Victoria University
of Wellington, New Zealand
Te Whare Wananga o te
Upoko o te Ika a Maui
Aotearoa

SWEN221:
Software Development

22: Optional and Streams

David J. Pearce & Marco Servetto
Engineering and Computer Science, Victoria University

Optional, an alternative to null

- Null pointers cause an overwhelming amount of errors.

- They break the intuitive promise the type system is giving us:

```
public static void foo(Person p){
  //Persons have name and age
  //let use this fact to do something
  System.out.println(p.name);//Nope!
}
```

Optional, an alternative to null

```
public static void foo(Person p){
  //Persons have name and age
  //let use this fact to do something
  System.out.println(p.name);}//Nope!
```

- use Optional<T> and be more explicit!

Optional, an alternative to null

- use Optional<T> and be more explicit!
  - isPresent + get if you are traditional

```
public static void foo(Optional<Person> p){
  //Persons have name and age,
  if(p.isPresent()){//but I'm not sure if I have a person
    System.out.println(p.get().name);//sure now!
  }}
```

  - ifPresent if you are a lambda fan!

```
public static void foo(Optional<Person> p){
  p.ifPresent(sureP->System.out.println(sureP.name));
}
```

Optional: create and manipulate

```
Optional<Person> marco=Optional.of(new Person("Marco",34));
```

- Optional is a proxy.
- Shows the programmer intention in the types.
- If used consistently instead of nulls
  ->code more readable and predictable.
- No agreement on when
  is good to use null and when is good to use Optional

Optional: create and manipulate

```
Optional<Person> marco=Optional.of(new Person("Marco",34));
Optional<Person> nope=Optional.empty();//no object created
```

- Optional is a proxy.
- Shows the programmer intention in the types.
- If used consistently instead of nulls
  ->code more readable and predictable.
- No agreement on when
  is good to use null and when is good to use Optional

Optional: create and manipulate

```
Optional<Person> marco=Optional.of(new Person("Marco",34));
Optional<Person> nope=Optional.empty();//no object created

if(marco.isPresent()){... marco.get();... };
```

- Optional is a proxy.
- Shows the programmer intention in the types.
- If used consistently instead of nulls
  ->code more readable and predictable.
- No agreement on when
  is good to use null and when is good to use Optional

Optional: create and manipulate

```
Optional<Person> marco=Optional.of(new Person("Marco",34));
Optional<Person> nope=Optional.empty();//no object created

if(marco.isPresent()){... marco.get();... };

marco.ifPresent(m->...);
```

- Optional is a proxy.
- Shows the programmer intention in the types.
- If used consistently instead of nulls
  ->code more readable and predictable.
- No agreement on when
  is good to use null and when is good to use Optional

Optional: create and manipulate

```
Optional<Person> marco=Optional.of(new Person("Marco",34));
Optional<Person> nope=Optional.empty();//no object created

if(marco.isPresent()){... marco.get();... };

marco.ifPresent(m->...);

nope.get();//Dynamic error
```

- Optional is a proxy.
- Shows the programmer intention in the types.
- If used consistently instead of nulls
  ->code more readable and predictable.
- No agreement on when
  is good to use null and when is good to use Optional

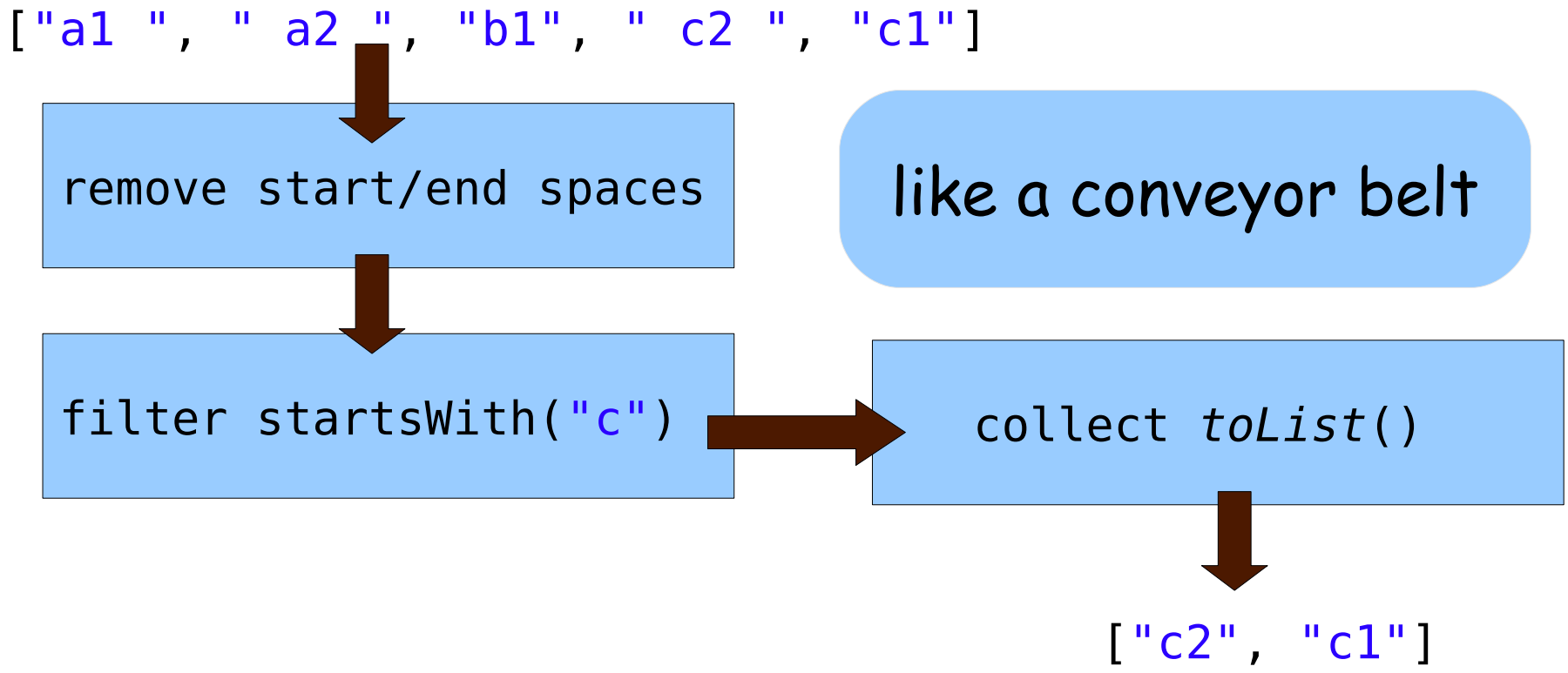
Java: Streams and collections

(Not correlated with InputStreams etc..)

Java: Streams and collections

(Not correlated with InputStreams etc..)

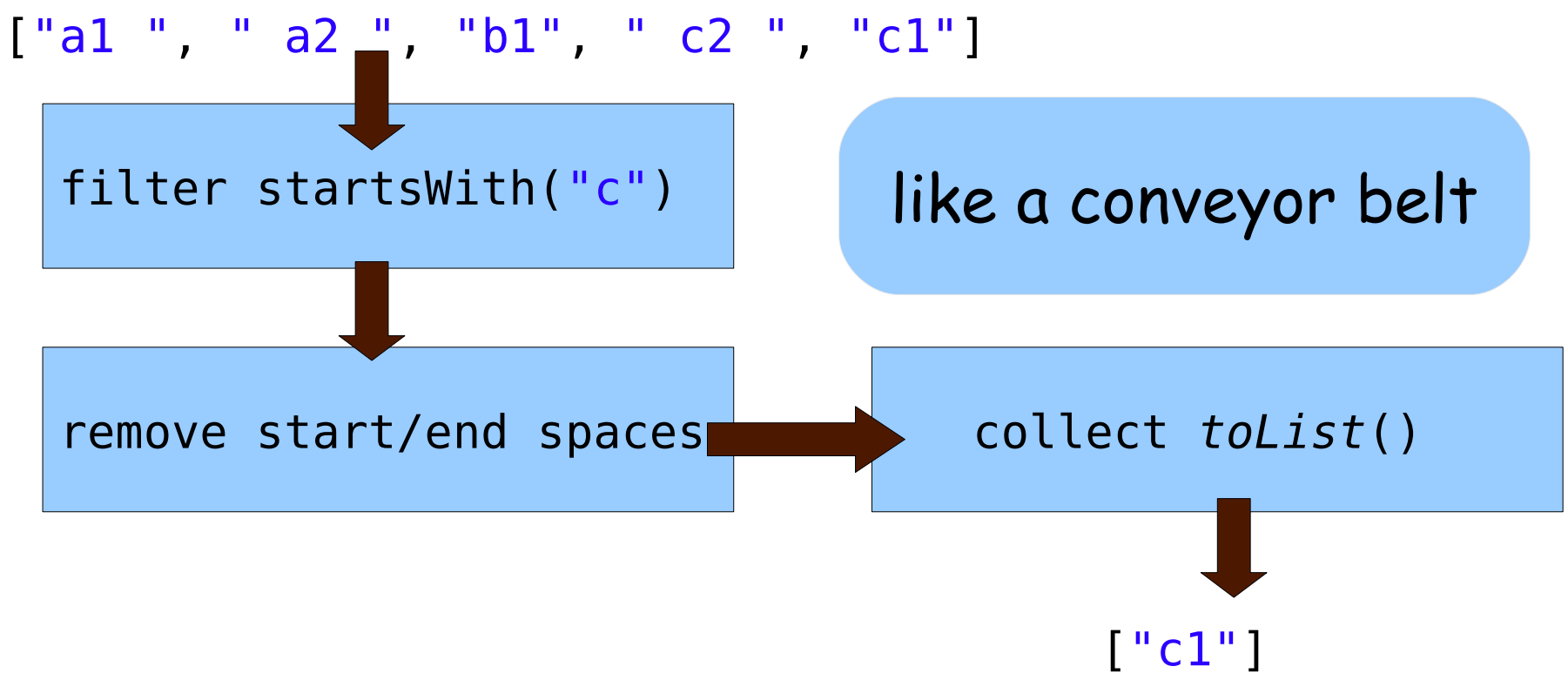
- Stream: Rich library defining sort of a sub-language to query and process collections.



Java: Streams and collections

(Not correlated with InputStreams etc..)

- Order of operations is important!



Java: Streams and collections

```
List<String> myList=//list from strings
  Arrays.asList("a1 ", " a2 ", "b1", " c2 ", "c1");
myList=myList.stream()//our entry point
 .filter(s -> s.startsWith("c"))//select only some stuff
 .map(s->s.trim())//mapping
 .collect(Collectors.toList());//myList == [c1]
```

Java: Streams and collections

```
List<String> myList=//list from strings
  Arrays.asList("a1 ", " a2 ", "b1", " c2 ", "c1");
myList=myList.stream()//our entry point
 .filter(s -> s.startsWith("c"))//select only some stuff
 .map(s->s.trim())//mapping
 .collect(Collectors.toList());//myList == [c1]
```

- .stream is our **entry point** for this rabbit-hole

Java: Streams and collections

```
List<String> myList=//list from strings
  Arrays.asList("a1 ", " a2 ", "b1", " c2 ", "c1");
myList=myList.stream()//our entry point
 .filter(s -> s.startsWith("c"))//select only some stuff
 .map(s->s.trim())//mapping
 .collect(Collectors.toList());//myList == [c1]
```

- .stream is our **entry point** for this rabbit-hole
- .filter and .map are **intermediate operations**: streams in, streams out!

Java: Streams and collections

```
List<String> myList=//list from strings
  Arrays.asList("a1 ", " a2 ", "b1", " c2 ", "c1");
myList=myList.stream()//our entry point
 .filter(s -> s.startsWith("c"))//select only some stuff
 .map(s->s.trim())//mapping
 .collect(Collectors.toList());//myList == [c1]
```

- .stream is our **entry point** for this rabbit-hole
- .filter and .map are **intermediate operations**: streams in, streams out!
- .collect is a **terminal operation**; in this case produces a list.

Map and filter

```
List<Person> persons=Arrays.asList(...);
List<Student> readyForSWEN222=persons.stream()
   .filter(p-> p instanceof Student)
   .map(p->(Student)p)
   .filter(s->s.marks.containsKey("SWEN221"))
   .collect(Collectors.toList());
```

Map and filter, equivalent for loop

```
List<Person> persons=Arrays.asList(...);          //a
List<Student> readyForSWEN222=new ArrayList<>();//b
for(Person p: persons){                           //c
  if(!(p instanceof Student)){continue;}          //d
  Student s=(Student)p;                           //e
  if(!s.marks.containsKey("SWEN221")){continue;}//f
  readyForSWEN222.add(s);                         //g
}

List<Person> persons=Arrays.asList(...);          //a
List<Student> readyForSWEN222=persons.stream()  //b+c
  .filter(p-> p instanceof Student)               //d
  .map(p->(Student)p)                             //e
  .filter(s->s.marks.containsKey("SWEN221"))    //f
  .collect(Collectors.toList());                  //g
```

Map, filter, reduce

```
List<Person> persons=Arrays.asList(...);
List<Student> readyForSWEN222=persons.stream()
   .filter(p-> p instanceof Student)
   .map(p->(Student)p)
   .filter(s->s.marks.containsKey("SWEN221"))
   .collect(Collectors.toList());
```

Here we collect in a list

```
List<Person> persons=Arrays.asList(...);
Optional<Student> younger=persons.stream()
   .filter(p-> p instanceof Student)
   .map(p->(Student)p)
   .filter(s->s.marks.containsKey("SWEN221"))
   .reduce((s1,s2)->{
      if(s1.age<s2.age){return s1;} return s2;});
```

Here we take the younger one

Reduce

```
//Optional if you just accumulate
Optional<Student> younger=...
  .reduce((s1,s2)->{...});
```

Reduce

```
//Optional if you just accumulate
Optional<Student> younger=...
  .reduce((s1,s2)->{...});

//Sure if you provide a starting point
Student alice=...
Person atLeastAlice=...
  .reduce(alice,(s1,s2)->{...});
```

Stream exercise: Genetic algorithm

- Compute fitness for candidate solutions and select the best ones.

- For example, you may have a list of paper aeroplanes, and you want to record the best ones!

Stream exercise: Genetic algorithm

You can easily do that using streams:

```
List<Aeroplane> attempts=...

attempts.stream()//first, cache the fitness
  .forEach(a->a.computeAverageFlightTime());

List<Aeroplane> best20=attempts.stream()
  .sorted((a1,a2)->a1.getFlightTime()-a2.getFlightTime())
  .limit(20)//take the first 20
  .collect(Collectors.toList());
```

`computeAverageFlightTime` can be slow,

- you could need to do it for all your attempts!

- Modern hardware have multiple processors.

Stream exercise: Genetic algorithm

In real life, you could try to fly aeroplanes using multiple rooms at the same time,

- To test 100 aeroplanes, for 1 minutes each, it would take 100 minutes.
- Having 10 friends and 10 testing chambers, you can run parallel tests, to finish in 10 minutes

- The same idea applies with multiple processors:

you may run 8 `computeAverageFlightTime`

at the same time using 8 cores.

Parallel Stream

- Streams can use multiple processors.
  No need to use threading explicitly.

As before

```
attempts.stream()//sequential
  .forEach(a->c.computeAverageFlightTime());
List<Aeroplane> best20=attempts.stream()
  .sorted((c1,c2)->c1.getFlightTime()-c2.getFlightTime())
  .limit(20)//take the first 20
  .collect(Collectors.toList());
```

Parallel Stream

- Streams can use multiple processors.
  No need to use threading explicitly.

As before

```
attempts.stream()//sequential
  .forEach(a->c.computeAverageFlightTime());
List<Aeroplane> best20=attempts.stream()
  .sorted((c1,c2)->c1.getFlightTime()-c2.getFlightTime())
  .limit(20)//take the first 20
  .collect(Collectors.toList());
```

In parallel

```
attempts.parallelStream()//parallel
  .forEach(a->c.computeAverageFlightTime());
List<Aeroplane> best20=attempts.parallelStream()
  .sorted((c1,c2)->c1.getFlightTime()-c2.getFlightTime())
  .limit(20)//take the first 20
  .collect(Collectors.toList());
```

Parallel Stream

- Just like that?
  Just call .parallelStream() instead of stream()

  No need to use threading explicitly?
- Yep!
- And by the way, in any application, if you have to use treads explicitly, you are doing something that is at research level, you are expanding what humans believe possible.
- So either you are doing it **wrong**, or be sure to be paid accordingly.

Parallel Stream and reduce

- A third form of reduce is useful when using parallel streams. Takes 3 parameters
  - an initial value (int in this case).
  - a way to compose a (int) value with a stream element (Person) to produce a new (int) value.

Parallel Stream and reduce

- A third form of reduce is useful when using parallel streams. Takes 3 parameters
  - an initial value (int in this case).
  - a way to compose a (int) value with a stream element (Person) to produce a new (int) value.
  - a way to compose two (int) values into a new one.

Parallel Stream and reduce

- A third form of reduce is useful when using parallel streams. Takes 3 parameters
  - an initial value (int in this case).
  - a way to compose a (int) value with a stream element (Person) to produce a new (int) value.
  - a way to compose two (int) values into a new one.
- parallelStream divides your work in jobs and put them together.

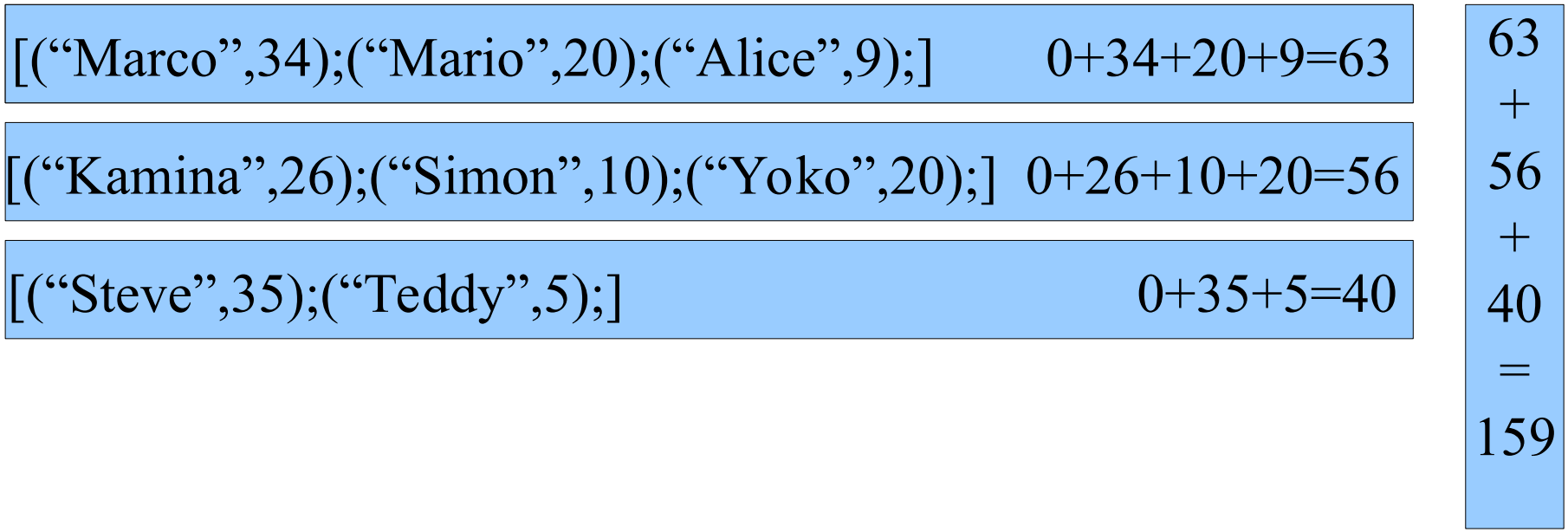
Parallel Stream and reduce

- A third form of reduce is useful when using parallel streams. Takes 3 parameters
  - an initial value (int in this case).
  - a way to compose a (int) value with a stream element (Person) to produce a new (int) value.
  - a way to compose two (int) values into a new one.
- parallelStream divides your work in jobs and put them together.

```
int ageSum = persons.parallelStream()
  .reduce(0,
   (sum, p) -> sum + p.age,
   (sum1, sum2) -> sum1 + sum2);
```

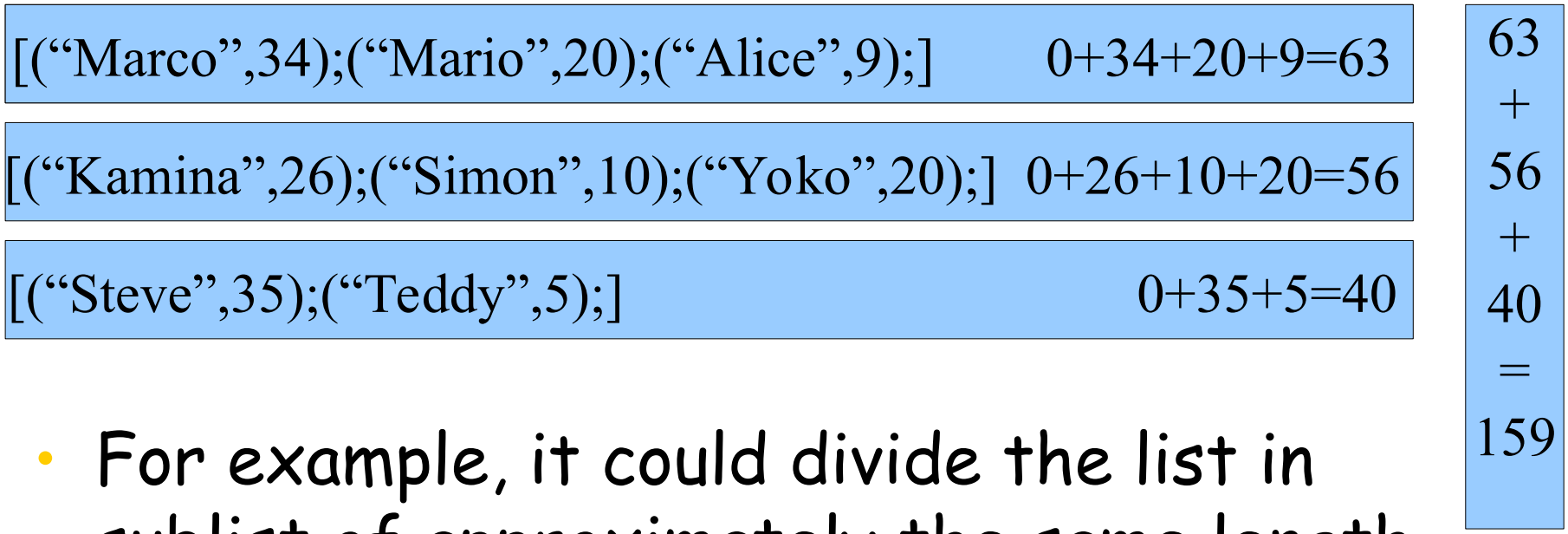
Parallel Stream and reduce

- parallelStream figures out how to divide your work in jobs and how to put them together.



Parallel Stream and reduce

- parallelStream figures out how to divide your work in jobs and how to put them together.



- For example, it could divide the list in sublist of approximately the same length, compute the sum of the sublists and then compute the grand total.

More parallel streams

- More Java parallel programming and more parallel streams in NWEN303