## 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

### Lines

1. One blank line after import
2. Each statement is followed by a line break

### 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**

**public/private** field is accessible/not accessible from outside the class
**static** associate field with class
**final** value will not change

## Abstract class

```
abstract class A {
        public abstract void ...
    public ...
    // note that a class is abstract as long as one
    // or more methods are abstract
}
```

## Interface

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

## Creating Exceptions

```
class NewCheckedExceptions extends Exception {
        public NewCheckedExceptions() {
                super("message")
        }
}
class NewUncheckedExceptions extends RunTimeException
```

## Throwing Exceptions

```
class C throws NewCheckedException {
        throw new NewCheckedException("message")
}
...
try {
} catch (NewCheckedException e) {
        System.out.println(e.getMessage())
}
```

## Annotations

```
@Override – for any overriden methods
    can be from parent class or interface
@SuppressWarnings("unchecked") – for typecasting
@SuppressWarnings("rawtypes") – for using rawtypes
@SuppressWarnings({"unchecked", "rawtypes"})
    – for supressing multiple warnings
```

## Formatting

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

## Loops

```
for (T curr : array);
do {...} while (condition);
```

## Generics

```
class Pair<S, T> {
    private S first;
    private T second;

    public Pair(S first, T second) {
        this.first = first;
        this.second = second;
    }

    public S getFirst() {
        return this.first;
    }
}

// DictEntry follows the T type in Pair
class DictEntry<T> extends Pair<String, T> {}

// bounded parameters
class Pair<S extends Comparable<S>, T> {}

// arrays and generics dont mix
// need to declare before assigning
class Seq<T> {
  private T[] array;

  public Seq(int size) {
    // The only way we can put an object into array
    // is through the method set() and we only put
    // object of type T inside. So it is safe to
    // cast `Object[]` to `T[]`.
    @SuppressWarnings("unchecked")
    T[] a = (T[]) new Object[size];
    this.array = a;
  }
}
```

### Generic methods

```
// type parameter must appear before the return type
public static <T> boolean contains(T[] arr, T obj) {
    for (T curr: array) {
        if (curr.equals(obj)) {
            return true;
        }
    }
    return false;
}

// bounded class generics method
public static <T extends GetAreable> T
    findLargest(T[] arr)

// instance method
public <U> void printSomething(U value)
```

## Wildcards - <?>

The parent class of all wildcards

### Lower bounded <? extends T>

When we want to **retrieve from** instead of adding to

```
public void copyFrom(Seq<? extends T> src) {
    int len = Math.min(this.array.length,
        src.array.length);
    for (int i = 0; i < len; i++) {
        this.set(i, src.get(i));
    }
}
```

### Upper bounded <? super T>

When we want to **add to** a collection instead of retrieving from

```
public void copyTo(Seq<? super T> dest) {
    int len = Math.min(this.array.length,
        dest.array.length);
    for (int i = 0; i < len; i++) {
        dest.set(i, this.get(i));
    }
}
```

## 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**

| | | Bounded Parameter Produces Ts? | |
|---|---|---|---|
| | | Yes | No |
| Bounded Parameter Consumes Ts? | Yes | MyClass<T> (Invariant in T) | MyClass<? super T> (Contravariant in T) |
| | No | MyClass<? extends T> (Covariant in T) | MyClass<?> (Independent of T) |