

Data Structures Part 1

Java Accelerator 7

Lesson 1.4



Learning Objectives

01

Compare the advantages and disadvantages of common collection structures such as Lists, Arrays, Stacks, and Queues.

02

Use testing libraries to utilize TDD.

Topics

In this lesson we will cover:



Unit testing with JUnit



TDD



Java Collections



Java Maps

Unit Testing and TDD



Instructor Demonstration

Calculator Test



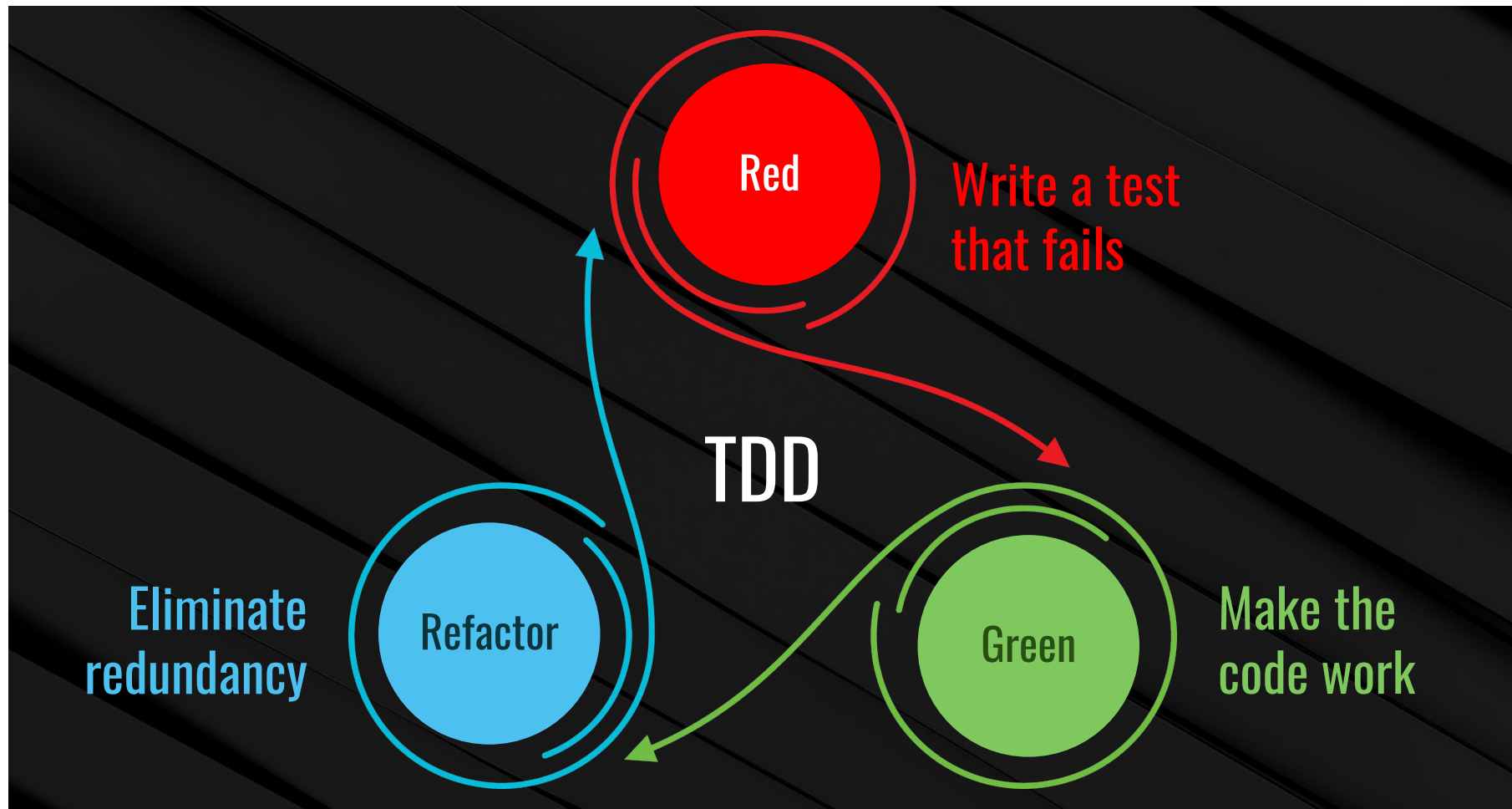
Activity: Calculator Test

Suggested Time:

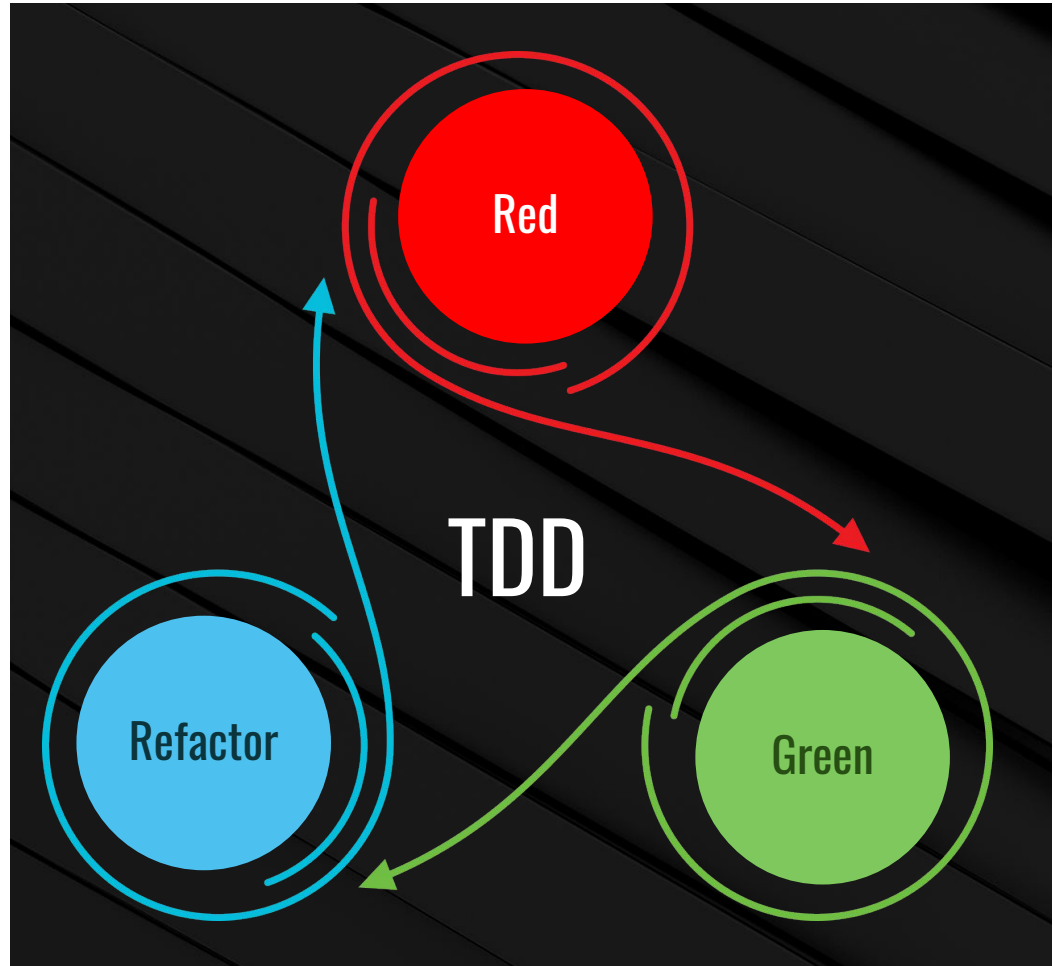
TDD and Unit/Integration Tests



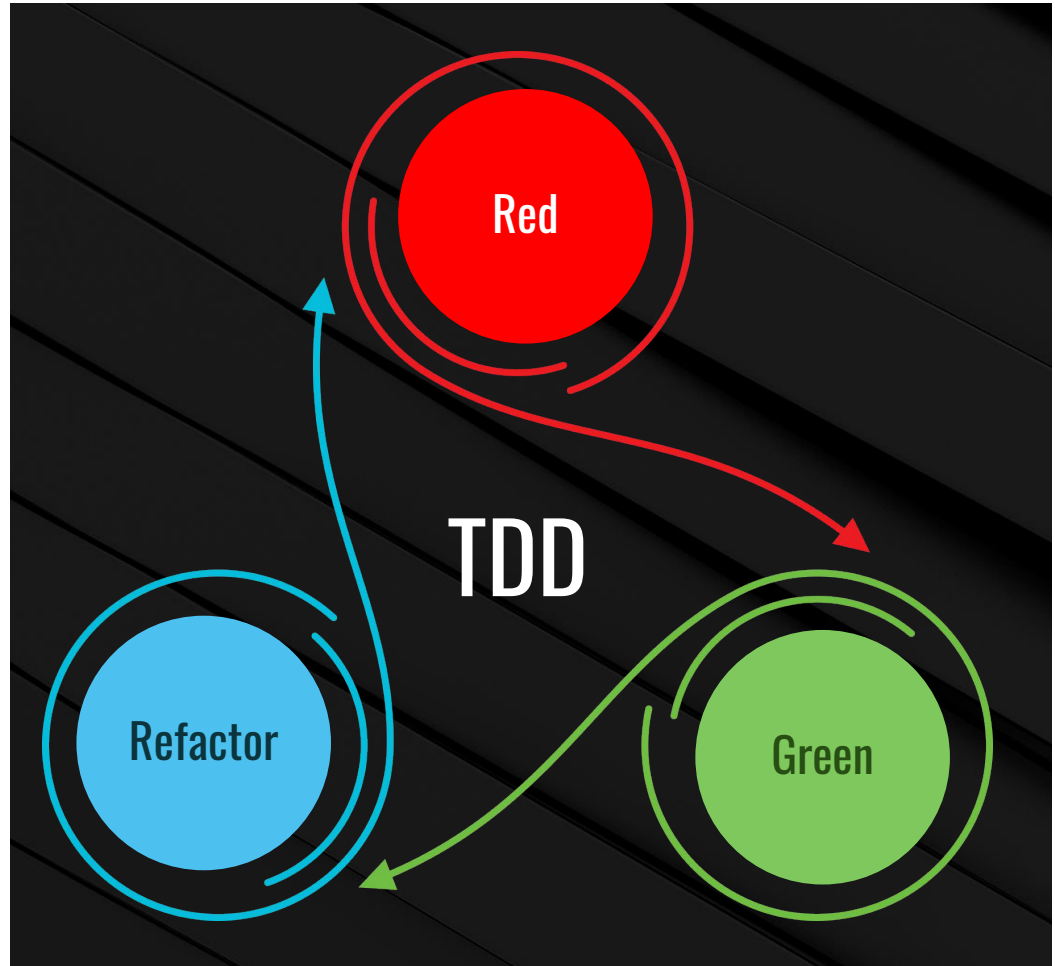
**TDD uses a technique called
Red, Green, Refactor.**



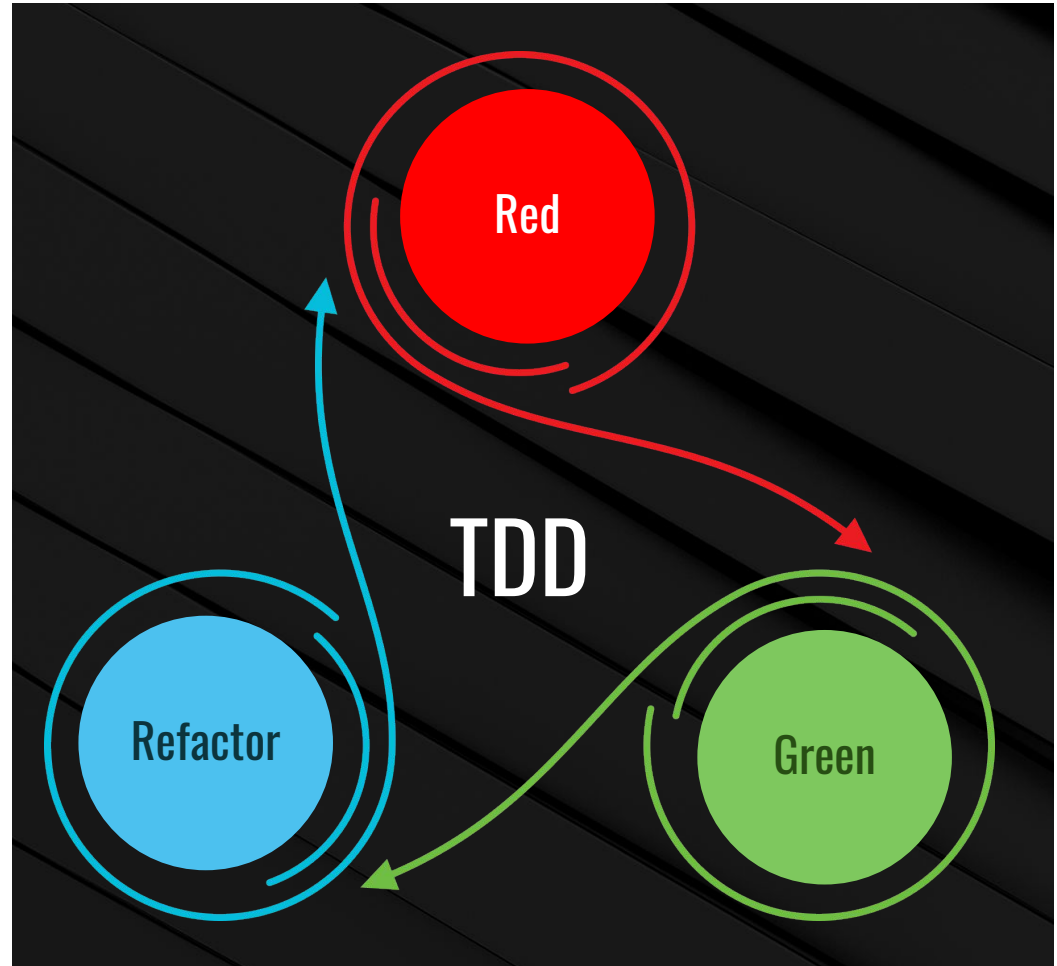
The tests will fail initially because we haven't implemented any of the logic. This is the **"Red"** part of Red, Green, Refactor.



Next, you implement code until all of the tests pass. This is the "Green" part.



Finally the team can feel confident in any refactoring they do moving forward because they have a robust set of unit tests that will tell them if they have broken anything during the **refactoring** process.



Types of Tests

Unit Tests	Testing individual units of code (usually a method).
Integration Tests	Testing the interaction between two or more units of code.
System Tests	Testing that the entire system meets requirements.
Acceptance Tests	Testing that the software meets business and user needs.

TDD Steps

01

Begin by writing failing tests.

02

Then write the minimum code necessary to pass the tests.

03

Then refactor the code and tests as needed, but not simultaneously.



Time to Code



TV TDD

Suggested Time:



Activity: SumArrays TDD

Suggested Time:

JUnit Concepts

Intermediate JUnit Concepts

<code>@Before</code>	Code to run before each test. Generally set up, variable creation, constructor calls, etc.
<code>@BeforeClass</code>	Code to run once before all tests.
<code>@AfterClass</code>	Code to run after all tests. Generally used to close DB connections, kill stubs, delete files created in testing, etc.
<code>assertArrayEquals</code>	Checks the equivalence of values and order in an array.
<code>assertEquals(double expected, double actual, double delta)</code>	Checks the equality of 2 doubles to within a defined margin of error.
<code>assertThat</code>	Extends the functionality of JUnit considerably.



**What does it mean for two
objects to be equal?**

Object Equality

There are two definitions of object equality.

01

Shallow Equality: The two variables are actually two references to the same object.

```
Dog fido = new Dog();  
Dog rex = fido;
```

02

Deep Equality: Two objects share the same value for all of their properties.

```
Dog fido = new Dog("brown", 2, "frisbee");  
Dog rex = new Dog("brown", 2, "frisbee");
```

Object Equality Questions



What's a scenario in which you would want to test for shallow equality?



What's a scenario in which you would want to test for deep equality?



When might we not want to check every property for equality?



Instructor Demonstration

Java Object Equality



Time to Code

Research the Java Collections Framework

Suggested Time:

Iterators



**What are three ways to visit all
of the elements in an array?**

Elements in an Array

Three ways to visit all of the elements in an array:

01

Sometimes we need to visit all of the elements in a collection as well.

02

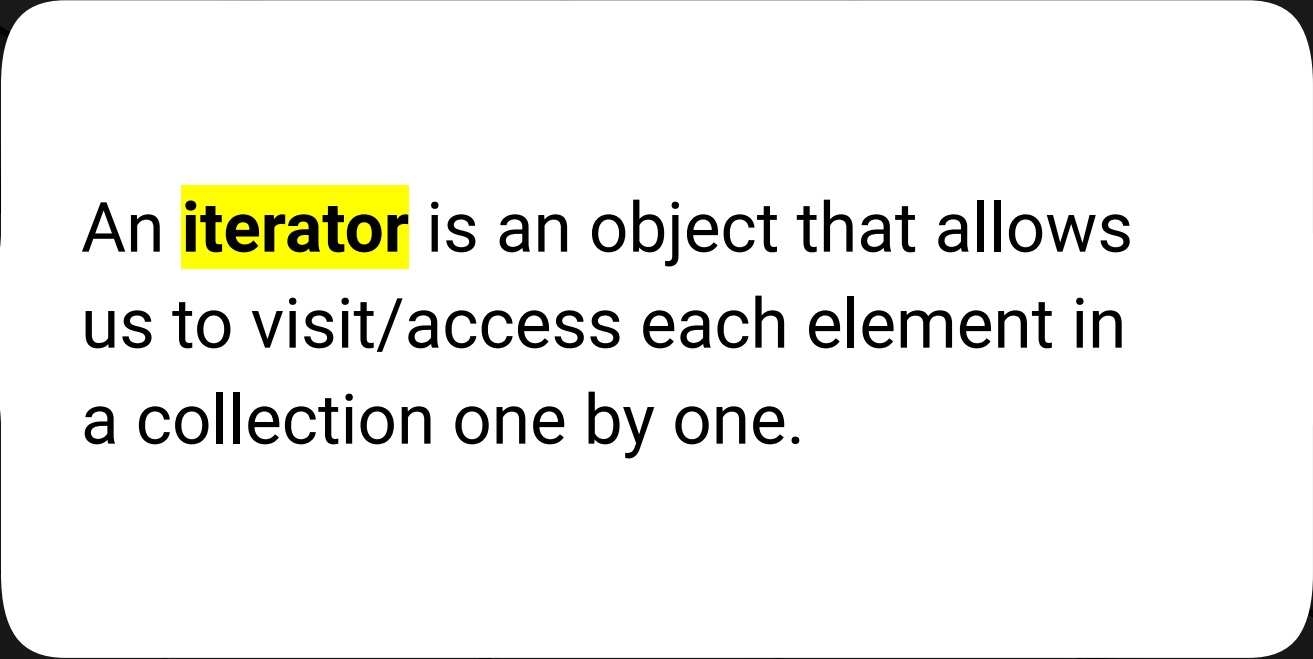
Not all collections are ordered like an array (for example, Sets).
This makes taking the array approach problematic.

03

We still need a way to visit/access each element in a collection.



This is where iterators come in.



An **iterator** is an object that allows us to visit/access each element in a collection one by one.

Iterator Methods

Iterators have two methods:

`hasNext()`

Returns true if there are one or more elements left to access

`next()`

Grabs the next element and returns it to us



Some iterators also have a `remove()` method, but this is not required.

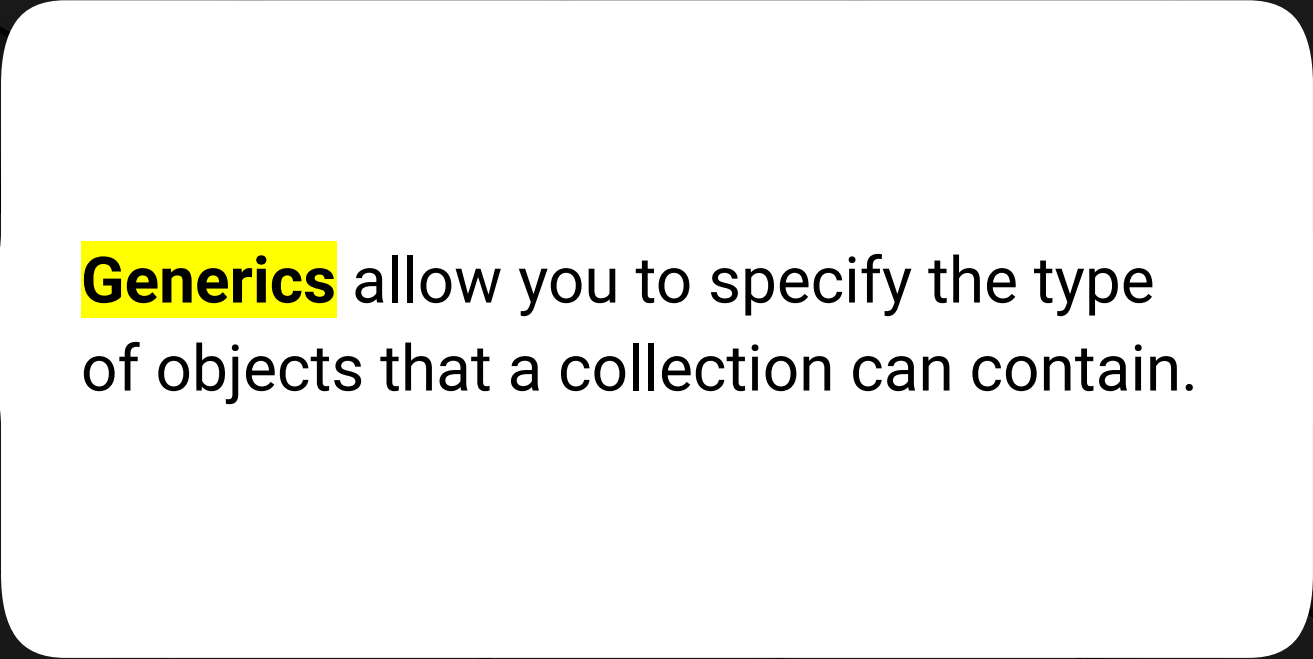


The pattern is that we keep calling `next()` as long as there is another element to visit (i.e., `hasNext()` returns true).



We have no control over the order in which the elements are returned, but we really don't care. We just care that we are guaranteed to visit/access each element exactly once.

Generics



Generics allow you to specify the type of objects that a collection can contain.

Generics

All of the data structures in the collections framework and in the maps family hold references to objects. For example, you could have a Set of Student objects or a List of Address objects.

```
List<Address> addressList;  
List<City> cityList;
```


Generics

Generics have a slightly different declaration syntax: not only do we have to specify the type of the collection (i.e., List, Set, etc.) but we also have to specify the type of object that the collection can hold.

```
List<Address> addressList;  
List<City> cityList;
```

Generics

Note that the declaration pattern for generics is basically the same as for other types: `data-type variable-name`

We are simply adding the angle brackets after the variable name to specify the type of object that can be held in the collection (List, in this case).

```
List<Address> addressList;  
List<City> cityList;
```



Instructor Demonstration

Explore List and ArrayList Javadoc



Instructor Demonstration

List Syntax and Operations



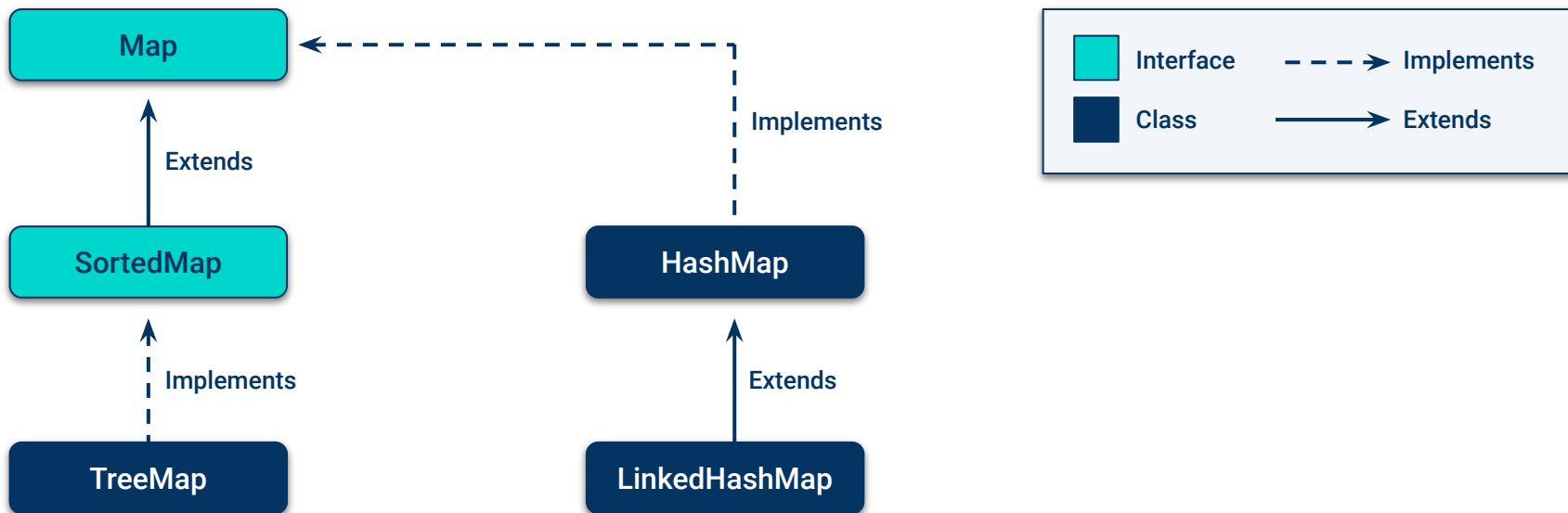
Activity: Using Lists

Suggested Time:

Java Map Concepts

Java Map Concepts

Maps are similar in many ways to collections and lists: the Map interface defines an API that allows us to manipulate groups of related objects.



Java Map Concepts

Maps are a key/value data structure—we store values in the map and associate those values with keys.

This is similar to a coat check:

they store your coat in the closet and give you a token with a number that maps to the location of your coat in the closet. When you want your coat, you present your token and they use that to find your coat.





Since the values are accessed by key, the Map interface makes no claim about the order of the values—order doesn't matter in a map. If you need a guaranteed order to your values, you should use a list.

Java Map Concepts

Each key can map to one and only one value and duplicate keys are not allowed.

This makes sense if we go back to our coat check analogy:

if the coat check token mapped to more than one coat, it would be impossible to keep track of who owned what coat. Similarly, if there are duplicate coat tokens that point to one coat, how do you know who to give the coat to?



Java Map Concepts

The values have fewer restrictions; for example, we can have duplicate values. Going back to the coat check, it is possible that two people have the exact same type and size of coat. We can still check both and we will be able to tell them apart because we have unique coat check tokens (keys).



323

Claim Check

In case of any loss
claim before leaving

Not responsible for contents of
garments or anything left overnight.



376

Claim Check

In case of any loss
claim before leaving

Not responsible for contents of
garments or anything left overnight.



Instructor Demonstration

Using Maps



Activity: Using Maps

Suggested Time:



Activity: Largest Cities

Suggested Time:



Recap: Testing

You should now be able to do the following:



Explain the purpose of testing.



Explain the different types of tests on which we will focus.



Explain the role of equivalence classes in testing.



Use JUnit to write tests for existing methods.



Implement Test Driven Development to inform coding decisions.



Override the equals and hashCode methods to aid in testing objects.

Recap: Collections

You should now be able to do the following:



Instantiate an ArrayList that holds any type and refer to it using a list reference.



Add elements to a list.



Determine the size of a list.



Remove elements from a list.



Visit all the elements in a list using: a regular for loop; an enhanced for loop; and an iterator.



Test collections.

Recap: Maps

You should now be able to do the following:



Compare and contrast the Map and List interfaces.



Explain the purpose of the Java map.



Declare and initialize a map (using the HashMap implementation) for various types of key and values.



Add entries to a map.



Determine the size of a map.



Retrieve an entry from a map.

Recap: Maps

You should now be able to do the following:



Replace an entry in a map.



Remove an entry from a map.



Get all the keys from a map.



Visit all the keys in a map.



Get all the values from a map.



Visit all the values in a map.

Recap: Maps

You should now be able to do the following:



Describe the `Map.Entry` data structure.



Get all the `Map.Entry` objects from a map.



Print out the values in all the `Map.Entry` object in a map.



Explain why the keys in a map are returned in a set.



Explain why the values in a map are returned in a collection.