

Assignment	<ol style="list-style-type: none"> 1. Write a class named <code>ArrayIterator</code> that implements the <code>Iterator</code> interface and iterates over elements in an array, but skips <code>null</code> elements in the array. 2. For type safety, include a type parameter <code><T></code> in the class. 3. Use the package <code>ku.util</code> for your code.
What to Submit	Commit your project to Github as project name <code>arrayiterator</code> . Write good Javadoc for class and methods.
Due	Monday, 6 February 2017.

Iterators

Many collections and data structures provide an *Iterator* so we can iterate over all the elements in the collection *without knowing the structure* of the collection.

In Java, an *Iterator* is any object that implements the `java.util.Iterator` interface. This interface has a type parameter that describes the type of element the *Iterator* returns.

Iterator Interface in Java

The `java.util.Iterator` interface has 3 methods. The interface has a type parameter (usually shown as "T" or "E"). If you omit the type parameter, the default value is `Object`. Here are the 3 methods (shown with and without type param):

Type parameter T	No type parameter	Meaning
T <code>next()</code>	Object <code>next()</code>	Return the <i>next non-null</i> element in the array. If there are no more elements, it throws <code>NoSuchElementException</code> .
boolean <code>hasNext()</code>	boolean <code>hasNext()</code>	Returns <code>true</code> if <code>next()</code> can return another non-null array element, <code>false</code> if no more elements.
void <code>remove()</code>	void <code>remove()</code>	(Optional) Remove most recent element returned by <code>next()</code> from the array by setting it to <code>null</code> . This method may only be called <u>once</u> after a call to <code>next()</code> . If this method is called without calling <code>next()</code> , or called more than once after calling <code>next()</code> , it throws <code>IllegalStateException</code> .

Example:

`Scanner` is a `String` *Iterator*. In the Java API documents for `Scanner`, it shows that `Scanner` implements `Iterator<String>`. This just means that `next()` will return a `String`.

Every time you call `next()` it returns another `String` from the source. The "source" can be an `InputStream` like `System.in`, another `String`, or a `File`. In this example, we create a `Scanner` to split the `String` *"Iterating is so easy."* into words. Each time you call `next()` it returns the next word.

```
Scanner input = new Scanner( "Iterating is so easy!" );
while( input.hasNext() ) {
    String s = input.next();
    System.out.println( s );
}
```

```
Iterating
is
```

so
easy!

Assignment

Arrays don't have an `Iterator`, but it would be really useful to have one. Your assignment is to write an `ArrayIterator` class that provides an *Iterator* for any array.

For *convenience*, we will design the `ArrayIterator` so it will skip null elements in the array.

1. Write a class named `ArrayIterator` that implements `java.util.Iterator`.
2. Use a *type parameter* in the class declaration and methods. Declare the class like this:

```
public class ArrayIterator<T> implements Iterator<T>
```

`T` is a *type parameter*, which is a placeholder for the name of a class or Interface. We will study type parameters later, but you can use it by just following the sample code below.

3. The *type parameter* should match the type of elements in the array. If we have an array of `String`, we want `ArrayIterator` to return `Strings` so we would write "`new ArrayIterator<String>(array)`". If we have an array of `Student`, we would write "`new ArrayIterator<Student>(students)`" and "`T`" would become `Student`.

Define `ArrayIterator` like this. The "`T`" means the type of thing in the `ArrayIterator`. At run-time it will be replaced by the name of an actual class (like `Student` if we ask for `ArrayIterator<Student>`).

```
public class ArrayIterator<T> implements Iterator<T> {
    /** attribute for the array we want to iterate over */
    private T[ ] array;

    /**
     * Initialize a new array iterator with the array to process.
     * @param array is the array to iterate over
     */
    public ArrayIterator(T[] array) {
        this.array = array;
        //TODO: initialize any other variables you need
    }

    /**
     * Return the next non-null element from array, if any.
     * @return the next non-null element in the array.
     * @throws NoSuchElementException if there are no more elements
     *         to return.
     */
    public T next( ) {

    }
    //TODO the other methods don't use the type parameter,
    // so you should have no problem writing them.
}
```

4. The *constructor* has a parameter that is an array of type `T`. In Java, you can use a type parameter just like a class name (except that you can't create "new" objects using a type parameter).
5. `ArrayIterator` may *not* use any Java collections (like `ArrayList`). `ArrayIterator` needs only a reference to the **array** and a variable (or two) to keep track of the next element to return..

6. The `next()` and `hasNext()` methods should *skip null values* (see example below).
7. If the user calls `next()` when there are no more elements, `next` throws a `NoSuchElementException`. Your `ArrayIterator` should do this, too. Here is how to throw an exception:

```
if ( foo ) throw new NoSuchElementException( );
```

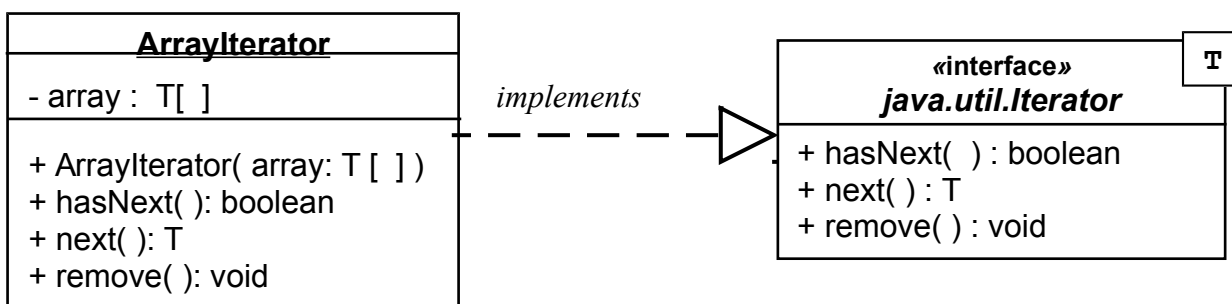
See the Programming Notes below for more on throwing an exception.

remove() method

This method is optional. You can leave the `remove()` method empty.

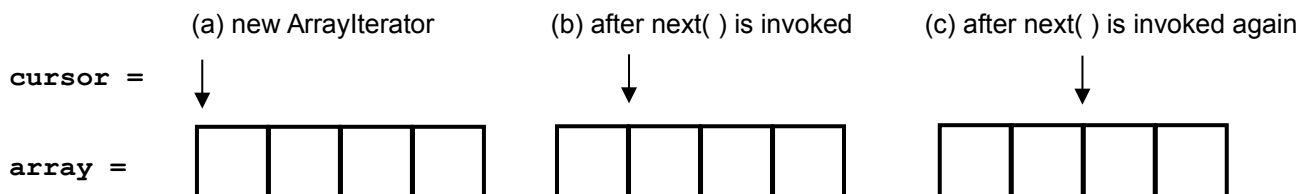
Extra credit will be given if you write a 100% correct `remove` method (see specification above).

Class Diagram for ArrayIterator



Programming Notes

1. An Iterator needs a variable (often called the **cursor**) to remember its position in the collection. Initially the cursor points to the first element. Each time **next** is called, the Iterator returns the current element and increments the cursor. `hasNext()` may also advance the cursor to skip null elements.



2. The **hasNext** method *does most of the work!* It is the job of `hasNext` to decide if there is another element available and *move the cursor* to the start of the next (non-null) element.
3. **Don't duplicate** code or logic! The `next` method should ask `hasNext` if there is another element, and let `hasNext` do the work of skipping nulls. Don't copy the `hasNext` logic into the `next` method.

4. It is legal for the user to call `hasNext()` *many times* consecutively without calling `next`. The iterator must not skip any elements if the user does this!

```
iterator.hasNext( );
iterator.hasNext( ); // no change. Duplicate calls to hasNext don't change the iterator.
iterator.hasNext( ); // no change, again.
```

5. It is also legal for the user to call `next` *without* calling `hasNext`. Therefore, you must not *assume* the user will always call `hasNext` before `next`.

```
String [] array = { "apple", "banana", null, "carrot" };
ArrayIterator<String> iter = new ArrayIterator( array );
iter.next( );      // returns "apple"  User is not required to call hasNext.
iter.hasNext( );   // true
```

```
iter.hasNext( ); // true again
iter.hasNext( ); // true again      User can call hasNext many times
iter.next( );      // returns "banana"
iter.next( ):      // returns "carrot" (skip over null element)
iter.hasNext();    // false
iter.next( );      // throws NoSuchElementException
```

6. To throw an Exception, simply write `throw new NoSuchElementException()`. Throwing an exception causes an immediate return from the method. Don't write `return` after `throw`. For example:

```
/** get the n-th element from double array[ ] */
public double get(int n) {
    if (n >= 0 && n < array.length) return array[n];
    else throw new NoSuchElementException( );
    // "throws" exits from the method so don't write "return" here.
}
```

Example using BlueJ Interactive Mode

```
> String [] fruit = { "apple", null, null, "banana"};
> ArrayIterator<String> it = new ArrayIterator<String>(fruit);
> it.hasNext()
true
> it.next()
"apple"
> it.next()
"banana"
> it.hasNext()
false          // no more elements: hasNext() is false forever
> it.next( )
java.util.NoSuchElementException at ArrayIterator:xx
```

Example using an empty array:

```
> Object [ ] array = new Object[1]; // array containing null
> ArrayIterator it = new ArrayIterator( array );
> it.hasNext( )
false
> it.next()
java.util.NoSuchElementException at ArrayIterator:xx
```