Weird Questions

- A new Thread object must be used each time a runnable code is to be started.
  - true
- UDP message passing may imply space uncoupling.
  - false
- Spatial uncoupling is a property which some indirect communication services provide.
  - true
- Consider a multi-threaded server which is answering to client requests. Describe 3 policies how incoming client requests can be assigned to server threads.
  - thread per request/connection/service
- Message queue systems may implement different queue models
  - true

General

- volatile
  - JVM loads value always from memory instead of local cache
  - reading from cache might cause problems with multiple threads accessing the same variable
- defensive copies
  - if mutable object should only be changed by the native class, the a defensive copy must be made any time it's passed into (constructor, setter) or out of (getter) the class.
  - otherwise caller can break encapsulation, by changing the object
  - prevents mutation from outside
    - mutable object should only be changed within the native class
  - security
    - protects against ill-behaved clients and attacks
- varargs
  - method(int first_arg, int... remaining_args)
- optionals
  - alternative to `null`
  - avoids `null` checking
  - `Optional<Car> optCar = Optional.ofNullable(car);`
  - `maxSpeed = optCar.map(Car::getMaxSpeed);`

FX

- basic elements
  - stage - window
    - first stage - initial window
  - scene graph - tree representing UI elements
  - scene - container
    - may fill stage
    - uses observable collections
- classes
  - Node - JF base class
  - Pane - container base class
  - Control - base class for interactive elements (buttons)
  - Shape - base of vector shapes
- observer
  - implements EventHandler<ActionEvent>
  - may be registered to observable object
  - notified when object triggers an event
- Properties
  - have value
  - observable
    - method to register observers
    - notifies on value change
- Bindings

    - property change affects other properties
    - bidirectional
        * both directions possible
    - unidirectional
        * one way, otherwise exception
- MVP
    - Model
        * logic, data, methods, independent of GUI
    - View
        * GUI creation and manipulation
    - Presenter
        * flow control, handles UI interaction and manipulates model
- FX threads
    - application thread
        * only thread able to access FX elements
            ◆ Platform.runLater()
        * handles all events
        * executes observer methods on event
    - tasks
        * thread executing single long running task
    - service
        * repeating tasks
        * scheduled => schedule when to repeat tasks

8 golden rules

- strive for consistency
- enabled frequent users to use shortcuts
- offer informative feedback
- design dialog to yield closure.
- offer simple error handling.
- permit easy reversal of actions.
- support internal locus of control.
- reduce short-term memory load

Threads

- concurrency/parallelism
    - true
        * multiple CPUs/cores
    - pseudo
        * thread/process switching
    - improves resource sharing, less idle time/busy wait
- distributed system
    - mutliple network components, coordinated via messages
    - server uses threads for concurrent user requests
    - e.g. remote method invocation, remote procedure call
    - server
        * states
            ◆ stateful
                ■ client tables
            ◆ soft state
                ■ for given time
            ◆ session state
                ■ for session
            ◆ stateless
        * listen for/evaluate requests in dispatcher thread
        * delegate to worker threads

        - fixed number/dynamic - depends on creation/deletion costs
        - reusage of threads
            - thread pool
            - list of non-active, waiting threads
            - create all during server startup
        - thread per request/connection/service-object (each service with queue)
- thread
    - run once
        - end if run ends
        - object may still live
    - methods to pause/end/react_to_end
        - join
        - isAlive
        - resultListener with putResult()
        - yield
    - states
        - new => ready
        - ready => running
        - running => blocked/waiting/timed_waiting , terminated
        - blocked/waiting/timed_waiting => ready
    - priority from 1 to 10
        - high if interactive
        - low if computationally intensive
    - jvm has several background threads
        - garbage collection
        - program start => new thread executing main

Synchronization

- synchronized method
    - locks access to object
    - or sleep and wait
- semaphores
    - limit number of threads accessing critical region simultaneously
    - counter to limit parallel access
        - block - `p()`/`down()`
        - unblock - `v()`/`up()`
        - can be applied to sets of semaphores
    - execution sequences with semaphore array
    - reader/writer problem
        - exclusive write, shared read
        - policies
            - prioritize writers
                - finish current reads
                - execute write
            - prioritize read
                - execute reads until no more
                - execute writes
        - semaphore solution
            - `read()/write()` execute `before/really/afterRead/Write()`
            - `before/afterRead/Write` synchronized
- possible problems and solutions

```
// Problem: lost update in case thread switch happens between the read
// and the update of the account, and another thread changes the
// account value in between
public void transferMoney(int accountNumber, float amount)
{
        float oldBalance = account[accountNumber].getBalance();
        float newBalance = oldBalance + amount;
        account[accountNumber].setBalance(newBalance);
}
```

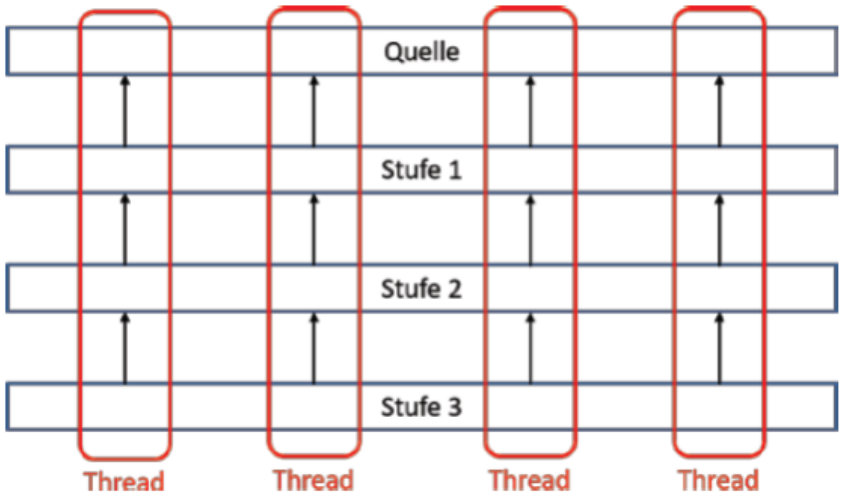
possible thread switch

```
public synchronized void transferMoney(int accountNumber, float amount)
{
        float oldBalance = account[accountNumber].getBalance();
        float newBalance = oldBalance + amount;
        account[accountNumber].setBalance(newBalance);
}
```

```
public void transferMoney(int accountNumber, float amount)
{
        synchronized(account[accountNumber])
        {
                float oldBalance = account[accountNumber].getBalance();
                float newBalance = oldBalance + amount;
                account[accountNumber].setBalance(newBalance);
        }
}
```

Data Stream Processing

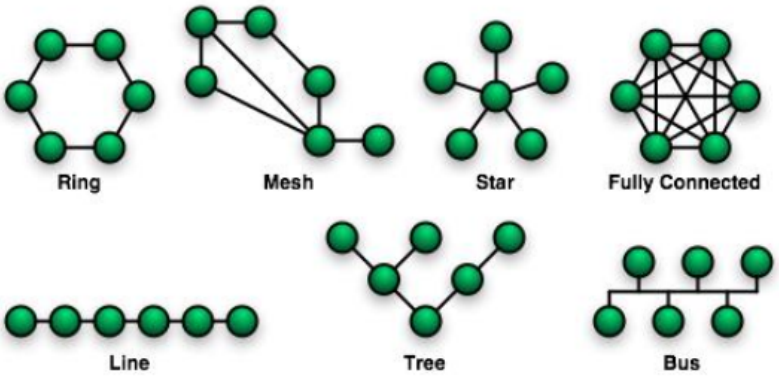
- create stream
  - (e.g. from list)
- process data
  - filter
  - map
  - sum
  - etc
- parallel processing
  - executed in threads parallel for batches of the stream



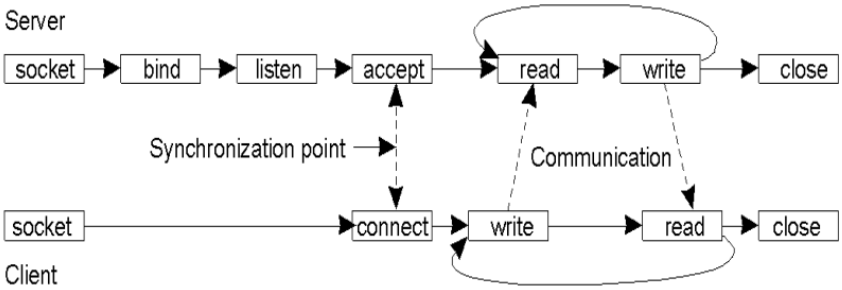
[[**Networking**]]

- network types
  - (W)PAN
    - USB
    - Bluetooth
  - (W)LAN
    - Ethernet

        * WiFi
    - (W)MAN
        * DSL
    - (W)WAN
        * IP-Routing, LTE, 5G
- topologies
    - minimum spanning tree in practice



    -
- TCP
    - `Socket` and `ServerSocket`
        * `getInput/OutputStream()`
        * `accept()`
    - connection oriented
        * reliable
        * handles order
        * loss
        * duplicates
        * flow management
    - bidirectional



    -
    - abstract client/server

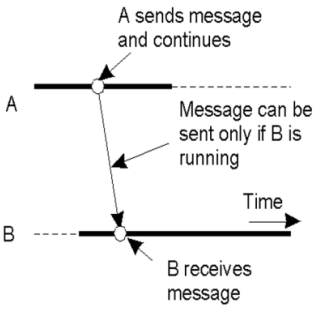
```
// Client side
create a TCP Socket s; /* s gets assigned by the TCP/IP stack an unused port number */
create via s a connection to a target IP address and port number;
/* client blocks until the connection is established */
repeat until done: {
          send a request via s;
          /* note: it is not necessary or possible to specify IP address or port number */
          wait to receive a reply at s;
          process the reply; }
close the connection via s;

// Server side
create a TCP Socket sConnect with specific port number;
repeat: {
          wait for a connection request via sConnect;
          accept the connection; // as a result, a new socket sConnection is created
                                    (having a new port number)
          repeat until connection is closed: {
                    wait until a request via sConnection is received;
                    if request is received: {
                              process the request;
                              create answer message m;
                              send m via sConnection;
                              /* note: it is not necessary or possible to specify IP
                                        address or port number */
                    }
                    else: {
                              /* connection was closed by client */
                              close the connection sConnect;
                              break; // return to outer loop
```

- UDP
  - `DatagramPacket` and `DatagramSocket`
  - connectionless
    - unreliable, short message, use error connection codes
  - unidirectional
  - supports broad/multicasting
    - `MulticastSocket`
  - abstract client/server

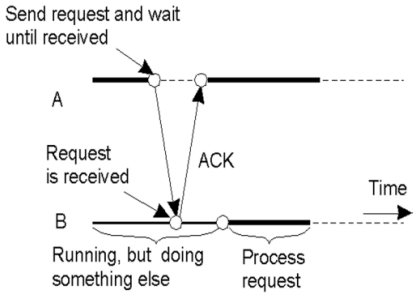
```
// Client side

create a UDP socket s;
/* s gets assigned by the TCP/IP stack an unused port number */
repeat until done:
{
          send via s message m to a destination IP address and port number;
          wait to receive a reply at s;
          process the reply;
}

// Server side

create a UDP socket s with a specific port number;
repeat:
{
          wait for request at s;
          process the request;
          create answer message m;
          send m to IP address and port number of requester;
}
```

Communication

- temporal uncoupling
  - asynchronous
  - sender and receiver must not be running/exist at the same time
- space uncoupling
  - identity of sender must not be known to other side
    - e.g. broadcast, MQTT
  - allow exchange, update or addition of recipients
- transient communication
  - receiver must be currently running
  - request discarded if delivery not possible



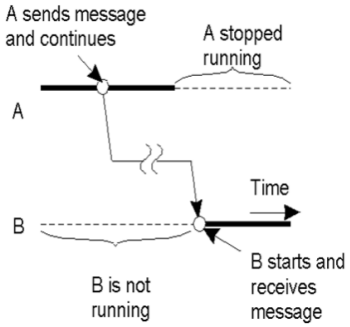
Transient asynchronous
only if receiver is running



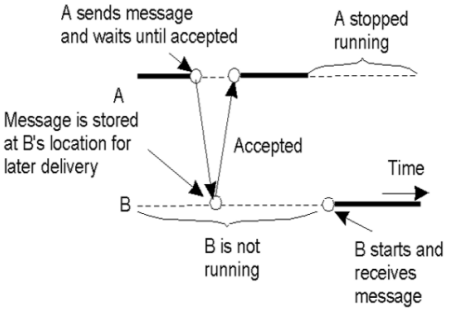
Receiver-oriented transient synchronous
Store on receiver side, sender blocks until acknowledgment of receipt, issued immediately

–

- persistent communication
  - receiver must not be currently runnning
  - message stored on middleware
    - asynchronous
      - near client
    - synchronous
      - near receiver



Persistent asynchronous
store on sender side



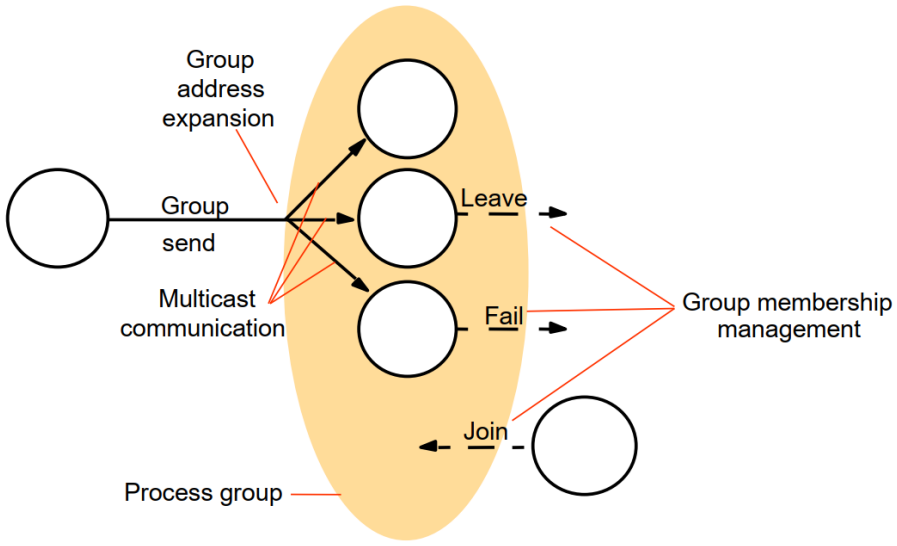
Persistent synchronous
store on receiver side

–

- pull notification
  - receivers requests/polls information
- push notification
  - send notifications on event
  - multicast

Group Communication

- requirements
  - message integrity
  - delivery guarantee
  - receiving order
- add/remove client, update list status

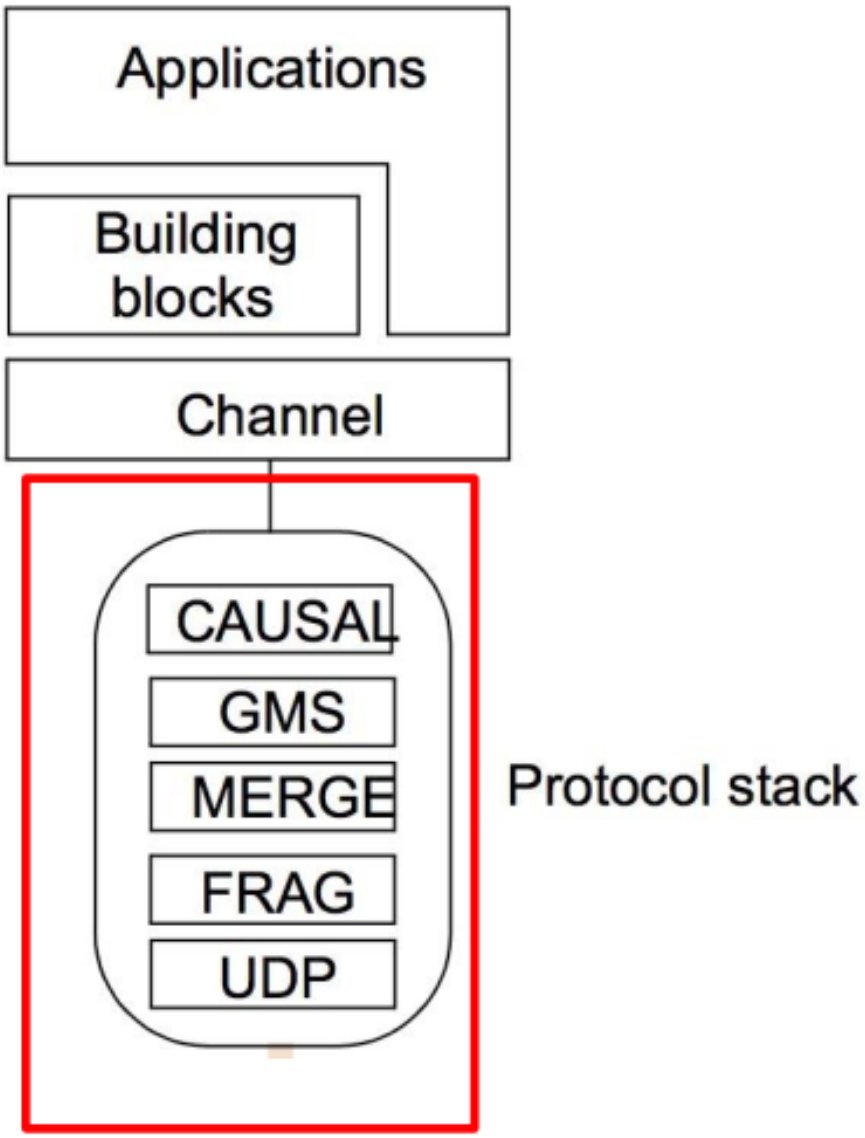


–

- JGroups
  - protocol stack
    - layered
    - UDP (optional TCP)
    - FRAG (fragmentation)
    - MERGE
      - Deal with unexpected network partitioning and subsequent merging of subgroups
    - GMS (group membership protocol)
    - CAUSAL (ordering of messages and answers)
  - channel
    - low level functionality for join/leave/send/receive
      - Handle for threads onto groups
      - Send via reliable multicast
      - Constructor
        - Creates unbound group
        - Connect to existing group, or create a new
      - Operations bind to named groups
        - connect
        - disconnect
        - close
      - Management operations
        - Get member lists and historical information
    -
  - building blocks
    - high level functionality

- MessageDispatcher
  - castMessage() Method to block and await results from callee(s)
- RPCDispatcher
  - Used for remote procedure calling
  - Invoke methods with parameters on all objects associated with a group
  - Caller may block to await results from callee(s)
- NotificationBus
  - Distributed event bus
  - Events as any serializable Java object
  - Used, e.g., for implementation of consistency in replicated caches

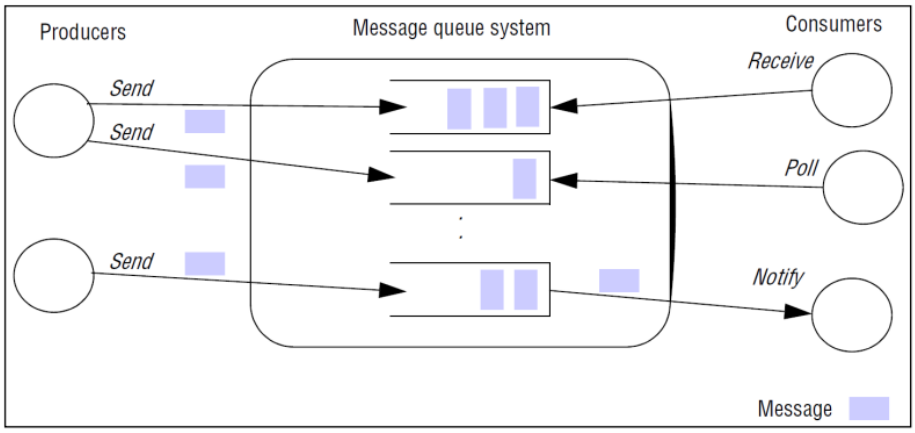
*

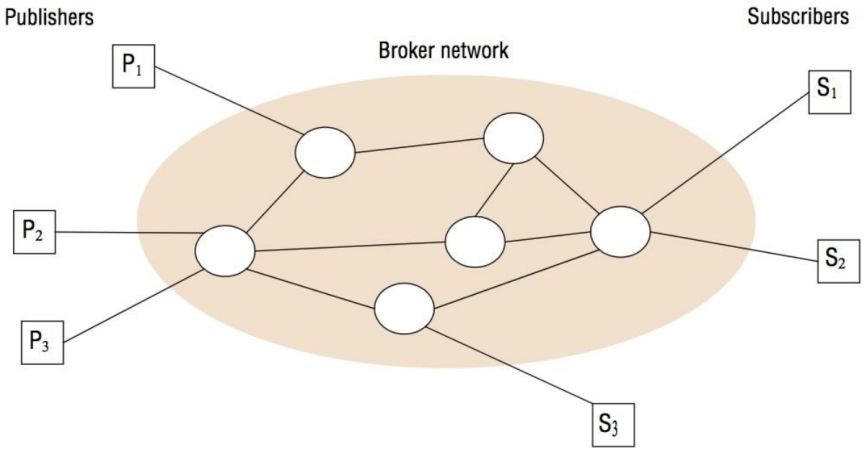


–

– CODE?

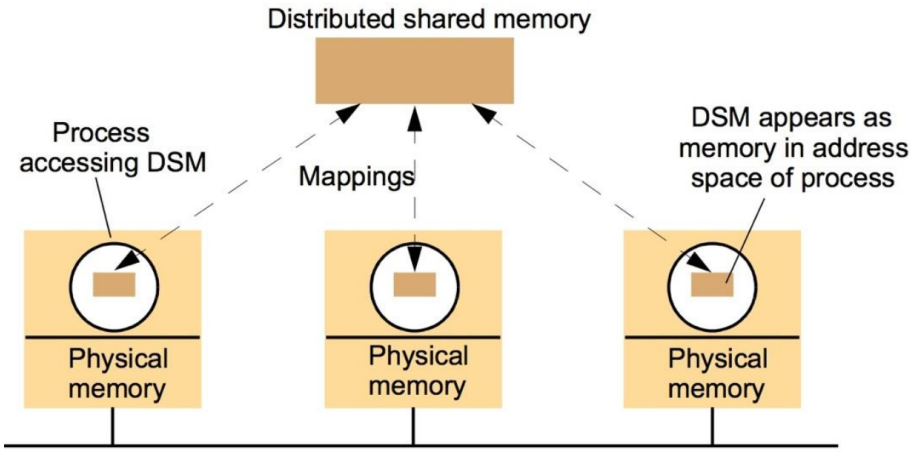
- message queues

  - middleware for persistent message storage
    - potentially SSL, transactional, data transformation
  - decoupling of space and time
  - 

- publish subscribe model
  - publishers create topics and publishes to them
  - subscribers subscribe to topics and are notified/receive
  - participants independent
    - do not know each other
  - broker
    - Assign published events to subscriptions
    - Provide event notifications to subscribers
    - Distribution of brokers in network possible
    - 

  - subscription models
    - channel (topics given by channel)
    - topic (event attribute)
    - content (event attributes)
    - type (topics of object type)
- distributed shared memory = DRAM

- Advantages
  - Avoids explicit messaging
  - No marshalling needed
- Implications
  - Needs memory protection (failing processes)
  - Use locks and semaphores
  - DSM can be made persistent
  - No control over messaging process (ie network costs) by application developer
  - How to select block size?



- Tuple Space
  - Distributed Associative Memory
  - content-addressable memory
  - asynchronous publish-subscribe system

Tuple = Message with ordered and typed attributes

- Tuple(attrib$_1$, attrib$_2$, attrib$_3$, ..., attrib$_N$)

Tuplespace = temporary database of tuples

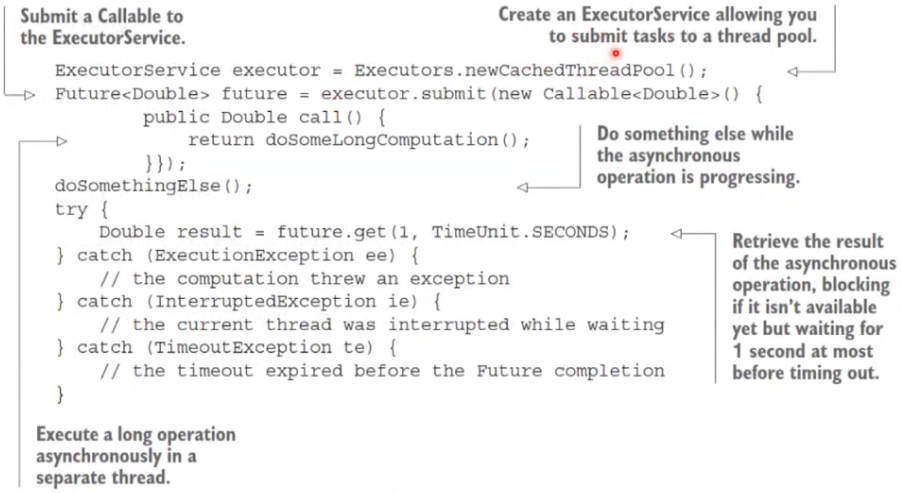
  - implementation
    - Publisher: write(<„this“, „is“, „a“, 5, „tuple“>)
    - Subscriber reads (blocking)
      - Non-desctructive: read; destructive: take
      - Access via pattern matching: read(String, String, „a“, Integer, „tuple“)
    - No direct data access
      - Tuples are replaced, not modified (immutable objects)

*

- Space uncoupling
  - Tuple may be read (written) by any number of clients (publishers)
- Temporal uncoupling
  - Tuple remains in the tuple space until it is taken (persistence)
  - Sender and receiver do not need to exist at the same time
- Variations
  - Central vs. distributed tuple spaces
  - Tuples vs. objects (object spaces)
  - Single server vs. replication

Reactive Software

- goals
  - reponsive
  - resilient
    - responsive despite failure
  - elastic
    - reponsive under varying workload
  - message-driven
    - asynchronous message passing
    - boundary between components
- high-level asynchronous Java APIs
  - `Future`
    - provides reference to result once completed
    - blocked when accessing result before completion
      - `isDone()` before `get()`
      - `get(1, TimeUnit.SECONDS)` throws `TimeoutException`
    - `CompletableFuture`
      - combines several depending futures
      - wait for completion of all tasks/the fastest
      - completion notifications - callbacks
  - `Callable`
    - hosts asynchronous code to execute
  - `ExecutorService`
    - accepts, schedules and runs `Runnable/Callable` objects in threads
    - interface to submits tasks and obtain results later
    - returns future on submit
    - `newFixed/CachedThreadPool()`
    - blocking threads reduce throughput
    - needs to be shutdown
  - code snippets



  *
  *

```
ExecutorService executorService = Executors.newFixedThreadPool(2);
Future<Integer> y = executorService.submit(() -> f(x));
Future<Integer> z = executorService.submit(() -> g(x));
System.out.println(y.get() + z.get());
```

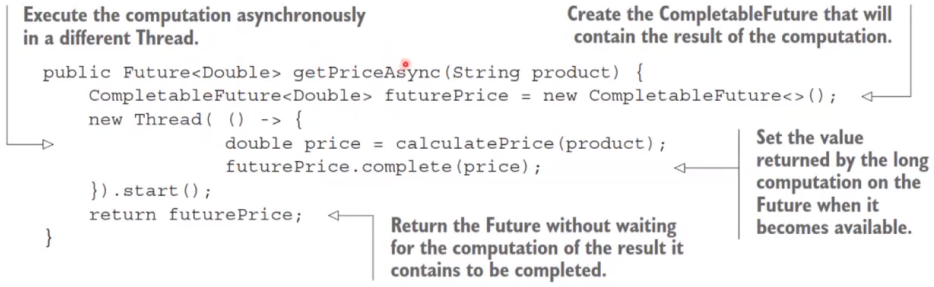
  * future-style

```
// change the signature of f and g to
Future<Integer> f(int x);
Future<Integer> g(int x);

// change the calls to
Future<Integer> y = f(x);
Future<Integer> z = g(x);
System.out.println(y.get() + z.get());
```

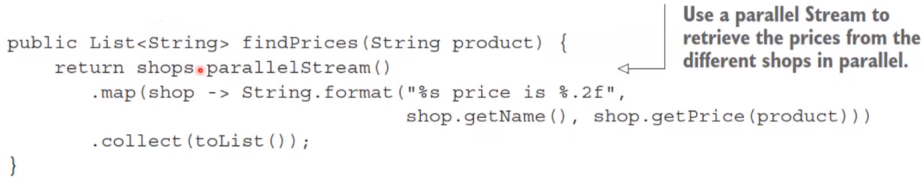
- callback-style

```
// change the signature of f and g to accept callback
void f(int x, IntConsumer dealWithResult);
void g(int x, IntConsumer dealWithResult);


public class CallbackStyleExample {
    public static void main(String[] args) {

        int x = 1337;
        Result result = new Result();

        f(x, (int y) -> {
            result.left = y;
            System.out.println((result.left + result.right));
        } );

        g(x, (int z) -> {
            result.right = z;
            System.out.println((result.left + result.right));
        });
```

- `CompletableFuture`



- `parallelStream`



[[Transaction]] Concepts

- set of operations executed atomically
- provides [[ACID]]

- **A**tomicity
  - Transaction executed all or nothing
- **C**onsistency
  - Transactions transform objects from one consistent state to another
- **I**solation
  - Transactions performed without interference of other transactions
- **D**urability
  - Result of successful transaction is permanently stored

–

- serial equivalence
  - interleaving of transactions/operations is serially equivalent if the combined effect is the same as if performed sequentially
- potential problems
  - lost update

Transaction *T*:		Transaction *U*:	
balance = b.getBalance(); *b.setBalance(balance*1.1);* *a.withdraw(balance/10)*		*balance = b.getBalance();* *b.setBalance(balance*1.1);* *c.withdraw(balance/10)*	
balance = b.getBalance();	$200		
		balance = b.getBalance();	$200
		*b.setBalance(balance*1.1);*	$220
*b.setBalance(balance*1.1);*	$220		
a.withdraw(balance/10)	$80		
		c.withdraw(balance/10)	$280

A=100, B=200, C=300.
Increase B by 10% from each A and C.

* T overwrites U's update. Correct result should be B=242 not B=220.

A serially equivalent interleaving of *T* and *U*

Transaction *T*:		Transaction *U*:	
balance = b.getBalance() *b.setBalance(balance*1.1)* *a.withdraw(balance/10)*		*balance = b.getBalance()* *b.setBalance(balance*1.1)* *c.withdraw(balance/10)*	
balance = b.getBalance()	$200		
*b.setBalance(balance*1.1)*	$220		
		balance = b.getBalance()	$220
		*b.setBalance(balance*1.1)*	$242
a.withdraw(balance/10)	$80		
		c.withdraw(balance/10)	$278

* U reads the new value of `b`. T does all its operations on `b` before U

  – inconsistent retreival

Transaction *V*:		Transaction *W*:	
a.withdraw(100) *b.deposit(100)*		*aBranch.branchTotal()*	
a.withdraw(100);	$100		
		total = a.getBalance()	$100
		total = total+b.getBalance()	$300
		total = total+c.getBalance()	
b.deposit(100)	$300	⋮	

A=B=200. Inconsistency as W has read an intermediate state of the transfer (V has only performed withdrawal part not the deposit part.

A serially equivalent interleaving of *V* and *W*

Transaction *V*:		Transaction *W*:	
a.withdraw(100); *b.deposit(100)*		*aBranch.branchTotal()*	
a.withdraw(100);	$100		
b.deposit(100)	$300		
		total = a.getBalance()	$100
		total = total+b.getBalance()	$400
		total = total+c.getBalance()	
		...	

- dirty reads

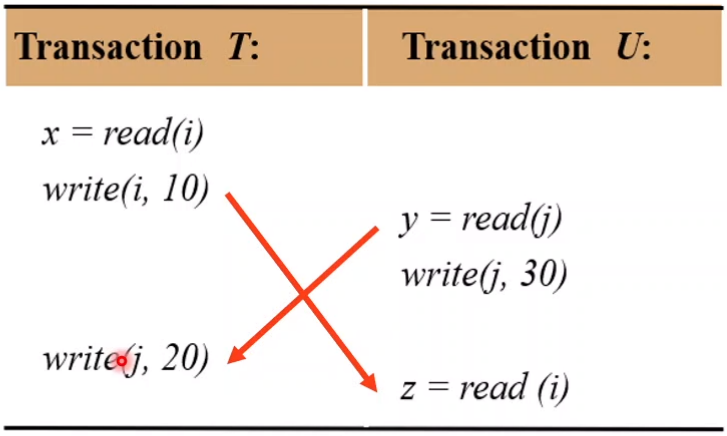
A dirty read when transaction *T* aborts

Transaction *T*:		Transaction *U*:	
a.getBalance() *a.setBalance(balance + 10)*		*a.getBalance()* *a.setBalance(balance + 20)*	
balance = a.getBalance()	$100		
a.setBalance(balance + 10)	$110		
		balance = a.getBalance()	$110
		a.setBalance(balance + 20)	$130
		commit transaction	
abort transaction			

U has read a value which was not committed.
Problem: U has committed and cannot be undone (durability, atomicity).

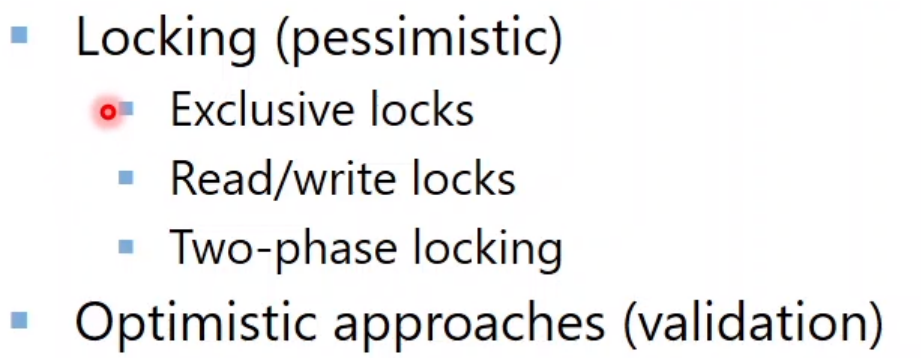
- compatibility

Operations of different transactions		*Conflict*	*Reason*
read	*read*	No	Because the effect of a pair of *read* operations does not depend on the order in which they are executed
read	*write*	Yes	Because the effect of a *read* and a *write* operation depends on the order of their execution
write	*write*	Yes	Because the effect of a pair of *write* operations depends on the order of their execution



Observation:
T is serially before U regarding i: T <(i) U
T is serially after U regarding j: T >(j) U

- locking



Transactions *T* and *U* with exclusive locks

Transaction *T*:		**Transaction *U*:**	
balance = b.getBalance() *b.setBalance(bal*1.1)* *a.withdraw(bal/10)*		*balance = b.getBalance()* *b.setBalance(bal*1.1)* *c.withdraw(bal/10)*	
Operations	Locks	Operations	Locks
openTransaction			
bal = b.getBalance()	lock *B*		
*b.setBalance(bal*1.1)*		*openTransaction*	
a.withdraw(bal/10)	lock *A*	*bal = b.getBalance()*	waits for *T*'s lock on *B*
closeTransaction	unlock *A*, *B*	• • •	
			lock *B*
		*b.setBalance(bal*1.1)*	
		c.withdraw(bal/10)	lock *C*
		closeTransaction	unlock *B*, *C*

For one object		*Lock requested*	
		read	*write*
Lock already set	*none*	OK	OK
	read	OK	wait
	write	wait	wait

- two-phase locking
  1. When an operation accesses an object within a transaction:
     (a) If the object is not already locked, it is locked and the operation proceeds.
     (b) If the object has a conflicting lock set by another transaction, the transaction must wait until it is unlocked.
     (c) If the object has a non-conflicting lock set by another transaction, the lock is shared and the operation proceeds.
     (d) If the object has already been locked in the same transaction, the lock will be promoted if necessary and the operation proceeds. (Where promotion is prevented by a conflicting lock, rule (b) is used.)
  2. When a transaction is committed or aborted, the server unlocks all objects it locked for the transaction.
- deadlock detection
  - Maintain wait-for-graph
  - Test for cycles on each operation, or periodically
  - On detection, select transaction(s) to abort. Criteria e.g.,
    - age of transaction
    - number of cycles broken
    - assigned prioritization
    - etc.

- Optimistic Concurrency Control
  - Assume conflicts are sparse
  - Working phase, validation phase, update phase
- Timestamp Ordering
  - Validate each operation on execution based on time stamps
  - Abort executing transaction on conflict
- Nested Transactions
  - Substructured Transactions
- Distributed transactions
- Replication and cache consistency