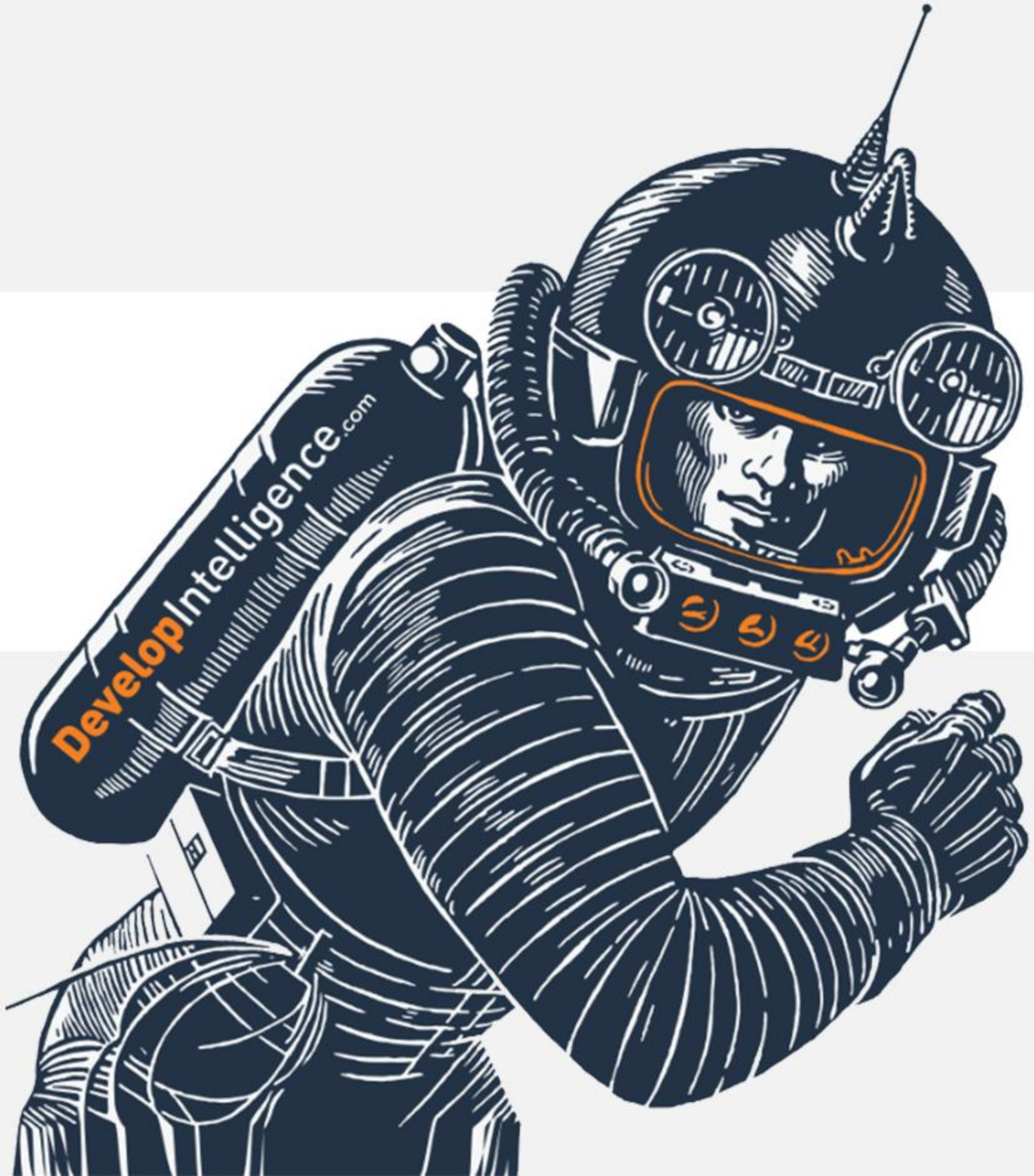
Collections and Generics

Develop
Intelligence



Why Collections? Why Generics?

Collections are all about storing and processing bulk data

Generics allow them to do this in a type-safe way

The Java Collections Framework

- `Collection`
  - Base interface for `Set`, `List`, and `Queue` interfaces
- `Set`
  - Elements are unique and unordered – except for…
  - `SortedSet/NavigableSet`
    - Ordered sets with navigation methods
- `List`
  - Elements are stored sequentially, by position
- `Queue`
  - Holds elements, yielding them in the order required for processing
- `Map/SortedMap/NavigableMap`
  - Store key-value pairs

Implementations

- Arrays
  - fast for traversal, slow for insertions and removals
  - eg `ArrayList`, `CopyOnWriteArrayList`, `Queues`
- Linked Lists
  - slow for traversal and accessing by position, fast for insertion/removal
  - eg `LinkedBlockingQueue, LinkedList`
- Hash Tables
  - fast access by content, fast insertion/removal
  - eg `HashSet, HashMap, ConcurrentHashMap`
- Trees
  - access by content, maintain sorted order
  - eg `TreeSet, TreeMap`

Unmodifiable Collections

- Static methods in `java.util.Collections:`
  - `unmodifiableCollection(),unmodifiableList(), unmodifiableMap/Set() unmodifiableNavigableMap/Set()`
- Factory methods in `List, Set, Map` (from Java 9)
  - `List.of(), Set.of(). // up to 10 elements`
  - `Map.of(k,v,k1,v1, …) // up to 10 key-value pairs`
- `List.copyOf()`   (from java 10)

Iterators

- Three kinds of iterators in the Collections Framework:
- Thread-safe: first-generation collections, obsolete now – `Vector, Stack, Hashtable`
- Fail-fast – `ArrayList, HashMap`, etc
  - throw `ConcurrentModificationException`
- Weakly consistent – concurrent collections: `ConcurrentHashMap`, etc
  - reflect only some changes made after they are created

Tips for equals( ) and hashCode()

- Decide what identifies your objects
  - e.g. for a socket: ipAddress and port
- Make `equals()` method compare both
- Calculate the `hashCode()` depending on the hash codes of these fields – and no others!

```
class Person {
   @Override
   public boolean equals(Object o) {
       if (this == o) return true;
       if (o == null || getClass() != o.getClass()) return false;
       Person person = (Person) o;
       return fName.equals(person.fName) && lName.equals(person.lName);
   }
}
```

Ordering Collection Elements

- Java classes may have a ***natural order***: they implement

```
public interface Comparable<T> {
   int compareTo(T o);
}
```

- Wrapper classes, `String`, and many other platform classes
- Sorting methods and sorted data structures accept `Comparable` objects
  - Negative value if argument is greater than this
  - Zero if the argument is equal
  - Positive if the argument is less
  - `a.compareTo(b) == -b.compareTo(a)`
  - Algorithm must be transitive
  - Zero should correspond to equals() true

Ordering Collection Elements

- Also, classes can have an external order imposed on them

```
public interface Comparator<T> {
   int compareTo(T o1, T o2);
}
```

- The `compare( )` method must return a negative integer, zero, or a positive integer if the first argument is less than, equal to, or greater than the second, respectively

Comparator

Comparators were enhanced with static and default methods in Java 8. Now, instead of writing this:

```
Comparator<Person> cmpr = new Comparator<>() {
  @Override
  public int compare(Person p1, Person p2) {
    int cmp = p1.getLastName().compareTo(p2.getLastName());
    if (cmp == 0) {
      return p1.getFirstName().compareTo(p2.getFirstName());
    } else {
      return cmp;
    }
  }
};
```

Comparator

We can instead write

```
Comparator<Person> cmp =
    Comparator.comparing(Person::getLastName)
        .thenComparing(Person::getFirstName)
        .thenComparing(Person::getAge);
```

using default and factory methods on Comparator

Comparator Methods returning Comparator<T>

	name	argument(s)
static	comparing	Function<T,U>
	comparing	Function<T,U>, Comparator<U>
	comparingXxx	ToXxxFunction<T>
	naturalOrder	
	nullsFirst/Last	Comparator<T>
	reverseOrder	Comparator<T>
default	reversed	
	thenComparing	Comparator<T>
	thenComparingXxx	ToXxxFunction<T>

Comparators Lab

Person
firstName: String lastName: String age: int

Why Generics?

- Provide type safety for classes and methods
  - No possibility of adding an incorrect value
  - No need to cast a retrieved value
- Provide information for type inference – clearer and more concise code
  - See in detail with streams and lambdas

Variance

- Arrays are *covariant*:
  - `Integer` is a subtype of `Number`, so
  - `Integer[]` is a subtype of `Number[]`
- But that means that this is legal:

```
Integer[] ints = ...;
Number[] numbers = ints;
numbers[0] = 3.14;          // boom!
```

- You can't safely put subtype values (except `null`) into covariant data structures
- The variable `numbers` refers to a collection of some subtype of `Number`, but you don't know which

Variance

- Generic data structures are *invariant*:
  - `List<Integer>` is *not* a subtype of `List<Number>`
  - and `List<Number>` is not a subtype of `List<Integer>`
- So a run-time exception has become a compile error:

```
List<Integer> ints = ...;
List<Number> numbers = ints; // compile error
```

- That’s a win!
- But why *were* arrays given covariant typing?

Covariance

- Arrays were covariantly typed so that you could write methods like

```
Arrays.sort(Number[] numbers);
```

  - That works on arrays of `Integer`, `Double`, ...
- Covariantly typed structures are good for *retrieving* things
- So we need covariantly typed generic data structures too!
  - `List<? extends Number>`
- `You could write a method`

```
sort(List<? extends Number);
```

  - That would work the same way as the `Arrays` method

Contravariance

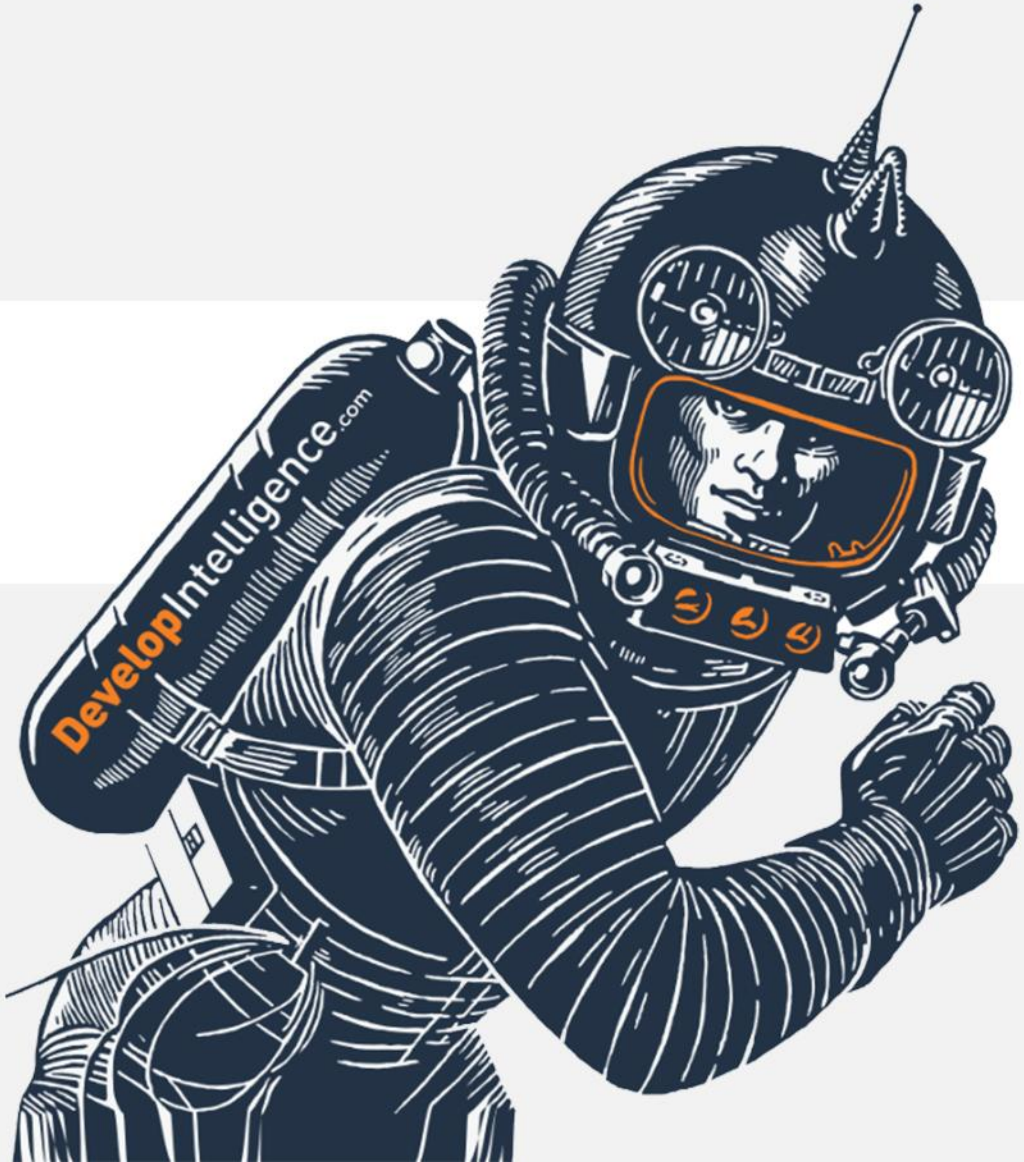
- What about putting things into collections?
- Think of something that consumes things:
  - A `Feeder<T>`, say
  - A `Feeder<Plant>` would accept a `Banana` or a `Lettuce` (subtypes of `Plant`)
- Contravariantly typed structures are good for *accepting* things
  - `List<? super Employee>` will accept a `Manager` or a `Director`
  - an example method declaration, from `java.util.Collections`:

```
void <T> copy(List<? super T> dest, List<? extends T> src)
```

Writing a Generic Class

- Let's write a class `Pair` that will take two objects of the same class
- And then extend it with a class that allows those two objects to be compared

Questions?



Develop
Intelligence

Collections & Generics Exercise

A `Bag`, or `Multiset`, is like a `Set` in that the order of elements is not significant. However, it can contain more than one occurrence of the same element. Adding an element to a `Bag` increases the count of its occurrences by one; removing decreases it by one.

The Google Guava collection defines a `Multiset` interface (`https://guava.dev/releases/18.0/api/docs/com/google/common/collect/Multiset.html`). Using whatever Java classes seem most useful to you, implement and test the methods of the Guava `Multiset`.

For a bonus, maintain your `Multiset` in sorted order, like a `NavigableSet`.

With the right implementation, this exercise isn't difficult!