Java Concurrency in Practice

Chapter-12:
Testing Concurrent Programs

Upcode Software
Engineer Team

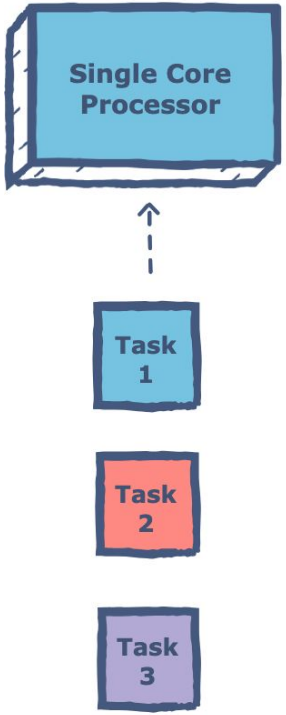
-2023-

CONTENT

1. What is Concurrent Programs?
2. Difference between Concurrent & Parallel programming.
3. Why Concurrent Programming?
4. What is Software Testing?
5. What is concurrency testing?
6. Concurrent testing methods.
7. Testing for Correctness.
8. Basic unit tests.
9. Testing blocking operations.
10. Generating more interleavings.
11. Extending PutTakeTest to add timing.

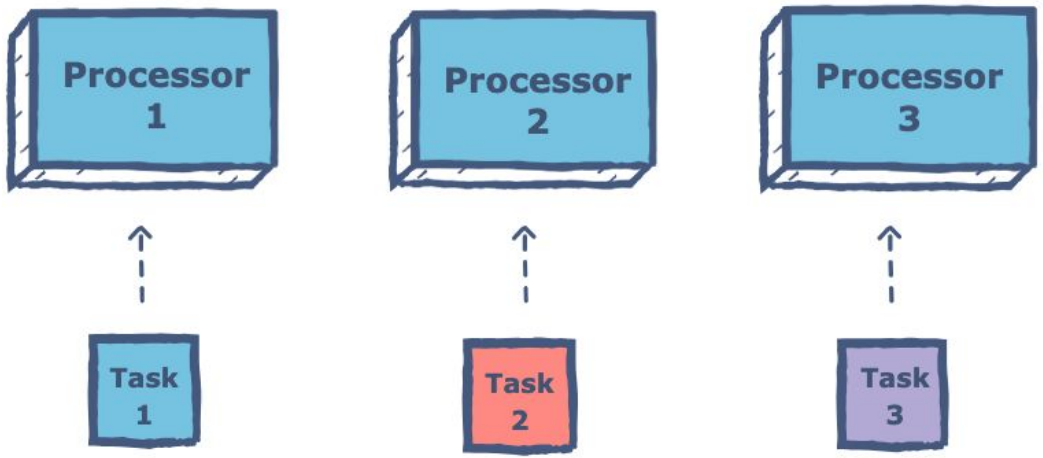
What is Concurrent Programs?

- In programming terms, **concurrent programming** is a technique in which two or more processes start, run in an *interleaved fashion* through **context switching** and complete in an overlapping time period by managing access to *shared resources* e.g. on a single core of CPU.
- This doesn't necessarily mean that multiple processes will be *running at the* **same instant** – even if the results might make it seem like it.



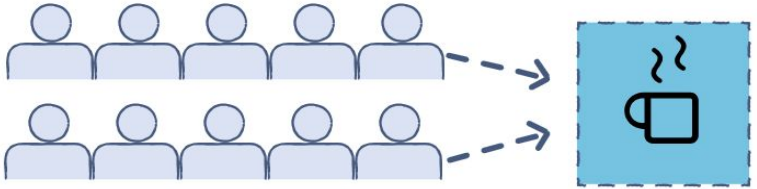
Concurrent & Parallel programming

- In *parallel programming*, parallel processing is achieved through hardware parallelism e.g. executing two processes on two separate CPU cores simultaneously.

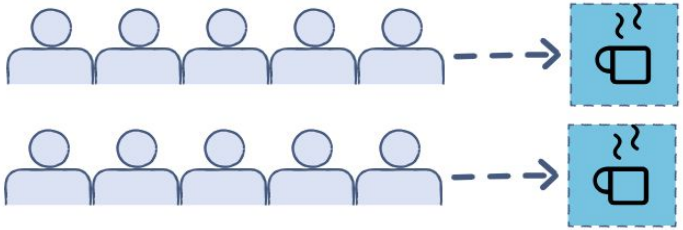


A Real World Example

Concurrent: *Two Queues* & a *Single* Espresso machine.



Parallel: *Two Queues* & *Two* Espresso machines.



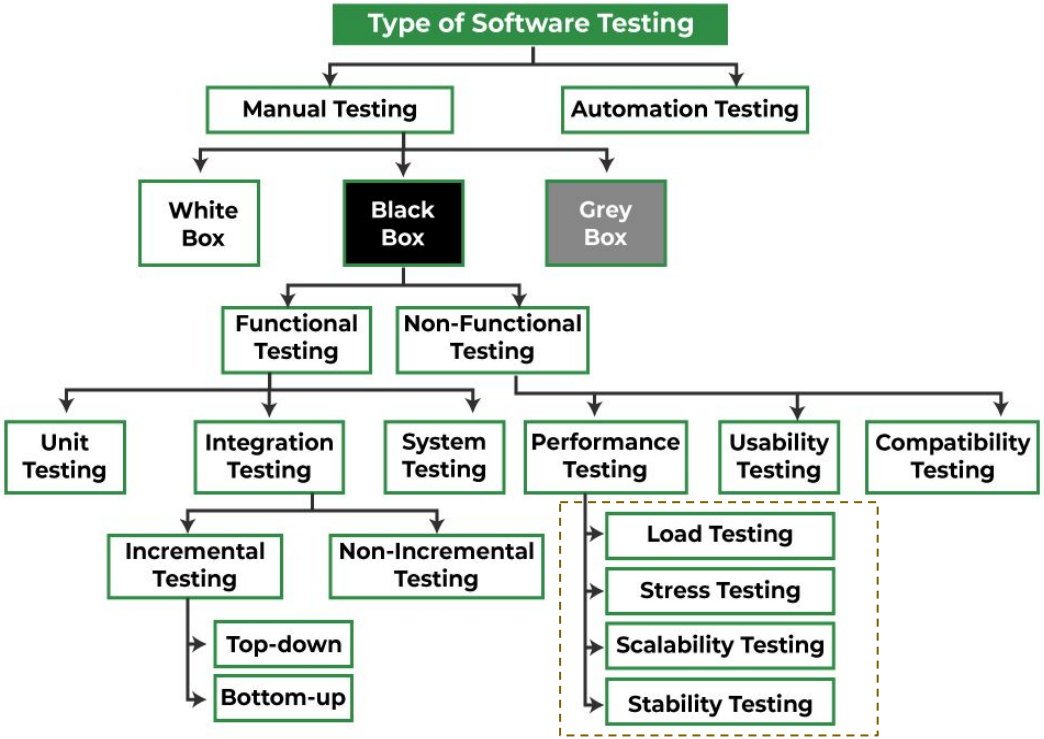
Why Concurrent Programming?

- **Performance Improvement:** Concurrent programs can utilize multiple CPU cores efficiently, which can lead to significant performance improvements for CPU-bound tasks.
- **Responsiveness:** For applications with user interfaces, concurrency ensures that the application remains responsive even when performing time-consuming operations in the background.
- **Resource Utilization:** Concurrent programs can make better use of system resources, such as CPU time, memory, and I/O devices.
- **Parallelism:** Some tasks naturally lend themselves to parallel execution, like rendering graphics, processing data, or handling multiple client requests in a server application.

What is Software Testing? (1/n)

- Software Testing is a method to assess the functionality of the software program. The process checks whether the actual software matches the expected requirements and ensures the software is bug-free.
- The purpose of software testing is to identify the errors, faults, or missing requirements in contrast to actual requirements. It mainly aims at measuring the specification, functionality, and performance of a software program or application.

What is Software Testing? (2/n)

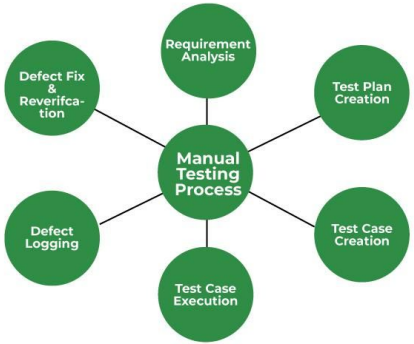


What is Software Testing? (3/n)

- **Software testing can be divided into two steps:**
  - **Verification:** It refers to the set of tasks that ensure that the software correctly implements a specific function. It means “Are we building the product right?”.
  - **Validation:** It refers to a different set of tasks that ensure that the software that has been built is traceable to customer requirements. It means “Are we building the right product?”.

Manual Testing (1/n)

- **Manual Testing:** Manual testing includes testing software manually, i.e., without using any automation tool or script. In this type, the tester takes over the role of an end-user and tests the software to identify any unexpected behavior or bug.
- There are different stages for manual testing such as unit testing, integration testing, system testing, and user acceptance testing.
- Testers use test plans, test cases, or test scenarios to test software to ensure the completeness of testing. Manual testing also includes exploratory testing, as testers explore the software to identify errors in it.



Automation Testing (1/n)

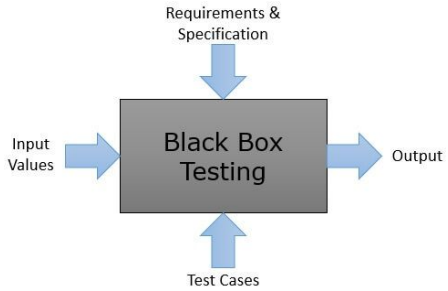
- **Automation Testing:** Automation testing, which is also known as Test Automation, is when the tester writes scripts and uses another software to test the product. This process involves the automation of a manual process. Automation Testing is used to re-run the test scenarios quickly and repeatedly, that were performed manually in manual testing.
  Apart from regression testing, automation testing is also used to test the application from a load, performance, and stress point of view. It increases the test coverage, improves accuracy, and saves time and money when compared to manual testing.

White box Testing (1/n)

- **White box testing** techniques analyse the internal structures the used data structures, internal design, code structure, and the working of the software rather than just the functionality as in black box testing. It is also called glass box testing or clear box testing or structural testing. White Box Testing is also known as transparent testing or open box testing.
- White box testing is a software testing technique that involves testing the internal structure and workings of a software application. The tester has access to the source code and uses this knowledge to design test cases that can verify the correctness of the software at the code level.
- White box testing is also known as structural testing or code-based testing, and it is used to test the software's internal logic, flow, and structure. The tester creates test cases to examine the code paths and logic flows to ensure they meet the specified requirements.

Black Box Testing (1/n)

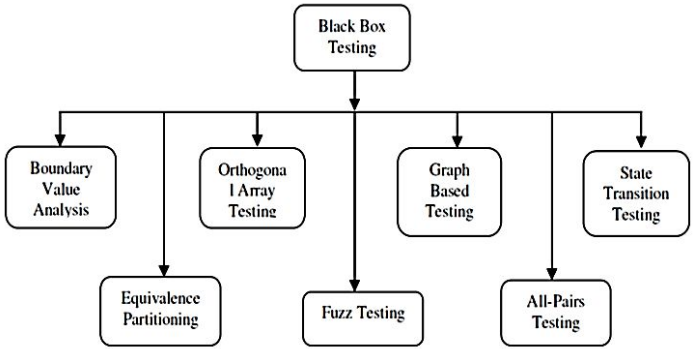
- **Black-box** testing is a type of software testing in which the tester is not concerned with the internal knowledge or implementation details of the software but rather focuses on validating the functionality based on the provided specifications or requirements.
- **Syntax-Driven Testing** – This type of testing is applied to systems that can be syntactically represented by some language. For example, language can be represented by context-free grammar. In this, the test cases are generated so that each grammar rule is used at least once.



Black Box Testing

Black Box Testing (2/n)

- **Equivalence partitioning** – It is often seen that many types of inputs work similarly so instead of giving all of them separately we can group them and test only one input of each group. The idea is to partition the input domain of the system into several equivalence classes such that each member of the class works similarly, i.e., if a test case in one class results in some error, other members of the class would also result in the same error.



Gray Box Testing (1/n)

- **Gray Box Testing** is a software testing technique which is a combination of Black Box Testing technique and White Box Testing technique. In Black Box Testing technique, tester is unknown to the internal structure of the item being tested and in White Box Testing the internal structure is known to tester. The internal structure is partially known in Gray Box Testing. This includes access to internal data structures and algorithms for purpose of designing the test cases.
- Gray Box Testing is named so because the software program is like a semitransparent or grey box inside which tester can partially see. It commonly focuses on context-specific errors related to web systems. It is based on requirement test case generation because it has all the conditions presented before the program is tested.

Functional Testing (1/n)

- Functional Testing is a type of Software Testing in which the system is tested against the functional requirements and specifications. Functional testing ensures that the requirements or specifications are properly satisfied by the application. This type of testing is particularly concerned with the result of processing. It focuses on the simulation of actual system usage but does not develop any system structure assumptions. The article focuses on discussing function testing.
- If you’re aiming to excel in roles like Software Tester, QA Engineer, or Test Automation Engineer, explore Complete Guide to Software Testing and Automation.

Functional Testing (2/n)

- Functional testing mainly involves black box testing and can be done manually or using automation. The purpose of functional testing is to:
  - **Test each function of the application:** Functional testing tests each function of the application by providing the appropriate input and verifying the output against the functional requirements of the application.
  - **Test primary entry function:** In functional testing, the tester tests each entry function of the application to check all the entry and exit points.
  - **Test flow of the GUI screen:** In functional testing, the flow of the GUI screen is checked so that the user can navigate throughout the application.

Non - Functional Testing (1/n)

- Behavior **Non-functional Testing** is a type of Software Testing that is performed to verify the non-functional requirements of the application. It verifies whether the behavior of the system is as per the requirement or not. It tests all the aspects that are not tested in functional testing.
- Non-functional testing is a software testing technique that checks the non-functional attributes of the system. Non-functional testing is defined as a type of software testing to check non-functional aspects of a software application. It is designed to test the readiness of a system as per nonfunctional parameters which are never addressed by functional testing. Non-functional testing is as important as functional testing.

Non - Functional Testing (2/n)

- The objectives of non-functional testing are:
  - **Increased usability:** To increase usability, efficiency, maintainability, and portability of the product.
  - **Reduction in production risk:** To help in the reduction of production risk related to non-functional aspects of the product.
  - **Reduction in cost:** To help in the reduction of costs related to non-functional aspects of the product.
  - **Optimize installation:** To optimize the installation, execution, and monitoring way of the product.
  - **Collect metrics:** To collect and produce measurements and metrics for internal research and development.
  - **Enhance knowledge of product:** To improve and enhance knowledge of the product behavior and technologies in use.

Maintenance Testing (1/n)

- **Software Maintenance** refers to the process of modifying and updating a software system after it has been delivered to the customer. This can include fixing bugs, adding new features, improving performance, or updating the software to work with new hardware or software systems. The goal of software maintenance is to keep the software system working correctly, efficiently, and securely, and to ensure that it continues to meet the needs of the users.
- Software maintenance is a continuous process that occurs throughout the entire life cycle of the software system. It is important to have a well-defined maintenance process in place, which includes testing and validation, version control, and communication with stakeholders.

Maintenance Testing (2/n)

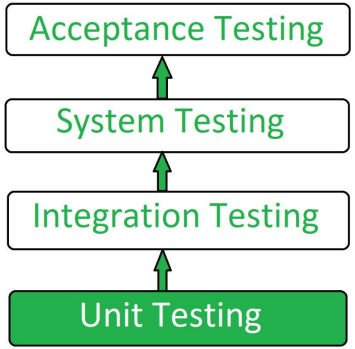
- **Several Key Aspects of Software Maintenance**
  - **Bug Fixing:** The process of finding and fixing errors and problems in the software.
  - **Enhancements:** The process of adding new features or improving existing features to meet the evolving needs of the users.
  - **Performance Optimization:** The process of improving the speed, efficiency, and reliability of the software.
  - **Porting and Migration:** The process of adapting the software to run on new hardware or software platforms.
  - **Re-Engineering:** The process of improving the design and architecture of the software to make it more maintainable and scalable.
  - **Documentation:** The process of creating, updating, and maintaining the documentation for the software, including user manuals, technical specifications, and design documents.

Unit Testing – Software Testing (1/n)

- **Unit Testing** is a software testing technique using which individual units of software i.e. group of computer program modules, usage procedures, and operating procedures are tested to determine whether they are suitable for use or not. It is a testing method using which every independent module is tested to determine if there is an issue by the developer himself.
- It is correlated with the functional correctness of the independent modules. Unit Testing is defined as a type of software testing where individual components of a software are tested. Unit Testing of the software product is carried out during the development of an application. An individual component may be either an individual function or a procedure.
- Unit Testing is typically performed by the developer. In SDLC or V Model, Unit testing is the first level of testing done before integration testing. Unit testing is a type of testing technique that is usually performed by developers. Although due to the reluctance of developers to test, quality assurance engineers also do unit testing.

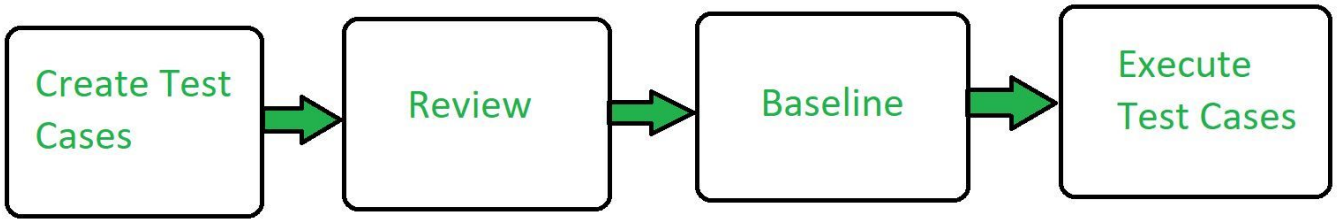
Unit Testing – Software Testing (2/n)

- The objective of Unit Testing is:
  - To isolate a section of code.
  - To verify the correctness of the code.
  - To test every function and procedure.
  - To fix bugs early in the development cycle and to save costs.
  - To help the developers understand the code base and enable them to make changes quickly.
  - To help with code reuse.



Unit Testing – Software Testing (3/n)

- **Workflow of Unit Testing:**



Integration Testing (1/n)

What is concurrency testing? (1/n)

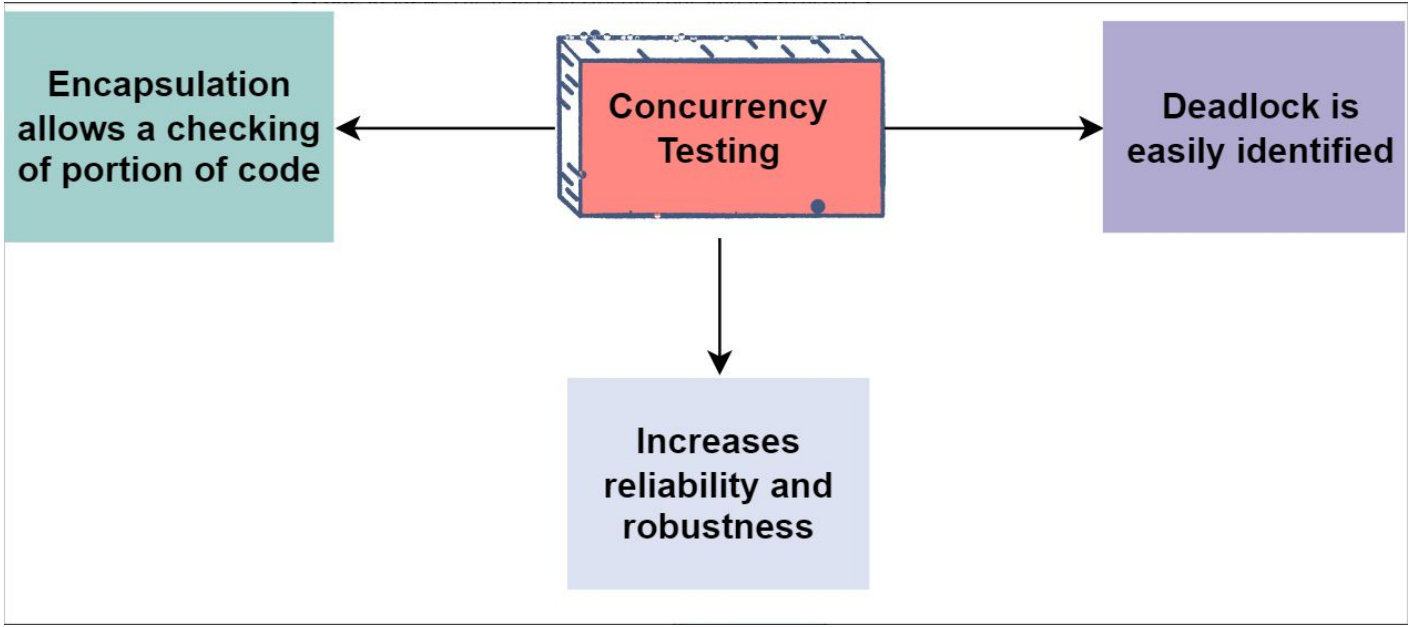
- **Concurrency testing** is a type of software testing that checks the performance of software when multiple users are logged in and perform actions simultaneously. Hence, it is also referred to as **multi-user testing**.
- Concurrency testing is challenging to conduct compared to sequential testing, due to the following three problems:
  - synchronization problems
  - non-determinism
  - time-related defects difficult to detect
- Concurrency testing mainly monitors the system for deadlocking, locking, and single-threaded code.

Concurrent testing methods (1/n)

- Some common methods for performing concurrency testing include the following:
  - **Fuzz Test**: The tester inserts incorrect random data and monitors the system's response.
  - **Random Test**: Multiple strands are tested simultaneously by randomizing the test inputs.
  - **Code Review**: The testers verify the code and its structures.
  - **Static Analysis**: The tester checks the coding system before the code's execution.

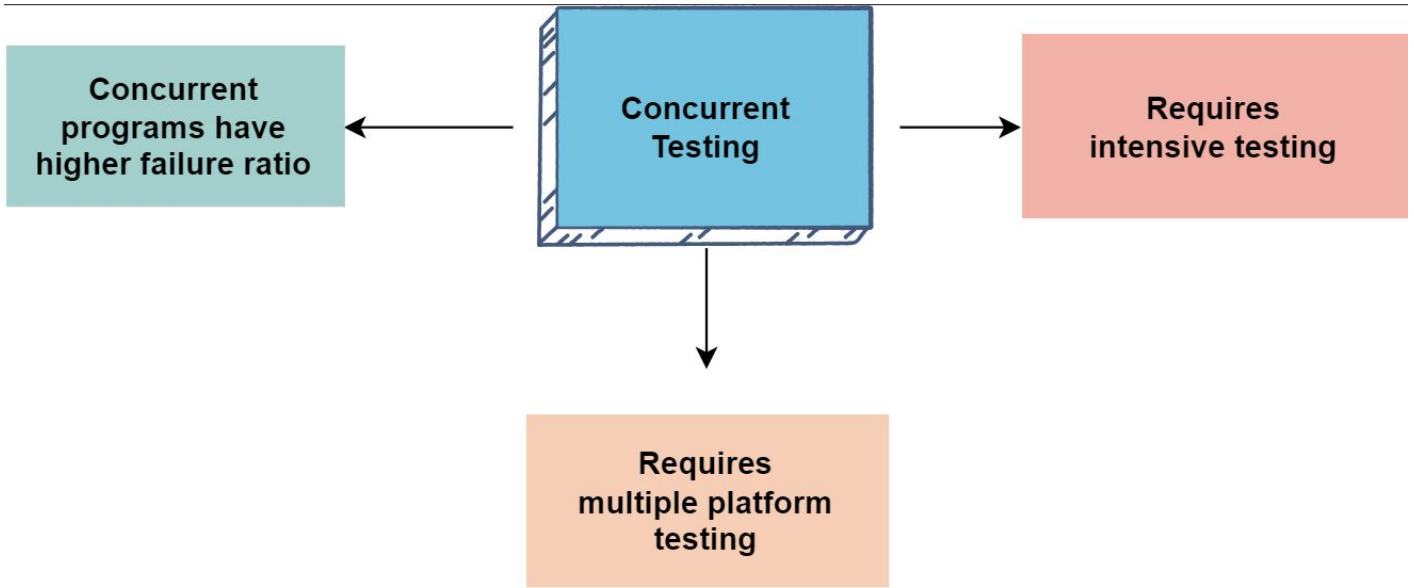
Advantages

- Some benefits of concurrency testing include the following:



Disadvantages

- Some drawbacks of concurrency testing include the following:



Testing for Correctness (1/n)

```
@ThreadSafe
public class BoundedBuffer<E> {
    private final Semaphore availableItems, availableSpaces;
    @GuardedBy("this") private final E[] items;
    @GuardedBy("this") private int putPosition = 0, takePosition = 0;

    public BoundedBuffer(int capacity) {
        availableItems = new Semaphore(0);
        availableSpaces = new Semaphore(capacity);
        items = (E[]) new Object[capacity];
    }

    public boolean isEmpty() {
        return availableItems.availablePermits() == 0;
    }

    public boolean isFull() {
        return availableSpaces.availablePermits() == 0;
    }

    public void put(E x) throws InterruptedException {
        availableSpaces.acquire();
        doInsert(x);
        availableItems.release();
    }

    public E take() throws InterruptedException {
        availableItems.acquire();
        E item = doExtract();
        availableSpaces.release();
        return item;
    }

    private synchronized void doInsert(E x) {
        int i = putPosition;
        items[i] = x;
        putPosition = (++i == items.length)? 0 : i;
    }

    private synchronized E doExtract() {
        int i = takePosition;

      E x = items[i];
      items[i] = null;
      takePosition = (++i == items.length)? 0 : i;
      return x;
    }
}
```

Testing for Correctness (2/n)

- The usage is **@GuardedBy(lock)** which means the guarded fields or methods can be accessed by some thread only when the thread is holding the lock. We can specify the lock to the following types:
- **this** : The intrinsic lock of the object in whose class the field is defined.
- **class-name.this** : For inner classes, it may be necessary to disambiguate ‘this’; the class-name.this designation allows you to specify which ‘this’ reference is intended
- **itself** : For reference fields only; the object to which the field refers.
- **field-name** : The lock object is referenced by the (instance or static) field specified by field-name.

Testing for Correctness (3/n)

- **class-name.field-name** : The lock object is reference by the static field specified by class-name.field-name.
- **method-name()** : The lock object is returned by calling the named nil-ary method.
- **class-name.class** : The Class object for the specified class should be used as the lock object.

An example:

```
public class BankAccount {
  private Object credential = new Object();

  @GuardedBy("credential")
  private int amount;
}
```

Testing for Correctness (4/n)

- BoundedBuffer implements a fixed-length array-based queue with blocking put and take methods controlled by a pair of counting semaphores. The availableItems semaphore represents the number of elements that can be removed from the buffer, and is initially zero (since the buffer is initially empty). Similarly, availableSpaces represents how many items can be inserted into the buffer, and is initialized to the size of the buffer.
- A take operation first requires that a permit be obtained from availableItems. This succeeds immediately if the buffer is nonempty, and otherwise blocks until the buffer becomes nonempty. Once a permit is obtained, the next element from the buffer is removed and a permit is released to the availableSpaces semaphore.
- If you need a bounded buffer you should use ArrayBlockingQueue or LinkedBlockingQueue rather than rolling your own, but the technique used here illustrates how insertions and removals can be controlled in other data structures as well

Basic unit tests (1/n)

```
class BoundedBufferTest extends TestCase {
    void testIsEmptyWhenConstructed() {
        BoundedBuffer<Integer> bb = new BoundedBuffer<Integer>(10);
        assertTrue(bb.isEmpty());
        assertFalse(bb.isFull());
    }

    void testIsFullAfterPuts() throws InterruptedException {
        BoundedBuffer<Integer> bb = new BoundedBuffer<Integer>(10);
        for (int i = 0; i < 10; i++)
            bb.put(i);
        assertTrue(bb.isFull());
        assertFalse(bb.isEmpty());
    }
}
```

Testing blocking operations (1/n)

```
void testTakeBlocksWhenEmpty() {
    final BoundedBuffer<Integer> bb = new BoundedBuffer<Integer>(10);
    Thread taker = new Thread() {
        public void run() {
            try {
                int unused = bb.take();
                fail(); // if we get here, it's an error
            } catch (InterruptedException success) { }
        }};
    try {
        taker.start();
        Thread.sleep(LOCKUP_DETECT_TIMEOUT);
        taker.interrupt();
        taker.join(LOCKUP_DETECT_TIMEOUT);
        assertFalse(taker.isAlive());
    } catch (Exception unexpected) {
        fail();
    }
}
```

Testing blocking operations (2/n)

- **taker.start()** starts the thread.
- **Thread.sleep(LOCKUP_DETECT_TIMEOUT)** pauses the main thread for a specified timeout (**LOCKUP_DETECT_TIMEOUT**).
- **taker.interrupt()** interrupts the **taker** thread.
- **taker.join(LOCKUP_DETECT_TIMEOUT)** waits for the taker thread to finish for a specified timeout **(LOCKUP_DETECT_TIMEOUT)**.
- **assertFalse(taker.isAlive())** checks that the taker thread is no longer alive, indicating it has completed.

- In summary, this test is checking that the take method of the BoundedBuffer class behaves correctly when attempting to take an element from an empty buffer, specifically testing the handling of InterruptedException and the termination of the thread after interruption.
- The use of Thread.sleep and interruption is likely related to detecting potential deadlocks or other concurrency issues.

Test method to verify thread pool expansion (1/n)

- If the core pool size is smaller than the maximum size, the thread pool should grow as demand for execution increases. Submitting long-running tasks to the pool makes the number of executing tasks stay constant for long enough to make a few assertions, such as testing that the pool is expanded as expected, as shown

```
public void testPoolExpansion() throws InterruptedException {
    int MAX_SIZE = 10;
    ExecutorService exec = Executors.newFixedThreadPool(MAX_SIZE);

    for (int i = 0; i < 10 * MAX_SIZE; i++)
        exec.execute(new Runnable() {
            public void run() {
                try {
                    Thread.sleep(Long.MAX_VALUE);
                } catch (InterruptedException e) {
                    Thread.currentThread().interrupt();
                }
            }
        });
    for (int i = 0;
         i < 20 && threadFactory.numCreated.get() < MAX_SIZE;
         i++)
        Thread.sleep(100);
    assertEquals(threadFactory.numCreated.get(), MAX_SIZE);
    exec.shutdownNow();
}
```

Test method to verify thread pool expansion (1/n)

- If the core pool size is smaller than the maximum size, the thread pool should grow as demand for execution increases. Submitting long-running tasks to the pool makes the number of executing tasks stay constant for long enough to make a few assertions, such as testing that the pool is expanded as expected, as shown

```
public void testPoolExpansion() throws InterruptedException {
    int MAX_SIZE = 10;
    ExecutorService exec = Executors.newFixedThreadPool(MAX_SIZE);

    for (int i = 0; i < 10 * MAX_SIZE; i++)
        exec.execute(new Runnable() {
            public void run() {
                try {
                    Thread.sleep(Long.MAX_VALUE);
                } catch (InterruptedException e) {
                    Thread.currentThread().interrupt();
                }
            }
        });
    for (int i = 0;
         i < 20 && threadFactory.numCreated.get() < MAX_SIZE;
         i++)
        Thread.sleep(100);
    assertEquals(threadFactory.numCreated.get(), MAX_SIZE);
    exec.shutdownNow();
}
```

Generating more interleavings (1/n)

- Using Thread.yield to generate more interleavings

```
public synchronized void transferCredits(Account from,
                                         Account to,
                                         int amount) {
    from.setBalance(from.getBalance() - amount);
    if (random.nextInt(1000) > THRESHOLD)
        Thread.yield();
    to.setBalance(to.getBalance() + amount);
}
```

Extending PutTakeTest to add timing (1/n)

- Barrier-based timer.

```
this.timer = new BarrierTimer();
this.barrier = new CyclicBarrier(npairs * 2 + 1, timer);

public class BarrierTimer implements Runnable {
    private boolean started;
    private long startTime, endTime;

    public synchronized void run() {
        long t = System.nanoTime();
        if (!started) {
            started = true;
            startTime = t;
        } else
            endTime = t;
    }
    public synchronized void clear() {
        started = false;
    }
    public synchronized long getTime() {
        return endTime - startTime;
    }
}
```

Extending PutTakeTest to add timing (2/n)

- Testing with a barrier-based timer

```
public void test() {
    try {
        timer.clear();
        for (int i = 0; i < nPairs; i++) {
            pool.execute(new Producer());
            pool.execute(new Consumer());
        }
        barrier.await();
        barrier.await();
        long nsPerItem = timer.getTime() / (nPairs * (long)nTrials);
        System.out.print("Throughput: " + nsPerItem + " ns/item");
        assertEquals(putSum.get(), takeSum.get());
    } catch (Exception e) {
        throw new RuntimeException(e);
    }
}
```

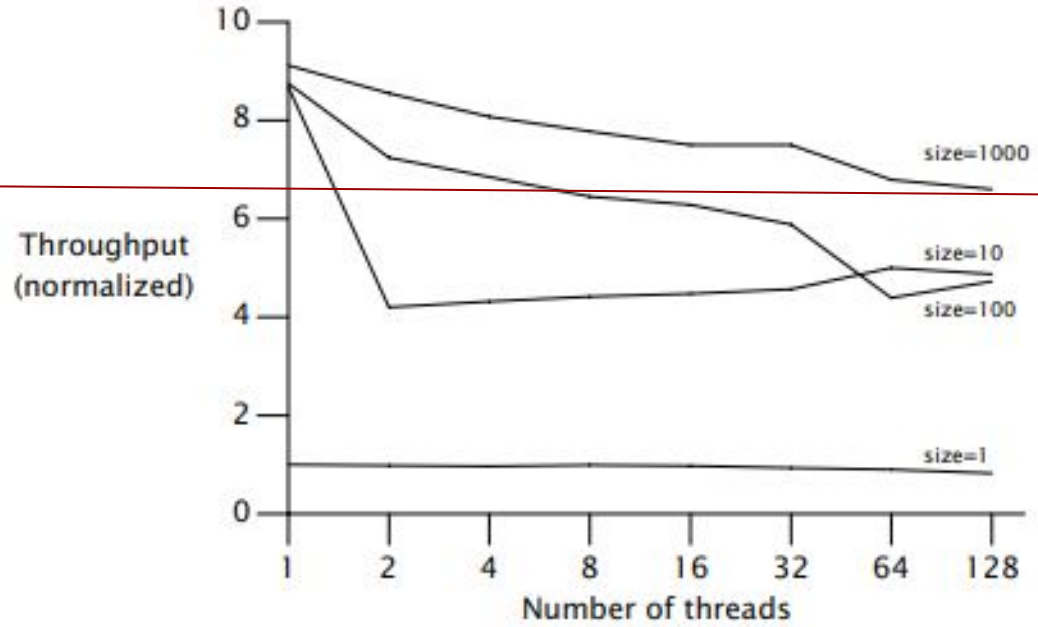
Extending PutTakeTest to add timing (3/n)

- Driver program for TimedPutTakeTest.

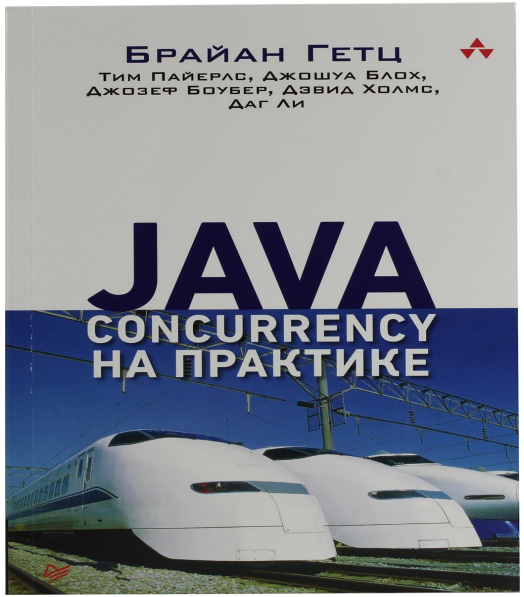
```
public static void main(String[] args) throws Exception {
    int tpt = 100000; // trials per thread
    for (int cap = 1; cap <= 1000; cap *= 10) {
        System.out.println("Capacity: " + cap);
        for (int pairs = 1; pairs <= 128; pairs *= 2) {
            TimedPutTakeTest t =
                new TimedPutTakeTest(cap, pairs, tpt);
            System.out.print("Pairs: " + pairs + "\t");
            t.test();
            System.out.print("\t");
            Thread.sleep(1000);
            t.test();
            System.out.println();
            Thread.sleep(1000);
        }
    }
    pool.shutdown();
}
```

Extending PutTakeTest to add timing (4/n)

- TimedPutTakeTest with various buffer capacities.



Resources


Брайан Гетц
Тим Пайерлс, Джошуа Блох,
Джозеф Боубер, Дэвид Холмс,
Даг Ли
JAVA
CONCURRENCY
НА ПРАКТИКЕ

Reference

1. Concurrency book.
2. Concurrent programming.
3. What is Conccurent programming?
4. Software Testing.
5. What is Software Testing?
6. Testing Concurrent.

Thank you!

Presented by

Sherjonov Jahongir

(jakhongirsherjonov@gmail.com)