```java
public abstract class Maybe<T> {
  private static final Maybe<?> NONE = new None();
```

This has to be static and have a wildcard

can only access static fields and static methods of the containing class

```java
  static class None extends Maybe<Object> {
    @Override
```

↳ I need it to be typecasted to any maybe.

```java
    public Maybe<Object> filter(BooleanCondition<? super Object> test) {
      return none();
      //                              E;
    }
```

Don't forget the @Override annotation

```java
    @Override
    public <R> Maybe<R> map(Transformer<? super Object, ? extends R> transformer) {
      return none();
    }
```

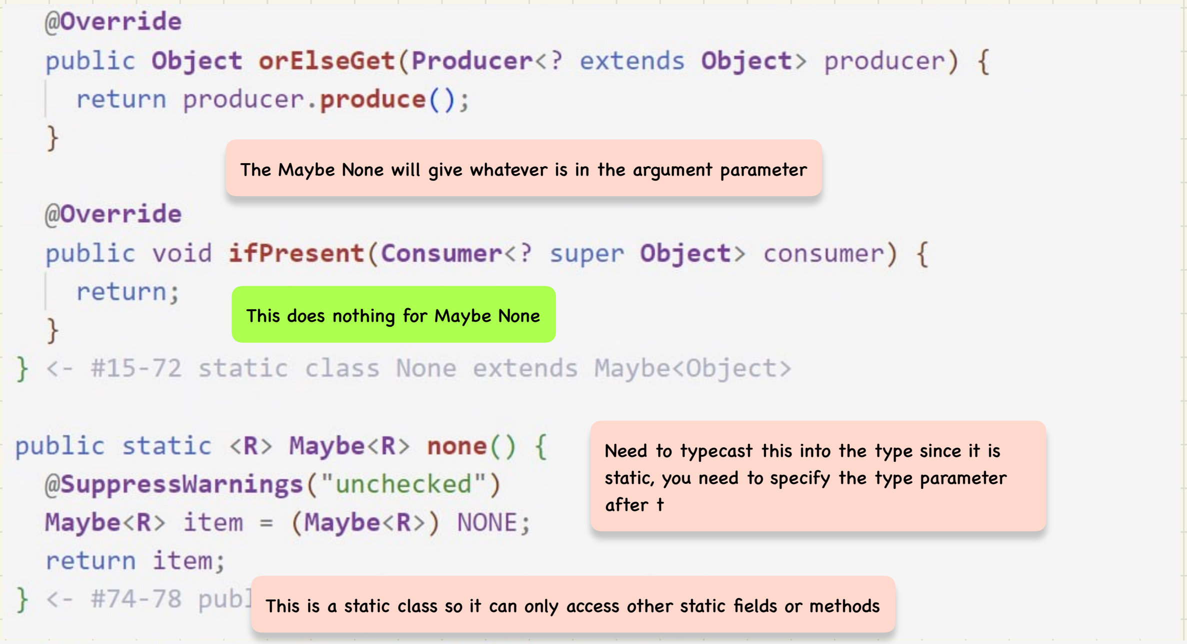
Return type is the same as the output of the transformer

```java
    @Override
    public <R> Maybe<R> flatMap(Transformer<? super Object,
        ? extends Maybe<? extends R>> transformer) {
      return none();
    }
```

This is the same as ArrayList<Dog> <: List<Animal> argument

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

  @Override
  protected Maybe<Object> get() {
    throw new NoSuchElementException();
  }

  @Override
  public boolean equals(Object obj) {
    if (obj == this) {
      return true;
    }

    if (obj instanceof Maybe<?>) {
      if (obj == NONE) {
        return true;
      }
    } <- #49-53 if (obj instanceof Maybe<?>)
    return false;

  } <- #44-56 public boolean equals(Object obj)

  @Override
  public Object orElse(Object s) {
    return s;
  }
```

There is nothing in Maybe None so it will return the argument in the parameter



```java
  @Override
  public Object orElseGet(Producer<? extends Object> producer) {
    return producer.produce();
  }
```

The Maybe None will give whatever is in the argument parameter

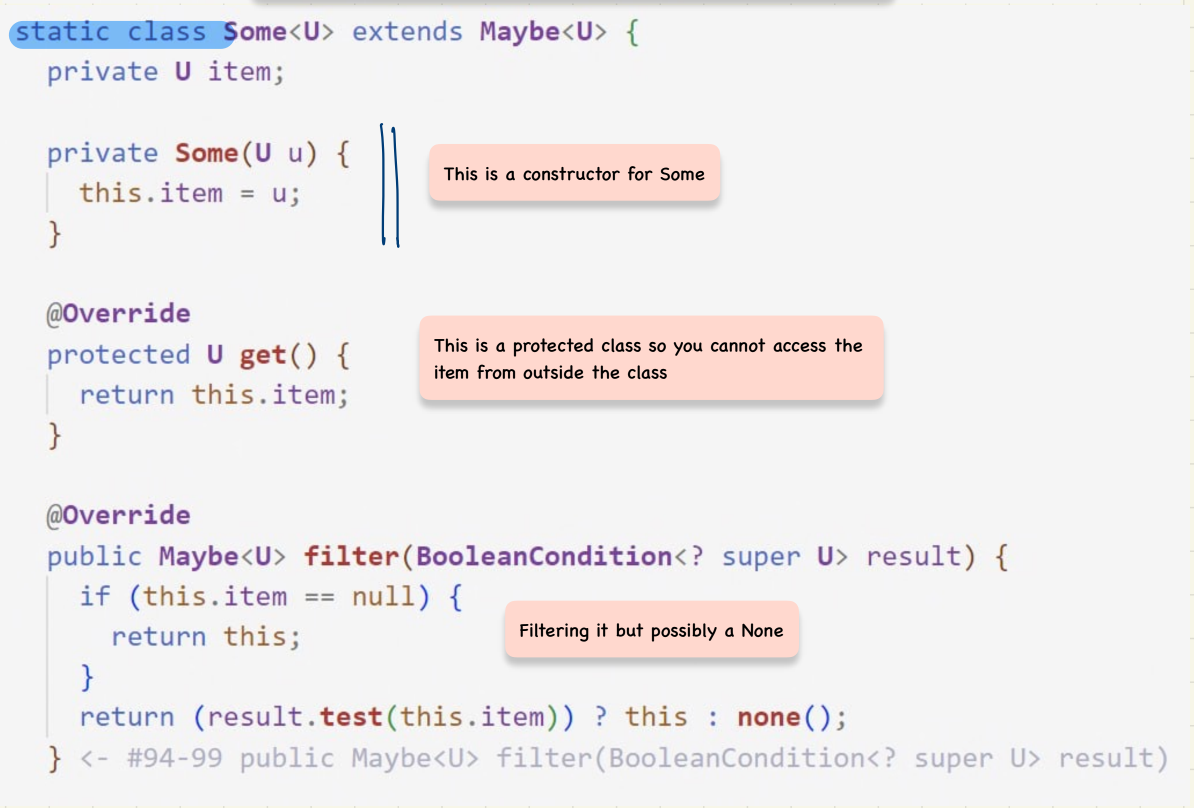
```java
  @Override
  public void ifPresent(Consumer<? super Object> consumer) {
    return;
  }
} <- #15-72 static class None extends Maybe<Object>
```

This does nothing for Maybe None

```java
public static <R> Maybe<R> none() {
  @SuppressWarnings("unchecked")
  Maybe<R> item = (Maybe<R>) NONE;
  return item;
} <- #74-78 publ
```

Need to typecast this into the type since it is static, you need to specify the type parameter after t

This is a static class so it can only access other static fields or methods

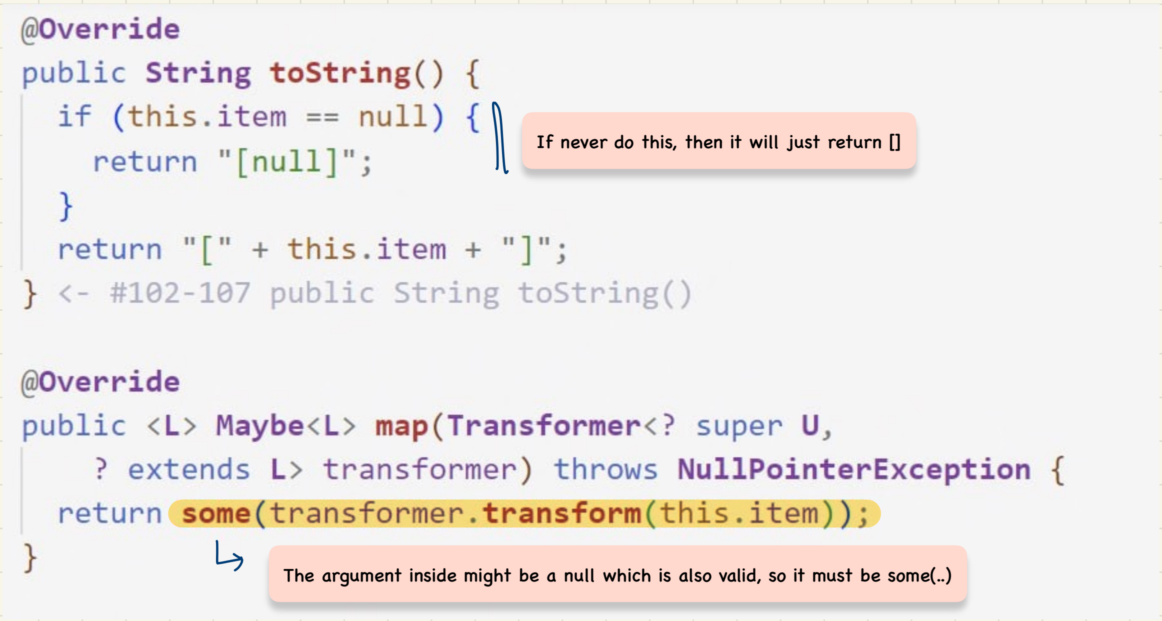


```java
static class Some<U> extends Maybe<U> {
  private U item;

  private Some(U u) {
    this.item = u;
  }

  @Override
  protected U get() {
    return this.item;
  }

  @Override
  public Maybe<U> filter(BooleanCondition<? super U> result) {
    if (this.item == null) {
      return this;
    }
    return (result.test(this.item)) ? this : none();
  } <- #94-99 public Maybe<U> filter(BooleanCondition<? super U> result)
```

This is a constructor for Some

This is a protected class so you cannot access the item from outside the class

Filtering it but possibly a None

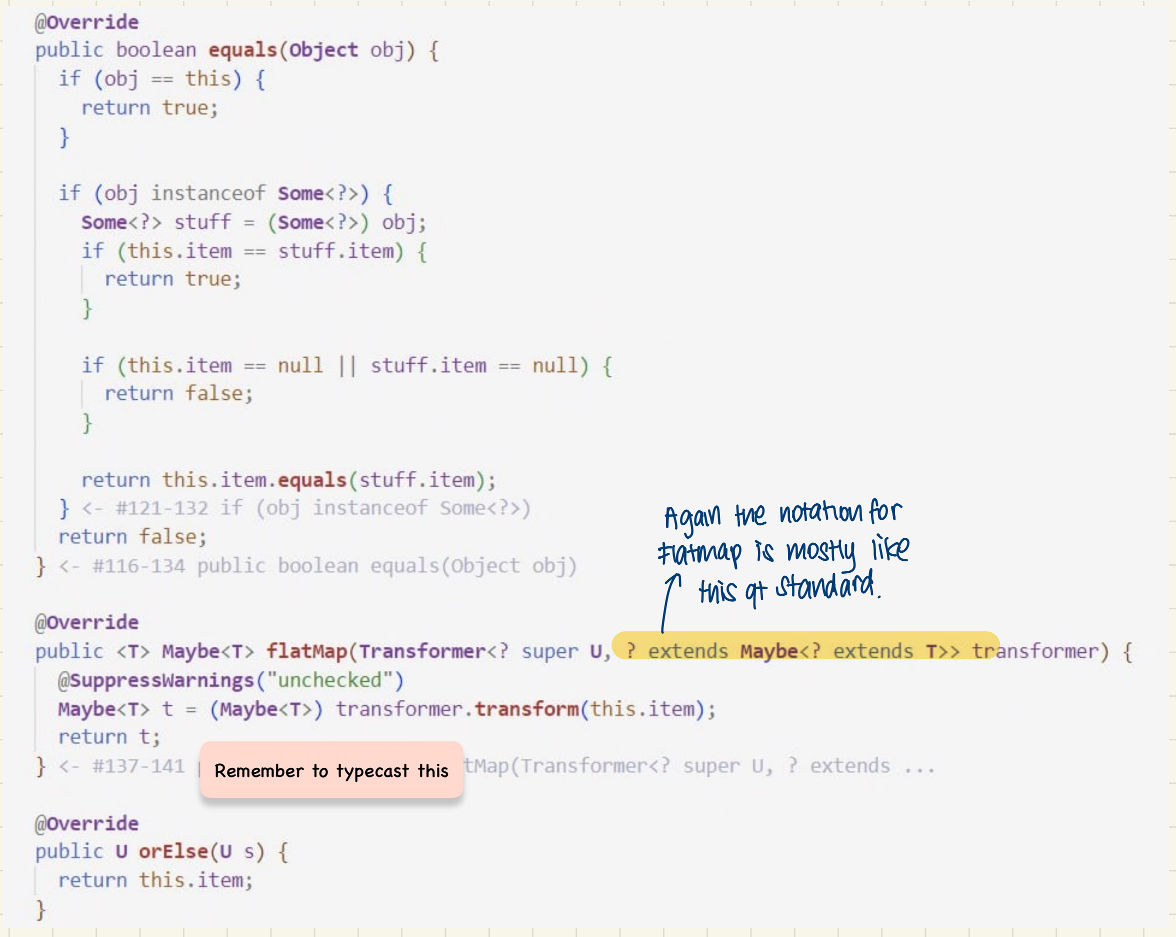


```java
@Override
public String toString() {
  if (this.item == null) {
    return "[null]";
  }
  return "[" + this.item + "]";
} <- #102-107 public String toString()
```

If never do this, then it will just return []

```java
@Override
public <L> Maybe<L> map(Transformer<? super U,
    ? extends L> transformer) throws NullPointerException {
  return some(transformer.transform(this.item));
}
```

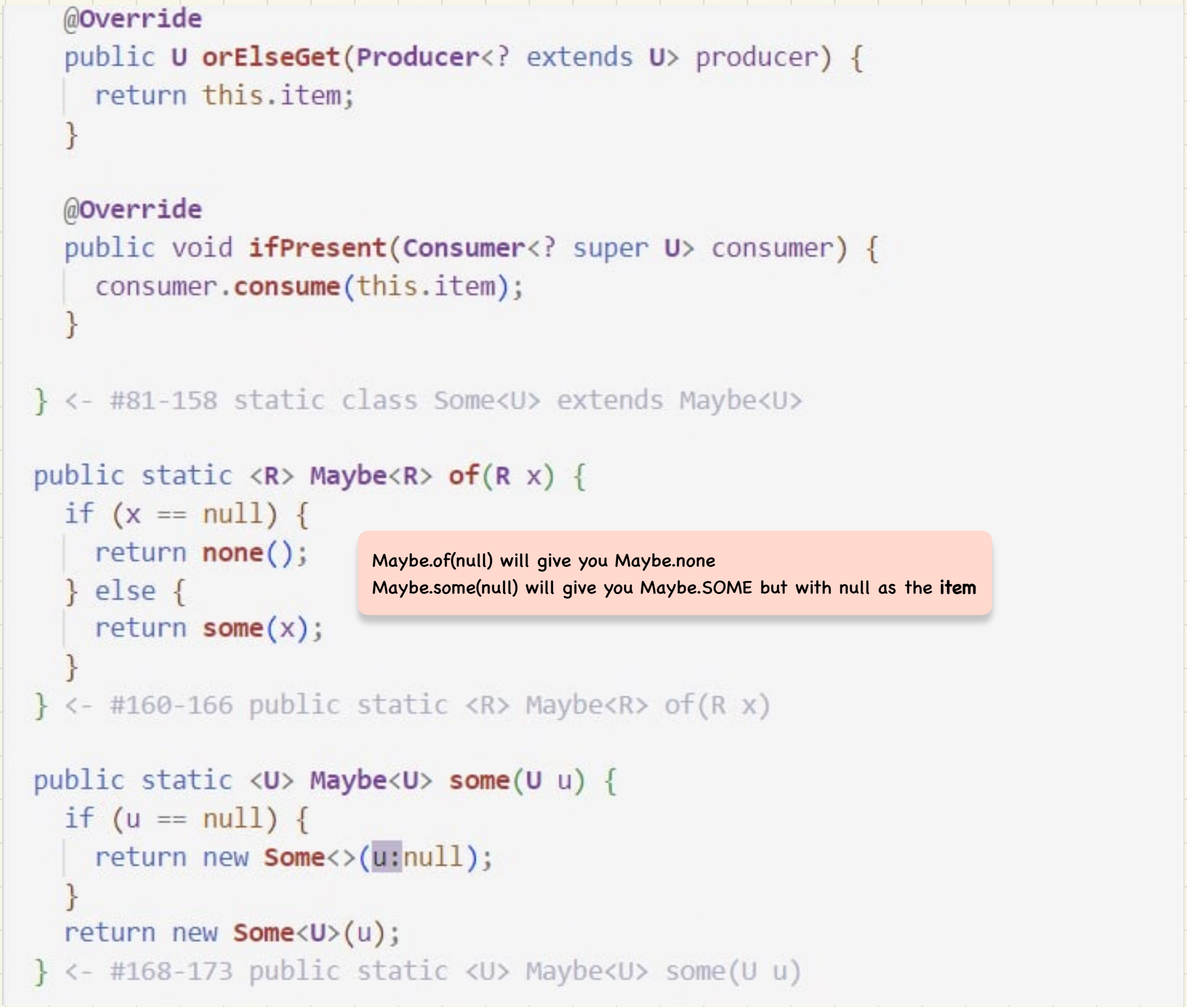
The argument inside might be a null which is also valid, so it must be some(..)



```java
@Override
public boolean equals(Object obj) {
  if (obj == this) {
    return true;
  }

  if (obj instanceof Some<?>) {
    Some<?> stuff = (Some<?>) obj;
    if (this.item == stuff.item) {
      return true;
    }

    if (this.item == null || stuff.item == null) {
      return false;
    }

    return this.item.equals(stuff.item);
  } <- #121-132 if (obj instanceof Some<?>)
  return false;
} <- #116-134 public boolean equals(Object obj)

@Override
public <T> Maybe<T> flatMap(Transformer<? super U, ? extends Maybe<? extends T>> transformer) {
  @SuppressWarnings("unchecked")
  Maybe<T> t = (Maybe<T>) transformer.transform(this.item);
  return t;
} <- #137-141 tMap(Transformer<? super U, ? extends ...

@Override
public U orElse(U s) {
  return this.item;
}
```

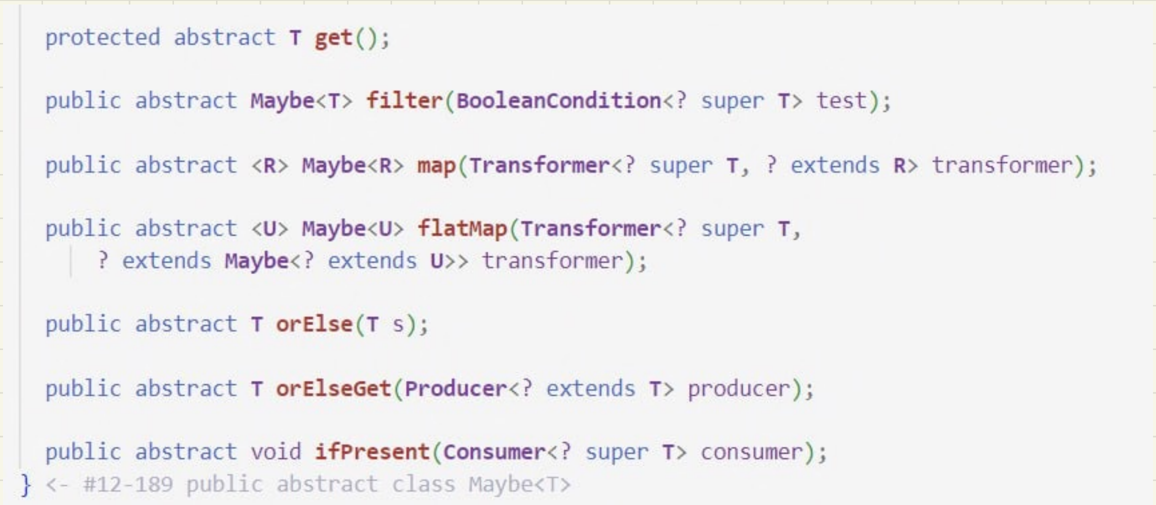
Again the notation for flatmap is mostly like this at standard.

Remember to typecast this



```java
    @Override
    public U orElseGet(Producer<? extends U> producer) {
      return this.item;
    }

    @Override
    public void ifPresent(Consumer<? super U> consumer) {
      consumer.consume(this.item);
    }

  } <- #81-158 static class Some<U> extends Maybe<U>

  public static <R> Maybe<R> of(R x) {
    if (x == null) {
      return none();
    } else {
      return some(x);
    }
  } <- #160-166 public static <R> Maybe<R> of(R x)

  public static <U> Maybe<U> some(U u) {
    if (u == null) {
      return new Some<>(u:null);
    }
    return new Some<U>(u);
  } <- #168-173 public static <U> Maybe<U> some(U u)
```

Maybe.of(null) will give you Maybe.none
Maybe.some(null) will give you Maybe.SOME but with null as the **item**



```java
  protected abstract T get();

  public abstract Maybe<T> filter(BooleanCondition<? super T> test);

  public abstract <R> Maybe<R> map(Transformer<? super T, ? extends R> transformer);

  public abstract <U> Maybe<U> flatMap(Transformer<? super T,
      ? extends Maybe<? extends U>> transformer);

  public abstract T orElse(T s);

  public abstract T orElseGet(Producer<? extends T> producer);

  public abstract void ifPresent(Consumer<? super T> consumer);
} <- #12-189 public abstract class Maybe<T>
```