Lambdas & Streams

Suggested Solutions

Table of Content

Lambdas & Method References

Exercise 01.01.a: Use forEach and Lambda Expressions

Use the forEach method from interface Iterable<T> and a lambda expression to print all points in a collection of points. Print them in one line and insert two blanks after each point.

The output should look like this:

```
java.awt.Point[x=1,y=1]  java.awt.Point[x=2,y=2]  java.awt.Point[x=3,y=3]  ...
```

Exercise 01.01.b: Access to Variables from Enclosing Context

Create a method filterPoints that takes two int-arguments: a modVal and a residue, which are used for filtering. The method shall print all points with the property: point.x % modVal == residue. For instance, filterPoints(2,0) would print all points with even x-coordinate.

Lambdas

Lambda Expressions

Angelika Langer
Trainer/Consultant

http://www.AngelikaLanger.com/

lab: forEach + lambda expression

- use `forEach` and a lambda expression
- print all points in a collection of points
- print them in one line and insert two blanks after each point

- required output :

```
java.awt.Point[x=1,y=1]  java.awt.Point[x=2,y=2]  java.awt.Point[x=3,y=3]  ...
```


solution

- use collection's `forEach` method
- pass lambda as argument

```
List<Point> points = ...
points.forEach(p -> System.out.print(p + "  "));
System.out.println();
```


lab: access context variable

- print all points with the property:
  `point.x % modVal == residue`

```
void filterPoints(int modVal, int residue) {
  points.forEach(p -> { if (p.x % modVal == residue)
                            System.out.print(p + "  ");
                      });
  System.out.println();
}
```

- note:
  - access to context variable from inside lambda
  - no `final` declaration needed

Exercise 01.02.a: Use forEach and Method References

This exercise is similar to the previous ones: use the forEach method from interface Iterable<T> to print all points in a collection of points. This time, use a method reference instead of a lambda expression. Use a reference to an existing method from the JDK libraries.

Exercise 01.02.b: Reference to a User-Defined Method

Define a helper method that formats the output, e.g. insert two blanks after each point. Print all points using a reference to the newly created helper method.

Lambdas

Method References

Angelika Langer
Trainer/Consultant

http://www.AngelikaLanger.com/

lab: forEach + method reference

- print all points in a collection of points
- use `forEach` and a method reference
- use a reference to an existing library method

solution

- use collection's `forEach` method
- pass method reference as argument

```
List<Point> points = ...
points.forEach(System.out::print);
System.out.println();
```

- compact code, but illegible output

```
java.awt.Point[x=1,y=1]java.awt.Point[x=2,y=2]java.awt.Point[x=3,y=3]  ...
```


lab: reference to user-define method

- define helper method for proper formatting
- use reference to helper method

```
class MyClass {
  private static void format(Point p) {
    System.out.print(p + "  ");
  }
  public static void main(String... args) {
    List<Point> points = ...
    points.forEach(MyClass::format);
    System.out.println();
  }
}
```

Exercise 01.03: With Lock

Using an explicit lock for synchronization purposes requires redundant code of the form:

```
public class NumberRange {
    private final Lock lock = new ReentrantLock();
    private volatile int lower = 0;
    private volatile int upper = 0;
    …
    public void setLower(int i) {
        lock.lock();
        try {
            if (i > upper)
                throw new IllegalArgumentException(
                        "can't set lower to " + i + " > upper");
            lower=i;
        } finally {
            lock.unlock();
        }
    }
}
```

Eliminate the redundancy by means of the Execute-Around-Pattern, i.e., implement a utility method withLock so that the code above can be simplified to:

```
public class NumberRange {
    private final Lock lock = new ReentrantLock();
    private volatile int lower = 0;
    private volatile int upper = 0;
    …
    public void setLower(int i) {
      withLock(lock, ()->{
            if (i > upper)
                throw new IllegalArgumentException(
                        "can't set lower to " + i + " > upper");
            lower=i;
      });
    }
}
```

The class in this lab uses explicit locks in methods with a void return type as well as in methods that return references or primitive type values. In addition it has a method that throws checked and unchecked exceptions.

Find a way to get rid of the redundant lock-related code in all of these cases.

Lambdas

Method-Around Pattern

Angelika Langer
Trainer/Consultant

http://www.AngelikaLanger.com/

lab: withLock

- eliminate redundant code related to explicit locks
  - by means of Execute-Around-Pattern

```
void setLower(int i) {
  lock.lock();
  try {
    if (i > upper) throw new IllegalArgumentException();
    lower=i;
  } finally {
    lock.unlock();
  }
}
```

- implement a `withLock` utility

```
void setLower(int i) {
  withLock(lock, ()->{
    if (i > upper) throw new IllegalArgumentException();
    lower=i;
  });
}
```


helper class `Utilities`

- split into a *functional type* and a *helper method*

```
public class Utilities {
  @FunctionalInterface
  public interface CriticalRegion {
    void apply();
  }

  public static void withLock(Lock lock, CriticalRegion cr) {
    lock.lock();
    try {
      cr.apply();
    } finally {
      lock.unlock();
    }
  }
}
```


void return type

- *user code*

```
public class NumberRange {
  private final Lock lock = new ReentrantLock();
  private volatile int lower = 0;
  private volatile int upper = 0;

  public void setLower(int i) {
    withLock(lock, () -> {
      if (i > upper)
        throw new IllegalArgumentException();
      lower = i;
    });
  }

  ...
```

lambda converted to functional type `CriticalRegion`

reference return type

- more user code

```
  ...

  public int[] getRange() {
     withLock(lock, ()->{

        return new int[] {lower,upper};

     });
  }
}
```

local return from lambda

- error:
  - `CriticalRegion::apply` does not permit return value
  - return in lambda is local, i.e., returns from lambda, not from `getRange`

reference return type (cont.)

No!

- more user code

```
  ...

  public int[] getRange() {
    int[][] retVal = new int[][] { null };
    withLock(lock, () -> {

      retval[0] = new int[] {lower,upper};

    });
    return retVal[0];
  }
}
```

array boxing hack

- implementation uses dubious array boxing hack

need more `CriticalRegion` interfaces

- `CriticalRegion` has signature:

```
interface CriticalRegion {
  void apply();
}
```

- but we also need this signature
  - in order to avoid array boxing hack

```
interface GenericCriticalRegion<T> {
  T apply();
}
```


more variants

- which requires a corresponding `withLock()` helper

```
static <T> T withLock(Lock lock,
                      GenericCriticalRegion<? extends T> cr) {
    lock.lock();
    try {
      return cr.apply();
    } finally {
      lock.unlock();
    }
  }
}
```

- which simplifies the `getRange()` method

```
int[] getRange() {
  return withLock(lock, () -> {

    return new int[] {lower,upper};

  });
}
```

no array boxing hack needed

more variants (cont.)

- but creates problems for the `setLower()` method
  - which returns `void`

- best solution
  - two interfaces: `CriticalRegion`, `GenericCriticalRegion<T>`
  - plus two overloaded methods:

```
void  withLock(Lock l, CriticalRegion cr)
<T> T withLock(Lock l, GenericCriticalRegion<? extends T> cr)
```


variants for primitive types

- additional variations for primitive return types
  - avoid autoboxing

```
interface CriticalRegionBoolean {
  boolean apply();
}
...
interface CriticalRegionInt {
  int apply();
}
interface CriticalRegionLong {
  long apply();
}
...
```

  - plus corresponding `withLock` helper methods

variants for primitive types (cont.)

- additional variations for primitive return types
  - create overload resolution problems
  - require (ugly) casts

```
boolean isValid() throws BrokenRangeException {
  return
  withLock(lock l, (CriticalRegionBoolean) () -> {
    if (lower <= upper) {
       return true;
    else
       throw new BrokenRangeException(this);
  });
}
```

  - ambiguous target typing
    - both Boolean and generic version match
    - add cast for disambiguation

data access

- input parameters can be captured
  - if they are effectively final
- fields can be accessed
  - without restrictions

```
void setLower(int i) {
    withLock(lock, () -> {
      if (i > upper)
         throw new IllegalArgumentException();
      lower = i;
    });
}
```

effectively final method argument is captured

mutable access to fields

alternative: no variants

- use only one generic interface
  `GenericCriticalRegion<T>`
- plus one generic method
```
<T> T withLock(Lock l,
          GenericCriticalRegion<? extends T> cr)
```
- requires modification of operations with `void` return

```
public void setLower(int i) {
  withLock(lock, () -> {
    if (i > upper)
       throw new IllegalArgumentException();
    lower = i;
    return (Void)null;
  });
}
```


checked exceptions are a problem

- only runtime exceptions are fine
  - checked exceptions from a lambda cause trouble

```
boolean isValid() throws BrokenRangeException {
  return withLock(lock, () -> {
    if (lower <= upper)
       return true;
    else
       throw new BrokenRangeException(this);
  });
}
```

checked exception

- error:
  - `CriticalRegion::apply` must not throw a checked exception

exception propagation

- how can we propagate checked exceptions ...
  - ... thrown by lambda back to surrounding user code?

- two options for propagation:
  - *tunnelling*
    - wrap it in a `RuntimeException`
  - *exception transparency*
    - transparently pass it back as is

self-made exception transparency

- declare functional interfaces with checked exceptions
  - functional type declares the checked exception(s):

```
interface CriticalRegionBooleanWE<E extends Exception> {
  void apply() throws E;
}
```

  - helper method declares the checked exception(s):

```
static <E extends Exception> void withLockWE(Lock lock,
          CriticalRegionBooleanWE<? extends E> cr) throws E {
    lock.lock();
    try {
       cr.apply();
    } finally {
      lock.unlock();
  } }
}
```

self-made exception transparency (cont.)

– user code simply throws checked exception

```
boolean isValid() throws BrokenRangeException {
  withLockWE(lock, () -> {
    if (lower <= upper)
       return true;
    else
       throw new BrokenRangeException(this);
  });
}
```

- caveat:
  – need more functional interfaces and helper method
  – for arbitrary number of checked exception

naive "tunnelling"

- naive approach
  – wrap checked exception into `RuntimeException`

```
static RuntimeException throwUnchecked(Throwable t) {
   throw new RuntimeException(t);
}
```

```
boolean isValid() throws BrokenRangeException {
  withLock(lock, () -> {
    if (lower <= upper)
       return true;
    else {
       throwUnchecked(new BrokenRangeException(this));
       return false;
    }
  });
}
```


naive "tunnelling" (cont.)

- downside
  – type information is lost
  – catch clause for checked exception type does no longer match

```
try {
  ... range.isValid() ...
}
catch (BrokenRangeException bre) {
  ... the intended error handling ...
}
catch (RuntimeException re) {
  ...
}
```

matching EH
after "tunnelling"

"tunnelling" via cast-hack

- idea
  – disguise checked exception as `RuntimeException`

```
static RuntimeException throwUnchecked(Throwable t) {
   throw (RuntimeException) t;
}
```

fails with
`ClassCastException`

- hurdle
  – cast fails at runtime with `ClassCastException`
- trick
  – use generics and take advantage of type erasure

"tunnelling" via cast-hack (cont.)

- trick
  - use generics and take advantage of type erasure

```
class Exceptions {
  @SuppressWarnings("unchecked")
  private static <T extends Throwable>
  T doThrowUnchecked(Throwable t) throws T {
    throw (T) t;
  }
  public static RuntimeException throwUnchecked(Throwable t) {
     throw Exceptions.<RuntimeException>doThrowUnchecked(t);
  }
}
```

- `throwUnchecked`
  - maps a `Throwable` to a `RuntimeException`
  - via a generic method

how does the cast hack work?

- generic method performs a cast to `T`
  - at compile-time T:=RuntimeException
  - at runtime (due to type erasure) T:=Throwable

before type erasure

```
@SuppressWarnings("unchecked")
private static <T extends Throwable>
T doThrowUnchecked(Throwable t) throws T {
  throw (T) t;
}
```

cast succeeds

after type erasure

```
private static
Throwable doThrowUnchecked(Throwable t) throws Throwable {
  throw (Throwable) t;
}
```


why is it a hack?

- `throwUnchecked` disables compiler checks
  - defeats the purpose of checked exceptions
  - unchecked warning must be ignored
- all checks are off
  - throws clause not checked for correctness
  - caller need not handle or declare the exception
- use the hack only where needed

exception transparency

- what we would like to have ...
  - compiler figures out all checked exception thrown by the lambda
  - transparently passes them on to the lambda's context

```
boolean isValid() throws BrokenRangeException {
  return withLock(lock, () -> {
    if (lower <= upper)
       return true;
    else
       throw new BrokenRangeException(this);
  });
}
```

checked exception

- compiler support for exception transparency
  - was discussed and discarded (too much effort)

why is there no `withLock` in the JDK?

- hidden performance cost
- critical region usually has side effects
  – uses context variables => *capturing lambda*

```
int[] getRange() {
  return withLock(lock, () -> {
    return new int[] {lower,upper};
  });
}
```

captured fields

- lambda translation (via invokedynamic + metafactory)
  – causes allocation (of so-called *lambda object*)
  – on evaluating (not invoking) a capturing lambda

lambda translation

- lambda expression replaced by *lambda capture*
  – lambda capture `:=` a *lambda object* created via invokedynamic
  – lambda object `:=` a description of "how to invoke the lambda"

```
int[] getRange() {
  return withLock(lock, () -> {
    return new int[] {lower,upper};
  });
}
```

lambda expression

```
int[] getRange() {
  return withLock(lock,
     INDY[metafactory
         ,MH(CriticalRegionGeneric.apply)
         ,MH(NumberRange.lambda$0)]
     (lower,upper);
  });
}
```

lambda capture

lambda translation (cont.)

- lambda objects for *non-capturing* lambdas
  – are independent of context (no bindings)
  – similar to static methods/objects (no `this` + static initialization)
  – are created only once by lazy evaluation

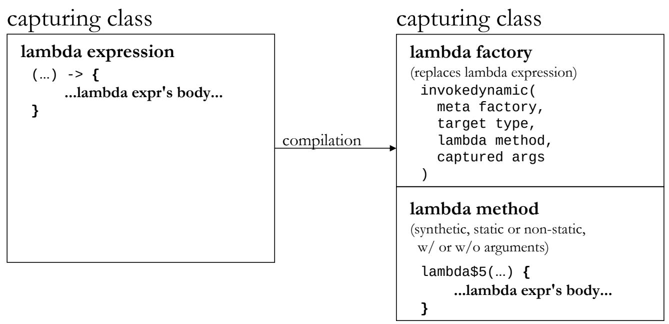
- lambda objects for *capturing* lambdas
  – have bindings to context variables
  – similar to non-static methods/objects (`this` + construction)
  – are created on each evaluation

⇩

cost of allocation

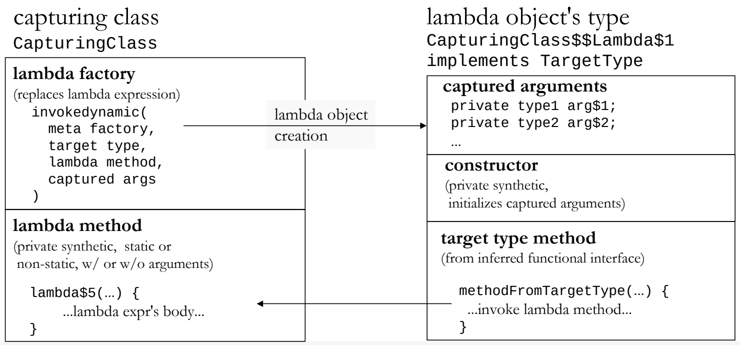
runtime representation #1

- compiler replace lambda expression by *lambda factory*
  – and creates synthetic *lambda method*



runtime representation #2

- first use of lambda calls meta factory
  => creates *lambda object type* and *lambda object*
  - defers initialization cost to first use + no overhead if lambda is never used



Non-Abstract Interface Methods

Exercise 02.01: Named Person

Add default methods to interfaces. Consider the following interfaces:

```
interface Name {
    String getFirstName();
    String getMiddleName();
    String getLastName();
    String getName();
}
```

and

```
interface Age {
    LocalDate getDateOfBirth();
    long getAge();
    MonthDay getBirthday();
}
```

The following yet incomplete class is supposed to implement the two interfaces:

```
public class Person implements Name, Age {
    private String firstName;
    private String middleName;
    private String lastName;
    private LocalDate dateOfBirth;

    public Person(String first, String middle, String last,
                  int day, Month month, int year) {
        firstName  = first;
        middleName = middle;
        lastName   = last;
        dateOfBirth= LocalDate.of(year,month,day);
    }
}
```

Provide the missing methods. Which methods can you provide as default interface methods? Which methods must be implemented in the class?

Lambdas

Default Methods

Angelika Langer
Trainer/Consultant

http://www.AngelikaLanger.com/

lab: withLock

- given interfaces

```
interface Name {
  String getFirstName();
  String getMiddleName();
  char getMiddleInitial();
  String getLastName();
  String getName();
}
```

```
interface Age {
  LocalDate getDateOfBirth();
  long getAge();
  MonthDay getBirthday();
}
```

- given class

```
class Person implements Name, Age {
  private String firstName, middleName, lastName;
  private LocalDate dateOfBirth;

  public Person(String first, String middle, String last,
                int    day,   Month  month,  int    year){
  } ...
}
```

- implement missing methods
  - either in class or interface

default methods

- implement default methods
  - on top of abstract interface methods

```
interface Name {
  String getFirstName();
  String getMiddleName();
  String getLastName();

  default char getMiddleInitial() {
    if (getMiddleName()!=null&&getMiddleName().length()>0)
      return getMiddleName().charAt(0);
    return ' ';
  }
  default String getName() {
    char c = getMiddleInitial();
    return getFirstName()+(c!=' '?" "+c+'.':"")
                         +" "+getLastName();
  }
}
```


default methods

- implement default methods
  - on top of abstract interface methods

```
interface Age {
  LocalDate getDateOfBirth();

  default long getAge() {
    return ChronoUnit.YEARS.between
                   (getDateOfBirth(),LocalDate.now());
  }
  default MonthDay getBirthday() {
    return MonthDay.from(getDateOfBirth());
  }
}
```

methods implemented in class

- implement remaining methods in class
  - since they need access to fields

```
class Person implements Name, Age {
  private String firstName, middleName, lastName;
  private LocalDate dateOfBirth;
  ...
  public String getFirstName()      {return firstName;}
  public String getMiddleName()     {return middleName;}
  public String getLastName()       {return lastName;}
  public LocalDate getDateOfBirth() {return dateOfBirth;}
  public String toString() {
    return String
     .format("%-12s= %s\n%-12s= %s\n%-12s= %s\n%-12s= %s"
            ,"firstName"  ,firstName
            ,"middleName" ,middleName
            ,"lastName"   ,lastName
            ,"dateOfBirth",dateOfBirth
  } );
}
```

Exercise 02.02: Refactoring Class Collections

The JDK class java.util.Collections is a classic example of a "bag of static methods" class. How would you refactor it now that static interface methods are permitted? Would you stuff everything as a static method into the java.util.Collection interface?

Obviously, this is a pure (and hypothetical) design exercise, since we cannot really refactor a JDK class.

Consider in particular the following methods from class java.util.Collections:

- static <T> boolean addAll(Collection<? super T> c, T... elements)
- static <T extends Object & Comparable<? super T>>
  T max(Collection<? extends T> coll)
- static <T extends Comparable<? super T>> void sort(List<T> list)
- static final <T> List<T> emptyList()
- static <T> ArrayList<T> list(Enumeration<T> e)

Lambdas

Static Interface Methods

Angelika Langer
Trainer/Consultant

http://www.AngelikaLanger.com/

lab: refactor class `java.util.Collections`

- refactor a typical "bag of static methods" class
  - would we stuff all static methods into the related interface?

```
public class Collections {
 public static <T>
   boolean addAll(Collection<? super T> c, T... elements)
 public static <T extends Object & Comparable<? super T>>
   T max(Collection<? extends T> coll)
 public static <T extends Comparable<? super T>>
   void sort(List<T> list)
 public static final <T>
   List<T> emptyList()
 public static <T>
   ArrayList<T> list(Enumeration<T> e)
 ...
}
```


Collections.addAll()

```
class Collections {
  public static <T>
  boolean addAll(Collection<? super T> c, T... elements) {
    ...
  }
}
```

- pro:
  - might be considered closely related to interface `Collection`
- con:
  - leave in class if there are too many other static interface methods

```
interface Collection<E> extends Iterable<E> {
  static <T>
  boolean addAll(Collection<? super T> c, T... elements) {
        boolean result = false;
        for (T element : elements)  // uses iterator()
            result |= c.add(element);
        return result;
  }
}
```


Collections.max()

```
class Collections {
  public static <T extends Object & Comparable<? super T>>
  T max(Collection<? extends T> coll){
    ...
  }
}
```

- pro:
  - might be considered closely related to interface `Collection`
- con:
  - leave in class if there are too many other static interface methods

```
interface Collection<E> extends Iterable<E> {
  static <T extends Object & Comparable<? super T>>
  T max(Collection<? extends T> coll) {
    Iterator<? extends T> i = coll.iterator();
    T candidate = i.next();
    while (i.hasNext()) {
      T next = i.next();
      if (next.compareTo(candidate) > 0) candidate = next;
    }
    return candidate;
  }
}
```

Collections.sort()

```
class Collections {
  public static <T extends Comparable<? super T>>
  void sort(List<T> list) {
     ...
  }
}
```

- con:
  - not at all related to interface `Collection`
  - perhaps suitable for interface `List`
  - leave in class if there are too many other static interface methods

```
interface List<E> extends Collection<E> {
  static <T extends Comparable<? super T>>
  void sort(List<T> list) {
        Object[] a = list.toArray();
        Arrays.sort(a);
        ListIterator<T> i = list.listIterator();
        for (int j=0; j<a.length; j++) {
            i.next();
            i.set((T)a[j]);
        }
  }
}
```


Collections.emptyList()

```
class Collections {
  private static class EmptyList<E> extends AbstractList<E> {...}
  public static final List EMPTY_LIST = new EmptyList<>();
  public static final <T> List<T> emptyList() {
    return (List<T>) EMPTY_LIST;
  }
}
```

- con:
  - not at all related to interface `Collection`; more suitable for interface `List`
  - requires nested class `EmptyList` in interface => debatable
  - formerly private class would become public
    - everything in an interface is public
  - can use anonymous class instead to preserve privateness => still weird

```
interface List<E> extends Collection<E> {
  static final List EMPTY_LIST = new AbstractList() {...};
  static <T> List<T> emptyList() {
    return (List<T>) EMPTY_LIST;
  }
}
```


Collections.list()

```
class Collections {
  public static <T> ArrayList<T> list(Enumeration<T> e) {
        ArrayList<T> l = new ArrayList<>();
        while (e.hasMoreElements())
            l.add(e.nextElement());
        return l;
  }
}
```

- con:
  - neither belongs to interface `Collection`
    nor interfaces `List` or `Enumeration`

Stream Basics

Exercise 03.01: Filtering

Step 1: In a list of `Point` objects find all points with positive x-coordinate and print these *points*.

Step 2: In a list of `Point` objects find all points with positive x-coordinate, and print these *coordinates*.

Step 3: In a list of `Point` objects find all points with *distinct* positive x-coordinate, and print these *coordinates*.

Step 4 (optional): In a list of `Point` objects find all points with *distinct* positive x-coordinate, and print these *points*.

Lambdas

Filtering

Angelika Langer
Trainer/Consultant

`http://www.AngelikaLanger.com/`

lab: filter

- find all points with positive x-coordinate and print these *points*

```
List<Point> points = ...
points.stream()
      .filter (p -> p.getX() > 0)
      .forEach(p -> System.out.print(p + "  "));
```


lab: filter

- print the *coordinates* instead of the points themselves

```
List<Point> points = ...
points.stream()
      .filter (p -> p.x > 0)
      .forEach(p -> System.out.print(p.x + "  "));
```

- alternatively:

```
List<Point> points = ...
points.stream()
      .mapToInt(p->p.x)
      .filter(x -> x > 0)
      .forEach(x -> System.out.print(x + "  "));
```


lab: distinct

- find all points with *distinct* positive x-coordinate, and print these *coordinates*

```
List<Point> points = ...
points.stream()
      .mapToInt(p -> p.x)
      .filter  (x -> x > 0)
      .distinct()
      .forEach (x -> System.out.print(x + "  "));
```

lab: distinct

- print the *points* instead of the coordinates

- solution #1: using `distinct()`
  - distinct compares via `equals()`
  - map the points to point wrapper with an `equals()` method that compares only the x-coordinates

solution #1: using `distinct()`

```
List<Point> points = ...
points.stream()
      .filter(p -> p.x > 0)
      .map(p -> new Point(p) {
         public boolean equals(Object that) {
           if (this == that) return true;
           if (that == null) return false;
           if (this.getClass() != that.getClass())
              return false;
           return this.x == ((Point)that).x;
         }
         public int hashCode() {
           return this.x;
         }
       })
      .distinct()
      .forEach(p -> System.out.print(p + "  "));
```


solution #2: using a TreeSet

- solution #2: using a `TreeSet`
  - sets reject duplicates
  - `TreeSet`s work with a `Comparator`
  - stuff points into `TreeSet` with a `Comparator` that compares only the x-coordinates

solution #2: using a TreeSet

```
List<Point> points = ...
points.stream()
      .filter(p -> p.x > 0)
      .collect(Collectors.toCollection(
        () -> new TreeSet<Point>((p1,p2) -> p1.x-p2.x))
       )
      .forEach(p -> System.out.print(p + "  "));
```

Comparator

- collect to a collection
  - that is a tree set
  - with a special purpose comparator
- caveat: not order preserving
  - points printed in an order different from order in stream source

solution #3: using a LinkedHashMap

- solution #3: using a `LinkedHashMap`
  - use x-coordinate as key and `Point` as value
  - `HashMap` part eliminates duplicate x-coordinates
  - `LinkedList` part provides iteration in insertion order

solution #3: using a LinkedHashMap

```
List<Point> points = ...
points.stream()
      .filter(p -> p.x > 0)
      .collect(Collectors.toMap(
          p->p.x                    // key mapper
         ,p->p                      // value mapper
         ,(pOld,pNew)->pOld         // merger
         ,LinkedHashMap::new))      // map supplier
      .values()
      .forEach(p -> System.out.print(p + "  "));
```

- collect to a `LinkedHashMap<Integer,Point>`
  - key mapper extracts x-coordinate
  - merger retains first occurrence & discards subsequent points
- order preserving
  - value set iterator uses insertion order

solution #4: filter using a Map

- solution #4: using a `Map`
  - maps also reject duplicate keys
  - use the x-coordinate as a key (+ a dummy value)

solution #4: filter using a Map

```
List<Point> points = ...
final Map<Integer,Boolean> map = new HashMap<>();
points.stream()
      .filter (p -> p.x > 0)
      .filter (p -> map.putIfAbsent(p.x,true) == null)
      .forEach(p -> System.out.print(p + "  "));
```

Predicate

- use stateful (!) filter with a predicate that ...
  - adds each coordinate to a map
    - if absent => fine (point goes downstream)
    - if present => bad (point is eliminated)
- note: order is preserved
  - filtering only suppresses value, but does not re-order them

Exercise 03.02: Reduction

In a list of `Point` objects sum up the x-coordinates of all points and print the resulting sum. For tracing purposes, print all x-coordinates before printing the sum.

Step 1: Use the `reduce()` operation *with* an initial value.

Step 2: Use the `reduce()` operation *without* an initial value.

Step 3: Try to omit the mapping to the x-coordinate and reduce the points directly.

Lambdas

Reduction

Angelika Langer
Trainer/Consultant

http://www.AngelikaLanger.com/

peek

- for tracing purposes, print all x-coordinates

```
List<Point> points = ...
int result = points.stream()
      .mapToInt(p->p.x)
      .peek(x->System.out.print(x + " "))
      .reduce(0, (x1,x2)->x1+x2);
System.out.println("sum is: " + result);
```

```
1 2 3 1 2 3 -1 -2 -3 -1 -2 -3 sum is: 0
```


lab: reduce with initial value

- sum up the x-coordinates of all points and print the sum
  - use `reduce()` *with* an initial value

```
List<Point> points = ...
int result = points.stream()
      .mapToInt(p->p.x)
      .reduce(0, (x1,x2)->x1+x2);
System.out.println("sum is: " + result);
```

```
sum is: 0
```

- if stream is empty
  - `reduce()` returns initial value
- can't tell whether stream was empty ...

lab: reduce without initial value

- use `reduce()` *without* an initial value

```
List<Point> points = ...
points.stream()
      .mapToInt(p->p.x)
      .peek(x->System.out.print(x + " "))
      .reduce((x1,x2)->x1+x2)
      .ifPresent(s->System.out.println("sum is: "+s));
```

- `reduce()` returns an `Optional`
- which has methods
  - `isPresent()`
    - returns true if there is a value present, otherwise false
  - `ifPresent()`
    - invoke specified function with the value, otherwise do nothing

sum

- primitive streams have a `sum()` operation
  - `sum()` implicitly uses an initial value

```
List<Point> points = ...
int result = points.stream()
      .mapToInt(p -> p.x)
      .peek(x -> System.out.print(x + " "))
      .sum();
System.out.println("sum is: " + result);
```

- `sum()` is equivalent to:

```
reduce(0, Integer::sum);
```


lab: reduce with combiner

- omit mapping to x-coordinate + reduce points directly

```
List<Point> points = ...
int result = points.stream()
      .peek(p->System.out.print(p.x+" "))
      .reduce(0, (x,p2)->x+p2.x, Integer::sum);
System.out.println("sum is: " + result);
```

Identity Accumulator Combiner

Advanced Streams

Exercise 04.01 Stream Creation

Step 1: Create a sequential and a parallel stream of strings that has an array as the underlying stream source. The stream should contain the days of the week. Find at least two ways of doing it.

Step 2: Create primitive streams:

- a stream of natural constants containing 2.997 924 58 (speed of light), 3.1415 9265 359 (Pi) and 6.67384 (gravitational constant), and
- a stream of the integral numbers from 51 thru 100 (inclusive).

Step 3: Create a stream that has a character sequence (say, a string containing your name) as the underlying stream source.

Step 4: Create infinite streams:

- a stream of pseudo random numbers,
- the stream of the powers of 2, i.e., 2 4 8 16 32 64 …
- the stream of `BigIntegers` with all positive integral numbers

Hint: Interface `Stream` has `generate` and `iterate` methods for this purpose.

Lambdas

Stream Creation

Angelika Langer
Trainer/Consultant
http://www.AngelikaLanger.com/

lab: array-based streams

- create stream of strings with an array as stream source

```
String[] weekdays = {"Monday", "Tuesday", "Wednesday",
                     "Thursday", "Friday", "Saturday",
                     "Sunday"};
Stream<String> sequentialStringStream
  = Arrays.stream(weekdays);
```

```
Stream<String> sequentialStringStream
  = Stream.of("Monday", "Tuesday", "Wednesday",
              "Thursday", "Friday", "Saturday",
              "Sunday");
```

```
Stream<String> parallelStringStream
                   = sequentialStringStream.parallel();
```


lab: primitive streams

- create stream of natural constants

```
DoubleStream naturalConstants
= DoubleStream.of(
            2.997_924_58     // speed of light
           ,3.1415_9265_359  // Pi
           ,6.67384          // gravitational constant
);
```

- create stream of integrals from 51 thru 100 (inclusive)

```
IntStream range = IntStream.rangeClosed(51,100);
IntStream range = IntStream.range      (51,101);
```


lab: character sequence-based streams

- create a stream with a character sequence as stream source

```
IntStream characterStream = "Angelika Langer".chars();
```

- remember cast from `int` to `char` for character

```
static void print(IntStream stream, boolean asChar) {
  String fmt = (asChar)?"%c":"%d";
  stream.forEach(i -> System.out.format(fmt, i));
}
```

chars() vs. codepoints()

- `chars()`: code units in UTF-16 encoding
- `codePoints()`: Unicode codes

- difference:
  - *one code point in Unicode* may have *several code units in UTF-16*

- example: 𝕫 (mathematical symbol for integral numbers)
  - Unicode code point: U+1D56B
  - 2 code units in UTF-16: \uD835 \uDD6B

- conversions via:

```
char[] Character.toChars     (int codePoint)
int    Character.toCodePoint (char high, char low)
```


lab: infinite streams

- create stream of pseudo random numbers

```
Stream<Double> randomNumbers
                     = Stream.generate(Math::random);
```

- `generate()`
  - takes a generator function with no argument

- evaluated lazily
  - whenever a new element is needed, the generator is called

lab: infinite streams

- create stream of powers of 2, i.e., 2 4 8 16 32 64 …

```
Stream<Integer> powersOfTwo
                   = Stream.iterate(2, i -> i * 2);
```

- create stream of `BigInteger`s with all positive integral numbers

```
Stream<BigInteger> bigIntegers
        = Stream.iterate(BigInteger.ZERO,
                         i->i.add(BigInteger.ONE));
```

- `iterate()`
  - takes a “seed” value and a function
  - repeatedly applies the function to the previous result
  - returns `seed`, `f(seed)`, `f(f(seed))`, etc.

Exercise 04.02: Modification of Points

Step 1: In a list of `Point` objects set the y-coordinate to 0 for all points where the x-coordinate is positive. Then print the modified points.

Step 2: Mutation of sequence elements is not always a good idea. Can you produce the same output (as in step 1) without mutating the points? Idea: Generate a stream that contains the respective modified points and print them. Verify that the originals are unchanged by printing them, too.
Don't forget to reset the list of points to the original values; after all we modified them in step 1.

Step 3: Generate new points (from the original points) with 10-times larger coordinate, i.e., calculate as follows:

```
Point np = new Point(10 * p.x, 10 * p.y);
```

Add these new points to the original list of points or produce a stream that contains the original points followed by the newly generated points.

Lambdas

Side Effects

Angelika Langer
Trainer/Consultant

http://www.AngelikaLanger.com/

lab: in-place modification of elements

- set y-coordinate to 0 for all points with positive x-coordinate

```
List<Point> points = ...
points.stream()
      .filter (p -> p.getX()>0)
      .forEach(p -> p.setLocation(p.x,0));
points.forEach(x->System.out.print(x+"  "));
```

- in-place modification of sequence elements is dangerous
  - okay for array and `List`
  - disastrous for `Set`

lab: no modification => no side effects

- generate a stream that contains the modified points
  - no side effects - does not modify anything

```
List<Point> points = ...
points.stream()
      .map(p -> p.x > 0 ? new Point(p.x,0) : p)
      .forEach(x->System.out.print(x+"  "));
```


lab: modification of stream source

- generate new points with 10-times larger coordinates
- add new points to original list or
- produce a stream with original points followed by new points

```
List<Point> points = ...
points.stream()
      .map(p -> new Point(10*p.x,10*p.y))
      .forEach(points::add);
points.forEach(x->System.out.print(x+"  "));
```

- triggers `ConcurrentModificationException`
  - non-interference requirement violated
  - function modifies underlying stream source

lab: concatenation of streams

- generate new stream with new points
- concatenate original and new stream afterwards

```
List<Point> points = ...
Stream.concat(
         points.stream(),
         points.stream().map(p -> new Point(10*p.x,10*p.y))
       )
      .forEach(x->System.out.print(x+"  "));
```

- no modification => no side effects

Exercise 04.03: Author Fee

Authors are paid according to the length of their manuscripts or, more precisely, per non-space character.

Calculate the author fee for the text in file `text.txt`. In other words, count all non-space characters in the text file and print the count.

Hint: a space character can be identified via `Character.isSpace()`.

Trouble Shooting: If the text file `text.txt` cannot be found, then make sure that you place the file into the working directory. If you use IntelliJ you can specify the working directory as follows: open menu Run => Edit Configurations => Working Directory.

Lambdas

FlatMap

Angelika Langer
Trainer/Consultant

http://www.AngelikaLanger.com/

lab: use Reader::lines

- count all non-space characters in a text file

```
try (BufferedReader in
     = new BufferedReader(new FileReader("text.txt"))){
   long cnt = in.lines()
                   ... next slide ...

   System.out.println("non-spaces="+cnt);
} catch (IOException | UncheckedIOException e) {
   e.printStackTrace();
}
```

  – open file and create a `BufferedReader`
  – create `Stream<String>` via `BufferedReader::lines`

use flatMap()

- map all lines to a single character stream
  – via `String::chars`
- eliminate all space characters
  – needs opposite of `Character::isSpaceChar`
- count remaining non-spaces

```
...
  long cnt = in.lines()
     .flatMapToInt(String::chars)
     .filter(Character::isSpaceChar.negate())
     .count();
   System.out.println("non-white-space chars: " + cnt);
...
```


Predicate::negate

- create opposite of `Character::isSpaceChar`
  – via `negate()` in interface `Predicate`
- `Character::isSpaceChar.negate()` does not compile
  – *method invocation* is no inference context
- must insert cast to predicate type
  – *cast* is an inference context

```
...
in.lines()
  .flatMapToInt(String::chars)
  .filter(((IntPredicate)Character::isSpaceChar).negate() )
  .count();
...
```

Exercise 04.04: Concatenating Character Sequences

Step 1: From a list of Point objects build a String object that represents all points. Print the string. The output should look like this:

```
java.awt.Point[x=1,y=1] java.awt.Point[x=2,y=2] java.awt.Point[x=3,y=3]
...
```

In this first step use the collect() operation and the joining() collector from class Collectors. There are two versions available:

- without delimiter
- with delimiter

Try out both

Step 2: Now, as an alternative, use the reduce() operation and string concatenation via the +-operator of class String.

Note, how equally concise this approach looks. However, consider the performance of both solutions. Which one do you expect to run faster?

Lambdas

Joining

Angelika Langer
Trainer/Consultant

http://www.AngelikaLanger.com/

lab: concatenate via joining

- use `StringBuilder`
  - use pre-defined `joining` collector

```
List<Point> points = ...
String result = points.stream()
                      .map(p->p+"  ")
                      .collect(Collectors.joining());
System.out.println(result.toString());
```

- substantially more efficient
  - uses `StringBuilder` internally

joining with delimiter

- `joining` collector may take delimiters, prefix & suffix

```
List<Point> points = ...
String result = points.stream()
                      .map(Point::toString)
                      .collect(Collectors.joining("  "));
System.out.println(result);
```

delimiter

lab: concatenate strings via +-operator

- build a `String` that represents all points
  - use `String` => perform reduction using `+`-operator for `String`

```
List<Point> points = ...
String result = points.stream()
                      .map(p->p+"  ")
                      .reduce("",(s1,s2)->s1+s2);
System.out.println(result);
```

- inefficient
  - creates lots of temporary strings
  - compiler optimization (via `StringBuilder.append`) not possible
  - compiler only sees two strings at a time

Exercise 04.05: Collecting Points

In a list of `Point` objects find all points with positive x-coordinate and store them in an `ArrayList`. Print the new `ArrayList`.

Lambdas

toCollection

Angelika Langer
Trainer/Consultant

http://www.AngelikaLanger.com/

lab: collect to a collection

- find all points with positive x-coordinate and store them in an `ArrayList`

```
List<Point> points = ...
ArrayList<Point> result
  = points.stream()
          .filter(p -> p.getX() > 0)
          .collect(Collectors.toCollection(ArrayList::new));
```

- `toList` collector return a `List`, not an `ArrayList`
  - must use `toCollection`
  - with constructor reference `ArrayList::new` as supplier

Exercise 04.06: Author Fee Revisited

Authors are paid according to the length of their manuscripts or, more precisely, per non-space character.

To get a feeling for a text file's "noise" ratio, count the space and non-space characters in one pass through the text in file `text.txt` and print the results.

Lambdas

Grouping

Angelika Langer
Trainer/Consultant

http://www.AngelikaLanger.com/

lab: use Reader::lines

- count space and non-space characters in one pass through a text file

```
try (BufferedReader inFile
     = new BufferedReader(new FileReader("text.txt"))){
  long cnt = inFile.lines()
                    ... next slide ...

   System.out.println("non-spaces="+cnt);
} catch (IOException | UncheckedIOException e) {
   e.printStackTrace();
}
```

  - open file and create a `Reader`
  - create `Stream<String>` via `Reader::lines`

flat-map

- flat-map all lines to a single character stream
  - needs `flatMapToInt()` because `chars()` yields `IntStream`

```
...
Map<Boolean,List<Integer>> map = inFile
   .lines()                                  // Stream<String>
   .flatMapToInt(String::chars)              // IntStream
   .boxed()
   .collect(Collectors.partitioningBy
                       (Character::isSpaceChar));
...
```


boxing

- convert to `Stream<Integer>` via `boxed()`
  - because `IntStream` has no `collect(Collector)` method

```
...
Map<Boolean,List<Integer>> map = inFile
   .lines()                                  // Stream<String>
   .flatMapToInt(String::chars)              // IntStream
   .boxed()                                  // Stream<Integer>
   .collect(Collectors.partitioningBy
                       (Character::isSpaceChar));
...
```

grouping

- group by `isSpaceChar()`
  - yields `Map<Boolean, List<Integer>>`
  - associates true => list of space characters
    false => list of non-space characters

```
...
Map<Boolean,List<Integer>> map = inFile
   .lines()                              // Stream<String>
   .flatMapToInt(String::chars)          // IntStream
   .boxed()                              // Stream<Integer>
   .collect(Collectors.partitioningBy    // Map<Boolean,List<Integer>>
                       (Character::isSpaceChar));
int chars  = map.get(false).size();
int spaces = map.get(true).size();
...
```


using a downstream collector

- use `counting()` downstream collector

```
...
Map<Boolean,Long> map = inFile
   .lines()                              // Stream<String>
   .flatMapToInt(String::chars)          // IntStream
   .boxed()                              // Stream<Integer>
   .collect(Collectors.partitioningBy    // Map<Boolean,Long>
       (Character::isSpaceChar,Collectors.counting()));

long chars  = map.get(false);
long spaces = map.get(true);
...
```

Exercise 04.07: Collecting Points Revisited

Partition a list of Point objects into the points with positive (>=0) and the points with negative (<0) x-coordinate.

Step 1: For each partition find a point with maximum *distance* from the origin.

Step 2: For each partition find the maximum *y-coordinate*.

Step 3 (optional): For each partition find *all* points with maximum distance from the origin (similar to step 1).

Lambdas

Downstream Collectors

Angelika Langer
Trainer/Consultant

http://www.AngelikaLanger.com/

lab: downstream collector maxBy

- partition a list of `Point` objects
  - into points with positive and negative x-coordinate
- for each partition find a point
  - with maximum distance from origin

```
List<Point> points = ...
Map<Boolean, Optional<Point>> result
= points.stream()
 .collect(Collectors.partitioningBy(p->p.x>=0,
          Collectors.maxBy(
           (p1,p2)->Double.compare(distanceFromOrigin(p1),
                                   distanceFromOrigin(p2))
          )));
```

```
{false=Optional[Point[x=-3,y=-3]],true=Optional[Point[x=3,y=3]]}
```


alternative comparator for maxBy

```
Comparator<Point> comparator = ...;
Map<Boolean, Optional<Point>> result
= points.stream()
 .collect(Collectors.partitioningBy(p->p.x>=0,
          Collectors.maxBy(comparator)));
```

- various ways of expressing the comparator:

as a lambda

```
Comparator<Point> comparator
= (p1,p2) -> Double.compare(distanceFromOrigin(p1),
                            distanceFromOrigin(p2));
```

created from a “key extractor”

```
Comparator<Point> comparator
= Comparator.comparing(Test::distanceFromOrigin);
```


lab: downstream collector maxBy

- for each partition find the maximum y-coordinate

```
List<Point> points = ...
Map<Boolean, Optional<Integer>> result
= points.stream()
 .collect(Collectors.partitioningBy(p->p.x>=0,
          Collectors.mapping(p->p.y,
          Collectors.maxBy(Integer::compareTo))));
```

```
{false=Optional[3], true=Optional[3]}
```

lab: find all points with max distance

- for each partition find *all* points with maximum distance from origin

- approach:
  - apply a finisher to the map that is returned after partitioning

```
points.stream()
  .collect(Collectors.collectingAndThen(
           Collectors.partitioningBy(p -> p.x >= 0),
           … a finisher that takes the map …
           )
  );
```


finisher

- for each key get the associated list of points
  - map the points to their distance from origin and call max()
  - filter the list, find all points with maximum distance and print them

```
grouping -> {
  grouping.keySet().stream()
    .forEach(booleanKey ->
       grouping.get(booleanKey).stream()
         .mapToDouble(Lab_04_07::distanceFromOrigin)
         .max()
         .ifPresent(maxDist->
            grouping.get(booleanKey).stream()
              .filter(p->distanceFromOrigin(p)==maxDist)
              .forEach(System.out::println)
    ));
  return (Void)null;
}
```

Exercise 04.08: Using Optional

Step 1: Wrap a Legacy Method

Consider a legacy method that takes null values and returns null values. Adapt it to a context, in which null has been replaced by Optional.empty(). As an example we use the getProperty() method from class System:

public static String getProperty(String key, String def)

The method returns the string value of the system property key, or the default value def if there is no property with that key.

The method would be used like this:

```
String value1 = System.getProperty("none",null);
System.out.println(value1);
String value2 = System.getProperty("none","default");
System.out.println(value2);
```

The method returns a null reference or the string "default".

Provide a wrapper around this legacy method for a context in which null is not used and no longer permitted. The wrapper shall take Optional values where appropriate and return an Optional value instead of null.

The skeleton is in class TestLegacyWrapper.

Step 2: Use Optional as Element

Consider a map that has null values associated with some of the keys. When you retrieve a value from this map via the get() method, then the get() method might return null. We cannot tell whether the null return value means "there was no value for this key" or whether it means "there was a value and the value is null". It requires an additional call to the containsKey() method in order to tell the difference.

Using an empty Optional instead of null as the associated value would solve the problem. Try it out!

Lambdas

Using Optional

Angelika Langer
Trainer/Consultant

http://www.AngelikaLanger.com/

lab: a legacy wrapper

- legacy method takes `null` and returns `null`

```
String v1 = System.getProperty("none",null);
System.out.println(v1);
String v2 = System.getProperty("none","default");
System.out.println(v2);
```

```
null
default
```


lab: a legacy wrapper (cont.)

- provide a wrapper that takes and returns `Optional`

```
Optional<String> v1
= wrappedGetProperty("none",Optional.empty());
System.out.println(v1);
Optional<String> v2
= wrappedGetProperty("none",Optional.of("default"));
System.out.println(v2);
```

```
Optional.empty
Optional[default]
```


the wrapper

- provide a wrapper that takes and returns `Optional`

```
Optional<String>
wrappedGetProperty(String key, Optional<String> def)  {
  return Optional.ofNullable(
     System.getProperty(key,def.orElse(null))
  );
}
```

  - convert possible `null` return from `getProperty()` method via `Optional<T> ofNullable(T t)`
  - convert `Optional` to plain value or `null`) via `orElse(null)`

lab: optional stream elements

- map with associated `null` values

```
Map<Integer, String> map = new HashMap<>();
map.put(1, "one");
map.put(2, null);
...
for (int i=0; i<3; i++) {
  String s = map.get(i);
  if (s == null)
     if (map.containsKey(i))
        System.out.println(i + " value is null");
     else
        System.out.println(i + " has no entry");
  else
     System.out.println(i + " -> " + s);
}
```

```
0 has no entry
1 -> one
2 value is null
```


lab: optional stream elements (cont.)

- map with associated `Optional` values

```
Map<Integer, Optional<String>> map = new HashMap<>();
map.put(1, Optional.ofNullable("one"));
map.put(2, Optional.empty());
...
for (int i=0; i<3; i++) {
  Optional<String> os = map.get(i);
  if (os == null)
     System.out.println(i + " has no entry");
  else
     System.out.println(i + " -> " + os.orElse("null"));
}
```

```
0 has no entry
1 -> one
2 -> null
```

Exercise 04.09: Closing I/O-Based Streams

Consider the following stream factory method:

```
static Stream<String> createStreamFromFile (String filename) {
  try {
    final BufferedReader in
             = new BufferedReader(new FileReader(filename));
    Stream<String> myStream = in.lines();
    return myStream;
  } catch (FileNotFoundException e) {
    e.printStackTrace();
    return Stream.empty();
  }
}
```

It creates a stream of strings from a text file and returns the resulting stream. Here is an example of using the createStreamFromFile method:

```
static void countCharacters(String filename) {
  try (Stream<String> myStream = createStreamFromFile(filename)) {
    long cnt = myStream.flatMapToInt(String::chars).count();
    System.out.println("number of chars: " + cnt);
  }
}
```

The user correctly calls the stream factory method in the resource part of a try-with-resources block. This way, it is ensured that the stream's close() method is automatically called, both on normal and on exceptional return from the try-block.

As the stream's close() method does not invoke the underlying reader's close() method, the file stays open and none of the resources associated with the open file is released.

Your task in the exercise is to ensure that the stream's underlying readers are properly closed. Note, the createStreamFromFile method must not close the readers, because the readers must remain valid as long as the stream is used. The countCharacters method cannot close the underlying readers either, because it does not have access to the readers. Figure out a solution!

Hint: Streams have an onClose() method (defined in the super-interface BaseStream). By means of onClose() you can add *close handlers* to a stream.

Lambdas

Close Handlers

Angelika Langer
Trainer/Consultant

http://www.AngelikaLanger.com/

lab: closing streams

- a stream factory

```
Stream<String> createStreamFromFile (String filename) {
  try {
    final BufferedReader in
    = new BufferedReader(new FileReader(filename));
    return in.lines();
  } catch (FileNotFoundException e) {
    e.printStackTrace();
    return Stream.empty();
  }
}
```


lab: closing streams

- the user

```
void countCharacters(String filename) {
  try (Stream<String> myStream
           = createStreamFromFile(filename)) {
    long cnt = myStream.flatMapToInt(String::chars)
                       .count();
    System.out.println("number of chars: " + cnt);
  }
}
```

- problem
  – `Stream.close()` is called automatically
  – but does not call `BufferedReader.close()`
  – file remains open

close handlers & `onClose()`

- solution
  – add a *close handler* to the created stream

- via method from interface `Stream`
  `Stream<T> onClose(Runnable closeHandler)`
  – returns an equivalent stream with an additional close handler
  – may return itself

  – close handlers are run when stream's `close()` method is called
  – executed in the order they were added
  – all close handlers run, even if earlier handlers throw exceptions
  – first exception thrown is relayed to caller of `close()`
  (with remaining exceptions as suppressed exceptions)

add a close handler

```
Stream<String> createStreamFromFile (String filename) {
  try {
    final BufferedReader in
    = new BufferedReader(new FileReader(filename));
    return in.lines().onClose(()->in.close()) ;
  } catch (FileNotFoundException e) {
    ...
  }
}
```

- problem:
  - checked `IOException` from reader's `close()`
  - close handler is a `Runnable` => must not throw checked exceptions

exception tunnelling

- resort to *exception tunnelling*
  - via pre-defined `UncheckedIOException`

```
Stream<String> createStreamFromFile (String filename) {
  try {
    ...
    return in.lines()
             .onClose(()->{
                try { in.close(); }
                catch(IOException ioe) {
                  throw new UncheckedIOException(ioe);
                }
              });
  } catch (FileNotFoundException e) { ... }
}
```


exception handling

- user should add a catch clause
  - for the `UncheckedIOException`

```
void countCharacters(String filename) {
  try (Stream<String> myStream
            = createStreamFromFile(filename)) {
    long cnt = myStream.flatMapToInt(String::chars)
                       .count();
    System.out.println("number of chars: " + cnt);
  } catch (UncheckedIOException e) {
    e.printStackTrace();
  }
}
```

Parallel Streams

Exercise 05.01: Benchmark: Sequential vs. Parallel Execution

Step 1: Compare the performance of regular for-loops with the performance of sequential and parallel streams. For the purpose of benchmarking, search for the maximum element in a collection and implement the search via a reduce() operation.

Compare the performance of sequential and parallel execution for various types of collections:

- primitive type array
- array of the boxed type
- ArrayList
- LinkedList
- HashSet
- TreeSet
- more if you like.

In the skeletal implementation (see class TestMax) the collections will contain int / Integer elements. A simplistic benchmark frame is already provided (see class BenchmarkTest).

Study the benchmark results. Are the results plausible, surprising, or otherwise noteworthy?

Step 2: The execution phase of a search for the maximum element in a collection does no more than comparing two elements. In other words, the execution phase is dominated by memory access rather than CPU consumption.

In order to get a feeling for the issues that affect execution performance, run a second benchmark with a slightly modified functionality: trigger an expensive CPU-bound computation before the elements are compared. Map the int / Integer elements to a corresponding sine value. Use the slowSine() method from class Sine in package math, that comes with the skeleton.

Instead of measuring the performance of this:

```
BenchmarkTest.doTest(
     () -> Arrays.stream(ints)
                 .reduce(Integer.MIN_VALUE, (i, j) -> Math.max(i, j)),
     "primitive array, sequential"
);
```

Measure the performance of that:

```
BenchmarkTest.doTest(
     () -> Arrays.stream(ints)
                 .mapToDouble(Sine::slowSin)
                 .reduce(Integer.MIN_VALUE, (i, j) -> Math.max(i, j)),
     "primitive array, sequential"
);
```

Study the benchmark results. Are the results different now? Why?

Lambdas

Benchmark
Sequential vs. Parallel

Angelika Langer
Trainer/Consultant

http://www.AngelikaLanger.com/

lab: sequential vs. parallel benchmark

- compare for-loop

```
int m = Integer.MIN_VALUE;
for (int i : ints)
  if (i > m) m = i;
```

- to sequential stream

```
Arrays.stream(ints)
      .reduce(Integer.MIN_VALUE, (i, j) -> Math.max(i, j))
```

- to parallel stream

```
Arrays.stream(ints).parallel()
      .reduce(Integer.MIN_VALUE, (i, j) -> Math.max(i, j))
```

- repeat for
  - `int`-array, `Integer`-array, `ArrayList`, `LinkedList`, `HashSet`, `TreeSet`

for-loop vs. sequential stream

```
primitive array, for-loop  :  0.64 ms
primitive array, sequential:  1.36 ms

boxed array, for-loop      :  3.34 ms
boxed array, sequential    : 14.82 ms

array list, for-loop       :  3.82 ms
array list, sequential     : 19.07 ms

linked list, for-loop      :  8.68 ms
linked list, sequential    : 18.77 ms

hash set, for-loop         : 27.69 ms
hash set, sequential       : 49.43 ms

tree set, for-loop         : 22.38 ms
tree set, sequential       : 33.49 ms
```

- for-loop always faster than sequential stream

evaluation

- JIT compilers know how to optimize for-loops
  - especially array-based loops with constant size and equal stride
- streams add overhead
  - always work through a splitterator and many layers of method calls

sequential vs. parallel stream

```
primitive array, sequential:  1.36 ms
primitive array, parallel  :  0.77 ms

boxed array, sequential    : 14.82 ms
boxed array, parallel      :  9.17 ms

array list, sequential     : 19.07 ms
array list, parallel       :  9.45 ms

linked list, sequential    : 18.77 ms
linked list, parallel      : 22.99 ms

hash set, sequential       : 49.43 ms
hash set, parallel         : 28.85 ms

tree set, sequential       : 33.49 ms
tree set, parallel         : 20.28 ms
```

- major speed-up
- not convincing
- major speed-up
- really bad
- not convincing
- not convincing

evaluation

- arrays are easy to split + fast to access
  - index calculation, no overhead
- collection suffer from autoboxing
  - autoboxing almost optimized away for `ArrayList`
- linked list is expensive to split
  - copies all elements into an array + splits the array

additional work in execution phase

- compare

```
double m = Double.MIN_VALUE;
for (int i = 0; i < ints.length; i++) {
  double d = Sine.slowSin(ints[i]);
  if (d > m) m = d;
}
```

- to

```
Arrays.stream(ints)
      .mapToDouble(Sine::slowSin)
      .reduce(Double.MIN_VALUE, (i, j) -> Math.max(i, j))
```

- to

```
Arrays.stream(ints).parallel()
      .mapToDouble(Sine::slowSin)
      .reduce(Double.MIN_VALUE, (i, j) -> Math.max(i, j))
```


parallel vs. sequential vs. for-loop

```
primitive array, for-loop   : 6.55 ms
primitive array, sequential : 6.79 ms
primitive array, parallel   : 3.60 ms

boxed array, for-loop       : 6.65 ms
boxed array, sequential     : 6.70 ms
boxed array, parallel       : 3.37 ms

array list, for-loop        : 6.67 ms
array list, sequential      : 6.73 ms
array list, parallel        : 3.40 ms

linked list, for-loop       : 6.75 ms
linked list, sequential     : 6.70 ms
linked list, parallel       : 3.66 ms

hash set, for-loop          : 6.59 ms
hash set, sequential        : 6.69 ms
hash set, parallel          : 3.46 ms

tree set, for-loop          : 6.67 ms
tree set, sequential        : 6.68 ms
tree set, parallel          : 3.91 ms
```

- sequential streams nearly as fast as for-loop
- parallel streams yield major speed-up in all collections

evaluation

- previously
  - execution phase dominated by RAM access
- now
  - execution phase dominated by CPU access

- conclusion
  - parallel execution pays for CPU intensive computations

Exercise 05.02: Managing Blocking Lambdas

As a rule, functionality that is passed to stream operations shall not block or wait; for instance, your lambdas shall neither wait for locks or signals nor use blocking operations such as synchronous i/o.

This is a reasonable rule because the functionality that is passed to a stream operation is performed as a fork-join task in the fork-join common pool. Blocking lambdas might put all fork-join worker threads into a waiting or blocking state and as a result the pool will be clogged. This is detrimental because there might be other parts of your application that also want to use the fork-join common pool, but the pool cannot execute anything anymore because all its threads are waiting for something and no thread is available to execute anything.

Fortunately, the fork-join framework has a feature for proper handling of blocking tasks: the so-called "managed blocker". A managed blocker is an implementation of the ForkJoinPool.ManagedBlocker interface. It indicates to the fork-join worker thread that a task is entering a waiting or blocking state and the fork-join pool compensates for the now unavailable, blocking thread by creating a new, additional worker thread. This way the fork-join pool maintains its parallelism, i.e. it tries to have a certain number of threads actively running in parallel plus an arbitrary number of waiting or blocking threads.

If a fork-join task wants to perform a blocking or waiting activity - without risking to clog the pool - it can wrap the activity into a managed blocker and pass the managed blocker to the fork-join framework.

Here is how it works:

Without a managed blocker the fork-join task would directly invoke the blocking activity:

```
class Task extends RecursiveTask<Result> {
  …
  public Result compute() {
    …
    result = runTheBlockingActivity();
    …
    return result;
  }
}
```

With a managed blocker the fork-join task would wrap the blocking part into a managed blocker:

```
class ManagedActivity implements ForkJoinPool.ManagedBlocker {
  private volatile Result result;

  public boolean block() throws InterruptedException {
    result = runTheBlockingActivity();
    return true;
  }
  public boolean isReleasable() {
    return info != null;
  }
  public Result getResult() {
    return result;
  }
}
```

It would then pass the managed blocker to the worker thread like this:

```
class Task extends RecursiveTask<Result> {
  …
  public Result compute() {
    …
    ManagedActivity action = new ManagedActivity ();
    ForkJoinPool.managedBlock(action);
    result = action.getResult();
    …
    return result;
```

```
  }
}
```

You can manage blocking lambdas in just the same way.

The skeleton provides a Stock class with a getStockInfo() method that retrieves stock information from http://finance.yahoo.com. It connects to the URL and receives the stock information via a synchronous read operation.

```
Arrays.stream(stockSymbols)
        .parallel()
        .map(s -> Stock.getStockInfo(s))
        .map(s -> Stock.createStockData(s))
        .filter(s -> s != null)
        .reduce((s1,s2) -> s1.getChange()>s2.getChange()?s1:s2)
        .ifPresent(s -> System.out.println(s));
```

The synchronous read operation puts the worker thread into a wait states, namely waiting for data to be returned from a synchronous read. Use the ManagedBlocker to make sure that the pool creates further worker threads while other worker threads are waiting for a response from their read request.

Note: As is recommended, make sure that all your lambdas work for both parallel <u>and</u> sequential streams.

Lambdas

Blocking Lambdas

Angelika Langer
Trainer/Consultant

http://www.AngelikaLanger.com/

lab: blocking lambdas

- mapper performs synchronous socket read
  - might clog common pool

```
Arrays.stream(stockSymbols).parallel()
      .map(s -> Stock.getStockInfo(s))
      .map(s -> Stock.createStockData(s))
      .filter(s -> s != null)
      .reduce((s1,s2) -> s1.getChange()>s2.getChange()?s1:s2)
      .ifPresent(s -> System.out.println(s));
```

```
main: "TWTR",38.49,"0.00 - 0.00%"
ForkJoinPool.commonPool-worker-1: "MSFT",48.42,"0.00 - 0.00%"
ForkJoinPool.commonPool-worker-1: "YHOO",50.99,"0.00 - 0.00%"
main: "FB",76.36,"0.00 - 0.00%"
ForkJoinPool.commonPool-worker-1: "GOOG",525.26,"0.00 - 0.00%"
main: "AMZN",312.63,"0.00 - 0.00%"
ForkJoinPool.commonPool-worker-1: "AAPL",115.00,"0.00 - 0.00%"
main: "ORCL",41.93,"0.00 - 0.00%"
StockData{symbol='ORCL', price='41.93', increase=0.0}
```


define managed blocker

- wrap blocking operation into managed blocker

```
class StockInfoFetcher implements ForkJoinPool.ManagedBlocker {
    private final String symbol;
    private volatile String info = null;

    public StockInfoFetcher(String symbol) {
        this.symbol = symbol;
    }
    public boolean block() throws InterruptedException {
        if (info == null)
            info = Stock.getStockInfo(symbol);
        return true;
    }
    public boolean isReleasable() {
        return info != null;
    }
    public String getInfo() {
        return info;
    }
}
```


use managed blocker

- replace direct use of blocking operation by managed blocker

```
Function<String, String> mapper = s -> {
    StockInfoFetcher infoFetcher = new StockInfoFetcher(s);
    try { ForkJoinPool.commonPool().managedBlock(infoFetcher); }
    catch (Throwable e) { return null; }
    return infoFetcher.getInfo();
};
Arrays.stream(stockSymbols).parallel()
      .map(mapper)
      ... as before ...
```

```
main: "TWTR",38.49,"0.00 - 0.00%"
ForkJoinPool.commonPool-worker-1: "MSFT",48.42,"0.00 - 0.00%"
ForkJoinPool.commonPool-worker-2: "AMZN",312.63,"0.00 - 0.00%"
ForkJoinPool.commonPool-worker-0: "GOOG",525.26,"0.00 - 0.00%"
main: "FB",76.36,"0.00 - 0.00%"
ForkJoinPool.commonPool-worker-1: "YHOO",50.99,"0.00 - 0.00%"
ForkJoinPool.commonPool-worker-0: "AAPL",115.00,"0.00 - 0.00%"
ForkJoinPool.commonPool-worker-2: "ORCL",41.93,"0.00 - 0.00%"
StockData{symbol='ORCL', price='41.93', increase=0.0}
```

managed blocker (cont.)

Q: does mapper only work for parallel streams ?

A: also works for sequential streams

- managed blocker checks if current thread is fork-join worker thread
- if regular (main) thread, simply calls `block()`, which calls blocking i/o

use larger fork-join pool

- replace common pool by a larger fork-join pool

```
ForkJoinPool myPool = new ForkJoinPool(HIGHER_PARALLELISM);
myPool.submit(()->
 Arrays.stream(stockSymbols).parallel()
       .map(s -> Stock.getStockInfo(s))
       .map(s -> Stock.createStockData(s))
       .filter(s -> s != null)
       .reduce((s1,s2) -> s1.getChange()>s2.getChange()?s1:s2)
       .ifPresent(s -> System.out.println(s));
myPool.awaitQuiescence(FINISH_AWAIT_TIME, TimeUnit.SECONDS);
```

```
ForkJoinPool-1-worker-2: "MSFT",48.42,"0.00 - 0.00%"
ForkJoinPool-1-worker-1: "TWTR",38.49,"0.00 - 0.00%"
main: "AMZN",312.63,"0.00 - 0.00%"
ForkJoinPool-1-worker-4: "GOOG",525.26,"0.00 - 0.00%"
ForkJoinPool-1-worker-2: "YHOO",50.99,"0.00 - 0.00%"
main: "ORCL",41.93,"0.00 - 0.00%"
ForkJoinPool-1-worker-1: "FB",76.36,"0.00 - 0.00%"
ForkJoinPool-1-worker-4: "AAPL",115.00,"0.00 - 0.00%"
StockData{symbol='ORCL', price='41.93', increase=0.0}
```


evaluation

- managed blocker
  - risk of creating too many threads
  - no internal resource limit; might run out of resources
  - slightly limited by number of leaf tasks
- larger fork-join pool
  - cannot run out of resources; number of threads is limited
  - undocumented feature

make common pool larger

- conceivable alternative:
  - configure common pool with higher parallelism
  - via system property
- rarely a good idea
  - static setting at application startup
    - no support for dynamic resizing of common pool
  - larger common affects entire application
    - more threads than cores/cpus => more thread scheduling overhead

Exercise 05.03: Benchmark Ordered vs. Unordered Execution

Step 1: The distinct() Operation

Compare the performance of sequential vs. parallel ordered and unordered execution of the stream operation distinct(). Apply the distinct() operation to the stream

- without any mapping,
- with a cheap mapping, and
- with an expensive mapping

prior to the execution of the distinct() operation.

A skeletal implementation (see class DistinctTest) is provided. It uses a simplistic benchmark frame (see class BenchmarkTest).

Study the benchmark results. Are the results plausible, surprising, or otherwise noteworthy?

Step 2: The forEach() Operation

Compare the performance of sequential vs. parallel execution of a transformation performed on each element in a collection. Apply the transformation sequentially and in parallel via forEach() and forEachOrdered().

Use two different transformations:

- a cheap transformation (see class AddTransformer in package transformer)
- an expensive transformation (see class SineTransformer in package transformer)

Perform the transformation

- in the terminal forEach and forEachOrdered operation, and
- in an intermediate map operation before the forEach and forEachOrdered operation.

A skeletal implementation (see class ForEachTest) is provided.

Study the benchmark results. Are the results plausible, surprising, or otherwise noteworthy?

Lambdas

Benchmark
Ordered vs. Unordered

Angelika Langer
Trainer/Consultant

http://www.AngelikaLanger.com/

lab: distinct benchmark

- compare

```
Arrays.stream(ints)
      .distinct()
      .count()
```

- to

```
Arrays.stream(ints).parallel()
      .distinct()
      .count()
```

- and to

```
Arrays.stream(ints).parallel().unordered()
      .distinct()
      .count()
```


results (Intel dual core, 2x 3.17 GHz)

```
sequential          - int: 3.22 ms
parallel ordered    - int: 8.56 ms
parallel unordered  - int: 7.44 ms
```

- parallel `distinct`
  - substantially slower than sequential
- parallel <u>un</u>ordered `distinct`
  - only marginally faster than ordered

lab: distinct with "cheap" mapping

- compare

```
Arrays.stream(ints)
      .mapToObj(String::valueOf)
      .distinct()
      .count()
```

- to

```
Arrays.stream(ints).parallel()
      .mapToObj(String::valueOf)
      .distinct()
      .count()
```

- and to

```
Arrays.stream(ints).parallel().unordered()
      .mapToObj(String::valueOf)
      .distinct()
      .count()
```

results

```
sequential          - String: 10.24 ms
parallel ordered    - String: 15.70 ms
parallel unordered - String: 13.84 ms
```

- parallel `distinct`
  - still slower than sequential
- parallel <u>un</u>ordered `distinct`
  - still only marginally faster than ordered

lab: `distinct` with "expensive" mapping

- compare

```
Arrays.stream(ints)
  .map(i->new Double(SIZ/DIV*Sine.slowSin(i*offset))
             .intValue())
  .distinct()
  .count()
```

- to

```
Arrays.stream(ints).parallel()
  .map(i->new Double(SIZ/DIV*Sine.slowSin(i*offset))
             .intValue())
  .distinct()
  .count()
```

- and to

```
Arrays.stream(ints).parallel().unordered()
  .map(i->new Double(SIZ/DIV*Sine.slowSin(i*offset))
             .intValue())
  .distinct()
  .count()
```


results

```
sequential          - sine: 6.96 ms
parallel ordered    - sine: 4.06 ms
parallel unordered - sine: 4.22 ms
```

- parallel `distinct`
  - eventually faster than sequential
- parallel <u>un</u>ordered `distinct`
  - slightly slower (!) than ordered
  - platform specific effect
    - slower CPU => less collisions => greater performance gain

evaluation

- *parallel* may (but need not) be faster than *sequential*
  - substantial overhead (for buffering or synchronization)
- *unordered* may (but need not) be faster than *ordered*
  - no guarantee

lab: forEach benchmark

```
class AddTransformer extends Transformer {
    private double summand, value;
    ...
    public void transform() { value += summand; }
}
```

- compare
```
Arrays.stream(addTransformers)
      .forEach(st -> st.transform())
```
- to
```
Arrays.stream(addTransformers).parallel()
      .forEach(st -> st.transform()
```
- and to
```
Arrays.stream(addTransformers).parallel()
      .forEachOrdered(st -> st.transform()
```


results

```
add - sequential:          0.03 ms
add - parallel unordered:  0.05 ms
add - parallel ordered:    0.12 ms
```

- parallel ordered `forEach`
  - significantly slower (!) than sequential
  - due to overhead of task creation/scheduling
  - computation in `forEach` is executed sequentially anyway

lab: forEach benchmark

```
class SineTransformer extends Transformer {
  private double value;
  ...
  public void transform() {value = Sine.slowSin(value); }
}
```

- compare
```
Arrays.stream(SineTransformers)
      .forEach(st -> st.transform())
```
- to
```
Arrays.stream(SineTransformers).parallel()
      .forEach(st -> st.transform()
```
- and to
```
Arrays.stream(SineTransformers).parallel()
      .forEachOrdered(st -> st.transform()
```


results

```
sine - sequential:          6.52 ms
sine - parallel unordered:  3.40 ms
sine - parallel ordered:    6.70 ms
```

- parallel unordered `forEach`
  - substantial speed-up
  - computation in `forEach` is executed in parallel

lab: `forEach` with mapping

- so far,
  - cpu-expensive computation as part of terminal `forEach` operation
- what if ... ?
  - cpu-expensive computation happens in intermediate operation (before `forEach`)

lab: `forEach` with "cheap" mapping

- compare

```
Arrays.stream(doubles)
      .mapToObj(d -> new AddTransformer(d, d))
      .forEach(st -> st.transform())
```

- to

```
Arrays.stream(doubles).parallel()
      .mapToObj(d -> new AddTransformer(d, d))
      .forEach(st -> st.transform()
```

- and to

```
Arrays.stream(doubles).parallel()
      .mapToObj(d -> new AddTransformer(d, d))
      .forEachOrdered(st -> st.transform()
```


results

```
map with add - sequential:         0.14 ms
map with add - parallel unordered: 0.18 ms
map with add - parallel ordered:   0.27 ms
```

- same as before (more or less)

lab: forEach with "expensive" mapping

- compare

```
Arrays.stream(doubles)
      .map(Sine::slowSin)
      .mapToObj(d -> new AddTransformer(d, d))
      .forEach(st -> st.transform())
```

- to

```
Arrays.stream(doubles).parallel()
      .map(Sine::slowSin)
      .mapToObj(d -> new AddTransformer(d, d))
      .forEach(st -> st.transform()
```

- and to

```
Arrays.stream(doubles).parallel()
      .map(Sine::slowSin)
      .mapToObj(d -> new AddTransformer(d, d))
      .forEachOrdered(st -> st.transform()
```

results

```
map with sine - sequential:            6.62 ms
map with sine - parallel unordered:    3.92 ms
map with sine - parallel ordered:      3.63 ms
```

- conclusion
  – parallel execution pays if cpu-expensive intermediate operations are involved

evaluation

- parallel execution pays
  – if expensive intermediate operation is involved

- unordered execution pays
  – if for-each-action is expensive

Exercise 05.04: Benchmark: Sequential vs. Parallel Collect

Step 1: **String Concatenation**

Compare the performance of sequential vs. parallel string concatenation. Implement several approaches:

- using reduce() and the "+" operator for strings,
- using forEach() and the append() method of StringBuilder or StringBuffer, and
- using collect() and the joining() collector.

The skeletal implementation (see class StringConcatTest) converts integers into their respective string representations and concatenates these strings.

A simplistic benchmark frame is provided (see class BenchmarkTest).

Complete the skeleton and study the benchmark results. Are the results plausible, surprising, or otherwise noteworthy?

Step 2: **Collecting into a Collection**

Compare the performance of collecting elements into a collection sequentially vs. in parallel. Use an array of ints as the stream source, convert the ints into their respective string representations, and put the strings into a collection.

Use various types of collections as a sink:

- ArrayList
- LinkedList
- HashSet
- TreeSet
- Map
- ConcurrentMap
- more if you like.

A simplistic benchmark frame is provided (see class BenchmarkTest).

Complete the skeleton and study the benchmark results. Are the results plausible, surprising, or otherwise noteworthy?

Lambdas

Benchmark
Collect Algorithms

Angelika Langer
Trainer/Consultant

http://www.AngelikaLanger.com/

lab: benchmark string concatenation

- compare performance
  - of sequential vs. parallel string concatenation
- several approaches:
  - using `reduce()` & "+" operator for strings,
  - using `forEach()` & `append()` method of `StringBuilder` or `StringBuffer`, and
  - using `collect()` & `joining()` collector.

string concatenation - reduce & "+"

- compare sequential & parallel execution of

```
Arrays.stream(ints)
 .boxed()
 .reduce("",(i,s)->String.valueOf(i)+" "+s,(s1,s2)->s1+s2)
```

- to

```
Arrays.stream(ints)
 .mapToObj(i->String.valueOf(i)+" ")
 .reduce("",(s1,s2)->s1+s2)
```

- and to

```
Arrays.stream(ints)
 .mapToObj(i -> String.valueOf(i))
 .collect(Collectors.joining(" "))
```


results - reduce & "+"

```
'+' with boxed / sequential: 25.82 ms
'+' with boxed / parallel:    6.82 ms
'+' with map / sequential:    6.77 ms
'+' with map / parallel:      1.95 ms
joining() / sequential:       0.15 ms
joining() / parallel:         0.12 ms
```

- boxing is expensive
- string concatenation via "+" operator of string is slow
- joining is fast
  - barely any difference between sequential and parallel

string concatenation - forEach & append

- compare sequential

```
StringBuilder sb = new StringBuilder();
Arrays.stream(newInts)
  .mapToObj(i->String.valueOf(i))
  .forEach(s->{sb.append(s); sb.append(" ");})
```

- to parallel (ordered & unordered)

```
StringBuffer sb = new StringBuffer();
Arrays.stream(newInts).parallel()
  .mapToObj(i->String.valueOf(i))
  .forEachOrdered(s->{ sb.append(s); sb.append(" ");})
```

- and to

```
Arrays.stream(ints)
 .mapToObj(i -> String.valueOf(i))
 .collect(Collectors.joining(" "))
```


results - forEach & append

```
StringBuilder / sequential:          1.57 ms
StringBuffer / parallel + unordered: 2.66 ms
StringBuffer / parallel + ordered:   1.97 ms
joining() / sequential:              1.57 ms
joining() / parallel:                1.19 ms
```

- concurrent use of `StringBuffer` is expensive
  - due to contention and synchronization overhead
- unordered is massively worse than ordered
  - also due to contention and synchronization overhead
- parallel `joining` is fastest

string concatenation - joining

- compare sequential & parallel execution of

```
Arrays.stream(ints).parallel()
 .mapToObj(i -> String.valueOf(i))
 .collect(Collectors.joining(" "))
```

- to

```
Arrays.stream(ints).parallel()
  .mapToObj(i->String.valueOf(Sine.slowSin(i*0.001)))
  .collect(Collectors.joining(" "))
```


results - joining

```
joining() / sequential:           1.60 ms
joining() / parallel:             1.26 ms
joining() with sine / sequential: 2.18 ms
joining() with sine / parallel:   1.46 ms
```

- expensive mapping in execution phase
  - makes parallel execution more attractive

lab: benchmark collecting into a collection

- compare performance
  - of sequential vs. parallel collect in a collection
- several target collections
  - `ArrayList`
  - `LinkedList`
  - `HashSet`
  - `TreeSet`
  - `Map`
  - `ConcurrentMap`
- example

```
Arrays.stream(ints)
 .mapToObj(String::valueOf)
 .collect(Collectors.toCollection(ArrayList<String>::new))
```


results - collecting into a collection

```
collect to ArrayList / sequential:          2.75 ms
collect to ArrayList / parallel:            1.93 ms  43%
collect to LinkedList / sequential:         2.65 ms
collect to LinkedList / parallel:           2.42 ms   9%
collect to HashSet / sequential:            5.54 ms
collect to HashSet / parallel:              6.20 ms -11%
collect to TreeSet / sequential:           15.23 ms
collect to TreeSet / parallel:             21.38 ms -29%
collect to Map / sequential:                4.66 ms
collect to Map / parallel:                  5.14 ms -10%
collect to ConcurrentMap / sequential:     10.85 ms
collect to ConcurrentMap / parallel:        7.76 ms  40%

with sine collect to ArrayList / sequential: 5.10 ms
with sine collect to ArrayList / parallel:   2.81 ms  80%
```

- `Set` and `Map` suffer from parallel execution
- `ArrayList` and `ConcurrentMap` speed up under parallel execution
- parallel execution more attractive with expensive intermediate operations

Exercise 05.05: Implement User-Defined Collectors

Part 1: A User-Defined Collector for a Combination of Results

In this exercise we re-visit our earlier Point examples. We have been retrieving various pieces of information from a sequence of Point objects, among them

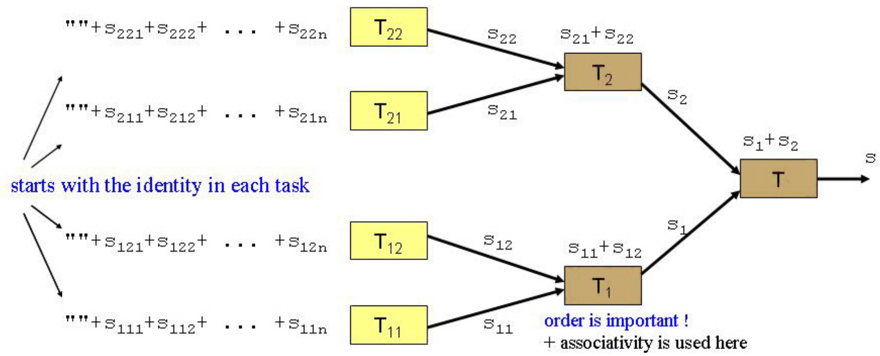
- the sum of the x-coordinates of all points,
- the sum of the distance from origin for all points with a positive x-coordinate, and
- the concatenation of the string represention of all points separated by a separator.

In this exercise you are invited to implement a collector that gathers the results in one pass over the sequence and stores all the results in a suitable data structure.

The skeleton for this part of the exercise is in class TestCollectorForSeveralPointResults.

Part 2: An Alternative Max-Collector

The stream operation max() delegates to the reduce() operation. The reduce() operation - if applied to a parallel stream - uses an accumulator in its execution phase and a combiner in its joining phase. The joining phase ensures that the encounter order of the elements in the stream source is preserved - even for combiners that are not commutative, but only associative.



If accumulator and combiner were commutative, then order would not matter and the joining phase would not be needed.

The accumulator and combiner for a max() operation are always commutative. For example, the maximum of "abc" and "xyz" is the same as the maximum of "xyz" and "abc". Consequently, a max() operation does not need a joining phase. It could instead be implemented via concurrently accumulating collect algorithm.

Your task in this exercise is to implement a concurrently accumulating collector that produces the maximum of all elements in a stream source.

Compare the performance of your newly defined collector to the performance of the JDK-provided max() operation.

Lambdas

User-Defined Collectors

Angelika Langer

Trainer/Consultant

http://www.AngelikaLanger.com/

lab: collector for a combination of results

- implement a collector that
  - gathers several results in one pass over the sequence and
  - stores all the results in a suitable data structure

```
interface PointsResults {
  int getXCoordinate();      // sum of the x-coordinates of all points
  double getDfo();           // sum of distance from origin
  String pointsAsString();   // string representation of all points
}
```

```
PointsResults ps = points.stream().parallel()
                         .collect( ... to be done ... );

System.out.println("sum x coordinate: " + ps.getXCoordinate());
System.out.println("sum dfo where x > 0: " + ps.getDfo());
System.out.println(ps.pointsAsString());
```


implement accumulator + combiner

```
class PRCollector implements PointsResults {
  private int xCoordinate = 0;
  private double dfo = 0.0;
  private StringBuilder pointsAsString = new StringBuilder();
  ... getter methods ...

  public void accept(Point p) { // accumulator
    xCoordinate += p.x;
    if (p.x > 0) dfo += Math.sqrt(p.x * p.x + p.y * p.y);
    pointsAsString.append(p.toString()).append("  ");
  }
  public PRCollector combine(PRCollector ps) { // combiner
    xCoordinate += ps.xCoordinate;
    dfo += ps.dfo;
    pointsAsString.append(ps.pointsAsString);
    return this;
  }
}
```


using the user-defined collector

- simple via `collect()` operation

```
PointsResults ps = points.stream().parallel()
  .collect(PointResultCollector::new,
           PointResultCollector::accept,
           PointResultCollector::combine);
```

lab: concurrently accumulating max collector

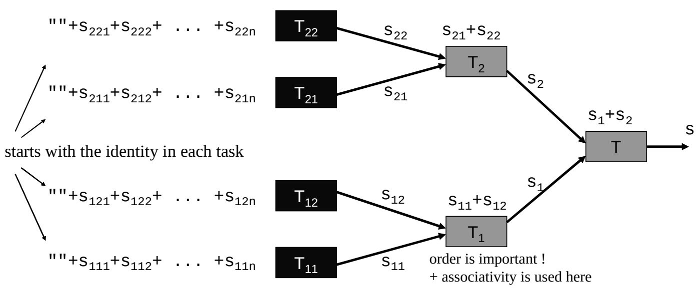
- motivation
  - determining the maximum of two values is commutative

  => order does not matter
  - implement a max collector based on concurrent accumulation (rather than reduction)

- compare the performance of
  - `max()` stream operation (based on `reduce()`), and
  - `collect()` operation with your new collector

why are identity + associativity needed ?

```
Arrays.stream(ints).parallel().reduce("", (s1,s2) -> s1+s2);
```



- does not work with a non-associative operation, e.g. "`-`" for `int`

approach #1

- collector must have a thread-safe accumulation function
  - will be invoked concurrently in execution phase
- use an `AtomicReference`
  - as the container to hold the maximum value

```
class ConcurrentMaxFinder {
  private AtomicReference<String> max
                              =  new AtomicReference<>("");
  public void accept(String s) {
    String old = null;
    do {
         old = max.get();
         if (s.compareTo(old) <= 0)
            break;
    } while(!max.compareAndSet(old,s));
  }
}
```


approach #1 (cont.)

- combiner must be provided, but will not be used

```
class ConcurrentMaxFinder {
  ...
  public ConcurrentMaxFinder combine(ConcurrentMaxFinder cmf) {
    return (max.get().compareTo(cmf.max.get()) >= 0 ? this : cmf);
  }
  String getMax() {
    return max.get();
  }
}
```

approach #1 (cont.)

- create collector via `Collector.of()`

```
Collector<String,ConcurrentMaxFinder,String> collector
  = Collector.of(ConcurrentMaxFinder::new,
                 ConcurrentMaxFinder::accept,
                 ConcurrentMaxFinder::combine,
                 ConcurrentMaxFinder::getMax,
                 CONCURRENT, UNORDERED);
String m = stringList.stream().parallel().collect(collector);
System.out.println("max: " + m);
```


conceivable mistake

- use stream operation `collect()`

```
String m = stringList.stream().parallel()
                     .collect(ConcurrentMaxFinder::new,
                              ConcurrentMaxFinder::accept,
                              ConcurrentMaxFinder::combine)
                     .getMax();
System.out.println("max: " + m);
```

- has two bugs
  1. performs a reduction-based collect
     - due to default characteristics
  2. might produce incorrect results
     - due to incorrect combiner implementation

conceivable mistake (cont.)

- subtle difference regarding combiner
  - `collect()`'s combiner is of type `BiConsumer<A,A>`
    - returns `void`

    => must combine, i.e. merge second into first 'collection'

    => first 'collection' is mutated => mutation required
  - `of()` factory's combiner is of type `BinaryOperator<A>`
    - returns the resulting combined 'collection'
    - may return a new collection => no mutation necessary
    - may merge and return the mutated 'collection'

- we implemented a `BinaryOperator`,
  - but `collect()` uses is as a `BiConsumer`, i.e. ignores the result

```
public ConcurrentMaxFinder combine(ConcurrentMaxFinder cmf) {
   return (max.get().compareTo(cmf.max.get()) >= 0 ? this : cmf);
}
```


approach #2

- implement `Collector` interface from scratch

```
class MyCollector
implements Collector<String, AtomicReference<String>, String> {

  ...
}
```

- type parameters
  - `T:=String` - type of input elements
  - `A:=AtomicReference<String>` - mutable accumulation type
  - `R:=String` - result type

supplier

- creation of a new result container

```
class MyCollector
implements Collector<String, AtomicReference<String>, String> {

  public Supplier<AtomicReference<String>> supplier() {
    return (() -> new AtomicReference<String>(""));
  }
  ...
}
```


accumulator

- incorporating a new data element into a result container

```
class MyCollector
implements Collector<String,AtomicReference<String>,String> {
  ...
  public BiConsumer<AtomicReference<String>,String> accumulator() {
    return ((max, s) -> {
      String old = null;
      do {
         old = max.get();
         if (s.compareTo(old) <= 0)
            break;
      } while(!max.compareAndSet(old, s));
    });
  }
  ...
}
```


combiner

- combining two result containers into one

```
class MyCollector
implements Collector<String,AtomicReference<String>,String> {
  ...
  public BinaryOperator<AtomicReference<String>> combiner() {
    return (m1,m2)->m1.get().compareTo(m2.get())>=0 ? m1 : m2;
  }
  ...
}
```


finisher

- performing an optional final transform on the container

```
class MyCollector
implements Collector<String,AtomicReference<String>,String> {
  ...
  public Function<AtomicReference<String>, String> finisher() {
          return (m -> m.get());
  }
}
```

characteristics

- provide hints that trigger concurrent accumulation
  - used by `collect()`

```
class MyCollector
implements Collector<String,AtomicReference<String>,String> {
  ...
  public Set<Characteristics> characteristics() {
    return Collections.unmodifiableSet(
                              EnumSet.of(CONCURRENT, UNORDERED));
  }
}
```


approach #2 (cont.)

- create new collector

```
String m = stringList.stream().parallel()
                     .collect(new MyCollector());
System.out.println("max: " + m);
```


benchmark results - platform #1

```
max() / sequential:                       17.04 ms
max() / parallel:                         11.04 ms

collect() (Collector.of) / sequential: 14.62 ms
collect() (Collector.of) / parallel:   10.59 ms

collect() (MyCollector) / sequential:  15.80 ms
collect() (MyCollector) / parallel:    11.10 ms
```

- concurrently accumulating max is only slightly faster, if at all
  - on dual core (2x 3.16 GHz)

benchmark results- platform #2

```
max() / sequential:                       29.79 ms
max() / parallel:                         31.71 ms

collect() (Collector.of) / sequential: 27.01 ms
collect() (Collector.of) / parallel:   17.24 ms

collect() (MyCollector) / sequential:  25.72 ms
collect() (MyCollector) / parallel:    16.08 ms
```

- concurrently accumulating max is substantially faster
  - on dual core (2x 2.2 GHz) , i.e. a slower CPU

- reason:
  - faster CPU produces more collisions and retries on `AtomicReference`

Exercise 05.06: Reduce vs. Collect

Explore the difference between reducing and collecting using string concatenation as an example.

Throughout the seminar we have seen various ways of concatenating strings:

Using a joining() collector.

```
Stream<String> strings = …;
String concatenated = strings.collect(Collectors.joining());
```

Using a user-defined collector created with Collector.of():

```
Stream<String> strings = …;
String concatenated
= strings.collect(Collector.of(StringBuilder::new,
                               StringBuilder::append,
                               StringBuilder::append,
                               StringBuilder::toString)
                 );
```

Using the 3-argument version of collect():

```
Stream<String> strings = …;
String concatenated = strings.collect(StringBuilder::new,
                                      StringBuilder::append,
                                      StringBuilder::append)
                             .toString();
```

The last version uses the 3-argument version of the Stream operation collect(), which takes a supplier, an accumulator, and a combiner:

```
<R> R collect(Supplier<R> supplier,
              BiConsumer<R, ? super T> accumulator,
              BiConsumer<R, R> combiner)
```

There is a similar 3-argument version of the Stream operation reduce(), which takes an identity value, an accumulator, and a combiner:

```
<U> U reduce(U identity,
             BiFunction<U, ? super T, U> accumulator,
             BinaryOperator<U> combiner)
```

Is it possible to implement string concatenation using the 3-argument version of reduce()? Try it and test it - with sequential and parallel streams!

Lambdas

Reducing vs. Collecting

Angelika Langer
Trainer/Consultant

http://www.AngelikaLanger.com/

lab: reducing vs. collecting

- implement string concatenation using the 3-argument version of `reduce()`
  - in analogy to the 3-argument version of `collect()`

```
<R> R collect(Supplier<R> supplier,
              BiConsumer<R, ? super T> accumulator,
              BiConsumer<R, R> combiner)
```

```
Stream<String> strings = …;
String concatenated = strings.collect(StringBuilder::new,
                                      StringBuilder::append,
                                      StringBuilder::append)
                             .toString();
```


obvious approach

- 3-argument version of `reduce()`

```
<U> U reduce(U identity,
             BiFunction<U, ? super T, U> accumulator,
             BinaryOperator<U> combiner)
```

```
Stream<String> strings = …;
String concatenated = strings.reduce(new StringBuilder(),
                                     StringBuilder::append,
                                     StringBuilder::append)
                             .toString();
```

- only difference:
  - `collect()` takes a supplier where `reduce()` takes an identity value

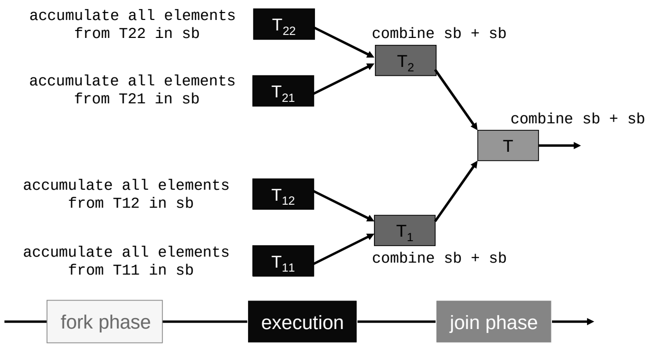
fails under parallel execution

- prints

```
SEQUENTIAL
(collect) Hänschen klein ging allein in die weite Welt hinein
(reduce)  Hänschen klein ging allein in die weite Welt hinein
PARALLEL
(collect) Hänschen klein ging allein in die weite Welt hinein
(reduce)  in die Welt hinein weite in die Welt hinein weite ging
allein in die Welt hinein weite in die Welt hinein weite ging allein
Hänschen klein in die Welt hinein weite in die Welt hinein weite ging
allein in die Welt hinein weite in die Welt hinein weite ging allein
Hänschen klein in die Welt hinein weite in die Welt hinein weite ging
allein in die Welt hinein weite in die Welt hinein weite ging allein
Hänschen klein in die Welt hinein weite in die Welt hinein weite ging
allein in die Welt hinein weite in die Welt hinein weite ging allein
Hänschen klein
```

parallel reduce

- uses only one `StringBuilder`
  - namely the identity value

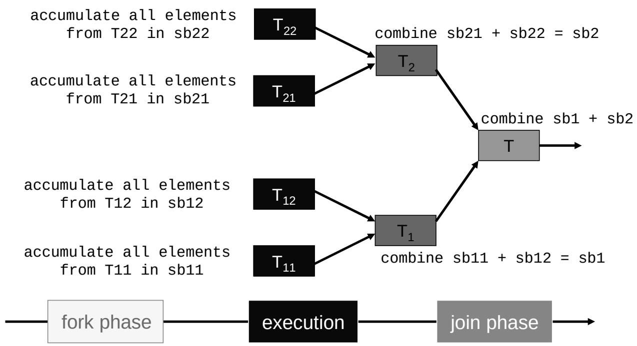


parallel reduce

- execution phase:
  - all threads accumulate into same `StringBuilder`
- join phase:
  - combines same `StringBuilder` repeatedly with itself
- realization
  - a parallel *mutating* reduction is neither deterministic nor thread-safe

parallel collect

- uses a new `StringBuilder` per thread
  - the one that the supplier creates



parallel collect

- execution phase:
  - each thread accumulates into a different `StringBuilder`
- join phase:
  - combines the different `StringBuilder`s in correct order
- realization
  - a parallel collect is both deterministic and thread-safe

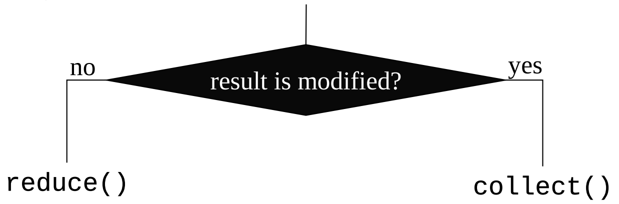
point to take home

- `reduce()` is for *non-mutating* reduction
- `collect()` is for *mutating* reduction

- difficult to understand
  - without knowing the underlying algorithms
- only hint is in the javadoc of `collect()`:
  - *Performs a **mutable reduction** operation on the elements of this stream. A mutable reduction is one in which the reduced value is a mutable result container ...*
  - more information in section on "Mutable Reduction" in javadoc of package `java.util.stream`

decision chart

goal: reduce all stream elements to one result value



correct, but highly inefficient approach

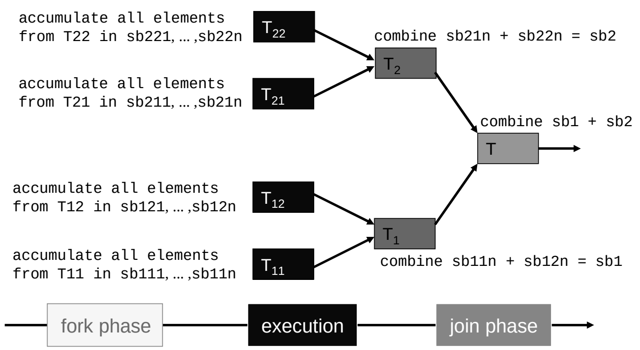
- do not mutate the result value
  - create an new `StringBuilder` in each step

```
Stream<String> strings = …;
String concatenated
= strings.reduce(new StringBuilder(),
                 (sb,s)->new StringBuilder(sb).append(s),
                 (sb1,sb2)->new StringBuilder(sb1).append(sb2))
         .toString();
```

- `StringBuilder` is effectively treated as an immutable type
- boils down to the inefficient reduction using `String`'s `+`-operator

parallel reduce

- creates a new `StringBuilder` for each accumulation step (and for each combination step, of course)



Extensions

Exercise 06.01: Creating a Stream from a char[] Array

Say, you need to create a stream from a given array of type char[]. Class Arrays has several static methods for creating streams from arrays. For instance, you can create an IntStream from an int[] or a Stream<Character> from a Character[]. But there is nothing to create a stream from a char[].

To make up for this deficiency, build a stream from a char[] as underlying stream source. Consider several approaches:

1 Implement an Iterator<Character> and use the factory method Spliterators.spliterator() to create a spliterator from this iterator. Use this spliterator to create a Stream<Character> backed by the char[] as underlying stream source.

2 Similar to the approach above: Implement an iterator of type PrimitiveIterator.OfInt and use it to create an IntStream backed by the char[] as underlying stream source.

3 Implement a Spliterator.OfInt from scratch and use it to create an IntStream backed by the char[] as underlying stream source. Hint: Copy the source code of Spliterators.IntArraySpliterator and change the type of the underlying array from int[] to char[].

4 Implement a Spliterator.OfInt by deriving from Spliterators.AbstractIntSpliterator and implement the abstract tryAdvance() method. Use the newly defined spliterator type to create an IntStream backed by the char[] as underlying stream source. Hint: Use the same implementation technique as in approach #3, i.e., copy the source code of Spliterators.IntArraySpliterator and change the type of the underlying array from int[] to char[].

5 Same as above, but this time do not inherit the spliterator's forEachRemaining() method, instead override it. Hint: Use the same implementation technique as in approach #3, i.e., copy the source code of Spliterators.IntArraySpliterator and change the type of the underlying array from int[] to char[].

Compare the (sequential and parallel) performance of the various approaches by means of a benchmark.

Lambdas

Stream for char[]

Angelika Langer

Trainer/Consultant

http://www.AngelikaLanger.com/

lab: stream backed by char[]

- build a stream from a `char[]`
- several approaches
  1. build `Iterator<Character>`
     create spliterator from iterator (via `Spliterators.spliterator()`)
     create `Stream<Character>` from spliterator
  2. similar as above: build `PrimitiveIterator.OfInt`
     create spliterator from iterator
     create `IntStream` from spliterator
  3. build `Spliterator.OfInt` from scratch
     create `IntStream` from spliterator
     (hint: copy source code of `IntArraySpliterator` and change `int[]` to `char[]`)
  4. build `Spliterator.OfInt` by deriving from `AbstractIntSpliterator`
     i.e. implement abstract `tryAdvance()`
     create `IntStream` from spliterator
  5. same as above, but this time override spliterator's `forEachRemaining()` method

#1: with Iterator<Character>

- build `Iterator<Character>`

```
class CharArrayIterator implements Iterator<Character> {
  private int i = 0;
  private final char[] array;

  public CharArrayIterator(char[] array) {
    this.array = array;
  }
  public boolean hasNext() {
    return (i < array.length ? true : false);
  }
  public Character next() {
    if (i < array.length) return array[i++];
    else                  throw new NoSuchElementException();
  }
}
```


#1: with Iterator<Character> (cont.)

- create spliterator from iterator
  - characteristics SIZED and SUBSIZED are automatically added
- create `Stream<Character>` from spliterator

```
public static Stream<Character> streamOfCharArray(char[] array) {
  Iterator<Character> iter = new CharArrayIterator(array);

  Spliterator<Character> spliter
  = Spliterators.spliterator(iter, array.length,
                             Spliterator.ORDERED   |
                             Spliterator.IMMUTABLE |
                             Spliterator.NONNULL);

  return StreamSupport.stream(spliter, false);
}
```

#2: with PrimitiveIterator.OfInt

- build `PrimitiveIterator.OfInt`

```
class CharArrayIterator implements PrimitiveIterator.OfInt {
  private int i = 0;
  private final char[] array;

  public CharArrayIterator(char[] array) {
    this.array = array;
  }
  public boolean hasNext() {
    return (i < array.length ? true : false);
  }
  public int nextInt() {
    if (i < array.length) return array[i++];
    else                  throw new NoSuchElementException();
  }
}
```


#2: with PrimitiveIterator.OfInt (cont.)

- create spliterator from iterator
  - characteristics SIZED and SUBSIZED are automatically added
  - no longer `NONNULL` due to primitive type
- create `IntStream` from spliterator

```
public static IntStream streamOfCharArray(char[] array) {
  PrimitiveIterator.OfInt iter = new CharArrayIterator(array);

  Spliterator.OfInt spliter
  = Spliterators.spliterator(iter, array.length,
                             Spliterator.ORDERED |
                             Spliterator.IMMUTABLE);

  return StreamSupport.intStream(spliter, false);
}
```


#3: `Spliterator.OfInt` from scratch

- build `Spliterator.OfInt` from scratch
  - copy source of `Spliterators.IntArraySpliterator`
    + change `int[]` to `char[]`

```
class CharArraySpliterator implements Spliterator.OfInt {
  private final char[] array;
  private int index;          // current index, modified on advance/split
  private final int fence;    // one past last index
  private final int characteristics;

  public CharArraySpliterator(char[] array,
                              int additionalCharacteristics) {
    this(array, 0, array.length, additionalCharacteristics);
  }
  …
  public OfInt trySplit() {
    int lo = index, mid = (lo + fence) >>> 1;
    return (lo >= mid) ? null : new CharArraySpliterator(…);
  }
}
```


#3: `Spliterator.OfInt` from scratch (cont.)

- create `IntStream` from spliterator

```
public static IntStream streamOfCharArray(char[] array) {
  Spliterator.OfInt spliter
  = new CharArraySpliterator(array, Spliterator.ORDERED |
                                    Spliterator.IMMUTABLE |
                                    Spliterator.SIZED |
                                    Spliterator.SUBSIZED);

  return StreamSupport.intStream(spliter, false);
}
```

#4: derive from AbstractIntSpliterator

- build `Spliterator.OfInt` by deriving from `AbstractIntSpliterator`
  - i.e. implement abstract `tryAdvance()`
  - copied from `Spliterators.IntArraySpliterator`

```
class CharArrayDerivedSpliterator
extends Spliterators.AbstractIntSpliterator
implements Spliterator.OfInt {
  private final char[] array;
  private int index;         // current index, modified on advance/split
  private final int fence;   // one past last index
  public CharArrayDerivedSpliterator(char[] array,
                                     int additionalCharacteristics) {
    ... as before ...
  }
  public boolean tryAdvance(IntConsumer action) {
    if (action == null) throw new NullPointerException();
    if (index >= 0 && index < fence) {
       action.accept(array[index++]);
       return true;
    }
    return false;
  }
}
```


#5: override forEachRemaining()

- derive from `AbstractIntSpliterator`
  - and override `forEachRemaining()`
  - copied from `Spliterators.IntArraySpliterator`

```
class CharArrayDerivedSpliterator
extends Spliterators.AbstractIntSpliterator
implements Spliterator.OfInt {
  ...
  public boolean tryAdvance      (IntConsumer action) { ... as before ...  }
  public void    forEachRemaining(IntConsumer action) {
    char[] a;
    int i, hi; // hoist accesses and checks from loop
    if (action == null) throw new NullPointerException();
    if ((a = array).length >= (hi = fence) &&
        (i = index) >= 0 && i < (index = hi)) {
      do {
        action.accept(a[i]);
      } while (++i < hi);
    }
  }
}
```


benchmark

```
int[]  - IntStream (JDK with Arrays.stream(),                         seq:   0.56 ms
int[]  - IntStream (JDK with Arrays.stream()),                        par:   0.32 ms

Character[] - Stream<Character> (JDK with Arrays.stream()),           seq:   0.61 ms
Character[] - Stream<Character> (JDK with Arrays.stream()),           par:   0.33 ms

char[] - Stream<Character>:#1 (spliterator from Iterator<Character>),  seq:  77.50 ms
char[] - Stream<Character>:#1 (spliterator from Iterator<Character>),  par: 253.94 ms

char[] - IntStream:#2 (spliterator from PrimitiveIterator.OfInt),      seq:   0.58 ms
char[] - IntStream:#2 (spliterator from PrimitiveIterator.OfInt),      par:  18.35 ms

char[] - IntStream:#3 (Spliterator.OfInt from scratch),                seq:   0.60 ms
char[] - IntStream:#3 (Spliterator.OfInt from scratch),                par:   0.33 ms

char[] - IntStream:#4 (derived from AbstractIntSpliterator),           seq:  56.72 ms
char[] - IntStream:#4 (derived from AbstractIntSpliterator),           par:  33.90 ms

char[] - IntStream:#5 (derived w/ forEachRemaining()),                 seq:   0.60 ms
char[] - IntStream:#5 (derived w/ forEachRemaining()),                 par:  20.32 ms
```


benchmark

```
int[]  - IntStream (JDK with Arrays.stream(),                         seq:   0.56 ms
int[]  - IntStream (JDK with Arrays.stream()),                        par:   0.32 ms

Character[] - Stream<Character> (JDK with Arrays.stream()),           seq:   0.61 ms
Character[] - Stream<Character> (JDK with Arrays.stream()),           par:   0.33 ms

...

char[] - IntStream:#3 (Spliterator.OfInt from scratch),                seq:   0.60 ms
char[] - IntStream:#3 (Spliterator.OfInt from scratch),                par:   0.33 ms

...
```

- equally fast solutions
- not suprising:
  - "spliterator from scratch" based on copy of `IntArraySpliterator`
- `Character[]` profits from the fact ...
  - ... that the benchmark does not access the array elements
  - otherwise indirection through reference would cost

benchmark

```
...

char[] - Stream<Character>:#1 (spliterator from Iterator<Character>),  seq:  77.50 ms
char[] - Stream<Character>:#1 (spliterator from Iterator<Character>),  par: 253.94 ms

char[] - IntStream:#2 (spliterator from PrimitiveIterator.OfInt),      par:   0.58 ms
char[] - IntStream:#2 (spliterator from PrimitiveIterator.OfInt),      par:  18.35 ms

char[] - IntStream:#3 (Spliterator.OfInt from scratch),                seq:   0.60 ms
char[] - IntStream:#3 (Spliterator.OfInt from scratch),                par:   0.33 ms

...
```

- boxing costs a lot
  - done on each invocation of iterator's `next()` method
- spliterators created from iterators are slow in parallel case
  - do not split well
    - `trySplit()` copies into an array + returns an `ArraySpliterator`

benchmark

```
...

char[] - IntStream:#3 (Spliterator.OfInt from scratch),                seq:   0.60 ms
char[] - IntStream:#3 (Spliterator.OfInt from scratch),                par:   0.33 ms

char[] - IntStream:#4 (derived from AbstractIntSpliterator),           seq:  56.72 ms
char[] - IntStream:#4 (derived from AbstractIntSpliterator),           par:  33.90 ms

char[] - IntStream:#5 (derived w/ forEachRemaining()),                 seq:   0.60 ms
char[] - IntStream:#5 (derived w/ forEachRemaining()),                 par:  20.32 ms
```

- derived spliterator
  - slow with default `forEachRemaining()`
    - even in parallel case (due to huge segment per task)
  - better with overriden `forEachRemaining()`
    - still splits poorly (same as with spliterators created from iterators)

benchmark

```
...

char[] - IntStream:#2 (spliterator from PrimitiveIterator.OfInt),      seq:   0.58 ms
char[] - IntStream:#2 (spliterator from PrimitiveIterator.OfInt),      par:  18.35 ms

char[] - IntStream:#3 (Spliterator.OfInt from scratch),                seq:   0.60 ms
char[] - IntStream:#3 (Spliterator.OfInt from scratch),                par:   0.33 ms

char[] - IntStream:#4 (derived from AbstractIntSpliterator),           seq:  56.72 ms
char[] - IntStream:#4 (derived from AbstractIntSpliterator),           par:  33.90 ms

char[] - IntStream:#5 (derived w/ forEachRemaining()),                 seq:   0.60 ms
char[] - IntStream:#5 (derived w/ forEachRemaining()),                 par:  20.32 ms
```

- derived spliterator
  - still splits poorly
    - no optimized `trySplit()`
  - same as with spliterators created from iterators

performance tuning

- performance of iterator's `forEachRemaining()` has impact on spliterator's `forEachRemaining()`
  - platform dependent

default implementation in interface `Iterator<E>`

```
default void forEachRemaining(Consumer<? super E> action) {
  Objects.requireNonNull(action);
  while (hasNext())
    action.accept(next());
}
```

- overhead:
  - two method calls per loop step
  - indirect access to iterator's field via `this` reference

performance tuning (cont.)

- loop only uses local variables
  - enables caching
- do-while-loop faster than while-loop

improved implementation of `Iterator<Character>`

```
public void forEachRemaining(Consumer<? super Character> action) {
  Objects.requireNonNull(action);
  char[] a = array;
  int    l = array.length;
  int    j = i;
  i = l;
  if (j >= 0 && j < l) {
     do { action.accept(a[j]); } while (++j < l);
  }
}
```

Contact Info

Angelika Langer Training & Mentoring
Neumarkter Str. 86d
81673 München, Germany
email: contact@AngelikaLanger.com
http://www.AngelikaLanger.com /
twitter: @AngelikaLanger