Eclipse Collections and Java 8 Lambdas

Mini Code Kata
Basic Ideas, Basic Implementations
March 2016

Copyright © 2016 Goldman Sachs. All rights reserved.
What is Eclipse Collections?

• A supplement or replacement for the Java Collections Framework (JCF).
• Developed originally at Goldman Sachs.
• "Inspired by" Smalltalk Collection protocol.

What is a Code Kata?

• A programming exercise which helps hone your skills through practice.
• This one is set up as a series of unit tests which fail.
• Your task is to make them pass, using Eclipse Collections –

  I hear and I forget.

  I see and I remember.

  I do and I understand.

  - Confucius

Eclipse Collections Mini Code Kata

• New concepts are introduced in the slides
• Coding exercises are at the end of each section.
Iteration Patterns
• What is an iteration pattern?
• **Sort** is one example.
• We want to sort a list of people by last name, first name.
• Which method in the JCF can we use?

```
Person john = new Person("John", "Smith");
Person jane = new Person("Jane", "Smith");
Person z = new Person("Z.", "Jones");

List<Person> people = new ArrayList<Person>();
people.add(john);
people.add(jane);
people.add(z);
```
public static void java.util.Collections.sort(
        List<T> list, Comparator<? super T> c)

Sorts the specified list according to the order induced by the specified comparator. All elements in the list must be mutually comparable.

Anonymous inner class syntax

Collections.sort(people, new Comparator<Person>() {
    public int compare(Person o1, Person o2) {
        int lastName = o1.getLastName().compareTo(o2.getLastName());
        if (lastName != 0) {
            return lastName;
        } else {
            return o1.getFirstName().compareTo(o2.getFirstName());
        }
    }
});
public static void java.util.Collections.sort(
    List<T> list, Comparator<? super T> c)

Sorts the specified list according to the order induced by the specified comparator. All elements in the list must be *mutually comparable*.

Does anything bother you about Collections.sort()?
public static void java.util.Collections.sort(
    List<T> list, Comparator<? super T> c)

Sorts the specified list according to the order induced by the specified comparator. All elements in the list must be mutually comparable.

Does anything bother you about Collections.sort()?

JCF problems

- Why isn’t sort() a method on every List?
  
  Collections.sort(list, comparator);
  
  vs.
  
  list.sort(comparator);
public static void java.util.Collections.sort(
    List<T> list, Comparator<? super T> c)

Sorts the specified list according to the order induced by the specified comparator. All elements in the list must be mutually comparable.

Does anything bother you about Collections.sort()?  

JCF problems

- Where are all the iteration patterns?
- java.util.Collections provides methods for sort(), min(), max() and just a few others.
- The most common iteration patterns are missing:
  - Collect a list of each person’s first name.
  - Select only those people whose last name is “Smith”.


Iteration Patterns

• We want the methods `sort()`, `min()`, `max()`, `collect()`, `select()`, etc. on every collection.
• How can we accomplish this in code?
• Create interfaces that extend the JCF interfaces.

Eclipse Collections Interfaces

```java
public interface MutableList<T> extends List<T>
{
    MutableList<T> sortThis(Comparator<? super T> comparator);
    <V> MutableList<V> collect(Function<? super T, ? extends V> function);
    MutableList<T> select(Predicate<? super T> predicate);
    ...
}
```
• **Collect** pattern (aka *map* or *transform*).
• Returns a new collection where each element has been transformed
  • e.g. collect each pet’s name.

• **Function** is the type that takes an object and returns an object of a different type
  • aka *Transformer*

### JCF Example

```java
List<Pet> pets = ...;
List<String> petNames = new ArrayList<String>();
for (Pet pet : pets)
{
    petNames.add(pet.getName());
}
```
- **Function** is the type that takes an object and returns another type.

**Eclipse Collections Example (Anonymous Inner Class Syntax)**

```java
MutableList<Pet> pets = ...
MutableList<String> petNames = pets.collect(
    new Function<Pet, String>()
    {
        public String valueOf(Pet pet)
        {
            return pet.getName();
        }
    });
```
• **Function** is the type that takes an object and returns another type.

**Eclipse Collections Example (Java 8)**

```java
MutableList<Pet> pets = ...
MutableList<String> petNames =
    pets.collect(pet -> pet.getName());
```

```java
MutableList<String> petNames =
    pets.collect(Pet::getName);
```
• Select pattern (aka filter).
• Returns the elements of a collection that satisfy some condition
  • e.g. select only those people who have a pet.
• Predicate is the type that takes an object and returns a boolean.

```java
MutableList<Person> people = ...
MutableList<Person> petPeople = people.select(
    new Predicate<Person>()
    {
        public boolean accept(Person person)
        {
            return person.isPetPerson();
        }
    });
```
**Iteration Pattern**

- *Select* pattern (aka *filter*).
- Returns the elements of a collection that satisfy some condition
  - e.g. select only those people who have a pet.
- *Predicate* is the type that takes an object and returns a boolean.

**Eclipse Collections Example (Java 8)**

```java
MutableList<Person> people = ...;
MutableList<Person> petPeople = people.select(person -> person.isPetPerson());
```
### Examples

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Collect</strong></td>
<td>Returns a new collection where each element has been transformed.</td>
</tr>
<tr>
<td><strong>Select</strong></td>
<td>Returns the elements of a collection that satisfy some condition.</td>
</tr>
</tbody>
</table>

### Code Blocks

- `sortThis()` takes a `Comparator`, which is a strategy interface.
- Similarly, `collect()` and `select()` take “code block” parameters.
- `collect()` takes a `Function`.
- `select()` takes a `Predicate`.
- Don’t get hung up on these names because the IDE will remind you.
Kata Introduction
Domain

- Eclipse Collections Mini Kata uses Person, Pet, PetType
- Data is available to you through `this.people` which has some person and pet objects set for you
Kata Exercise 1
Exercise 1

- Fix `Exercise1Test`; they have failures.
- Figure out how to get the tests to pass using what you have seen so far.
- Should take about 15 minutes.
Verify

• Eclipse Collections distribution includes `eclipse-collections-testutils.jar`.
  • Includes helpful utility for writing unit tests.
  • Collection-specific.
  • Implemented as an extension of JUnit.
  • Better error messages
• Most important class is called `Verify`.

Code Example

Example from the previous solution

`Verify.assertEquals(2, peopleWithCats);`

Instead of

`Assert.assertEquals(2, peopleWithCats.size());`
• flatCollect() is a special case of collect().
• With collect(), when the Function returns a collection, the result is a collection of collections.

Code Example

```java
MutableList<Person> people = ...;
Function<Person, Iterable<PetType>> function =
    person -> person.getPetTypes();

MutableList<MutableList<PetType>> pets =
    people.collect(function);
```
**flatCollect()**

- `flatCollect()` outputs a single “flattened” collection instead of a collection of collections.
- The signature of `flatCollect()` is similar to `collect()`, except that the `Function` parameter *must* map to an `Iterable` type.

```java
flatCollect(Function<? super T, ? extends Iterable<V>> function);
```

**Code Example**

```java
MutableList<Person> people = ...;

MutableList<PetType> pets =
    people.flatCollect(person -> person.getPetTypes());
```
MORE ITERATION PATTERNS
<table>
<thead>
<tr>
<th><strong>Other patterns that use Predicate</strong></th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Select</strong></td>
</tr>
<tr>
<td><strong>Reject</strong></td>
</tr>
<tr>
<td><strong>Count</strong></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th><strong>Short-circuit patterns that use Predicate</strong></th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Detect</strong></td>
</tr>
<tr>
<td><strong>Any Satisfy</strong></td>
</tr>
<tr>
<td><strong>All Satisfy</strong></td>
</tr>
</tbody>
</table>
Lists

- **MutableList** extends **List**.
- **FastList** is a drop-in replacement for **ArrayList**.
Inheritance Hierarchy

Sets

- **MutableSet** extends **Set**.
- **UnifiedSet** is a drop in replacement for **HashSet**.
“With” Methods
collectWith(), selectWith(), and rejectWith()

- collectWith(), selectWith(), and rejectWith() are alternate forms of collect(), select(), and reject().
- Original forms all take a single parameter, a code block which takes a single parameter.
- What if we want to find groups of people at different ages?

Code Example

```java
MutableList<Person> voters = people.select(
    person -> person.getAge() > 18;
);
```
collectWith(), selectWith(), and rejectWith()

- collectWith(), selectWith(), and rejectWith() are alternate forms of collect(), select(), and reject().
- ...with() forms take two parameters:
  - a code block which takes two parameters,
  - an object that gets passed as the second argument to the code block.
- Store the two-argument block in a constant to avoid object creation.

Code Example

```java
Predicate2<Person, Integer> age =
   (person, age) -> person.getAge() > age;

MutableList<Person> drivers = people.selectWith(age, 17);
MutableList<Person> voters = people.selectWith(age, 18);
MutableList<Person> drinkers = people.selectWith(age, 21);
MutableList<Person> sunsetRobotSquad = people.selectWith(age, 160);
```
For almost every common iteration pattern, there is an equivalent “With” method.
- `selectWith()`
- `detectWith()`
- `rejectWith()`
- `anySatisfyWith()`
... etc

How they can be used

These are useful when you want to use method references rather than lambdas.

```java
// does not compile
MutableList<Person> peopleWithCatsLambda = this.people.select(Person::hasPet(PetType.CAT));
```
For almost every common iteration pattern, there is an equivalent “With” method.
- `selectWith()`
- `detectWith()`
- `rejectWith()`
- `anySatisfyWith()`
- `... etc`

**How they can be used**

These are useful when you want to use method references rather than lambdas.

```java
MutableList<Person> peopleWithCatMethodReference = this.people.selectWith(Person::hasPet, PetType.CAT);
```
Exercise 2

• Fix `Exercise2Test`.
• Use the other iteration patterns that take a `Predicate`.
• Use the `flatMap` pattern
• Should take about 15 minutes.
New Type

- Useful when you would otherwise use `Map<K, Integer>`
  - For example, find the number of each pet type.

Code Example

```java
MutableList<Person> people = ...;
MutableList<PetType> pets =
    people.flatCollect(Person::getPetTypes);
MutableMap<PetType, Integer> petTypeCounts =
    UnifiedMap.newMap();
...

int cats = petTypeCounts.get(PetType.CAT);
```
New Type

- Useful when you would otherwise use `Map<K, Integer>`
  - For example, find the number of each pet type.
  - Lots of boilerplate code to deal with uninitialized counts.

Map Example

```java
MutableMap<PetType, Integer> petTypeCounts = UnifiedMap.newMap();
for (PetType petType : pets) {
    Integer count = petTypeCounts.get(petType);
    if (count == null) {
        count = 0;
    }
    petTypeCounts.put(petType, count + 1);
}
```
New Type

• Implemented as a map of key to count.
• Like a List, but unordered.
• Like a Set, but allows duplicates.

Bag Example

```java
MutableBag<PetType> petTypeCounts = pets.toBag();

int cat = petTypeCounts.occurrencesOf(PetType.CAT);
```
## Bag API

<table>
<thead>
<tr>
<th>Methods</th>
<th>Inherited from</th>
</tr>
</thead>
<tbody>
<tr>
<td>select(), collect(), etc.</td>
<td>RichIterable</td>
</tr>
<tr>
<td>add(), remove(), iterator(), etc.</td>
<td>MutableCollection (java.util.Collection)</td>
</tr>
<tr>
<td>occurrencesOf(), forEachWithOccurrences(), toMapOfItemToCount()</td>
<td>Bag</td>
</tr>
<tr>
<td>addOccurrences(), removeOccurrences()</td>
<td>MutableBag</td>
</tr>
</tbody>
</table>

### Code Example

```java
MutableBag<String> bag = HashBag.newBagWith("one", "two", "two", "three", "three", "three");

Assert.assertEquals(3, bag.occurrencesOf("three"));

bag.add("one");
Assert.assertEquals(2, bag.occurrencesOf("one"));

bag.addOccurrences("one", 4);
Assert.assertEquals(6, bag.occurrencesOf("one"));
```
**New Type**

- **Multimap** is similar to **Map**, but associates a key to *multiple values*.
- Useful when you would otherwise use **Map<K, Collection<V>>**
  - For example, find *which* people have the same last name.

**Code Example**

```java
MutableList<Person> people = ...;
MutableMap<String, MutableList<Person>>
    lastNamesToPeople = UnifiedMap.newMap();

...

MutableList<Person> smiths =
    lastNamesToPeople.get("Smith");
```
**New Type**

- **Multimap** is similar to **Map**, but associates a key to *multiple values*.
- Useful when you would otherwise use **Map<K, Collection<V>>**
  - For example, find *which* people have the same last name.
  - Lots of boilerplate code to deal with uninitialized backing collections.

**Map Example**

```java
MutableMap<String, MutableList<Person>> lastNamesToPeople = UnifiedMap.newMap();
for (Person person : people) {
    String lastName = person.getLastName();
    MutableList<Person> peopleWithLastName = lastNamesToPeople.get(lastName);
    if (peopleWithLastName == null) {
        peopleWithLastName = FastList.newList();
        lastNamesToPeople.put(lastName, peopleWithLastName);
    }
    peopleWithLastName.add(person);
}
```
Before

```java
MutableMap<String, MutableList<Person>> lastNamesToPeople = UnifiedMap.newMap();
for (Person person : people) {
    String lastName = person.getLastName();
    MutableList<Person> peopleWithLastName = lastNamesToPeople.get(lastName);
    if (peopleWithLastName == null) {
        peopleWithLastName = FastList.newList();
        lastNamesToPeople.put(lastName, peopleWithLastName);
    }
    peopleWithLastName.add(person);
}
```

After

```java
MutableListMultimap<String, Person> lastNamesToPeople = people.groupBy(Person.TO_LAST_NAME);

MutableList<Person> smiths = lastNamesToPeople.get("Smith");
```
What happens if you add the same key and value twice?

```java
Mutable Multimap<String, Person> multimap = ...;

multimap.put("Smith", person);
multimap.put("Smith", person);
RichIterable<Person> smiths = multimap.get("Smith");

Verify.assertIterableSize(? , smiths);
```
Multimap

- What happens if you add the same key and value twice?
- Depends on the type of the backing collection.

Code Example

```java
MutableListMultimap<String, Person> multimap = FastListMultimap.newMultimap();
multimap.put("Smith", person);
multimap.put("Smith", person);
MutableList<Person> smiths = multimap.get("Smith");
Verify.assertIterableSize(2, smiths);
```
**Multimap**

- What happens if you add the same key and value twice?
- Depends on the type of the backing collection.

**Code Example**

```java
MutableSetMultimap<String, Person> multimap =
    UnifiedSetMultimap.newMultimap();
multimap.put("Smith", person);
multimap.put("Smith", person);
MutableSet<Person> smiths = multimap.get("Smith");
Verify(assertIterableSize(1, smiths));
```
groupByEach()

- `groupByEach()` is a special case of `groupBy()`.
- Analogous to the difference between `collect()` and `flatCollect()`.
- Appropriate when the `Function` returns a collection.
- The return type is the same as `groupBy()`.

Code Example

```java
MutableListMultimap<String, Person> lastNamesToPeople = people.groupBy(LAST_NAME_FUNCTION);

MutableListMultimap<PetType, Person> petsToPeople = people.groupByEach(PET_TYPE_FUNCTION);
```
TARGET COLLECTIONS
• Let's say we have 3 people: mrSmith, mrsSmith, mrJones.
• The first two share the same address.
• What will get printed by the following code?

```java
MutableSet<Person> people =
    UnifiedSet.newSetWith(mrSmith, mrsSmith, mrJones);

int numAddresses =
    people.collect(addressFunction).size();

System.out.println(numAddresses);
```
Covariant return types

- `select()`, `collect()`, etc. are defined with covariant return types:
  - `MutableCollection.collect()` returns a `MutableCollection`
  - `MutableList.collect()` returns a `MutableList`
  - `MutableSet.collect()` returns a `MutableSet`

- Alternate forms take target collections.

Example Code

```java
MutableSet<Person> people =
    UnifiedSet.newSetWith(mrSmith, mrsSmith, mrJones);

int numAddresses =
    people.collect(addressFunction).size();

System.out.println(numAddresses);
```
Covariant return types

- `select()`, `collect()`, etc. are defined with covariant return types:
  - `MutableCollection.collect()` returns a `MutableCollection`
  - `MutableList.collect()` returns a `MutableList`
  - `MutableSet.collect()` returns a `MutableSet`

- Alternate forms take target collections.

Example Code

```java
MutableSet<Person> people =
    UnifiedSet.newSetWith(mrSmith, mrsSmith, mrJones);

MutableList<Address> targetList =
    FastList.<Address>newList();

int numAddresses =
    people.collect(addressFunction, targetList).size();

System.out.println(numAddresses);
```
Exercise 3

- Fix Exercise3Test.
- Use Bag, Multimap.
- Should take about 10 minutes.
Each data structure has an equivalent primitive side.

Eclipse Collections supports all primitives – int, double, short, long, float, boolean, char, byte
Primitive API

There is also the same rich and fluent API that the non primitive side offers.

Examples

```java
MutableIntList intListOfAges =
    this.people.flatCollect(Person::getPets)
    .collectInt(Pet::getAge);

MutableIntList selection =
    intListOfAges.select(age -> age > 2);

MutableIntSet intSetOfAges = intListOfAges.toSet();
```
Kata Exercise 4
<table>
<thead>
<tr>
<th>Exercise 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>• Fix <em>Exercise4Test</em>.</td>
</tr>
<tr>
<td>• Refactor Java 8 streams to Eclipse Collections.</td>
</tr>
<tr>
<td>• Should take about 20 minutes.</td>
</tr>
</tbody>
</table>
Kata Exercise 5
Exercise 4

• Fix Exercise5Test.
• This is extra-credit
• Use target versions, methods on RichIterable
• Should take about 20 minutes.
Anonymous Inner Class
- An inner class with no name.
- Looks like a normal constructor call, but it works on abstract types (including interfaces).
- Has a body afterwards which makes the type concrete or overrides method definitions.
- Has a random-looking class name after compilation.
  - OuterClass$1 for the first anonymous inner class in OuterClass.

### Code Example

```java
Comparator<Person> comparator = new Comparator<Person>() {
    public int compare(Person o1, Person o2) {
        return o1.getLastName().compareTo(o2.getLastName());
    }
};
System.out.println(comparator.getClass());
```
Closure
Wikipedia definition:

“[A] closure is a first-class function with free variables that are bound in the lexical environment.

“Such a function is said to be ‘closed over’ its free variables.

“A closure is defined within the scope of its free variables, and the extent of those variables is at least as long as the lifetime of the closure itself.’’

• Java does not have closures.
• Java 8 has lambdas.
Java does **not** have closures.

In the code example, the `Predicate` is a class with a `String` field. It is **not** the same copy as the method parameter.

```java
public Customer getCustomerNamed(String name) {
    Predicate<Customer> customerNamed =
        Predicates.attributeEqual(Customer.TO_NAME, name);
    return this.customers.detect(customerNamed);
}
```
Java does **not** have closures.
In the code example, the `Predicate` is a class with a `String` field. It is **not** the same copy as the method parameter.
- Changing this copy of name has no effect on the result.
- Maybe obvious because of Java’s rules for parameter passing.

```java
public Customer getCustomerNamed(String name) {
    Predicate<Customer> customerNamed =
        Predicates.attributeEqual(Customer.TO_NAME, name);
    name = name + "!";
    return this_customers.detect(customerNamed);
}
```
Java does **not** have closures.

In the code example, the `Predicate` is a class with a `String` field.
It is **not** the same copy as the method parameter.

- If you use an anonymous inner class to implement the `Predicate`, you have to make name `final` in order to satisfy the compiler.

```java
public Customer getCustomerNamed(final String name) {
    Predicate<Customer> attributeEqual =
        new Predicate<Customer>() {
            public boolean accept(Customer customer) {
                return customer.getName().equals(name);
            }
        };
    return this.customers.detect(attributeEqual);
}
```
Closure Definition

Java does **not** have closures.

In the code example, the `Predicate` is a class with a `String` field. It is **not** the same copy as the method parameter.

- If you use an anonymous inner class to implement the `Predicate`, you have to make `name` final in order to satisfy the compiler.
- Refactor the anonymous inner class to a named inner class and you can see why.

Code Example

```java
private static final class CustomerNamed implements Predicate<Customer> {
    private final String name;
    private CustomerNamed(String name) {
        this.name = name;
    }
    public boolean accept(Customer customer) {
        return customer.getName().equals(this.name);
    }
}
```
Function transforms one type (T) to another (V):

```java
public interface Function<T, V>
    extends Serializable
{
    V valueOf(T object);
}
```

Used in:
- collect
- flatCollect
- groupBy
- minBy
- maxBy
- toSortedListBy
- sortThisBy
- toMap
**Predicate** takes an object of some type (T) and returns a `boolean`:

```java
public interface Predicate<T>
    extends Serializable
{
    boolean accept(final T each);
}
```

Used in:
- select
- reject
- detect
- count
- anySatisfy
- allSatisfy
Procedure takes an object of some type (T) and doesn’t return anything:

```java
public interfaceProcedure<T>
    extends Serializable
{
    void value(final T each);
}
```

Used in:
- forEach
<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>collect</td>
<td>Transforms elements using a <code>Function</code> into a new collection.</td>
</tr>
<tr>
<td>flatCollect</td>
<td>Transforms and flattens the elements using a <code>Function</code>.</td>
</tr>
<tr>
<td>groupBy</td>
<td>Gets a key for each element using a <code>Function</code> and puts the key and element into a <code>Multimap</code>.</td>
</tr>
</tbody>
</table>
Cheat Sheet

**sortThisBy**
Gets some attribute from each element using a *Function* and sorts the list by the natural order of that attribute.

**toSortedListBy**
Gets some attribute from each element using a *Function* and returns a new list sorted by the natural order of that attribute.

**minBy**
Gets some attribute from each element and returns the element whose attribute is minimal.

**maxBy**
Gets some attribute from each element and returns the element whose attribute is maximal.

**toMap**
Gets a key and value for each element using two *Functions* and returns a new map containing all the key/value pairs.
<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>select</code></td>
<td>Returns the elements of a collection that satisfy some condition (Predicate).</td>
</tr>
<tr>
<td><code>reject</code></td>
<td>Returns the elements of a collection that do not satisfy the Predicate.</td>
</tr>
<tr>
<td><code>detect</code></td>
<td>Finds the first element that satisfies the Predicate.</td>
</tr>
<tr>
<td><code>count</code></td>
<td>Returns the number of elements that satisfy the Predicate.</td>
</tr>
<tr>
<td><code>anySatisfy</code></td>
<td>Returns true if any element satisfies the Predicate.</td>
</tr>
<tr>
<td><code>allSatisfy</code></td>
<td>Returns true if all elements satisfy the Predicate.</td>
</tr>
<tr>
<td><code>forEach</code></td>
<td>Executes the Procedure on each element, doesn't return anything.</td>
</tr>
<tr>
<td>Method</td>
<td>Description</td>
</tr>
<tr>
<td>---------------</td>
<td>---------------------------------------------------------------------------------------------------------------------------------------------</td>
</tr>
<tr>
<td>min</td>
<td>Returns the minimum element using either the natural order or a Comparator.</td>
</tr>
<tr>
<td>max</td>
<td>Returns the maximum element using either the natural order or a Comparator.</td>
</tr>
<tr>
<td>toSortedList</td>
<td>Returns a new list, sorted using either the natural order or a Comparator.</td>
</tr>
<tr>
<td>makeString</td>
<td>Converts the collection to a string using optional start, end, and separator strings.</td>
</tr>
<tr>
<td>appendString</td>
<td>Converts the collection to a string and appends it to an Appendable.</td>
</tr>
<tr>
<td>zip</td>
<td>Takes a second RichIterable and pairs up all the elements. If one of the two RichIterables is longer than the other, its remaining elements are ignored.</td>
</tr>
<tr>
<td>zipWithIndex</td>
<td>Zips the collection with the Integer indexes 0 to n-1.</td>
</tr>
<tr>
<td>chunk</td>
<td>Splits a collection into fixed size chunks. The final chunk will be smaller if the collection doesn't divide evenly.</td>
</tr>
</tbody>
</table>
• `collectWith()`, `selectWith()`, and `rejectWith()` are alternate forms of `collect()`, `select()`, and `reject()`.

• Original forms all take a single parameter, a code block which takes a single parameter.

• What if we want to find groups of people at different ages?

```java
MutableList<Person> voters = people.select(
    person -> person.getAge() > 18;
);
```
**collectWith(), selectWith(), and rejectWith()**

- `collectWith()`, `selectWith()`, and `rejectWith()` are alternate forms of `collect()`, `select()`, and `reject()`.
- `...with()` forms take two parameters:
  - a code block which takes two parameters,
  - an object that gets passed as the second argument to the code block.
- Store the two-argument block in a constant to avoid object creation.

**Code Example**

```java
Predicate2<Person, Integer> age =
    (person, age) -> person.getAge() > age;

MutableList<Person> drivers = people.selectWith(age, 17);
MutableList<Person> voters = people.selectWith(age, 18);
MutableList<Person> drinkers = people.selectWith(age, 21);
MutableList<Person> sunsetRobotSquad = people.selectWith(age, 160);
```
“We should forget about small efficiencies, say about 97% of the time: premature optimization is the root of all evil”

– Donald Knuth
we
BUILD

Learn more at GS.com/Engineering

© 2016 Goldman Sachs. This presentation should not be relied upon or considered investment advice. Goldman Sachs does not warrant or guarantee to anyone the accuracy, completeness or efficacy of this presentation, and recipients should not rely on it except at their own risk. This presentation may not be forwarded or disclosed except with this disclaimer intact.