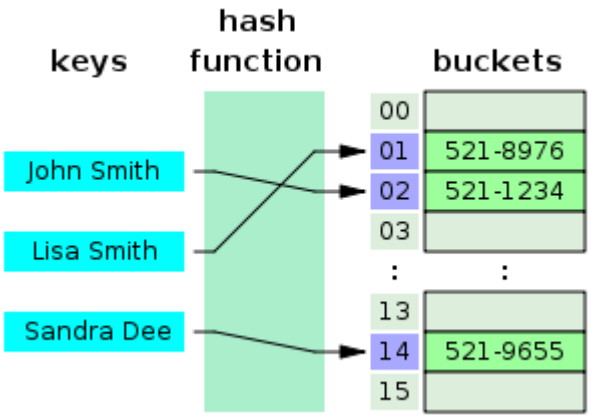
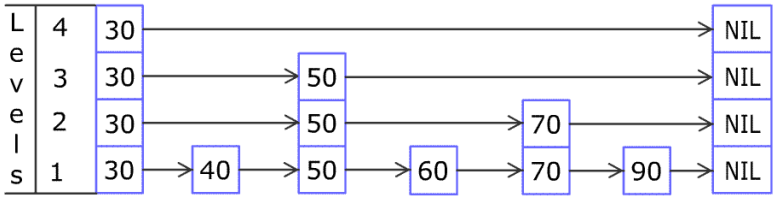
# COSC 222 Data Structures

Yves Lucet




"Hash table 3 1 1 0 1 0 0 SP" by Jorge Stolfi - Own work. Licensed under CC BY-SA 3.0 via Commons - https://commons.wikimedia.org/wiki/File:Hash_table_3_1_1_0_1_0_0_SP.svg#/media/File:Hash_table_3_1_1_0_1_0_0_SP.svg



# COSC 222

Generics

Complexity

Streams

| Wk | Class | Date | Activity | Reading/Prep given | Lab | Peer |
|---|---|---|---|---|---|---|
| 1 | | Sep 06 | **NO CLASS** | | | |
| | 1 | Sep 08 | blacklists, Syllabus, TBL | Java, Generics, Testing | | |
| 2 | 2 | Sep 13 | RAT: generics: unit testing | Complexity | 1 unit testing | |
| | 3 | Sep 15 | Lect: Complexity, streams | Lists | | |
| 3 | 4 | Sep 20 | Build & Critic (training) | iMAT | 2 coverage testing | |
| | 5 | Sep 22 | tMAT | Recursion | | |
| 4 | 6 | Sep 27 | RAT | Stack, Queue | 3 generics | iPeer1 |
| | 7 | Sep 29 | Build & Critic | Iterators | | |
| 5 | 8 | Oct 04 | mini-lecture+exercises | iMAT | 4 streams | |
| | 9 | Oct 06 | tMAT | Trees | | |
| 6 | 10 | Oct 11 | RAT | Priority Queues, heap | 5 Data Structure | iPeer2 |
| | 11 | Oct 13 | Dijsktra+Competition ex. | | | |
| 7 | 12 | Oct 18 | Jigsaw: sparse table | Hash Tables, skip list | 6 iterator | |
| | 13 | Oct 20 | Union-find+Build & Critic | | | |
| 8 | 14 | Oct 25 | Summary+Competition ex. | Union-find; iMAT | 7 collisions | |
| | 15 | Oct 27 | tMAT | Search Trees AVL/RB | | |
| 9 | 16 | Nov 01 | RAT | B-Trees | 8 skip list | iPeer3 |
| | 17 | Nov 03 | Lecture: RB trees | kd-Trees | | |
| 10 | 18 | Nov 08 | Lecture: kd-Trees | iMAT | NO LAB | |
| | 19 | Nov 10 | Midterm break | | | |
| 11 | 19 | Nov 15 | tMAT | Text processing | 9 search tree | |
| | 20 | Nov 17 | Lecture | KMP, BM | | |
| 12 | 21 | Nov 22 | RAT | Tries | 10 search engine | |
| | 22 | Nov 24 | Huffman coding | iMAT | | |
| 13 | 23 | Nov 29 | tMAT | | NO LAB | iPeer4 |
| | 24 | Dec 01 | Review/Course Evaluation | | | |

Legend: Assessment:
RAT: Readiness Assessment Test: iClicker + scratch test
tMAT: team Module Assessment Test: Code challenge=build & critic
iMAT: individual Module Assessment Test: Connect quiz

# Survey: Generics

# Do not work around generics!

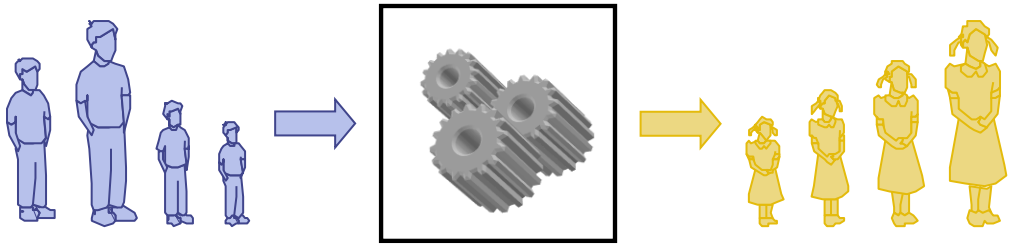
```
List<?>[] lsa = new List<?>[10];
Object[] oa = lsa;//OK because List<String> is a subtype of Object
List<Integer> li = new ArrayList<Integer>();
li.add(Integer.valueOf(3));
oa[0] = li;
String s = lsa[0].get(0).toString();
System.out.println(s);
```

The code works, but

1. is complicated, and very hard to read
2. is not taking advantage of generics

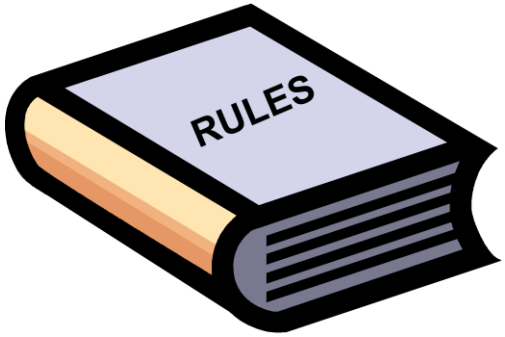
Keep it simple!

# Analysis of Algorithms

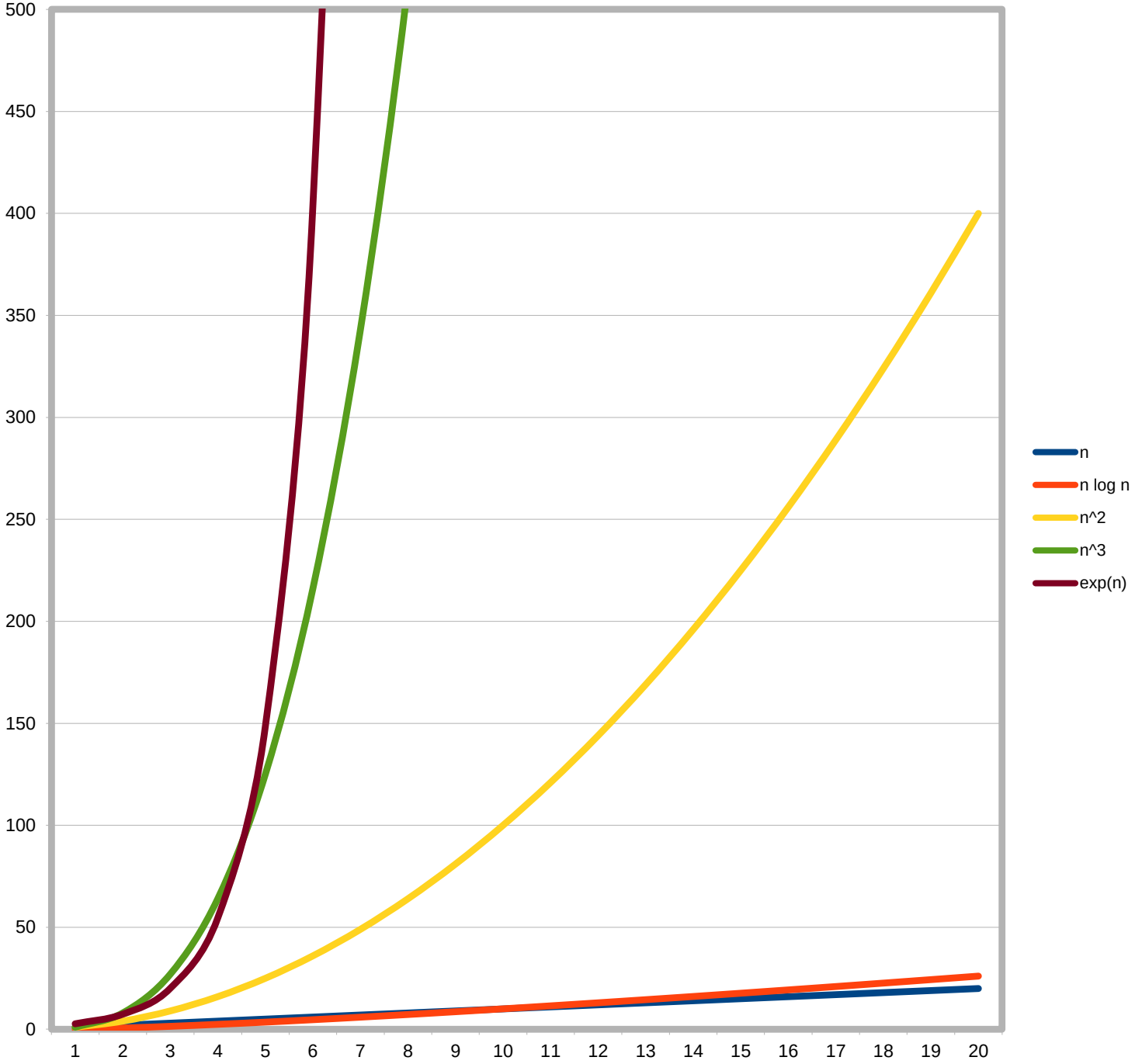



An **algorithm** is a step-by-step procedure for solving a problem in a finite amount of time.

# Big-Oh Rules




- If is $f(n)$ a polynomial of degree $d$, then $f(n)$ is $O(n^d)$, i.e.,
  1. Drop lower-order terms
  2. Drop constant factors
- Use the smallest possible class of functions
  - Say "$2n$ is $O(n)$" instead of "$2n$ is $O(n^2)$"
- Use the simplest expression of the class
  - Say "$3n + 5$ is $O(n)$" instead of "$3n + 5$ is $O(3n)$"


500
450
400
350
300
250
200
150
100
50
0
1
2
3
4
5
6
7
8
9
10
11
12
13
14
15
16
17
18
19
20
n
n log n
n^2
n^3
exp(n)

# Tips

- log < polynomial < exponential
- Polynomials are ordered by increasing powers
- Most common complexities: $\log n$, $n$, $n \log n$, $n^2$, $2^n$, $e^n$

# Survey


InteDashboard™
Empowering Team-Based Learning With Technology

# Streams in Java 8

# Javascript jQuery

```
$(document).ready(function(){
    $("div").find("span").css({"color": "red", "border": "2px solid red"});
});
```

Get the unique surnames in uppercase of the first 15 book authors that are 50 years old or over.

```
library.stream()
       .map(book -> book.getAuthor())
       .filter(author -> author.getAge() >= 50)
       .distinct()
       .limit(15)
       .map(Author::getSurname)
       .map(String::toUpperCase)
       .collect(toList());
```

Stream

Lambda expression

```
int sum = 0;
for (int x : numbers) {
    sum += x;
}
```

```
int sum = numbers.stream().reduce(0, (x,y) -> x+y);
```

# Lambda expressions

- Anonymous inner classes
- short form replacement for anonymous inner class
  - neat and concise syntax
  - easy to use and make program more readable

# Anonymous Inner Classes

- provide a way to implement classes that may occur only once in an application

```
JButton testButton = new JButton("Test Button");
testButton.addActionListener(new ActionListener(){
        @Override public void actionPerformed(ActionEvent ae){
                System.out.println("Click Detected by Anon Class");
        }
});
```

# Functional Interfaces

- an interface with only one method

```
package java.awt.event;
import java.util.EventListener;

public interface ActionListener extends EventListener {

public void actionPerformed(ActionEvent e);

}
```

# Functional Interfaces

- an interface with only one method
  - Exception: other methods must be methods defined in java.lang.Object

```
public interface SayHello {
        String say(String message);
        String toString();
}
```

```java
public class FirstExample {

    interface SayHello {
        String say(String message);
    }

    public static void main(String[] args) {

        //Saying hello through anonymous class
        SayHello helloAnonymous = new SayHello() {

            @Override
            public String say(String message) {
                return "Hello " + message;
            }
        };
        System.out.println(helloAnonymous.say("Annonymous Class"));

        //Saying hello through lambda expression
        SayHello helloLambdaExpr = (String message) -> "Hello " + message;
        System.out.println(helloLambdaExpr.say("Lambda Expression"));
    }
}
```

http://javaattop.blogspot.in/2013/09/lambda-expressions-in-java.html

RULE 1:

If the abstract method of functional interface is a zero argument method, then in the left hand side of the arrow(->) we must use an empty parentheses.

```
public class SecondExample {

    interface StaticMessage {
        String say();
    }

    public static void main(String[] args) {

        //Observe the empty parentheses
        StaticMessage msg = () -> "A Fixed Message";
        System.out.println(msg.say());
    }
}
```

http://javaattop.blogspot.in/2013/09/lambda-expressions-in-java.html

## RULE 2:

If the abstract method of functional interface is a one-argument method, then the parentheses is not mandatory.

```
public class ThirdExample {

    interface SayHello {
        String say(String message);
    }

    public static void main(String[] args) {

        SayHello sh = message -> "Hello " + message;
        System.out.println(sh.say("Lambda Expression"));
    }
}
```

http://javaattop.blogspot.in/2013/09/lambda-expressions-in-java.html

## RULE 4:

Mentioning type of the formal parameters are not mandatory. In case you have not mentioned the type of formal parameters, then its type will be determined by the Java compiler from the corresponding Target Type.

```
SayHello sh =
(msg, name, sex) -> {
        if(sex == Sex.MALE) {
                return "Hello Mr. " + name + ", " + msg;
        } else {
                return "Hello Ms. " + name + ", " + msg;
        }
};
```

http://javaattop.blogspot.in/2013/09/lambda-expressions-in-java.html

RULE:

The body of a lambda expression can either be a single expression or one or more statements.

```
public class FifthExample {
   interface SayHello {
      String hello(String name);
   }

   public static void main(String...args) {

   //Saying hello through lambda expression using single expression
   SayHello sh1 = msg -> "Hello " + msg;
   System.out.println(sh1.hello("Lambda Expression With Expression"));

   //Saying hello through lambda expression using statement
   SayHello sh2 = msg -> { return "Hello " + msg; };
   System.out.println(sh2.hello("Lambda Expression With Statement"));

   //With multiple statements
   SayHello sh3 = msg -> { String hello = "Hello " + msg;
      return hello;
   };
   System.out.println(sh3.hello("Lambda Expression With Multiple Statement"));
   }
}
```



http://javaattop.blogspot.in/2013/09/lambda-expressions-in-java.html

# Example: Comparators

```
Comparator<Person> comparator = (p1, p2) -> p1.firstName.compareTo(p2.firstName);

Person p1 = new Person("John", "Doe");
Person p2 = new Person("Alice", "Wonderland");

comparator.compare(p1, p2);             // > 0
comparator.reversed().compare(p1, p2);  // < 0
```



http://winterbe.com/posts/2014/03/16/java-8-tutorial/

# Lambda Expression in Array Initializers:

```
public class ArrayInitExample {

    interface Algebra {
        int operate(int a, int b);
    }

    public static void main(String[] args) {
        Algebra al[] = new Algebra[] {
                                    (a, b) -> a+b,
                                    (a, b) -> a-b,
                                    (a, b) -> a*b,
                                    (a, b) -> a/b
                                  };
        System.out.println("10 + 20 = " + al[0].operate(10, 20));
        System.out.println("10 - 20 = " + al[1].operate(10, 20));
        System.out.println("10 * 20 = " + al[2].operate(10, 20));
        System.out.println("10 / 20 = " + al[3].operate(10, 20));
    }
}
```

http://javaattop.blogspot.ca/2013/09/lambda-expressions-in-java.html

QuizQuestion1.java | MyInterface.java

```java
import java.io.IOException;
import java.io.OutputStream;

public interface MyInterface {

    void printIt(String text);

    default public void printUtf8To(String text, OutputStream outputStream){
        try {
            outputStream.write(text.getBytes("UTF-8"));
        } catch (IOException e) {
            throw new RuntimeException("Error writing String as UTF-8 to OutputStream", e);
        }
    }

    static void printItToSystemOut(String text){
        System.out.println(text);
    }
}
```

QuizQuestion1.java | MyInterface.java

```java
public class QuizQuestion1 {
    public static void main(String[] args) {
        MyInterface myInterface = (String text) -> {
            System.out.print(text);
        };
        myInterface.printIt("Hello World!");
    }
}
```

Is the following code correct?
A:Yes B:No

# Streams

Not I/O streams

# Streams

- represents sequence of elements
- supports operations on these elements

```
List<String> myList = Arrays.asList("a1", "a2", "b1", "c2", "c1");

myList
     .stream()
     .filter(s -> s.startsWith("c"))
     .map(String::toUpperCase)
     .sorted()
     .forEach(System.out::println);

// C1
// C2
```

# Streams

operations are either

- intermediate: return a stream
  - chained in *operation pipeline*
- terminal: void or return non-stream

```
List<String> myList = Arrays.asList("a1", "a2", "b1", "c2", "c1");

myList
      .stream()
      .filter(s -> s.startsWith("c"))
      .map(String::toUpperCase)
      .sorted()
      .forEach(System.out::println);
```



http://winterbe.com/posts/2014/07/31/java8-stream-tutorial-examples/

# Streams

most stream operations accept lambda expression as parameter that needs to be

- non-interfering
  - does not modify underlying data source
- stateless
  - execution is deterministic: does not depend on variables/states from the outer scope which might change during execution



http://winterbe.com/posts/2014/07/31/java8-stream-tutorial-examples/

# Streams

- stream()
  - Arrays.asList("a1", "a2", "a3").stream().findFirst().
  - Stream.of("a1", "a2", "a3").findFirst().
- parallelStream()
  - operate on multiple threads



http://winterbe.com/posts/2014/07/31/java8-stream-tutorial-examples/

# Sequential Streams

- IntStream, LongStream, DoubleStream
  - `IntStream.range(1, 4)`
    `.forEach(System.out::println);`
  - **use specialized lambda expression**
    - **IntFunction instead of Function**
    - **IntPredicate instead of Predicate**
  - **support**
    - **sum()**
    - **average()**



http://winterbe.com/posts/2014/07/31/java8-stream-tutorial-examples/

# Sequential Streams

- transform
  - Stream.of("a1", "a2", "a3")
    .map(s -> s.substring(1))
    .mapToInt(Integer::parseInt)
    .max()
    .ifPresent(System.out::println);  // 3
  - IntStream.range(1, 4)
    .mapToObj(i -> "a" + i)
    .forEach(System.out::println);// a1, a2, a3



http://winterbe.com/posts/2014/07/31/java8-stream-tutorial-examples/

# Processing order

- laziness
  - Stream.of("d2", "a2", "b1", "b3", "c")
    .filter(s -> {
    System.out.println("filter: " + s);
    return true;
    });//nothing printed (no terminal operation)
  - Stream.of("d2", "a2", "b1", "b3", "c")
    .filter(s -> {
    System.out.println("filter: " + s);
    return true;
    })
    .forEach(s -> System.out.println("forEach: " + s));//print as expected



http://winterbe.com/posts/2014/07/31/java8-stream-tutorial-examples/

# Processing order

- laziness
  - Stream.of("d2", "a2", "b1", "b3", "c")
    .filter(s -> {
    System.out.println("filter: " + s);
    return true;
    })
    .forEach(s -> System.out.println("forEach: " + s));
- operations executed vertically

```
filter:     d2
forEach: d2
filter:     a2
forEach: a2
filter:     b1
forEach: b1
filter:     b3
forEach: b3
filter:     c
forEach: c
```



http://winterbe.com/posts/2014/07/31/java8-stream-tutorial-examples/

# Processing order

```
map:      d2
anyMatch: D2
map:      a2
anyMatch: A2
```

- map called only twice
  - Stream.of("d2", "a2", "b1", "b3", "c")
    .map(s -> {
        System.out.println("map: " + s);
        return s.toUpperCase();
    })
    .anyMatch(s -> {
        System.out.println("anyMatch: " + s);
        return s.startsWith("A");
    });
- order matters when you wish to minimize number of operations
  - filter before sort



http://winterbe.com/posts/2014/07/31/java8-stream-tutorial-examples/

# Last points

- streams cannot be reused after terminal operation
  - Stream<String> stream =
    Stream.of("d2", "a2", "b1", "b3", "c")
    .filter(s -> s.startsWith("a"));

    stream.anyMatch(s -> true);  // ok
    stream.noneMatch(s -> true);  // exception



http://winterbe.com/posts/2014/07/31/java8-stream-tutorial-examples/

# Advanced

# Advanced uses

## Collect

- ```
  List<Person> filtered =
      persons
          .stream()
          .filter(p -> p.name.startsWith("P"))
          .collect(Collectors.toList());

  System.out.println(filtered);    // [Peter, Pamela]
  ```

- ```
  Map<Integer, List<Person>> personsByAge = persons
      .stream()
      .collect(Collectors.groupingBy(p -> p.age));

  personsByAge
      .forEach((age, p) -> System.out.format("age %s: %s\n", age, p));

  // age 18: [Max]
  // age 23: [Peter, Pamela]
  // age 12: [David]
  ```



http://winterbe.com/posts/2014/07/31/java8-stream-tutorial-examples/

# Advanced uses

- Map
  - transforms one object into exactly one object
- FlatMap
  - transforms each element of a stream into a stream of other objects
  - 0, 1, 2, … elements for each element



http://winterbe.com/posts/2014/07/31/java8-stream-tutorial-examples/

# Advanced uses

- Reduce
  - combine all elements into a single result
  - find oldest person

```
persons
    .stream()
    .reduce((p1, p2) -> p1.age > p2.age ? p1 : p2)
    .ifPresent(System.out::println);    // Pamela
```



http://winterbe.com/posts/2014/07/31/java8-stream-tutorial-examples/

# Advanced uses

```
filter:     b1 [main]
filter:     a2 [ForkJoinPool.commonPool-worker-1]
map:        a2 [ForkJoinPool.commonPool-worker-1]
filter:     c2 [ForkJoinPool.commonPool-worker-3]
map:        c2 [ForkJoinPool.commonPool-worker-3]
filter:     c1 [ForkJoinPool.commonPool-worker-2]
map:        c1 [ForkJoinPool.commonPool-worker-2]
forEach:    C2 [ForkJoinPool.commonPool-worker-3]
forEach:    A2 [ForkJoinPool.commonPool-worker-1]
map:        b1 [main]
forEach:    B1 [main]
filter:     a1 [ForkJoinPool.commonPool-worker-3]
map:        a1 [ForkJoinPool.commonPool-worker-3]
forEach:    A1 [ForkJoinPool.commonPool-worker-3]
forEach:    C1 [ForkJoinPool.commonPool-worker-2]
```

- Debugging parallel streams
  - 
```
Arrays.asList("a1", "a2", "b1", "c2", "c1")
    .parallelStream()
    .filter(s -> {
        System.out.format("filter: %s [%s]\n", s, Thread.currentThread().getName());
        return true;
    })
    .map(s -> {
        System.out.format("map: %s [%s]\n", s, Thread.currentThread().getName());
        return s.toUpperCase();
    })
    .forEach(s -> System.out.format("forEach: %s [%s]\n",
        s, Thread.currentThread().getName()));
```



http://winterbe.com/posts/2014/07/31/java8-stream-tutorial-examples/

# Reference

- Java 8 Streams: plenty of tutorial/reference material online, for example
  - https://winterbe.com/posts/2014/07/31/java8-stream-tutorial-examples/
  - https://stackify.com/streams-guide-java-8/

# Survey


InteDashboard™
Empowering Team-Based Learning With Technology

# Summary

- Complexity. You should be able to
  - sort functions by asymptotic worst-case complexity
  - compute the space and time complexity of a simple code
- Stream. You should be able to
  - understand code written using streams
  - write simple code using streams
  - explain the reason to use streams:
    - code readability
    - parallelism

# Questions?

# Readings for next time

- See Readings page on Canvas

# THE END