Collections

ArrayList

```java
public class WithoutGenerics {
    public static void main(String[] args) {
        Product product1 = new NormalGood("ikea sofa", 100);
        Product product2 = new VeblenGood("rolex watch", 1000);
        ArrayList objects = new ArrayList();
        objects.add(product1);
        objects.add("some text");
        Product product = (Product) objects.get(1);
    }
}
```

Generics

A generic class is parameterized over types

```java
public class WithGenerics {
    public static void main(String[] args) {
        Product product1 = new NormalGood("ikea sofa", 100);
        Product product2 = new VeblenGood("rolex watch", 1000);
        ArrayList<Product> objects = new ArrayList<>();
        objects.add(product1);
        //objects.add("some text");
        Product product = objects.get(0);
    }
}
```

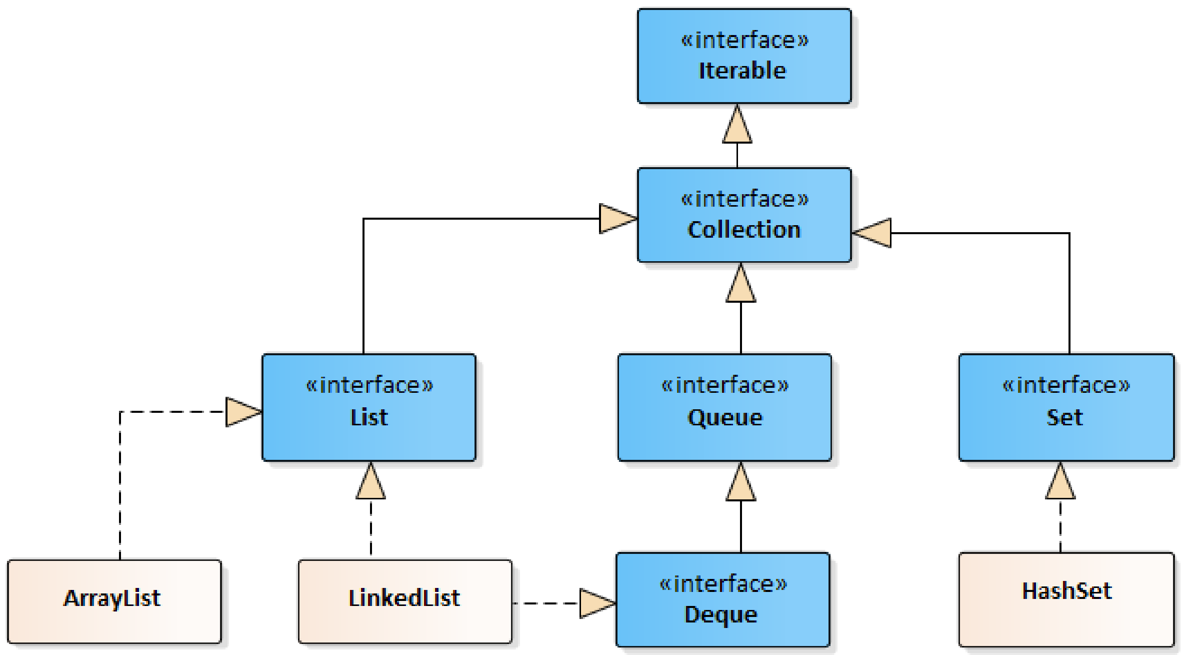
A generic class

```java
package examples.collections;

public class MyCollection<E>  {
    private Object[] elementData = new Object[10];
    private int size;
    public boolean add(E e) {
        elementData[size++] = e;
        return true;
    }
    public E get(int index){
        return (E) elementData[index];
    }
}
```

Type Parameter Naming Conventions

- E - Element
- K - Key
- N - Number
- T - Type
- V - Value

Collections Framework



Interfaces

An interface is a group of related methods with empty bodies. They're used to specify behaviour that classes must implement. Classes can implement multiple interfaces: LinkedList is an example.

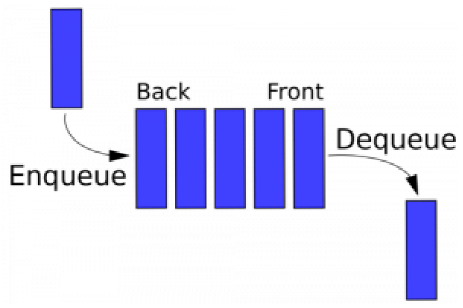
```
public interface Iterable<T> {
    Iterator<T> iterator();
}

public interface Collection<E> extends Iterable<E> {
    int size();
    boolean contains(Object o);
    boolean add(E e);
    boolean remove(Object o);
}

public interface List<E> extends java.util.Collection<E> {
    E get(int index);
    int indexOf(Object o);
}
```

Object reference conversion enables an instance of a class that implements an interface to be assigned to an interface variable

```
Collection<String> collection = new ArrayList<>();
```

Deque

A Deque (**Double Ended Queue)** allows adding and removing elements at the back or front (LIFO or FIFO)



```
public static void main(String[] args) {
    Product product1 = new Product("table", 70 , 200);
    Product product2 = new Product("table", 70 , 200);
    Product product3 = new Product("chair", 10, 30);
    Product product4 = new Product("lamp", 5 , 15);
    Deque<Product> productDeque = new LinkedList<>();
    productDeque.addFirst(product1);
    productDeque.addLast(product2);
    int n = productDeque.size();
```

Reference and Value Equality

```
public static void main(String[] args) {
    Product product1 = new Product("sofa", 200, 550);
    Product product2 = new Product("sofa", 200, 550);
    boolean areEqual = product1.equals(product2);
    System.out.println(areEqual);
}


public class ProductTest {
    @Test
    public void arrayListWorksCorrectlyWithProducct(){
        Product product1 = new Product();
        product1.setId(1L);
        Product product2 = new Product();
        product2.setId(1L);

        ArrayList<Product> products = new ArrayList<>();
        products.add(product1);
        products.remove(product2);
        Assert.assertEquals(0, products.size());
    }
}
```

Override equals

```
public class Product {
    private long id;
    private String name;
    private double costPrice;
    private double retailPrice;

    @Override
    public boolean equals(Object obj) {
        return obj instanceof Product && ((Product)obj).id == id;
    }
```

Set

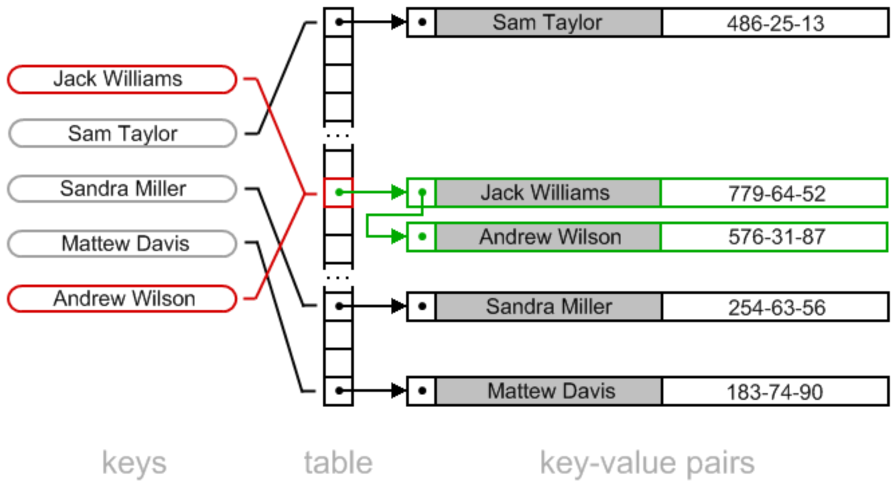
A Set is an unordered collection that doesn’t store duplicates. Elements in a Set must override hashCode and equals methods, inherited from the Object class.

```
public class ProductTest {
    @Test
    public void hashSetWorksCorrectlyWithProducct(){
        Product product1 = new Product();
        product1.setId(1L);
        Product product2 = new Product();
        product2.setId(1L);

        HashSet<Product> products = new HashSet<>();
        products.add(product1);
        products.remove(product2);
        Assert.assertEquals(0, products.size());
    }
}
```

Override hashCode

A hash table uses a hash function to compute an index into an array of buckets or slots, from which the desired value can be found. Ideally, the hash function will assign each key to a unique bucket, but it is possible that two keys will generate an identical hash causing both keys to point to the same bucket. Instead, most hash table designs assume that hash collisions—different keys that are assigned by the hash function to the same bucket—will occur and must be accommodated in some way.

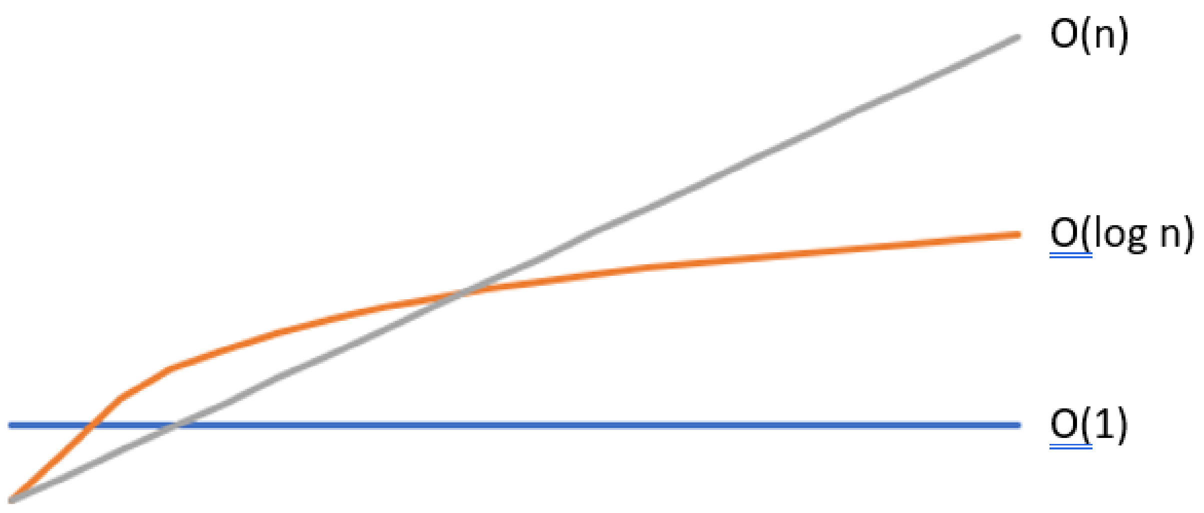
Java uses a strategy called chaining, in which each slot of the array contains a link to a singly-linked list containing key-value pairs with the same hash. New key-value pairs are added to the end of the list. Lookup algorithm searches through the list to find matching key. Initially table slots contain nulls. The value returned by hashCode() is the object's hash code, which is the object's memory address in hexadecimal. If two objects are equal, their hash code must also be equal.



```java
public class Product {
    private long id;
    private String name;
    private double costPrice;
    private double retailPrice;

    @Override
    public boolean equals(Object obj) {
        return obj instanceof Product && ((Product)obj).id == id;
    }

    @Override
    public int hashCode() {
        return (int)id;
    }
```

Big O notation

Used to classify collection methods according to how their running time increases with size. The letter O is used because the growth rate of a function is also referred to as the order of the function.

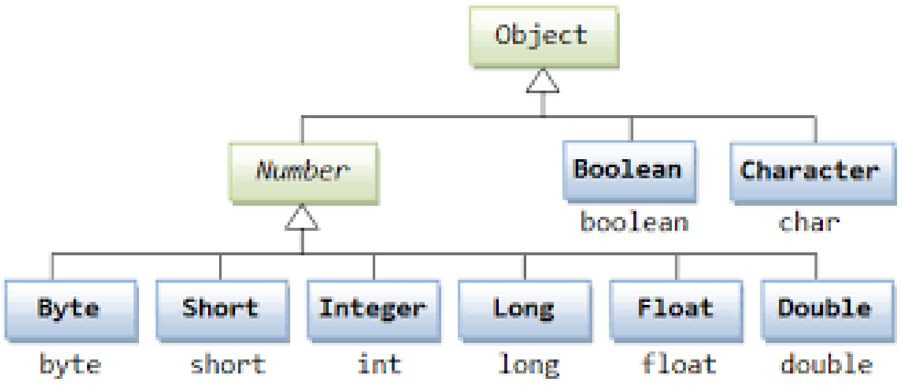


	get	add	Contains
ArrayList	O(1)	O(1)	O(n) – linear, slower
LinkedList	O(n)	O(1)	O(n) – linear, slower
HashSet		O(1)	O(1) – fixed, faster
TreeSet		O(log n)	O(log n) – logarithmic, medium

Cheat sheet

Wrapper classes

Each primitive data type has an associated class. These are known as wrapper classes because they "wrap" the primitive data type into an object of that class.



Primitives can’t be added to the collections classes, so instead use a wrapper class:

```
ArrayList<Long> numbers = new ArrayList<>();
numbers.add(17L);
```

The classes provide utility functions for converting primitive types to and from string objects.

```
String str = "5";
int i = Integer.parseInt(str);
```