1. **Nested Generics**

   (a) Suppose we have the following classes:

```
class Animal {
}

class Dog extends Animal {
}

class Box<T> {
}

class A {
  static <T> void foo(Box<List<T>> box) {
  }
}
```

   Which of the following compiles?

```
A.<Animal>foo(new Box<List<Animal>>());
A.<Animal>foo(new Box<List<Dog>>());
A.<Animal>foo(new Box<ArrayList<Animal>>());
A.<Animal>foo(new Box<ArrayList<Dog>>());
```

   Notes for Tutors:

```
A.<Animal>foo(new Box<List<Animal>>());      // ok
A.<Animal>foo(new Box<List<Dog>>());         // not ok
A.<Animal>foo(new Box<ArrayList<Animal>>()); // not ok
A.<Animal>foo(new Box<ArrayList<Dog>>());    // not ok
```

   (b) Suppose we change `foo` to:

```
class A {
  static <T> void foo(Box<? extends List<T>> box) {
  }
}
```

   Which of the following compiles?

```
A.<Animal>foo(new Box<List<Animal>>());
A.<Animal>foo(new Box<List<Dog>>());
A.<Animal>foo(new Box<ArrayList<Animal>>());
A.<Animal>foo(new Box<ArrayList<Dog>>());
```

Notes for Tutors:

```
A.<Animal>foo(new Box<List<Animal>>());      // ok
A.<Animal>foo(new Box<List<Dog>>());         // not ok
A.<Animal>foo(new Box<ArrayList<Animal>>()); // ok
A.<Animal>foo(new Box<ArrayList<Dog>>());    // not ok
```

(c) Suppose we change `foo` to:

```
class A {
  static <T> void foo(Box<? extends List<? extends T>> box) {  }
}
```

Which of the following compiles?

```
A.<Animal>foo(new Box<List<Animal>>());
A.<Animal>foo(new Box<List<Dog>>());
A.<Animal>foo(new Box<ArrayList<Animal>>());
A.<Animal>foo(new Box<ArrayList<Dog>>());
```

Notes for Tutors:

```
A.<Animal>foo(new Box<List<Animal>>());      // ok
A.<Animal>foo(new Box<List<Dog>>());         // ok
A.<Animal>foo(new Box<ArrayList<Animal>>()); // ok
A.<Animal>foo(new Box<ArrayList<Dog>>());    // ok
```

2. **Anonymous Class**

Suppose we have a class `AddK` that is used only once and never again.

```
class AddK implements Transformer<Integer, Integer> {
  int k;
  AddK(int k) {
    this.k = k;
  }
```

```
  @Override
  public Integer transform(Integer t) {
    return t + k;
  }
}
```

`AddK` is used as follows:

```
Box.of(4).map(new AddK(3));
```

Rewrite the class as an anonymous class.

Notes for Tutors:

```
Box.of(4).map(
  new Transformer<Integer, Integer>() {
    @Override
    public Integer transform(Integer t) {
      return t + 3;
    }
  }
)
```

For the next two questions, you need to copy the directory `~cs2030s/lab8` to your working directory and edit the files.

3. **Nested Class**

Consider the following simplified version of `Box<T>`.

```
class Box<T> {
  private final T t;
  private static final Box<?> EMPTY = new Box<>(null);

  private Box(T t) {
    this.t = t;
  }

  public static <T> Box<T> empty() {
    @SuppressWarnings("unchecked")
    Box<T> box = (Box<T>) EMPTY;
    return box;
  }

  public static <T> Box<T> ofNullable(T t) {
    if (t != null) {
      return (Box<T>) new Box<>(t);
    }
    return empty();
  }

  public boolean isPresent() {
    if (this.t != null) {
      return false;
    }
    return true;
  }

  public Box<T> filter(BooleanCondition<? super T> condition) {
    if (this.t != null) {
      if (condition.test(this.t) == false) {
        return empty();
      }
      return (Box<T>) this;
    }
    return empty();
  }

  @Override
  public String toString() {
    if (this.t != null) {
      return "[" + t + "]";
    }
    return "[]";
  }
}
```

Observe that for most of the methods, we need to cater to two cases

```
if (this.t != null) {
  // box not empty
  // do something to t
} else {
  // box is empty
```

```
  // handle case where t is null
}
```

Whenever we add a method to `Box<T>`, we need to remember to handle both cases, for the case when `t` is `null`, and when `t` is not `null`. There is a better way to do this – by exploiting abstract classes, dynamic binding, and nested classes.

We are going to re-implement `Box<T>` using an idiom that (i) separates the logic for handling the empty and non-empty box into two nested classes, and (ii) makes `Box` and its methods abstract. This approach forces us to handle both cases explicitly.

(a) Make `Box<T>` an abstract class and make all its instance methods abstract.

(b) Create a private nested class for `Box<T>` called `NonEmpty<T>` that inherits from `Box<T>`. Provide the concrete implementation for the three abstract methods for `Box<T>` that corresponds to the non-empty box. The field `t` should be moved to this class.

(c) Create a private nested class for `Box<T>` called `Empty` that inherits from `Box<Object>`. Provide the concrete implementation for the three abstract methods for `Box<T>` that corresponds to the empty box.

(d) Change the factory methods of `Box<T>` so that it returns an instance of `Empty` or `NonEmpty<T>` appropriately.

```
abstract class Box<T> {

  private static final Box<?> EMPTY = new Empty();

  public static <T> Box<T> empty() {
    @SuppressWarnings("unchecked")
    Box<T> box = (Box<T>) EMPTY;
    return box;
  }

  public static <T> Box<T> ofNullable(T t) {
    if (t != null) {
      return new NonEmpty<T>(t);
    }
    return empty();
  }

  public abstract boolean isPresent();
  public abstract Box<T> filter(BooleanCondition<? super T> condition);
  public abstract String toString();

  private static class Empty extends Box<Object> {
    @Override
    public boolean isPresent() {
      return false;
    }

    @Override
    public Empty filter(BooleanCondition<? super Object> condition) {
      return this;
    }

    @Override
    public String toString() {
      return "[]";
```

```
    }
  }

  private static class NonEmpty<T> extends Box<T> {
    private final T t;

    private NonEmpty(T t) {
      this.t = t;
    }

    @Override
    public boolean isPresent() {
      return true;
    }

    @Override
    public Box<T> filter(BooleanCondition<? super T> condition) {
      if (condition.test(this.t) == false) {
        return empty();
      }
      return (Box<T>) this;
    }

    @Override
    public String toString() {
      return "[" + t + "]";
    }
  }
}
```

Make sure that your code compiles and behaves as follows:

```
jshell> /open BooleanCondition.java
jshell> /open IsPositive.java
jshell> /open Box.java

jshell> Box.ofNullable(6)
$.. ==> [6]

jshell> Box<Integer> box;
$.. ==> null

jshell> box = Box.ofNullable(null)
$.. ==> []
jshell> box.isPresent()
$.. ==> false
jshell> box.filter(new IsPositive())
$.. ==> []

jshell> box = Box.ofNullable(8)
$.. ==> [8]
jshell> box.isPresent()
$.. ==> true
jshell> box.filter(new IsPositive())
$.. ==> [8]

jshell> box = Box.ofNullable(-4)
$.. ==> [-4]
```

```
jshell> box.isPresent()
$.. ==> true
jshell> box.filter(new IsPositive())
$.. ==> []

jshell> Box.ofNullable(6).filter(new IsPositive())
$.. ==> [6]
jshell> Box.ofNullable(null).filter(new IsPositive())
|  Error:
|  incompatible types: IsPositive cannot be converted to BooleanCondition<? super java.lan
|  Box.ofNullable(null).filter(new IsPositive())
|                              ^--------------^
```

(e) What is the compile-time type of the expression `Box.ofNullable(4)`? What is the run-time type of the expression `Box.ofNullable(4)`?

Notes for Tutors:
`Box<Integer>` and `NonEmpty<Integer>`

(f) Explain why the following would lead to a compilation error.

```
Box.ofNullable(null).filter(new IsPositive())
```

Notes for Tutors:
`T` is inferred as `Object` from `Box.ofNullable(null)` and so the type of the expression `Box.ofNullable(null)` is `Box<Object>`. `filter` now expects an argument that is compatible with `Box<? super Object>`, which `IsPositive` is not.

(g) Show how you can remove the compilation error by specifying the type argument of the generic method `ofNullable` explicitly.

Notes for Tutors:
`Box.<Integer>ofNullable(null).filter(new IsPositive())`

4. **Package**

You have now seen that we do not necessarily have a single layer of abstraction barrier, but we can create multiple such barriers at different granularity. Nested classes allow us to add another layer of abstraction barrier to the implementation details within a class. We can also add another layer of abstraction barrier across different classes, by grouping related classes into a *package* in Java.

A `package` also helps us to manage the namespace of classes. Every package has a name using hierarchical dot notation (e.g., `com.google.common.math`, `java.io`). We have not been using packages in CS2030s. If we do not specify the package that a class belongs to, it belongs to the *default* package by default.

we can control whether a field/method/class is accessible outside a package. Without any access modifier, a field/method is accessible by any class within the package only. With the 'protected' modifier, a field/method is accessible by any class within the package and outside the package through inheritance.

(a) We will now create a package that we will use for the rest of the semester. We call the package `cs2030s.fp`. Java looks for the classes in a package under the directory whose name mirrors the hierarchy of the package. The classes belonging to the package `cs2030s.fp` should be located under the directory `cs2030s/fp`.

Create a subdirectory `cs2030s/fp` under your current directory.

(b) For a start, our package contains a single class `BooleanCondition.java`. You will add more classes to this package in Exercise 5.

Move the file `BooleanCondition.java` to the subdirectory `cs2030s/fp`.

(c) Tell Java that `BooleanCondition` is part of a package. Add the line

```
package cs2030s.fp;
```

as the first line of `BooleanCondition.java`.

(d) Classes and interfaces inside a package can be public (accessible outside the package) or private (used within the package only). We wish to use `BooleanCondition` outside of the package. Let's make the class accessible from outside the package by adding the access modifier 'public' to the declaration:

```
public interface BooleanCondition<T> {  ... }
```

(e) we can now use `cs2030s.fp.BooleanCondition` in our `Box<T>`. To avoid typing its full name, import it at the top of `Box.java`, and add this line:

```
import cs2030s.fp.BooleanCondition;
```