# ***Collections:***

**List**: Duplikate sind erlaubt und haben eine spezifische Reihenfolge

**Set**: Duplikate sind nicht erlaubt und haben keine spezifische Reihenfolge

## List:

- ArrayList (Zugang mit index)
- LinkedList (Zugang durch iterieren)

## Set:

- HashSet (Schneller Zugang)
- LinkedHashSet
- TreeSet (Automatisches Sortieren)

## Map:

- Hash Map (Schneller Zugang)
- LinkedHashMap
- TreeMap (Automatisches Sortieren mit key)

| Class | Map | Set | List | Ordered | Sorted |
|---|---|---|---|---|---|
| `HashMap` | x | | | No | No |
| `HashTable` | x | | | No | No |
| `TreeMap` | x | | | Sorted | By *natural order* or custom comparison rules |
| `LinkedHashMap` | x | | | By insertion order or last access order | No |
| `HashSet` | | x | | No | No |
| `TreeSet` | | x | | Sorted | By *natural order* or custom comparison rules |
| `LinkedHashSet` | | x | | By insertion order | No |
| `ArrayList` | | | x | By index | No |
| `Vector` | | | x | By index | No |
| `LinkedList` | | | x | By index | No |
| `PriorityQueue` | | | | Sorted | By to-do order |

# ***Exeption Handling:***

## try,catch, finally:

```
try {
 // Action in question
}
catch (SomeException e) {
 // What to do in case of SomeException
}
finally {
 // What to do in any case (exception or no exception)
}
System.out.println("Normal flow");
```

## Runtime Exceptions:

| RuntimeException | Root Cause |
|---|---|
| ArithmeticException | Division by 0 |
| ArrayIndexOutOfBoundsException | Well, you know … |
| ClassCastException | Cast is not possible |
| EmptyStackException | Java.util.Stack.pop() |
| IllegalArgumentException | Frequently used exception to report illegal arguments |
| NullPointerException | Well, you know … |
| UnsupportedOperationException | Operation not allowed |

# Eigene Exceptions:

```
class MyException extends RuntimeException {

    private static final long serialVersionUID = 1L;

    public MyException() {
        super();
    }
    public MyException(String message) {
        super(message);
    }
}
```

# ***Observer:***

## Subject = Interface

```
public abstract void registerObserver(Observer observer);
public abstract void unregisterObserver(Observer observer);
public abstract void notify(Present present);
```

## Observer = Interface

```
public abstract void update(Present present);
```

## Klassen die Subject implimentieren -> (alt+enter):

```
List<Observer> list;
List<Present> presents;
ctor() { list = new LinkedList<>(); }
Notify(Present p) { for( Observer obs : list) { ob.update(present); } }
public void addPresentToPile(Present present) { presents.add(0,present); }
public boolean removePresentFromPile() {
      if(presents.size()==0){ return false; } notify(presents.remove(0)); return true;
  }
```

## Klassen die Observer implimentieren ->(alt+enter):

```
private String name;
ctor (String name) { this.name = name; }
public void update(Present present) { System.out.println(name); }
```

# ***Decorator:***

## Decorable = abstrakte Klasse:

String beschreibung = „unbekannt“;

getter -> beschreibung;

public abstract int getSpirit();

## Decorator = abstracte Klasse extends Decorator:

Public abstract String getBeschreibung(); //getter die nicht abstrakt sind werden hier abstrakt

## Klassen die Decorable extenden (alt+enter):

ctor() { beschreibung = „test“; }

## Klassen die Decorator extenden (alt+enter):

Decorable _dec;

ctor(Decorable dec) { _dec = dec; }