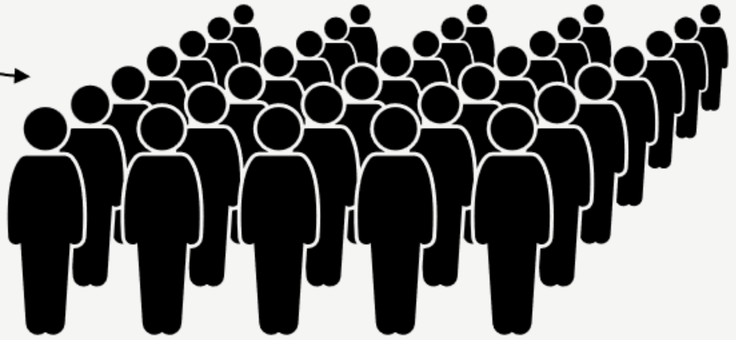
Arrays have limitations

You started an app and since it is new, you created an array to hold 50 customers. Because you don't want to waste memory by definig the size of the array with a bigger number



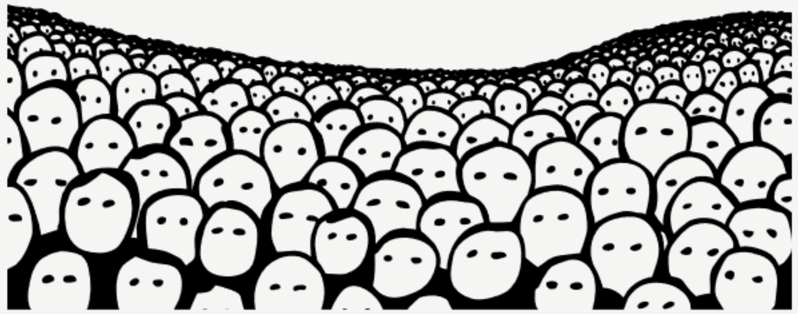
Things were running smoothly until the unexpected arrival of customer number 51. As you input data for this new customer, you find yourself wishing that the array, which was initially designed to hold only 50 components, could magically expand to accommodate your growing list.

However, your wish is not granted as **arrays do not have the ability to expand dynamically**. Consequently, the program crashes with an ArrayIndexOutOfBoundsException.

To avoid future occurrences of this issue, you make the necessary modifications to your program by increasing the array size to 100, in hopes that this will be sufficient to handle the maximum number of customers you expect to encounter.

In addition to the restriction of having a fixed size, manipulating the position of elements in an array can be a cumbersome and resource-intensive task. This is due to the fact that arrays are stored in contiguous memory locations, which can make it challenging to insert new elements or rearrange their positions.

Another limitation of Java arrays is that they can only store elements of a single data type. This means that it is not possible to store different data types within the same array.



The same story repeats as you grow with the number of customers. As a frustrated developer you are desperately looking for a approach that dynamically expands & shrinks

Arrays have limitations

eazy bytes

Intially, the Array is intialized with three people which will fix the size of the array as 3

```
String[] people = new String[] {"Madan", "Henry", "John"};
```

0	1	2
Madan	Henry	John

To add one more person to the array, we need to create a new array with the size 4

```
String[] newPeople = new String[4];
```

0	1	2	3
null	null	null	null

```
String[] newPeople = Arrays.copyOf(people,people.length);
```

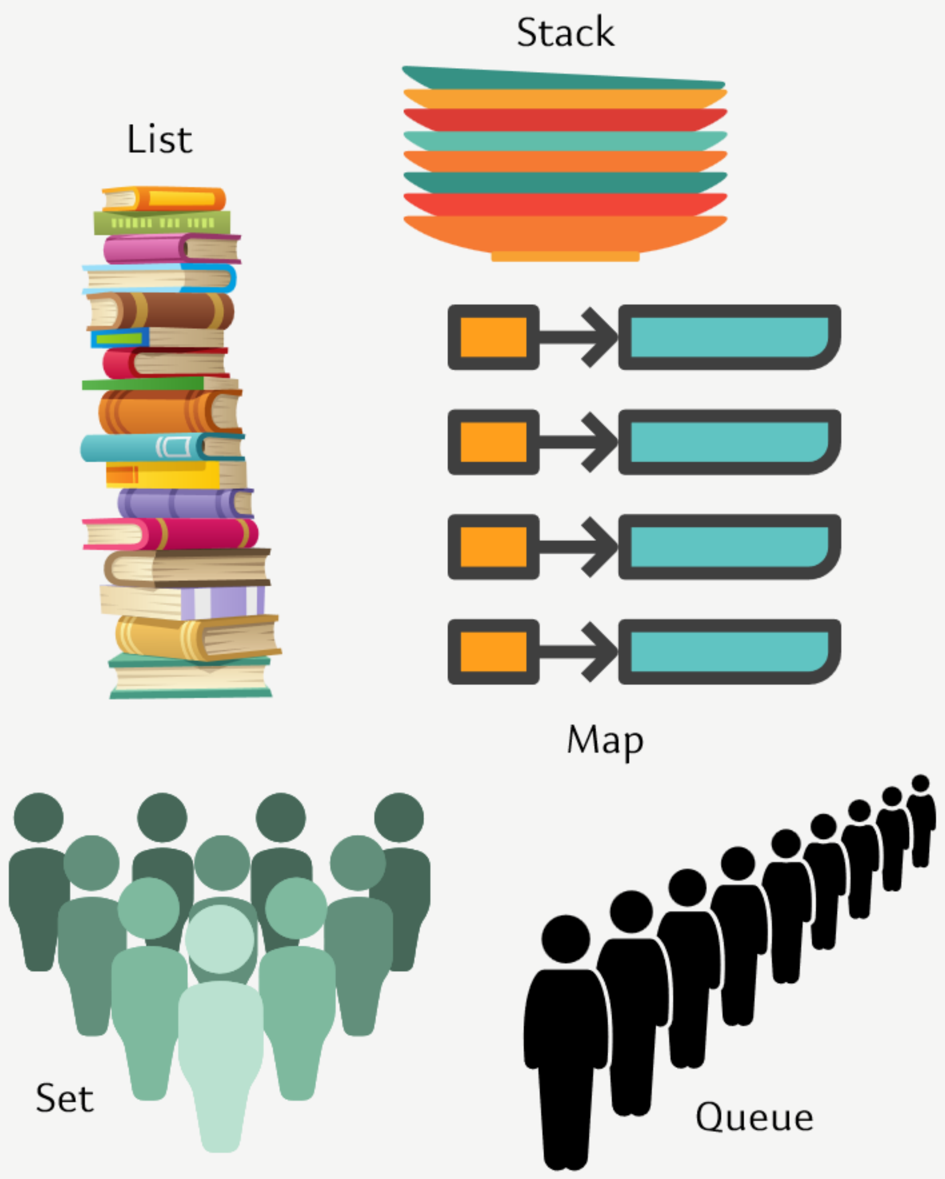
0	1	2	3
Madan	Henry	John	null

```
newPeople[3] = "Sita";
```

0	1	2	3
Madan	Henry	John	Sita

It is a very cumbersome process that needs to be followed when ever we want to resize the array as it is not automcally done

Collections



For quite some time, computer scientists have been addressing the drawbacks of arrays. While no universal solution has been found, several clever techniques have been identified and the same are incorporated into the Java collections.

The Java collections framework consists of classes and interfaces that implement a collection data structure. Collections, which are capable of holding object references and can be managed as a group, are similar to arrays. **The key distinction between the two is that arrays necessitate the specification of their capacity prior to usage, whereas collections can dynamically increase or decrease their size as required. By simply adding or removing object references to a collection, its size can be adjusted accordingly.**

Java collections support various data structures for storing and accessing elements of a collection: **an ordered list, a unique set, a dictionary (called a map in Java), a stack, a queue, and some others.** All classes and interfaces of the Java collections framework belong to the java.util package

A general rule to follow with collections is that storing elements as primitive types like short, int, or double is not possible. Instead, only reference types or object types can be used for storage.

Why Collections accept only Objects

The reason why Collections works with Objects only is to make collections more flexible and generic, allowing them to store a wide variety of types of objects. By accepting only Objects, collections can store any type of object, including user-defined classes, which is a fundamental feature of object-oriented programming.

Furthermore, using Objects allows collections to support polymorphism, a powerful programming concept that allows you to write generic code that can operate on different types of objects.

Overall, the decision to accept only Objects in Java collections was a design choice made to increase their flexibility and genericity, enabling them to store and manipulate a wide variety of objects.

You cannot use primitive data types in collections. If you want to store primitive types in a collection, you must wrap the primitive value before storing it and unwrap it after retrieving it.

Wrapper Classes

eazy bytes

The wrapper classes in Java are used to convert or represent primitive types (int, char, float, etc) into corresponding objects. These classes are called wrapper classes, as they wrap a primitive value in an object. There exist wrapper classes for each of the 8 primitive data types.

Primitive Type	Wrapper Class
byte	Byte
short	Short
int	Integer
long	Long
float	Float
double	Double
boolean	Boolean
char	Character

Wrapper classes are necessary when we want to use primitive data types in contexts that require objects. For instance, if we want to use an int value as an element in a collection, we must wrap it in an Integer object. Similarly, if we want to pass a primitive data type as an argument to a method that expects an object, we must wrap it in the corresponding wrapper class.

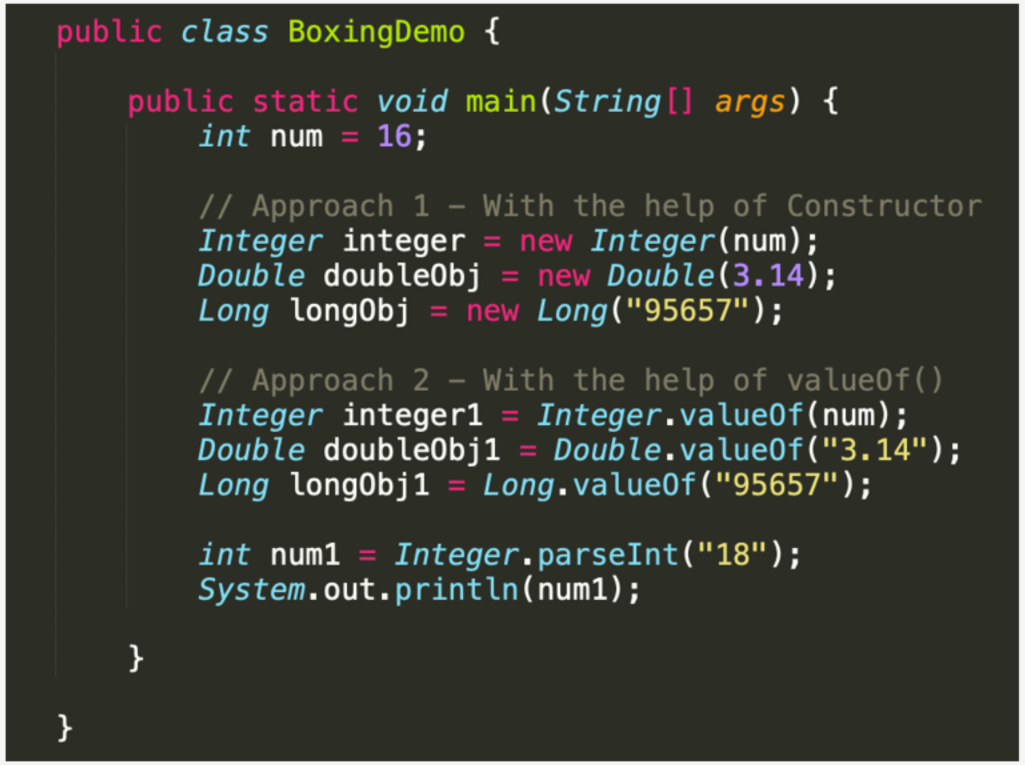
Wrapper classes also provide useful methods for performing operations on the primitive data types. For example, the Integer class provides methods for converting an int to a String, parsing a String to an int, and comparing two int values.

Convert Primitive Type to Wrapper Objects

eazy bytes

To convert a primitive type to a wrapper object in Java, we can use the appropriate wrapper class constructor or static factory method. For example, to convert an int to an Integer, we can use the Integer constructor or the valueOf method. In addition to constructors, static factory methods, wrapper classes also provides methods for converting string representations to the corresponding primitive types.

```
int num = 16;
Integer obj = new Integer(num);  // constructor approach is deprecated
Integer obj2 = Integer.valueOf(num);  // static factory method
int num1 = Integer.parseInt("18");  // String to int
```



```
public class BoxingDemo {

    public static void main(String[] args) {
        int num = 16;

        // Approach 1 - With the help of Constructor
        Integer integer = new Integer(num);
        Double doubleObj = new Double(3.14);
        Long longObj = new Long("95657");

        // Approach 2 - With the help of valueOf()
        Integer integer1 = Integer.valueOf(num);
        Double doubleObj1 = Double.valueOf("3.14");
        Long longObj1 = Long.valueOf("95657");

        int num1 = Integer.parseInt("18");
        System.out.println(num1);

    }

}
```

Similarly, every wrapper class has utility methods to create a wrapper object from primitive data type or to convert from string representations to the corresponding primitive types.

The process of converting a primitive data type to wrapper object is called as Boxing

Convert Wrapper Objects into Primitive Types

To convert objects into the primitive types, we can use the corresponding xxxValue() methods like intValue(), doubleValue(), longValue() etc. present in each wrapper class with the help of objects like shown below,

```java
public class UnboxingDemo {

    public static void main(String[] args) {

        int num = 16;

        // Boxing
        Integer integer1 = Integer.valueOf(num);
        Double doubleObj1 = Double.valueOf("3.14");
        Long longObj1 = Long.valueOf("95657");

        // Unboxing
        int num1 =  integer1.intValue();
        double num2 = doubleObj1.doubleValue();
        long num3 = longObj1.longValue();

        System.out.println(num1);
        System.out.println(num2);
        System.out.println(num3);
    }

}
```

Similarly, every wrapper class has utility methods to convert a wrapper object to primitive data type.

The process of converting a wrapper object to primitive data type is called as Unboxing

Autoboxing and unboxing

In Java, it is a common practice to convert a primitive data type to a wrapped object and vice versa. To simplify this process, the compiler performs automatic boxing and unboxing, which are commonly known as autoboxing and unboxing.

Java Autoboxing – Converting Primitive Type to Wrapper Object

Autoboxing is a feature in Java where the compiler automatically converts primitive data types to their corresponding wrapper class objects. This feature is particularly useful when working with Java collections. Below is an example of Autoboxing,

```
int num = 16;
// autoboxing
Integer numObj = num;
```

Java Unboxing – Converting Wrapper Objects to Primitive Types

Unboxing is a feature in Java where the compiler automatically converts wrapper class objects to their corresponding primitive data types. Below is an example of Auto unboxing,

```
Integer numObj = 16;
// unboxing
int num = numObj;
```

Autoboxing & unboixing are introduced in Java 5. So from Java 5, there is no longer a requirement to use the valueOf() method of wrapper classes to convert primitives to objects, nor is there a need to use the intValue() method of wrapper classes to convert wrapper types to primitives.

Autoboxing and unboxing

Autoboxing/unboxing is performed when you compile the code. The JVM is completely unaware of the boxing and unboxing performed by the compiler.

Beware of null Values

Autoboxing and unboxing offer the convenience of reducing code verbosity and enhancing code readability. However, they introduce certain unexpected scenarios, such as the potential for encountering a NullPointerException where it might not be anticipated. Unlike primitive types, reference types can hold a null value. The process of boxing and unboxing occurs between primitive types and reference types.

Consider the following code snippet:

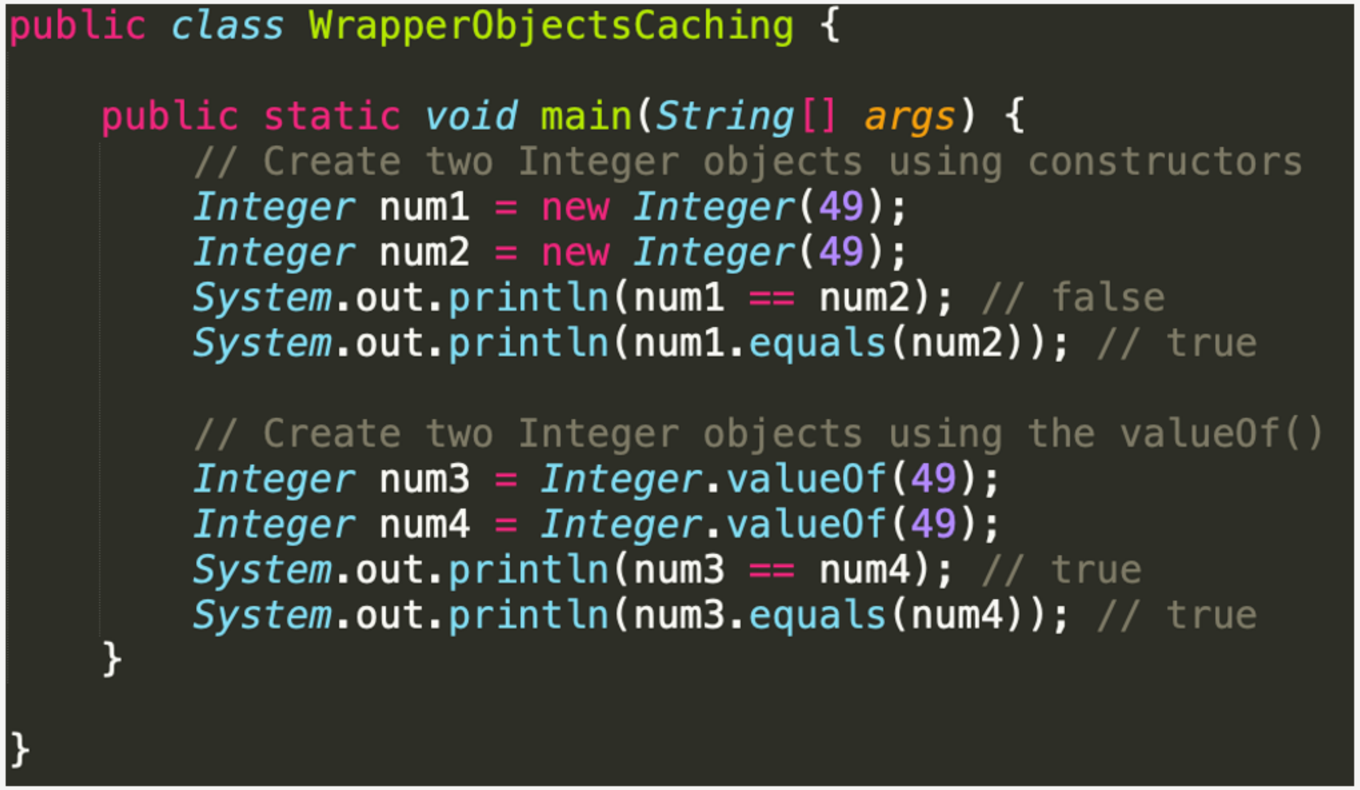
```
Integer n = null;
int a = n;
```

In this example, if you don't have control over the assignment of null to n, and it results from a method call like int a = getSomeValue(), where getSomeValue() returns an Integer object, you may be surprised by a NullPointerException. This is because int a = n is essentially converted to int a = n.intValue(), and in this case, n is null. Being aware of such surprises is integral to understanding the advantages and considerations associated with autoboxing and unboxing.

Caching with valueOf() methods

The utilization of the valueOf() method for creating objects with integer numeric values (byte, short, int, and long) offers improved memory efficiency due to its caching mechanism. The wrapper classes corresponding to these primitive types cache wrapper objects for values falling within the range of −128 to 127.

To illustrate, invoking Integer.valueOf(49) multiple times will yield the reference to the same Integer object from the cache. Conversely, using new Integer(49) for multiple calls results in the creation of a new Integer object each time. The distinction between constructors and valueOf() methods for the Integer wrapper class is demonstrated with the below example,



```java
public class WrapperObjectsCaching {

    public static void main(String[] args) {
        // Create two Integer objects using constructors
        Integer num1 = new Integer(49);
        Integer num2 = new Integer(49);
        System.out.println(num1 == num2); // false
        System.out.println(num1.equals(num2)); // true

        // Create two Integer objects using the valueOf()
        Integer num3 = Integer.valueOf(49);
        Integer num4 = Integer.valueOf(49);
        System.out.println(num3 == num4); // true
        System.out.println(num3.equals(num4)); // true
    }

}
```

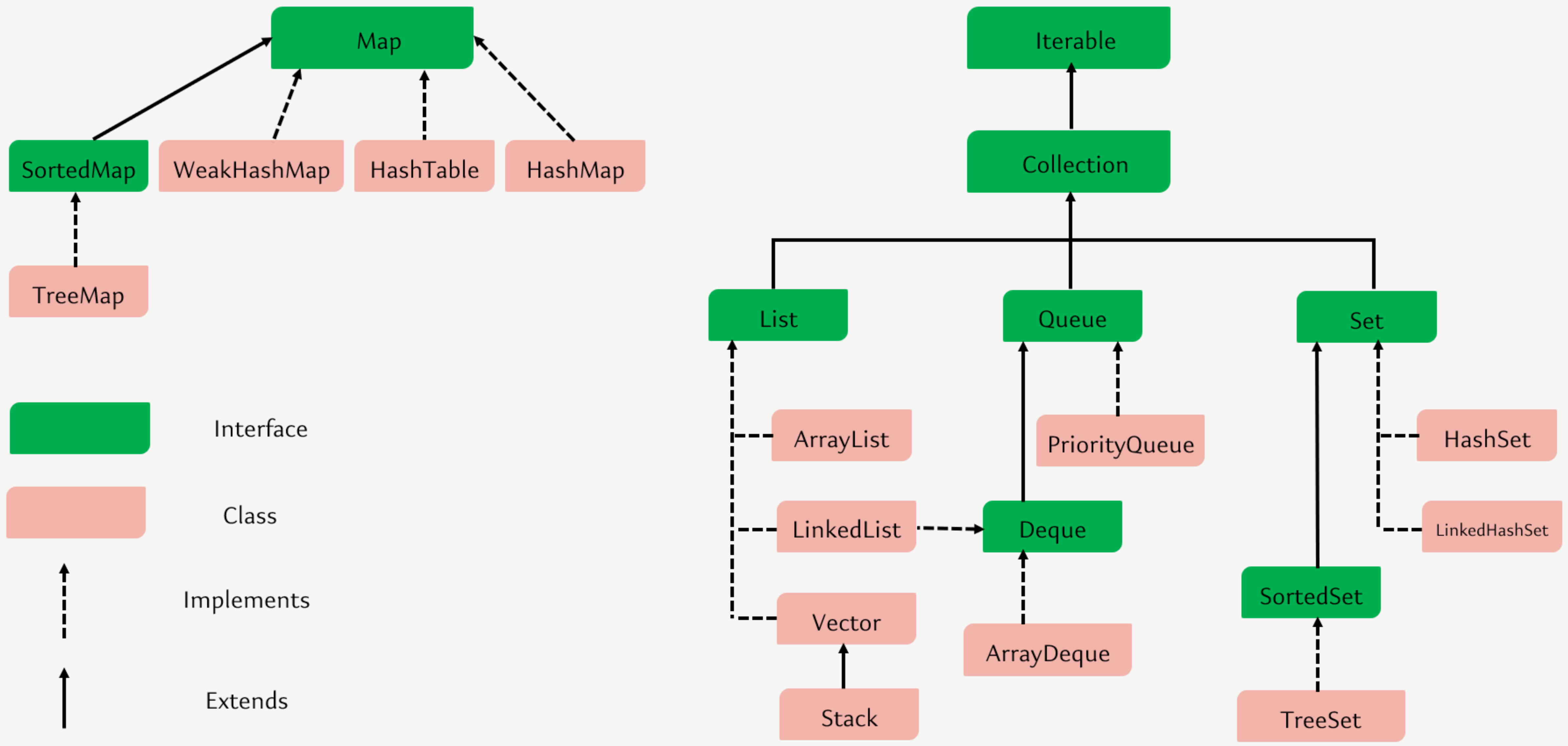
Take note that `num1` and `num2` point to distinct objects, as evidenced by the false result of `num1 == num2`. On the other hand, `num3` and `num4` reference the same object, as confirmed by the true result of `num3 == num4`.

It's important to emphasize that despite these differences in references, all four variables encapsulate the same primitive value of 49, as verified by the outcome of the `equals()` method.

It's worth mentioning that in typical scenarios, programs often deal with smaller integer literals. However, when wrapping larger integers, it's essential to be aware that the `valueOf()` method generates a new object with each invocation.

Java Collection Framework Hierarchy

eazy bytes



Java Collection Framework Hierarchy

The Java Collection Framework is a set of classes and interfaces that provide a unified and efficient way to store and manipulate groups of objects. The framework includes classes for various types of collections, such as lists, sets, queues and maps, as well as interfaces that define common operations and behaviors for these collections.

The collection classes in the framework are generic, meaning they can work with any type of object, not just a specific type. This makes the framework flexible and reusable, as we can create collections that store any type of object.

Some of the key interfaces in the collection framework include:

Collection: the root interface for all collection classes, defines basic operations such as adding, removing, and checking for the presence of elements

List: an ordered collection of elements that allows duplicates and provides methods for accessing elements by index

Set: an unordered collection of elements that does not allow duplicates

Queues: provides methods for managing elements in a specific order for processing. It follows the "first-in, first-out" (FIFO) order, where the element added first is processed first.

Map: a collection of key-value pairs that provides methods for accessing and manipulating the elements based on the keys

Reviewing all the classes and interfaces of the java.util package would require a dedicated course. So, we will just have a brief overview of the most commonly used interfaces and classes.