```
class MergeSort {
  xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
  xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
  xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
  xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
  xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
  xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
  xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
  xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
  xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
  xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
  xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
  xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
  xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
}
```

Miss Pumpkin’s solution is shown step by step

```java
class MergeSort {
  public <T extends Comparable<? super T>> List<T> sort(List<T> c) {
    List<T> result = new ArrayList<>(c.size());
    sorted(c).forEachRemaining(e -> result.add(e));
    return Collections.unmodifiableList(result);
  }

  <T extends Comparable<? super T>> PeekIterator<T> sorted(List<T> input) {
    ..
    ..
    ..
    ..
    ..
    ..
  }
}
```

```java
class MergeSort {
  public <T extends Comparable<? super T>> List<T> sort(List<T> c) {
    List<T> result = new ArrayList<>(c.size());
    sorted(c).forEachRemaining(e -> result.add(e));
    return Collections.unmodifiableList(result);
  }

  <T extends Comparable<? super T>> PeekIterator<T> sorted(List<T> input) {
    ..
    ..
    ..
    ..
    ..
    ..
  }
}
```

```java
class MergeSort {
  public <T extends Comparable<? super T>> List<T> sort(List<T> c) {
    List<T> result = new ArrayList<>(c.size());
    sorted(c).forEachRemaining(e -> result.add(e));
    return Collections.unmodifiableList(result);
  }

  <T extends Comparable<? super T>> PeekIterator<T> sorted(List<T> input) {
    ..
    ..
    ..
    ..
    ..
    ..
  }
}
```

```java
class MergeSort {
  public <T extends Comparable<? super T>> List<T> sort(List<T> c) {
    List<T> result = new ArrayList<>(c.size());
    sorted(c).forEachRemaining(e -> result.add(e));
    return Collections.unmodifiableList(result);
  }

  <T extends Comparable<? super T>> PeekIterator<T> sorted(List<T> input) {
    ..
    ..
    ..
    ..
    ..
    ..
  }
}
```

The sort method looks standard

```java
class MergeSort {
  public <T extends Comparable<? super T>> List<T> sort(List<T> c) {
    List<T> result = new ArrayList<>(c.size());
    sorted(c).forEachRemaining(e -> result.add(e));
    return Collections.unmodifiableList(result);
  }

  <T extends Comparable<? super T>> PeekIterator<T> sorted(List<T> input) {
    ..
    ..
    ..
    ..
    ..
    ..
  }
}
```

The sort method looks standard

It allocates space for the result
and delegates to ‘sorted’

```
class MergeSort {
  public <T extends Comparable<? super T>> List<T> sort(List<T> c) {
    List<T> result = new ArrayList<>(c.size());
    sorted(c).forEachRemaining(e -> result.add(e));
    return Collections.unmodifiableList(result);
  }

  <T extends Comparable<? super T>> PeekIterator<T> sorted(List<T> input) {
    ..
    ..
    ..
    ..
    ..
    ..
  }
}
```

The sort method looks standard

It allocates space for the result
and delegates to ‘sorted’

‘sorted’ returns an iterator

```
class MergeSort {
  public <T extends Comparable<? super T>> List<T> sort(List<T> c) {
    List<T> result = new ArrayList<>(c.size());
    sorted(c).forEachRemaining(e -> result.add(e));
    return Collections.unmodifiableList(result);
  }

  <T extends Comparable<? super T>> PeekIterator<T> sorted(List<T> input) {
    ..
    ..
    ..
    ..
    ..
    ..
  }
}
```

The sort method looks standard

It allocates space for the result
and delegates to ‘sorted’

‘sorted’ returns an iterator

so we use forEachRemaining
to extract the sorted data

```java
class MergeSort {
  public <T extends Comparable<? super T>> List<T> sort(List<T> c) {
    List<T> result = new ArrayList<>(c.size());
    sorted(c).forEachRemaining(e -> result.add(e));
    return Collections.unmodifiableList(result);
  }

  <T extends Comparable<? super T>> PeekIterator<T> sorted(List<T> input) {
    ..
    ..
    ..
    ..
    ..
    ..
  }
}
```

```
class MergeSort {
  public <T extends Comparable<? super T>> List<T> sort(List<T> c) {
    List<T> result = new ArrayList<>(c.size());
    sorted(c).forEachRemaining(e -> result.add(e));
    return Collections.unmodifiableList(result);
  }

  <T extends Comparable<? super T>> PeekIterator<T> sorted(List<T> input) {
    ..
    ..
    ..
    ..
    ..
    ..
  }
}
```

PeekIterator?

```java
class MergeSort {
  public <T extends Comparable<? super T>> List<T> sort(List<T> c) {
    List<T> result = new ArrayList<>(c.size());
    sorted(c).forEachRemaining(e -> result.add(e));
    return Collections.unmodifiableList(result);
  }

  <T extends Comparable<? super T>> PeekIterator<T> sorted(List<T> input) {
    ..
    ..
    ..
    ..
    ..
    ..
  }
}
```

PeekIterator?

What is a PeekIterator?
I had no idea it existed

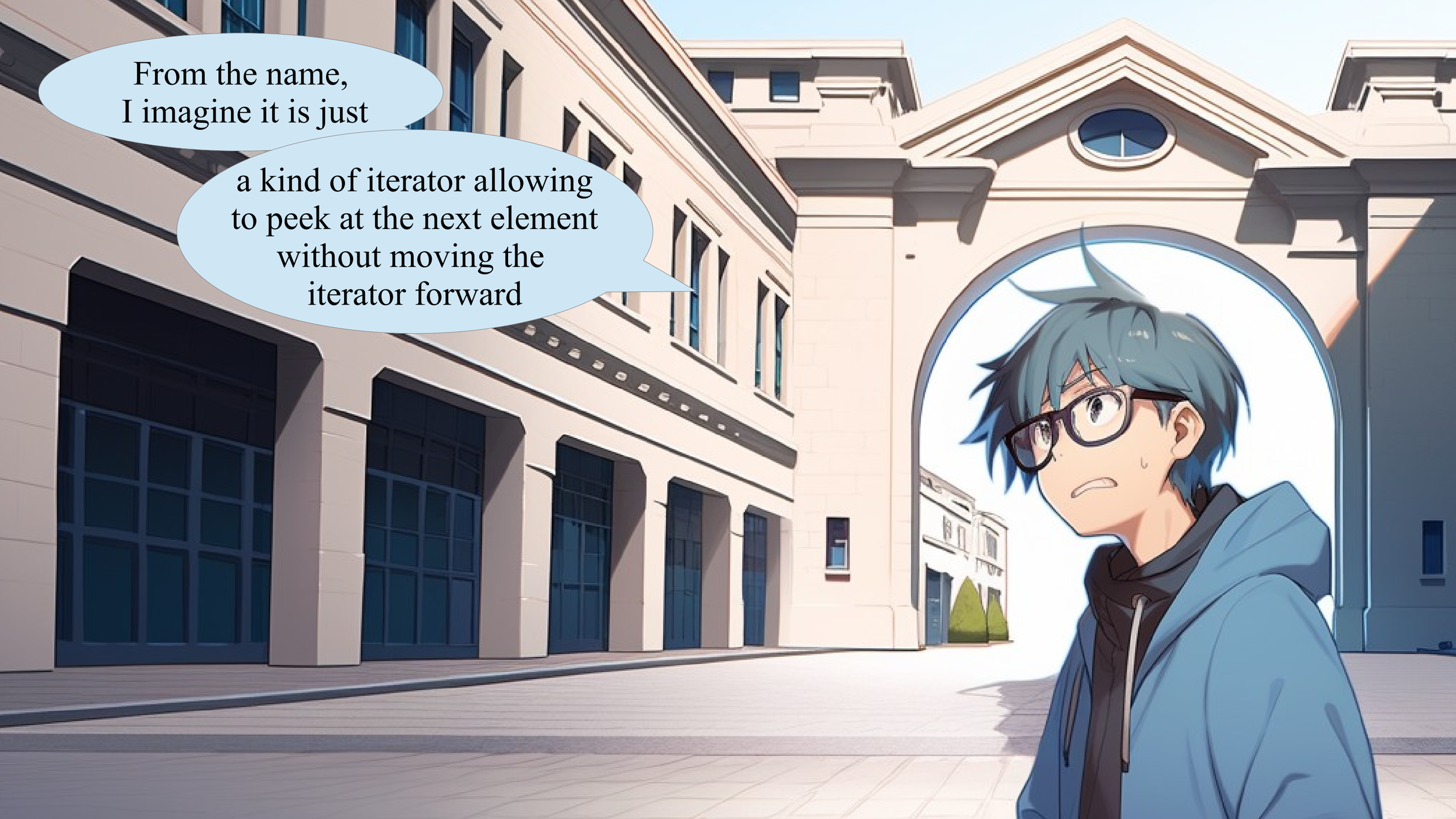
```java
class MergeSort {
  public <T extends Comparable<? super T>> List<T> sort(List<T> c) {
    List<T> result = new ArrayList<>(c.size());
    sorted(c).forEachRemaining(e -> result.add(e));
    return Collections.unmodifiableList(result);
  }

  <T extends Comparable<? super T>> PeekIterator<T> sorted(List<T> input) {
    ..
    ..
    ..
    ..
    ..
    ..
  }
}
```

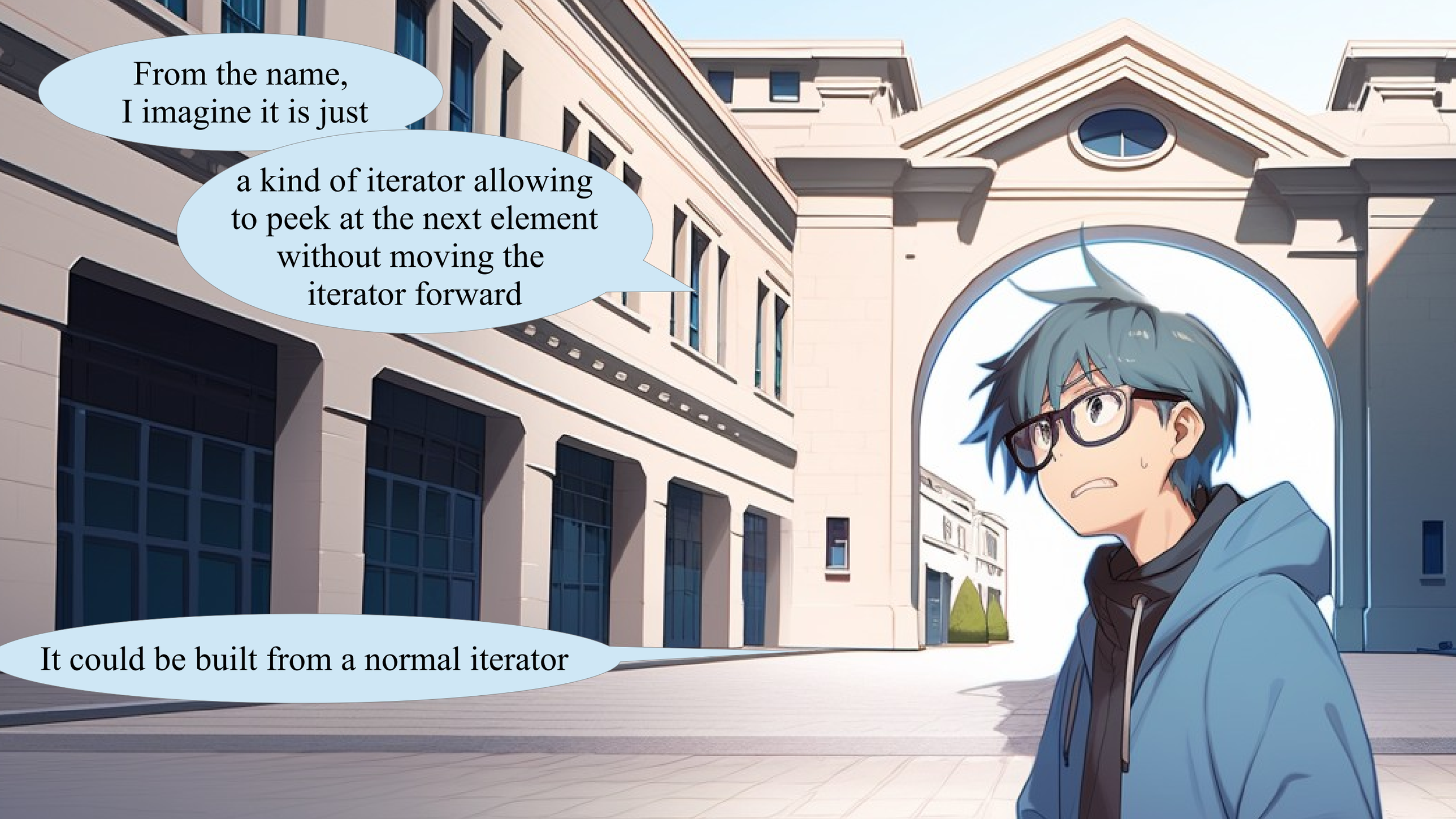
PeekIterator?

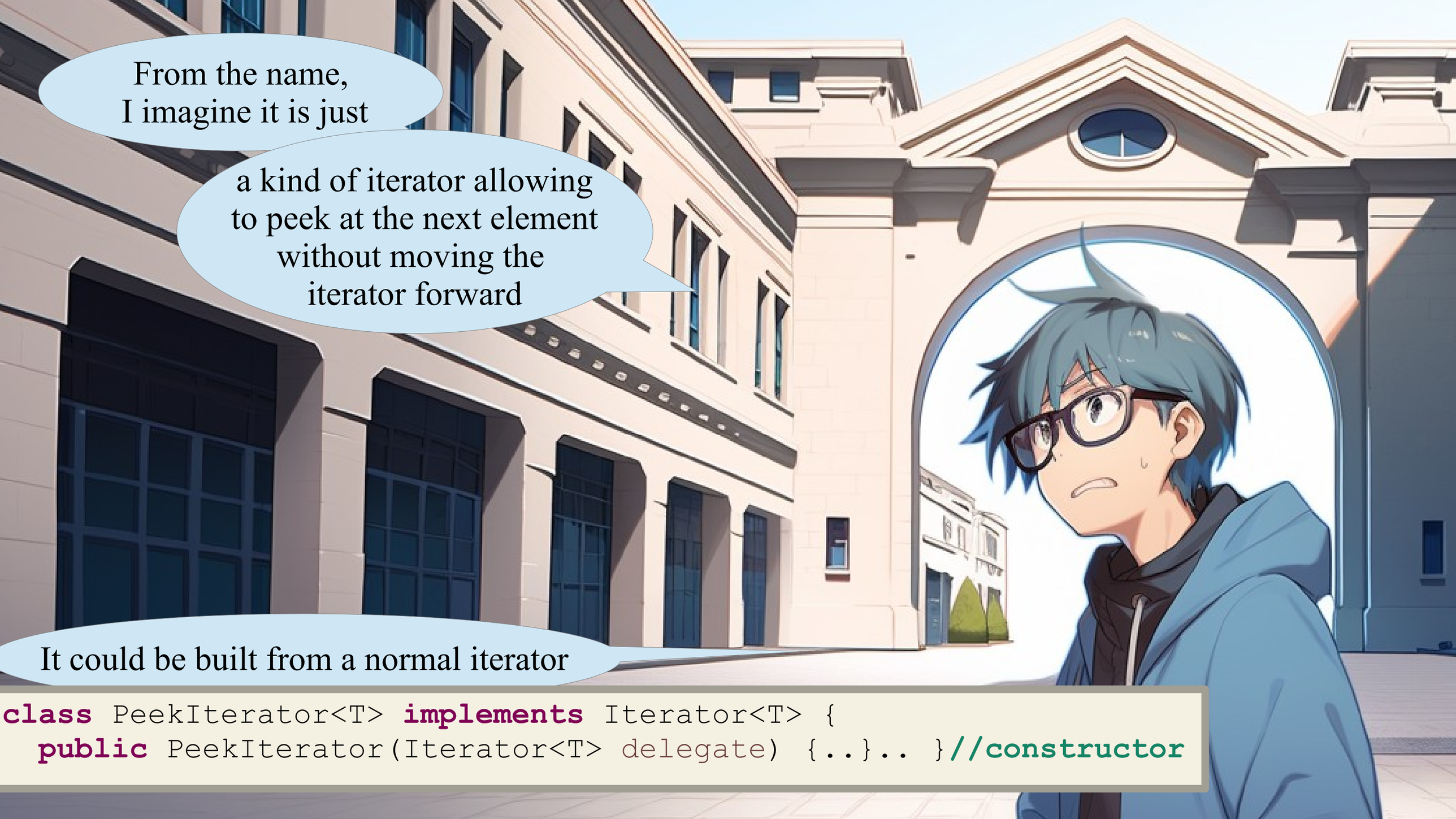
What is a PeekIterator?
I had no idea it existed

Is there a limit to the breadth
of the Java standard library?

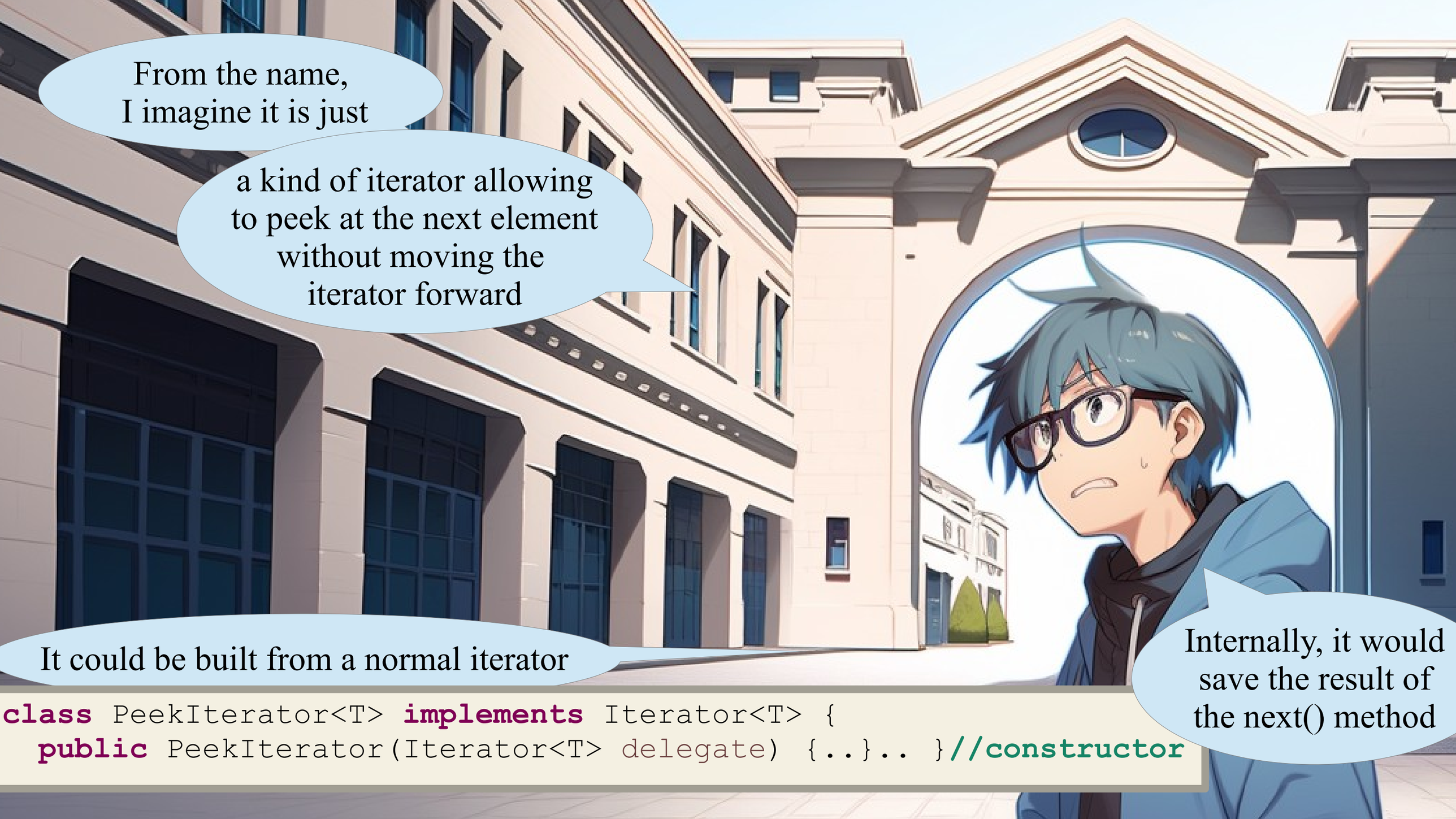

From the name,
I imagine it is just


From the name,
I imagine it is just
a kind of iterator allowing
to peek at the next element
without moving the
iterator forward


From the name,
I imagine it is just
a kind of iterator allowing to peek at the next element without moving the iterator forward
It could be built from a normal iterator




```
class PeekIterator<T> implements Iterator<T> {
  public PeekIterator(Iterator<T> delegate) {..}.. }//constructor
```




```
class PeekIterator<T> implements Iterator<T> {
  public PeekIterator(Iterator<T> delegate) {..}.. }//constructor
```

```java
class MergeSort {
  public <T extends Comparable<? super T>> List<T> sort(List<T> c) {
    List<T> result = new ArrayList<>(c.size());
    sorted(c).forEachRemaining(e -> result.add(e));
    return Collections.unmodifiableList(result);
  }

  <T extends Comparable<? super T>> PeekIterator<T> sorted(List<T> input) {
    int size = input.size();
    if (size < 2){ return new PeekIterator<>(input.iterator()); }
    int mid = size / 2;
    PeekIterator<T> left=  sorted(input.subList(0, mid));
    PeekIterator<T> right= sorted(input.subList(mid, size));
    ..
  }
}
```

```java
class MergeSort {
  public <T extends Comparable<? super T>> List<T> sort(List<T> c) {
    List<T> result = new ArrayList<>(c.size());
    sorted(c).forEachRemaining(e -> result.add(e));
    return Collections.unmodifiableList(result);
  }

  <T extends Comparable<? super T>> PeekIterator<T> sorted(List<T> input) {
    int size = input.size();
    if (size < 2){ return new PeekIterator<>(input.iterator()); }
    int mid = size / 2;
    PeekIterator<T> left=  sorted(input.subList(0, mid));
    PeekIterator<T> right= sorted(input.subList(mid, size));
    ..
  }
}
```

The sorted implementation
follows the same strategy as my code

```java
class MergeSort {
  public <T extends Comparable<? super T>> List<T> sort(List<T> c) {
    List<T> result = new ArrayList<>(c.size());
    sorted(c).forEachRemaining(e -> result.add(e));
    return Collections.unmodifiableList(result);
  }

  <T extends Comparable<? super T>> PeekIterator<T> sorted(List<T> input) {
    int size = input.size();
    if (size < 2){ return new PeekIterator<>(input.iterator()); }
    int mid = size / 2;
    PeekIterator<T> left=  sorted(input.subList(0, mid));
    PeekIterator<T> right= sorted(input.subList(mid, size));
    ..
  }
}
```

The sorted implementation follows the same strategy as my code

Lists of one or zero elements are intrinsically sorted

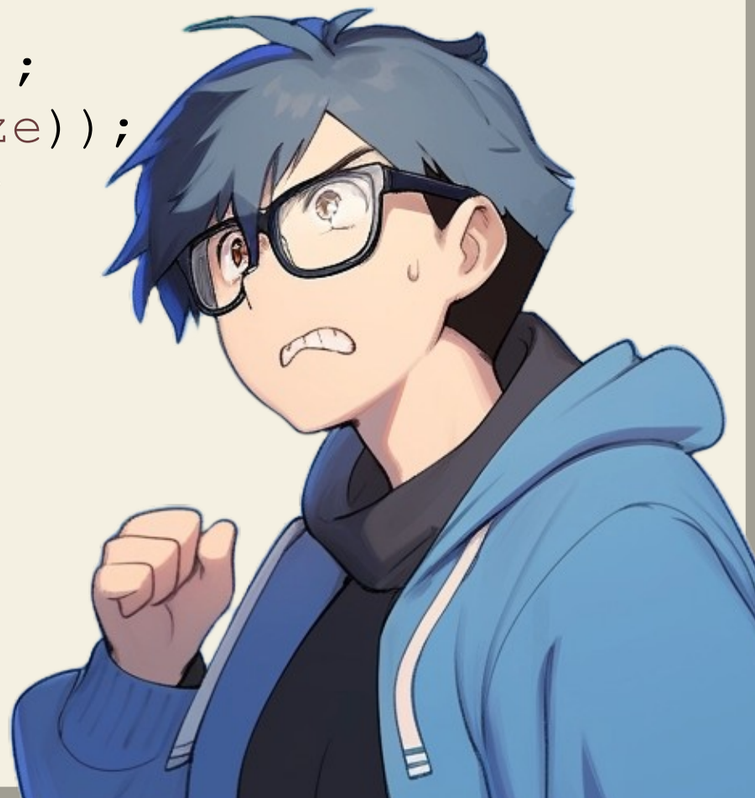
```java
class MergeSort {
  public <T extends Comparable<? super T>> List<T> sort(List<T> c) {
    List<T> result = new ArrayList<>(c.size());
    sorted(c).forEachRemaining(e -> result.add(e));
    return Collections.unmodifiableList(result);
  }

  <T extends Comparable<? super T>> PeekIterator<T> sorted(List<T> input) {
    int size = input.size();
    if (size < 2){ return new PeekIterator<>(input.iterator()); }
    int mid = size / 2;
    PeekIterator<T> left=  sorted(input.subList(0, mid));
    PeekIterator<T> right= sorted(input.subList(mid, size));
    ..
  }
}
```

The sorted implementation follows the same strategy as my code

Lists of one or zero elements are intrinsically sorted

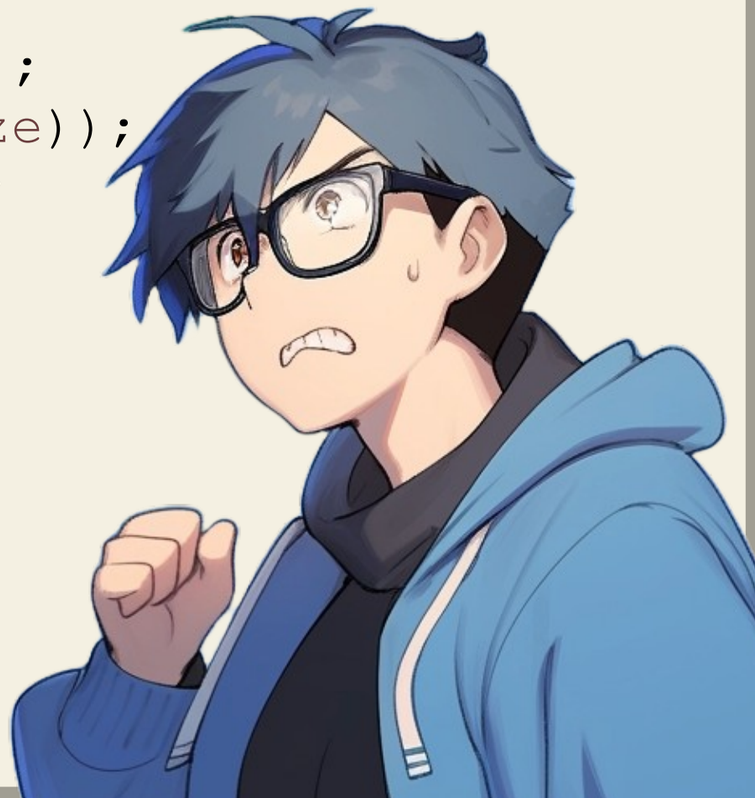
Otherwise, we split the list in two and sort the two halves

```java
class MergeSort {
  public <T extends Comparable<? super T>> List<T> sort(List<T> c) {
    List<T> result = new ArrayList<>(c.size());
    sorted(c).forEachRemaining(e -> result.add(e));
    return Collections.unmodifiableList(result);
  }

  <T extends Comparable<? super T>> PeekIterator<T> sorted(List<T> input) {
    int size = input.size();
    if (size < 2){ return new PeekIterator<>(input.iterator()); }
    int mid = size / 2;
    PeekIterator<T> left=  sorted(input.subList(0, mid));
    PeekIterator<T> right= sorted(input.subList(mid, size));
    return new PeekIterator<>(new Merge<>(left, right));
  }
}
```



```java
class MergeSort {
  public <T extends Comparable<? super T>> List<T> sort(List<T> c) {
    List<T> result = new ArrayList<>(c.size());
    sorted(c).forEachRemaining(e -> result.add(e));
    return Collections.unmodifiableList(result);
  }

  <T extends Comparable<? super T>> PeekIterator<T> sorted(List<T> input) {
    int size = input.size();
    if (size < 2){ return new PeekIterator<>(input.iterator()); }
    int mid = size / 2;
    PeekIterator<T> left=  sorted(input.subList(0, mid));
    PeekIterator<T> right= sorted(input.subList(mid, size));
    return new PeekIterator<>(new Merge<>(left, right));
  }
}
```

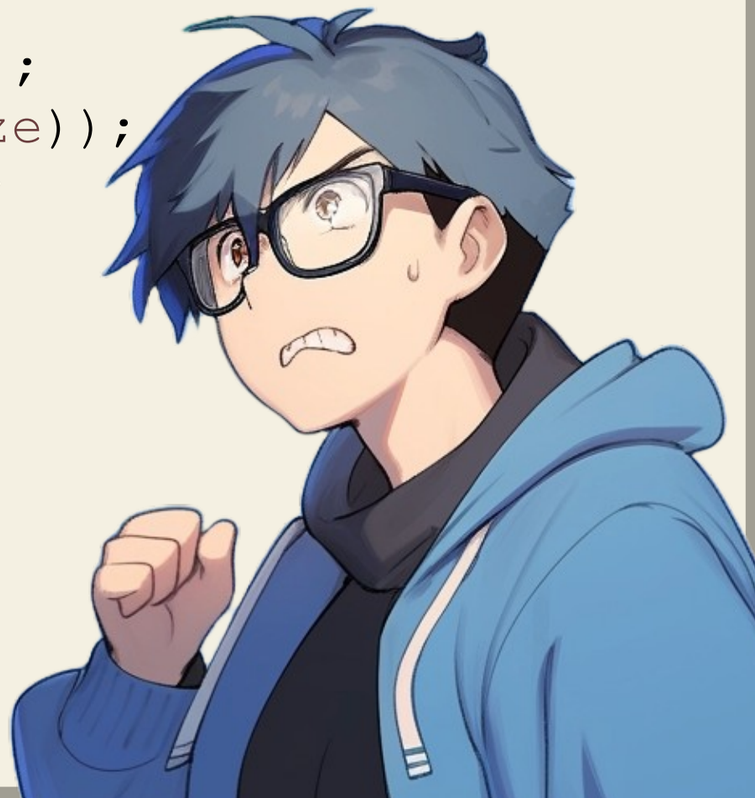
Finally, to merge the two halves..



```java
class MergeSort {
  public <T extends Comparable<? super T>> List<T> sort(List<T> c) {
    List<T> result = new ArrayList<>(c.size());
    sorted(c).forEachRemaining(e -> result.add(e));
    return Collections.unmodifiableList(result);
  }

  <T extends Comparable<? super T>> PeekIterator<T> sorted(List<T> input) {
    int size = input.size();
    if (size < 2){ return new PeekIterator<>(input.iterator()); }
    int mid = size / 2;
    PeekIterator<T> left=  sorted(input.subList(0, mid));
    PeekIterator<T> right= sorted(input.subList(mid, size));
    return new PeekIterator<>(new Merge<>(left, right));
  }
}
```

Finally, to merge the two halves..

What?

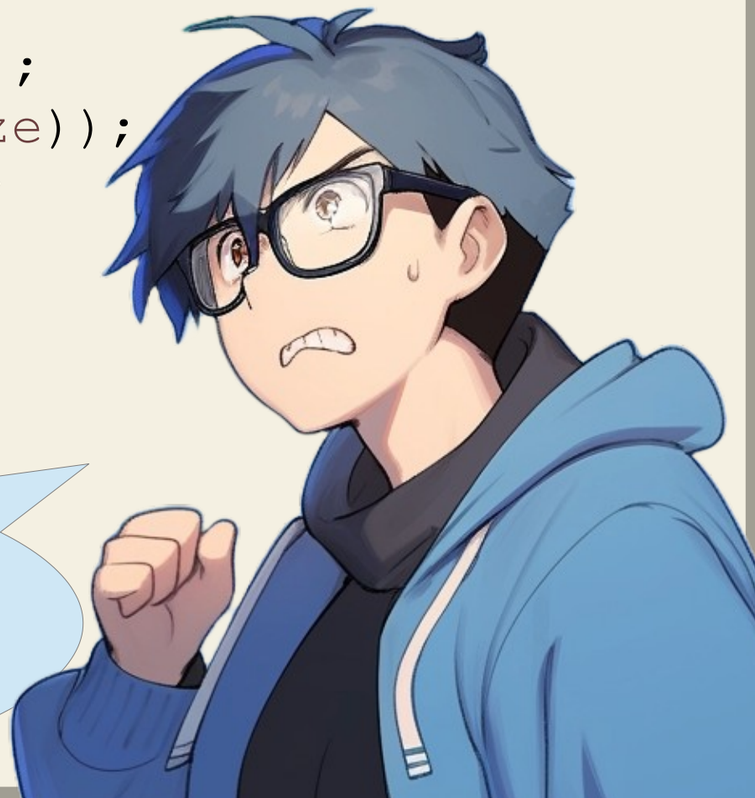


```java
class MergeSort {
  public <T extends Comparable<? super T>> List<T> sort(List<T> c) {
    List<T> result = new ArrayList<>(c.size());
    sorted(c).forEachRemaining(e -> result.add(e));
    return Collections.unmodifiableList(result);
  }

  <T extends Comparable<? super T>> PeekIterator<T> sorted(List<T> input) {
    int size = input.size();
    if (size < 2){ return new PeekIterator<>(input.iterator()); }
    int mid = size / 2;
    PeekIterator<T> left=  sorted(input.subList(0, mid));
    PeekIterator<T> right= sorted(input.subList(mid, size));
    return new PeekIterator<>(new Merge<>(left, right));
  }
}
```

Finally, to merge the two halves..

What?

Merge is also an iterator?
How else could we be building
a PeekIterator from it?



```java
record Merge<T extends Comparable<? super T>>(
    PeekIterator<T> left, PeekIterator<T> right) implements Iterator<T> {

  public boolean hasNext() { return left.hasNext() || right.hasNext(); }

  public T next() {
    if (!left.hasNext()) { return right.next(); }
    if (!right.hasNext()) { return left.next(); }
    boolean leftWin= left.peek().compareTo(right.peek()) <= 0;
    return (leftWin ? left : right).next();
  }
}
```

```java
record Merge<T extends Comparable<? super T>>(
    PeekIterator<T> left, PeekIterator<T> right) implements Iterator<T> {

  public boolean hasNext() { return left.hasNext() || right.hasNext(); }

  public T next() {
    if (!left.hasNext()) { return right.next(); }
    if (!right.hasNext()) { return left.next(); }
    boolean leftWin= left.peek().compareTo(right.peek()) <= 0;
    return (leftWin ? left : right).next();
  }
}
```

The definition of ‘Merge’ is part of the model solution!

```java
record Merge<T extends Comparable<? super T>>(
    PeekIterator<T> left, PeekIterator<T> right) implements Iterator<T> {

  public boolean hasNext() { return left.hasNext() || right.hasNext(); }

  public T next() {
    if (!left.hasNext()) { return right.next(); }
    if (!right.hasNext()) { return left.next(); }
    boolean leftWin= left.peek().compareTo(right.peek()) <= 0;
    return (leftWin ? left : right).next();
  }
}
```

The definition of ‘Merge’ is part of the model solution!

It is user defined code implementing Iterator

```java
record Merge<T extends Comparable<? super T>>(
    PeekIterator<T> left, PeekIterator<T> right) implements Iterator<T> {

  public boolean hasNext() { return left.hasNext() || right.hasNext(); }

  public T next() {
    if (!left.hasNext()) { return right.next(); }
    if (!right.hasNext()) { return left.next(); }
    boolean leftWin= left.peek().compareTo(right.peek()) <= 0;
    return (leftWin ? left : right).next();
  }
}
```

The definition of ‘Merge’ is part of the model solution!

It is user defined code implementing Iterator

Wow!


They asked us to ‘use iterators’


They asked us to ‘use iterators’
And so I got stuck on only using the existing iterators


They asked us to ‘use iterators’
And so I got stuck on only using the existing iterators
Instead of relying on the infinite set of all possible iterators that we could code ourselves!


They asked us to ‘use iterators’
And so I got stuck on only using the existing iterators
Instead of relying on the infinite set of all possible iterators that we could code ourselves!
It feels like I’m still unable to understand what real programming is!

```java
record Merge<T extends Comparable<? super T>>(
    PeekIterator<T> left, PeekIterator<T> right) implements Iterator<T> {

  public boolean hasNext() { return left.hasNext() || right.hasNext(); }

  public T next() {
    if (!left.hasNext()) { return right.next(); }
    if (!right.hasNext()) { return left.next(); }
    boolean leftWin= left.peek().compareTo(right.peek()) <= 0;
    return (leftWin ? left : right).next();
  }
}
```

```java
record Merge<T extends Comparable<? super T>>(
    PeekIterator<T> left, PeekIterator<T> right) implements Iterator<T> {

  public boolean hasNext() { return left.hasNext() || right.hasNext(); }

  public T next() {
    if (!left.hasNext()) { return right.next(); }
    if (!right.hasNext()) { return left.next(); }
    boolean leftWin= left.peek().compareTo(right.peek()) <= 0;
    return (leftWin ? left : right).next();
  }
}
```

Merge has two PeekIterators and implements Iterator

```java
record Merge<T extends Comparable<? super T>>(
    PeekIterator<T> left, PeekIterator<T> right) implements Iterator<T> {

  public boolean hasNext() { return left.hasNext() || right.hasNext(); }

  public T next() {
    if (!left.hasNext()) { return right.next(); }
    if (!right.hasNext()) { return left.next(); }
    boolean leftWin= left.peek().compareTo(right.peek()) <= 0;
    return (leftWin ? left : right).next();
  }
}
```

Merge has two PeekIterators and implements Iterator

hasNext holds if either
left or right have a next

```java
record Merge<T extends Comparable<? super T>>(
    PeekIterator<T> left, PeekIterator<T> right) implements Iterator<T> {

  public boolean hasNext() { return left.hasNext() || right.hasNext(); }

  public T next() {
    if (!left.hasNext()) { return right.next(); }
    if (!right.hasNext()) { return left.next(); }
    boolean leftWin= left.peek().compareTo(right.peek()) <= 0;
    return (leftWin ? left : right).next();
  }
}
```

Merge has two PeekIterators and implements Iterator

hasNext holds if either left or right have a next

To go next we take the smallest between the left and the right next values

```java
record Merge<T extends Comparable<? super T>>(
    PeekIterator<T> left, PeekIterator<T> right) implements Iterator<T> {

  public boolean hasNext() { return left.hasNext() || right.hasNext(); }

  public T next() {
    if (!left.hasNext()) { return right.next(); }
    if (!right.hasNext()) { return left.next(); }
    boolean leftWin= left.peek().compareTo(right.peek()) <= 0;
    return (leftWin ? left : right).next();
  }
}
```

```java
record Merge<T extends Comparable<? super T>>(
    PeekIterator<T> left, PeekIterator<T> right) implements Iterator<T> {

  public boolean hasNext() { return left.hasNext() || right.hasNext(); }

  public T next() {
    if (!left.hasNext()) { return right.next(); }
    if (!right.hasNext()) { return left.next(); }
    boolean leftWin= left.peek().compareTo(right.peek()) <= 0;
    return (leftWin ? left : right).next();
  }
}
```

This is where we peek!

```java
record Merge<T extends Comparable<? super T>>(
    PeekIterator<T> left, PeekIterator<T> right) implements Iterator<T> {

  public boolean hasNext() { return left.hasNext() || right.hasNext(); }

  public T next() {
    if (!left.hasNext()) { return right.next(); }
    if (!right.hasNext()) { return left.next(); }
    boolean leftWin= left.peek().compareTo(right.peek()) <= 0;
    return (leftWin ? left : right).next();
  }
}
```

This is where we peek!

We observe both Iterators, but we move forward only one of the two

```java
final class PeekIterator<T> implements Iterator<T> {
  public boolean hasNext(){ return top != null; }
  public T next() {
    try { return top; }
    finally{ top = (delegate.hasNext() ? delegate.next() : null); }
  }
  private T top;
  final private Iterator<T> delegate;
  public PeekIterator(Iterator<T> delegate) {
    this.delegate = delegate;
    next();
  }
  public T peek(){ return top; }
}
```

```java
final class PeekIterator<T> implements Iterator<T> {
  public boolean hasNext(){ return top != null; }
  public T next() {
    try { return top; }
    finally{ top = (delegate.hasNext() ? delegate.next() : null); }
  }
  private T top;
  final private Iterator<T> delegate;
  public PeekIterator(Iterator<T> delegate) {
    this.delegate = delegate;
    next();
  }
  public T peek(){ return top; }
}
```

And here we find PeekIterator

```java
final class PeekIterator<T> implements Iterator<T> {
  public boolean hasNext(){ return top != null; }
  public T next() {
    try { return top; }
    finally{ top = (delegate.hasNext() ? delegate.next() : null); }
  }
  private T top;
  final private Iterator<T> delegate;
  public PeekIterator(Iterator<T> delegate) {
    this.delegate = delegate;
    next();
  }
  public T peek(){ return top; }
}
```

And here we find PeekIterator

Yes, PeekIterator is user-defined code, too

```java
final class PeekIterator<T> implements Iterator<T> {
  public boolean hasNext(){ return top != null; }
  public T next() {
    try { return top; }
    finally{ top = (delegate.hasNext() ? delegate.next() : null); }
  }
  private T top;
  final private Iterator<T> delegate;
  public PeekIterator(Iterator<T> delegate) {
    this.delegate = delegate;
    next();
  }
  public T peek(){ return top; }
}
```

And here we find PeekIterator

Yes, PeekIterator is user-defined code, too

Significantly, its ‘hasNext’ and ‘next’ methods operate in constant time, regardless of the type of iterator it encapsulates

Riky and Hanton are also discussing the solution

Riky and Hanton are also discussing the solution

Riky and Hanton are also discussing the solution


'sorted' recursively
constructs a binary tree


'sorted' recursively constructs a binary tree
Peak(Merge(_,_))


'sorted' recursively constructs a binary tree
Peak(Merge(_,_))


'sorted' recursively constructs a binary tree
Peak(Merge(_,_))
Peak


'sorted' recursively constructs a binary tree
Peak(Merge(_,_))
Peak
Peak(Merge(_,_))


'sorted' recursively constructs a binary tree
Peak(Merge(_,_))
Peak
Peak(Merge(_,_))

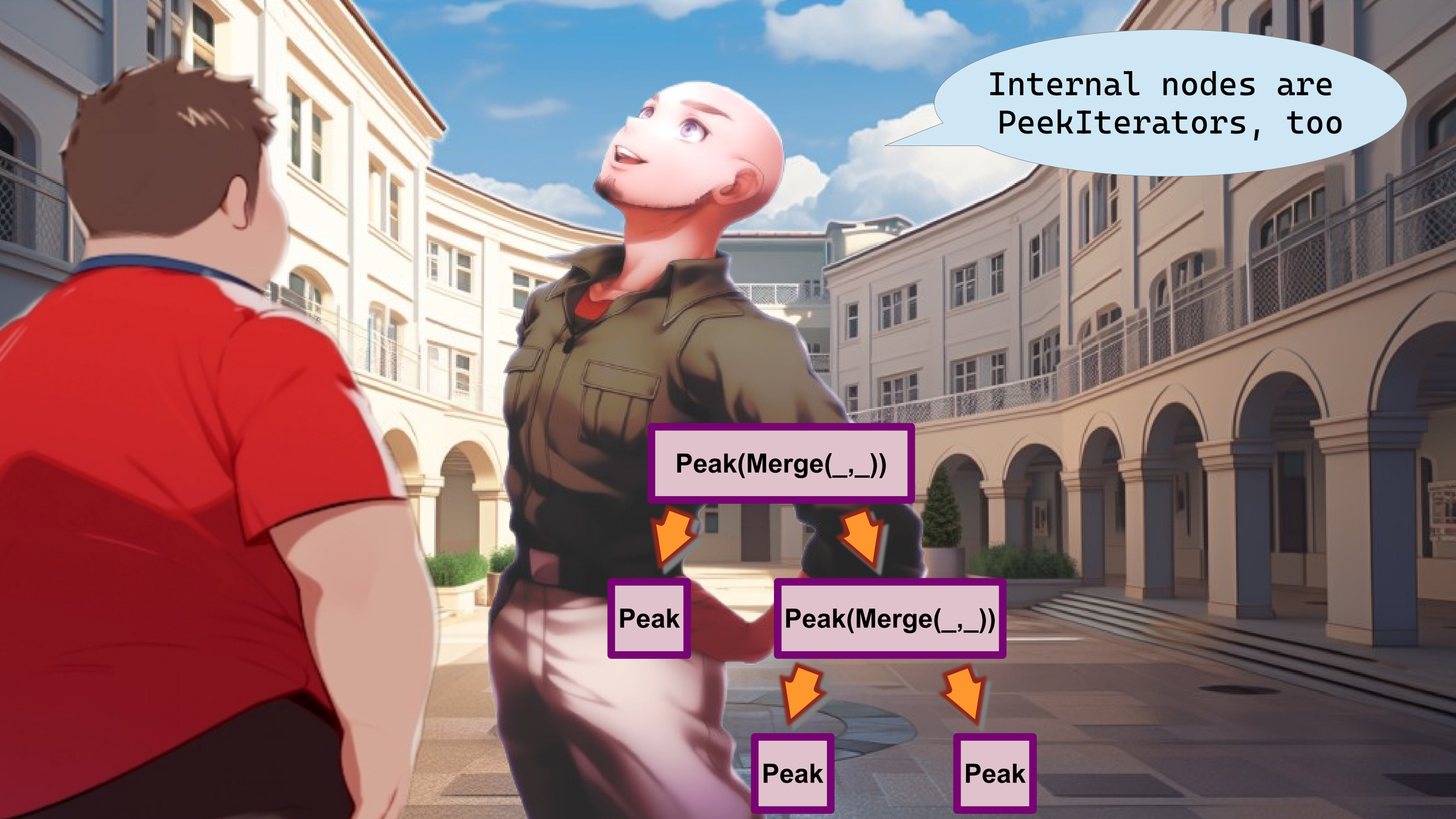

'sorted' recursively constructs a binary tree
Peak(Merge(_,_))
Peak
Peak(Merge(_,_))
Peak


'sorted' recursively constructs a binary tree
Peak(Merge(_,_))
Peak
Peak(Merge(_,_))
Peak
Peak


'sorted' recursively constructs a binary tree
Leaves are PeekIterators
Peak(Merge(_,_))
Peak
Peak(Merge(_,_))
Peak
Peak


'sorted' recursively constructs a binary tree
Leaves are PeekIterators
wrapping single-element iterators
Peak(Merge(_,_))
Peak
Peak(Merge(_,_))
Peak
Peak


Peak(Merge(_,_))
Peak
Peak(Merge(_,_))
Peak
Peak


Internal nodes are PeekIterators, too
Peak(Merge(_,_))
Peak
Peak(Merge(_,_))
Peak
Peak


Internal nodes are PeekIterators, too
Wrapping Merge instances, which in turn wrap two child iterators
Peak(Merge(_,_))
Peak
Peak(Merge(_,_))
Peak
Peak


Yes


Yes
The tree lays out in memory the merge sort divide-and-conquer strategy before running it


Yes
The tree lays out in memory the merge sort divide-and-conquer strategy before running it
The leafs are the base cases and the internal nodes represent merge operations to be performed

```java
class MergeSort {
  public <T extends Comparable<? super T>> List<T> sort(List<T> c) {
    List<T> result = new ArrayList<>(c.size());
    sorted(c).forEachRemaining(e -> result.add(e));
    return Collections.unmodifiableList(result);
  }
  <T extends Comparable<? super T>> PeekIterator<T> sorted(List<T> input) {
    int size = input.size();
    if (size < 2){ return new PeekIterator<>(input.iterator()); }
    int mid = size / 2;
    PeekIterator<T> left=  sorted(input.subList(0, mid));
    PeekIterator<T> right= sorted(input.subList(mid, size));
    return new PeekIterator<>(new Merge<>(left, right));
  }
}
record Merge<T extends Comparable<? super T>>( … )
  implements Iterator<T>{ … }

final class PeekIterator<T> implements Iterator<T> { … }
```

Miss Pumpkin’s full solution 1/3

```java
record Merge<T extends Comparable<? super T>>(
    PeekIterator<T> left, PeekIterator<T> right) implements Iterator<T> {

  public boolean hasNext() { return left.hasNext() || right.hasNext(); }

  public T next() {
    if (!left.hasNext()) { return right.next(); }
    if (!right.hasNext()) { return left.next(); }
    boolean leftWin= left.peek().compareTo(right.peek()) <= 0;
    return (leftWin ? left : right).next();
  }
}
```

Miss Pumpkin’s full solution 2/3

```java
final class PeekIterator<T> implements Iterator<T> {
  public boolean hasNext() { return top != null; }
  public T next() {
    try { return top; }
    finally { top = (delegate.hasNext() ? delegate.next() : null); }
  }
  private T top;
  final private Iterator<T> delegate;
  public PeekIterator(Iterator<T> delegate) {
    this.delegate = delegate;
    next();
  }
  public T peek() { return top; }
}
```

Miss Pumpkin’s full solution 3/3


I'm looking at Pupon's solution of the Optionals question
No surprise Pumpkin made this question
She loves functional programming
I wonder how Pupon handled it since he clearly favors the imperative style

```java
record Some<T>(T get) implements Optional<T>{
  public T orElseGet(Supplier<T> unused){ return get; }
  public Optional<T> or(Supplier<Optional<T>> s){ return this; }
  public <U> Optional<U> map(Function<T, U> m){
    return Optional.ofNullable(m.apply(get));
  }
  public <U> Optional<U> flatMap(Function<T, Optional<U>> m){
    return Objects.requireNonNull(m.apply(get));
  }
}

enum Empty implements Optional<Object>{ Instance; }
```

Pupon’s full solution 1/2

```java
public sealed interface Optional<T>
    extends Serializable permits Empty, Some<T>{
  @SuppressWarnings("unchecked")
  static <E> Optional<E> empty(){ return (Optional<E>)Empty.Instance; }
  static <T> Optional<T> of(T value){
    return new Some<T>(Objects.requireNonNull(value));
  }
  static <T> Optional<T> ofNullable(T value){
    return value == null ? empty() : new Some<T>(value);
  }
  default T orElseGet(Supplier<T> s){ return s.get(); }
  default Optional<T> or(Supplier<Optional<T>> s){
    return Objects.requireNonNull(s.get());
  }
  default <U> Optional<U> map(Function<T, U> m){ return Optional.empty(); }
  default Optional<T> filter(Predicate<T> p){
    return map(e->p.test(e) ? e : null);
  }
  default <U> Optional<U> flatMap(Function<T, Optional<U>> m){
    return Optional.empty();
  }
}
```

Pupon’s full solution 2/2