Coding in the small with Google Collections

Robert Konigsberg, Jerome Mourits and myself have written several snippets that highlight the carefully designed Google Collections Library:

Preconditions

```
Preconditions.checkNotNull(order.getAddress(), "order address");
```

Iterables.getOnlyElement

```
assertEquals(jesse, Iterables.getOnlyElement(usersOnDuty));
```

Comparators.max

```
return Comparables.max(perKm, minimumDeliveryCharge);
```

Objects.equal and hashCode

```
return Objects.hashCode(address, targetArrivalDate, lineItems);
```

Lists.immutableList

```
this.steps = Lists.immutableList(steps);
```

Objects.nonNull

```
his.familyName = Objects.nonNull(familyName);
```

Iterables.concat

```
for    (LineItem    lineItem    :    Iterables.concat(getPurchasedItems(),
getFreeItems())) {
  ...
}
```

Constraints.constrainedList

```
public void checkElement(LineItem element) { ... }
```

Multimap

```
Multimap<Salesperson, Sale> multimap
    = new ArrayListMultimap<Salesperson,Sale>();
```

Join

```
return Join.join(" and ", items);
```

Maps, Sets and Lists

```
Set<String> workdays = Sets.newLinkedHashSet(
    "Monday", "Tuesday", "Wednesday", "Thursday", "Friday");
```

Comparators.fromFunction

```
return Comparators.fromFunction(new Function<Product,Money>() { ... });
```

BiMap

```
return NUMBER_TO_NAME_BIMAP.inverse().get(elementName);
```

ClassToInstanceMap

```
T result = optionalValuesByType.getInstance(type);
```

Preconditions

n this N-part series, I'll demonstrate my favourite APIs from the recently announced Google Collections project. I'll present before and after examples that show how you can use Google Collections to write more concise code.
Preconditions provides methods for state validation. It makes input validation compact enough that you'll always want to do it! And unlike Java's built-in **assert**, Preconditions is always enabled.

Before:

```
public Delivery createDelivery(Order order, User deliveryPerson) {
    if(order.getAddress() == null) {
      throw new NullPointerException("order address");
    }
    if(!workSchedule.isOnDuty(deliveryPerson, order.getArrivalTime())) {
      throw new IllegalArgumentException(
          String.format("%s is not on duty for %s", deliveryPerson, order));
    }

    return new RealDelivery(order, deliveryPerson);
  }
```

After:

```
  public Delivery createDelivery(Order order, User deliveryPerson) {
    Preconditions.checkNotNull(order.getAddress(), "order address");
    Preconditions.checkArgument(
        workSchedule.isOnDuty(deliveryPerson, order.getArrivalTime()),
        "%s is not on duty for %s", deliveryPerson, order);
    return new RealDelivery(order, deliveryPerson);
  }
```

In addition to a constraint check, Preconditions also works as programmer documentation. When the code is under maintenance, the original author's expectations are right in the code where they belong. As a special treat, Preconditions throws detailed error messages that include the offending line of code.

Iterables.getOnlyElement

Iterables.getOnlyElement makes sure your collection or iterable contains exactly one element, and returns that. If it contains 0 or 2+ elements, it throws a RuntimeException. This comes up often in unit tests:

Before:

```
  public void testWorkSchedule() {
    workSchedule.scheduleUserOnDuty(jesse, mondayAt430pm, mondayAt1130pm);

    Set<User> usersOnDuty = workSchedule.getUsersOnDuty(mondayAt800pm);
    assertEquals(1, usersOnDuty.size());
    assertEquals(jesse, usersOnDuty.iterator().next());
  }
```

After:

```
  public void testWorkSchedule() {
    workSchedule.scheduleUserOnDuty(jesse, mondayAt430pm, mondayAt1130pm);

    Set<User> usersOnDuty = workSchedule.getUsersOnDuty(mondayAt800pm);
    assertEquals(jesse, Iterables.getOnlyElement(usersOnDuty));
  }
```

Iterables.getOnlyElement() describes intent more directly than Set.iterator().next() and List.get(0). As a special treat, there's an overloaded version to use if your Iterable might be empty.

Comparators.max

Comparables.max takes two Comparables and returns the larger of the two. It improves upon the standard approach, which requires both a comparison and a ternary:

Before:

```
public Money calculateDeliveryCharge(Order order) {
   double distanceInKm = Geography.getDistance(
       storeAddress, order.getAddress());
   Money perKm = pricePerKm.times(distanceInKm);

   return perKm.compareTo(minimumDeliveryCharge) > 0
       ? perKm
       : minimumDeliveryCharge;
 }
```

After:

```
  public Money calculateDeliveryCharge(Order order) {
   double distanceInKm = Geography.getDistance(
       storeAddress, order.getAddress());
   Money perKm = pricePerKm.times(distanceInKm);

   return Comparables.max(perKm, minimumDeliveryCharge);
 }
```

Of course the Comparables.min method is also included. As a special treat, there's overloaded versions of each that allow you to specify your own comparator, such as String.CASE_INSENSITIVE_ORDER.

Objects.equal and hashCode

Objects.equal(Object,Object) and Objects.hashCode(Object...) provide built-in null-handling, which makes implementing your own `equals()` and `hashCode()` methods easy.

Before:

```
  public boolean equals(Object o) {
    if (o instanceof Order) {
      Order that = (Order)o;

      return (address != null
              ? address.equals(that.address)
              : that.address == null)
          && (targetArrivalDate != null
              ? targetArrivalDate.equals(that.targetArrivalDate)
              : that.targetArrivalDate == null)
          && lineItems.equals(that.lineItems);
    } else {
      return false;
    }
  }

  public int hashCode() {
    int result = 0;
    result = 31 * result + (address != null ? address.hashCode() : 0);
      result    =    31    *    result    +    (targetArrivalDate    !=    null    ?
targetArrivalDate.hashCode() : 0);
    result = 31 * result + lineItems.hashCode();
    return result;
  }
```

After:

```
  public boolean equals(Object o) {
    if (o instanceof Order) {
      Order that = (Order)o;
      return Objects.equal(address, that.address)
          && Objects.equal(targetArrivalDate, that.targetArrivalDate)
          && Objects.equal(lineItems, that.lineItems);
    } else {
      return false;
    }
  }

  public int hashCode() {
    return Objects.hashCode(address, targetArrivalDate, lineItems);
  }
```

This is much more concise than handwritten or IDE-generated code. As a special treat, there's deepEquals and deepHashcode methods that 'do the right thing' for arrays and nested arrays, that otherwise use identity for `equals` and `hashCode`.

Lists.immutableList

Lists.immutableList() creates an immutable copy of it's arguments as a List.

Wherever your code stores a List parameter, it may need an immutable copy. With the JDK, coping and preventing modification requires two steps. Lists.immutableList simplifies this code:

Before:

```
public Directions(Address from, Address to, List<Step> steps) {
  this.from = from;
  this.to = to;
  this.steps = Collections.unmodifiableList(new ArrayList<Step>(steps));
}
```

After:

```
public Directions(Address from, Address to, List<Step> steps) {
  this.from = from;
  this.to = to;
  this.steps = Lists.immutableList(steps);
}
```

As usual with Google Collections, all the expected overloadings are available. There's versions that accept Iterable, Iterator, and varargs/arrays. As a special treat, there's even an overloading that takes zero parameters which allows you to add and remove elements freely without changing the signature.

Objects.nonNull

In the first episode, Jesse talked about the Preconditions class. That's not the only mechanism supplied by the Google Collections library for data validation. We're going to look at Objects.nonNull(T) and, by association, Objects.nonNull(T, String).

Objects.nonNull accepts an object, and throws a NullPointerException if it is null. Otherwise it returns the value passed into the method.

Objects.nonNull is wonderfully useful in constructors. Since any use of super in a constructor must be the first statement in the method body, Preconditions can't be used. before the superclass initialization. The Precondition can be used after the superclass initialization, but that doesn't work very well; it forces initialization prior to validation, and besides, Preconditions have a semantic inference; they're meant to be used as pre-conditions, not post-conditions.

Before:

```
public ConnectAction extends Action {
  private final Connector connector;
  public ConnectAction(ContextManager mgr, Connector connector) {
    super(mgr);
    Preconditions.checkNotNull(mgr);
    this.connector = connector;
  }
  public void run(ActionArguments args) { ... }
}
```

After:

```
public ConnectAction extends Action {
  private final Connector connector;
  public ConnectAction(ContextManager mgr, Connector connector) {
    super(Objects.nonNull(mgr));
    this.connector = connector;
  }
  public void run(ActionArguments args) { ... }
}
```

See? Objects.nonNull operate like a filter within an expression, whereas Preconditions do not.

Objects.nonNull also works well when supplying a large number of arguments:

Before:

```
public Person(String givenName, String familyName, Address address, Phone phone,
String email) {
  Preconditions.checkNotNull(givenName);
  Preconditions.checkNotNull(familyName);
  Preconditions.checkNotNull(address);
  Preconditions.checkNotNull(phone);
  Preconditions.checkNotNull(email);

  this.givenName = givenName;
  this.familyName = familyName;
  this.address = address;
  this.phone = phone;
  this.email = email;
}
```

After:

```
public Person(String givenName, String familyName, Address address, Phone phone,
String email) {
  this.givenName = Objects.nonNull(givenName);
  this.familyName = Objects.nonNull(familyName);
  this.address = Objects.nonNull(address);
  this.phone = Objects.nonNull(phone);
  this.email = Objects.nonNull(email);
}
```

Another interesting use of Objects.nonNull is as a post-condition operator:

Before:

```
public V get(K key) {
  V value = map.get(key);
  if (value == null) {
    throw new NullPointerException();
  } else {
    return value;
  }
}
```

After:

```
public V get(K key) {
  return Objects.nonNull(map.get(key));
}
```

So, when should you use Preconditions.checkNotNull and Objects.nonNull? There's a soft answer: see what feels right. You're a smart programmer, I'm sure.

Iterables.concat()

Iterables.concat() combines multiple iterables (such as ArrayList and HashSet) so you can go through multiple collections' elements in a single pass:

Before:

```
public boolean orderContains(Product product) {
  List<LineItem> allLineItems = new ArrayList<LineItem>();
  allLineItems.addAll(getPurchasedItems());
  allLineItems.addAll(getFreeItems());

  for (LineItem lineItem : allLineItems) {
    if (lineItem.getProduct() == product) {
      return true;
    }
  }

  return false;
}
```

After:

```
public boolean orderContains(Product product) {
    for    (LineItem    lineItem    :    Iterables.concat(getPurchasedItems(),
getFreeItems())) {
    if (lineItem.getProduct() == product) {
      return true;
    }
  }

  return false;
}
```

If ever you only have the Iterator and not the Iterable, the equivalent method is Iterators.concat. As a special treat, both concat methods are optimized so that they don't need a private collection.

Constraints.constrainedList

Constraints.constrainedList allows you to enforce rules about what can be added to a List instance. By letting the List manage its constraints, your API users get instant feedback if they insert invalid values. It also allows you to expose convenient methods like addAll() and set() without writing additional code:

Before:

```
  private final List<LineItem> purchases = new ArrayList<LineItem>();

  /**
   * Don't modify this! Instead, call {@link #addPurchase(LineItem)} to add
   * new purchases to this order.
   */
  public List<LineItem> getPurchases() {
    return Collections.unmodifiableList(purchases);
  }

  public void addPurchase(LineItem purchase) {
    Preconditions.checkState(catalog.isOffered(getAddress(),
purchase.getProduct()));
    Preconditions.checkState(purchase.getCharge().getUnits() > 0);
    purchases.add(purchase);
  }

  ...

  public static Order createGreyCupSpecialOrder(Customer customer) {
    Order order = new Order(customer);
    for (LineItem lineItem : GREY_CUP_SPECIAL_ITEMS) {
      order.addPurchase(lineItem);
    }
    return order;
  }
```

After:

```
  private final List<LineItem> purchases = Constraints.constrainedList(
      new ArrayList<LineItem>(),
      new Constraint<LineItem>() {
        public void checkElement(LineItem element) {
          Preconditions.checkState(catalog.isOffered(getAddress(),
element.getProduct()));
          Preconditions.checkState(element.getCharge().getUnits() > 0);
        }
      });

  /**
```

```
   * Returns the modifiable list of purchases in this order.
   */
  public List<LineItem> getPurchases() {
    return purchases;
  }

  ...

  public static Order createGreyCupSpecialOrder(Customer customer) {
    Order order = new Order(customer);
    order.getPurchases().addAll(GREY_CUP_SPECIAL_ITEMS);
    return order;
  }
```

This new code is both more robust and more convenient. Possibly the most common constraint, NOT_NULL is provided to make that easy. And as a special treat, there's constraint factory methods for Sets, Maps and the supplementary collections.

Multimap

Multimap is like a Map, but lets you store multiple values for every key. It turns out this is frequently useful. Consider how often you have had to create, for instance, a Map<K, List<V>>.

Before:

```
  Map<Salesperson, List<Sale>> map = new Hashmap<SalesPerson, List<Sale>>();

  public void makeSale(Salesperson salesPerson, Sale sale) {
    List<Sale> sales = map.get(salesPerson);
    if (sales == null) {
      sales = new ArrayList<Sale>();
      map.put(salesPerson, sales);
    }
    sales.add(sale);
  }
```

After:

```
  Multimap<Salesperson, Sale> multimap
      = new ArrayListMultimap<Salesperson,Sale>();
  public void makeSale(Salesperson salesPerson, Sale sale) {
    multimap.put(salesperson, sale);
  }
```

This is just one facet of Multimaps. Consider finding the largest value across all keys:

Before:

```
  public boolean getBiggestSale() {
    Sale biggestSale = null;
    for (List<Sale> sales : map.values()) {
      Sale biggestSaleForSalesman
          = Collections.max(sales, SALE_COST_COMPARATOR);
      if (biggestSale == null
          || biggestSaleForSalesman.getCharge() > biggestSale().getCharge()) {
        biggestSale = biggestSaleForSalesman;
      }
    }
    return biggestSale;
  }
```

After:

```
  public boolean getBiggestSale() {
    return Collections.max(multimap.values(), SALE_COST_COMPARATOR);
  }
```

The Google Collections Library comes with a large variety of Multimap implementations that you can choose from to meet your specific performance characteristics. Do you want to prevent duplicate key/value pairs? Use HashMultimap. Do you want to iterate over the maps in insertion-order? Use LinkedHashMultimap. Do you want the keys and values ordered by their natural ordering? Use TreeMultimap.

Don't forget to read the Javadoc for Multimaps, which provides access to all the supplied implementations, and much more.

Join

Join makes it easy to join Strings separated by a delimiter.

Before:

```
public class ShoppingList {
  private List<Item> items = ...;

  ...

  public String toString() {
    StringBuilder stringBuilder = new StringBuilder();
    for (Iterator<Item> s = items.iterator(); s.hasNext(); ) {
      stringBuilder.append(s.next());
      if (s.hasNext()) {
        stringBuilder.append(" and ");
      }
    }
    return stringBuilder.toString();
  }
}
```

After:

```
public class ShoppingList {
  private List<Item> items = ...;

  ...

  public String toString() {
    return Join.join(" and ", items);
  }
}
```

Easy! Join supports Iterators, Iterables, arrays and varargs. You can also have join append the tokens to a supplied Appendable, like StringBuilder.

Maps, Sets and Lists

Generics are good, but they can be really wordy!

Before:

```
Map<CustomerId, BillingOrderHistory> customerOrderHistoryMap
```

```
    = new HashMap<CustomerId, CustomerOrderHistory>();
```

After:

```
Map<CustomerId, CustomerBillingOrderHistory> customerOrderHistoryMap
    = Maps.newHashMap();
```

Look Ma! I don't have to specify my type parameters twice! (The compiler figures it out through Type Inference from Assignment Context). Maps, Sets and Lists contain factory methods to create Collections objects. Here's another useful one:

Before:

```
Set<String> workdays = new LinkedHashSet<String>();
workdays.add("Monday");
workdays.add("Tuesday");
workdays.add("Wednesday");
workdays.add("Thursday");
workdays.add("Friday");
```

Or:

```
Set<String> workdays = new LinkedHashSet<String>(
  Arrays.asList("Monday", "Tuesday", "Wednesday", "Thursday", "Friday"));
```

After:

```
Set<String> workdays = Sets.newLinkedHashSet(
  "Monday", "Tuesday", "Wednesday", "Thursday", "Friday");
```

Google Collections provides factory methods for Maps, Sets, Lists, Multimaps, Multisets and other types.

Comparators.fromFunction

Comparators.fromFunction() allows you to create a Comparator based on the properties and attributes of a your business objects. Since the same function is applied to both of the values to compare, this method makes it easier to write a symmetric comparator:

Before:

```
public Comparator<Product> createRetailPriceComparator(
    final CurrencyConverter currencyConverter) {
  return new Comparator<Product>() {
    public int compare(Product a, Product b) {
      return getRetailPriceInUsd(a).compareTo(getRetailPriceInUsd(b));
    }
    public Money getRetailPriceInUsd(Product product) {
      Money retailPrice = product.getRetailPrice();
      return retailPrice.getCurrency() == CurrencyCode.USD
          ? retailPrice
          : currencyConverter.convert(retailPrice, CurrencyCode.USD);
    }
  };
}
```

After:

```
public Comparator<Product> createRetailPriceComparator(
    final CurrencyConverter currencyConverter) {
  return Comparators.fromFunction(new Function<Product,Money>() {
    /** returns the retail price in USD */
    public Money apply(Product product) {
      Money retailPrice = product.getRetailPrice();
      return retailPrice.getCurrency() == CurrencyCode.USD
          ? retailPrice
          : currencyConverter.convert(retailPrice, CurrencyCode.USD);
    }
  });
}
```

This makes it easy to write expressive comparators. As a special treat, there's an overloaded version of fromFunction that lets you to specify which Comparator for ordering the function's return values. For example, null is supported with Comparators.nullGreatestOrder() as the second argument. Or put the priciest products first using Collections.reverseOrder() as the second argument.

BiMap

A BiMap is a map that supports an inverse view. If you need to map from key to value and also from value to key, then a BiMap is what you want! Note that the values of a BiMap must be unique so that the inverse view makes sense.

Before:

```
private static final Map<Integer, String> NUMBER_TO_NAME;
private static final Map<String, Integer> NAME_TO_NUMBER;

static {
  NUMBER_TO_NAME = Maps.newHashMap();
  NUMBER_TO_NAME.put(1, "Hydrogen");
  NUMBER_TO_NAME.put(2, "Helium");
  NUMBER_TO_NAME.put(3, "Lithium");

  /* reverse the map programatically so the actual mapping is not repeated */
  NAME_TO_NUMBER = Maps.newHashMap();
  for (Integer number : NUMBER_TO_NAME.keySet()) {
    NAME_TO_NUMBER.put(NUMBER_TO_NAME.get(number), number);
  }
}

public static int getElementNumber(String elementName) {
  return NUMBER_TO_NAME.get(elementName);
}

public static string getElementName(int elementNumber) {
  return NAME_TO_NUMBER.get(elementNumber);
}
```

After:

```
private static final BiMap<Integer,String> NUMBER_TO_NAME_BIMAP;

static {
  NUMBER_TO_NAME_BIMAP = Maps.newHashBiMap();
  NUMBER_TO_NAME_BIMAP.put(1, "Hydrogen");
  NUMBER_TO_NAME_BIMAP.put(2, "Helium");
  NUMBER_TO_NAME_BIMAP.put(3, "Lithium");
}

public static int getElementNumber(String elementName) {
  return NUMBER_TO_NAME_BIMAP.inverse().get(elementName);
}
```

```
  public static string getElementName(int elementNumber) {
    return NUMBER_TO_NAME_BIMAP.get(elementNumber);
  }
```

Even Better:

```
  private static final BiMap<Integer,String> NUMBER_TO_NAME_BIMAP
    = new ImmutableBiMapBuilder<Integer,String>()
        .put(1, "Hydrogen")
        .put(2, "Helium")
        .put(3, "Lithium")
        .getBiMap();
  ...
```

The Google Collections Library provides three BiMap implementations:

- EnumBiMap is backed by EnumMaps in both directions.
- EnumHashBiMap is backed by an EnumMap in the foward direction and a Hashmap in the inverse direction.
- HashBiMap is backed by HashMaps in both directions.

ClassToInstanceMap

ClassToInstanceMap is a specialized Map whose keys are class literals like **PizzaPromotion.class** or **RestockingInformation.class** and whose values are instances of those types. It provides a convenient balance between type safety and model flexibility.

In some code I wrote recently, I needed to attach a RestockingInformation object to a Product without explicitly mentioning (and depending on) RestockingInformation in the Product class. I also wanted to allow other objects to be attached as necessary in the future:

Before:

```
public class Product {

   ...

  private final Map<Class<?>, Optional> optionalValuesByType
      = new HashMap<Class<?>, Optional>();

  public <T extends Optional> void putOptional(Class<T> type, T value) {
    type.cast(value); /* ensure the value is assignable to T */
    optionalValuesByType.put(type, value);
  }

  @SuppressWarnings({"unchecked"})
  public <T extends Optional> T getOptional(Class<T> type) {
    return Objects.nonNull((T)optionalValuesByType.get(type));
  }

  @SuppressWarnings({"unchecked"})
  public <T extends Optional> T getOptional(Class<T> type, T defaultValue) {
    T result = (T)optionalValuesByType.get(type);
    return result != null
        ? result
        : defaultValue;
  }

  private interface Optional {
    /* marker interface */
  }
}
```

After:

```
public class Product {

   ...

  private final ClassToInstanceMap<Optional> optionalValuesByType
      = Maps.newClassToInstanceMap();

  public <T extends Optional> void putOptional(Class<T> type, T value) {
    optionalValuesByType.putInstance(type, value);
  }

  public <T extends Optional> T getOptional(Class<T> type) {
    return Objects.nonNull(optionalValuesByType.getInstance(type));
  }

  public <T extends Optional> T getOptional(Class<T> type, T defaultValue) {
    T result = optionalValuesByType.getInstance(type);
    return result != null
        ? result
        : defaultValue;
  }
  private interface Optional {
    /* marker interface */
  }
}
```

Class literals are ideal map keys because they already have a global namespace - Java class names are distinct! Anyone can introduce their own **Optional** implementation without fear of clobbering someone else's value.

Suppose the inventory team added an optional type called **com.publicobject.pizza.inventory.DeliveryInfo** to describe how a product is transported from the warehouse to the storefront. This doesn't prevent the customer service team from tracking a 30-minute guarantee in their own **com.publicobject.pizza.customers.DeliveryInfo** class. Both keys can be added to a map without clobbering each other. Using Strings for keys is not nearly as safe!

The idea of using class literals as map keys is useful everywhere, but Guice made it famous. ClassToInstanceMap makes it easy to do everywhere else.