GS Collections and Java 8 Lambdas

Code Kata

March 2014

10 Second Intro

What is GS Collections?

A Java Collections Library.

What is a Code Kata?

A programming exercise which helps hone your skills through practice.

What is GS Collections?

- A supplement or replacement for the Java Collections Framework (JCF).
- Developed 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 GS Collections –

I hear and I forget.

I see and I remember.

I do and I understand.

- Confucius

GS Collections Code Kata

- New concepts are introduced in the slides.
- Coding exercises are at the end of each section.

Iteration Patterns

Sort

- 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?

Domain

```
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);
```

Javadoc

```
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;
    }
    return o1.getFirstName().compareTo(o2.getFirstName());
  }
});
```

Javadoc

```
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*.

Lambda syntax

```
Collections.sort(people, (Person o1, Person o2) -> {
    int lastName = o1.getLastName().compareTo(o2.getLastName());
    if (lastName != 0) {
      return lastName;
    }
    return o1.getFirstName().compareTo(o2.getFirstName());
  });
```

Javadoc

```
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()`?

Javadoc

```
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);
```

Javadoc

```
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 address.
  - Select only those people whose age is 18 or higher.

- We want the methods `sort()`, `min()`, `max()`, `collect()`, `select()`, etc. on every collection.
- How can we accomplish this in code?

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.

GS Collections Interfaces

```
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);
  ...
}
```

Iteration Pattern

- *Collect* pattern (aka *map* or *transform*).
- Returns a new collection where each element has been transformed
  - e.g. collect each person's address.
- **`Function`** is the type that takes an object and returns an object of a different type
  - aka *Transformer*

JCF Example

```
List<Person> people = ...
List<Address> addresses = new ArrayList<Address>();
for (Person person : people)
{
  addresses.add(person.getAddress());
}
```

Iteration Pattern

- **`Function`** is the type that takes an object and returns another type.

GS Collections Example

```
MutableList<Person> people = ...
MutableList<Address> addresses = people.collect(
  new Function<Person, Address>()
  {
    public Address valueOf(Person person)
    {
      return person.getAddress();
    }
  });
```

Iteration Pattern

- **`Function`** is the type that takes an object and returns another type.

GS Collections Example

```
MutableList<Person> people = ...
MutableList<Address> addresses =
   people.collect(person -> person.getAddress());

MutableList<Address> addresses =
  people.collect(Person::getAddress);
```

Iteration Pattern

- The loop moves in to the implementation of **`collect()`**.
- Let's look at a realistic implementation of **`collect()`** for **`FastList`**.

GS Collections Example

```
public <V> MutableList<V> collect(
  Function<? super T, ? extends V> function)
{
  MutableList<V> result = FastList.newList(this.size);
  for (int i = 0; i < this.size; i++) {
    result.add(function.valueOf(this.items[i]));
  }
  return result;
}
```

Iteration Pattern

- *Select* pattern (aka *filter*).
- Returns the elements of a collection that satisfy some condition
  - e.g. select only those people whose age is 18 or higher.
- **`Predicate`** is the type that takes an object and returns a boolean.

GS Collections Example

```
MutableList<Person> people = ...
MutableList<Person> adults = people.select(
  new Predicate<Person>()
  {
    public boolean accept(Person each)
    {
      return each.getAge() >= 18;
    }
  });
```

Iteration Pattern

- *Select* pattern (aka *filter*).
- Returns the elements of a collection that satisfy some condition
  - e.g. select only those people whose age is 18 or higher.
- **`Predicate`** is the type that takes an object and returns a boolean.

GS Collections Example

```
MutableList<Person> people = ...
MutableList<Person> adults =
   people.select(each -> each.getAge() >= 18);
```

Examples

Collect Returns a new collection where each element has been transformed.

Select Returns the elements of a collection that satisfy some condition.

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

To set up the Kata, follow these instructions:

- To locate the zip, navigate to the GitHub page's kata repository (https://github.com/goldmansachs/gs-collections-kata), and click on Download ZIP in the bottom right corner.

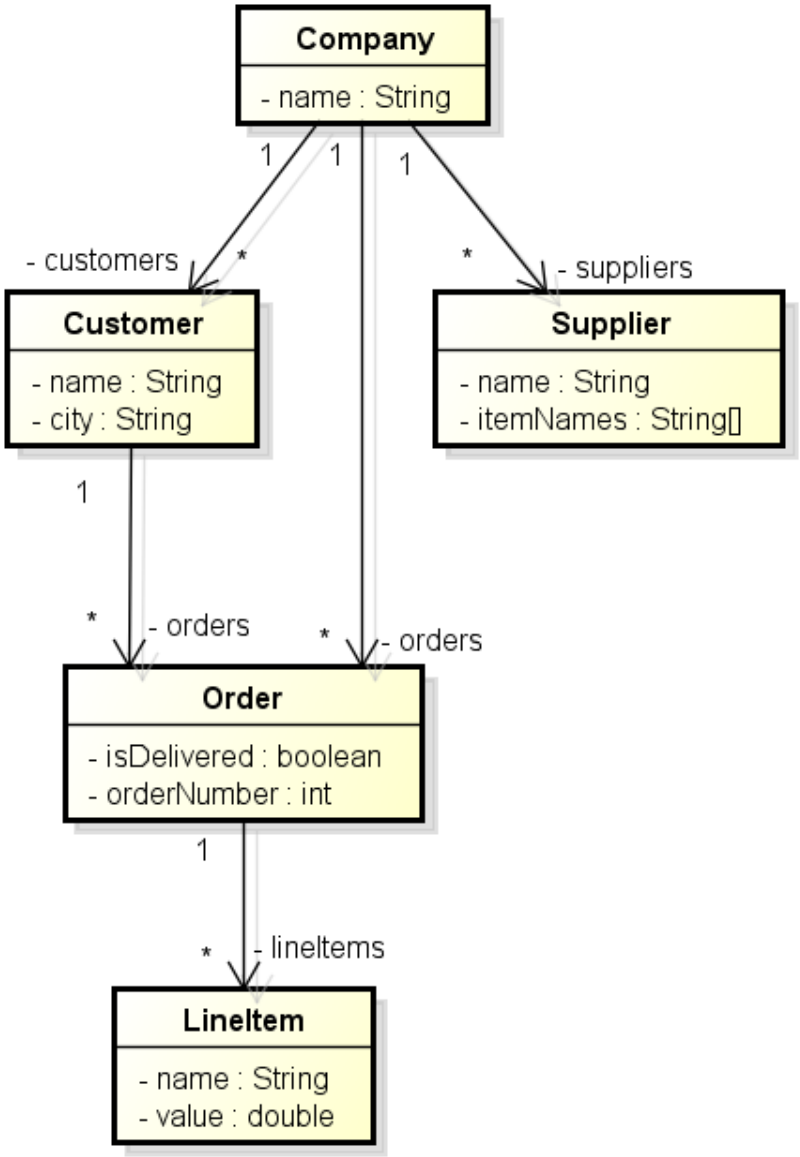
Eclipse Setup Instructions

1. Unzip **gs-collections-kata-master.zip** - should result in a folder called gs-collections-kata-master
2. Open the project folder using Eclipse. If prompted, choose Java project for the type and name it "kata".
3. Select the **com.gs.collections.kata** package under **src/test/java**, right click and select **RunAs -> JUnit Test**.

IntelliJ Setup Instructions

1. Unzip **gs-collections-kata-master.zip** - should result in a folder called gs-collections-kata-master
2. Open the project: **File -> Open**, navigate to where you checked out the project, and open the project marked by the "IJ" icon.
3. Select the com.gs.collections.kata package under **src/test/java**, right click and select **Run "Tests in 'com.gs.collections.kata'"**

You should get several test failures. You will fix these tests as part of the Kata exercises.


Company
- name : String
1
1
1
- customers
*
*
- suppliers
Customer
- name : String
- city : String
Supplier
- name : String
- itemNames : String[]
1
*
- orders
*
- orders
Order
- isDelivered : boolean
- orderNumber : int
1
*
- lineItems
LineItem
- name : String
- value : double

Domain

- GS Collections Kata uses LineItems, Companies, Orders, Customers, and Suppliers.
- Our tests extend `CompanyDomainForKata`, which sets up some customers, suppliers and orders for a company.
- Data is available to you through getters on `this.company`
  - For example `this.company.getCustomers();`

Hints

- Most changes will be under **src/test**.
- Some changes will be under **src/main**.
- Feel free to refactor the domain under **src/main**.
- Pay close attention to the Javadoc for instructions and hints.
- Use the IDE support for Javadoc **@link**s.

Kata Exercise 1

Exercise 1

- Find `Exercise1Test`; it has assertion failures if you run it as a test.
- Figure out how to get the tests to pass using what you have seen so far.
- Should take about 15 minutes.

Solution 1

```
public void getCustomerNames()
{
  MutableList<Customer> customers =
    this.company.getCustomers();

  MutableList<String> customerNames =
    customers.collect(nameFunction);

  ...
  Assert.assertEquals(expectedNames, customerNames);
}
```

Solution 1

```
public void getCustomerCities()
{
  MutableList<String> customerCities =
    customers.collect(customer -> customer.getCity());

  ...
  Assert.assertEquals(expectedCities, customerCities);
}
```

Solution 1

```
public void getLondonCustomers()
{
   ...
   MutableList<Customer> customersFromLondon =
     customers.select(
          customer -> customer.getCity().equals("London")
     );
   Verify.assertSize(2, customersFromLondon);
}
```

TESTUTILS

Verify

- GS Collections distribution includes **gs-collections-testutils.jar**.
  - Includes helpful utility for writing unit tests.
  - Collection-specific.
  - Implemented as an extension of JUnit.
  - Most important class is called **`Verify`**.

Code Example

Example from the previous solution

```
Verify.assertSize(2, customersFromLondon);
```

Instead of

```
Assert.assertEquals(2, customersFromLondon.size());
```

Verify

Several other self-explanatory examples:

Code Example

```
Verify.assertEmpty(FastList.newList());
Verify.assertNotEmpty(FastList.newListWith(1));
Verify.assertContains(1, FastList.newListWith(1));
```

GS Collections Testutils

- It's possible to go too far.
- The first form is more verbose.
- The second form asserts more because of the contract of **`equals()`**.

Bad

```
Verify.assertSize(3, list);
Verify.assertContainsAll(list, 1, 2, 3);
```

Good

```
Assert.assertEquals(
  FastList.newListWith(1, 2, 3),
  list);
```

Benefits of GS Collections

Pros

- **Increased readability.**
- Reduced duplication – less to test.
- Simplified iteration.
- Optimized by type for memory and speed.

Cons

- Before Java 8 there were no lambdas or closures
  - Verbose anonymous inner class syntax

Readability

- **Object-oriented programming** is a paradigm that groups data with the methods that predicates on the data.
- **Functional programming** is a paradigm that treats computation as the evaluation of mathematical functions and avoids state and mutable data.

- Possible to merge the two paradigms and wind up with very readable code.
- Where they sometimes clash is mutable state.

Example

x = 5

x = 6

Imperative Programming

- In Java this is OK.
- The value of *x* changes over time.

Math

- In pure math this makes no sense.
- The value of x is 5, it cannot be reassigned.

Imperative Programming

- In imperative programming, **time** is important.
- The truth of conditions changes with the state of the program.
- This can decrease readability.
  - can be difficult to reason about.
  - can be difficult to test.
- Think architecture that is concurrent and based on asynchronous callbacks.

Code Example

```
List<Person> people = ...
List<Address> addresses = new ArrayList<Address>();
// Not true yet! It's an empty list, not a list of addresses!
for (Person person : people)
{
  addresses.add(person.getAddress());
}
```

Imperative vs. Functional

- `fibonacci()` function returns the same output for the same input.
- We can tell it doesn't use mutable state; it can be static.

- Iterators are not functional.
- `next()` takes no input, yet returns different values over time.
- It obviously uses internal state.

Code Example

```
for (int i = 0; i < 10; i++)
{
  System.out.println(this.fibonacci(i));
}
// vs.
for (int i = 0; i < 10; i++)
{
  System.out.println(this.fibonacciIterator.next());
}
```

Benefits

- Iteration patterns are methods on our collections.
- Simply naming the patterns enhances readability:

```
people.collect(...)
```

Drawbacks

- These methods take a parameter which is a bit of code:
  - how to collect or select.
- The thing inside the parentheses is now less readable:
  - represented as an anonymous inner class.

Readability

- Ideally, Java would have native support for anonymous functions (aka *lambda expressions*, erroneously aka *closures*).
- The syntax below works with Java 8.

JDK 8 Prototype

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

MutableList<Address> addresses =
  people.collect(person -> person.getAddress());

MutableList<Address> addresses =
  people.collect(Person::getAddress);
```

Readability

- Often you need to write out a `Function`.
- It makes sense to hide the boilerplate code on the domain object.

Collect with hidden Function

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

MutableList<Address> addresses =
   people.collect(Person.TO_ADDRESS);
```

Function on domain Object

```
public class Person {
  public static final Function<Person, Address> TO_ADDRESS =
      person -> person.getAddress();
...
}
```

Readability

Encapsulating the `Function` looks very similar to the proposed lambda syntax.

Encapsulated Function

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

MutableList<Address> addresses =
  people.collect(Person.TO_ADDRESS);
```

JDK 8 Prototype

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

MutableList<Address> addresses =
   people.collect(Person::getAddress);
```

Readability

- We have other tricks for dealing with boilerplate code.
- GS Collections provides several “lambda factories,” like `Predicates`.

GS Collections Select with Predicates Factory

```
MutableList<Integer> mutableList =
  FastList.newListWith(25, 50, 75, 100);

MutableList<Integer> selected =
  mutableList.select(Predicates.greaterThan(50));
```

Iteration Pattern

- `Predicates` also lets you create common `Predicate`s from `Function`s.
- Combines well with the previous tip.

GS Collections Select with Predicates Factory

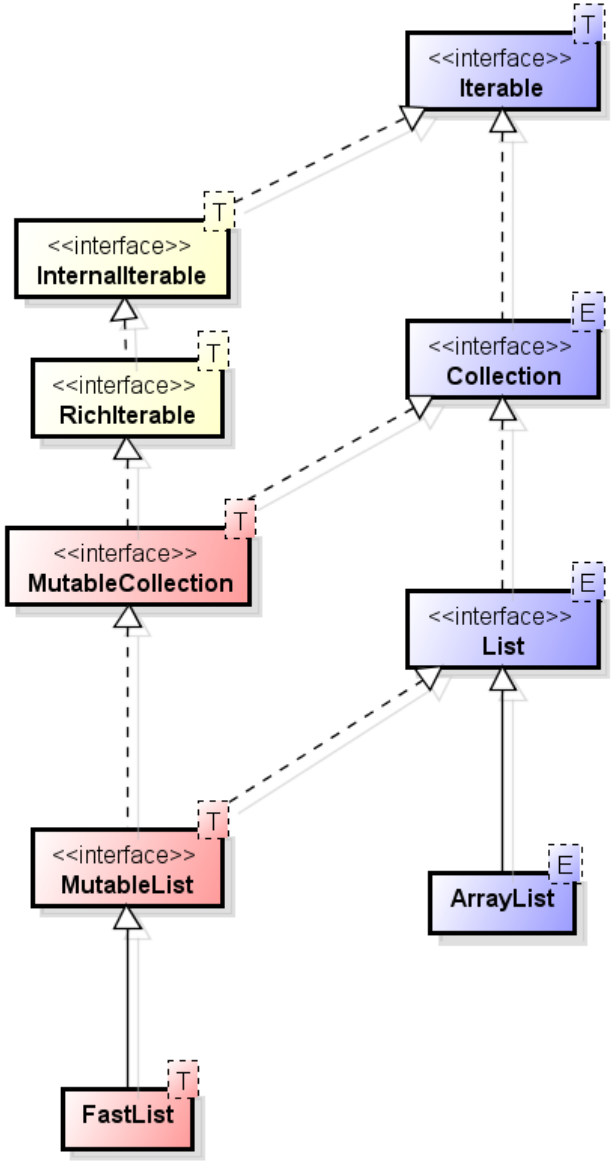
```
MutableList<Person> people = ...;

MutableList<Person> theSmiths = people.select(
  Predicates.attributeEqual(Person.LAST_NAME, "Smith"));

MutableList<Person> theSmiths = people.select(
  Predicates.attributeEqual(Person::getLastName, "Smith"));
```

KATA EXERCISE 2

Exercise 2

- Fix `Exercise2Test`.
- Solve the same questions as `Exercise1Test`.
- Use the tips to make your answers as readable as possible.
- From now on, you'll have to access the domain objects on your own.
  - Start with `this.company.getSomething()`.
- Should take about 15 minutes.

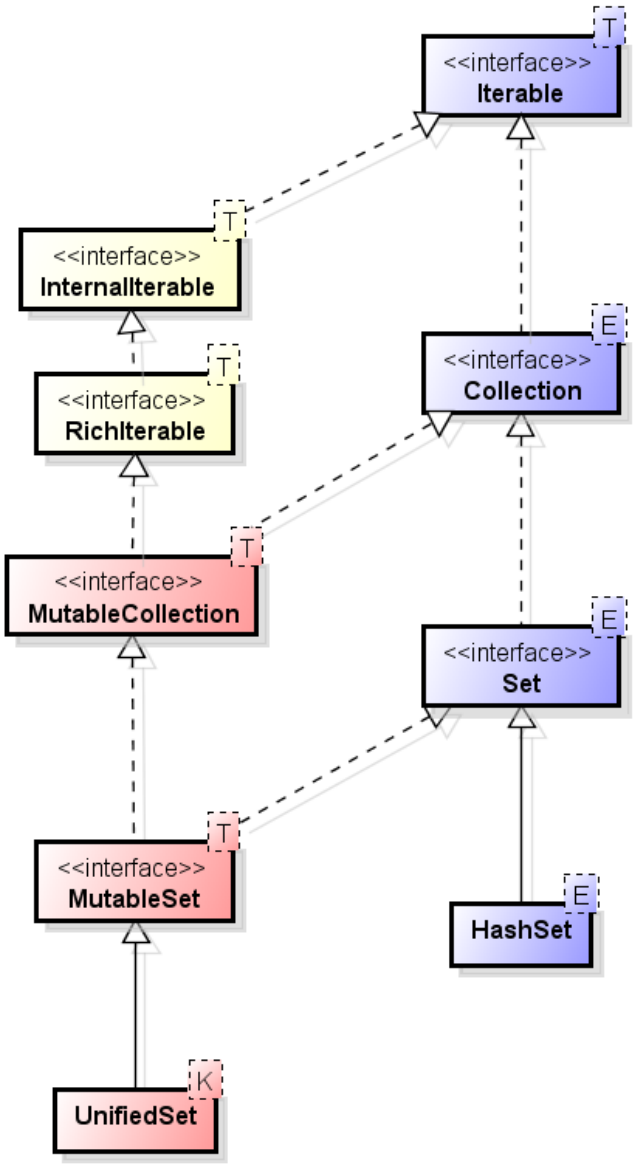
Inheritance Hierarchy



Lists

- **`MutableList`** extends **`List`**.
- **`FastList`** is a drop-in replacement for **`ArrayList`**.
- **`FastList`** could extend **`ArrayList`**, but we chose not to.
- Iteration patterns were pulled up higher into **`RichIterable`**.

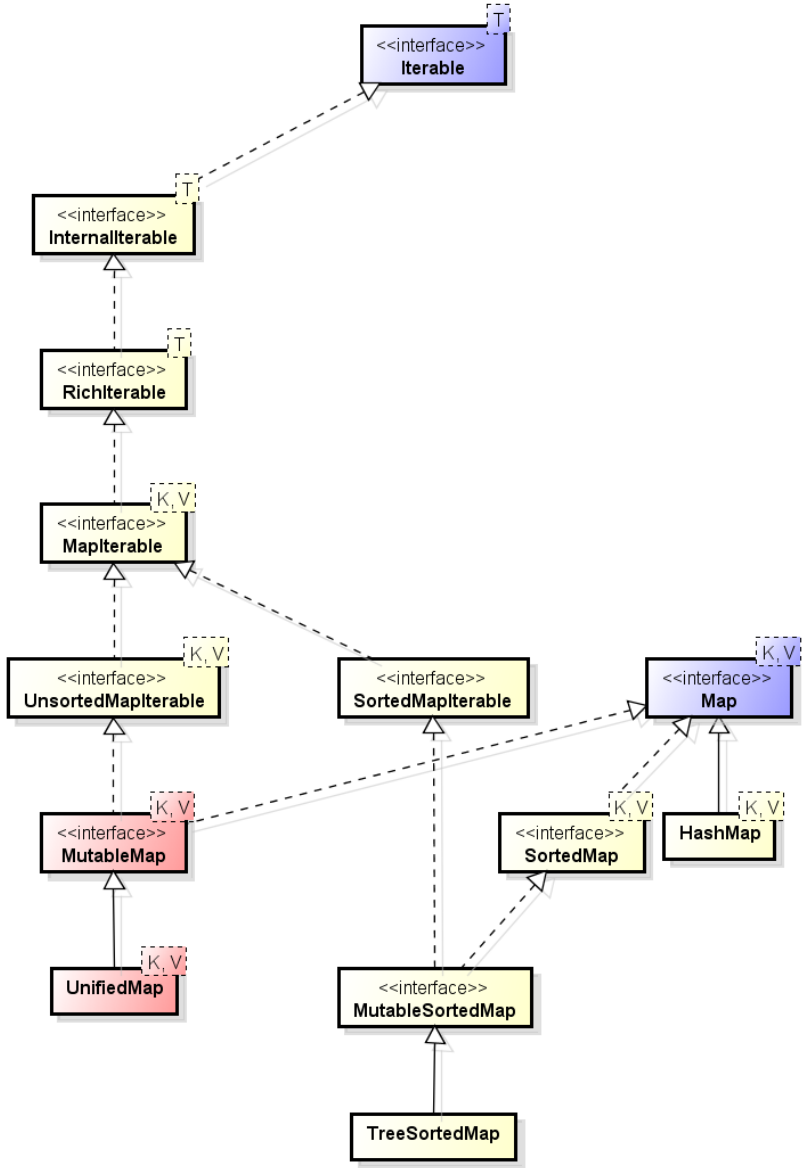
Inheritance Hierarchy



Sets

- **`MutableSet`** extends **`Set`**.
- **`UnifiedSet`** is a drop in replacement for **`HashSet`**.
- Everything about Lists is analogous for Sets (and Bags).
- This is why iteration patterns were pulled up higher into **`RichIterable`** –
  - `min()` for example is identical across collection types.

Inheritance Hierarchy



Maps

- **`MutableMap`** extends **`Map`**.
- **`UnifiedMap`** is a drop in replacement for **`HashMap`**.

Solution 2

```
public class Customer
{
   public static final Function<Customer, String> TO_NAME =
     Customer::getName;
   ...
}
```

Solution 2

```
public void getCustomerCities()
{
  MutableList<String> customerCities =
    this.company.getCustomers().collect(Customer.TO_CITY);


  ...
  Assert.assertEquals(expectedCities, customerCities);
}
```

Solution 2

```
public class Customer
{
   ...
   public static final
   Function<Customer, String> TO_CITY = Customer::getCity;
   ...
}
```

Solution 2

```
public void getLondonCustomers()
{
  MutableList<Customer> customersFromLondon =
    this.company.getCustomers().select(
      Predicates.attributeEqual(
        Customer.TO_CITY,
        "London"));

  Verify.assertSize(2, customersFromLondon);

  Verify.assertSize(2, this.company.getCustomers().select(
      Predicates.attributeEqual(Customer::getCity, "London")));

  Verify.assertSize(2, this.company.getCustomers().select(
      customer -> “London”.equals(customer.getCity())));
}
```

Immutability

Pros

Why would we prefer immutable data structures?

Cons

Why wouldn't we prefer immutable data structures?

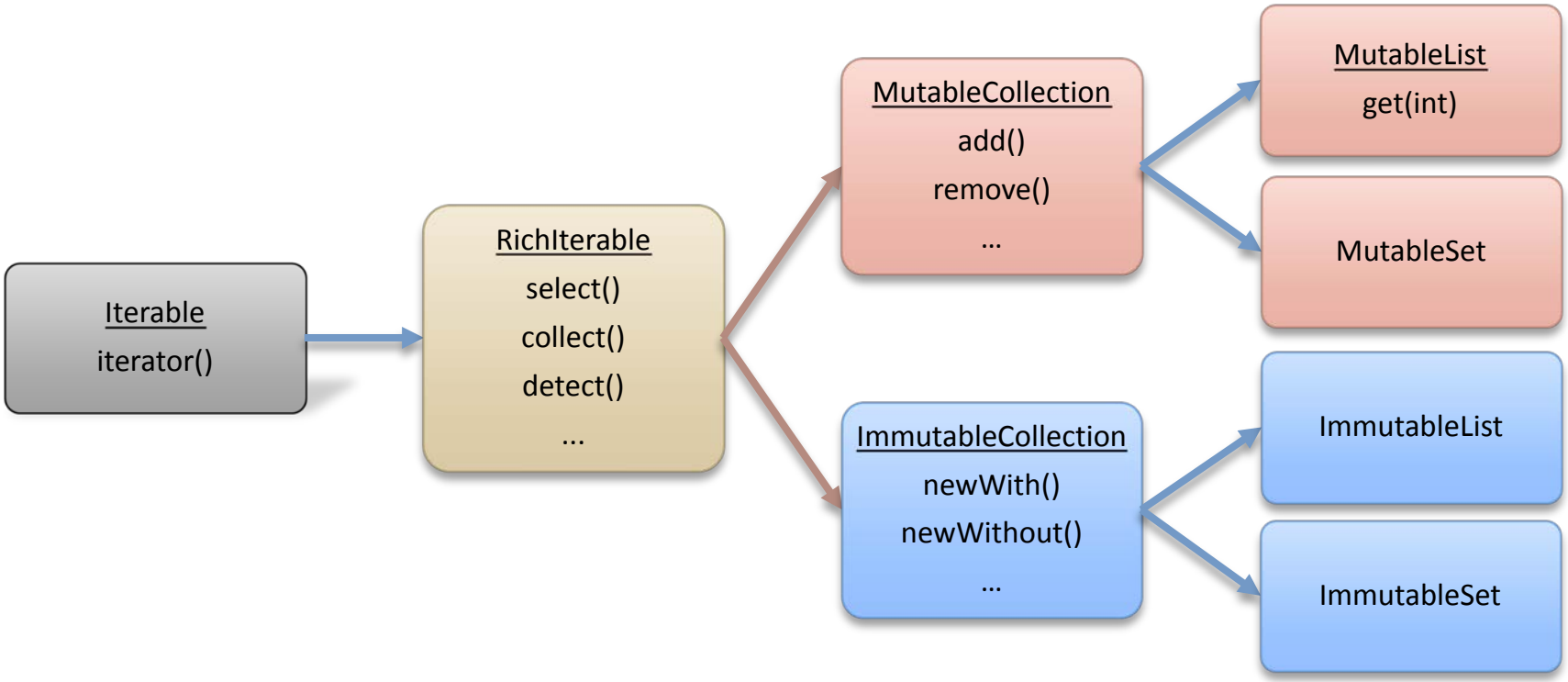
Pros

- Easier to reason about because they do not have complex state spaces that change over time.
- Can pass them around freely without making defensive copies.
- No way to corrupt through concurrent access.
- They make safe hash-table keys.
  - If an object is mutated after it is placed into a `HashSet`, for example, that object may not be found the next time you look into the `HashSet`.

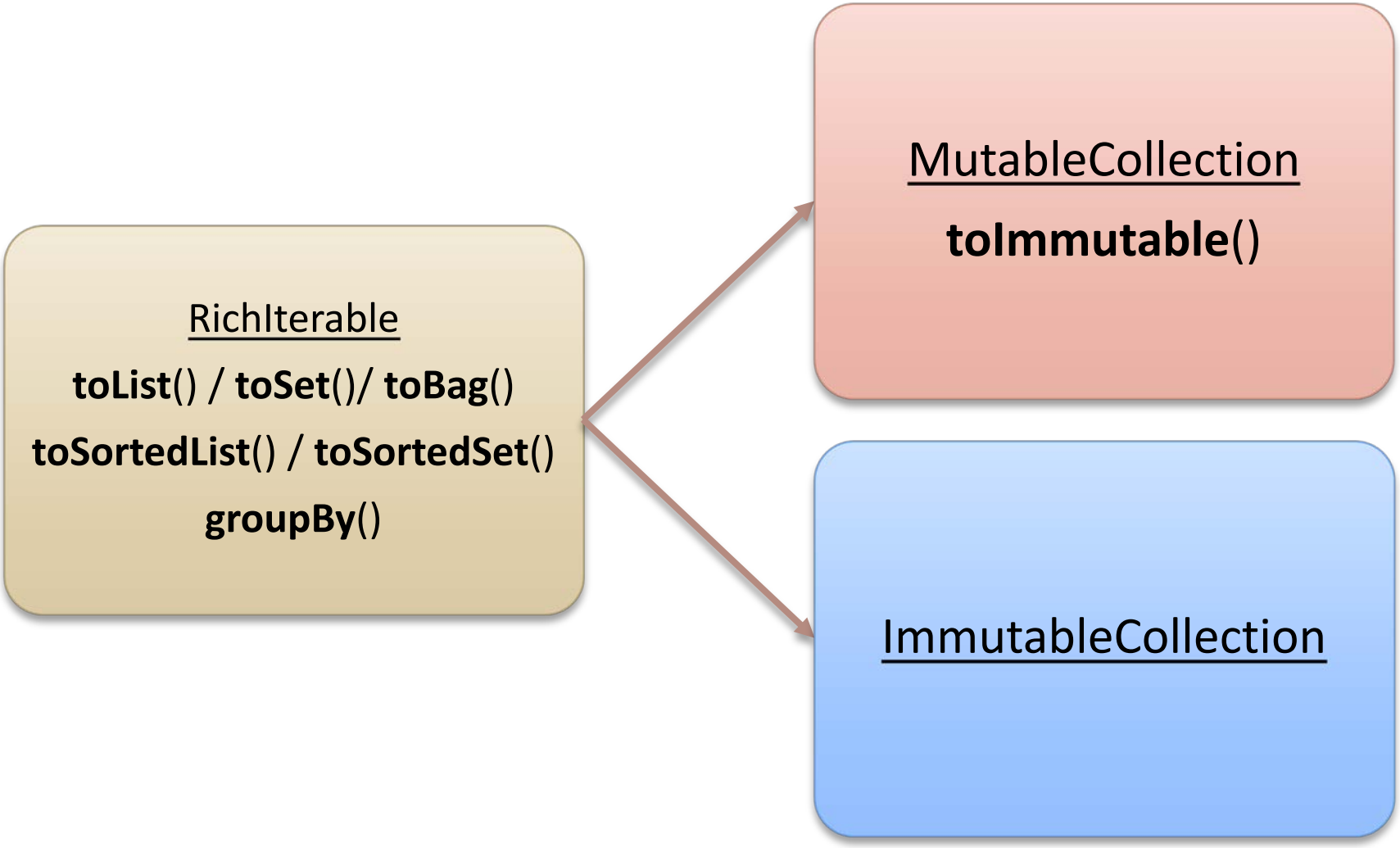
Cons

- They sometimes require that a large object graph be copied where otherwise an update could be done in place.
- As a result, it is common for libraries to provide mutable alternatives to immutable classes.
  - For example, `StringBuilder` is a mutable alternative to `String`.

Class Diagram



Class Diagram



Conversion Methods

`toList(), toSortedList(), toSet(), toSortedSet(), toBag()`

- Return mutable copies.
- Return new copy even when called on a collection of the same type.

Code Example

```
MutableList<Integer> list = FastList.newListWith(3, 1, 2, 2, 1);

MutableList<Integer> noDupes =
  list.toSet().toSortedList();

Assert.assertEquals(
  FastList.newListWith(1, 2, 3),
  noDupes);
```

Overview

- `ImmutableCollection` interface does not extend `Collection`:
  - No mutating methods.
  - Mutating requires a new copy.
- GS Collections also has "memory-efficient" collections but they are largely superseded by `ImmutableCollection`s.

Truly Immutable

```
ImmutableList<Integer> immutableList =
  FastList.newListWith(1, 2, 3).toImmutable();

ImmutableList<Integer> immutableList2 =
  Lists.immutable.of(1, 2, 3);

Verify.assertInstanceOf(
  ImmutableTripletonList.class,
  immutableList);
```

Equality

Should a mutable list equal an immutable list?

Code Example

```
MutableList<Integer> mutable =
  FastList.newListWith(1, 2, 3);

ImmutableList<Integer> immutable =
  Lists.immutable.of(1, 2, 3);

mutable.equals(immutable);
```

Equality

Should a list and a set be equal?

Code Example

```
MutableList<Integer> list =
  FastList.newListWith(1, 2, 3);

MutableSet<Integer> set =
  UnifiedSet.newSetWith(1, 2, 3);

list.equals(set);
```

Equality

Should a mutable list equal an immutable list?

A list is a `List` because it allows duplicates and preserves order, so **yes**.

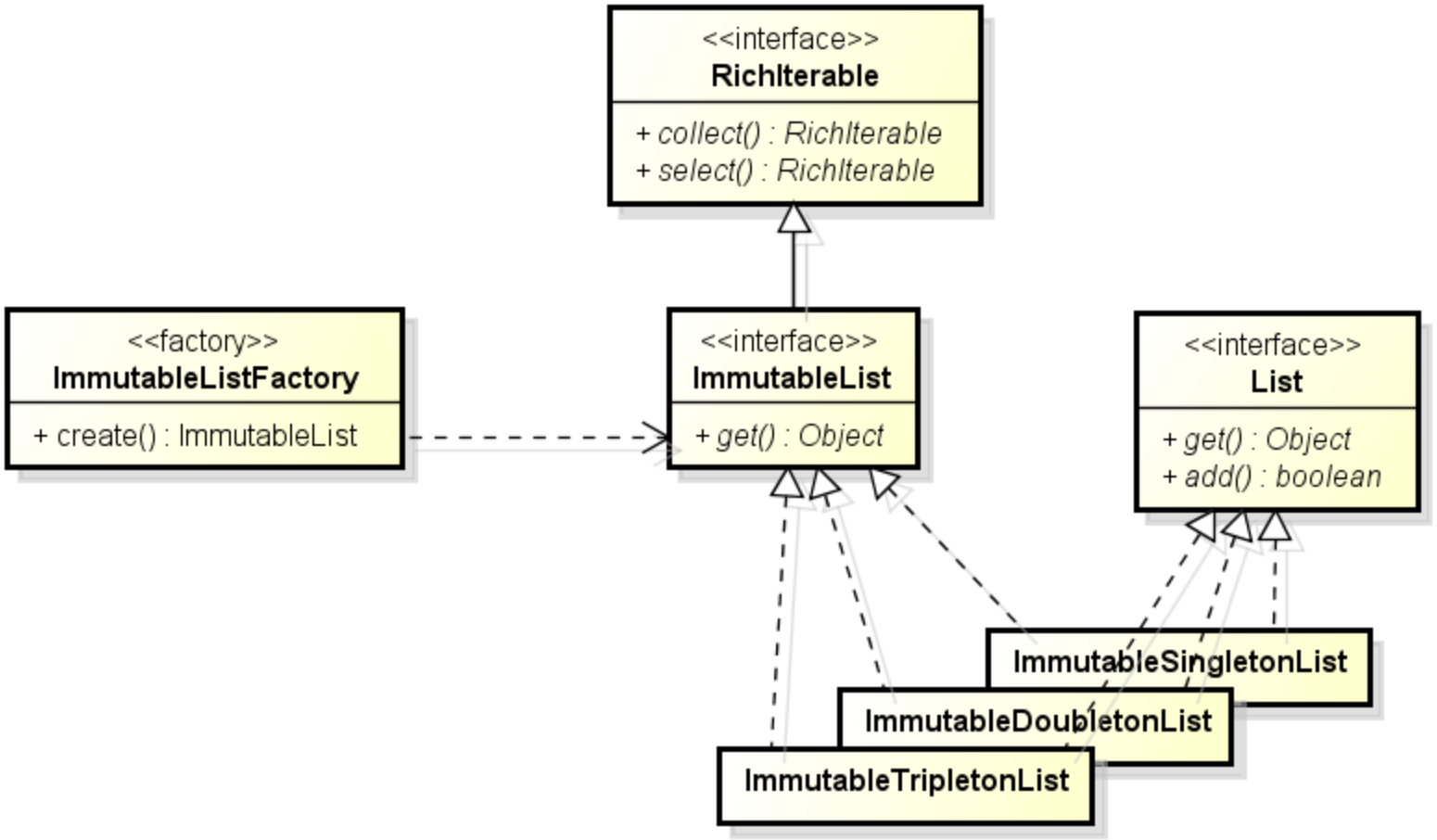
Code Example

```
MutableList<Integer> mutable =
  FastList.newListWith(1, 2, 3);

ImmutableList<Integer> immutable =
  Lists.immutable.of(1, 2, 3);

Assert.assertEquals(mutable, immutable);
```

Here's the implementation of `ArrayList.equals()`, really on `AbstractList`:

```
public boolean equals(Object o) {
  if (o == this)
    return true;
  if (!(o instanceof List))
    return false;
  ListIterator<E> e1 = this.listIterator();
  ListIterator e2 = ((List) o).listIterator();
  while (e1.hasNext() && e2.hasNext()) {
    E o1 = e1.next();
    Object o2 = e2.next();
    if (!(o1 == null ? o2 == null : o1.equals(o2))) {
      return false;
    }
  }
  return !(e1.hasNext() || e2.hasNext());
}
```

Inheritance diagram



Overview

- Implementations *are* `Collection`s in order to satisfy the existing contract of `equals()` on `List` and `Set`.
- Can be cast to `Collection`.
  - Brings back the mutating methods.
  - Brings back the `UnsupportedOperationException`s.
  - Convenient for interop.

Effectively Immutable

```
List<Integer> list = immutableList.castToList();
Verify.assertThrows(
  UnsupportedOperationException.class,
  () -> list.add(4));
```

More Iteration Patterns

Other patterns that use Predicate

Select	Returns the elements of a collection that satisfy the `Predicate`.
Reject	Returns the elements of a collection that *do not* satisfy the `Predicate`.
Count	Returns the number of elements that satisfy the `Predicate`.

Short-circuit patterns that use Predicate

Detect	Finds the first element that satisfies the `Predicate`.
Any Satisfy	Returns true if any element satisfies the `Predicate`.
All Satisfy	Returns true if all elements satisfy the `Predicate`.

KATA EXERCISE 3

Exercise 3

- Fix `Exercise3Test`.
- Use the other iteration patterns that take a `Predicate`.
- Should take about 20 minutes.

KATA

Solution 3

```
private static final Predicate<Customer> CUSTOMER_FROM_LONDON =
   customer -> "London“.equals(customer.getCity());

public void doAnyCustomersLiveInLondon()
{
   boolean anyCustomersFromLondon =
   this.company.getCustomers().anySatisfy(CUSTOMER_FROM_LONDON);
   Assert.assertTrue(anyCustomersFromLondon);
}
```

Solution 3

```
public void doAllCustomersLiveInLondon()
{
   boolean allCustomersFromLondon =
   this.company.getCustomers().allSatisfy(CUSTOMER_FROM_LONDON);
   Assert.assertFalse(allCustomersFromLondon);
}
```

Solution 3

```
public void howManyCustomersLiveInLondon()
{
   int numberOfCustomerFromLondon =
     this.company.getCustomers()
      .count(CUSTOMER_FROM_LONDON);

   Assert.assertEquals(2, numberOfCustomerFromLondon);
}
```

Solution 3

```
public void getLondonCustomers()
{
  MutableList<Customer> customersFromLondon =
    this.company.getCustomers()
      .select(CUSTOMER_FROM_LONDON);

  Verify.assertSize(2, customersFromLondon);
}
```

Solution 3

```
public void getCustomersWhoDontLiveInLondon()
{
  MutableList<Customer> customersNotFromLondon =
    this.company.getCustomers()
      .reject(CUSTOMER_FROM_LONDON);

  Verify.assertSize(1, customersNotFromLondon);
}
```

Solution 3

```
public class Company
{
   ...

   public Customer getCustomerNamed(String name)
   {
      return this.customers.detect(
         Predicates.attributeEqual(Customer::getName, name));
   }
}
```

Not closure

Advanced Testutils

Verify

Verify includes additional assertions based on iteration patterns.

Code Example

```
MutableList<Integer> list =
  FastList.newListWith(1, 2, 0, -1);

Verify.assertAllSatisfy(list,
  IntegerPredicates.isPositive());
```

```
junit.framework.AssertionFailedError: The following
  items failed to satisfy the condition <[0, -1]>
```

Target Collections

Iteration Pattern

- 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?

Example Code

```
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

```
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

```
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);
```

FlatCollect Pattern

Background on collect()

- `flatCollect()` is a special case of `collect()`.
- With `collect()`, when the `Function` returns a collection, the result is a collection of collections.

Code Example

```
MutableList<Person> people = ...;
Function<Person, MutableList<Address>> addressesFunction =
    person -> person.getAddresses();

MutableList<MutableList<Address>> addresses =
  people.collect(addressesFunction);
```

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.

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

Code Example

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

MutableList<Address> addresses =
  people.flatCollect(person -> person.getAddresses());
```

collect()

```
MutableList<MutableList<Address>> addresses =
  people.collect(Person::getAddresses);
```

flatCollect()

```
MutableList<Address> addresses =
  people.flatCollect(Person::getAddresses);
```

Kata Exercise 4

Exercise 4

- Fix `Exercise4Test`.
- Should take about 20 minutes.

Solution 4

```
public MutableList<Order> getOrders()
{
  return this.customers.flatCollect(
      customer -> customer.getOrders()
  );
}
// or
public MutableList<Order> getOrders()
{
  return this.customers.flatCollect(Customer::getOrders);
}
```

Solution 4

```
MutableSet<String> actualItemNames =
  this.company.getOrders()
    .flatCollect(Order.TO_LINE_ITEMS)
    .collect(LineItem.TO_NAME,
   UnifiedSet.<String>newSet());
```

Solution 4

```
public void findCustomerNames()
{
  MutableList<String> names =
    this.company.getCustomers()
      .collect(Customer.TO_NAME);

  MutableList<String> expectedNames = ...;
  Assert.assertEquals(expectedNames, names);
}
```

Static Utility

Iteration Pattern

- Using methods on the interfaces is the preferred, object-oriented approach.

GS Collections Select with Predicates Factory

```
MutableList<Integer> mutableList = ...;

MutableList<Integer> selected =
  mutableList.select(Predicates.greaterThan(50));
```

Iteration Pattern

- Using methods on the interfaces is the preferred, object-oriented approach.
  - But it's not always feasible.
- Static utility classes like `Iterate`, `ListIterate`, etc. provide interoperability with JCF.

GS Collections Select with Predicates Factory

```
List<Integer> list = ...;

MutableList<Integer> selected =
  ListIterate.select(list, Predicates.greaterThan(50));
```

Iteration Pattern

- Using methods on the interfaces is the preferred, object-oriented approach.
  - But it's not always feasible.
- Static utility classes like `Iterate`, `ListIterate`, etc. provide interoperability with JCF.
- Static utility classes like `ArrayIterate` and `StringIterate` show that iteration patterns work on other types as well.

GS Collections Select with Predicates Factory

```
Integer[] array = ...;
MutableList<Integer> selected =
  ArrayIterate.select(array, Predicates.greaterThan(50));

String string = StringIterate.select(
  "1a2a3",
  CharPredicate.IS_DIGIT);
Assert.assertEquals("123", string);
```

Iteration Pattern

- Static utility for parallel iteration.
- Hides complexity of writing concurrent code.
- Looks like the serial case.
- Notice the lack of locks, threads, pools, executors, etc.
- Order is preserved in the result.

GS Collections Select with Predicates Factory

```
List<Integer> list = ...;

Collection<Integer> selected =
  ParallelIterate.select(list, Predicates.greaterThan(50));
```

Cheat Sheet

- `Iterate` (Iterable)
  - `ListIterate`
  - `ArrayListIterate`
- `MapIterate` (Map)
- `LazyIterate` (Iterable)
- `ArrayIterate` (T[])
- `StringIterate` (String)
- `ParallelIterate` (Iterable)
- `ParallelMapIterate` (Map)
- `ParallelArrayIterate` (T[])

Benefits of the OO API

Static Utility

- Let's look at the full implementation of `Collections.sort()`
- What's wrong with this code?

JCF Sort

```
public static <T> void sort(List<T> list, Comparator<? super T> c) {
  Object[] array = list.toArray();
  Arrays.sort(array, (Comparator) c);
  ListIterator iterator = list.listIterator();
  for (int i = 0; i < array.length; i++) {
    iterator.next();
    iterator.set(array[i]);
  }
}
```

Static Utility

- This code is fine for `LinkedList`.
- The code is suboptimal for `ArrayList` (and `FastList`).
  - Unnecessary array copy.
  - Unnecessary iterator created.
  - Unnecessary calls to `set()`.

JCF Sort

```
public static <T> void sort(List<T> list, Comparator<? super T> c) {
  Object[] array = list.toArray();
  Arrays.sort(array, (Comparator) c);
  ListIterator iterator = list.listIterator();
  for (int i = 0; i < array.length; i++) {
    iterator.next();
    iterator.set(array[i]);
  }
}
```

Object Oriented

- `FastList` has a simpler and more optimal implementation.
- Objects group logic with the data it operates on.
- This logic makes sense for an array-backed structure.

FastList Sort

```
public FastList<T> sortThis(Comparator<? super T> comparator)
{
  Arrays.sort(this.items, 0, this.size, comparator);
  return this;
}
```

KATA EXERCISE 5

Exercise 5

- Fix the five methods in `Exercise5Test`.
- Solve them using the static utility classes.
- Exercise 5 should take about 25 minutes.

Solution 5

```
public void findSupplierNames()
{
  MutableList<String> supplierNames =
    ArrayIterate.collect(
      this.company.getSuppliers(),
      Supplier.TO_NAME);

  MutableList<String> expectedSupplierNames = ...;
  Assert.assertEquals(expectedSupplierNames, supplierNames);
}
```

Solution 5

```
public void countSuppliersWithMoreThanTwoItems()
{
  Predicate<Supplier> moreThanTwoItems =
    Predicates.attributeGreaterThan(
      Supplier.TO_NUMBER_OF_ITEMS,
      2);

  int suppliersWithMoreThanTwoItems =
    ArrayIterate.count(
      this.company.getSuppliers(),
      moreThanTwoItems);

  Assert.assertEquals(5, suppliersWithMoreThanTwoItems);
}
```

Solution 5

```
public class Supplier
{
  ...
  public static final
   Function<Supplier, Integer> TO_NUMBER_OF_ITEMS =
       supplier-> supplier.itemNames.length;
...
}
```

Solution 5

```
public void whoSuppliesSandwichToaster()
{
   Predicate<Supplier> suppliesToaster =
       supplier-> supplier.supplies("sandwich toaster");

   Supplier toasterSupplier = ArrayIterate.detect(
     this.company.getSuppliers(),
     suppliesToaster);

  Assert.assertNotNull("toaster supplier", toasterSupplier);
  Assert.assertEquals("Doxins", toasterSupplier.getName());
}
```

Solution 5

```
public class Supplier
{
   ...
   public boolean supplies(String name)
   {
      return ArrayIterate.contains(this.itemNames, name);
   }
   ...
}
```

Solution 5

```
public void filterOrderValues()
{
   List<Order> orders =
      this.company.getMostRecentCustomer().getOrders();

   MutableList<Double> orderValues =
      ListIterate.collect(orders, Order.TO_VALUE);

   MutableList<Double> filtered =
      orderValues.select(Predicates.greaterThan(1.5));

   Assert.assertEquals(
      FastList.newListWith(372.5, 1.75),
      filtered);
}
```

Solution 5

```
// Given
public static final Function<Order, Double> TO_VALUE =
    order-> order.getValue();

// or
public static final Function<Order, Double> TO_VALUE =
    Order::getValue;

// When Lambdas are available, just inline ;-)
```

Solution 5

```
// Given
public double getValue()
{
  Collection<Double> itemValues = Iterate.collect(
    this.lineItems,
    lineItem -> lineItem.getValue();
  );

  return CollectionAdapter.adapt(itemValues)
    .injectInto(0.0, (x, y) x + y);
});
```

Solution 5

```
public void filterOrders()
{
   List<Order> orders =
     this.company.getMostRecentCustomer().getOrders();

   MutableList<Order> filtered = ListIterate.select(
     orders,
     Predicates.attributeGreaterThan(Order.TO_VALUE,
    2.0));


   Assert.assertEquals(

   FastList.newListWith(Iterate.getFirst(this.company.getMostRecentCustomer().getOrders())),

      filtered);
}
```

Refactoring to GS Collections

Before

```
List<Integer> integers = new ArrayList<Integer>();
integers.add(1);
integers.add(2);
integers.add(3);
```

After

```
List<Integer> integers = new FastList<Integer>();
integers.add(1);
integers.add(2);
integers.add(3);
```

Why?

- `FastList` is a drop-in replacement for `ArrayList`.
- More memory efficient.
- Opens up the refactoring opportunities coming up next.

Before

```
List<Integer> integers = new FastList<Integer>();
integers.add(1);
integers.add(2);
integers.add(3);
```

After

```
List<Integer> integers = FastList.newList();
integers.add(1);
integers.add(2);
integers.add(3);
```

Why?

- The static factory methods can infer generic types.

Before

```
List<Integer> integers = FastList.newList();
integers.add(1);
integers.add(2);
integers.add(3);
```

After

```
List<Integer> integers =
  FastList.newListWith(1, 2, 3);
```

Why?

- Varargs support; any number of arguments.
- Never mutated; so you could make it unmodifiable:

```
FastList.newListWith(1, 2, 3).asUnmodifiable();
```

- There is also a form that takes another iterable:

```
FastList.newList(list);
```

Before

```
List<Integer> integers =
  FastList.newListWith(1, 2, 3);
```

After

```
MutableList<Integer> integers =
  FastList.newListWith(1, 2, 3);
```

Why?

- `MutableList` is a drop in replacement for `List`
- Methods available on interface instead of utility classes.
- Better type information:
  - `Iterate.select()` returns a `Collection`, but
  - `MutableList.select()` returns a `MutableList`

These refactorings are analogous for `UnifiedSet` and `UnifiedMap`:

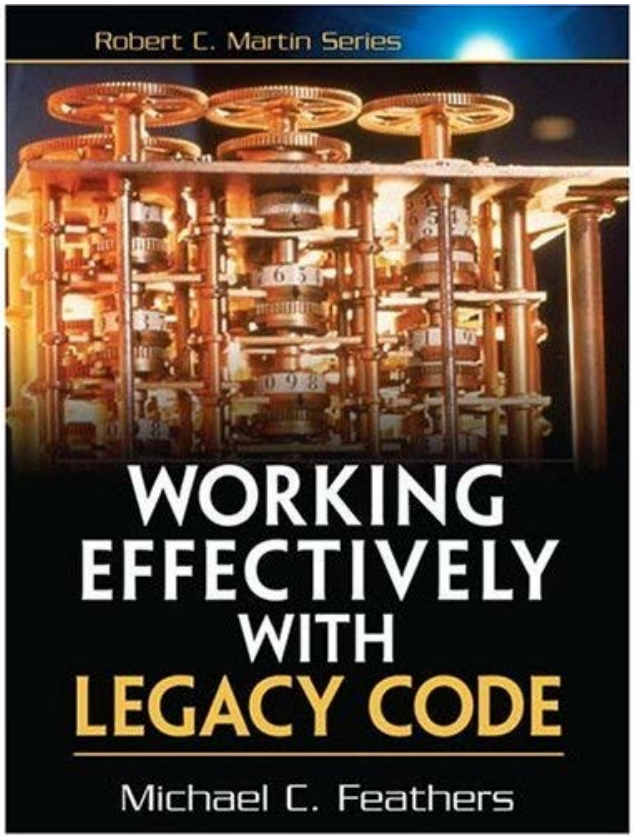
```
UnifiedSet<Integer> set =
  UnifiedSet.newSetWith(1, 2, 3);

UnifiedMap<Integer, String> map =
  UnifiedMap.newWithKeysValues(
    1, "1",
    2, "2",
    3, "3");
```

KATA EXERCISE 6

Exercise 6

- Fix `Exercise6Test`.
- Two of the ones you just solved.
- This time, don't use static utility, refactor the domain instead.
- Exercise 6 should take about 10 minutes.

Solution 6



Quote

"The primary purpose of a compiler is to translate source code into some other form, but in statically typed languages, you can do much more with a compiler. You can take advantage of its type checking and use it to identify changes you need to make. I call this practice ***leaning on the compiler***."

More Iteration Patterns

Patterns seen so far

`select`	Returns the elements of a collection that satisfy the `Predicate`.
`reject`	Returns the elements of a collection that *do not* satisfy the `Predicate`.
`count`	Returns the number of elements that satisfy the `Predicate`.
`collect`	Transforms the elements using a `Function`.
`flatCollect`	Transforms and flattens the elements using a `Function`.

Short-circuit patterns

`detect`	Finds the first element that satisfies the `Predicate`.
`anySatisfy`	Returns true if any element satisfies the `Predicate`.
`allSatisfy`	Returns true if all elements satisfy the `Predicate`.

Additional patterns

`forEach`	Executes the `Procedure` on each element, doesn't return anything.
`injectInto`	Starts with an accumulator and executes a `Function2` (a two-argument function) over each element passing the previous accumulated result.
`chunk`	Splits the collection into chunks of the given size.
`zip`	Joins two collections into one collection of `Pair`s.
`makeString`	Like `toString()`, with a customizable separator, start, and end string.
`toList` / `toSet`	Converts the collection to a new copy of the correct type.
`toSortedList`	Returns a new list sorted according to some `Comparator`.
`sortThis`	Sorts the list in place (mutating method) according to some `Comparator`.
`min` / `max`	Returns the min / max element of a collection according to some `Comparator`.

makeString()

- `makeString()` returns a `String` representation, similar to `toString()`.
- Three forms:
  - `makeString(`*`start, separator, end`*`)`
  - `makeString(`*`separator`*`)` defaults start and end to empty strings
  - `makeString()` defaults the separator to ", " (comma and space)

Code Example

```
MutableList<Integer> list = FastList.newListWith(1, 2, 3);
assertEquals("[1/2/3]", list.makeString("[", "/", "]"));
assertEquals("1/2/3", list.makeString("/"));
assertEquals("1, 2, 3", list.makeString());
assertEquals(
  list.toString(),
  list.makeString("[", ", ", "]"));
```

appendString()

- `appendString()` is similar to `makeString()`, but it appends to an `Appendable` and is `void`.
  - Common `Appendable`s are `StringBuilder`, `PrintStream`, `BufferedWriter`, etc.
  - Same three forms, with additional first argument, the `Appendable`.

Code Example

```
MutableList<Integer> list = FastList.newListWith(1, 2, 3);
Appendable appendable = new StringBuilder();
list.appendString(appendable, "[", "/", "]");
assertEquals("[1/2/3]", appendable.toString());
```

chunk()

- chunk() splits a `RichIterable` into fixed size pieces.
- Final chunk will be smaller if the size doesn't divide evenly.

Code Example

```
MutableList<Integer> list =
  FastList.newListWith(1, 2, 3, 4, 5, 6, 7, 8, 9, 10);
RichIterable<RichIterable<Integer>> chunks =
   list.chunk(4);
System.out.println(chunks);
// prints [[1, 2, 3, 4], [5, 6, 7, 8], [9, 10]]
```

zip()

- `zip()` takes a second `RichIterable` and pairs up all the elements.
- If one of the two `RichIterable`s is longer, its extra elements are ignored.

Code Example

```
MutableList<String> list1 =
   FastList.newListWith("One", "Two", "Three",
   "Truncated");
MutableList<String> list2 =
   FastList.newListWith("Four", "Five", "Six");
MutableList<Pair<String, String>> pairs =
   list1.zip(list2);
System.out.println(pairs);
// prints [One:Four, Two:Five, Three:Six]
```

zipWithIndex()

- A special case is when you want to `zip()` the elements in a collection with their index positions.
- You could accomplish that with `zip()` and `Interval`, or use `zipWithIndex()` .

Code Example

```
MutableList<String> list =
  FastList.newListWith("One", "Two", "Three");
MutableList<Pair<String, Integer>> pairs =
  list.zipWithIndex();
System.out.println(pairs);
// prints [One:0, Two:1, Three:2]
```

Iteration Pattern

- `min()` and `max()` take a `Comparator` and return the extreme elements.
- Overloads don't take a `Comparator`.
  - If the elements are not `Comparable`, you get a `ClassCastException`.
- What if you don't want the maximum age, but instead the oldest `Person`?

Code Example

```
MutableList<Person> people = ...;
Integer maxAge = people.collect(Person.TO_AGE).max();
```

Iteration Pattern

- `min()` and `max()` take a `Comparator` and return the extreme elements.
- Overloads don't take a `Comparator`.
  - If the elements are not `Comparable`, you get a `ClassCastException`.
- What if you don't want the maximum age, but instead the oldest `Person`?
  - Use `minBy()` and `maxBy()` instead.

Code Example

```
MutableList<Person> people = ...;
Integer maxAge = people.collect(Person.TO_AGE).max();
Person oldestPerson = people.maxBy(Person.TO_AGE);
```

Iteration Pattern

- `toSortedList()` takes a `Comparator` and returns a new sorted list.
- Overload doesn't take a `Comparator`.
  - If the elements are not `Comparable`, you get a `ClassCastException`.
- What if you don't want to sort the ages, but instead sort the people by age?
  - Use `sortThisBy()` instead.

Code Example

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

MutableList<Integer> sortedAges =
  people.collect(Person.TO_AGE).toSortedList();

MutableList<Person> peopleSortedByAge =
  people.toSortedListBy(Person.TO_AGE);
```

Kata Exercise 7

Exercise 7

- Fix `Exercise7Test`.
- Exercises use some of the iteration patterns you have just learned.
- Some use a combination of iteration patterns you have already seen.
- Exercise 7 should take about 20 minutes.

Solution 7

```
public void sortedTotalOrderValue()
{
  MutableList<Double> sortedTotalValues =
    this.company.getCustomers()
      .collect(Customer.TO_TOTAL_ORDER_VALUE)
      .toSortedList();

  Assert.assertEquals(Double.valueOf(857.0),
   sortedTotalValues.getLast());
  Assert.assertEquals(Double.valueOf(71.0),
   sortedTotalValues.getFirst());
}
```

Solution 7

```
public void maximumTotalOrderValue()
{
  Double maximumTotalOrderValue =
    this.company.getCustomers()
      .collect(Customer.TO_TOTAL_ORDER_VALUE)
      .max();

  Assert.assertEquals(Double.valueOf(857.0), maximumTotalOrderValue);
}
```

Solution 7

```
public void customerWithMaxTotalOrderValue()
{
   Customer customerWithMaxTotalOrderValue =
      this.company.getCustomers()
         .maxBy(Customer.TO_TOTAL_ORDER_VALUE);

   Assert.assertEquals(
      this.company.getCustomerNamed("Mary"),
      customerWithMaxTotalOrderValue);
}
```

Solution 7

```
public void supplierNamesAsTildeDelimitedString()
{
  MutableList<String> supplierNames =
   ArrayIterate.collect(
     this.company.getSuppliers(),
     Supplier.TO_NAME);

  String tildeSeparatedNames =
   supplierNames.makeString("~");

  Assert.assertEquals(
    "Shedtastic~Splendid Crocks~Annoying Pets~Gnomes 'R' Us~Furniture
   Hamlet~SFD~Doxins",
    tildeSeparatedNames);
}
```

Solution 7

```
public void deliverOrdersToLondon()
{
  this.company.getCustomers()
    .select(Predicates.attributeEqual(
      Customer.TO_CITY,
      "London"))
    .flatCollect(Customer.TO_ORDERS)
    .forEach(Order.DELIVER);

  Verify.assertAllSatisfy(
    this.company.getCustomerNamed("Fred").getOrders(),
    Order.IS_DELIVERED);
  Verify.assertAllSatisfy(
    this.company.getCustomerNamed("Mary").getOrders(),
    Predicates.not(Order.IS_DELIVERED));
  Verify.assertAllSatisfy(
    this.company.getCustomerNamed("Bill").getOrders(),
    Order.IS_DELIVERED);
}
```

Stack

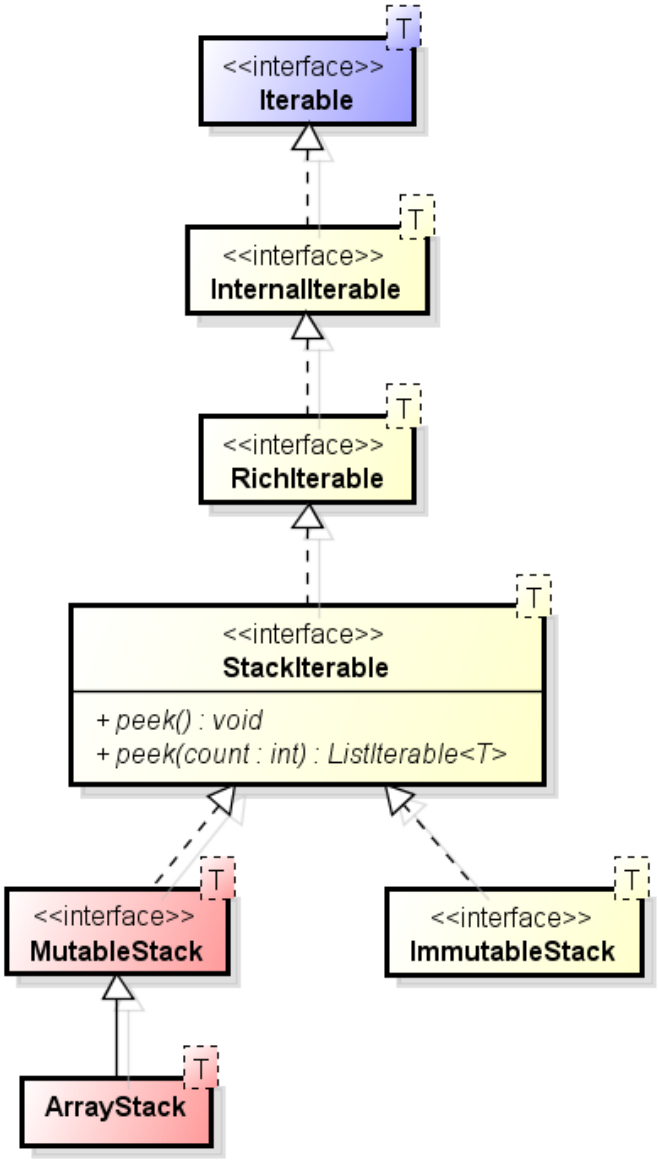
Stack

- `java.util.Stack` extends `Vector`
- How does `java.util.Stack` iterate?

JCF Problems

```
java.util.Stack stack = new java.util.Stack();
stack.push(1);
stack.push(2);
stack.push(3);
System.out.println(stack);
// Prints [1, 2, 3]
```

Inheritance Hierarchy



Stacks

- **`ArrayStack`** is not a drop-in replacement for **`java.util.Stack`**.
- **`MutableStack`** does not extend **`Collection`**.

push()

push() adds an item to the top of the MutableStack

```
MutableStack<Integer> stack =
  ArrayStack.newStackWith(1, 2, 3);
System.out.println(stack);
// Prints [3, 2, 1]
stack.push(4);
System.out.println(stack);
// Prints [4, 3, 2, 1]
```

The different ways to create a `MutableStack`

```
System.out.println(ArrayStack.newStackWith(1, 2, 3));
// Prints [3, 2, 1]

System.out.println(
  ArrayStack.newStackFromTopToBottom(1, 2, 3));
// Prints [1, 2, 3]

System.out.println(
  ArrayStack.newStack(FastList.newListWith(1, 2, 3)));
// Prints [3, 2, 1]

System.out.println(
  ArrayStack.newStackFromTopToBottom(
    FastList.newListWith(1, 2, 3)));
// Prints [1, 2, 3]
```

pop()

Overloaded pop( ) methods:

```
pop()
pop(int count)
pop(int count, R targetCollection)
```

Code Example

```
ArrayStack<Integer> stack1 = ArrayStack.newStackWith(1, 2, 3);
Assert.assertEquals(
  FastList.newListWith(3, 2),
  stack1.pop(2));

ArrayStack<Integer> stack2 = ArrayStack.newStackWith(1, 3, 3);
Assert.assertEquals(
  UnifiedSet.newSetWith(3),
  stack2.pop(2, UnifiedSet.<Integer>newSet()));

ArrayStack<Integer> stack3 = ArrayStack.newStackWith(1, 2, 3);
Assert.assertEquals(
  ArrayStack.newStackWith(3, 2),
  stack3.pop(2, ArrayStack.<Integer>newStack()));
```

peek(), peek(int count)

MutableStack has an overloaded peek() method that returns a ListIterable

Code Example

```
MutableStack<Integer> stack =
  ArrayStack.newStackWith(1, 2, 3);

Assert.assertEquals(
  Integer.valueOf(3),
  stack.peek());

Assert.assertEquals(
  FastList.newListWith(3, 2),
  stack.peek(2));
```

MutableStack

`MutableStack` does **not** extend `java.util.List` (or `Collection`)

JCF Problems

```
java.util.Stack stack = new java.util.Stack();
stack.push(1);
stack.push(2);
stack.push(3);
Assert.assertEquals(FastList.newListWith(1, 2, 3), stack);
stack.add(2, 4);
Assert.assertEquals(FastList.newListWith(1, 2, 4, 3), stack);
stack.remove(1);
Assert.assertEquals(FastList.newListWith(1, 4, 3), stack);
```

Stack API

Methods	Inherited from
select(), collect(), etc.	RichIterable
peek()	StackIterable
push(), pop()	MutableStack

Code Example

```
StackIterable<Integer> stack =
  ArrayStack.newStackFromTopToBottom(1, 2, 3, 4, 5);
StackIterable<Integer> evens = stack.select(integer ->
{
  System.out.print(integer + " ");
  integer % 2 == 0
});
// Prints 1 2 3 4 5
Assert.assertEquals(ArrayStack.newStackFromTopToBottom(2, 4), evens);
```

Bag

New Type

- Useful when you would otherwise use `Map<K, Integer>`
  - For example, find the number of people who live in each state

Code Example

```
MutableList<Person> people = ...;
MutableList<String> usStates =
   people.collect(US_STATE_FUNCTION);
MutableMap<String, Integer> stateCounts =
   UnifiedMap.newMap();
...

int newYorkers = stateCounts.get("NY");
```

New Type

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

Map Example

```
MutableMap<String, Integer> stateCounts =
   UnifiedMap.newMap();
for (String state : usStates) {
  Integer count = stateCounts.get(state);
  if (count == null) {
    count = 0;
  }
  stateCounts.put(state, count + 1);
}
```

Before

```
MutableMap<String, Integer> stateCounts =
   UnifiedMap.newMap();
for (String state : usStates) {
  Integer count = stateCounts.get(state);
  if (count == null) {
    count = 0;
  }
  stateCounts.put(state, count + 1);
}
```

After

```
MutableBag<String> stateCounts = HashBag.newBag();
for (String state : usStates) {
  stateCounts.add(state);
}

int newYorkers = stateCounts.occurrencesOf("NY");
```

Before

```
MutableBag<String> stateCounts = HashBag.newBag();
for (String state : usStates) {
  stateCounts.add(state);
 }

int newYorkers = stateCounts.occurrencesOf("NY");
```

After

```
MutableBag<String> stateCounts = usStates.toBag();

int newYorkers = stateCounts.occurrencesOf("NY");
```

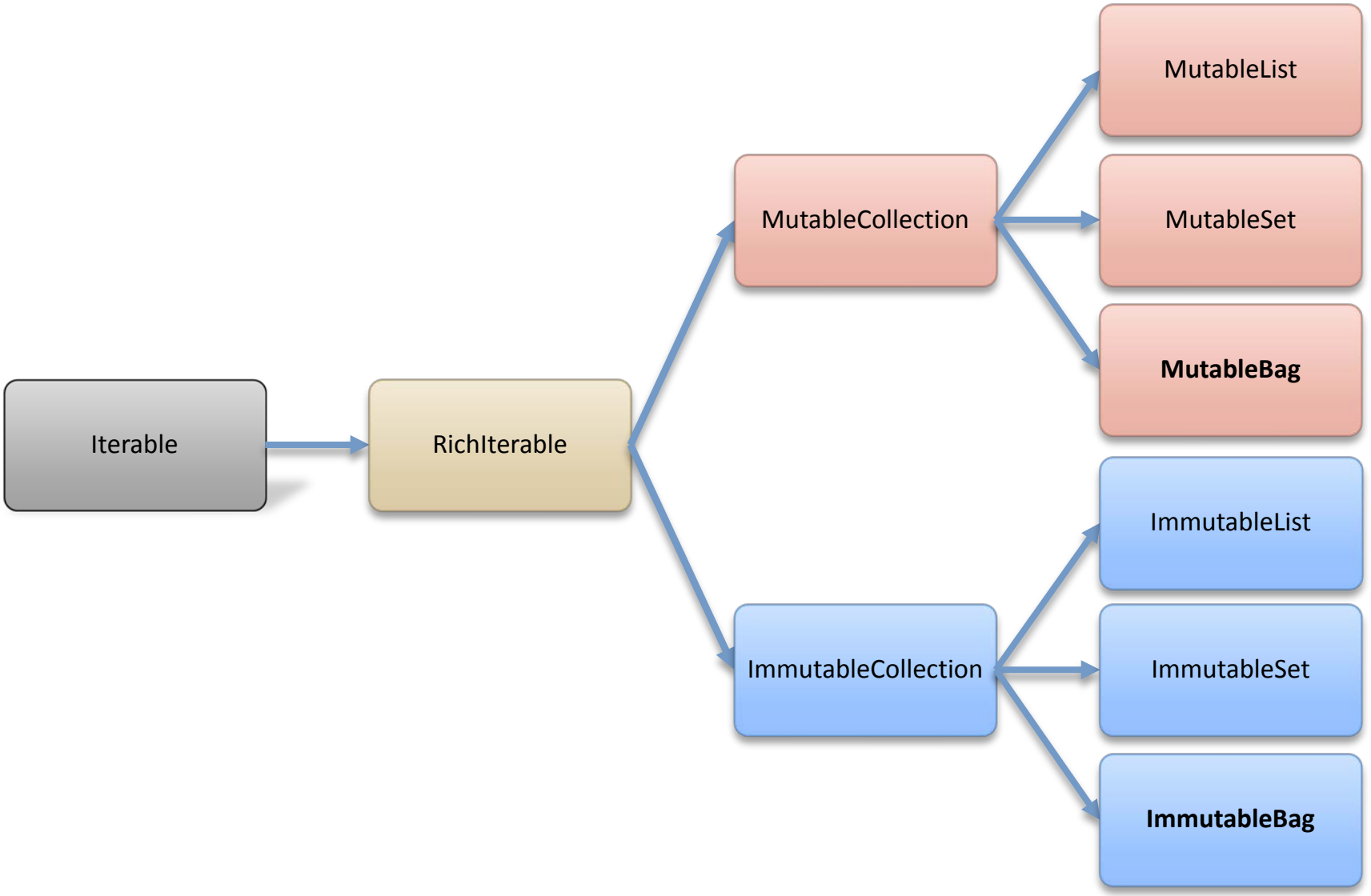
New Type

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

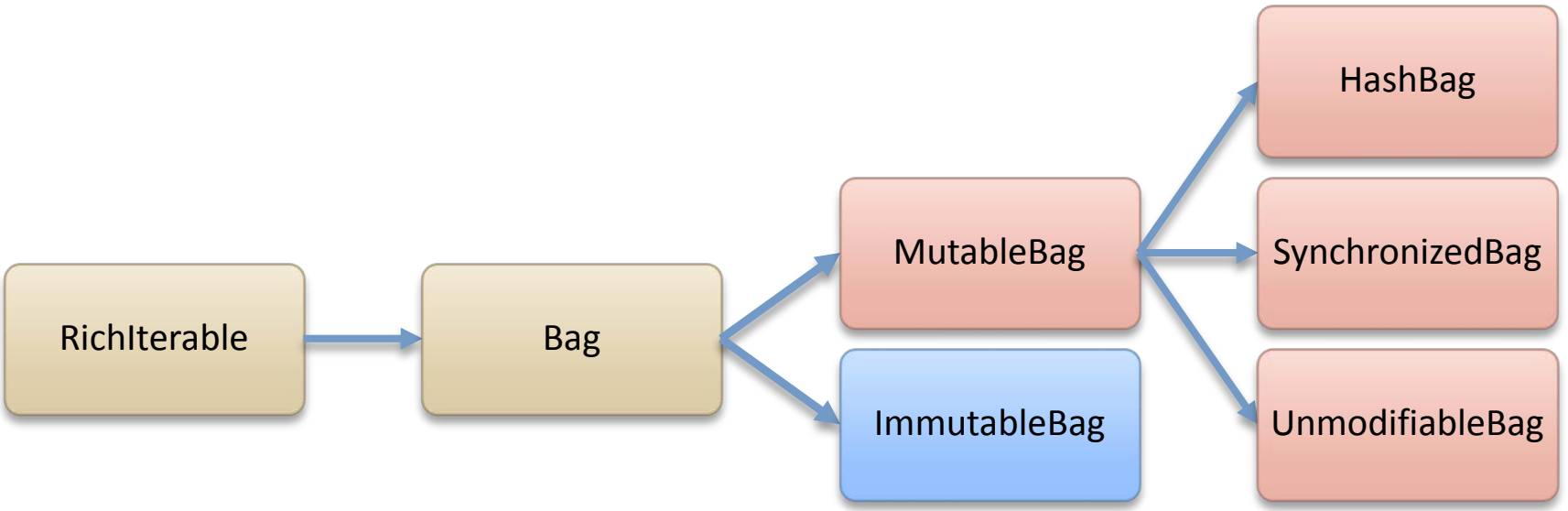
Bag Example

```
MutableBag<String> stateCounts = usStates.toBag();

int newYorkers = stateCounts.occurrencesOf("NY");
```

Class Diagram



Class Diagram



Bag API

Methods	Inherited from
select(), collect(), etc.	RichIterable
add(), remove(), iterator(), etc.	MutableCollection (java.util.Collection)
occurrencesOf(), forEachWithOccurrences(), toMapOfItemToCount()	Bag
addOccurrences(), removeOccurrences()	MutableBag

Code Example

```
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"));
```

MULTIMAP

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 live in each state.

Code Example

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

...

MutableList<Person> newYorkers =
   statesToPeople.get("NY");
```

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 live in each state.
  - Lots of boilerplate code to deal with uninitialized backing collections.

Map Example

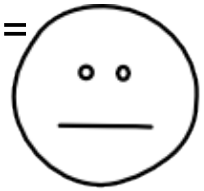
```
MutableMap<String, MutableList<Person>> statesToPeople =
    UnifiedMap.newMap();
for (Person person : people) {
  String state = US_STATE_FUNCTION.valueOf(person);
  MutableList<Person> peopleInState = statesToPeople.get(state);
  if (peopleInState == null) {
    peopleInState = FastList.newList();
    statesToPeople.put(state, peopleInState);
  }
  peopleInState.add(person);
}
```

Before

```
MutableMap<String, MutableList<Person>> statesToPeople =
    UnifiedMap.newMap();
for (Person person : people) {
  String state = US_STATE_FUNCTION.valueOf(person);
  MutableList<Person> peopleInState = statesToPeople.get(state);
  if (peopleInState == null) {
    peopleInState = FastList.newList();
    statesToPeople.put(state, peopleInState);
  }
  peopleInState.add(person);
}
```

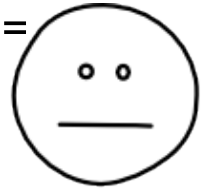
After

```
MutableListMultimap<String, Person> statesToPeople =
  FastListMultimap.newMultimap();
for (Person person : people) {
  String state = US_STATE_FUNCTION.valueOf(person);
  statesToPeople.put(state, person);
}
MutableList<Person> newYorkers =
  statesToPeople.get("NY");
```



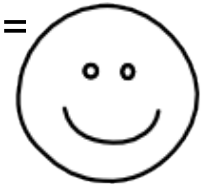
Before

```
MutableListMultimap<String, Person> statesToPeople =
  FastListMultimap.newMultimap();
for (Person person : people) {
  String state = US_STATE_FUNCTION.valueOf(person);
  statesToPeople.put(state, person);
}
MutableList<Person> newYorkers =
   statesToPeople.get("NY");
```



After

```
MutableListMultimap<String, Person> statesToPeople =
  people.groupBy(US_STATE_FUNCTION);

MutableList<Person> newYorkers =
   statesToPeople.get("NY");
```



Multimap

- What happens if you add the same key and value twice?

Code Example

```
MutableMultimap<String, Person> multimap = ...;

multimap.put("NY", person);
multimap.put("NY", person);
RichIterable<Person> ny = multimap.get("NY");

Verify.assertIterableSize(?, ny);
```

Multimap

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

Code Example

```
MutableListMultimap<String, Person> multimap =
  FastListMultimap.newMultimap();
multimap.put("NY", person);
multimap.put("NY", person);
MutableList<Person> ny = multimap.get("NY");

Verify.assertIterableSize(2, ny);
```

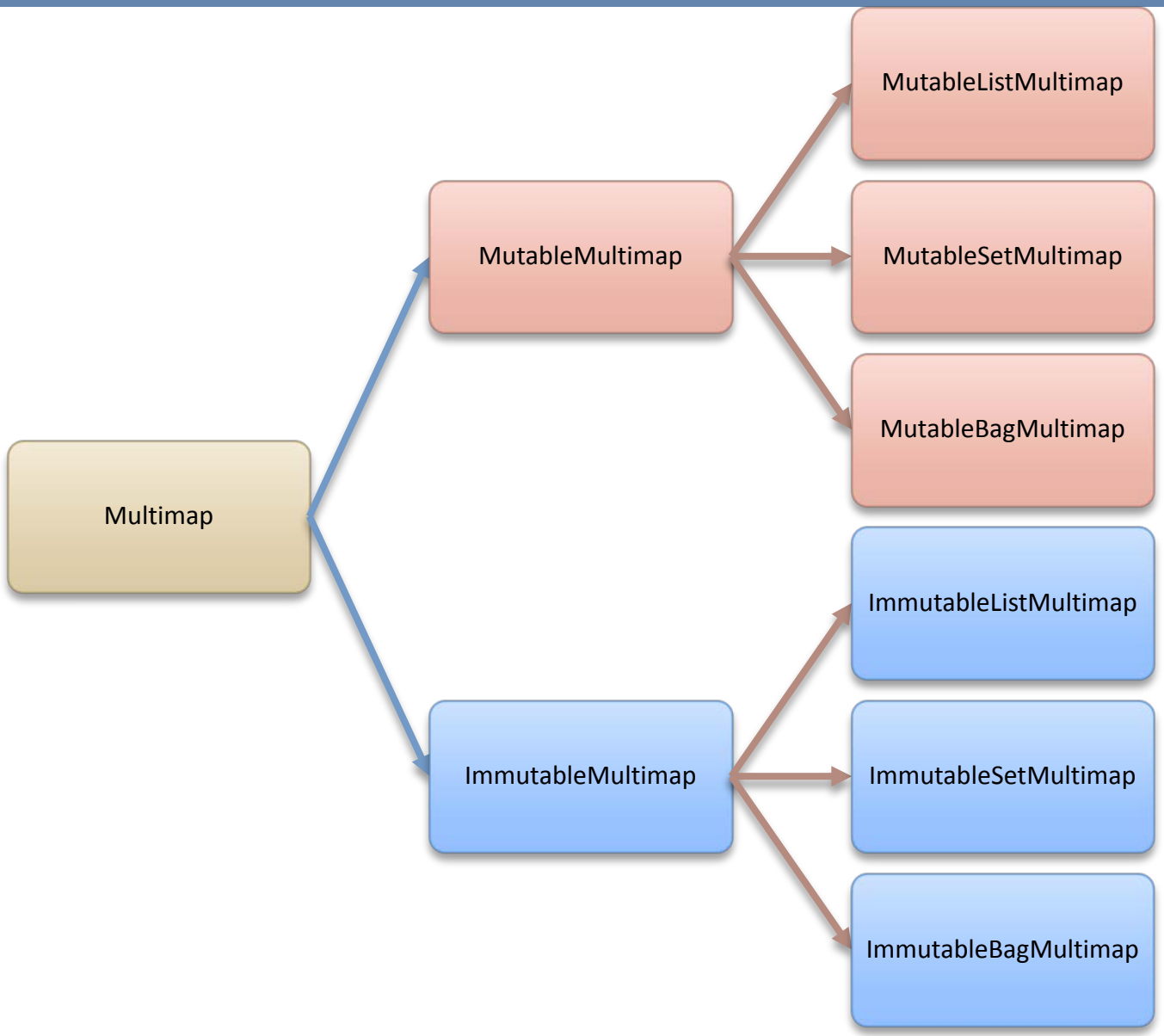
Multimap

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

Code Example

```
MutableSetMultimap<String, Person> multimap =
  UnifiedSetMultimap.newMultimap();
multimap.put("NY", person);
multimap.put("NY", person);
MutableSet<Person> ny = multimap.get("NY");

Verify.assertIterableSize(1, ny);
```

Class Diagram



Collection Types

- List
- Set
- Map
- Stack
- Bag
- Multimap

Mutability

- Mutable
- Immutable

Decorators

- Synchronized
- Unmodifiable

KATA EXERCISE 8

Exercise 8

- Fix `Exercise8Test`.
- Refactor the repetition at TODO 8 in `CompanyDomainForKata` without breaking anything.
- Exercise 8 should take about 30 minutes.

Solution 8

```
public void customersByCity()
{
  MutableListMultimap<String, Customer> multimap =
    this.company.getCustomers().groupBy(Customer.TO_CITY);

   ...
}
```

Solution 8

```
public void mapOfItemsToSuppliers()
{
  final MutableMultimap<String, Supplier> itemsToSuppliers =
    FastListMultimap.newMultimap();
  ArrayIterate.forEach(
    this.company.getSuppliers(),
    supplier ->
        ArrayIterate.forEach(
          supplier.getItemNames(),
          itemName -> itemsToSuppliers.put(itemName, supplier);
        );
  );
  Verify.assertIterableSize(2, itemsToSuppliers.get("sofa"));
}
```

Solution 8

```
public class Order
{
  private final MutableBag<LineItem> lineItems =
    HashBag.newBag();

  ...

  public void addLineItems(LineItem aLineItem, int
   occurrences)
  {
    this.lineItems.addOccurrences(aLineItem, occurrences);
  }

  ...

}
```

Solution 8

```
private void setUpCustomersAndOrders()
{
   ...
   order.addLineItems(new LineItem("cup", 1.5), 3);
   ...
}
```

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().

Refactor Exercise8Test.mapOfItemsToSuppliers() to use groupByEach().

Code Example

```
MutableListMultimap<String, Person> statesToPeople =
  people.groupBy(US_STATE_FUNCTION);

MutableListMultimap<String, Person> statesToPeople =
  people.groupByEach(US_STATES_FUNCTION);
```

Refactor `Exercise8Test.mapOfItemsToSuppliers()` to use `groupByEach()`.

Use an `ArrayAdapter`.

```
public void mapOfItemsToSuppliers_refactored()
{
  MutableList<Supplier> suppliersList =
    ArrayAdapter.adapt(this.company.getSuppliers());

  ...
}
```

Solution 8

```
public void mapOfItemsToSuppliers_refactored()
{
  MutableList<Supplier> suppliersList =
    ArrayAdapter.adapt(this.company.getSuppliers());

  MutableListMultimap<String, Supplier> itemsToSuppliers =
    suppliersList.groupByEach(Supplier.TO_ITEM_NAMES);

  Verify.assertIterableSize(2, itemsToSuppliers.get("sofa"));
}
```

Lazy Evaluation

Eager Evaluation

- This example uses **eager evaluation**.
- When do the calls to `valueOf()` and `accept()` take place?
- We can create our own `Function` and `Predicate` to answer the question.

Code Example

```
Person person1 = new Person(address1);
Person person2 = new Person(address2);
Person person3 = new Person(address3);
MutableList<Person> people =
   FastList.newListWith(person1, person2, person3);
MutableList<Address> addresses =
  people.collect(Person::getAddress);
Assert.assertTrue(addresses.anySatisfy(address2::equals));
```

Eager Evaluation

- `Function` from `Person` to `Address`.
- Maintains internal **mutable** state.
  - Not functional style.
  - Not thread-safe.

Code Example

```
public class AddressFunction
  implements Function<Person, Address> {
    private int counter = 1;

    public Address valueOf(Person person) {
      System.out.println("Function: " + this.counter);
      this.counter++;
      return person.getAddress();
    }
}
```

Eager Evaluation

- `Predicate` returns true when `address` is the same reference as `this.address`
- Maintains internal **mutable** state.
  - Not functional style.
  - Not thread-safe.

Code Example

```
public class EqualsAddressPredicate implements Predicate<Address> {
  private final Address address;
  private int counter = 1;

  private EqualsAddressPredicate(Address address) {
    this.address = address;
  }

  public boolean accept(Address address) {
    System.out.println("Predicate: " + this.counter);
    this.counter++;
    return address == this.address;
  }
}
```

Eager Evaluation

- When do the calls to `valueOf()` and `accept()` take place?

Code Example

```
MutableList<Address> addresses =
  people.collect(new AddressFunction());

addresses.anySatisfy(
  new EqualsAddressPredicate(address2));
```

Eager Evaluation

- When do the calls to `valueOf()` and `accept()` take place?

Code Example

```
MutableList<Address> addresses =
  people.collect(new AddressFunction());
// Function: 1
// Function: 2
// Function: 3

addresses.anySatisfy(
  new EqualsAddressPredicate(address2));
// Predicate: 1
// Predicate: 2
```

Definition

- According to Wikipedia, **lazy evaluation** is "the technique of delaying a computation until its value is actually required."
- When do the calls to `valueOf()` and `accept()` take place?

Code Example

```
LazyIterable<Person> peopleLazy = people.asLazy();
LazyIterable<Address> addressesLazy =
  peopleLazy.collect(new AddressFunction());
addressesLazy.anySatisfy(
  new EqualsAddressPredicate(address2));
```

Definition

- According to Wikipedia, **lazy evaluation** is "the technique of delaying a computation until its value is actually required."
- When do the calls to `valueOf()` and `accept()` take place?

Code Example

```
LazyIterable<Person> peopleLazy = people.asLazy();
LazyIterable<Address> addressesLazy =
  peopleLazy.collect(new AddressFunction());
addressesLazy.anySatisfy(
  new EqualsAddressPredicate(address2));
// Function: 1
// Predicate: 1
// Function: 2
// Predicate: 2
```

Eager Evaluation

```
MutableList<Address> addresses = people.collect(new AddressFunction());
// Function: 1
// Function: 2
// Function: 3

addresses.anySatisfy(new EqualsAddressPredicate(address2));
// Predicate: 1
// Predicate: 2
```

Lazy Evaluation

Why would we prefer lazy evaluation?

```
LazyIterable<Person> peopleLazy = people.asLazy();

LazyIterable<Address> addressesLazy =
  peopleLazy.collect(new AddressFunction());

addressesLazy.anySatisfy(new EqualsAddressPredicate(address2));
// Function: 1
// Predicate: 1
// Function: 2
// Predicate: 2
```

Definition

- According to Wikipedia, **lazy evaluation** is “the technique of delaying a computation until its value is actually required.”
- Benefits include:
  - Performance increases due to avoiding unnecessary calculations.
  - Avoiding error conditions in the evaluation of compound expressions.
  - The ability to construct potentially infinite data structures.

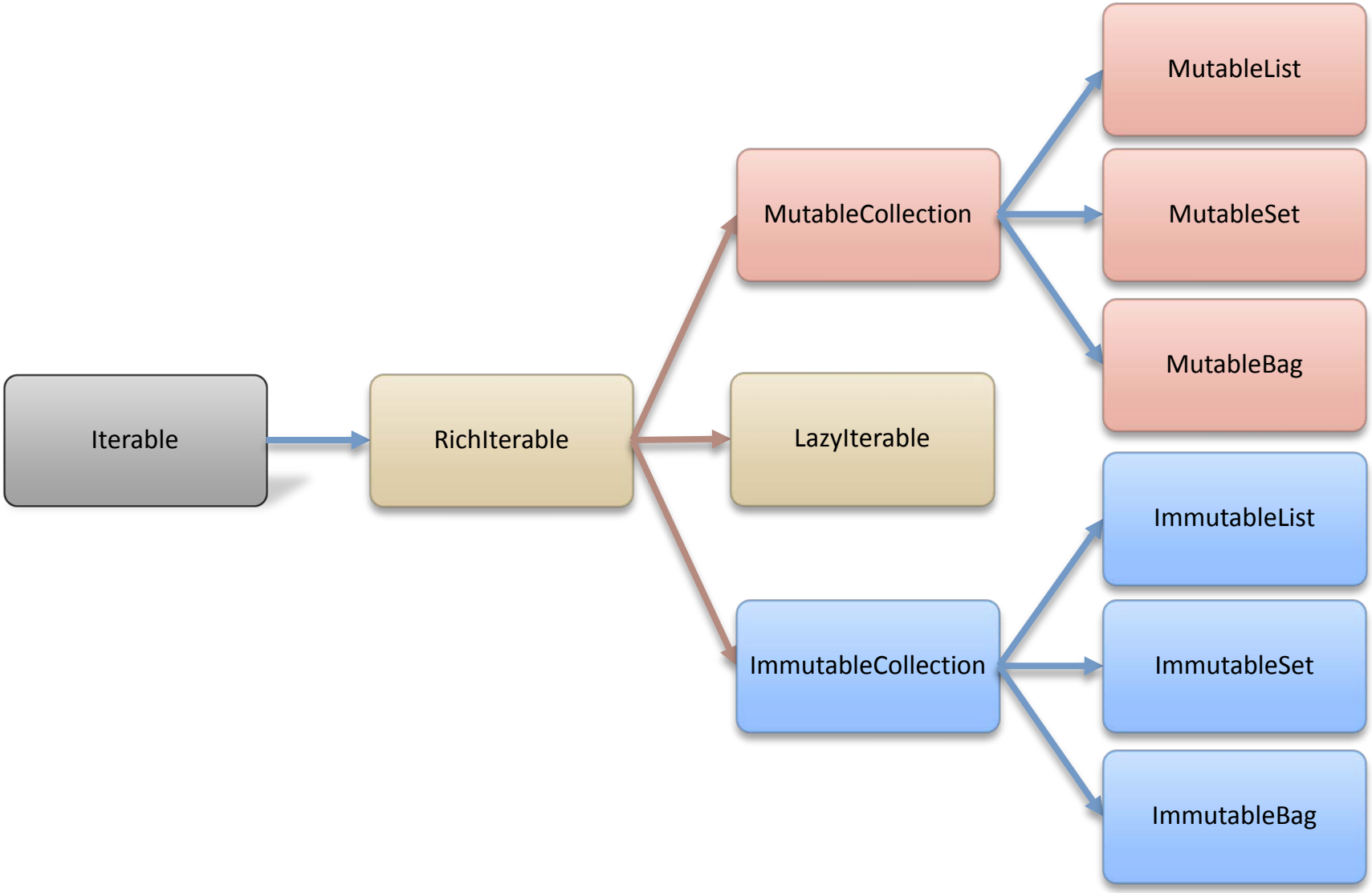
Avoid Error Conditions

- Test passes.
- No `NullPointerException`.

Code Example

```
MutableList<Person> people =
  FastList.newListWith(person1, person2, null);
LazyIterable<Person> peopleLazy = people.asLazy();
LazyIterable<Address> addressesLazy =
   peopleLazy.collect(new AddressFunction());
Assert.assertTrue(addressesLazy.anySatisfy(
  new EqualsAddressPredicate(address2)));
```

Class Diagram



Lazy Evaluation

- `LazyIterate` provides the utility methods for lazy evaluation.

Code Example

```
MutableList<Person> people =
  FastList.newListWith(person1, person2, null);

LazyIterable<Address> addresses =
  LazyIterate.collect(people, Person::getAddress);

Assert.assertTrue(
  addresses.anySatisfy(address2::equals));
```

Parallel Lazy Evaluation

Parallel Lazy Evaluation

- `asLazy` returns `LazyIterable`
- `asParallel` returns `ParallelIterable`
- API is similar to lazy-serial, and lazy methods return `ParallelIterable`
- `asParallel` takes an `ExecutorService` and a `batchSize`
- When evaluation is forced, the backing collections is divided into batches which are processed in parallel in the `ExecutorService`

Code Example

```
int numCores = Runtime.getRuntime().availableProcessors();
ExecutorService executorService =
  Executors.newFixedThreadPool(numCores);
ParallelListIterable<Person> peopleLazy =
  people.asParallel(executorService, 2);
ParallelListIterable<Address> addressesLazy =
  peopleLazy.collect(Person::getAddress);
Assert.assertTrue(addressesLazy.anySatisfy(address2::equals));
executorService.shutdownNow();
```

Cancellation

- It's possible to cancel a parallel-lazy computation in progress
- Just shut down the `ExecutorService`
- Batches in progress won't halt but new batches won't start
- Means you can't share the thread pools among multiple computations

- In the code example, `anySatisfy` will throw a `RuntimeException`

Code Example

```
// In one thread
addressesLazy.anySatisfy(address2::equals);
// In another thread
executorService.shutdownNow();
```

Unmodifiable and Synchronized Wrappers

Unmodifiable Wrapper

- `asUnmodifiable()` returns a wrapper which throws on mutating methods.

Test Code

```
Verify.assertThrows(
  UnsupportedOperationException.class,
  () -> richIterable.asUnmodifiable().add(0);
);
```

Java Collections Framework

```
Collection<Integer> synch =
  Collections.synchronizedCollection(collection);
synchronized (synch) {
  for (Integer integer : synch) {
    System.out.println(integer);
  }
}
```

GS Collections

```
MutableCollection<Integer> synch =
  collection.asSynchronized();

synch.forEach(integer -> System.out.println(integer));
```

More Benefits of the OO API

JCF For-Each

- Assume that `synchronizedList` is shared by several threads.
- What's wrong with this code?

Code Example

```
List<Integer> synchronizedList =
  Collections.synchronizedList(list);
printAll(synchronizedList);

<T> void printAll(List<T> list) {
  for (T element : list) {
    System.out.println(element);
  }
}
```

JCF For-Each

- For-Each loop syntax gets compiled to bytecode that uses an iterator.
- This code produces identical bytecode.

```
Iterator<T> iterator = list.iterator();
while (iterator.hasNext()) {
  T element = iterator.next();
  System.out.println(element);
}
```

Code Example

```
List<Integer> synchronizedList =
  Collections.synchronizedList(list);
printAll(synchronizedList);

<T> void printAll(List<T> list) {
  for (T element : list) {
    System.out.println(element);
  }
}
```

- `iterator()` is the one method that is not synchronized
- From the JavaDoc of `Collections.synchronizedList()`:
  - It is imperative that the user manually synchronize on the returned list when iterating over it:

```
List list = Collections.synchronizedList(new
 ArrayList());
...
synchronized (list) {
  Iterator i = list.iterator();
  // Must be in synchronized block
  while (i.hasNext())
    foo(i.next());
  }
}
```

Failure to follow this advice may result in non-deterministic behavior.

JCF For-Each

- Using `MutableList` does not help.
- It is not possible to use `Iterator` in a thread-safe way.
- How can we fix this code?

Code Example

```
MutableList<Integer> synchronizedList =
  list.asSynchronized();
this.printAll(synchronizedList);

<T> void printAll(List<T> list) {
  for (T element : list) {
    System.out.println(element);
  }
}
```

JCF For-Each

- We could put a synchronized block inside the `printAll()` method.
- Very strange, since the list might not be synchronized.
- We would have to do this in every method that takes a collection.

Code Example

```
<T> void printAll(List<T> list) {
  synchronized (list) {
    for (T element : list) {
      System.out.println(element);
    }
  }
}
```

Object Oriented

- The `forEach()` method is the safe way.
- The `forEach()` method is the object-oriented way.
- Why does this work?

Code Example

```
<T> void printAll(MutableList<T> list) {
  list.forEach(element -> System.out.println(element););
}
```

Object Oriented

- `SynchronizedMutableList` holds the lock for the duration of the iteration.
- This is the compelling reason to use the `forEach()` method despite the verbosity.

Code Example

```
public void forEach(Procedure<? super E> block) {
  synchronized (this.lock) {
    this.collection.forEach(block);
  }
}
```

Object Oriented

- The code does the correct thing for a:
  - `FastList`
  - `FastList` in a `SynchronizedMutableList`
  - `FastList` in a `SynchronizedMutableList` in a `ListAdapter` in a …
- Even if `FastList.forEach()` is implemented by using an `Iterator`.

Code Example

```
<T> void printAll(MutableList<T> list) {
  list.forEach(element -> System.out.println(element));
}
```

Thread-safe Collections

- `MultiReaderFastList` and `MultiReaderUnifiedSet` completely encapsulate synchronization.
- `iterator()` and `listIterator()` throw `UnsupportedOperationException`.
- `withReadLockAndDelegate()` and `withWriteLockAndDelegate()` allow complete access to the backing collection in a synchronized context.

Code Example

```
MultiReaderFastList<String> list =
MultiReaderFastList.newListWith("1", "2", "3");
list.withWriteLockAndDelegate(backingList -> {
      Iterator<String> iterator = backingList.iterator();
      iterator.next();
      iterator.remove();
});
Assert.assertEquals(FastList.newListWith("2", "3"), list);
```

KATA EXERCISE 9

Exercise 9

- Fix `Exercise9Test`.
- The final set of exercises is the most **difficult** and is **optional**.

Solution 9

```
public void whoOrderedSaucers()
{
  MutableList<Customer> customersWithSaucers =
    this.company.getCustomers().select(
      Predicates.attributeAnySatisfy(
        Customer.TO_ORDERS,
        Predicates.attributeAnySatisfy(
          Order.TO_LINE_ITEMS,
          Predicates.attributeEqual(LineItem.TO_NAME,
   "saucer"))));
  Verify.assertSize(2, customersWithSaucers);
}
```

Solution 9

```
public void mostExpensiveItem() {
  MutableListMultimap<Double, Customer> multimap =
    this.company.getCustomers().groupBy(customer ->
        customer.getOrders()
            .asLazy()
            .flatCollect(Order.TO_LINE_ITEMS)
            .collect(LineItem.TO_VALUE)
            .max();
  );
}
```

Appendix

Anonymous Inner Class

Definition

- 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

```
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

Closure Definition

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 will have **lambdas**.

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.

Code Example

```
public Customer getCustomerNamed(String name)
{
  Predicate<Customer> customerNamed =
    Predicates.attributeEqual(Customer.TO_NAME, name);
  return this.customers.detect(customerNamed);
}
```

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.

- Changing this copy of name has no effect on the result.
- Maybe obvious because of Java's rules for parameter passing.

Code Example

```
public Customer getCustomerNamed(String name)
{
  Predicate<Customer> customerNamed =
    Predicates.attributeEqual(Customer.TO_NAME, name);
  name = name + "!";
  return this.customers.detect(customerNamed);
}
```

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.

Code Example

```
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.

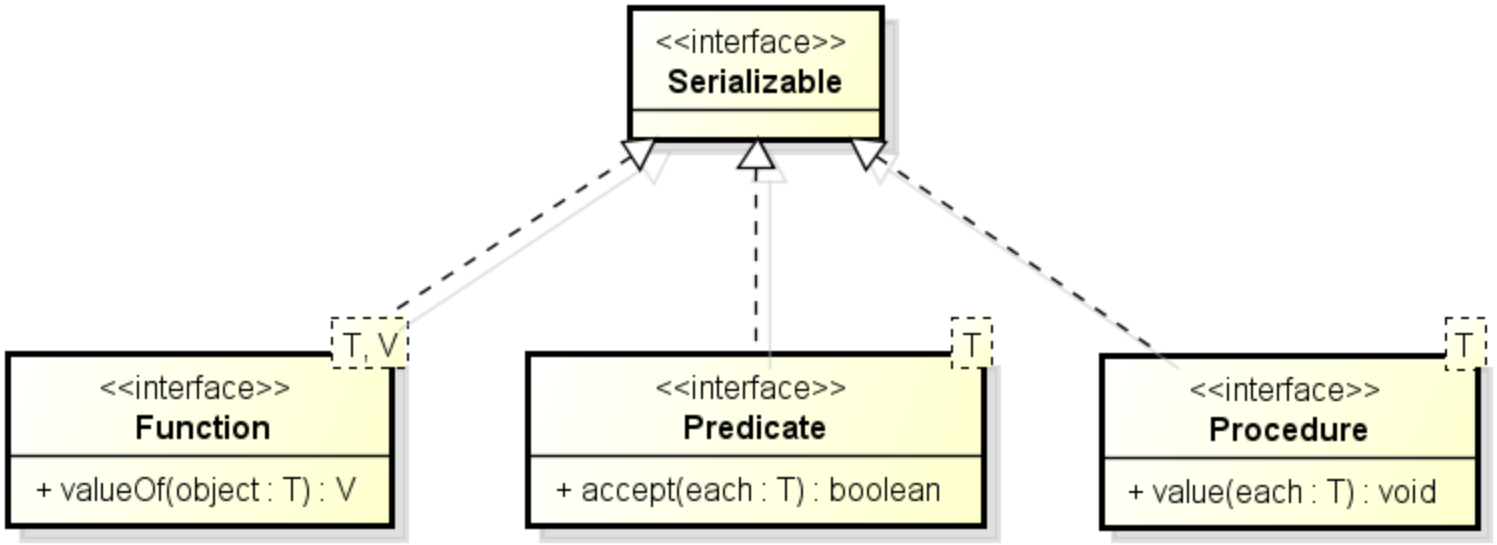
It is **not** the same copy as the method parameter.

- 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

```
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);
  }
}
```

Blocks


<<interface>>
Serializable
T, V
<<interface>>
Function
+ valueOf(object : T) : V
T
<<interface>>
Predicate
+ accept(each : T) : boolean
T
<<interface>>
Procedure
+ value(each : T) : void

Function

`Function` transforms one type (`T`) to another (`V`):

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

Used in:

- `collect`
- flatCollect
- `groupBy`
- `minBy`
- `maxBy`
- `toSortedListBy`
- `sortThisBy`
- `toMap`

Predicate

`Predicate` takes an object of some type (`T`) and returns a `boolean`:

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

Used in:

- `select`
- `reject`
- `detect`
- `count`
- `anySatisfy`
- `allSatisfy`

Predicate

`Procedure` takes an object of some type (`T`) and doesn't return anything:

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

Used in:

- forEach

RichIterable

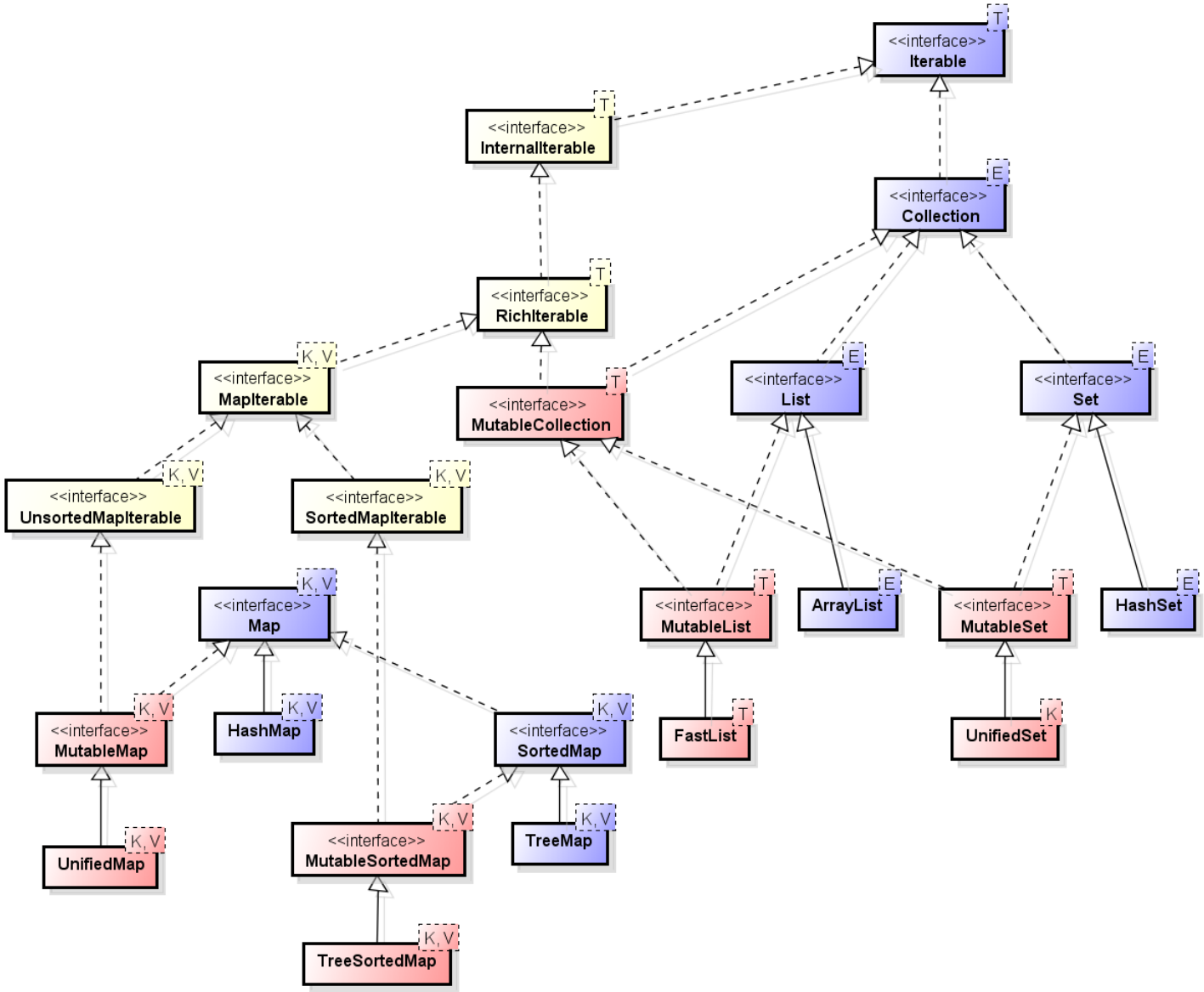
`collect`	Transforms elements using a `Function` into a new collection.
`flatCollect`	Transforms and flattens the elements using a `Function`.
`groupBy`	Gets a key for each element using a `Function` and puts the key and element into a `Multimap`.

`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 `Function`s and returns a new map containing all the key/value pairs.

`select`	Returns the elements of a collection that satisfy some condition (`Predicate`).
`reject`	Returns the elements of a collection that *do not* satisfy the `Predicate`.
`detect`	Finds the first element that satisfies the `Predicate`.
`count`	Returns the number of elements that satisfy the `Predicate.`
`anySatisfy`	Returns true if any element satisfies the `Predicate`.
`allSatisfy`	Returns true if all elements satisfy the `Predicate`.
`forEach`	Executes the `Procedure` on each element, doesn't return anything.

`min`	Returns the minimum element using either the natural order or a `Comparator`.
`max`	Returns the maximum element using either the natural order or a `Comparator`.
`toSortedList`	Returns a new list, sorted using either the natural order or a `Comparator`.
`makeString`	Converts the collection to a string using optional start, end, and separator strings.
`appendString`	Converts the collection to a string and appends it to an `Appendable`.
`zip`	Takes a second `RichIterable` and pairs up all the elements. If one of the two `RichIterable`s is longer than the other, its remaining elements are ignored.
`zipWithIndex`	Zips the collection with the `Integer` indexes 0 to n-1.
`chunk`	Splits a collection into fixed size chunks. The final chunk will be smaller if the collection doesn't divide evenly.

Inheritance Hierarchy



Optimization

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

```
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

```
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