### Important features in 1.8 Java

## Lamda Expressions :

## Parallel Array Operations

## Concurrent Counters

## Date Time APIs

## Nashorn JavaScript

## Garbage Collection

## HashMap Collisions

2. Parallel Array Operations

1. **ParallelSort** implement a parallel sort-merge algorithm that recursively breaks the array into pieces, sorts them, then recombines them concurrently and transparently
2. **Stream Processing** from an Array, you can create a Stream object that allows for parallel processing on a set of data as a whole, treated as a stream. The difference between a Collection of data and a JDK8 Stream of data is that a Collection allows you to manage the elements individually, whereas aStream doesn't. For example, using a Collection, you can add elements, remove them, access them, and insert them in the middle. A Stream doesn't allow you to manipulate individual elements of the data set, but instead allows you to perform aggregate functions on the data as a whole. You can perform useful operations such as pulling out only distinct values (ignoring duplicates) from a set, data conversions, finding min and max values, map-reduce functions, and other mathematical operations.

### **Arrays.parallelSetAll() :** parallelSetAll() method creates an array and sets each element's value with a given generator function, using concurrency for added efficiency and performance. This method relies on lambdas (called "closures" in other languages)

### **Arrays.parallelPrefix() :** The parallelPrefix method performs a given mathematical function on the elements of the array cumulatively, setting the results within the array concurrently.

## 5. Date Time APIs

Java 1.0 introduced System.currentTimeMillis() and java.util.Date class. It soon became evident that there were several flaws with this class:

1. Constructors that accept year arguments require offsets from 1900, which has been a source of bugs.
2. January is represented by 0 instead of 1, also a source of bugs.
3. Date doesn't describe a date but describes a date-time combination.
4. Date's mutability makes it unsafe to use in multithreaded scenarios without external synchronization.
5. Date isn't amenable to internationalization.

Calendar was also riddled with flaws, including the following:

1. It isn't possible to format a calendar.
2. January is represented by 0 instead of 1, a source of bugs.
3. Calendar isn't type-safe; for example, you must pass an int-based constant to the get(int field) method. (In fairness, enums weren't available when Calendarwas released.)
4. Calendar's mutability makes it unsafe to use in multithreaded scenarios without external synchronization. (The companion java.util.TimeZoneand java.text.DateFormat classes share this problem.)
5. Calendar stores its state internally in two different ways -- as a millisecond offset from the epoch and as a set of fields -- resulting in many bugs and performance issues.

## 7. HashMap Collisions

Improve the performance of java.util.HashMap under high hash-collision conditions by using balanced trees rather than linked lists to store map entries. Same is done for LinkedHashMap class. The principal idea is that once the number of items in a hash bucket grows beyond a certain threshold, that bucket will switch from using a linked list of entries to a balanced tree. In the case of high hash collisions, this will improve worst-case performance from O(n) to O(log n).

## 7. Nashorn JavaScript

The Nashorn javascript engine can either be used programmatically from java programs or by utilizing the command line tool jjs

Java can call js methods and vice-versa

## 8. Concurrent Counters

[**Dirty**](https://www.google.com/search?q=oscar+sesame&safe=off&espv=210&es_sm=91&source=lnms&tbm=isch&sa=X&ei=AX1MU-qjH5Gj8gGx84GIBg&ved=0CAgQ_AUoAQ&biw=1680&bih=878) **counters** – writing / reading from a regular object or static field across multiple threads. Unfortunately, this doesn’t work for two reasons. The first is that in Java, an A += B operation isn’t Atomic. If you open up the output bytecode, you’ll see at least four instructions – one for loading the field value from the heap into the thread stack, a second for loading the delta, a third to add them and the fourth to set the result into the field.

**Synchronized** – the most basic of concurrency idioms, this blocks all other threads while reading or writing the value.

**RWLock** – this slightly more sophisticated version of the basic Java lock enables you to discern between threads that change the value and need to block others vs. ones that only read and don’t require a critical section. While this can be more efficient (assuming the number of writers is low) it’s a pretty meh approach, as you’re blocking the execution of all other threads when acquiring the write lock. It’s really only a good approach when you know the number of writers is materially limited compared to readers.

**Volatile** – this keyword essentially instructs the JIT compiler to de-optimize the run-time machine code, so that any modification to the field is immediately seen by other threads. The JIT compiler may change the order in which assignments to fields are made. If you have just one thread updating a value and multiple threads consuming it, this is a really good strategy – no contention at all.

**AtomicInteger** – this set of classes uses CAS (compare-and-swap) processor instructions to update the value of the counter. The downside is that if it fails to set the value due to a race with another thread, it has to try again. Under high contention this can turn into a spin lock, where the thread has to continuously try and set the value in an [infinite loop](http://grepcode.com/file/repository.grepcode.com/java/root/jdk/openjdk/6-b14/java/util/concurrent/atomic/AtomicInteger.java#AtomicInteger.addAndGet%28int%29), until it succeeds.

<http://grepcode.com/file/repository.grepcode.com/java/root/jdk/openjdk/6-b14/java/util/concurrent/atomic/AtomicInteger.java#AtomicInteger.addAndGet%28int%29>

public final int addAndGet(int delta) {

for (;;) {

int current = get();

int next = current + delta;

if (compareAndSet(current, next))

return next;

}

}

public final boolean compareAndSet(int expect, int update) {

return unsafe.compareAndSwapInt(this, valueOffset, expect, update);

}

**Java 8 Adders :** From a usage perspective it’s very similar to an AtomicInteger. Simply create a [LongAdder](http://docs.oracle.com/javase/8/docs/api/java/util/concurrent/atomic/LongAdder.html) and use intValue() and add() to get / set the value. What this class does is when a straight CAS fails due to contention, it stores the delta in an internal cell object allocated for that thread. It then adds the value of pending cells to the sum when intValue() is called. This reduces the need to go back and CAS or block other threads.

Concurrent Adders clean house with a 60-100% performance boost over atomic integers.

## 1. Lambda Expressions

This addition to the language brings Java to the forefront of functional programming, right there with other functional JVM-based languages such as Scala and Clojure.

A closure is a function or reference to a function together with a referencing environment — a table storing a reference to each of the non-local variables of that function.

A lambda expression is an anonymous function. A method without a declaration, i.e., access modifier, return value declaration, and name.

Especially useful in places where a method is being used only once, and the method definition is short. It saves you the effort of declaring and writing a separate method to the containing class.

Syntax :

(argument) -> (body).

For example:

|  |
| --- |
| (arg1, arg2...) -> { body }  (type1 arg1, type2 arg2...) -> { body } |

## Difference between Lambda Expression and Anonymous class

One key difference between using Anonymous class and Lambda expression is the use of this keyword. For anonymous class ‘this’ keyword resolves to anonymous class, whereas for lambda expression ‘this’ keyword resolves to enclosing class where lambda is written.

Another difference between lambda expression and anonymous class is in the way these two are compiled. Java compiler compiles lambda expressions and convert them into private method of the class.

================================================================

========================== Example Code ==========================

================================================================

Example Code : Parallel Array Operations

import java.awt.image.BufferedImage;

import java.io.File;

import java.util.Arrays;

import java.util.Random;

import java.util.function.IntBinaryOperator;

import java.util.stream.IntStream;

import javax.imageio.ImageIO;

/\*\*

\*

\* @author MMT5709

\*/

public class ParallelSort {

class MyIntOperator implements IntBinaryOperator {

@Override

public int applyAsInt(int left, int right) {

return left + right;

}

}

private int[] mySrc;

public static void main(String[] args) {

ParallelSort mySort = new ParallelSort();

int[] src = null;

System.out.println("\nSerial sort:");

src = mySort.getData();

mySort.mySrc = src.clone();

int[] clonedSrc = src.clone();

mySort.sortIt(src, false);

System.out.println("\nParallel sort:");

mySort.sortIt(clonedSrc, true);

mySort.accumulate();

System.out.println("\nParallelSetAll : ");

mySort.createLargeArray();

System.out.println("\nStream Processing");

mySort.streamProcessing();

}

public void streamProcessing() {

int[] src = mySrc;

int[] clonedSrc = src.clone();

long start = System.currentTimeMillis();

IntStream stream = Arrays.stream(src);

int sum = stream.sum();

long end = System.currentTimeMillis();

System.out.println("\nSum: " + sum);

System.out.println("--Elapsed sort time using parallel: " + (end - start));

sum = 0;

start = System.currentTimeMillis();

for (int i = 0; i < clonedSrc.length; ++i) {

sum += clonedSrc[i];

}

end = System.currentTimeMillis();

System.out.println("\nSum: " + sum);

System.out.println("--Elapsed sort time without using parallel: " + (end - start));

}

public void createLargeArray() {

Integer[] array = new Integer[1024 \* 1024 \* 4]; // 4M

long start = System.currentTimeMillis();

Arrays.parallelSetAll(array, i -> new Random().nextInt());

long end = System.currentTimeMillis();

System.out.println("--Elapsed sort time: " + (end - start));

for (int i = 0; i < 10; ++i) {

System.out.print(array[i] + ", ");

}

System.out.println("");

}

public void accumulate() {

int[] src = null;

// accumulate test

System.out.println("\nParallel prefix:");

src = getData();

for (int i = 0; i < 10; ++i) {

System.out.print(src[i] + ", ");

}

System.out.println("");

long start = System.currentTimeMillis();

Arrays.parallelPrefix(src, new MyIntOperator());

long end = System.currentTimeMillis();

System.out.println("--Elapsed sort time: " + (end - start));

System.out.println("");

for (int i = 0; i < 10; ++i) {

System.out.print(src[i] + ", ");

}

}

public void sortIt(int[] src, boolean parallel) {

try {

System.out.println("--Array size: " + src.length);

long start = System.currentTimeMillis();

if (parallel == true) {

Arrays.parallelSort(src);

} else {

Arrays.sort(src);

}

long end = System.currentTimeMillis();

System.out.println(

"--Elapsed sort time: " + (end - start));

} catch (Exception e) {

e.printStackTrace();

}

}

private int[] getData() {

try {

File file = new File("/Users/MMT5709/Downloads/PAN\_Site.png");

BufferedImage image = ImageIO.read(file);

int w = image.getWidth();

int h = image.getHeight();

int[] src = image.getRGB(0, 0, w, h, null, 0, w);

int[] data = new int[src.length \* 20];

for (int i = 0; i < 20; i++) {

System.arraycopy(

src, 0, data, i \* src.length, src.length);

}

return data;

} catch (Exception e) {

e.printStackTrace();

}

return null;

}

}

=====================NASHORN================

import java.io.FileNotFoundException;

import java.io.FileReader;

import javax.script.Invocable;

import javax.script.ScriptEngine;

import javax.script.ScriptEngineManager;

import javax.script.ScriptException;

/\*\*

\*

\* @author MMT5709

\*/

public class Nashorn {

public static void main(String[] args) throws ScriptException, FileNotFoundException, NoSuchMethodException {

ScriptEngine engine = new ScriptEngineManager().getEngineByName("nashorn");

engine.eval(new FileReader("/Users/MMT5709/sid/js/script.js"));

Invocable invocable = (Invocable) engine;

Object result = invocable.invokeFunction("fun1", "Peter Parker");

System.out.println(result);

System.out.println(result.getClass());

}

}

================= Lambda ===============

import java.util.Arrays;

import java.util.List;

import java.util.function.Predicate;

public class Lambda {

public static void main(String[] a) {

List<Integer> list = Arrays.asList(1, 2, 3, 4, 5, 6, 7);

System.out.println("Print all numbers:");

evaluate(list, (n) -> true);

System.out.println("Print no numbers:");

evaluate(list, (n) -> false);

System.out.println("Print even numbers:");

evaluate(list, (n) -> n % 2 == 0);

System.out.println("Print odd numbers:");

evaluate(list, (n) -> n % 2 == 1);

System.out.println("Print numbers greater than 5:");

evaluate(list, (n) -> n > 5);

}

public static void evaluate(List<Integer> list, Predicate<Integer> predicate) {

for (Integer n : list) {

if (predicate.test(n)) {

System.out.println(n + " ");

}

}

}

}