

# KOLEKCJE: LISTY, ZBIORY, KOLEJKI, MAPY

- Framework Collections
- Rodzaje kolekcji: listy, zbiory, kolejki, mapy
- Typy generyczne i opakowujące
- Sortowanie i porównywanie

**Wykład częściowo oparty na materiałach A. Jaskot**

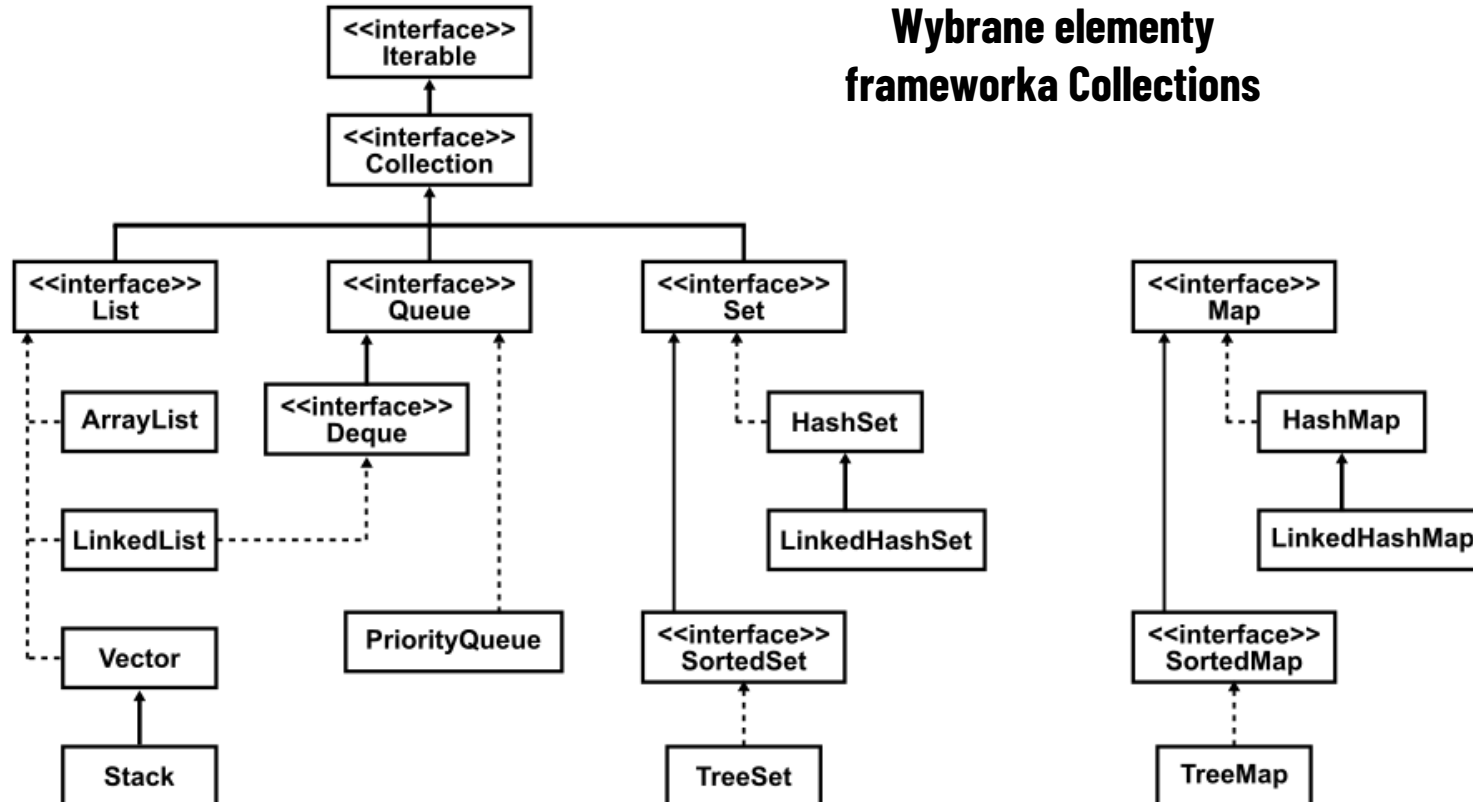
# FRAMEWORK COLLECTIONS

- zbiór interfejsów i klas służących do operacji na grupach (kolekcjach) obiektów
- zapewnia ujednolicony zestaw metod dla najczęściej używanych struktur danych

Dodatkowe funkcjonalności związane z kolekcjami zapewniają też inne, zewnętrzne biblioteki i frameworki, np. Google Guava, Eclipse Collections

# FRAMEWORK COLLECTIONS

## Wybrane elementy frameworka Collections

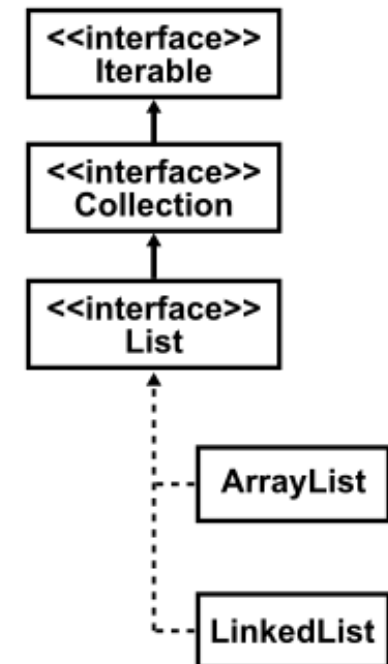


# LISTY (ang. Lists) – INTERFEJS LIST

- zachowują kolejność dodawania elementów
- pozwalają na dostęp do elementów po ich indeksie
- dopuszczają występowanie duplikatów

Najważniejsze metody:

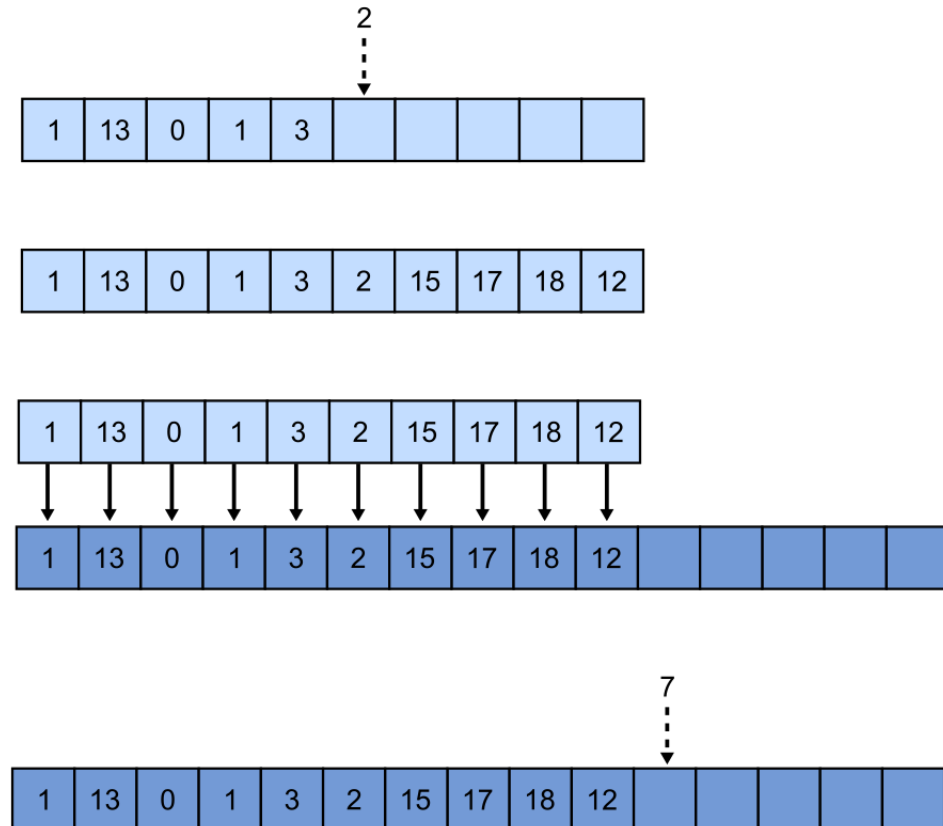
- *add(E e)*
- *get(int i)*
- *remove(int i), remove(Object o)*
- *clear()*
- *size()*
- *contains(Object o)*



# ArrayList

Także: *lista tablicowa*

- “opakowuje” tablicę i w razie potrzeby dynamicznie zwiększa jej rozmiar



# ArrayList

Także: *lista tablicowa*

- “opakowuje” tablicę i w razie potrzeby dynamicznie zwiększa jej rozmiar

```
ArrayList<String> arr1 = new ArrayList<>();  
  
arr1.add("XX"); // add(E e)  
arr1.add("YY");  
arr1.add("ZZ");  
  
String e1 = arr1.get(0); // get(int i)  
System.out.println(arr1); // toString()  
  
System.out.println(arr1.contains("ZZ")); // contains(E e)  
int s1 = arr1.size();  
  
arr1.remove(1); // remove(int i)  
arr1.clear();
```

## Uwagi:

- Domyślny rozmiar początkowy ArrayList to 10; inny rozmiar początkowy można podać w konstruktorze

```
ArrayList<String> arr1 = new ArrayList<>(100);
```

# TYPY GENERYCZNE (ang. Generic types)

```
ArrayList<String> arr1 = new ArrayList<>();  
ArrayList<String> arr2 = new ArrayList<>(100);  
  
ArrayList<Integer> arr3 = new ArrayList<>();  
  
ArrayList<Person> arr1 = new ArrayList<>();
```

**Co oznaczają nawiasy ostre przy tworzeniu obiektów klasy ArrayList?**

# TYPY GENERYCZNE (ang. Generic types)

```
public class Pair {  
    private Object firstItem;  
    private Object secondItem;  
  
    public Pair(Object firstItem, Object secondItem) {  
        this.firstItem = firstItem;  
        this.secondItem = secondItem;  
    }  
  
    public Object getFirstItem() {  
        return firstItem;    }  
  
    public void setFirstItem(Object firstItem) {  
        this.firstItem = firstItem;    }  
  
    public Object getSecondItem() {  
        return secondItem;    }  
  
    public void setSecondItem(Object secondItem) {  
        this.secondItem = secondItem;    }  
  
    // other Pair class-specific methods  
}
```

```
Pair pairOfStrings = new Pair("XXX", "YYY");  
  
Object firstAsObj = pairOfStrings.getFirstItem();  
  
String firstAsString  
    = (String) pairOfStrings.getFirstItem();  
  
pairOfStrings.setFirstItem(23);
```



# TYPY GENERYCZNE (ang. Generic types)

```
public class Pair {  
    private Object firstItem;  
    private Object secondItem;  
  
    public Pair(Object firstItem, Object secondItem) {  
        this.firstItem = firstItem;  
        this.secondItem = secondItem;  
    }  
  
    public Object getFirstItem() {  
        return firstItem;    }  
  
    public void setFirstItem(Object firstItem) {  
        this.firstItem = firstItem;    }  
  
    public Object getSecondItem() {  
        return secondItem;    }  
  
    public void setSecondItem(Object secondItem) {  
        this.secondItem = secondItem;    }  
  
    // other Pair class-specific methods  
}
```

```
Pair pairOfStrings = new Pair("XXX", "YYY");  
  
Object firstAsObj = pairOfStrings.getFirstItem();  
  
String firstAsString  
    = (String) pairOfStrings.getFirstItem();  
  
pairOfStrings.setFirstItem(23);
```

## Problem:

Zadeklarowanie pól typu Object pozwala na przechowywanie różnych typów, ale: nie zapewnia bezpieczeństwa typów i wprowadza konieczność rzutowania przy odczycie

# TYPY GENERYCZNE (ang. Generic types)

```
public class IntegerPair {
    private int firstItem;
    private int secondItem;

    public IntegerPair(int firstItem, int secondItem) {
        this.firstItem = firstItem;
        this.secondItem = secondItem;
    }

    public int getFirstItem() {
        return firstItem;    }

    public void setFirstItem(int firstItem) {
        this.firstItem = firstItem;    }

    public int getSecondItem() {
        return secondItem;    }

    public void setSecondItem(int secondItem) {
        this.secondItem = secondItem;    }

    // other IntegerPair class-specific methods
}
```

```
public class DoublePair {
    private double firstItem;
    private double secondItem;

    public DoublePair(double firstItem, double secondItem) {
        this.firstItem = firstItem;
        this.secondItem = secondItem;
    }

    public double getFirstItem() {
        return firstItem;    }

    public void setFirstItem(double firstItem) {
        this.firstItem = firstItem;    }

    public double getSecondItem() {
        return secondItem;    }

    public void setSecondItem(double secondItem) {
        this.secondItem = secondItem;    }

    // other DoublePair class-specific methods
}
```

# TYPY GENERYCZNE (ang. Generic types)

```
public class Pair<T> {  
    private T firstItem;  
    private T secondItem;  
  
    public Pair(T firstItem, T secondItem) {  
        this.firstItem = firstItem;  
        this.secondItem = secondItem;  
    }  
  
    public T getFirstItem() {  
        return firstItem;    }  
  
    public void setFirstItem(T firstItem) {  
        this.firstItem = firstItem;    }  
  
    public T getSecondItem() {  
        return secondItem;    }  
  
    public void setSecondItem(T secondItem) {  
        this.secondItem = secondItem;    }  
  
    // other Pair class-specific methods  
}
```

## TYP GENERYCZNY

- służy do tworzenia klasy uogólnionej, której podczas tworzenia obiektu jest przekazywany typ jako parameter
- parametr generyczny jest podawany w nawiasach ostrych <>

```
Pair<String> pair1 = new Pair<>("XX", "YY");  
String first1 = pair1.getFirstItem();  
  
Pair<Integer> pair2 = new Pair<>(5, 10);  
Integer first2 = pair2.getFirstItem();
```

przed Javą 7:

```
Pair<String> pair1 = new Pair<String>("XX", "YY");
```

# TYPY GENERYCZNE (ang. Generic types)

```
public class Pair<T, S> {  
    private T firstItem;  
    private S secondItem;  
  
    public Pair(T firstItem, S secondItem) {  
        this.firstItem = firstItem;  
        this.secondItem = secondItem;  
    }  
  
    public T getFirstItem() {  
        return firstItem;    }  
  
    public void setFirstItem(T firstItem) {  
        this.firstItem = firstItem;    }  
  
    public S getSecondItem() {  
        return secondItem;    }  
  
    public void setSecondItem(S secondItem) {  
        this.secondItem = secondItem;    }  
}
```

```
Pair<String, Integer> pair1 = new Pair<>("XX", 5);
```

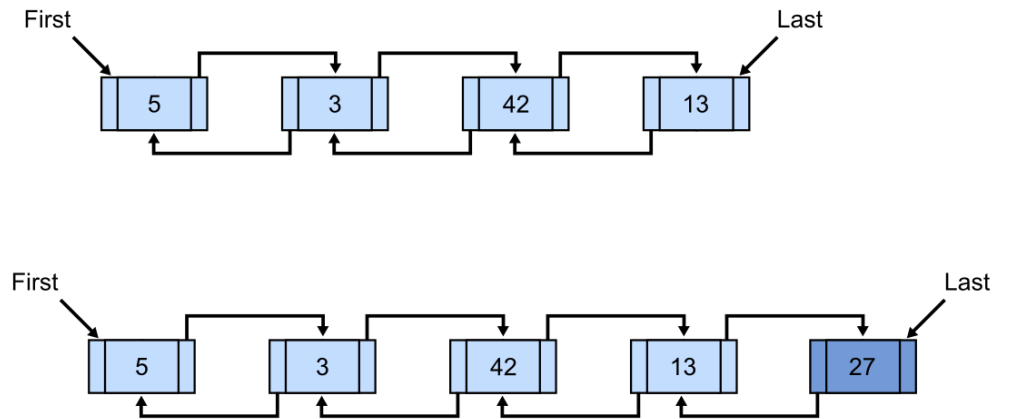
## TYP GENERYCZNY

- można zdefiniować dowolną liczbę parametrów generycznych
- zwykle stosowana jest konwencja nazw:
  - T, S, U, V itd. dla parametrów typów
  - K dla parametrów określających klucze w kolekcjach
  - E dla parametrów określających elementy w kolekcjach
  - N dla parametrów typu liczbowego

# LinkedList

Także: *lista wiązana (dwukierunkowa)*

- elementy nie są przechowywane w ciągłym bloku pamięci – każdy element (ang. node) przechowuje referencję na obiekt poprzedni i następny



# LinkedList

Także: *lista wiązana (dwukierunkowa)*

- elementy nie są przechowywane w ciągłym bloku pamięci – każdy element (ang. node) posiada referencję na obiekt poprzedni i następny

```
LinkedList<String> list1 = new LinkedList<>();

list1.add("XX"); // add(E e)
list1.add("YY");
list1.add("ZZ");

String e11 = list1.get(0); // get(int i)
System.out.println(list1); // toString()

System.out.println(list1.contains("ZZ")); // contains(E e)
int s1 = list1.size();

list1.remove(1); // remove(int i)
list1.clear();
```

## Uwagi:

- LinkedList zawiera dodatkowe metody *getFirst()* i *getLast()*

```
System.out.println(list1.getFirst());
System.out.println(list1.getLast());
```

# ArrayList vs LinkedList

Operacja	ArrayList	LinkedList
Dostęp do elementu	Szybki	Wolny
Dodawanie elementu	<ul style="list-style-type: none"><li>• Szybkie tylko na końcu</li><li>• Wolne w środku i na początku</li></ul>	Szybkie
Usuwanie elementu	<ul style="list-style-type: none"><li>• Szybkie tylko na końcu</li><li>• Wolne w środku i na początku</li></ul>	Szybkie

# ArrayList vs LinkedList

Operacja	ArrayList	LinkedList
Dostęp do elementu	Szybki	Wolny
Dodawanie elementu	<ul style="list-style-type: none"><li>• Szybkie tylko na końcu</li><li>• Wolne w środku i na początku</li></ul>	Szybkie
Usuwanie elementu	<ul style="list-style-type: none"><li>• Szybkie tylko na końcu</li><li>• Wolne w środku i na początku</li></ul>	Szybkie

**Lepszy wybór w przypadku  
wielu operacji odczytu**

**Lepszy wybór w przypadku wielu  
operacji dodawania i usuwania**



# KLASY OPAKOWUJĄCE (ang. Wrapper classes)

```
int intValue = 42;
Integer intObj = Integer.valueOf(intValue);

double doubleValue = 1.25;
Double doubleObj = Double.valueOf(doubleValue);

char charValue = 'x';
Character charObj = Character.valueOf(charValue);

Integer intObj2 = new Integer(intValue);
Double doubleObj2 = new Double(doubleValue);
Character charObj2 = new Character(charValue);
```

```
Integer intObj2 = new Integer(intValue);
Double doubleObj2 = new Double(doubleValue);
Character charObj2 = new Character(charValue);
```

'Integer(int)' is deprecated and marked for removal

## KLASY OPAKOWUJĄCE

- umożliwiają wykorzystanie typów prymitywnych w sytuacjach, kiedy dozwolone są tylko obiekty, np. w kolekcjach
- każdy typ prymitywny ma swój odpowiednik obiektowy, np. Integer, Double, Character
- do “opakowania” wartości typu prymitywnego można wykorzystać metodę *valueOf()*, ewentualnie konstruktor danej klasy opakowującej (niezalecane)
- klasy opakowujące mają także szereg dodatkowych metod, np. *Integer.max()*, *Character.isDigit()*, *Character.isLetter()*

# KLASY OPAKOWUJĄCE (ang. Wrapper classes)

```
Integer intObj = 42;  
// automatically does:  
// Integer intObj = Integer.valueOf(42);
```

```
ArrayList<Integer> intList = new ArrayList<>();  
intList.add(1);  
  
int intVal = intList.get(0);  
// automatically does:  
// int intVal = intList.get(0).intValue();
```

```
Integer x = 42;  
Integer y = 42;  
System.out.println(x.equals(y));
```

## AUTOBOXING

- proces automatycznego opakowania wartości typu prymitywnego w typ obiektowy

## AUTOUNBOXING

- proces automatycznego wypakowania typu obiektowego do typu prostego

## CACHE TYPÓW CAŁKOWITOLICZBOWYCH

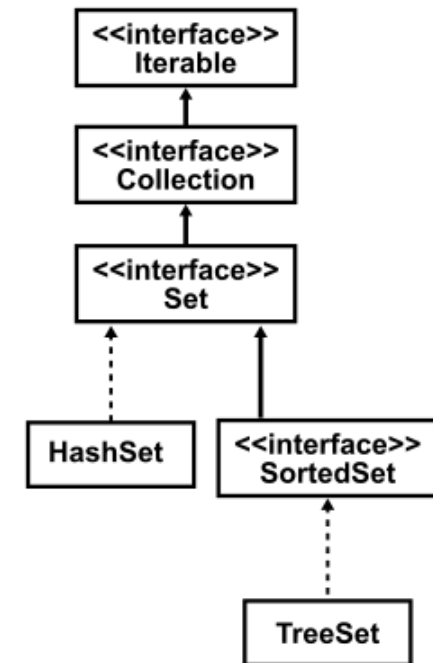
- często używane, małe wartości integerów (zwykle -128 do 127) posiadają cache “gotowych” obiektów
- użycie scache’owanych obiektów może prowadzić do błędów przy porównywaniu

# ZBIORY (ang. Sets) - INTERFEJS SET

- nie zachowują kolejności dodawania elementów
- nie dopuszczają występowania duplikatów (tylko unikalne elementy)

Najważniejsze metody:

- *add(E e)*
- *remove(Object o)*
- *clear()*
- *size()*
- *contains(Object o)*



# HashSet

- nie zapewnia stałej kolejności elementów – kolejność może się zmienić pomiędzy wykonaniami programu
- podstawowa, najbardziej wydajna implementacja zbioru
- dopuszcza dodanie elementu *null*

# HashSet

- nie zapewnia stałej kolejności elementów – kolejność może się zmienić pomiędzy wykonaniami programu
- podstawowa, najbardziej wydajna implementacja zbioru
- dopuszcza dodanie elementu *null*

```
HashSet<String> set1 = new HashSet<>();
```

```
set1.add("test"); // add(E e)
```

```
set1.add("XX");
```

```
set1.add("YY");
```

```
System.out.println(set1); // toString()
```

```
set1.add("XX");
```

```
System.out.println(set1);
```

```
[XX, YY, test]  
[XX, YY, test]
```

```
System.out.println(set1.contains("ZZ")); // contains(E e)  
int s1 = set1.size();
```

```
set1.remove("XX"); // remove(E e)
```

```
set1.clear();
```

## Uwagi:

- Brak metod wykorzystujących indeks elementu (*get(int i)*, *remove(int i)*)

# TreeSet

- przechowuje elementy w porządku naturalnym (np. liczby w kolejności rosnącej) lub posortowane w oparciu o własną kolejność (Comparator)
- zapewnia tę samą kolejność elementów pomiędzy wykonaniami programu

# TreeSet

- przechowuje elementy w porządku naturalnym (np. liczby w kolejności rosnącej) lub posortowane w oparciu o własną kolejność (Comparator)
- zapewnia tę samą kolejność elementów pomiędzy wykonaniami programu

```
TreeSet<Integer> set1 = new TreeSet<>();  
  
set1.add(15);  
set1.add(5);  
set1.add(10);  
System.out.println(set1);
```

[5, 10, 15]

```
TreeSet<String> set2 = new TreeSet<>();  
  
set2.add("Xxx");  
set2.add("test");  
set2.add("Test");  
set2.add("yy");  
System.out.println(set2);
```

[Test, Xxx, test, yy]

## Uwagi:

- TreeSet zawiera dodatkowe metody *first()* i *last()*

```
System.out.println(set1.first());  
System.out.println(set1.last());
```

- Może przechowywać tylko elementy implementujące interfejs Comparable

```
TreeSet<Person> set2 = new TreeSet<>();
```

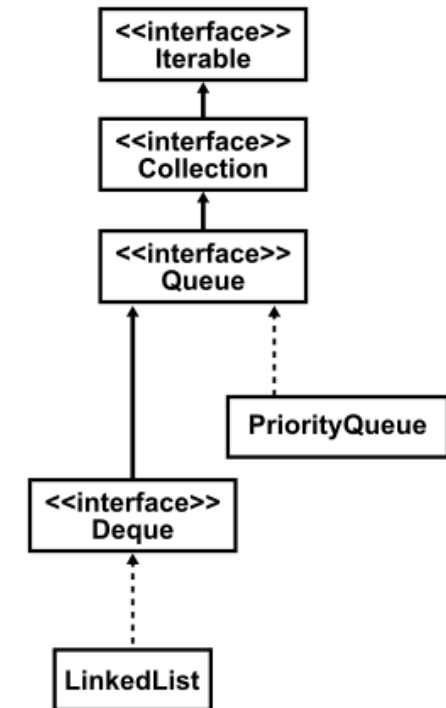
Construction of sorted collection with non-comparable elements

# KOLEJKI (ang. Queues) – INTERFEJS QUEUE

- kolekcje, w których elementy są pobierane w określonej kolejności

Najważniejsze metody:

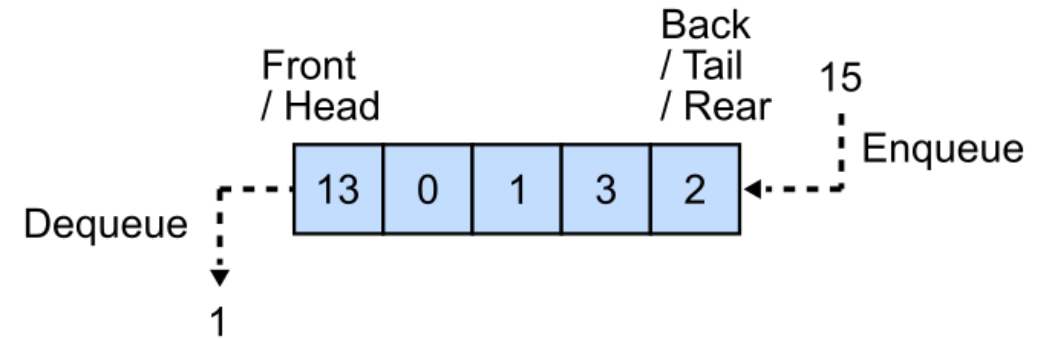
- *offer(E e)* lub *add()*
- *peek()* lub *element()*
- *poll()* lub *remove()*





# LinkedList

- reprezentuje kolejkę FIFO (z ang. first in, first out)



# LinkedList

- reprezentuje kolejkę FIFO (z ang. first in, first out)

```
LinkedList<String> queue1 = new LinkedList<>();  
  
queue1.offer("YY"); // offer(E e)  
queue1.offer("ZZ");  
queue1.offer("XX");  
System.out.println(queue1);
```

[YY, ZZ, XX]

```
System.out.println(queue1.peek());  
System.out.println(queue1);
```

YY  
[YY, ZZ, XX]

```
System.out.println(queue1.poll());  
System.out.println(queue1);
```

YY  
[ZZ, XX]

```
System.out.println(queue1.poll());  
System.out.println(queue1.poll());  
System.out.println(queue1.poll());
```

ZZ  
XX  
null

## Uwagi:

- Metody *peek()* i *poll()* zwracają null, jeżeli w kolejce nie ma więcej elementów; metody *element()* i *remove()* rzucają wyjątek `NoSuchElementException`

# PriorityQueue

- przy dodawaniu ustawia elementy zgodnie z ich priorytetem – naturalnym porządkiem lub w oparciu o własną kolejność (interfejs Comparable lub Comparator)

```
PriorityQueue<String> queue1 = new PriorityQueue<>();  
  
queue1.offer("YY");  
queue1.offer("ZZ");  
queue1.offer("XX");  
System.out.println(queue1);
```

[XX, ZZ, YY]

```
System.out.println(queue1.poll());  
System.out.println(queue1.poll());  
System.out.println(queue1.poll());  
System.out.println(queue1.poll());
```

XX  
YY  
ZZ  
null

```
PriorityQueue<Integer> queue2 = new PriorityQueue<>();  
  
queue2.offer(15);  
queue2.offer(5);  
queue2.offer(10);  
System.out.println(queue2);
```

[5, 15, 10]

```
System.out.println(queue2.poll());  
System.out.println(queue2.poll());  
System.out.println(queue2.poll());  
System.out.println(queue2.poll());
```

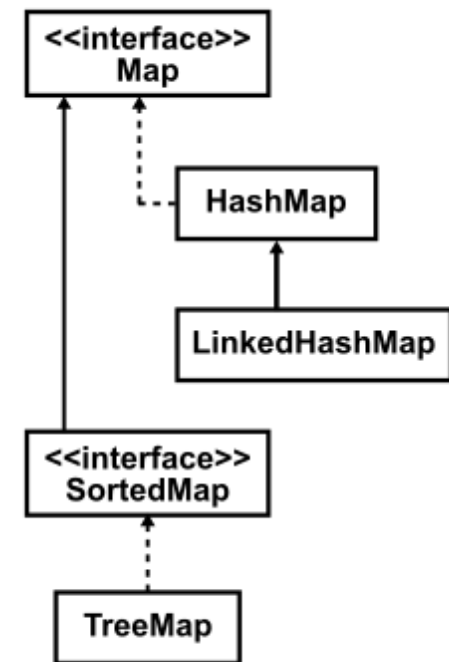
5  
10  
15  
null

# MAPY (ang. Maps) – INTERFEJS MAP

- przechowują elementy w postaci par klucz-wartość
- zarówno klucz, jak i wartość mogą być dowolnym typem obiekowym

Najważniejsze metody:

- *put(K key, V val)*
- *get(K key)*
- *remove (K key)*
- *size()*
- *keySet(), values()*
- *entrySet()*



# HashMap

- nie zapewnia stałej kolejności elementów (ani po kluczach, ani po wartościach)

```
HashMap<Integer, String> map1 = new HashMap<>();

map1.put(12345, "Xxx Yyy"); // put(K k, V v)
map1.put(96381, "Zzz Ww");
map1.put(75394, "Abc Def");

int s1 = map1.size();
System.out.println(map1); // toString()
```

```
{75394=Abc Def, 12345=Xxx Yyy, 96381=Zzz Ww}
```

```
System.out.println(map1.keySet());
System.out.println(map1.values());
System.out.println(map1.entrySet());
```

```
[75394, 12345, 96381]
[Abc Def, Xxx Yyy, Zzz Ww]
[75394=Abc Def, 12345=Xxx Yyy, 96381=Zzz Ww]
```

```
System.out.println(map1.get(96381)); // get(K k)
```

```
Zzz Ww
```

```
System.out.println(map1.remove(75394)); // remove(K k)
System.out.println(map1);
```

```
{12345=Xxx Yyy, 96381=Zzz Ww}
```

## Uwagi:

- Dodanie elementu z tym samym kluczem powoduje nadpisanie istniejącego

```
map1.put(12345, "000 Aaa");
System.out.println(map1.get(12345));
```

```
000 Aaa
```

# TreeMap

- przechowuje klucze w porządku naturalnym (np. liczby w kolejności rosnącej) lub posortowane w oparciu o własną kolejność (Comparator)

```
TreeMap<Integer, String> map1 = new TreeMap<>();
```

```
map1.put(12345, "Xxx Yyy");  
map1.put(96381, "Zzz Ww");  
map1.put(75394, "Abc Def");
```

```
System.out.println(map1);  
System.out.println(map1.keySet());  
System.out.println(map1.values());
```

```
{12345=Xxx Yyy, 75394=Abc Def, 96381=Zzz Ww}  
[12345, 75394, 96381]  
[Xxx Yyy, Abc Def, Zzz Ww]
```

```
TreeMap<String, String> map2 = new TreeMap<>();
```

```
map2.put("XX123", "Xxx Yyy");  
map2.put("Ab453", "Zzz Ww");  
map2.put("Hj999", "Abc Def");
```

```
System.out.println(map2);  
System.out.println(map2.keySet());  
System.out.println(map2.values());
```

```
{Ab453=Zzz Ww, Hj999=Abc Def, XX123=Xxx Yyy}  
[Ab453, Hj999, XX123]  
[Zzz Ww, Abc Def, Xxx Yyy]
```

# LinkedHashMap

- wykorzystuje listę wiążaną, żeby zachować kolejność, w jakiej elementy były do niej dodawane

```
LinkedHashMap<Integer, String> map1  
    = new LinkedHashMap<>();
```

```
map1.put(12345, "Xxx Yyy");  
map1.put(96381, "Zzz Ww");  
map1.put(75394, "Abc Def");
```

```
System.out.println(map1);  
System.out.println(map1.keySet());  
System.out.println(map1.values());
```

```
{12345=Xxx Yyy, 96381=Zzz Ww, 75394=Abc Def}  
[12345, 96381, 75394]  
[Xxx Yyy, Zzz Ww, Abc Def]
```

```
LinkedHashMap<String, String> map2  
    = new LinkedHashMap<>();
```


```
map2.put("XX123", "Xxx Yyy");  
map2.put("Ab453", "Zzz Ww");  
map2.put("Hj999", "Abc Def");
```

```
System.out.println(map2);  
System.out.println(map2.keySet());  
System.out.println(map2.values());
```

```
{XX123=Xxx Yyy, Ab453=Zzz Ww, Hj999=Abc Def}  
[XX123, Ab453, Hj999]  
[Xxx Yyy, Zzz Ww, Abc Def]
```

# ITEROWANIE PO KOLEKCJACH


```
ArrayList<Integer> numbers = new ArrayList<>();  
// add elements to list  
  
for (int num : numbers) {  
    System.out.println(num);  
}
```



## PĘTLA FOR-EACH

- po kolekcjach implementujących interfejs `Iterable` (listy, zbiory, kolejki) można iterować za pomocą pętli `for-each`
- mapy nie implementują interfejsu `Iterable`, ale można iterować za pomocą pętli `for-each` po ich kluczach, wartościach lub parach klucz-wartość (entries)

```
HashMap<Integer, String> students = new HashMap<>();  
// add key-value pairs to map  
  
for (int key : students.keySet()) {  
    System.out.println(key);  
}  
  
for (String val : students.values()) {  
    System.out.println(val);  
}
```





# ITEROWANIE PO KOLEKCJACH

```
LinkedList<Integer> tasks = new LinkedList<>();  
// add elements to list  
  
Iterator<Integer> iterator = tasks.iterator();  
while (iterator.hasNext()) {  
    System.out.println(iterator.next());  
}
```

```
HashSet<String> products = new HashSet<>();  
// add elements to set  
  
Iterator<String> iterator = products.iterator();  
while (iterator.hasNext()) {  
    System.out.println(iterator.next());  
}
```

## INTERFEJS ITERATOR

- po kolekcjach implementujących interfejs `Iterable` (listy, zbiory, kolejki) można iterować też za pomocą metody *iterator()*, zwracającej obiekty typu `Iterator`
- najważniejsze metody interfejsu `Iterator`:
  - *hasNext()*
  - *next()*
  - *remove()*

# KLASA COLLECTIONS

- klasa zawierająca metody użytkowe do wykonywania operacji na kolekcjach
- najważniejsze metody:
  - *min()*, *max()*
  - *frequency()*
  - *sort()*
  - *reverse()*, *shuffle()*
  - *addAll()*
  - *binarySearch()*

```
List<Integer> numbers  
    = Arrays.asList(1, 5, 0, 1, 3, 2, 9);  
  
int minNum = Collections.min(numbers);  
int maxNum = Collections.max(numbers);  
int freq1  = Collections.frequency(numbers, 1);
```

```
System.out.println(numbers);  
  
Collections.reverse(numbers);  
System.out.println(numbers);  
  
Collections.sort(numbers);  
System.out.println(numbers);  
  
Collections.shuffle(numbers);  
System.out.println(numbers);
```

```
[1, 5, 0, 1, 3, 2, 9]  
[9, 2, 3, 1, 0, 5, 1]  
[0, 1, 1, 2, 3, 5, 9]  
[3, 0, 1, 5, 9, 2, 1]
```

# SORTOWANIE KOLEKCJI

- listy można sortować za pomocą metody *Collections.sort()*
- wybrane rodzaje innych kolekcji zapewniają przechowywanie elementów w określonej kolejności (np. w naturalnym porządku):
  - zbiory: `TreeSet`
  - kolejki: `PriorityQueue`
  - mapy: `TreeMap`

# SORTOWANIE KOLEKCJI

```
List<Character> chars  
    = Arrays.asList('a', '.', 'x', 'o', '!', '-', 'c');  
  
System.out.println(chars);  
  
Collections.sort(chars);  
System.out.println(chars);
```

```
[a, ., x, o, !, -, c]  
[!, -, ., a, c, o, x]
```

```
List<String> words  
    = Arrays.asList("abc", "Abc", "Aaa", "aaa", "XYZ", "Xyz", "xyz");  
  
System.out.println(words);  
  
Collections.sort(words);  
System.out.println(words);
```

```
[abc, Abc, Aaa, aaa, XYZ, Xyz, xyz]  
[Aaa, Abc, XYZ, Xyz, aaa, abc, xyz]
```

## NATURALNY PORZĄDEK

- liczby: kolejność rosnąca
- char: kolejność rosnąca wartości Unicode
- String: porządek leksykograficzny

# SORTOWANIE KOLEKCJI

```
public class Product implements Comparable<Product> {  
    private String name;  
    private double price;  
  
    // constructor, getters, setters etc.  
  
    @Override  
    public int compareTo(Product p) {  
        if (this.price > p.price) {  
            return 1;  
        } else if (this.price == p.price)  
            return 0;  
        } else {  
            return -1;  
        }  
    }  
}
```

```
    } else if (this.price == p.price)  
        return this.name.compareTo(p.name);  
    }
```

## INTERFEJS COMPARABLE

- pozwala określić sposób porównywania obiektów klasy, która go implementuje
- zawiera jednoparametrową metodę *compareTo()*, która powinna zwracać:
  - dodatnią liczbę całkowitą, jeżeli obiekt, na którym wywołano metodę (this) jest większy niż obiekt przekazany do porównania
  - 0, jeżeli obiekty są równe
  - ujemną liczbę całkowitą, jeżeli obiekt, na którym wywołano metodę jest mniejszy

# SORTOWANIE KOLEKCJI

```
public class Product implements Comparable<Product> {  
    private String name;  
    private double price;  
  
    // constructor, getters, setters etc.  
  
    @Override  
    public int compareTo(Product p) {  
        if (this.price > p.price) {  
            return 1;  
        } else if (this.price == p.price)  
            return this.name.compareTo(p.name);  
        } else {  
            return -1;  
        }  
    }  
}
```

```
ArrayList<Product> products = new ArrayList<>();  
products.add(new Product("Milk", 3.85));  
products.add(new Product("Cheese", 7.15));  
products.add(new Product("Bread", 3.85));  
  
Collections.sort(products);  
System.out.println(products);
```

```
[Bread (3.85), Milk (3.85), Cheese (7.15)]
```

# SORTOWANIE KOLEKCJI

```
public class Product {  
    private String name;  
    private double price;  
  
    // constructor, getters, setters etc.  
}
```

```
public class PriceComparator  
    implements Comparator<Product> {  
  
    @Override  
    public int compare(Product p1, Product p2) {  
        if (p1.getPrice() > p2.getPrice())  
            return 1;  
        else if (p1.getPrice() == p2.getPrice())  
            return 0;  
        else return -1;  
    }  
}
```

## INTERFEJS COMPARATOR

- pozwala określić zewnętrzną strategię porównywania obiektów
- zawiera dwuparametrową metodę *compareTo()*, która powinna zwracać:
  - dodatnią liczbę całkowitą, jeżeli pierwszy obiekt jest większy od drugiego
  - 0, jeżeli obiekty są równe
  - ujemną liczbę całkowitą, jeżeli pierwszy obiekt jest mniejszy

# SORTOWANIE KOLEKCJI

```
public class PriceComparator
    implements Comparator<Product> {

    @Override
    public int compare(Product p1, Product p2) {
        if (p1.getPrice() > p2.getPrice()) return 1;
        else if (p1.getPrice() == p2.getPrice()) return 0;
        else return -1;
    }
}
```

```
ArrayList<Product> products = new ArrayList<>();
products.add(new Product("Milk", 3.85));
products.add(new Product("Cheese", 7.15));
products.add(new Product("Bread", 3.85));
```

```
PriceComparator priceComparator = new PriceComparator();
Collections.sort(products, priceComparator());
System.out.println(products);
```

```
[Milk (3.85), Bread (3.85), Cheese (7.15)]
```

```
public class NameComparator
    implements Comparator<Product> {

    @Override
    public int compare(Product p1, Product p2) {
        return p1.getName().compareTo(p2.getName());
    }
}
```

```
NameComparator nameComparator = new NameComparator();
TreeSet<Product> products
    = new TreeSet<>(nameComparator);
products.add(new Product("Milk", 3.85));
products.add(new Product("Cheese", 7.15));
products.add(new Product("Bread", 3.85));

System.out.println(products);
```

```
[Bread (3.85), Cheese (7.15), Milk (3.85)]
```



# KOLEKCJE A POLIMORFIZM

```
List<Integer> myList = new ArrayList<>();  
  
ArrayList<Integer> myList = new ArrayList<>(); // avoid
```

```
Set<Integer> mySet = new HashSet<>();  
  
HashSet<Integer> mySet = new HashSet<>(); // avoid
```

```
Queue<Integer> myQueue = new LinkedList<>();  
  
LinkedList<Integer> myQueue = new LinkedList<>(); // avoid
```

```
Map<String, Integer> myMap = new HashMap<>();  
  
HashMap<String, Integer> myHashMap = new HashMap<>(); // avoid
```

## ZALECANE PRAKTYKI

Jeżeli to możliwe:

- zmienne i parametry metod należy deklarować typem interfejsu, nie typem konkretnej klasy
- implementacja powinna odnosić się do interfejsu, nie konkretnej klasy