Java 8 – Deep Dive Workshop

Rony Keren
rkeren@johnbryce.co.il

Topics

- Java language
  - Lambdas and Functional Interfaces
  - Method References
  - Interface Default and Static Methods
  - Repeating annotations
  - Extended Annotations Support
  - Reflection Parameter names
- Java APIs
  - Optional
  - Optimistic Reading – Lock API
  - Adders
  - Streams
  - Date/Time API (JSR 310)
  - Nashorn JavaScript engine
  - Base64
  - Parallel Arrays
- New Java tools
  - Nashorn engine: jjs
  - Class dependency analyzer: jdeps
  - Java Mission Control (jmc)
- Java runtime (JVM) New Features

Java Language

- Lambdas and Functional Interfaces
  - Java 6 came with scripting engine API
  - Default RI was Rhino Mozilla Java Script Engine
  - It raised a problem: How should dynamic code get executed in a non-dynamic plug-in like the JVM ?

```
function main (function, arg1, arg2){
                window[function](arg1,arg2);
 }

function dynamicFunc(x, y){
    alert(x+y);
}

//invoking:
main('dynamicFunc','hello','world');
```

```
function getDynamic(x){
    var f =  doSomething(x){
               if(x>y) return true;
               else return false;
    }
    return f;
}
```

Java Language

- Lambdas and Functional Interfaces

- Java 7 Dynamic Invocation
- JSR 292: Java Platform Support for Dynamically Typed Languages
  - Engines can use additional runtime instructions in order to compile code
    - Invokedynamic – new bytecode instruction that allows to translate method invocation without relating to the object holds it
    - Java.dyn.MethodHandler – bounded to each Invokedynamic instruction
      - holds reference to a JVM method
      - cached
    - Both invokedynamic and MethodHandlers are updated during runtime
    - Are addition to specialInvoker (direct call), staticInvoker, virtualInvoker (abstract call) and interfaceInvoker

- That's great…we got this dynamic thing in our plugin but we can't use it in Java…

Java Language

- Lambdas and Functional Interfaces

- Java 8 came to the rescue
  - Provides support for LAMBDA expressions
  - All LAMBDA calls are done via native *invokedynamic*
  - Now we can assign some code as a method
    - No need in callbacks (event model)
    - Reduces anonymous classes
    - More up-to-date way of coding

Java Language

- Lambdas and Functional Interfaces
  - LAMBDA expressions are supported for **functional interfaces**
  - Functional interface
    - Denoted with @**FunctionalInterface**
    - Some interfaces were updated (Runnable, Comparable, Event….)
    - Many new interfaces are provided (later)
    - You may create your own

Java Language

- Lambdas and Functional Interfaces
  - Working with Functional Interfaces
  - Old code
    - Abuse classes
    - Uses anonymous classes

```
public class MyRunnable implements Runnable {
	public void run(){
		.......
	}
}

(new Thread(new MyRunnable())).start();
```

```
(new Thread(new Runnable(){
	public void run(){
		.......
	}
})).start();
```

Java Language

- Lambdas and Functional Interfaces

  - Since Runnable is now:

```
@FunctionalInterface
public  interface Runnable {
        public void run();
}
```

  - You may use it like that:

```
Runnable r = () -> { ……… };
(new  Thread(r)).start();
```

```
(new  Thread(
     () -> { ……… }
)).start();
```

    - If your method is single lined you may skip { }
    - Not just simple coding… don’t forget it is also executed dynamically !

Java Language

- Lambdas and Functional Interfaces
  - Single-lined & blocks examples:

```
Runnable r = () -> System.out.println("Hey!");
(new  Thread(r))).start();
```

```
Runnable r = () -> {
            for(int i=0;i<100;i++){
                System.out.println( i );
            }};
(new  Thread(r)).start();
```

```
(new  Thread(() -> System.out.println("Hey!"))).start();
```

```
(new  Thread(() -> {
            for(int i=0;i<100;i++){
                System.out.println( i );
            }}
)).start();
```

Java Language

- Behind the scene steps for execution:

Functional:

1. Method is allocated & interpreted
2. Thread object executes the method directly

```
Runnable r = () -> System.out.println("Hey!");
(new  Thread(r))).start();
```

The old way:

1. MyRunnable.class is loaded
2. <u>MyRunnable instance is allocated on heap – this one is heaviest</u>
3. Thread object executes the run() method by using the object reference

```
public class MyRunnable implements Runnable {
        public void run(){
                .......
        }
}

(new Thread(new MyRunnable())).start();
```

Both method & object.run() gets optimized
And cached when becomes hot

Java Language

- What about ‘this’ keyword in LAMBDA exp?
  - LAMBDA code is not executed from an object instance
  - Only the method gets loaded – so there is nothing but the method reflection (e.g Runnable.class → run())
  - This means it is treated & executed as a ‘static’ like content
  - So…..no ‘this’……
  - The following code will fail to compile with “cannot use this in a static content” error

```
Runnable r = () -> System.out.println(this);
(new Thread(r))).start();
```

Java Language

- Lambdas and Functional Interfaces
  - What about methods that accepts parameters ?
    - Type safety is determined & enforced during compile time
    - Example:

```
List<String> words = Arrays.asList{"David","Adam","Eve","Moses"};
Collections.sort(words, XXX);
```

      - here, the compiler assumes you're about to assign Comparator<String>
      - Comparator in now @FunctionalInterface
      - Therefore, the compiler checks any assigned function to have the following signature:

```
public  int XXX (String s1, String s2)
```

      - Just like Comparator<T>.compare(T,T) signature

Java Language

- Lambdas and Functional Interfaces

```
public  int XXX (String s1, String s2)
```

  - So, we may come up with our own implementations
  - But this time Instead of creating a class & force object instantiation…
  - Old coding:

```
public class ReverseNameComparator implements Comparator<String>{

    public  int compare (String s1, String s2){
        return s1.compareTo(s2)*(-1);
    }
}
```

Java Language

- Lambdas and Functional Interfaces

```
public  int XXX (String s1, String s2)
```

  - We can do it this way:

```
List<String> words = Arrays.asList{“David”,”Adam”,”Eve”,”Moses”};
Comparator<String> c=(String s1,String s2)-> {return  s1.compareTo(s2)*(-1);};
Collections.sort(words, c);
```

  - Or this way:

```
List<String> words = Arrays.asList{“David”,”Adam”,”Eve”,”Moses”};
Collections.sort(words, (String s1,String s2)-> s1.compareTo(s2) *(-1));
```

  - Or this way – counting on Generics to evaluate s1, s2 types:

```
List<String> words = Arrays.asList{“David”,”Adam”,”Eve”,”Moses”};
Collections.sort(words, (s1,s2)-> s1.compareTo(s2) *(-1));
```

Java Language

- Lambdas and Functional Interfaces

```
public  int XXX (String s1, String s2)
```

  - Polymorphism isn't checked on dynamic invocation (only method signature)…
  - This means we don't even need to implement Comparator (!)

```
public class Things {

    public  int doThings(String s1, String s2){
        return s1.compareTo(s2)*(-1);
    }
}
```

  - But how exactly can we assign it to Collections.sort() method ?? In a moment…

Java Language

- Lambdas and Functional Interfaces
  - You may create custom abstractions based on functional interfaces

```
@FunctionalInterface
public interface IncrementByOne<T> {
        public int increment();
}
```

```
public class RaiseValues<T> {
    public void raise(IncrementByOne<? super T> entity){
        System.out.println(entity.increment());
    }
}
```

  - As we'll see later, any method with this signature : `public int XXX ()` can be assigned to RaiseValue.raise()

Java Language

- Lambdas and Functional Interfaces

- Important functional interfaces
  - Existing & enhanced interfaces
    - Runnable.run() : void
    - Comparator<T>.compare(T, T) : int
  - Java provides several new Functional Interfaces for other purposes:
    - Predicate<T>.test(T) : boolean
    - Consumer<T>.accept(T) : void
    - Supplier<T>.get() : T
    - Function<T,R>.apply(T) : R

Java Language

- Lambdas and Functional Interfaces

- Important functional interfaces
  - Predicate<T>.test(T) : boolean
    - Accepts T and calculate a boolean result. True = passed the test
  - Consumer<T>.accept(T) : void
    - Accept T and perform an operation. No result
  - Supplier<T>.get() : T
    - Produces T. Therefore accepts no parameters and returns T
  - Function<T,R>.apply(T) : R
    - Maps T value to R. Accepts T and calculate result R

Java Language

- Lambdas and Functional Interfaces
- Functional interfaces cont.
  - More concrete interfaces for working with primitives and references
    - IntPredicate.test(int) : boolean   Tests int value and result with boolean
    - IntConsumer.accept(int) : void   Accepts int
    - IntSupplier.getAsInt() : int   Produces int
    - IntFunction<R>.apply(int) : R   Accepts int and produces Result R
  - Same exists for Double
    - DoublePredicate, DoubleConsumer, DoubleSupplier, DoubleFunction
  - And for Long
    - LongPredicate, LongConsumer, LongSupplier, LongFunction

Java Language

- Lambdas and Functional Interfaces

- Functional interfaces cont.

  - Bi – functional interfaces that reduces two inputs, T and U, into one result
    - BiPredicate<T,U>.test(T,U) : boolean
    - BiConsumer<T,U>.accept(T, U) : void
    - BiFunction<T,U,R>.apply(T,U) : R

Java Language

- Extra ‘default’ methods:
  - Predicate<T>
    - and(Predicate <T>)
    - or (Predicate <T>)
  - Consumer<T>
    - andThen(Consumer<T> after) – creates a ‘chained’ consumer by appending ‘after’
  - Function
    - compose(Function<T> before) – creates a ‘chained’ function by placing ‘before’ at first
    - andThen(Function<T> after) – creates a ‘chained’ function by appending ‘after’

Java Language

- Extra ‘default’ methods:
  - Example:

```
int x=100;
int y=200;

Predicate<Integer> false1=(num)->num>100;
Predicate<Integer> true1=(num)->num>200;

Predicate<Integer> result1=false1.and(true1);
Predicate<Integer> result2=false1.or(true1);
Predicate<Integer> result3=false1.or(num->num%2==0);

System.out.println(result1.test(x));
System.out.println(result2.test(y));
System.out.println(result3.test(x));
```

```
Output :

false
true
true
```

Java Language

- Method reference
  - If we implement functional interface we use () -> { …. }
  - But if we got a ready to use implementation that matches with its signature ?
    - We would like to reference and assign it
    - We would like to reuse it
    - Whether static or not, we would like to reference it through its class
  - Well, this is how you reference methods:
    - Non-static:

```
SomeClass c=new SomeClass();
c :: someMethod
```

    - Static :

```
SomeClass :: someMethod
```

Java Language

- Method reference
  - Back to our examples: Collection.sort(List<String>,)

```
public  int XXX (String s1, String s2)
```

  - And we have this code, which doesn't implements Comparator but got a method with a matching signature to compare() named doThings():

```
public class Things {

    public  int doThings(String s1, String s2){
        return s1.compareTo(s2)*(-1);
    }
}
```

  - Lets use method referencing to compare with doThings():

```
List<String> words = Arrays.asList{"David","Adam","Eve","Moses"};
Things t=new Things();
Collections.sort(words, t::doThings);
```

Java Language

- Method reference
  - Now, with a static method: Collection.sort(List<String>,)

```
public  int XXX (String s1, String s2)
```

  - In this case doThings() is static:

```
public class Things {

    public  static int doThings(String s1, String s2){
        return s1.compareTo(s2)*(-1);
    }
}
```

  - We use method referencing without instantiating an object:

```
List<String> words = Arrays.asList{"David","Adam","Eve","Moses"};
Collections.sort(words, Things::doThings);
```

Java Language

- Method reference
  - Back to our custom functional interface example:

```
@FunctionalInterface
public interface IncrementByOne<T> {
        public int increment();
}
```

```
public class RaiseValues<T> {
    public void raise(IncrementByOne<? super T> entity){
        System.out.println(entity.increment());
    }
}
```

  - As mentioned, any method with this signature : can be assigned to RaiseValue.raise()

```
public  int XXX ()
```

Java Language

- Method reference
  - So, here is a Person implementation that contains such a method:

```
public class Person {
        private  String name;
        private int age;

        public Person(String name, int age) {
                this.name = name;
                this.age = age;
        }
        public String getName() {return name;}
        public void setName(String name) {this.name = name;}
        public int getAge() {return age;}
        public void setAge(int age) {this.age = age;}

        //NOTE: method signature matches IncrementByOne.increment() method
        public int birthday(){
                this.age+=1;
                return this.age;
        }
}
```

Java Language

- Method reference
  - Usage:

```
Person p = new Person("David",20);
RaiseValues<Person> calc=new RaiseValues<Person>();
calc.raise(p::birthday);
```

```
Output :   (since raise() prints increased value)

21
```

Java Language

- Method reference
  - What if we would like to assign constructors ?
    - Java provides a functional interface that may be good choice for default constructors
    - java.util.function.Supplier<T> with the method: T get()
    - Example:

```
public class Factory<T> {
	public T getObject(Supplier<T> s){
		return s.get();
	}
}
```

```
Factory<Person> factory=new Factory<Person>();
factory.getObject(Person::new);
```

Java Language

- Method reference
  - Method reference performs better than LAMBDAs

```
test(p->p.getAge>18)..

test(Person::isMature)..
```

  - But what if we need more complex and extensible calculations on our Person class?
    - Java Beans should be just value objects….
    - Solution: use Utility Classes

```
test(PersonUtils::isMature)..
```

```
public class PersonUtils{

  public static boolean isMature (Person p) {..}
   ...
}
```

Java Language

- Interface Default and Static Methods
  - Interface can have default implemented methods
    - Unlike abstract methods – default doesn't require overriding
    - Can be overridden in implementation classes

```
public interface Dimensional {
        int getWidth();
        int getHeight();
        default  int getArea(int width, int height){
                return width*height;
        }
        boolean resize() ;
```

Java Language

- Interface Default and Static Methods
  - Interface can also have implemented static methods

```
public interface PersonFactory{
    static  Person getPerson( Supplier<Person> s){
        return s.get();
    }
}
```

```
PersonFactory.getPerson(Person::new);
```

Java Language

- Interface Default and Static Methods
  - What about 'multiple inheritance' potential collisions ?
    - Given these 3 interfaces:

```
public interface Dimensional {
        ...
        default  int getArea(int w, int h){
                return w*h;
        } ...
```

```
public interface TriangleDimensional {
        ...
        default  int getArea(int w, int h){
                return (w*h)/2;
        }  ...
```

```
public interface AbstractDimensional {
     ...
     int getArea(int w, int h);   //abstract
     ...
```

Java Language

- Interface Default and Static Methods

```
public class MyShape1 implements Dimensional, TriangleDimensional{
        ???
}
```

  – Which method is taken ?
    - Answer is that NONE is taken. When you get default collision it fails to compile.

  – And in this case, does default methods override abstract methods ?

```
public class MyShape2 implements Dimensional, AbstractDimensional{
        ???
}
```

    - NO. abstract methods NEVER gets overridden by default, so it fails to compile
    - But – you may override the collided method

Java Language

- Interface Default and Static Methods
  - Means that this code will compile:

```
public class Rectangle implements Dimensional, AbstractDimensional {

    @Override
    public int getArea(int w, int h) {
        return w*h;
    }
    ....
```

    - When default and abstract methods collide – you must provide your own implementation
    - Changes to an existing interfaces might break code

Java Language

- Interface Default and Static Methods

  - Keep in mind that
    - Changes to an existing interfaces might break code
    - At least there is no scenario in which you are running different code than what you think..
    - This is because
      - Multiple default method collision will not compile
      - Default & abstract method collision will force you to override

Java Language

- Interface Default and Static Methods

  - What about multiple method Functional Interfaces ?
  - Functional Interfaces are meant to assign a method
  - So, only one method can be abstract
  - Others must be default or static
  - This is useful in cases where several methods are invoked in some order along with the assigned one

```
@FunctionalInterface
public  interface  CalculationFlow{
                  default void logTx(){…}
                  default  void recordTx(){….}
                  void calculate(Consumer<T> c)
}
```

Java Language

- Interface Default and Static Methods

  - Possible multiple method invokers:

  - When providing CalculatorFlow implementations both record() & log() may be overridden

```
public void nonRecordedCalc(CalculationFlow cf){
                cf.calc((v)->{........});
}

public void recordedCalc(CalculationFlow cf){
                ct.recordTX();
                cf.calc((v)->{........});
                cf.logTX();
}
```

Java Language

- Repeating annotations
  - In previous Java versions we cannot repeat annotations
    - For example if we have this Authors annotation that holds author names:

```
@Target(ElementType.TYPE)
@Retention(RetentionPolicy.RUNTIME)
public @interface Authors{
		String [] value();
}
```

    - We must use String [] in order to hold multiple author names:

```
@Authors({“authorA”, ”authorB”, ”authorC”})
public @interface ConcreteBook extends Book{
	…
}
```

Java Language

- Repeating annotations
  - In some complex configuration, when annotations contained in others, code is not that readable..

```
@MessageDriven (
        description="Provides logging services",
        activationConfig={
                @ActivationConfigProperty (propertyName="destinationType",
                                          propertyValue="javax.jms.Queue"),
                @ActivationConfigProperty (propertyName="destination",
                                          propertyValue="QueuejndiName")}
)
```

Java Language

- Repeating annotations
  - In Java 8 we may repeat annotations
    - So now we'll change Authors annotation to Author – with a single name
    - We have to denote the annotation with @repeatable that specifies its container annotation class

```
@Target(ElementType.TYPE)
@Retention(RetentionPolicy.RUNTIME)
@Repeatable(Authors.class)
public @interface Author{
        String  value();
}
```

    - Then, we create a container annotation for @Author

```
@Target(ElementType.TYPE)
@Retention(RetentionPolicy.RUNTIME)
public @interface Authors{
    Author[] value();
}
```

Java Language

- Repeating annotations
  - In Java 8 we may repeat annotations
    - And reuse it:

```
@Author("authorA")
@Author("authorB")
@Author("authorC")
public @interface ConcreteBook extends Book{
      ....
}
```

    - These annotations are available in their container annotation through reflection

Java Language

- Extended annotations
  - Annotations are placed in code according to its @Target
  - Java 5 provides target options in ElementType Enum:

ANNOTATION_TYPE	Annotation type declaration
CONSTRUCTOR	Constructor declaration
FIELD	Field declaration (includes enum constants)
LOVAL_VARIABLE	Local variable declaration
METHOD	Method declaration
PACKAGE	Package declaration
PARAMETER	Formal parameter declaration
TYPE	Class, interface (including annotation type), or enum declaration

  - Java 8 provides two new powerful options

Java Language

- Extended annotations
  - New ElementType

TYPE_PARAMETER	Generics declaration
TYPE_USE	new(), casting, implements clauses, throws clauses

  - Examples:

```
@Target(ElementType.TYPE_USE)
@Retention(RetentionPolicy.RUNTIME)
public @interface Log{
        String  value()   default “test”;
}
```

```
@Target(ElementType.TYPE_PARAMETER)
@Retention(RetentionPolicy.RUNTIME)
public @interface Alert{
        String  value()   default “alert”;
}
```

Java Language

- Extended annotations

  - Usage :

```
public void doIt (Object o)  throws @Log Exception  {
        if(o instanceof String){
             String  data = (@Log  String)o;
             List <@Alert  String> storage = new  @Log  ArrayList<>();
             ....
        }
}
```

```
public class ReverseComparator implements @Log Comparator <@Alert  String> {
  ....
}
```

Java Language

- Reflection – Parameter names

  - When reflecting a method there is no way of getting real parameter names
  - We get logical names like ‘arg0’, ‘arg1’ .. instead
  - In Java 8 we can get the actual parameter names
  - Use the same reflection code but launch with –parameters flag:

```
public class Example{
      public static void main(String[] args) throws Exception {
      Method method = Example.class.getMethod( "main” );
      for( final Parameter parameter: method.getParameters() ) {
         System.out.println( "Parameter: " + parameter.getName() );
      }
   }
}
```

Output without –parameters flag:

```
arg0
```

Output with –parameters flag:

```
args
```

Java Language

- Reflection – Parameter names
  - This feature is supported only when you set your JVM (-parameter flag)
  - Reason for that is that it reduces performance and makes reflection a bit heavier
  - Conclusion – don’t activate it unless it is critical for you to get this information

Java API's

- **Optional**

- Main goal is to clear our code from null checks
- java.util.Optional
- An object container that provides comfortable behavior regarding nulls & voids
  - Creating Optional

```
Optional<String> eValue=Optional.empty();
Optional<String> nValue=Optional.ofNullable(null);
Optional<String> value=Optional.of("OptionalData");
```

    - Empty Optional is a void container
    - Nullable Optional may contain value (T) or nulls
    - Other holds values (T)

Java API's

- **Optional**

- Basic behavior
  - get() – results in T or NoSuchElementException for empty & nullable Optional
  - orElse(T) – results in wrapped T if present, else return assigned T
  - orElseGet(Supplier<? extends T>) – results in T or executes Supplier<T>

```java
System.out.println(value.get());
System.out.println(nValue.orElse("else None"));
System.out.println(eValue.orElse("else Empty"));
//Supplier<String>
System.out.println(value.orElseGet(()->"else-get ???"));
System.out.println(nValue.orElseGet(()->"else-get None"));
System.out.println(eValue.orElseGet(()->"else-get Empty"));
```

Supplier<T> functional interface
get() : T

Output

OptionalData
else None
else Empty

OptionalData
else-get None
else-get Empty

Java API's

- **Optional**

- Basic behavior
  - isPresent() – results true if Optional wraps an object or false otherwise
  - ifPresent(Consumer<? super T>) – if object is present, executes Consumer<T>.accept(T) assigning wrapped object T and result with void

```
System.out.println(value.isPresent());
System.out.println(nValue.isPresent());
System.out.println(eValue.isPresent());

//Consumer
value.ifPresent((String s)->System.out.println(s+" is in the house"));
```

Consumer<T> functional interface
accept(T) : void

```
Output

true
false
false

OptionalData is in the house
```

Java API's

- **Optional**

- Primitives Optionals

  - OptionalDouble, OptionalInt, OptionalLong

  - Let have a look at OptionalDoube, others are just the same idea..
    - Everything turns to be double oriented:
    - of(double) – returns OptionalDouble wrapping the given value
    - get() becomes getAsDouble() and results with the wrapped double value or null
    - orElse(double) – return wrapped double if present. If not – returns the assigned double
    - orElseGet (DoubleSupplier) – same but if not present invokes getAsDouble() : double
    - ifPresent(DoubleConsumer) – if present, assigns value to accept(double) and returns void

Java API's

- **Optional**
- Complex behavior
  - Filter
    - Accepts Predicate <? Super T>
    - If present, performs Predicate<T>.test(T) and result with boolean
    - If present & test resulted with true – returns Optional<T> with origin value
    - If not present or test() result with false – returns Empty optional
  - Map
    - Mapping means we generate a result or null from an input
    - Optional wrapped object is the input
    - If present, Function<T,U>.apply(T) is executed and returns U. The result is Optional<U>
    - If not present or apply() results with null - returns Empty Optional
  - Both provides powerful actions on large data collections – we'll explore it later

Java API's

- **Optimistic Reading**
- Optimistic reading
  - When a reader obtains a non-exclusive lock on a resource it receives a stamp
  - On any writer update to the resource – the stamp gets updated
  - Readers may use their stamps in order to validate the resource
    - If the stamp is the same as the one in the source – no writers obtained any lock
    - Writers always update the resource stamp on completion
    - Stamps are much like 'version' in Hibernate/JPA
- StampedLock class
  - tryOptimisticRead()
    - used by readers in order to manage optimistic read locks on a resource
    - results with a non-zero stamp or zero if exclusively locked
  - validate(stamp)
    - Returns true if resource has the same stamp (means no other writer flushed data)

Java API's

- **Adders**
- Goal is to maintain counters in a multithreaded environment
- Available solutions:
  - Dirty counters
  - Synchronization
  - RWLock via Lock API
  - Volatile
  - Atomic concurrent

  - Most recent one is Atomic Concurrent
    - Conditional update prevents race conditions
    - Means that thread in a race might re-try to acquire a lock and update
    - In high contention the thread might re-try for a very long time....
    - **Therefore, useful for <u>small</u> amounts of threads or limited amount of writer threads**

Java API's

- **Adders**
- In low concurrencies performs just like Atomic Concurrent
- But in high concurrencies performs MUCH better:
  - LongAdder holds a collection of cells. Each cell behaves like an AtomicLong
  - Threads in race conditions do not re-try. <u>They place their update in a cell instead</u>
  - Cells are evaluated every time we call for final result
- Good for frequent updates in high concurrency systems

```
//initialized with 0 by default
  final LongAdder adder=new LongAdder();
  Runnable task1 = ()->{ adder.add((long)(Math.random()*10000));};
  Runnable task2 = ()->{ adder.increment();};
  Runnable task3 = ()->{ adder.decrement();};
  Runnable task4 = ()->{ adder.longValue();
```

Hands On - 1

- Create new project
- Define search tasks functional interface
- Provide a dynamic search platform
- Enrich with search utilities referenced methods

Java API’s

- Streams
- New java.util.stream API
- A sequence of elements supporting sequential and parallel aggregate operations
- Mostly Relevant for dealing with huge data grids
- Therefore obtained mainly from Collections, but not only
- Introduces real-world functional-style programming
- Since it is functional based - it is much faster

Java API’s

- Streams

- Streams perform lazily. Means the code is evaluated only if must be executed

- All manipulations are put in a stream pipeline that can be consumed, collected or manipulated again

- If multiple threads are used – each has its own pipeline which the stream gathers

- Spliterators are used behind the scene for breaking huge processing into small parts

- When processing completes – we are at the end of the stream. Cannot re-iterated

Java API's

- Streams

- Streams vs. Collections
  - Collections uses External iteration
    - Means you should handle elements in-between iterations
    - All operations are eagerly executed
  - Streams uses Internal iterations
    - Means that elements are handled by hidden holders in between iterations if & when needed
    - Operations are executed lazily – when terminal operation is triggered (later)

Java API's

- Streams

- Stateless – dynamic code that doesn't use any external allocations only local variables.
- Stateful – dynamic code that references external allocations

- Best is to go stateless
  - Stateful might be annoying just like when working with Inner Classes
  - Dynamic invocation is not part of the containing Class reflection and uses its own dedicated stack and lifetime – Just like Inner classes

Java API’s

- Streams
- Best in this case is to cover features through code
  - For the next examples we’ll use the following:
    - Address class – Java Bean with City enum & street attributes
    - Person class – Java Bean with name, age, Gender enum & Address
    - List<Person> populated with several thousands Person instances
    - We’ll do some cool manipulations on List<Person> with stream API

Java API’s

- Streams
- Example POJOs:

```
public class Person {
      private  String name;
      private int age;
      private Address address;
      private Gender gender;
      public Person(String name, int age, Gender gender, City city, String street) {
            this.name = name;
            this.age = age;
            this.gender=gender;
            address=new Address(city, street);
      }
      public String getName() {return name;}
      public void setName(String name) {this.name = name;}
      public int getAge() {return age;}
      public void setAge(int age) {this.age = age;}
      public Address getAddress() {return address;}
      public void setAddress(Address address) {this.address = address;}
      public Gender getGender() {return gender;}
      public void setGender(Gender gender) {this.gender = gender;}
      @Override
      public String toString() {return name + "-" + age;}
}
```

Java API's

- Streams
- Example POJOs & Enums:

```java
public enum City {
    A,B,C,D,E,F,G,H
}
```

```java
public enum Gender {
    M,F
}
```

```java
public class Address {
    private City city;
    private String street;

    public Address(City city, String street) {
        this.city = city;
        this.street = street;
    }
    public City getCity() {return city;}
    public void setCity(City city) {this.city = city;}
    public String getStreet() {return street;}
    public void setStreet(String street) {this.street = street;}
    @Override
    public String toString() {return city.toString();}
}
```

Java API's

- Streams
- Loading lots of persons to memory

```
List<Person> people = new ArrayList<>();
for (int i = 0; i < 10000; i++) {
        int cityInx = (int) (Math.random() * 8);
        int genderInx = (int) (Math.random() * 2);
        int letterInx = (int) (Math.random() * 24);
        people.add(new Person("!" + (char) ('a' + letterInx) + "!",
        (int) (Math.random() * 121), Gender.values()[genderInx],
        City.values()[cityInx], (char) i + ".St"));
}
```

Java API's

- Streams
  - Obtaining streams and parallel streams

```
List<Person> people = new ArrayList<>();
....
Stream<Person> stream=people.stream();
Stream<Person> parallel=people.parallelStream();
Stream<Person>.of(new Person(..),new Person(..),...);
```

  - Arrays utility class provides array based streams

```
int [] nums =new int[100000];
....
IntStream= Arrays.stream(nums)
```

Java API’s

- Streams
- Basic collective operations with streams
  - count() – counts elements in the stream
  - distinct() – returns stream of unique values (uses Object.equals())
  - empty() – returns an empty stream
  - findAny() – peeks (randomly) a value from the stream and returns it wrapped in Optional
  - findFirst() – does the same but always with the first element in the stream
  - limit(long maxSize) – returns a sub-stream from first element to maxSize
  - skip(long n) – returns a sub-stream starting from n to last element
  - sorted() – returns a stream of sorted element – sorts according to naturally ordering
  - sorted(Comparator<? super T> comparator) – does the same but sorts with compare(T t1,T t2)
  - iterator()

Java API's

- Streams
- Basic collective operations with streams
  - Simple example with array stream:

```
int [] nums =new int[100000];
for(int i=0;i<nums.length;i++){
	nums[i]=(int)(Math.random()*100000);
}

System.out.println(Arrays.toString(nums));
int[] sorted=Arrays.stream(nums).sorted().toArray();
System.out.println(Arrays.toString(sorted));
System.out.println(Arrays.stream(nums).average().getAsDouble());
```

Java API's

- Streams
- Basic collective operations with IntStreams, LongStreams & DoubleStreams
- Can save the headache when handling index loops with ints, longs and doubles
  - range(long s, long e) – returns a sub-stream from s to e (e is excluded)
  - rangeClosed(long s, long e) – returns a sub-stream from s to e (e is included)
  - iterate() – dynamically streams value based on previous value
    - Accepts initial value and applyAsInt(int):int dynamic method
    - Comes from IntUnaryOperator functional interface
    - Also available for long & double
    - Iterates infinitely - use limit()
  - generate() – dynamically generates a stream with a given Supplier
    - Generates infinitely – use limit()
  - limit(long maxSize) – provides a stream with the given maxSize

Java API's

- Streams
- Basic collective operations with streams
  - Simple example with int stream:

```
IntStream.range(1,4);
IntStream.rangeClosed(1,4);

IntStream ist= IntStream.of(1,2,3,4,5,6);
ist.limit(3);

IntStream.iterate(0, (i) -> i+2).limit(10);   //start with zero, increment index in 2
IntStream.generate(()->(int)(Math.random()*10).limit(100));
```

Stream contains: 1,2,3

Stream contains: 1,2,3,4

Stream contains: 1,2,3

Stream contains: 0,2,4,6,8,10,12,14,16,18

Java API's

- Streams
- Preventing infinite executions
  - When using iterate / generate something must stop value generation

```
IntStream.generate(()->(int)(Math.random()*10).count();
IntStream.generate(()->(int)(Math.random()*10).average();
IntStream.rangeClosed(1,4);

IntStream.generate(()->(int)(Math.random()*10).anyMatch(8));
IntStream.generate(()->(int)(Math.random()*10).limit(100));
```

Infinite execution….
Nothing stops value generation. Count & average are waiting for value generation to complete…

This is OK.
Since terminal operation is conditional – it stops generation when condition isn't met

Hands On - 2

- Use streams collective operations
- Examine streams & parallel streams performance

Java API’s

- Streams
- Match operations with streams

Predicate <T> functional interface
public boolean test (T)

  - allMatch(Predicate <? super T>) - returns true is all elements in the stream passes the test
  - anyMatch(Predicate <? super T>) – return true is one element in the stream passes the test
  - noneMatch(Predicate <? super T>) – return true is no element in the stream passes the test

```
List<Person> people = new ArrayList<>();
//fill with 10,000 person instances
....
System.out.println(people.stream().allMatch(p->p.getName().startsWith("!")));
System.out.println(people.stream().allMatch(p->p.getName().startsWith("!h")));
System.out.println(people.stream().anyMatch(p->p.getName().startsWith("!h")));
System.out.println(people.stream().anyMatch(p->p.getName().startsWith("!hx")));
```

```
Output

true
false
true
false
```

Java API's

- Streams
- Match operations with streams
  - Using predefined tests :

```
public class MaturePredicate{
    //could be a Person method as  well...
    public static boolean isMature(Person t) {
         if(t.getAge()>=18)return true;
         return false;
    }
}
```

```
List<Person> people = new ArrayList<>();
//fill with 10,000 person instances
....

System.out.println(people.stream().anyMatch(MaturePredicate::isMature));
System.out.println(people.stream().allMatch(MaturePredicate::isMature));
```

```
Output

true
false
```

Java API’s

- Streams
- Lazily executed stream operations:
  - filter(…)
  - map(…)
  - peek(…)
  - flatMap(…)
  - empty()
  - distinct()
  - limit(…)
  - skip(…)
  - reduce(…)
  - sorted(), sorted(…)

Java API's

- Streams - Lazily executed stream operations

- Think of that every time an intermediate operation is added to a stream it is actually being added to the stream pipeline
- The stream pipeline is executed when terminal operation is invoked
- Each element in the stream source is passed through the pipeline
  - Result contains only those who passed all the way
  - Element that fails is not evaluated along the rest of the pipeline
  - Single iteration over the stream source
- Pipelines iterate over stream sources and therefore cannot be reused

Java API's

- Streams - Lazily executed stream operations

- Interfering
  - Basically, stream sources (Lists, arrays, IO streams…) shouldn't change during streaming.
  - Non-concurrent sources will throw *ConcurrentModificationException*
  - It is possible to change concurrent sources (like *ConcurrentLinkedList*) during streaming
    - Concurrent collections uses Lock API in order to perform effective atomic locks when serving multiple threads

Java API's

- Streams - Lazily executed operations:
  - Lazy execution means that when using these methods nothing really gets iterated…
    - Like creating a 'view' – which is basically a sub-stream
  - Since each method results with a sub-stream – many operations can be chained without actually being evaluated:

```
stream.map(…).filter(…).limit(100)….
```

  - Iteration starts when finalizing the stream via
    - Some basic operations like count(), findAny()…
    - Collectors (later)

Java API’s

- Streams
- Filtering streams
  - Filters uses Predicate<T> as well but result with an updated stream
  - The updated stream contains all the elements that passed the test
  - filter(Predicate <? super T>) : Stream<T>

```
List<Person> people = new ArrayList<>();
//fill with 10,000 person instances
....
System.out.println("Females: "+people.stream().filter(p->p.getGender()==Gender.F).count());
System.out.println("Males: "+people.stream().filter(p->p.getGender()==Gender.M).count());
```

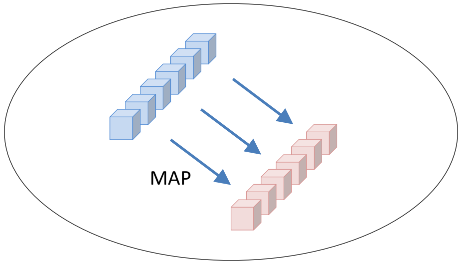
```
Output

Females: 5059
Males: 4941
```

Java API's

- Streams
- Mapping streams
  - Mapping means to take an input and generate a related result out of it
  - map() uses Function<T,R> with the apply method:
  - Function.apply(T) - returns R

  - Example – in order to calculate average age we need to
    map each person in the stream to its age collected in a new stream.
    than, having an integers stream, we may calculate average

Java API's



- Streams
- Mapping streams
  - So, Function.apply(T) - returns R
  - Means that a Stream<T> can be mapped to Stream<R>
  - Results are in the same order as T
  - Steams API provides the following Mappers:
    - mapToDouble – apply(T) : double, produces DoubleStream
    - mapToInt – apply(T) : int, produces IntStream
    - mapToLong – apply(T) : long, produces LongStream
    - map – apply(T) : R produces Stream<R>
  - Double, Int & Long Streams got additional collective methods:
    - average() – results in OptionalDouble/Int/Long
    - min(), max() – returns Optional<Double>/<Integer>/<Long>
    - boxed() – auto-box all primitives with their wrapper class
    - sum()

Java API's

- Streams
- Mapping streams
  - Let's calculate average age

```
System.out.println(people.stream().mapToDouble(p -> p.getAge()).average().getAsDouble());
```

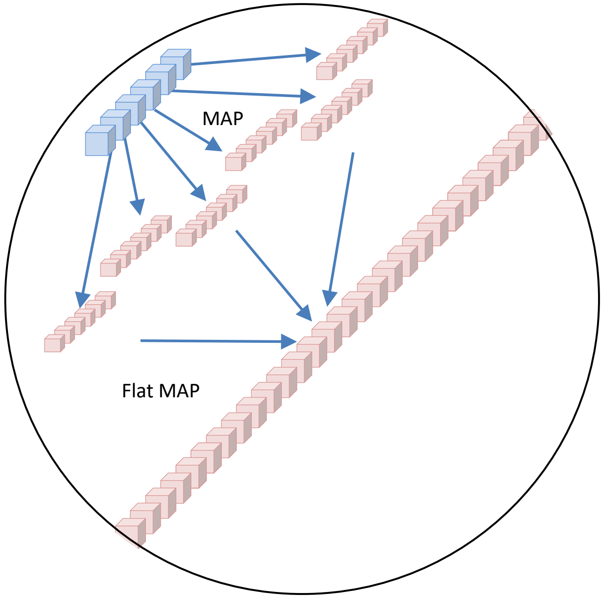
  - Now, let's print the highest age value among the males in the list:

```
System.out.println(people.stream().filter(p->p.getGender()==Gender.M)
                                  .mapToInt(p -> p.getAge()).max().getAsInt());
```

Java API's

- Streams
- Mapping streams
  - flatMap() uses Function<T,Stream<R>> with the apply method:
  - Function.apply(T) - returns Stream<R>
  - Each element result is flattened into one big Stream<R>

  - Steams API provides the following Mappers:
    - FlatMapToDouble – apply(T) : double, produces DoubleStream
    - FlatMapToInt – apply(T) : int, produces IntStream
    - FlatMapToLong – apply(T) : long, produces LongStream
    - FlatMap – apply(T) : R produces Stream<R>
  - Double, Int & Long Flat Streams got same additional collective methods



Java API's

- Streams
- Mapping streams
  - Here we gather people phone numbers
  - We assume each person holds a String array of phones available via getPhones() method
  - We end up with a phone (String) stream holding all phone numbers from all people

```
...people.stream().flatMap(p -> Stream<String>.of(p.getPhones()))...
```

```
...people.stream().flatMap(p -> p.getPhones().stream())...
```

Java API's

- Streams
- For Each with streams
  - For each means to perform operation for each element in the stream
  - forEach() takes a Consumer<T> and returns void
  - This means forEach() is EAGERLY executed
  - forEachOrdered() executes on each element in its original order in the stream
  - Here, we let all the babies (age < 2) introduce themselves:

```
people.stream().filter(p->p.getAge()<2).forEach(p->System.out.println("Hi I'm "+p.getName()));
```

Java API's

- Streams
- Peek with streams
  - Peek acts just like forEach(..) – but with one major difference:
  - peek() takes a Consumer<T> and returns a stream with the SAME elements
  - Consumer code is executed on each element on its way to the new stream
  - This is a big difference since peek() performs LAZILY
  - Here, we call a method on each person just before filtering…

```
people.stream().peek(p->System.out.println(p.sayHello()))…filter(…
```

Java API's

- Streams
- Reduce with streams
  - Merges elements into single result
  - Reduce() accepts:
    - T – which is the 'merged' value, starting with the first element
    - BiFunction<T,U,R> - U is current value to merge into T, R is the result of the merge
  - This method performs EAGERLY

  - Here, we sum letters of all person names

```
Stream<Integer>nameLengthStream = people.stream().map (p->p.getName().length());
Optional<Integer> sum = lengthStream.reduce((x, y) -> x + y); sum.ifPresent(System.out::println);
```

Hands On - 3

- Use streams matching operations
- Use filter, map and for-each

Java API’s

- Streams
- Streams Collectors
  - Collector generates a concrete result out of a stream
  - Collectors utility class factors Collector of various types:
    - Simple collection collectors
    - ‘Single Result’ collectors
    - Manipulated collection collectors

Java API's

- Streams
- Streams Collectors
  - Simple collection collectors simply returns stream elements in a new Collection
  - toList() – returns a List collector
  - toSet() – returns a Set collector

```
List<Person> babies= people.stream().filter(p->p.getAge()<2).collect(Collectors.toList());
System.out.println(babies.size());
```

Java API's

- Streams
- Streams Collectors
  - Simple collection collectors simply returns stream elements in a new Collection

  - toMap() – returns a Map collector
  - In this case each element is split to its logical key and value.
  - Keys must be unique !
  - Therefore, two mapping Function<T> must be provided - for both Key and Value
  - toMap(Function<? Super T, ? extends K >, Function<? Super T, ? extends V >)

```
Map<String,Address> babyAddresses=people.stream().filter(p->p.getAge()<2)
                                      .collect(Collectors.toMap(p->p.getID(), p->p.getAddress()));
```

Java API’s

- Streams
- Streams Collectors
  - Creating ‘Single Result’ Collectors
  - ‘Single Result’ collectors
    - Actually reduces the stream
    - Executes Function on each element and than fuses the outcome into single result
    - Counting collector is the simplest – simply counts elements

Java API's

- Streams
- Streams Collectors
  - Creating 'Single Result' Collectors – occurs when resulting in Collector sum value
    All these methods return Collector.sum() which is the calculated result wrapped in:
    - averagingDouble (ToDoubleFunction< ? super T >) – results in Double value
    - averagingInt (ToIntFunction< ? super T >) – results in Integer value
    - averagingLong (ToLongFunction< ? super T >) – results in Long value
    - summingInt(), summingLong(), summingDouble()…

    In order to simply count elements:
    - counting() – results in long value which is the element count
  - In this example we calculate the average age of all young persons (age<18)

```
System.out.println(people.stream().filter(p->p.getAge()<18).
                    collect(Collectors.averagingDouble(p->p.getAge())).doubleValue());
```

Java API’s

- Streams
- Streams Collectors
  - Creating ‘Single Result’ Collectors – occurs when resulting in Collector sum value
    When summarizing we get much more detailed information
    - Result is not Collator.sum() simple value – but SummaryStatistics instead
    - summarizingDouble (…) – returns DoubleSummaryStatistics
    - summarizingInt (…) - returns IntegerSummaryStatistics
    - summarizingLong (…) - returns LongSummaryStatistics

  - SummaryStatistics methods:
      - accept(T) – records new value to the statistics
      - getAverage()
      - getCount()
      - getMax()
      - getMin()
      - getSum()

Java API's

- Streams
- Streams Collectors
  - Creating Manipulated Collection Collectors
  - These collectors performs aggressive operation while collecting elements into collections:
    - Grouping by
    - Grouping by concurrent – works with ConcurrentMaps
    - Partitioning by
    - Joining

Java API's

- Streams
- Streams Collectors
  - Grouping by means generating a Map out of a Stream<T>
  - A Function<T,K> calculates the key from each element
  - Elements with identical keys are grouped into List / Set / Map Collectors
  - Eventually, each Key is paired with a list of grouped elements - List<T>

  - In this example we filter all persons which are 120 years old and group them by city:

```
Map<City, List<Person>> groups = people.stream().filter(p->p.getAge()==120)
                                        .collect(Collectors.groupingBy(p->p.getAddress().getCity(),Collectors.toList()));
System.out.println(groups);
```

Java API's

- Streams
- Streams Collectors
  - As mentioned, grouping by generates a Map in which a key is paired with a list of grouped elements - List<T>
  - We may group by assigning 'single result' collectors as well:
  - In this example we filter all persons which are 120 years old and count how many are living in each city:

```
Map<City, Long> groups = people.stream().filter(p->p.getAge()==120)
                    .collect(Collectors.groupingBy(p->p.getAddress().getCity(),Collectors.counting()));
System.out.println(groups);
```

```
Output:

{F=12, H=15, E=9, A=14, D=16, B=8, G=10, C=8}
```

Java API's

- Streams
- Streams Collectors
  - More group by examples:
  - Showing how many persons live in each city:

```
Map<City, Long> groups = people.stream().collect(
                Collectors.groupingBy(p->p.getAddress().getCity(), Collectors.counting()));
System.out.println(groups);
```

```
Output:
{A=1321, B=1272, C=1193, D=1257, E=1249, F=1204, G=1275, H=1229}
```

  - Here we show average ages by gender:

```
Map<Gender, Double> groups = people.parallelStream().collect(
                Collectors.groupingBy(Person::getGender,
                Collectors.averagingInt(Person::getAge))));
System.out.println(groups);
```

```
Output:
{F=60.81089743589744, M=55.755711775043935}
```

Java API's

- Streams
- Streams Collectors
  - Partitioning means creating a true/false key Map from a given stream
  - partitioningBy(Predicate<? super T>) results with a Map<Boolean,List<T>>
  - False key – holds a List<T> with elements that failed to pass the test
  - True key – holds a List<T> with elements that passes the test
  - Here we divide persons from city 'A' into 2 groups:
    - <u>false</u> - younger than 60
    - <u>true</u> – all the rest:

```
Map<Boolean,List<Person>> part=people.stream().filter(p -> p.getAddress().getCity().equals(City.A))
                                                 .collect(Collectors.partitioningBy(p->p.getAge()>60));
```

Java API's

- Streams
- Streams Collectors
  - Concatenates the input elements
  - Elements must be of type String
  - Collectors.joining()
  - A delimiter can be placed between each element: Collection.joining(String delimiter)

  - Example of creating a long String with all person names :

```
System.out.println(people.stream().map(Person::getName).collect(Collectors.joining()));
```

```
Output:
!s!!o!!g!!e!!t!!n!!b!!g!!g!!j!!d!!r!!j!!i!!n!!b!!t!!b!!o!!b!!t!!e!!j!....
```

Java API’s

- Streams
- Reducing Streams
  - Reducing means to fuse stream elements into single result
  - reduce() accepts reduced value U, and current element T
  - Reduce results is in fact an updated U value
  - Updated U value is delegated to next element in the stream and so on..
  - When reducing is started with initial value, result is U
  - When reducing is started without it – result is Optional<U> - for null value outcome
  - Mapping all person names and reducing into collection made of 2nd letter in each:
    - a is current reduced value
    - b is current element in the stream
    - b.name 2nd letter is added to a if not present already

```
System.out.println(people.stream().map(p -> p.getName())
            .reduce("", (a, b) -> {
                    if(a.indexOf(b.charAt(1))==-1)
                        a += b.charAt(1);
                        return a;
}));
```

Output:

{ajtbwhildxnrcspmuekgvfoq}

Hands On - 4

- Use streams collectors
- Use groupingBy and partitionBy
- Use reduce on streams

Java API's

- Parallel Streams
  - Fork-Join
    - All parallel stream share the same ForkJoinPool by default
    - The default ForkJoinPool uses a number of threads equals to the number of available processors -1
    - Fork-Join on stream includes:
      - Create a thread pool
      - Splitting the stream
      - Assigning each part to a thread consumed from the pool
      - Computing
      - Gathering results

Java API's

- Parallel Streams
  - Pros & Cons
    - Parallelism is effective when:
      - There are limited number of threads &
      - Tasks are blocking for a long time
    - When processing intensive requests
      - We usually count on JVM to do the parallelism
      - Adding another business logic parallelism will cause slower execution
      - Parallel execution performance might vary dramatically due to other processing on the hosting machine

Java API's

- Parallel Streams
  - When collecting parallel streams – use concurrent collections when possible

```
people.parallelStream().filter(p->p.getAge()>18).collect(Collectors.groupingByConcurrent(Person::getAge));
```

Java API’s

- Parallel Streams
  - Assigning dedicated pools to streams
    - Good for not sharing the default pool if too occupied

```
List<String> list = ……// A list of Strings
Stream<Integer> stream = list.parallelStream().map(String::length);

//we pause here and turn the stream into a Callable task.
//collect() eagerly starts the iteration and map Strings to their lengths:
Callable<List<Integer>> c = () -> stream.collect(Collectors.toList());
ForkJoinTask task=ForkJoinTask.adapt(c);

//now, we create a NEW thread pool and assign the task to it
ForkJoinPool forkJoinPool = new ForkJoinPool(4);
List<Integer> lengths = forkJoinPool.submit(task).get();
```

Java API’s

- CompletionStage<T> interface
  - Used for breaking tasks into conditional continues sub-tasks

- CompletableFuture
  - Implements CompletableStage

  - Provides operations for creating sub-tasks piplines
  - Supports LAMBDA exp.

Java API’s

- CompletableFuture
-
  - Provides operations like:
    - thenAccept(Consumer<? super T> action)
    - supplyAsync(Supplier<U> supplier)
    - thenApply(Function<? super T,? extends U> fn)
    - thenRun(Runnable action)

    - All result with CompletionStage<T> - so pipelines may be easily codes
    - All may use current thread or obtain diffirent one from given executor - asyncXXX()
    - All may be executed on different executors than ForkJoinPool..commonPool()

Java API's

- CompletableFuture

  – Examples:

```
Method ref pipeline:
CompletableFuture.supplyAsync(me::findRemoteControl)
                        .thenApply(codi::searchMovie)
                        .thenAccept(me::watchMovie);

LMBDA Exp. Pipline:
CompletableFuture.supplyAsync(()->me.findRemoteControl(“under the sofa”))
                        .thenApply((rc)-> codi.favorateChannel(rc,”Pulp Fiction”)
                        .thenAccept((mov)->me.watchMovie(mov));
```

  – Methods relates to other CompletableStage<T> instances:

    - acceptEither(CompletionStage<? extends T> other, Consumer<? super T> action)
    - applyToEither(CompletionStage<? extends T> other, Function<? super T,U> fn)
    - runAfterBoth(CompletionStage<?> other, Runnable action)
    - runAfterEither(CompletionStage<?> other, Runnable action)

Java API's

- Date/Time API
- JSR – 310
- Java can do much better that Date & Calendar
- Meet the new fellows in town:
  - Clock
  - ZoneID
  - ZonedDateTime
  - LocalDate
  - LocalTime
  - LocalDateTime
  - Duration & Period
  - ChronoLocalDate
  - ChronoUnit
- java.time is the parent package

Java API’s

- Date/Time API
- Clock

- Clock provides access to the current instant, date and time
- Can be used instead of *System.currentTimeMillis()* and *TimeZone.getDefault()*
- Clock shows you the time according to your time zone
- ZonedDateTime also provides zone-id based timestamp – but with different features

```
Clock clock=Clock.systemUTC();
System.out.println(clock.millis());
System.out.println(clock.instant());
```

```
Output:

1434478395137
2015-06-16T18:13:15.137Z
```

Java API's

- Date/Time API
- ZoneID
- A container of all supported time zones in Java
- Israel is also included: "Israel"
- Regions with more specific time zones uses this format: "America/Los_Angeles"

```
System.out.println(ZoneId.getAvailableZoneIds());
System.out.println(ZoneId.systemDefault());
```

Output:

```
[Asia/Aden, America/Cuiaba, Etc/GMT+9, Etc/GMT+8, Africa/Nairobi, America/Marigot....
Europe/Athens
```

Java API’s

- Date/Time API
- LocalDate, LocalTime, LocalDateTime
- Holds local date, time & timestamp respectively
- Can be created with clock and by using static now() method

```
LocalDate ld=LocalDate.now();
System.out.println(ld);
LocalTime lt=LocalTime.now();
System.out.println(lt);
LocalTime ltc=LocalTime.now(clock);
System.out.println(ltc);
LocalDateTime ldt=LocalDateTime.now();
System.out.println(ldt);
```

```
Output:

2015-06-17
12:40:40.143
09:40:40.143
2015-06-17T12:40:40.143
```

Java API's

- Date/Time API
- Duration and Period
- Both implement *TemporalAmount*
- Period is for these time units: days, weeks, months, years
- Duration is for nanos, millis, seconds, minutes, hours, days
- LocalTime, LocalDate and ZonedDateTime can be changed by *TemporalAmount*
  - *Got plus() & minus() methods*
- Note: Periods are too big for LocalTime (UnsupportedTemporalTypeException)

```
Period fiveDays=Period.ofDays(5);
Duration fewNanos=Duration.ofNanos(29999911);
System.out.println(ltc.plus(fewNanos));
System.out.println(ld.plus(fiveDays));
```

```
Output:

18:31:12.795999911
2015-06-21
```

Java API's

- Date/Time API
- Duration and Period

- Duration can be converted into other units via dedicated methods
- Duration supported converts:
  - toDays(), toHours(), toMinutes(), toMillis(), toNanos()

```
Duration fewNanos=Duration.ofNanos(29999911);
System.out.println(fewNanos.toMillis());
System.out.println(fewNanos.toMinutes());
```

```
Output:

29
0
```

Java API's

- Date/Time API
- Duration and Period

- Use methods *plus(TemporalAmount)* , *minus(TemporalAmmount)* to roll date & time
- LocalDate got specific plus/minusXXX(int) for day, week, month & year
- LocalTime got specific plus/minusXXX(long) for nano, milli, sec, min, hour & day
- ZonedDateTime and LocalDateTime got all combinations

```
LocalDate yesteday=LocalDate.now().minusDays(1);
LocalDate tomorrow=LocalDate.now().plusDays(1);
System.out.println("yesterday: "+yesteday);
System.out.println("tomorrow: "+tomorrow);
```

Output:

```
yesterday: 2015-06-16
tomorrow: 2015-06-18
```

Java API's

- Date/Time API
- ChronoLocalDate
- A date without time-of-day or time-zone in an arbitrary chronology
- Intended for advanced globalization use cases – like gaming
- Prefer LocalDate
  - Abstracting local calendar system is usually the wrong approach
  - Resulting in logic errors and hard to find bugs
  - As such, it should be considered an application-wide architectural decision to choose to use this interface as opposed to LocalDate.

```
ChronoLocalDate chronoNow=ChronoLocalDate.from(LocalDate.now());
System.out.println(chronoNow);
LocalDate yesteday=LocalDate.now().minusDays(1);
LocalDate tomorrow=LocalDate.now().plusDays(1);
System.out.println(yesteday.isAfter(chronoNow));
System.out.println(tomorrow.isAfter(chronoNow));
System.out.println(yesteday.isBefore(chronoNow));
System.out.println(tomorrow.isBefore(chronoNow));
```

Output:

```
2015-06-17
false
true
true
false
```

Java API’s

- Date/Time API
- ChronoUnit
- A handy Enum that provides unit based operations
- Defines the units used to measure time
  - CENTURIES, DECADES, MILLENNIA, ERAS, FOREVER
  - HALF_DAYS, DAYS, WEEKS, MONTHS, YEARS
  - SECONDS, MINUTES, HOURS
  - MILLIS , MICROS, NONOS
- Main operations:
  - getDuration() – each unit has a different duration
  - between() – returns the number of units between to Dates/Times (may be from different time zones)

Java API's

- Date/Time API
- ChronoUnit
- getDuration() Example:

```
System.out.println(ChronoUnit.DAYS.getDuration());  //enum's default duration unit
System.out.println(ChronoUnit.DAYS.getDuration().toDays());
System.out.println(ChronoUnit.HOURS.getDuration()); //enum's default duration unit
System.out.println(ChronoUnit.HOURS.getDuration().toMinutes());
System.out.println(ChronoUnit.SECONDS.getDuration().toMillis());
System.out.println(ChronoUnit.SECONDS.getDuration().toNanos());
System.out.println(ChronoUnit.MILLENNIA.getDuration().toHours());
System.out.println(ChronoUnit.MILLENNIA.getDuration().toDays());
```

Output:

```
PT24H    //default form
1
PT1H     //default form
60
1000
1000000000
8765820
365242
```

Java API's

- Date/Time API
- ChronoUnit
- between() Example:

```
LocalDateTime now=LocalDateTime.now();
LocalDateTime future=now.plus(Period.ofDays(328));
future.plus(Duration.ofHours(56));
System.out.println(ChronoUnit.DAYS.between(now,future));
System.out.println(ChronoUnit.HOURS.between(now,future));
System.out.println(ChronoUnit.SECONDS.between(now,future));
System.out.println(ChronoUnit.MILLENNIA.between(now,future));
```

Output:

```
328
7872
28339200
0
```

Java API’s

- Nashorn JavaScript engine

- Java 8 comes with a new implementation of javax.script.ScriptEngine

- Nashorn engine
  - Developed by Oracle
  - Capable of running standalone Java Script
  - Is default
  - Accepts *.js

Java API's

- Nashorn JavaScript engine
- Example:

```
public static void main(String[] args) {
        ScriptEngineManager manager = new ScriptEngineManager();
        ScriptEngine engine = manager.getEngineByName( "JavaScript" );
        System.out.println( engine.getClass().getName() );
        try {
            System.out.println( "Result:" + engine.eval(
                "function f(name) {
                        return 'hello '+name;
                };
                f('david‘);"
            ));
        } catch (ScriptException e) {
            e.printStackTrace();
        }
}
```

```
Output:
jdk.nashorn.api.scripting.NashornScriptEngine
Result:hello david
```

Java API’s

- Nashorn JavaScript engine
- Nashorn engine can be populated with a given JS files or streams as well
  - Accepts *.js

```
public static void main(String[] args) {
        ScriptEngineManager manager = new ScriptEngineManager();
        ScriptEngine engine = manager.getEngineByName( "JavaScript" );
        System.out.println( engine.getClass().getName() );
        try {
            System.out.println( "Result:" + engine.eval(
                new FileReader("myScript.js")
            ));
        } catch (ScriptException e) {
            e.printStackTrace();
        }
}
```

Java API's

- Base64

- Java 8 support Base64 encoding and decoding
- Base64
  - Binary to text encoding standard
  - Useful when passing binary data over textual protocols like HTTP
  - Uses ASCII format text representation

```
public static void main(String[] args) {
        String text = "This test is Base64 encoded and decoded !";
        System.out.println(text);
        byte[] bin=text.getBytes(StandardCharsets.UTF_8);
        //encode
        String encoded = Base64.getEncoder().encodeToString(bin);
        System.out.println( encoded );
        //decode
        String decoded = new String(Base64.getDecoder().decode( encoded ),
        StandardCharsets.UTF_8 );
        System.out.println( decoded );
}
```

```
Output:
This test is Base64 encoded and decoded !
VGhpcyB0ZXN0IGlzIEJhc2U2NCBlbmNvZGVkIGFuZCBkZWNvZGVkICE=
This test is Base64 encoded and decoded !
```

Java API's

- Parallel Arrays

- We already covered Arrays utility class stream support
- Arrays also offers powerful parallel operations that can be performed on arrays
- Both uses Fork-Join API – splits the array into smaller parallel processed tasks
  - parallelSetAll
    - fills an array with a calculated number
    - parallelSetAll(XXX[] , IntToXXXFunction)
    - Function accepts index and results with value
  - parallelSort
    - Sorts the given array according to naturally ordering

```
int[] nums=new int[100000];
Arrays.parallelSetAll(nums, index->ThreadLocalRandom.current().nextInt(100000));
System.out.println(Arrays.toString(nums));
Arrays.parallelSort(nums);
System.out.println(Arrays.toString(nums));
```

New Java Tools

- Nashorn JavaScript engine utility
- Class dependency analyzer

New Java Tools

- Nashorn JavaScript engine utility
  - Java based engine which can also be activated as a utility
  - jjs.exe found in your jdk/bin directory is used to run scripts
  - Assume we have this test.js file:

```
function f(name) {
    return 'hello '+name;
};
f('david‘);
```

  - In order to launch it do this in your prompt:

```
jjs  test.js
```

New Java Tools

- Class dependency analyzer
  - Java 8 comes with a great utility that check jar dependencies
  - jdeps.exe found in your JDK/bin directory
  - Accepts a given jar
  - Prints out its dependencies
  - If some dependencies are missing in your classpath – you'll see 'not fount' next to it

New Java Tools

- Class dependency analyzer
  - Lets examine the following scenario:
  - In projectA we have this A.class:

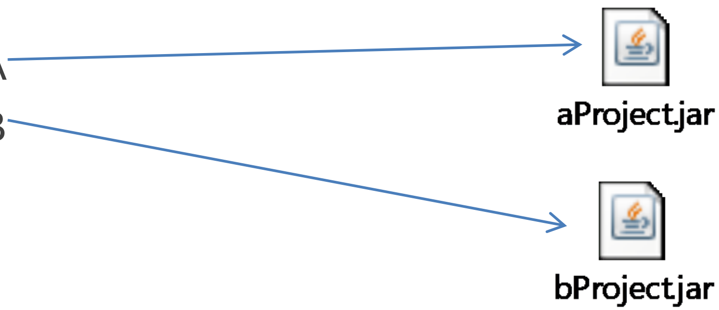
```
package aaa;

public class A {
    public void print(){
        System.out.println("A");
    }
}
```

  - In projectB we got B.class that uses A type:
  - Classpath dependencies were set by linking the 2 projects in our IDE so B can get compiled

```
package aaa;

public class B {
    public static String main(String [] args){
        A obj=new A();
        obj.print();
    }
}
```

New Java Tools

- Class dependency analyzer
  - Now, we export the two projects into 2 separated jars:
  - projectA
  - projectB



  - Note : aProject.jar is an independent jar that uses JSE libraries
    bProject is broken since it needs aProject.jar in its classpath

New Java Tools

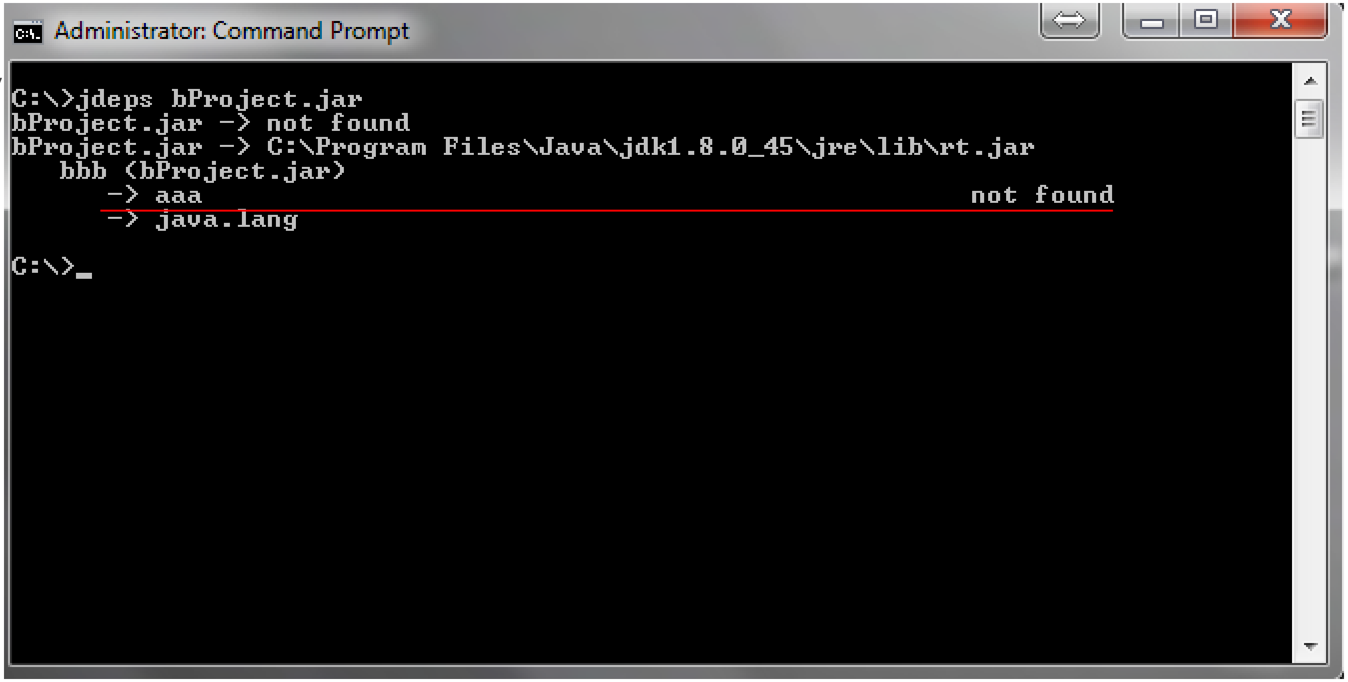
- Class dependency analyzer
  - When a client receives aProject.jar and check its dependencies – everything is fine:

Administrator: Command Prompt

```
C:\>jdeps aProject.jar
aProject.jar -> C:\Program Files\Java\jdk1.8.0_45\jre\lib\rt.jar
   aaa (aProject.jar)
      -> java.io
      -> java.lang

C:\>
```

New Java Tools

- Class dependency analyzer
  - But when doing the same thing with bProject.jar we can clearly see the broken dependency:
  - Jdeps may use –cp to include classpath in the check

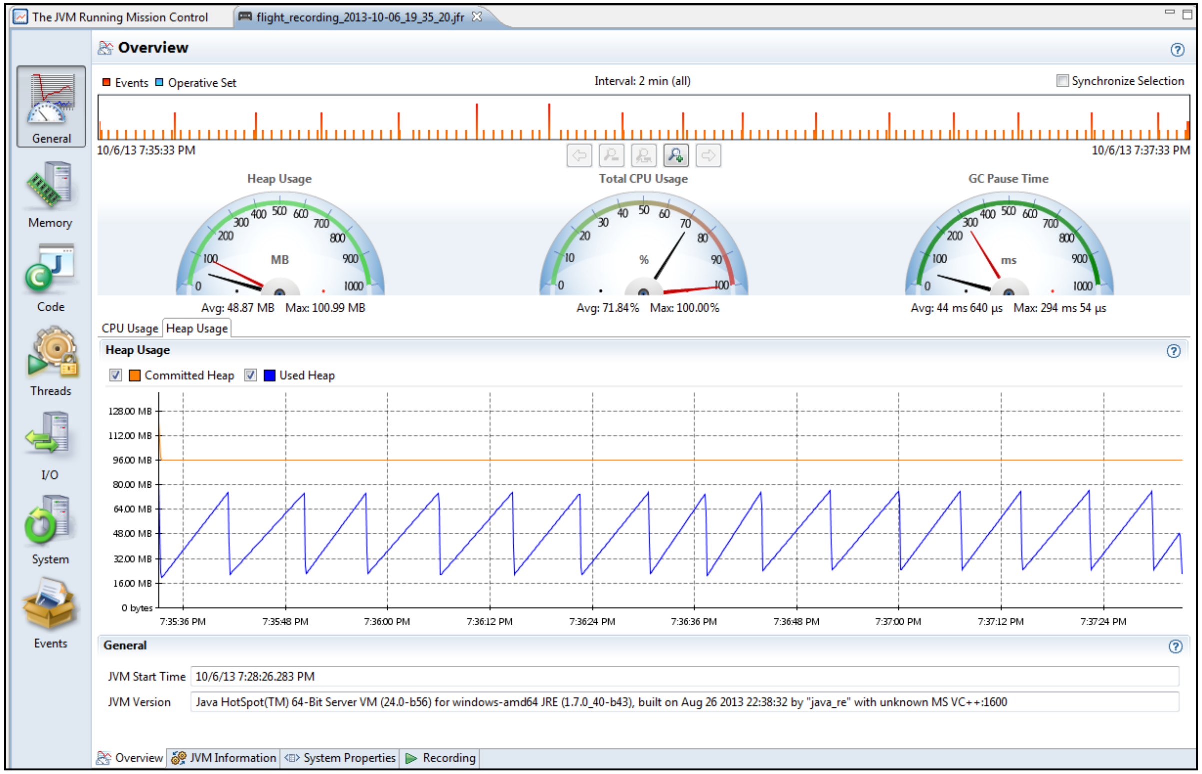


New Java Tools

- Java Mission Control
  - Available in the Oracle JDK since Java 7u40
  - Originates from JRockit JVM
  - Used for 2 main purposes:
    - Monitoring the state of multiple running Oracle JVMs
    - Java Flight Recorder dump file analysis
  - Built on JVM JMX Server
  - JAVA_HOME\bin\jmc.exe

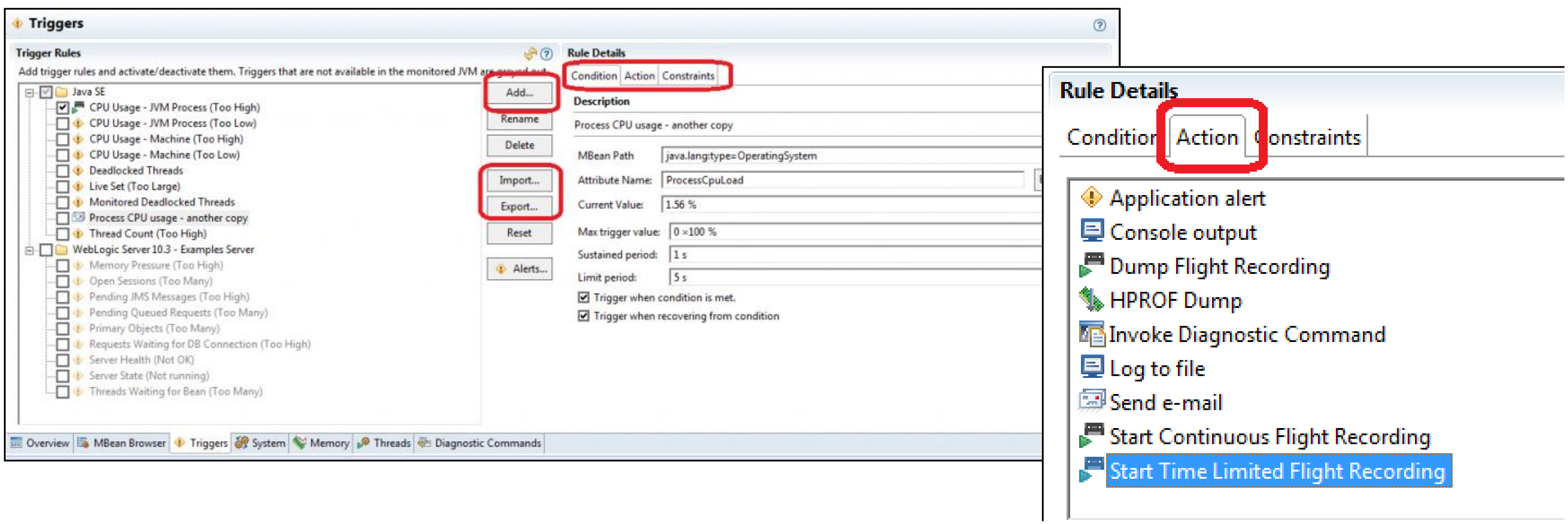
New Java Tools

- Java Mission Control
  - Real time processing
    - Heap
    - CPU
    - GC Pause time



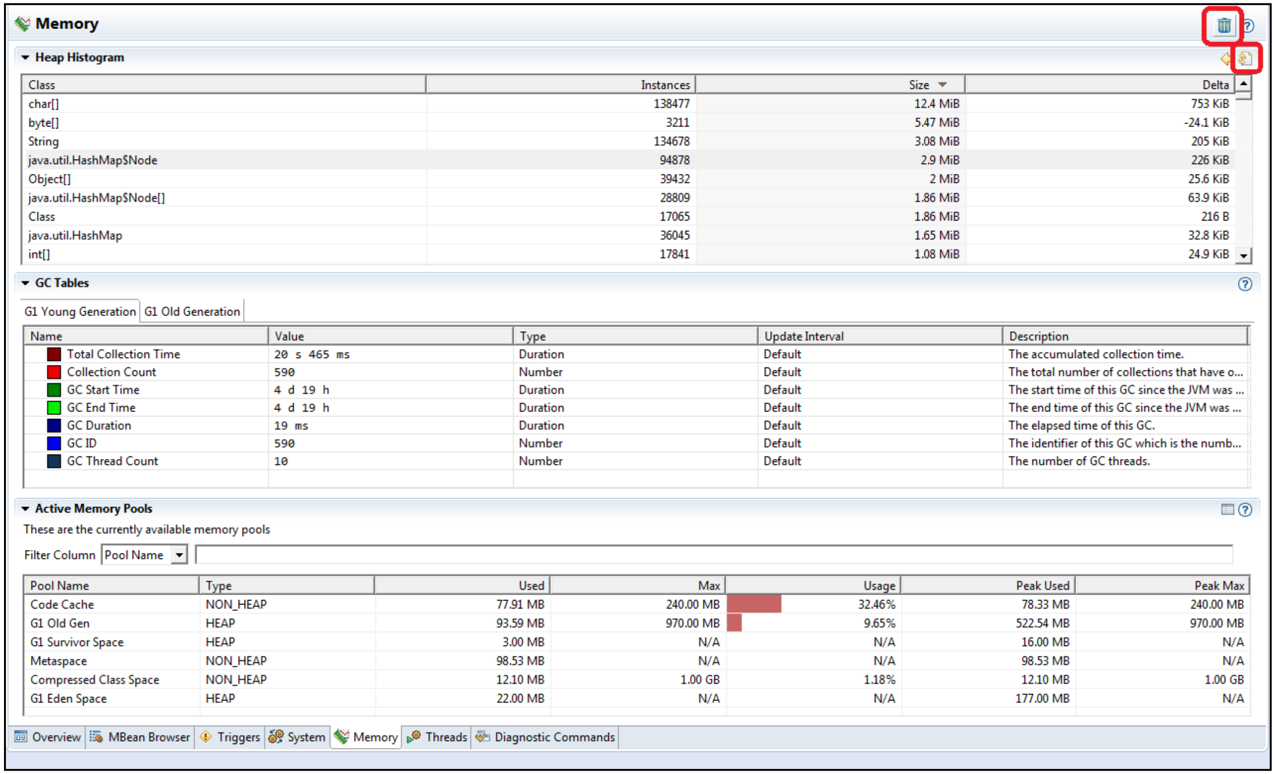
New Java Tools

- Java Mission Control
  - Event triggers
    - Act when JMX counter exceeds any pre-set limit in any given time



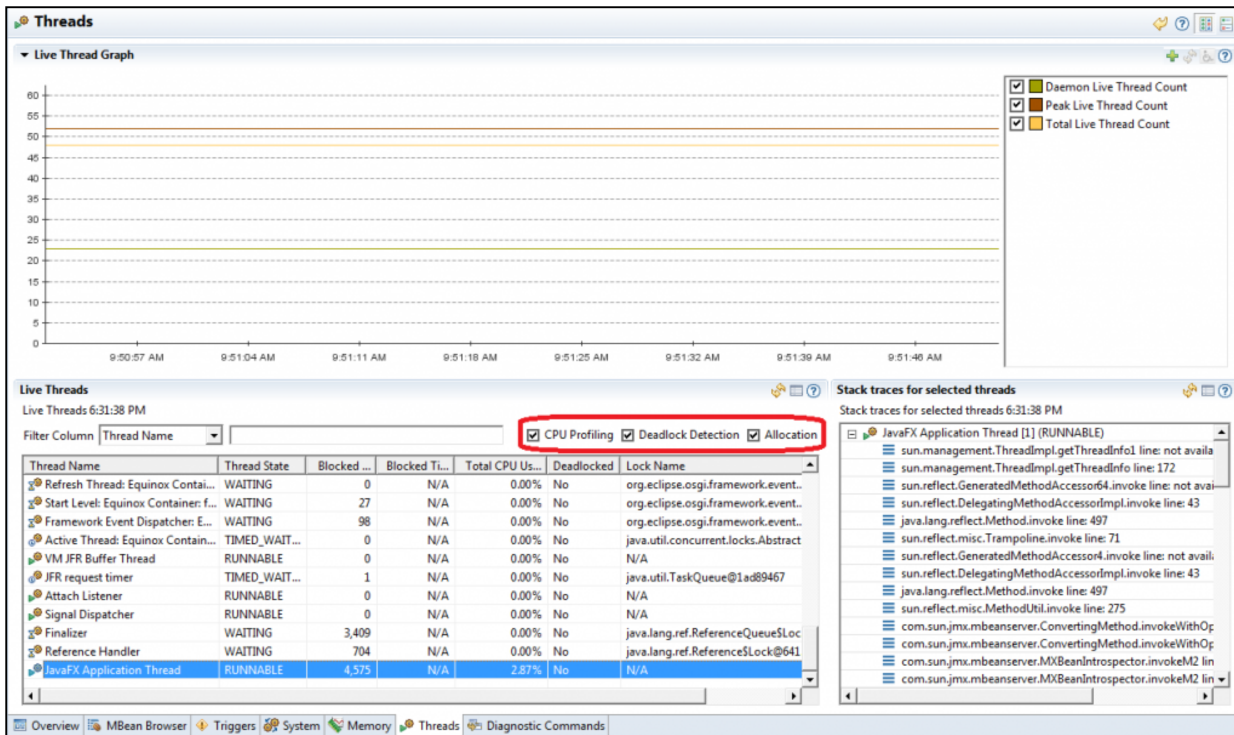
New Java Tools

- Java Mission Control
  - Memory tab
    - Heap generation
    - Minor & full GC Info



New Java Tools

- Java Mission Control
  - Threads tab



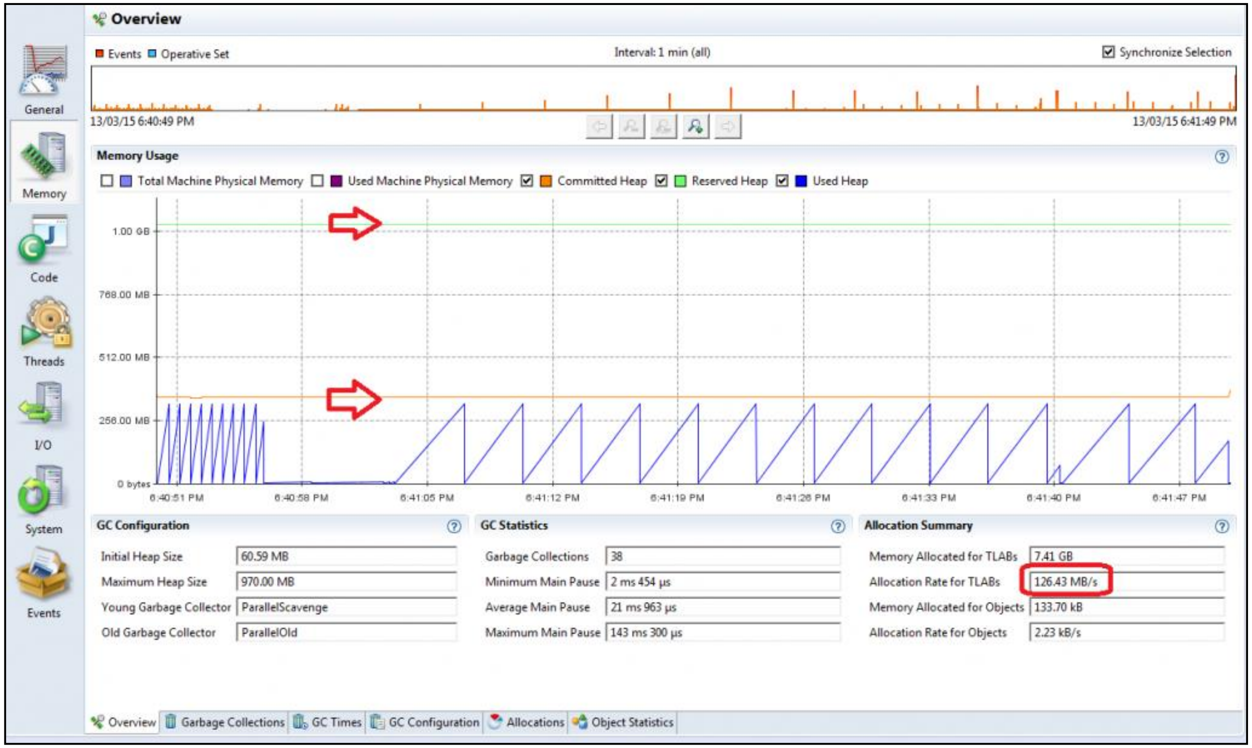
- Thread state
- Lock name
- Deadlock
- Blocked count
- Per thread CPU usage
- Amount of memory allocated by a given thread since it was started

New Java Tools

- Java Mission Control
  - Flight recorder - Initial screen
    - View JVM args
    - You may set time range for analyzing

New Java Tools

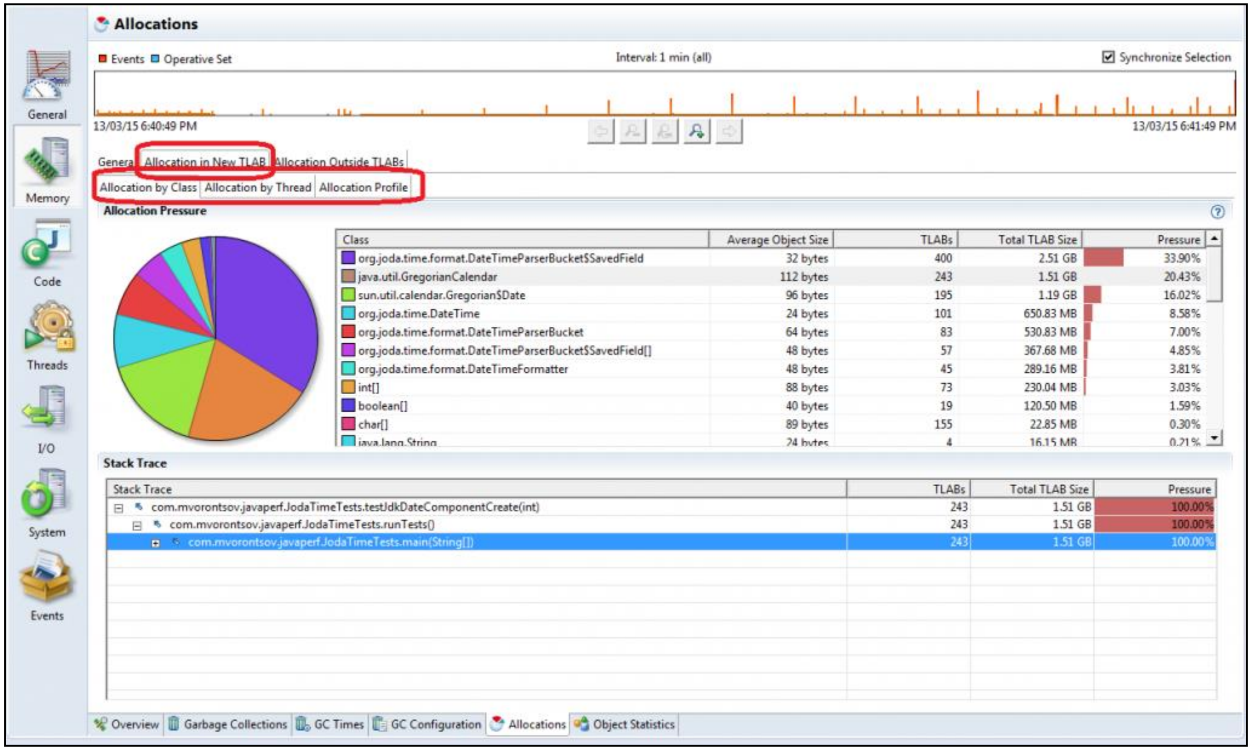
- Java Mission Control
  - – Flight recorder – memory tab



- Machine RAM
- Java heap usage
- Garbage collections – when, why, for how long and how much space was cleaned up.
- Memory allocation – inside / outside TLAB, by class/thread/stack trace.
- Heap snapshot – number / amount of memory occupied by class name

New Java Tools

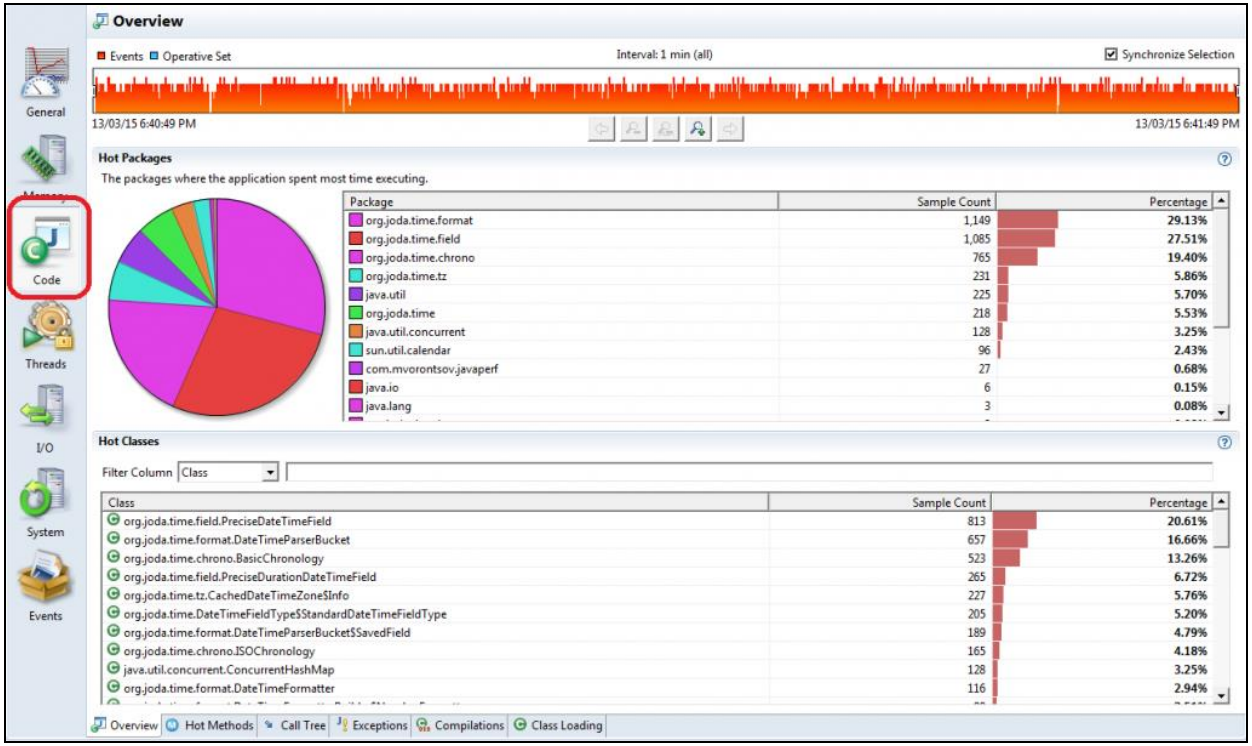
- Java Mission Control
  - Flight recorder – allocation tab



- Object allocations
- By class
- By thread

New Java Tools

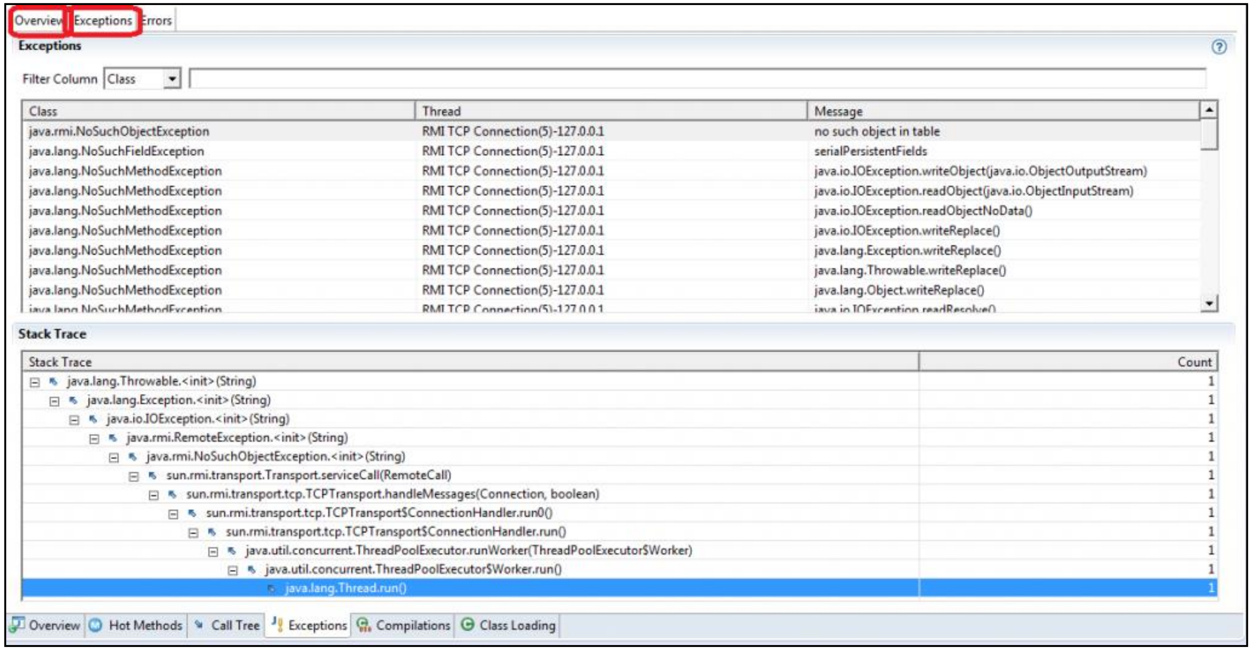
- Java Mission Control
  - – Flight recorder – code tab



- Method hotspots
- Packages hotspots
- Useful for CPU optimization

New Java Tools

- Java Mission Control
  - Other tabs
    - Exceptions
    - I/O
    - Threads



New Features In Java Runtime

- Java provides two changes in its runtime but both are very important
- Main goals are:
  - Reducing JVM footprint
  - Make JVM a faster plugin

  - Java wants to be everywhere…
  - We can see by now that
    - Stream API allows effective manipulations on huge data grids
    - This is great for cloud parallel computing and big data processing

  - But what about the emerging IoT ?
    - Java also gets ready to be ‘embedded’ in the tiny future internet clients

New Features In Java Runtime

- Feature 1 - Smaller VM
  - Up to Java 7 footprints averages are
    - 6Mb for –client
    - 9Mb for –server
  - In Java 8 JVM shouldn't use more than 3Mb on startup
  - Smaller plugins are better for small clients….
  - How what it achieved?
    - By getting rid of large components from Java Kernel
    - Kernel makefiles now differentiate between required and optional components

New Features In Java Runtime

- Feature 2 - No more PermGen
  - Java 6
    - Classes and interned strings were allocated in the PermGen which was part of the heap
  - Java 7
    - Puts interned strings in a new place – **Metaspace**
  - Java 8 doesn’t come with a PermGen at all. Only Metaspace.
    - Classes are also allocated on Metaspace
  - Metapsace
    - Native space used for holding classes and interned strings
    - Faster than PermGen
    - By default – has no max size –
    - Allocates memory according to needs (unlike PermGen which was part of the heap)
    - Native memory reduces footprint and executes faster

New Features In Java Runtime

- Feature 2 - No more PermGen

  - No more:
    - PermSize & MaxPermSize
    - OutOfMemoryError: PermGen Error
    - klasses that were used to describe class metadata

  - From now on:
    - MaxMetaspaceSize
    - OutOfMemoryError: Metaspace Error
    - Use –verbosegc to view Metaspace GC activity

New Features In Java Runtime

- Feature 3 – Tiered Compilation by Default

  – Client mode – compiles instant used code
    - Assuming that code will be reused frequently
    - Compilation starts sooner than in server mode
    - Fast launching – available JIT compilations for immediate execution

  – Server mode – compiles after more runtime analysis
    - Assuming there is much going on so better sit and analyze…
    - Optimizations are much more sophisticated
    - Slow launching – analyzing….. Code is re-interpreted until JIT kicks in…

New Features In Java Runtime

- Feature 3 – Tiered Compilation by Default

  - Tiered Compilation
    - Starts in client mode for fast launching and then shifts into server mode
    - Was experimental in Java 7 (XX:+TieredCompilation)
    - Default in Java 8 (for –server supported environments)

Java 9

- Java 9 – Most Important New Features
  - jshell – utility for Java REPL scripting
  - JMH – micro-benchmarking framework
  - G1 as default for OLD region (?)
  - HTTP2 compliant URLConnection & WebSocket
  - Process API – more info on JVM process
  - Project Jigsaw – Modular Java

- Expected release date – 9/2016

Last Words

- Java 8 addresses today & future major concerns
  - Rapid development
  - Parallel based APIs
  - Big data processing
  - Java Script support
  - Performance improvements
  - IoT relevance