Create a connection pool implementation

#1 Define all the interfaces

- Connection
- ConnectionFactory
- ObjectPool
- ConnectionPool

interface

```
public Connection<T> extends Supplier<T>, Closeable {
    @Override T get();
    boolean isValid();
    void close();
}
```

```
public interface ConnectionFactory<T extends Connection<?>> {
    boolean shouldCreateConnection();
    T create();          → // T create(long timeoutInMillis);
    void destroy(T connection);
}
```

```
public interface ObjectPool<T> {
    T get() throws ResourceExhaustedException, TimeoutException;
    T get(long millis) throws ____________;
    void release(T resource);
    void remove(T resource);
    void close();
}
```

```
public interface StatsBorrider {
    AtomicLong makeCounter(String name);
    - guage.
}
```

```
public final class ConnectionPool<S extends Connection<?>> implements ObjectPool<S>{
    private final Set<S> leasedConnections = Collections.newSetFromMap(new ConcurrentHashMap<>());
    private final Set<S> availableConnections = ____;
    private final Lock poolLock;
    private final Condition available;
    private final ConnectionFactory<S> connectionFactory;
    private final AtomicLong connectionsCreated;
    private final AtomicLong connectionsReturned;
    ____ connectionsDestroyed;

    private volatile boolean close;
    private final StatsProvider statsProvider;

public ConnectionPool(ConnectionFactory<S> connectionFactory){
    this(connectionFactory, DefaultStatsProvider.getInstance());
}

public ConnectionPool(ConnectionFactory<S> connectionFactory, StatsProvider statsProvider){
    checkNotNull(connectionFactory);
    checkNotNull(statsProvider);
    this.connectionFactory = connectionFactory;
    this.statsProvider = statsProvider;
    this.poolLock = new ReEntrantLock(true);
    this.available = this.poolLock.newCondition();
    this.connectionsCreated = statsProvider.makeCounter("cp-conn-created");
    this.connectionsReturned = ____;
    this.connectionsDestroyed = ____;
}

private void checkNotClosed(){                    <-- Important
    if(closed){
        throw new IllegalStateException(____);
    }
}
```

```
@Override
public S get() throws ResourceExhaustedException, TimeoutException {
    checkNotClosed();
    poolLock.lock();
    try {
        return leaseConnection(NO_TIMEOUT);
    } finally {
        poolLock.unlock();
    }
}

private S leaseConnection(long timeoutInMillis) throws ——— {
    S connection = getConnection(timeoutInMillis);
    if (connection == null) {
        throw new ResourceExhaustedException("———");
    }
    return connection.leaseConnection(connection);
}

private S leaseConnection(S connection) {
    leasedConnections.add(connection);
    return connection;
}

private S getConnection(final long timeoutInMillis) throws ——— {
    if (availableConnections.isEmpty()) {
        if (leasedConnections.isEmpty()) {
            try {
                return createConnection(timeout);
            } catch (Exception e) {
                throw new ConnectionPoolException("Unable to create connection");
            }
        } else {
            if (ConnectionFactory.shouldCreateConnection()) {
                try {
                    Connection connection = createConnection(timeout);
                    if (connection != null) {
                        addConnection(connection);
                    }
                } catch (Exception ex) {
                }
            }
        }
    }
}
```

if no connection has been created yet

```
// manage timeout based retrieval from pool.
try {
    if (timeout == 0) {
        while (availableConnections.isEmpty()) {
            available.await();
        }
    } else {
        long timeRemaining = getInMillis(timeoutInMillis);
        while (availableConnections.isEmpty()) {
            long start = System.getCurrentTimeMillis();
            if (!available.await(timeRemaining, TimeUnit.MILLISECONDS)) {
                throw new TimeoutException(——)
            } else {
                timeRemaining -= (System.currentTimeInMillis - start);
            }
        }
        if (availableConnections.isEmpty()) {
            throw new TimeoutException(——)
        }
    }
} catch (InterruptedException ex) {
    ——
}
return getAvailableConnection();
}

private S getAvailableConnection() {
    S start = availableConnections.iterator().next();
    return
}

private S createConnection(long timeout) {
    S conn = connectionFactory.createConnection(timeout);
    if (conn != null) {
        connectionsCreated.incrementAndGet();
    }
    return conn;
}
```

```
private void addConnection (S connection) {
        availableConnections.add (connection);
        available.signal();
}
```

```
@Override
public void release (S connection) {
    release (connection, false);
}
```

```
@override
public void remove (S Connection) {
    release (connection, true);
}
```

```
private void release ( Connection con, boolean remove) {
    checkNotClosed();
    poolLock.lock();
    try {
        if ( ! leasedConnection.remove (con))
            throw excep
        }
        if ( ! remove && conn.isValid()) {
            addConnection (conn);
            connectionReturned.incrementAndGet();
        } else {
            connectionFactory.destroy( connection);
            connectionDestroyed.incrementAndGet();
        }
    } finally {
        poolLock.unlock();
    }
}
```

```
@override
public void close() {
    poolLock.lock();
    try {
        for (Connection : availableConnections) {
            ConnectionFactory.destroy(connection);
        }
    }
    finally {
        closed = true;
        poolLock.unlock();
    }
}
```