Style Guide

Classes

1. Classes should only contain one class (non-nested)
2. Every class should have it's own source files
3. Overloaded methods should appear consecutively

Identifiers

1. One variable per declaration
2. Class names to be in **UpperCamelCases**
3. Method names to be in **lowerCamelCases**
4. Constants (static final) in **ALL_CAPS_SNAKE_CASE**

Access modifiers

- `public` - visible from outside the class
- `private` - only visible within enclosing class
- `protected` (`default`) - only visible within enclosing class and its subclasses
- `static` - associate field with class
- `final` - value will not change

Interface

```
interface B {}
class C extends A implements B {}
interface A extends B, C, D
```

Exceptions

```
class NewCheckedExceptions extends Exception {
        public NewCheckedExceptions() { super("msg") } }

class NewUncheckedExceptions extends RunTimeException

class C throws NewCheckedException {
        throw new NewCheckedException("message")
}
```

Checked exceptions must be handled
If an overloaded method throws an exception, it must be a **subtype** of the exception thrown in the parent class.

Annotations

```
@Override - for any overriden methods
    - can be from parent class or interface
@SuppressWarnings("unchecked") - for typecasting
@SuppressWarnings("rawtypes") - for using rawtypes
```

Formatting

```
String.format("%s\%d\%.2f", "hello", 123, 45.67)
```

Generics

```
class Pair<S, T> {}
class DictEntry<T> extends Pair<String, T> {}
class Pair<S extends Comparable<S>, T> {}
public static <T> boolean contains(T[] arr, T obj) {}
public static <T extends GetAreable> T findLargest(T[] arr)
public <U> void printSomething(U value)

    @SuppressWarnings("unchecked")
    T[] a = (T[]) new Object[size];
    this.array = a;
```

Wildcards

Unbounded `<?>` - The parent class of all wildcards
Lower bounded `<? super T>` - When we want to **add to** a collection instead of retrieving from
Upper bounded `<? extends T>` - When we want to **retrieve from** instead of adding to

PECS - Producer Extends Consumer Super

extends - to get values out of a structure
super - to put values into a structure
DO NOT use a wildcard when you both **get** and **put**

Functional Interfaces

```
BooleanCondition -> boolean Predicate<T>::test(T t)
Producer -> T Supplier<T>::get()
Consumer -> void Consumer<T>::accept(T t)
Transformer<T, R> -> R Function<T, R>::apply(T t)
Transformer<T, T> -> T UnaryOperator<T>::apply(T t)
Combiner<S, T, R> -> R BiFunction<S, T, R>::apply(S s, T t)
Combiner<T, T, T> -> T BinaryOperator<T>::apply(T t)
```

Streams

`of(T... values)` creates a stream of values
`generate(Supplier<T> s)` creates an unordered stream of producers
`iterate(T seed, UnaryOperator<T> f)` creates an unordered stream of seed, f(seed), f(f(seed)), ...
`filter(Predicate<? super T> pred)` filters stream according to predicate
`flatMap(Function<? super T, ? extends Stream<? extends R>> mapper)` returns a stream with transformed elements
`map(Function<? super T, ? extends R> mapper)` returns a stream with transformed elements
`limit(long maxSize)` returns a truncated stream, length == maxSize
`reduce(T identity, BinaryOperator<T> accumulator)` accumulate items in stream - e.g. `reduce(0, (acc, val) -> acc + val)`
`reduce(U identity, BiFunction<U, ? super T, U> accumulator, BinaryOperator<U> combiner)` accumulate items in stream, supports `parallel()`
`takeWhile(Predicate<? super T> pred)` only ordered stream, return stream of elements until predicate false

`parallel()` stream runs in parallel
`forEach(Consumer<? super T> action)` performs action on every element
`count()` returns number of elements
`distinct()` returns a stream of distinct elements
`concat(Stream<? extends T> a, Stream<? extends T> b)` creates a lazily evaluated stream of A+B
`peek(Consumer<? super T> action)` apply action to every element and return stream
`sorted(Comparator<? super T> comparator)` stream sorted based on comparator
`toList()` return stream in `List`

`allMatch(Predicate<? super T> pred)` returns true if all elements match the predicate
`anyMatch(Predicate<? super T> pred)` returns true if at least 1 element matches the predicate
`noneMatch(Predicate<? super T> pred)` returns true if no elements match the predicate

Useful Code

Maybe - item can be null

```
private static final None NONE = new None();

public static <T> Maybe<T> of(T item) {
    return item == null ? none() : some(item);
}

public static <T> Maybe<T> none() {
    @SuppressWarnings("unchecked")
    Maybe<T> tmp = (Maybe<T>) NONE;
    return tmp;
}

public void ifPresent(Consumer<? super T> c) {
    c.consume(this.get());
}

public <U> Maybe<U> flatMap(Transformer<? super T,
        ? extends Maybe<? extends U>> tfm) {
    @SuppressWarnings("unchecked")
    Maybe<U> res = (Maybe<U>) tfm.transform(this.item);
    return res;
}

public <U> Maybe<U> map(Transformer<? super T,
        ? extends U> tfm) {
    return this.some(tfm.transform(this.item));
}

public Maybe<T> filter(BooleanCondition<? super T> cond) {
    // if not null and failed test, return None
    if (this.item != null && !cond.test(this.item)) {
        return none();
    }
    return this;
}

public boolean equals(Object obj) {
    // if referring to itself, return true
    if (this == obj) {
        return true;
    }
    // return false if null or not a subtype of Some
    if (obj == null || !(obj instanceof Some<?>)) {
        return false;
```

```
    }
    // safe to type cast here since obj would be of type Some
    Some<?> s = (Some<?>) obj;
    // check if contents are the same
    if (this.item == null) {
        return s.item == null;
    }
    // *use content's own equals method to check
    return this.item.equals(s.item);
}
```

Lazy - delayed evaluation

```
public <U> Lazy<U> map(Transformer<? super T,
        ? extends U> tfm) {
    return Lazy.of(() -> tfm.transform(this.get()));
}

public <U> Lazy<U> flatMap(Transformer<? super T,
        ? extends Lazy<? extends U>> tfm) {
    return Lazy.of(() -> tfm.transform(this.get()).get());
}

public Lazy<Boolean> filter(BooleanCondition<? super T> cond){
    return Lazy.of(
        () -> this.value.map(x -> cond.test(x)).get());
}

public <Y, Z> Lazy<Z> combine(Lazy<Y> lazyObj,
        Combiner<? super T, ? super Y, ? extends Z> combiner) {
    return Lazy.of(
        () -> combiner.combine(this.get(), lazyObj.get()));
}

public boolean equals(Object obj) {
    if (obj instanceof Lazy<?> l) {
        // shorthand for checking if obj instanceof Lazy<?>
        // and casting it -> same as Lazy<?> l = (Lazy<?>) obj
        this.get();
        l.get();
        return this.value.equals(l.value);
    }
    return false;
}
```

InfiniteList

```
public static <T> InfiniteList<T> generate(
        Producer<T> producer) {
    return new InfiniteList<T>(
        Lazy.of(() -> Maybe.some(producer.produce())),
        Lazy.of(() -> InfiniteList.generate(producer)));
}

public T head() {
    return this.head.get().orElseGet(
        () -> this.tail.get().head());
}

public InfiniteList<T> tail() {
    this.head.get();
    return this.head.get().equals(Maybe.none()) ?
        this.tail.get().tail() : this.tail.get();
}

public <R> InfiniteList<R> map(Transformer<? super T,
        ? extends R> mapper) {
    return new InfiniteList<>(
        this.head.map(maybe -> maybe.map(mapper)),
        this.tail.map(ifl -> ifl.map(mapper)));
}

public InfiniteList<T> filter(
        BooleanCondition<? super T> predicate) {
    return new InfiniteList<>(
        this.head.map(maybe -> maybe.filter(predicate)),
        this.tail.map(ifl -> ifl.filter(predicate)));
}

public InfiniteList<T> limit(long n) {
    if (n <= 0 || this.isSentinel()) {
      return this.sentinel();
    } else {
      return new InfiniteList<T>(
          this.head,
          Lazy.of(
              () -> this.head.get().equals(Maybe.none()) ?
                // curr head == NONE, # valid items no change
                // call limits on tail without decrementing n
                // .get() to unwrap lazy
                this.tail.map(ifl -> ifl.limit(n)).get()
                :
                // curr head != NONE, # valid items + 1
                // decrement n and call limits on tail
                this.tail.map(ifl -> ifl.limit(n - 1)).get()));
    }
}

public InfiniteList<T> takeWhile(
        BooleanCondition<? super T> predicate) {
    // test if head passes predicate,
    // returns () -> true and () -> false
    Lazy<Boolean> test = Lazy.of(() -> this.head.get()
                .map(t -> predicate.test(t)).orElse(true));
    return new InfiniteList<>(
        Lazy.of(() -> test.get() ?
            this.head.get() : Maybe.none()),
        Lazy.of(() -> test.get() ?
            this.tail.get().takeWhile(predicate) :
            this.sentinel()));
}

public <U> U reduce(U identity,
        Combiner<U, ? super T, U> accumulator) {
    if (this.isSentinel()) {
      return identity;
    } else {
      U res = accumulator.combine(identity,
            this.head.get().orElse(null));
      return this.tail.get().reduce(res, accumulator);
    }
}

public long count() {
    long res = this.head.get().equals(Maybe.none()) ?
                    this.tail.get().count() :
                    1 + this.tail.get().count();
    return res;
}

public List<T> toList() {
    List<T> list = new ArrayList<T>();
    this.head.get().ifPresent(x -> list.add(x));
    list.addAll(this.tail.get().toList());
    return list;
}
```

General Tips - @zaidansani

Immutability

1. Set all methods, fields to `final`
2. Ensure that any methods changing state returns a new object instead

Maybe

1. `Maybe::get` is protected, cannot be used in most cases
2. `Maybe::map` can be used to traverse if `Maybe::Some`
3. `Maybe::orElseGet` can be used to traverse if `Maybe::None`
4. `Maybe::ifPresent` can be used to add element to data struct - `maybe.ifPresent(item -> struct.add(item))`
5. Adding multiple items in `Maybe` using `ifPresent` - `maybe1.ifPresent(item1 -> maybe2.ifPresent(item2 -> struct.add(item1, item2)))`

Streams

1. Use `Stream::map` to convert `Stream` into desired format - `intStream.map(i -> i.toString())`
2. Use `Stream::flatMap` to expand stream - `List.of([1, 2]).stream().flatMap(x -> Stream.iterate(x, y -> y * 2).limit(2))` returns `[1, 2, 2, 4]`