

# Chapter-5: Building Blocks

Upcode Software  
Engineer Team



# CONTENT

1. Synchronized vs Asynchronized
2. Synchronized collection
3. Concurrent modification collection
4. Producer-consumer design
5. The Thread Pool
6. Conculustion
7. Reference



## Synchronized vs Asynchronized

1. This chapter covers the most useful concurrent building blocks, especially those introduced in Java 5.0 and Java 6, and some patterns for using them to structure concurrent applications.
2. The synchronized collections are thread-safe

	Duplicates Allowed	Elements Ordered	Elements Sorted	Synchronized
ArrayList	YES	YES	NO	NO
LinkedList	YES	YES	NO	NO
Vector	YES	YES	NO	YES
HashSet	NO	NO	NO	NO
LinkedHashSet	NO	YES	NO	NO
TreeSet	NO	YES	YES	NO
HashMap	NO	NO	NO	NO
LinkedHashMap	NO	YES	NO	NO
HashTable	NO	NO	NO	YES
TreeMap	NO	YES	YES	NO

# Synchronized vs Asynchronized?

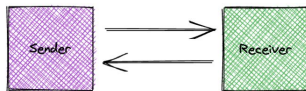
- Synchronous tasks happen in order—you must complete the current task before moving on to the next.
- Asynchronous tasks are executed in any order or even at once.
- Just let existing thread-safe classes manage all the state

## Sync vs Async Communication

Request and wait for a response  
Fire and forget with messages/events

@boyney123

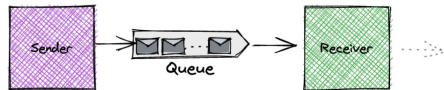
### Synchronous



Request-response model

Send a request and wait for some response  
Example: API requests

### Asynchronous

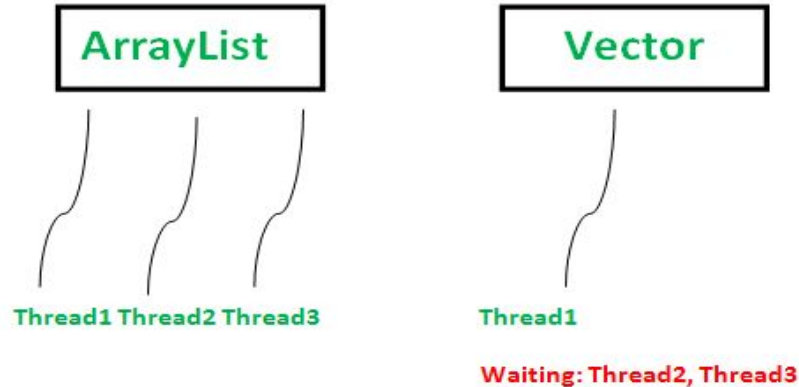


Fire and forget model

Send message/event and forget  
Example: Event Driven Architecture

## Synchronized collection

1. The synchronized collection classes include **Vector** and **Hashtable**, part of the original JDK, as well as their cousins added in JDK 1.2, the synchronized **wrapper classes** created by the **Collections.synchronizedXxx** factory methods.
2. These classes achieve thread safety by encapsulating their state and synchronizing every public method so that only one thread at a time can access the collection state





## Problems with synchronized collection

1. Compound actions on a Vector that may produce confusing results.

```
public static Object getLast(Vector list) {  
    int lastIndex = list.size() - 1;  
    return list.get(lastIndex);  
}
```



```
public static void deleteLast(Vector list) {  
    int lastIndex = list.size() - 1;  
    list.remove(lastIndex);  
}
```

2. If thread A calls `getLast` on a Vector with ten elements, thread B calls `deleteLast` on the same Vector, and the operations are interleaved as shown in Figure 5.1, `getLast` throws `ArrayIndexOutOfBoundsException`



## Solve with synchronized method

1. Compound actions on Vector using client-side locking.

```
public static Object getLast(Vector list) {  
    synchronized (list) {  
        int lastIndex = list.size() - 1;  
        return list.get(lastIndex);  
    }  
}  
  
public static void deleteLast(Vector list) {  
    synchronized (list) {  
        int lastIndex = list.size() - 1;  
        list.remove(lastIndex);  
    }  
}
```



## Problems with synchronized collection

Iteration that may throw `ArrayIndexOutOfBoundsException`

---

```
for (int i = 0; i < vector.size(); i++)  
    doSomething(vector.get(i));
```

---





## Solve with synchronized method

1. Iteration with client-side locking
2. The problem of unreliable iteration can again be addressed by client-side locking

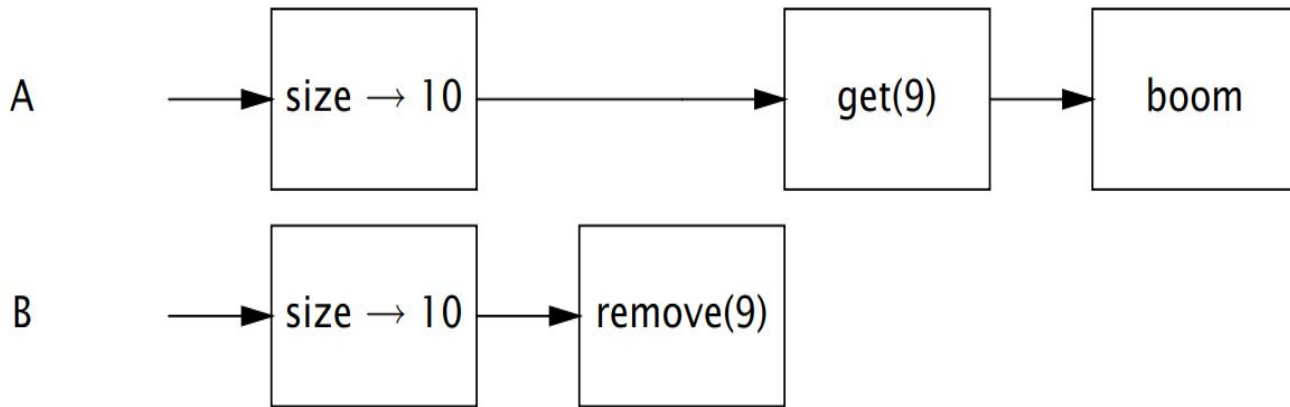
```
synchronized (vector) {  
    for (int i = 0; i < vector.size(); i++)  
        doSomething(vector.get(i));  
}
```

3. we prevent other threads from modifying the Vector while we are iterating it.
4. Unfortunately, we also prevent other threads from accessing it at all during this time, impairing concurrency



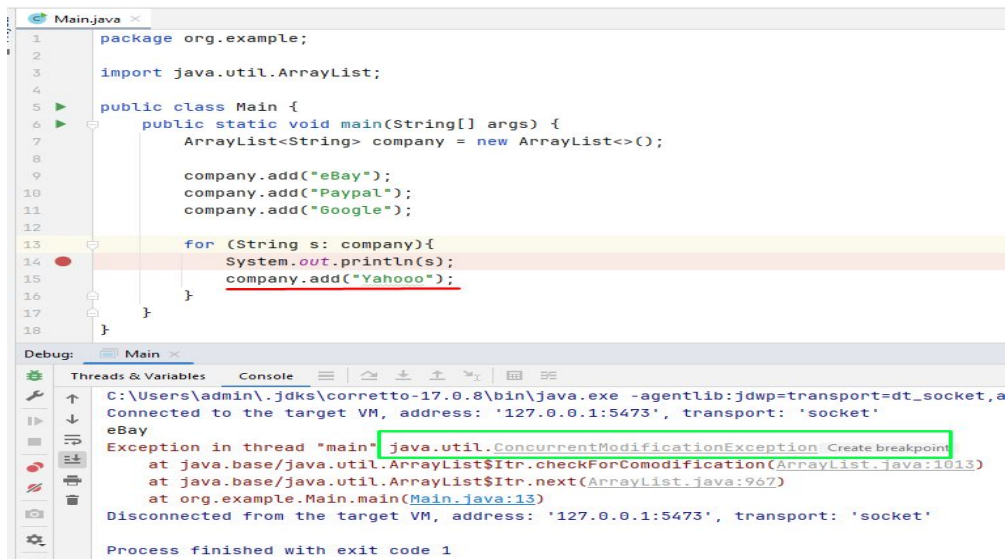
## Problems with synchronized collection

1. The synchronized collection classes guard each method with the lock on the synchronized collection object itself
  2. Interleaving of `getLast` and `deleteLast` that throws `ArrayIndexOutOfBoundsException`.
- 



# Problems with synchronized collection

1. The iterators returned by the synchronized collections are not designed to deal with concurrent modification, and they are fail-fast—meaning that if they detect that the collection has changed since iteration began, they throw the unchecked `ConcurrentModificationException`.



The screenshot shows an IDE with a Java file named `Main.java`. The code defines a `Main` class with a `main` method. Inside `main`, an `ArrayList` named `company` is created and populated with "eBay", "Paypal", and "Google". A `for` loop iterates over the `company` list. Inside the loop, the first iteration prints "eBay" and then adds "Yahooo" to the list. This concurrent modification causes a `ConcurrentModificationException` to be thrown. The exception is visible in the console output at the bottom of the IDE. The exception message is: `Exception in thread "main" java.util.ConcurrentModificationException Create breakpoint`. The stack trace shows the exception was thrown at `java.base/java.util.ArrayList$Itr.checkForComodification(ArrayList.java:1013)`, `at java.base/java.util.ArrayList$Itr.next(ArrayList.java:967)`, and `at org.example.Main.main(Main.java:13)`. The console also shows the command used to run the program: `C:\Users\admin\jdk\corretto-17.0.8\bin\java.exe -agentlib:jdwp=transport=dt_socket, address='127.0.0.1:5473', transport: 'socket'`. The process finished with exit code 1.

```
1 package org.example;
2
3 import java.util.ArrayList;
4
5 public class Main {
6     public static void main(String[] args) {
7         ArrayList<String> company = new ArrayList<>();
8
9         company.add("eBay");
10        company.add("Paypal");
11        company.add("Google");
12
13        for (String s: company){
14            System.out.println(s);
15            company.add("Yahooo");
16        }
17    }
18 }
```

Debug: Main

Threads & Variables Console

C:\Users\admin\jdk\corretto-17.0.8\bin\java.exe -agentlib:jdwp=transport=dt\_socket, address: '127.0.0.1:5473', transport: 'socket'

eBay

Exception in thread "main" java.util.ConcurrentModificationException Create breakpoint

at java.base/java.util.ArrayList\$Itr.checkForComodification(ArrayList.java:1013)

at java.base/java.util.ArrayList\$Itr.next(ArrayList.java:967)

at org.example.Main.main(Main.java:13)

Disconnected from the target VM, address: '127.0.0.1:5473', transport: 'socket'

Process finished with exit code 1

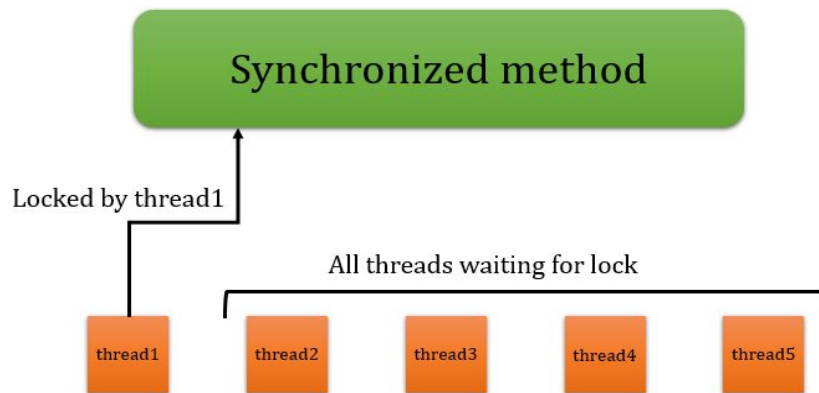


## Synchronized collection

1. If many threads are blocked waiting for a lock throughput and CPU utilization can suffer(see chapter 11)
2. An alternative to locking the collection during iteration is to clone the collection and iterate the copy instead

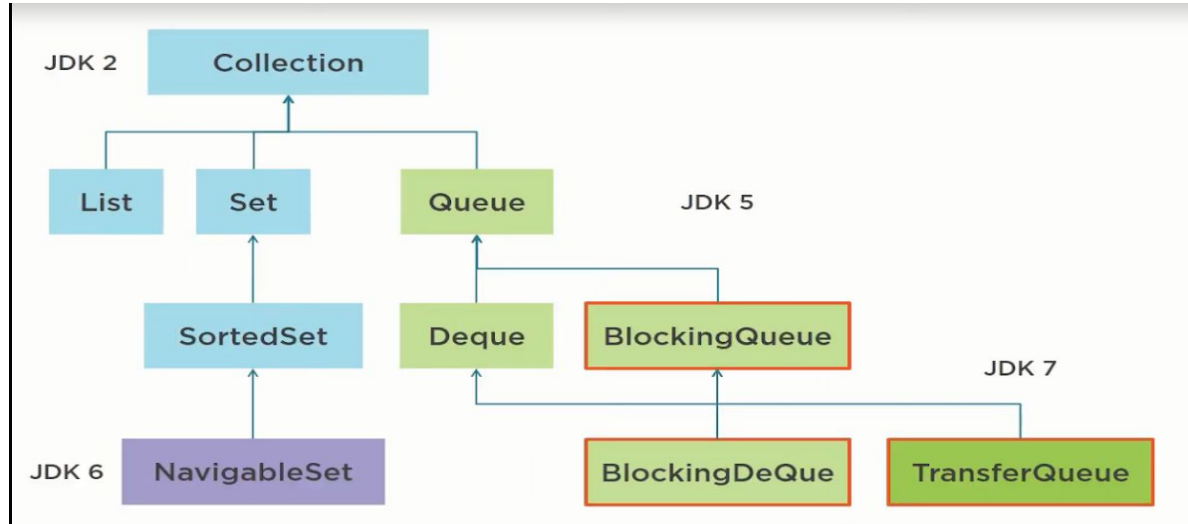
## Hidden iterators

1. Just as encapsulating an object's state makes it easier to preserve its invariants, encapsulating its synchronization makes it easier to enforce its synchronization policy.



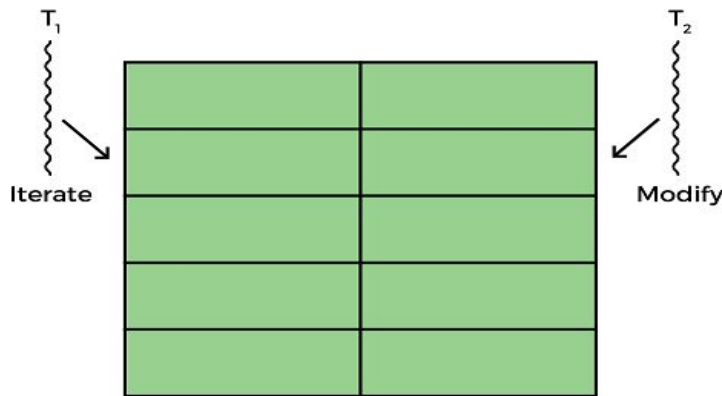
# Concurrent collection

1. Concurrent collection in Java refers to a set of classes that allow multiple threads to access and modify a collection concurrently, without the need for explicit synchronization. These collections are part of the `java.util`.



# Concurrent collection

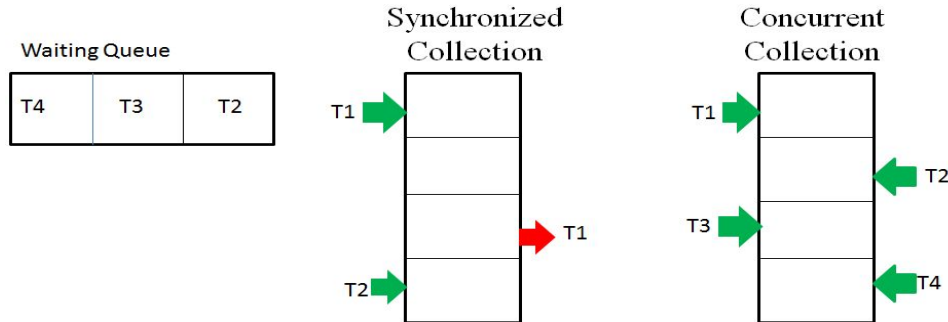
1. The iterators returned by the synchronized collections are not designed to deal with concurrent modification, and they are fail-fast
2. If they detect that the collection has changed since iteration began, they throw the unchecked `ConcurrentModificationException`.



Concurrent Modification Exception

# Concurrent collection

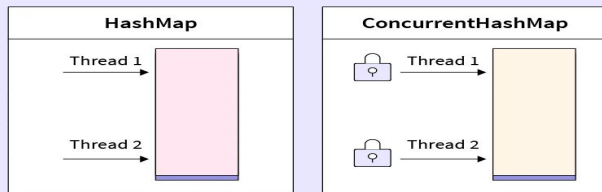
1. **Java 5.0** also adds **two** new collection type
  - a. Queue
  - b. BlockingQueue
2. Queue operations do not block; if the queue is empty, the retrieval operation returns null
3. The synchronized collections classes hold a lock for the duration of each operation.





# ConcurrentHashMap

- ❖ How it made possible to make ConcurrentHashMap thread-safe?
  - ConcurrentHashMap the read operation doesn't require any lock.
  - ConcurrentHashMap class achieves thread-safety by dividing the map into segments, the lock is required not for the entire object but for one segment, i.e one thread requires a lock of one segment.
- ❖ ConcurrentHashMap is in java.util.Concurrent package.





## ConcurrentHashMap

- ❖ ConcurrentHashMap is a hash table supporting full concurrency of retrievals and high expected concurrency for updates.
- ❖ This class obeys the same functional specifications as Hashtable and includes all methods of Hashtable.
- ❖ ConcurrentHashMap is in java.util.Concurrent package.

```
1 public class ConcurrentHashMap<K,V>
2 extends AbstractMap<K,V>
3 implements ConcurrentMap<K,V>, Serializable
```

# HashMap

```
1 package org.example.JAVA_Concurrency_in_Practice.example1;
2
3 import java.util.HashMap;
4 import java.util.Iterator;
5 import java.util.Set;
6
7 public class GFG extends Thread {
8     static HashMap m = new HashMap();
9
10    @Override
11    public void run() {
12        try {
13            Thread.sleep(2000);
14        } catch (InterruptedException e) {}
15    }
16    System.out.println("Child Thread updating Map");
17    //1
18    m.put(103, "C");
19    //2
20    m.put(103, "C");
21 //
22 }
23
24 public static void main(String[] args) throws InterruptedException {
25     m.put(101, "A");
26     m.put(102, "B");
27
28     GFG t = new GFG();
29
30     t.start();
31
32     Set set = m.keySet();
33
34     Iterator iterator = set.iterator();
35
36     while (iterator.hasNext()) {
37         System.out.println("2-example");
38         Integer i1 = (Integer) iterator.next();
39
40         System.out.println("Main Thread Iterating Map and Current Entry is:" + i1 + "..." +
41 m.get(i1));
42
43         Thread.sleep(3000);
44     }
45     System.out.println(m);
46 }
47 }
```

Run: GFG x

C:\Users\admin\jdk\corretto-17.0.8\bin\java.exe ...

1-example

Main Thread Iterating Map and Current Entry is:101...A

Child Thread updating Map

1-example

Exception in thread "main" java.util.ConcurrentModificationException Create breakpoint  
at java.base/java.util.HashMap\$HashIterator.nextNode(HashMap.java:1597)  
at java.base/java.util.HashMap\$KeyIterator.next(HashMap.java:1620)  
at org.example.JAVA\_Concurrency\_in\_Practice.example1.GFG.main(GFG.java:38)

Process finished with exit code 1

Run: GFG x

C:\Users\admin\jdk\corretto-17.0.8\bin\java.exe ...

2-example

Main Thread Iterating Map and Current Entry is:101...A

Child Thread updating Map

2-example

Main Thread Iterating Map and Current Entry is:102...B

{101=A, 102=B}

Process finished with exit code 0

# ConcurrentHashMap

```
1 import java.util.Iterator;
2 import java.util.Set;
3 import java.util.concurrent.ConcurrentHashMap;
4
5 public class Main extends Thread {
6     static ConcurrentHashMap<Integer, String> m = new ConcurrentHashMap<>();
7     // Method 1
8     // run() method for the thread
9     @Override
10    public void run()
11    {
12        // Try block to check for exceptions
13        try {
14            // Making thread to sleep for 2 seconds
15            Thread.sleep(2000);
16        }
17        // Catch block to handle the exceptions
18        catch (InterruptedException e) {
19        }
20        System.out.println("Child Thread updating Map");
21        // Inserting element
22        m.put(103, "C");
23    }
24
25    // Method 2
26    // Main driver method
27    public static void main(String arg[])
28        throws InterruptedException
29    {
30        // Adding elements to object created of Map
31        m.put(101, "A");
32        m.put(102, "B");
33        // Creating thread inside main() method
34        Main t = new Main();
35        // Starting thread
36        t.start();
37        // Creating object of Set class
38        Set<Integer> s1 = m.keySet();
39        // Creating iterator for traversal
40        Iterator<Integer> itr = s1.iterator();
41        // Condition holds true till there is single element
42        // in Set object
43        while (itr.hasNext()) {
44            // Iterating over elements
45            // using next() method
46            Integer i1 = itr.next();
47
48            // Display message
49            System.out.println("Main Thread Iterating Map and Current Entry is:" + i1 + "...");
50            // Making thread to sleep for 3 seconds
51            Thread.sleep(3000);
52        }
53        // Display elements of map objects
54        System.out.println(m);
55    }
56 }
57 }
58 }
```

Let us see control flow, so as we know that in ConcurrentHashMap while one thread is iterating the remaining threads are allowed to perform any modification in a safe manner. In the above program Main thread is updating Map, at the same time child thread is also trying to update the Map object

```
Run: org.example.JAVA_Concurrency_in_Practice.example2.Main x
C:\Users\admin\.jdk\corretto-17.0.8\bin\java.exe ...
Main Thread Iterating Map and Current Entry is:101...A
Child Thread updating Map
Main Thread Iterating Map and Current Entry is:102...B
Main Thread Iterating Map and Current Entry is:103...C
{101=A, 102=B, 103=C}

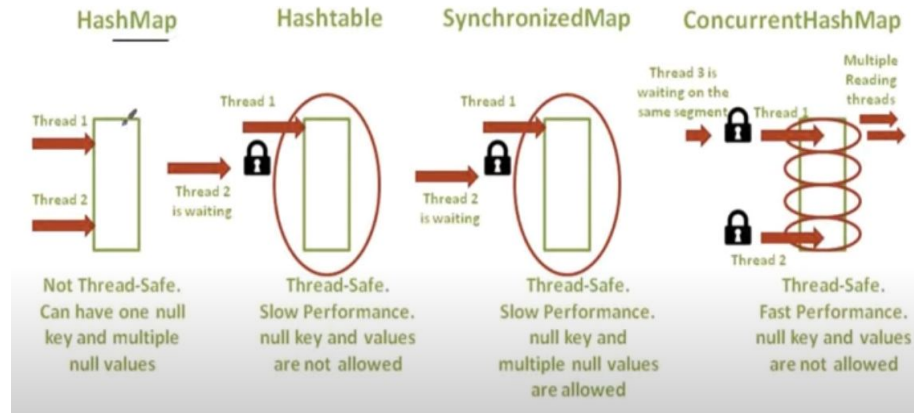
Process finished with exit code 0
```



## Concurrent collection

	HashMap	ConcurrentHashMap
Synchronized Internally?	✗	✓
Thread Safe	✗	✓
Introduced In	JDK 1.2	JDK 1.5
Package	java.util	java.util.concurrent
Null key	✓	✗
Null Value	✓	✗
Nature Of Iterators	Fail-Fast	Fail-Safe
Fast or Slow?	Fast	Slow
Where To Use?	Single threaded Applications	Multi threaded applications

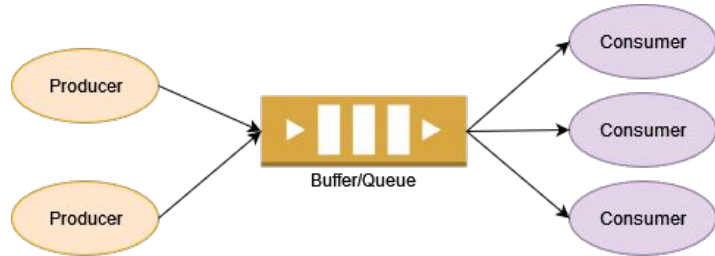
# Concurrent collection





# Producer-consumer design

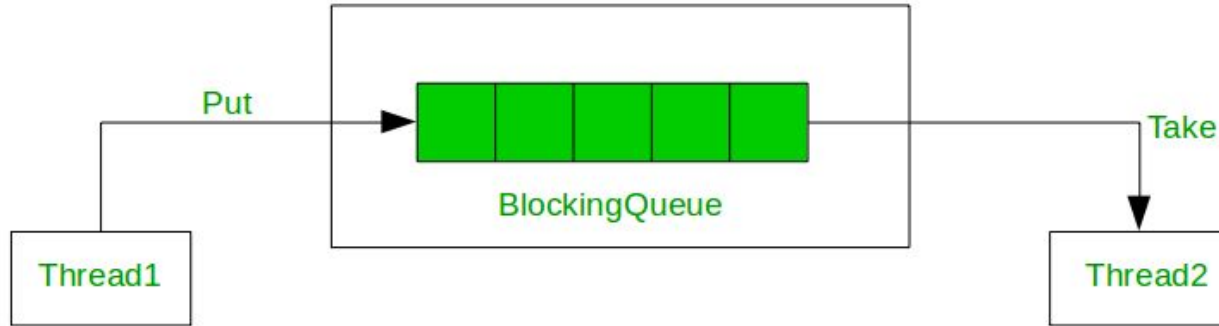
Single Producer-Consumer



Multi-Producers Multi-Consumers

# BlockingQueue

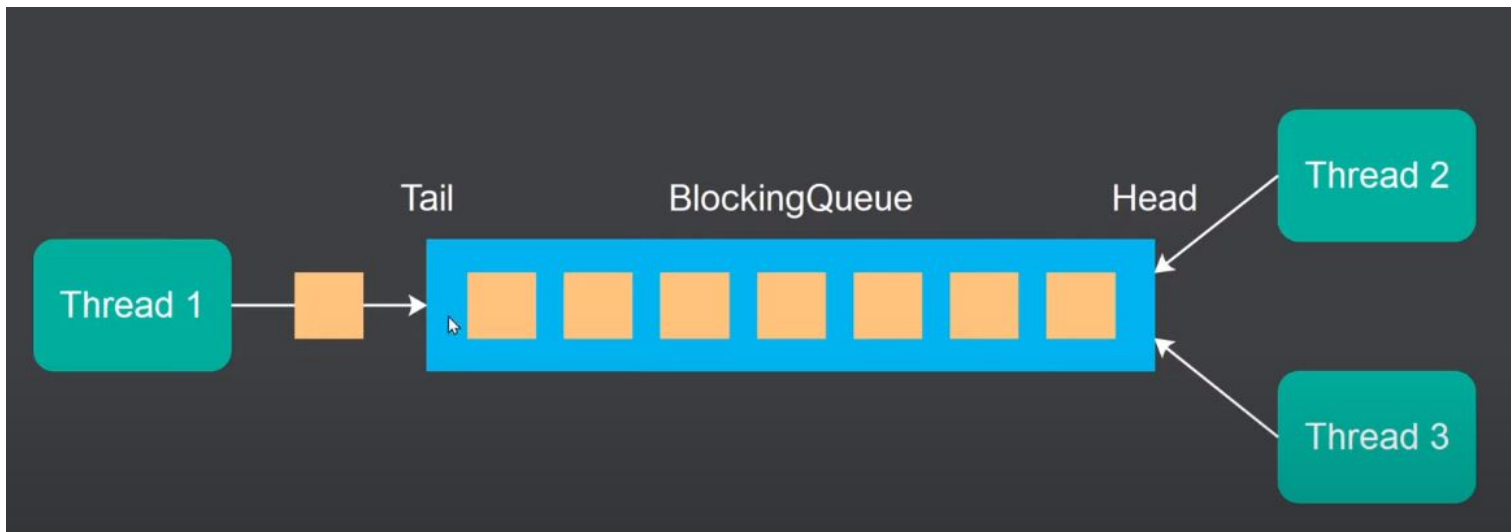
1. One of the most common producer-consumer designs is a thread pool coupled with a work queue;
2. BlockingQueue is a java Queue that support operations that wait for the queue to become non-empty when retrieving and removing an element, and wait for space to become available in the queue when adding an element





# BlockingQueue

1. Blocking queues support the **producer-consumer** design pattern.
2. If the queue is full, put blocks until space becomes available; if the queue is empty, take blocks until an element is available



# Queue

What is queue?

- Queue is a linear data structure in which the insertion and deletion operations are performed at two different ends

## Introduction to Queues

Queue ADT



Queue - First-In-First-Out  
(FIFO)

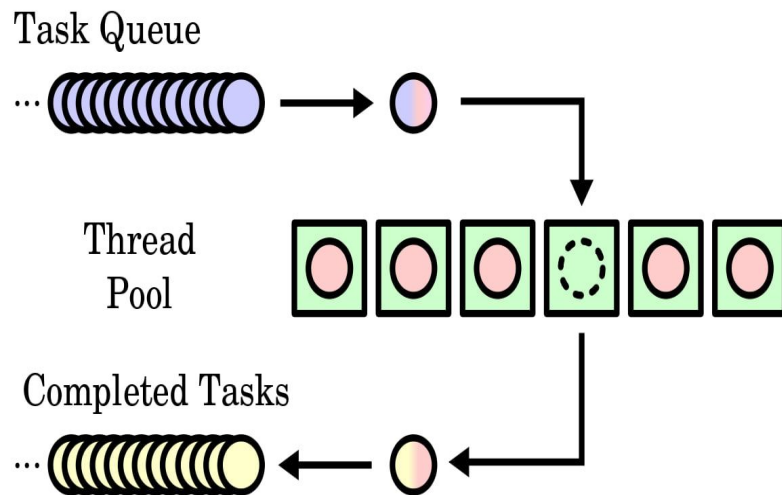
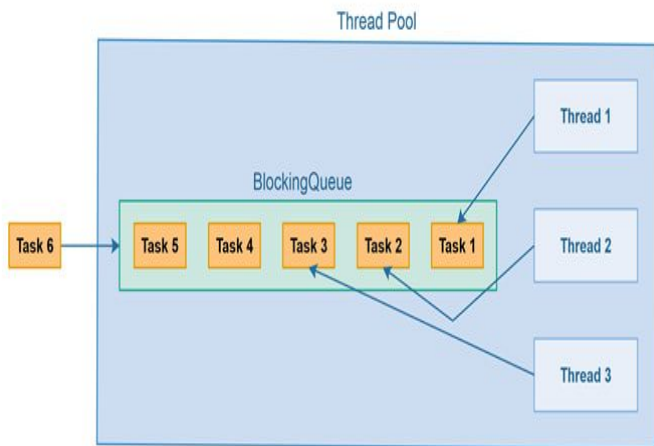


Stack - Last-In-First-Out  
(LIFO)

mycodeschool.com

# The Thread Pool

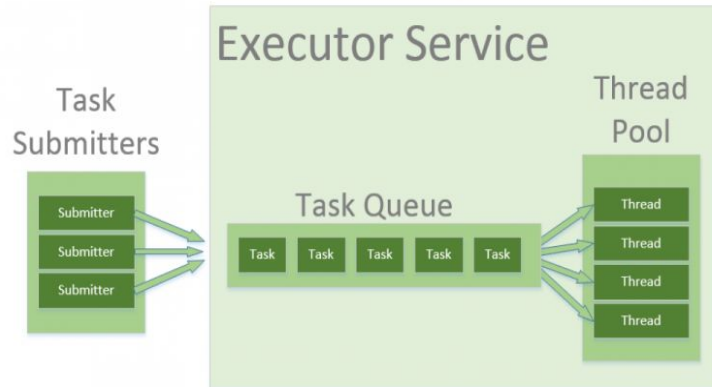
If the producers don't generate work fast enough to keep the consumers busy, the consumers just wait until more work is available



# The Thread Pool

1. The Thread Pool pattern helps to save resources in a multithreaded application and to contain the parallelism in certain predefined limits.
2. When we use a thread pool, we **write our concurrent code in the form of parallel tasks and submit them for execution to an instance of a thread pool**. This instance controls several re-used threads for executing these tasks.

The pattern allows us to **control the number of threads the application creates** and their life cycle. We're also able to schedule tasks' execution and keep incoming tasks in a queue





## The Thread Pool

- ❖ We use the *Executor* and *ExecutorService* interfaces to work with different thread pool implementations in Java.
- ❖ The *Executor* interface has a **single execute method** to submit *Runnable* instances for execution.
- ❖ The *ExecutorService* interface contains a large number of methods to **control the progress of the tasks and manage the termination of the service**.

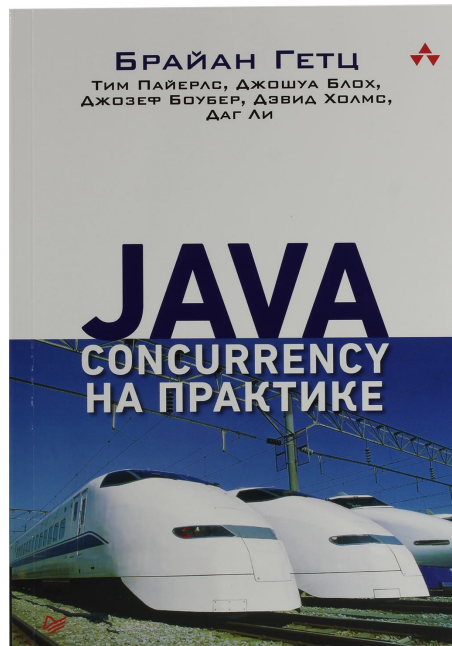


## Conculustion

- All concurrency issues boil down to coordinating access to mutable state. The less mutable state, the easier it is to ensure thread safety
- Make fields final unless they need to be mutable.
- Immutable objects are automatically thread-safe.
- Immutable objects simplify concurrent programming tremendously. They are simpler and safer, and can be shared freely without locking or defensive copying.
- Guard each mutable variable with a lock
- Guard all variables in an invariant with the same lock.
- Guard all variables in an invariant with the same lock.
- A program that accesses a mutable variable from multiple threads without synchronization is a broken program.



## Resources





## Reference

1. Java Concurrency in [Practice](#)
2. ConcurrentHashMap [Achieve](#)
3. The Thread pool in [java](#)





**Thank you!**

Presented by

**Temirmalik Nomozov**

**([temirmaliknomozov@gmail.com](mailto:temirmaliknomozov@gmail.com))**