# Wykład 3

## Wyjątki,

## Kolekcje,

### vector,

### hashtable,

### properties,

### Klasy Arrays i Collections

# Wyjątki i błędy

**To niechciane zdarzenie, które zaburza poprawną pracę programu.**

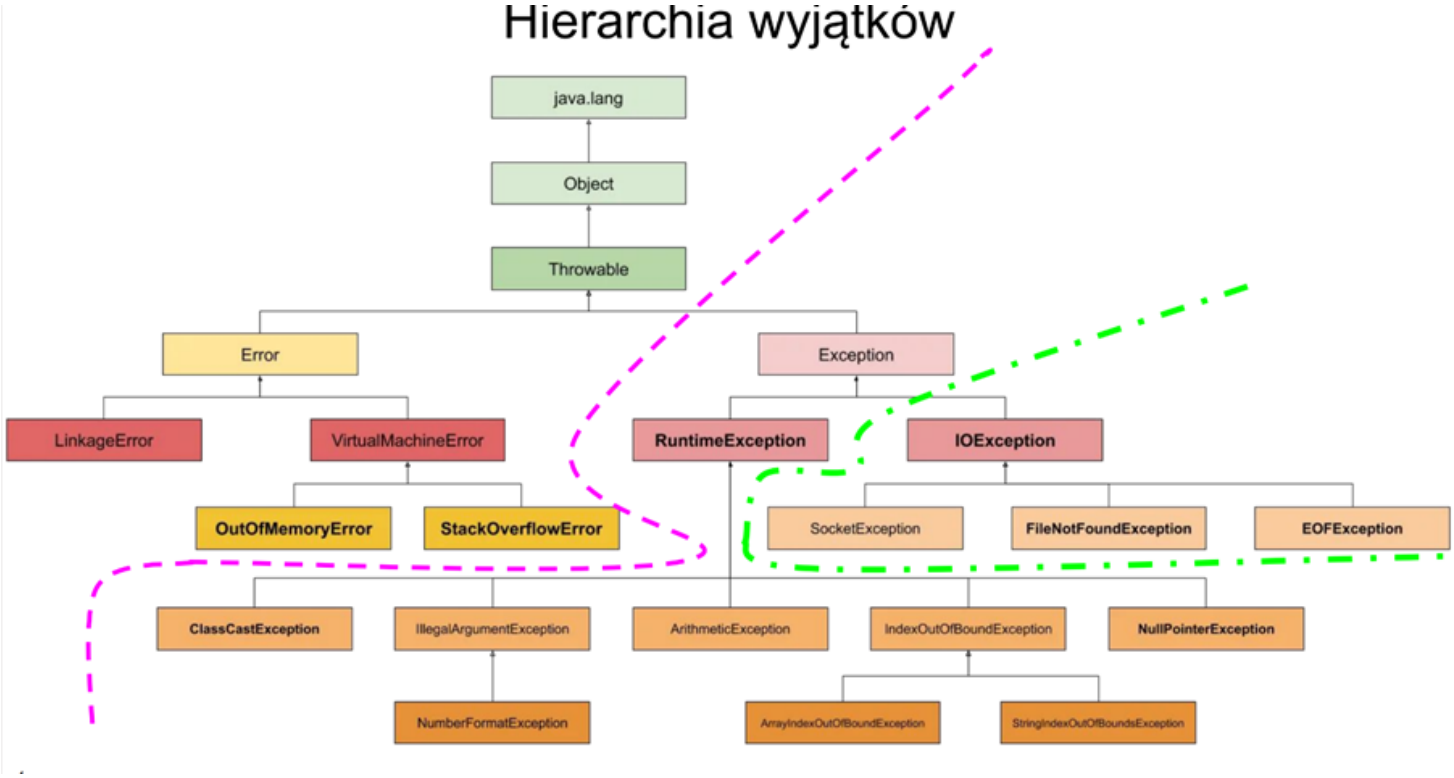
**Wyjątki (exceptions)** — to błędy, które tworzy człowiek na etapie pisania programu, mamy nad nimi większą kontrolę.
**Błędy (errors)** — to są zwykle błędy JVM np. *OutOfMemoryError*.

Błędy wykonania programu są sygnalizowane z wykorzystaniem obiektów `Throwable` . Klasa Throwable posiada dwie klasy potomne:

- wyjątki (Exception),
- błędy (Error).
  Wśród wyjątków znajduje

**RuntimeException** — określająca błędy pojawiające się w trakcie działania programu, których nie można było łatwo przewidzieć na etapie tworzenia oprogramowania np. *NullPointerException* lub *IndexOutOfBoundsException*.

Nie musimy obsługiwać wyjątków z grupy *RunTimeException*, ale **wszystkie inne są obowiązkowe do obsługi!**




# Obsługa wyjątków

- Podstawowa implementacja:

```
try {
    // Kod, który może generować wyjątek
} catch (TypWyjątku1 nazwaZmiennej1) {
    // Obsługa wyjątku TypWyjątku1
} catch (TypWyjątku2 | TypWyjątku3 nazwaZmiennej2) {
    // Obsługa wyjątku TypWyjątku2 lub 3
} catch (InneTypyWyjątków nazwaZmiennej3 ) {
    // Obsługa innych typów wyjątków
} finally {
    // Kod, który ma zostać wykonany niezależnie od tego, czy wyjątek został rzucony, czy nie
}
```

Jest to tzn. MultiCatch.

- Blok *Try-with-resources*

```
public void aMethod() throws FileNotFoundException{
    try(/*something*/){
        /*do if sucessful*/
    }catch(FileNotFoundException ex){
        ex.printStackTrace();
        ...
    }finally{
        file.close();
    }
}
```

Istnieje opcja zapisu `catch(Throwable t)` **zamiast catch(Exception e)**, ale nie powinno się tego robić, ponieważ jak widać po hierarchii, Throwable także obsługuje errory (błędy), a **błędów staramy się nie obsługiwać** i wyrzucać.

## Kolejność wykonywania

```
String s = null;
try{
    s.split(" "); // tutaj jest rzucany NullPointerException
}catch(NullPointerException ex){
    System.out.println("NullPointerException");
}catch(Exception ex){
    System.out.println("Exception");
}finally{
    System.out.println("Finally");
}
```

Wykonywane są wyjątki od góry, sekwencyjnie;

> NullPointerException
> Finally

# Ciekawostka

```
private static int printANumber(){
    try{
        return 3;
    } catch(Exception e){
        return 4;
    } finally {
        return 5;
    }
}
```

blok *return 3;* zostanie nadpisany przez *return 5;* z bloku finally{}, `wyświetli się „5”` .

## Czym różni się blok finally{} od tego co jest poniżej całej metody?

*finally{} wykona się* nawet wtedy, *gdy w bloku catch{} znajduje się return*.
finally{} **nie wykona się** tylko wtedy, gdy wywołamy w bloku catch{} `System.exit()` – metodę, która całkowicie kończy działanie wirtualnej maszyny javy.

# Porównanie 'throw' i 'throws' w Javie

| Czynnik | 'throw' w Javie | 'throws' w Javie |
|---|---|---|
| **Cel** | Słowo kluczowe 'throw' służy do **wyraźnego rzucenia wyjątku z bloku** kodu lub metody. | Słowo kluczowe 'throws' jest używane w sygnaturze metody do deklarowania wyjątków, które metoda może **potencjalnie** rzucać. |
| **Implementacja** | Słowo kluczowe 'throw' **może rzucić tylko jeden wyjątek naraz**. Nie jest możliwe jednoczesne rzucenie wielu wyjątków przy użyciu 'throw'. | Słowo kluczowe 'throws' pozwala na **deklarację wielu wyjątków**, które funkcja może rzucać. |
| **Typ wyjątku** | Słowo kluczowe 'throw' może wyrzucać tylko wyjątki typu `unchecked` (niekontrolowane). | Słowo kluczowe 'throws' może być używane do deklarowania **zarówno** wyjątków typu `checked` (kontrolowane), jak i `unchecked` (niekontrolowane). |

## Checked vs unchecked exceptions

- **Checked**
  (np. *FileNotFoundException* z grupy *IO Exception*) to wyjątki, z którymi musimy sobie poradzić i obsłużyć je jeszcze przed kompilacją, środowisko java nam o tym przypomni. Robimy to za pomocą try{}, catch{} lub throws – deklarujemy wyjątek, ale go nie obsługujemy.
- **Unchecked**
  (np. *NullPointerException* z grupy *RunTimeException*) to wyjątki, które nie przeszkadzają w kompilacji programu, ale dają o sobie znać podczas uruchomienia przez JVM.

Wszystkie podklasy klasy **RunTimeException są typu unchecked**
**Cała reszta** to wyjątki **typu checked**, trzeba je obsługiwać.

### Błędy

Błędy informują o nieprawidłowym działaniu Wirtualnej Maszyny Javy(np. *OutOfMemoryError*). Aplikacja nie powinna próbować ich obsługiwać, gdyż zwykle nie są one wynikiem jej nieprawidłowego działania.

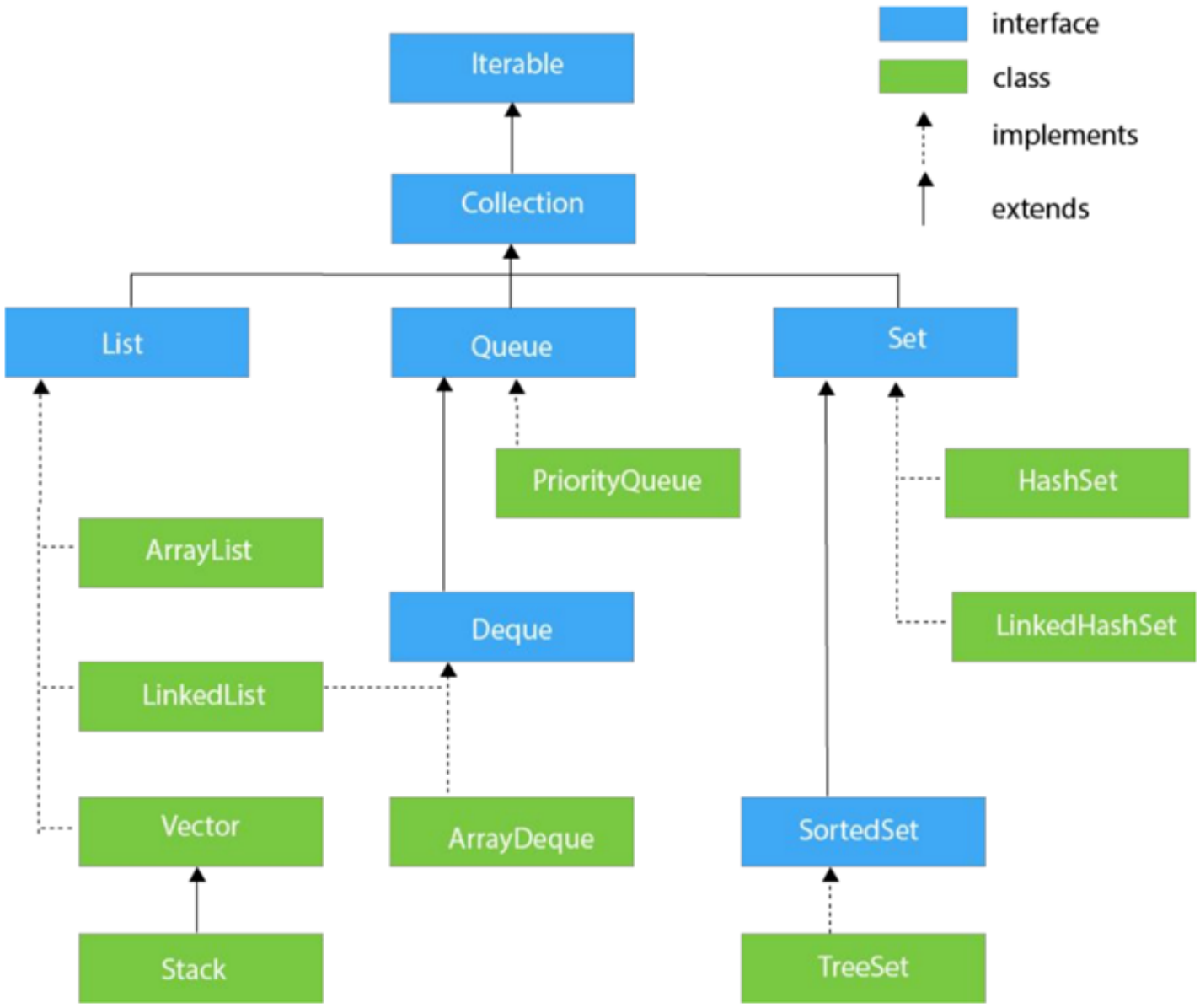
# Kolekcje

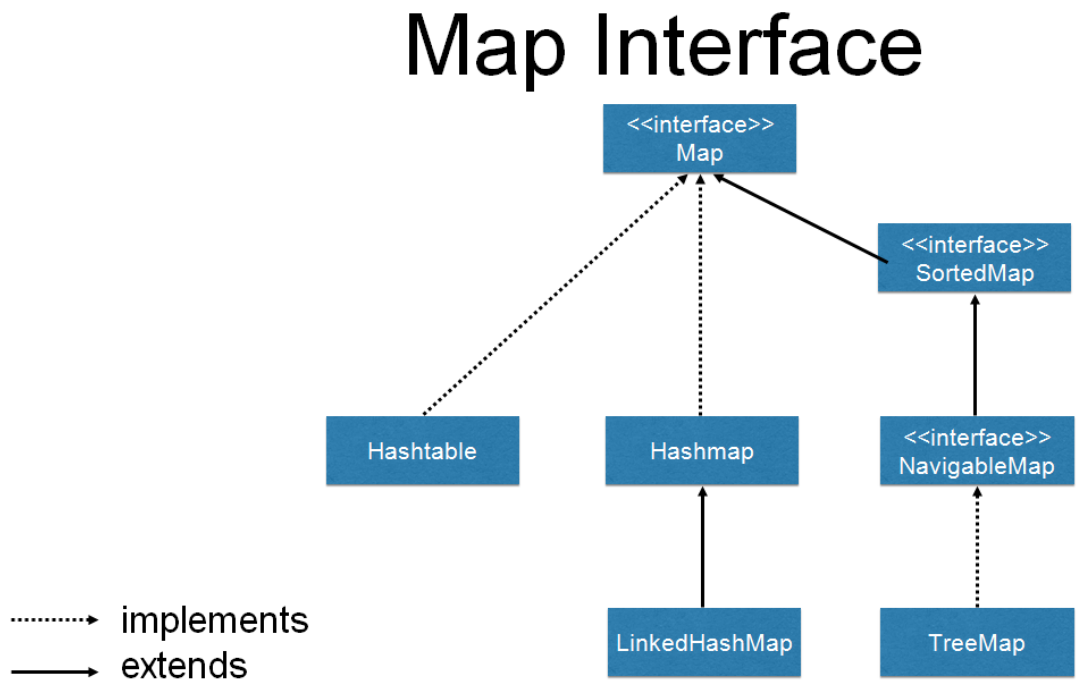
Najpopularniejszą kolekcją (zbiorem) danych jest tablica. Jednak w wielu zastosowaniach przydatne są inne struktury danych jak listy, zbiory, mapy, tablice haszujące itp. Standardowa biblioteka Javy zawiera implementacje najpopularniejszych kolekcji w pakiecie `java.util` . Ich podstawowa funkcjonalność jest zdefinowana w interfejsie *java.util.Collection.*

## Przykładowe metody z *java.util.Collection*

- ***add(Object o)*** — dodaje obiekt do kolekcji
- ***clear()*** — usuwa wszystkie obiekty z kolekcji
- ***contains(Object o)*** — sprawdza czy obiekt znajduje się w kolekcji
- ***isEmpty()*** — sprawdza czy kolekcja jest pusta
- ***iterator()*** — obiekt za pomocą którego przechodzimy przez kolekcję
- ***remove()*** — usuwa wskazany obiekt z kolekcji
- ***size()*** — zwraca rozmiar kolekcji
- ***toArray()*** — zwraca wszystkie elementy kolekcji w postaci tablicy

# Rodzaje Kolekcji






### Collection vs Collection Framework

- `Collection` — generalizacja, odnosi się do ogólnego pojęcia zbioru obiektów, gdzie te obiekty są grupowane i zarządzane razem.
- `Java Collections Framework` — jak nazwa mówi, *framework*, czyli jak coś zostało zaimplementowane - to zestaw klas i interfejsów dostępnych w pakiecie *java.util*, które dostarczają implementacje różnych typów kolekcji.

## List, Queue, Set

`List` — uporządkowana kolekcja, może zawierać duplikaty. Trzyma dane w dynamicznie alokowanej tablicy wartości

`Set` — nieuporządkowana kolekcja, **nie zawiera duplikatów**

`Queue` — implementacja kolejki **FIFO** (*offer()*, *poll()*) lub **LIFO** (*push()*, *pop()*)

`Map` — Zbiór w którym deklarujemy DWIA typy danych: klucz oraz jego wartość, klucze i wartości **są unikalne**.

### Typy kolekcji

`hash` — każdej przechowywanej wartości odpowiada jej unikalny klucz

`Linked` — zamiast tablicy wartości, każda z wartości jest przechowywana w innej części pamięci z referencjami do następnej i poprzedniej wartości

`Tree` — używa struktury drzewa zamiast tablicy wartości

**PriorityQueue** — Kolejka która ustala kombinację usuwania wartości na podstawie jakiegoś parametru. Może to być np. sortowanie po jakimś parametrze klasy.

**Deque** — *"double-ended queue"* reprezentuje kolejkę dwukierunkową, która umożliwia dodawanie lub usuwanie elementów zarówno z przodu, jak i z tyłu.

# Vector

Klasa Vector była jednym z pierwszych kontenerów w Javie 1.0, dostępnych od samego początku. Później został dodoany interfejs `List` a vector został zmodyfikowany aby go implementować.

Jest to w rzeczywistości dynamiczna tablica, której rozmiar jest *automatycznie* dostosowywany do ilości danych.

```
public class Vector<E> extends AbstractList<E> implements List<E>,
RandomAccess, Cloneable, Serializable
Vector v = new Vector();
v.add("Ala");
v.add(true); // dawniej v.add(new Boolean(true));
v.add(128.5);
v.add("Ola");
for(int i=0; i<v.size(); i++){
Object o = v.get(i);
System.out.println(o.getClass().getCanonicalName() + "\t" + o);
}
```

metoda **get(i)** w klasie Collection nie występuje, bo Collection obsługuje dowolne kolekcje, a **get()** dotyczy kolekcji, w których jest zachowana pewna kolejność elementów – dlatego metoda **get() jest w interfejsie List**.

**o.getClass().getCanonicalName()** - zwraca kanoniczną nazwę klasy danego obiektu (pełna nazwa klasy wraz z pakietem)

```
java.lang.String    Ala
java.lang. Boolean  true
java.lang.Double    128.5
java.lang.String    Ola
```

### Typ generyczny 'E'

**<E>** oznacza, że klasa Vector jest generyczna, co oznacza, że może przechowywać elementy dowolnego typu. Typ ten jest oznaczony jako `E`, ale można go zastąpić dowolnym innym typem podczas tworzenia obiektu klasy Vector.

## Różnice między ArrayList a Vector

| Cecha | ArrayList | Vector |
|---|---|---|
| **Wielowątkowość** | Nie jest domyślnie synchronizowany | Domyślnie synchronizowany |
| **Dodawanie elementów** | Szybsze, ale **niethread-safe** | Wolniejsze, ale **thread-safe** |
| **Synchronizacja** | Można zsynchronizować za pomocą `synchronizedList()` | Synchronizowany domyślnie |
| **Wyjątki przy wielowątkowości** | W przypadku braku synchronizacji, możliwe są wyjątki | Mniej podatny na wyjątki przy operacjach wielowątkowych |

# Cloneable

```
Vector v1, v2;
v1 = new Vector();
v2 = v1;                    // v2 i v1 referencje do tego samego obiektu

v2 = (Vector) v1.clone(); // tworzona jest kopia obiektu, v1 i v2 wskazuja na rozne, blizniacze
```

Implementacja interfejsu Cloneable informuje, że **nasz obiekt wspiera klonowanie**. Bazowa metoda clone jest zaimplementowana w klasie Object.

Podczas robienia kopii rzutujemy (Vector) na typ vector, bo metoda clone() pochodzi z klasy Object i służy do klonowania dowolnych obiektów, więc też **zwraca *Object***. Bez zrobienia tego przy kompilacji wystąpiłby błąd, że są niezgodne typy, bo pod typ *Vector* przypisujemy coś co jest obiektem.

> metoda **clone()** kopiuje typy prymitywne i podstawia referencje

Metoda `clone()` jest częścią interfejsu *Cloneable* w Javie. Ten interfejs nie zawiera żadnych metod, ale służy jako tzw. *marker interface*, **informujący klasę, że implementuje możliwość klonowania**. Klasa implementująca Cloneable wskazuje, że jej instancje mogą być klonowane, czyli tworzone jako kopie istniejącej instancji.

Metoda `clone()` jest **dziedziczona przez każdą klasę, która implementuje interfejs Cloneable**. Domyślna implementacja metody clone() w klasie Object wykonuje płytki klon obiektu:

> `clone()` - w klasie *object()* - kopiuje referencje do tego samego obiektu i typy prymitywne ale nie kopiuje tego co jest za tymi referencjami.

Jednak domyślne zachowanie może być niewystarczające w przypadku, gdy obiekt posiada referencje do innych obiektów. W takim przypadku konieczne może być przeprowadzenie klonowania głębokiego *(deep cloning)*, czyli kopiowania również obiektów, do których oryginalny obiekt ma referencje.
W takim przypadku deweloper musi dostosować implementację metody clone() odpowiednio do swoich potrzeb - nadpisujemy aby było to klonowanie głębokie, jeśli jest nam to potrzebne.

> Warto również zaznaczyć, że **korzystanie z metody clone() i interfejsu Cloneable jest uważane za przestarzałe**, a zaleca się korzystanie z innych mechanizmów, takich jak konstruktory kopiujące, metody fabrykujące, lub serializacja, w zależności od konkretnych potrzeb projektu.

# Serializable

Serializable to `interfejs`, którego zaimplementowanie przez daną klasę oznacza, że ta klasa jest serializable. Co to oznacza? Można stosować serializację.

- Polega to na zamianie obiektu w **sekwencję bajtów**
- Pozwala to na zapisanie obiektu w bezpieczny sposób aby wysłać go przez sieć, albo zapisać na dysku
- Dzięki temu możemy odtworzyć zapisany obiekt na innej maszynie/w późniejszym czasie przez JVM - proces ten nazywamy *deserializacją*

```
private transient String poleTransientne;
```

Jeśli pole w klasie jest oznaczone jako `transient`, nie będzie ono uwzględniane podczas serializacji.

```
import java.io.*;

public class MyClass implements Serializable {
    // ... ciało klasy ...
}

// Zapisywanie obiektu do pliku
try (ObjectOutputStream outputStream = new ObjectOutputStream(new FileOutputStream("plik.ser"))) {
    outputStream.writeObject(myObject);
}

// Odczytywanie obiektu z pliku
try (ObjectInputStream inputStream = new ObjectInputStream(new FileInputStream("plik.ser"))) {
    MyClass myObject = (MyClass) inputStream.readObject();
}
```

## Random Access

Interfejs *RandomAccess* w języku Java **jest interfejsem pustym**, co oznacza, że nie zawiera żadnych metod. Jego głównym celem jest służenie jako `znacznik`, który informuje algorytmy, że dostęp do elementów w danym obiekcie powinien być szybki i efektywny w przypadku indeksowanego dostępu.

większość implementacji standardowych kolekcji w Java, takich jak *ArrayList*, implementuje interfejs *RandomAccess*, ponieważ zapewniają efektywny dostęp do elementów poprzez indeksy. Z kolei struktury danych, które nie mają efektywnego dostępu do elementów poprzez indeksy (np. *linked listy*), nie implementują tego interfejsu.

# HashTable

*HashTable* została dodana w Javie 1.0, dostępna od samego początku, w przeciwieństwie do *HashMap* dodanej dopiero w wersji 1.2.

## Przydatne metody

- `keySet()` – zwraca wszystkie klucze ze zbioru kluczy

- `keys()` – zwraca typ *Enumeration*, który pozwoli przejść po wszystkich kluczach
- `values()` – zwraca wartości w postaci kolekcji
- `elements()` – zwraca wartości w postaci obiektu Enumeration
- `entrySet()` – zwraca zbiór, składający się z par, każda para to klucz-wartość
- `remove()` – usuwa wskazany obiekt z tablicy haszującej, podajemy klucz i jest usuwany klucz i wartość, plus jest jeszcze zwracana ta wartość

| Aspekt | HashTable | HashMap |
|---|---|---|
| **Synchronizacja** | **Synchronizowany** | **Niesynchronizowany** (domyślnie), można użyć `Collections.synchronizedMap()` dla synchronizacji |
| **Wartości *null*** | Brak akceptacji *null* dla kluczy i wartości | Akceptuje *null* dla wartości i jeden *null* klucz |
| **Dziedziczenie** | Rozszerza klasę `Dictionary` (starsza klasa) | Bezpośrednio implementuje interfejs `Map` |
| **Wydajność** | **wolniejszy** w środowisku jednowątkowym | **lepsza wydajność** w środowisku jednowątkowym |

# Properties

*Properties* to rozszerzenie tablicy haszującej

```
public class Properties extends Hashtable<Object,Object>;
```

mozna przechowywać klucze i wartości dowolnych typów, ale dodatkowo ma metody gdy para klucz-wartość zawiera tylko stringi.

nastawione na **przechowywanie par Stringów**:

```
public synchronized Object setProperty(String key, String value);
public String getProperty(String key);
```

`getProperties()` – jeśli program chce się dowiedzieć coś więcej o środowisku w jakim działa, to może dzięki getProperties() przejrzeć sobie te ustawienia systemu (zwraca ustawienia bieżącej wirtualnej maszyny javy np. katalog domowy użytkownika, system operacyjny pod którym działa, wersja itp.)

`list()` – wypisuje zawartość tej kolekcji na wskazany print stream

```
import java.io.FileOutputStream;
import java.io.IOException;
import java.util.Properties;
public class WritePropertiesToFile {
    public static void main(String[] args) {
        // Tworzenie obiektu Properties
        Properties properties = new Properties();

        // Ustawianie właściwości
        properties.setProperty("database.url","jdbc:mysql://localhost:3306/mydatabase");
        properties.setProperty("database.username", "admin");
        properties.setProperty("database.password", "secret");

        // Zapisywanie właściwości do pliku
        try (FileOutputStream fileOutputStream = new FileOutputStream("config.properties")) {
            properties.store(fileOutputStream, "Database Configuration");
            System.out.println("Pomyślnie zapisano właściwości do pliku.");
        } catch (IOException e) {
            System.err.println("Błąd podczas zapisywania właściwości do pliku: " + e.getMessage(
        }
    }
}
```

Po utworzeniu obiektu *Properties* i ustawieniu jego właściwości, używamy metody *store()* do zapisania tych właściwości do pliku "config.properties". Drugi argument metody *store()* to opcjonalny komentarz, który zostanie umieszczony na początku pliku.
Pamiętaj, że metoda *store()* może rzucać wyjątek *IOException*, więc umieszczamy ją w bloku try-catch, aby obsłużyć ewentualne problemy podczas zapisu do pliku.

- Klasa Properties “współpracuje” z plikami tekstowymi zapisanymi w

```
określonym formacie:
public synchronized void load(InputStream inStream) throws IOException

public void store(OutputStream out, String comments) throws IOException

public synchronized void loadFromXML(InputStream in)throws IOException, InvalidPropertiesFormatI

public synchronized void storeToXML(OutputStream os, String comment) throws IOException
```

# Arrays i Collections

W Javie, klasy Arrays i Collections są często używane do manipulacji i przechowywania danych.

- Arrays (Tablice):

Klasa Arrays **dostarcza różne metody do pracy z tablicami**. Udostępnia metody do `sortowania`, `wyszukiwania` i `porównywania` tablic. Zawiera także statyczne metody ułatwiające operacje na tablicach, takie jak kopiowanie, wypełnianie i konwertowanie. Przykład użycia klasy Arrays:

```
int[] numbers = {3, 1, 4, 1, 5, 9, 2, 6, 5, 3};
Arrays.sort(numbers);
System.out.println(Arrays.toString(numbers));
```

- Collections (Kolekcje):

Klasa Collections **zawiera metody pomocnicze do manipulacji obiektami kolekcji, takimi jak listy, zestawy (sety) i mapy**. Udostępnia metody do `sortowania`, `mieszania`, `odwracania` i `przekształcania kolekcji`. Działa z interfejsem Collection, co pozwala na ujednolicone operacje na różnych implementacjach kolekcji. Przykład użycia klasy Collections:

```
List<String> names = new ArrayList<>();
names.add("Alice");
names.add("Bob");
names.add("Charlie");
Collections.sort(names);
System.out.println(names);
```