear to take effect atomically at a single temporal point

Linearizability vs Seriazliability?

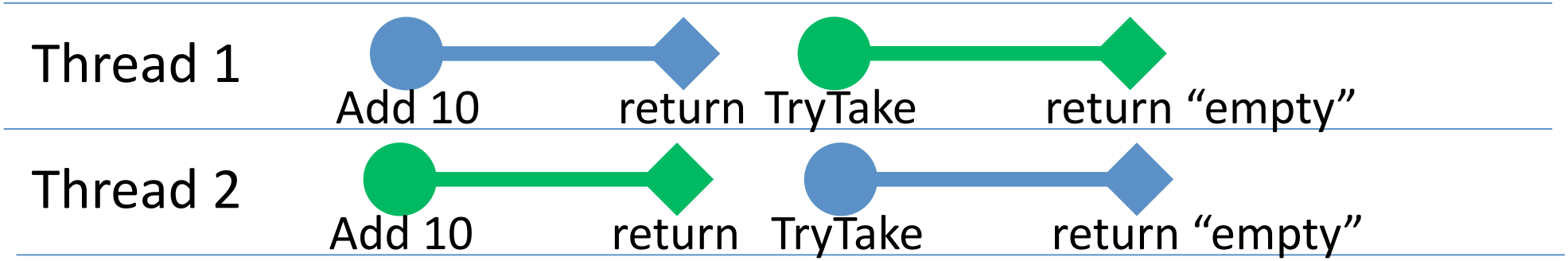
- Serializability
  - All operations (transactions) appear to take effect atomically at a single temporal point
- Linearizability
  - All operations to take effect atomically at a single temporal point
  - That point is between the call and return

Serializable behavior that is not Linearizable



- Linearizability assumes that there is a global observer that can observe that Thread 1 finished before Thread 2 started

Serializability does not compose



- The behavior of the blue queue and green queue are individually serializable
- But, together, the behavior is not serializable

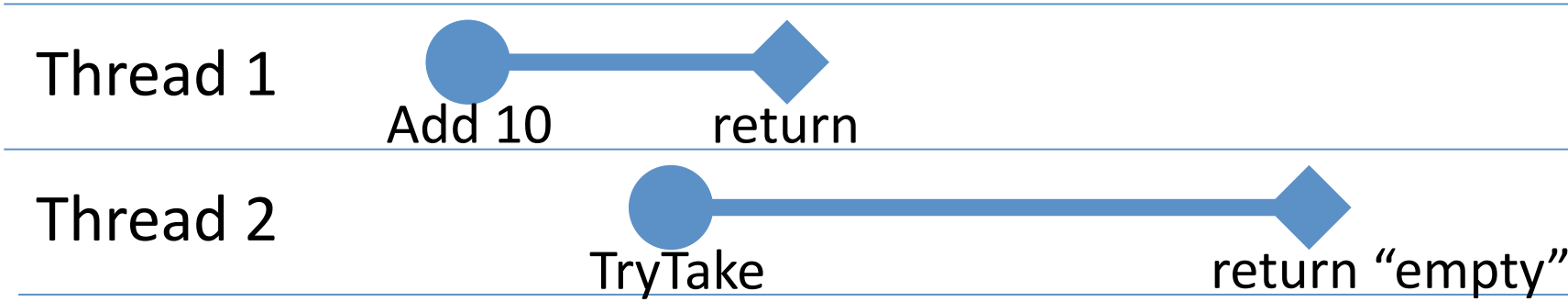
Formalizing Linearizability

- Define the set of observables for each operation
  - Call operation: value of all the arugments
  - Return operation:

- An event:
  - Thread Id, Object Id, Call/Return, Operation, Observables

A Concurrent History

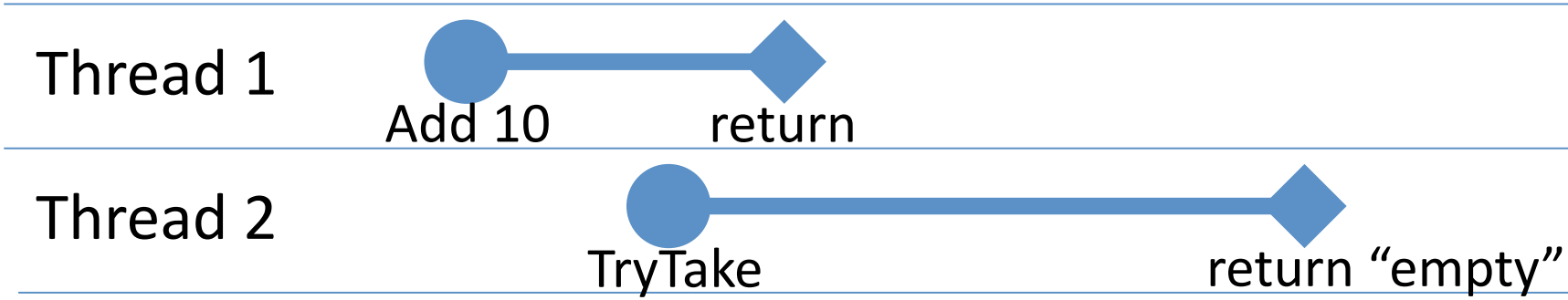
- Sequence of Events
  - <T1, q, Call, Add, 10>
  - <T2, q, Call, TryTake, void>
  - <T1, q, Ret, Add, void>
  - <T2, q, Ret, TryTake, “empty”>



A Concurrent History

- Sequence of Events
  - <T1, q, Call, Add, 10>
  - <T2, q, Call, TryTake, void>
  - <T1, q, Ret, Add, void>
  - <T2, q, Ret, TryTake, “empty”>

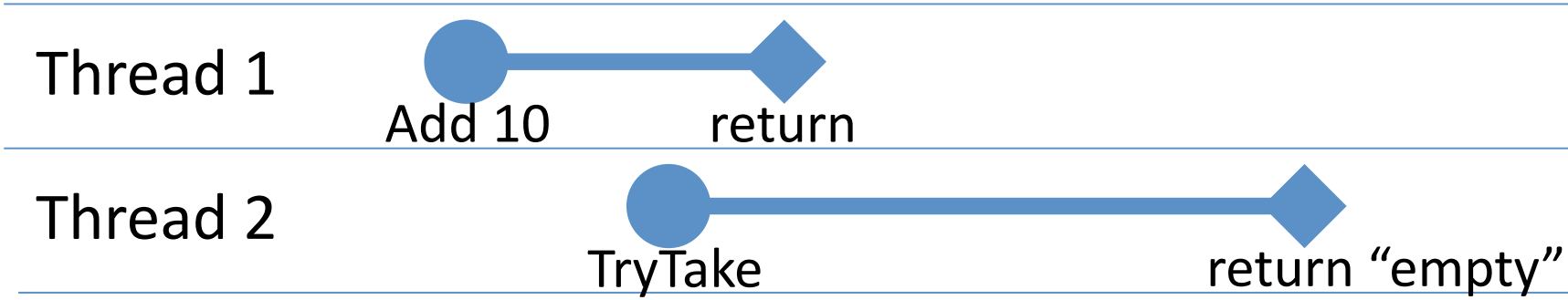
We will only focus on single object histories



A Concurrent History

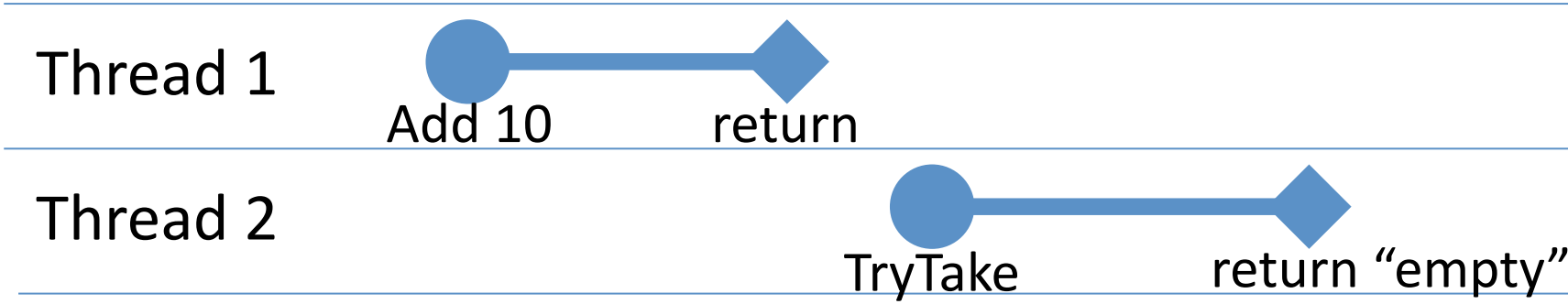
- Sequence of Events
  - <T1, q, Call, Add, 10>
  - <T2, q, Call, TryTake, void>
  - <T1, q, Ret, Add, void>
  - <T2, q, Ret, TryTake, “empty”>

Also, we will only focus on complete histories – every call has a return



A Serial History

- A concurrent history where every call is followed by its matching return
  - <T1, q, Call, Add, 10>
  - <T1, q, Ret, Add, void>
  - <T2, q, Call, TryTake, void>
  - <T2, q, Ret, TryTake, “empty”>

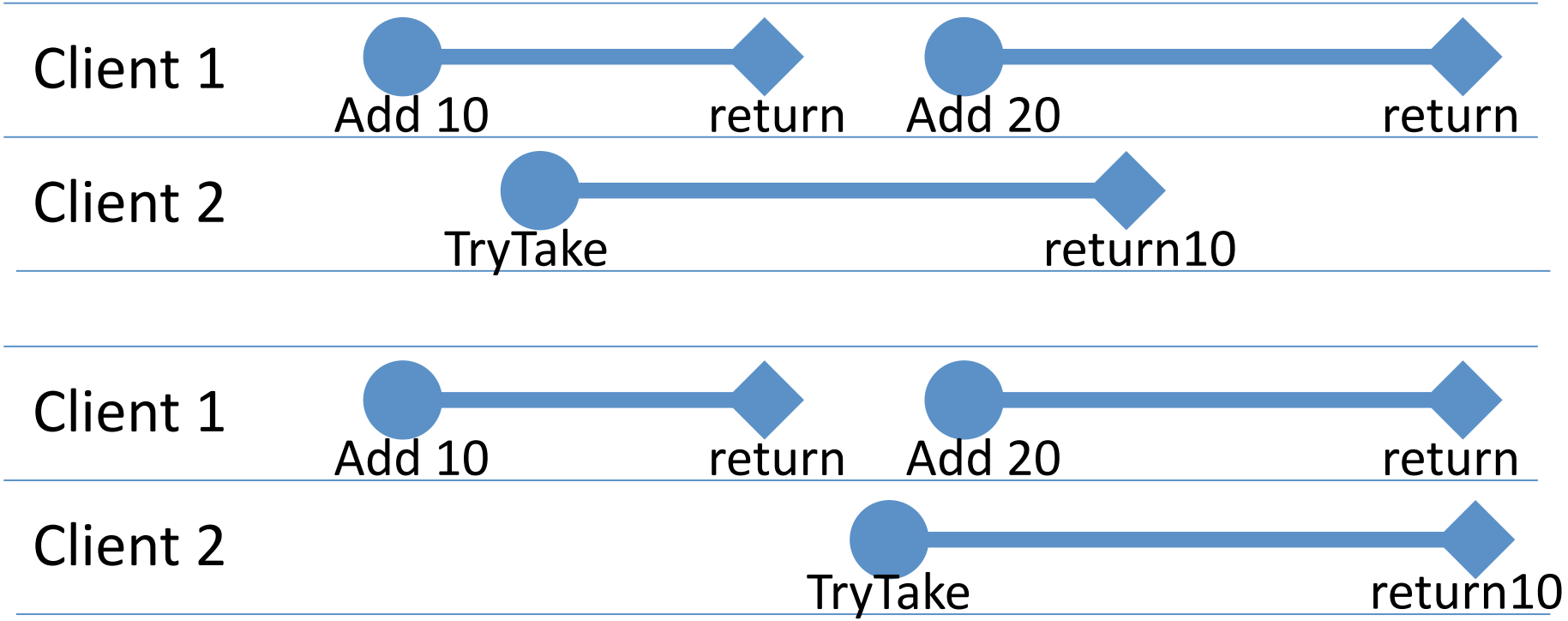


Sequential Specification of an Object

- The set of all serial histories define the sequential behavior of an object
- Assume we have a mechanism to enumerate this set and store the set in a database

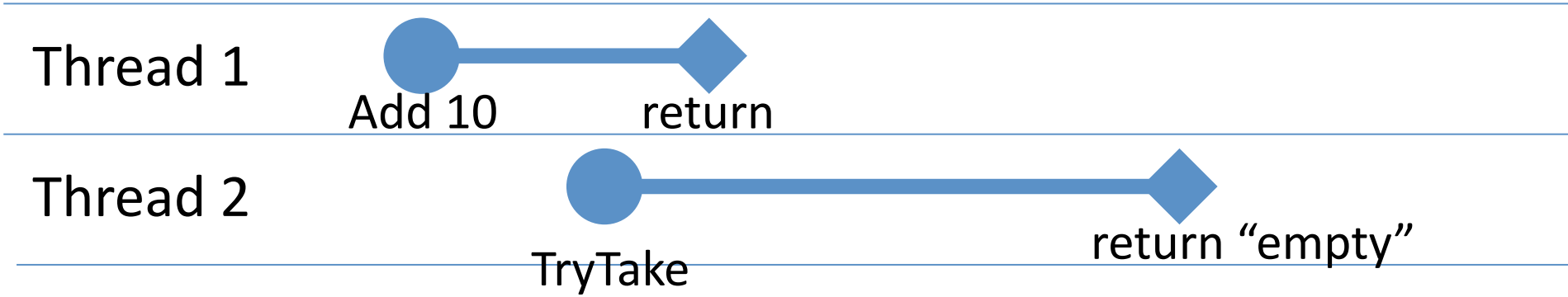
Equivalent Histories

- Two concurrent histories are equivalent if
  - Each thread performs operations in the same order
  - And sees the same observations



Concurrent Operations in a History

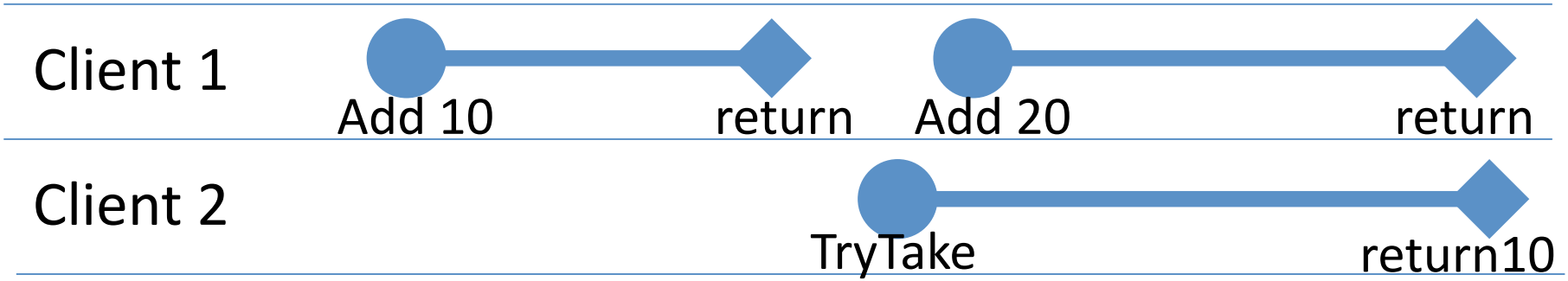
- Two operations p and q are concurrent in a history if their duration overlap
  - ! (p.ret < q.call || q.ret < p.call)



Concurrent Operations in a History

- Two operations p and q are concurrent in a history if their duration overlap
  - ! (p.ret < q.call || q.ret < p.call)

- Non-Concurrent operations define a "performed-before" order



Linearizability

- A concurrent history is linearizable if it is equivalent to a (serial) history in the sequential specification,
- Such that all operations that are “performed before” in the concurrent history are also “performed before” in the serial history