

API - Interfejs programowania aplikacji^{[1][2][3][4][5]}, **interfejs programistyczny aplikacji**^{[1][6][7]}, **interfejs programu aplikacyjnego**^{[2][8]} (*ang. application programming interface, API*) – zbiór reguł ściśle opisujący, w jaki sposób programy lub podprogramy komunikują się ze sobą^[9].

API jest przede wszystkim specyfikacją wytycznych, jak powinna przebiegać interakcja między komponentami programowymi. Implementacja API jest zestawem rutyn, protokołów i rozwiązań informatycznych do budowy aplikacji komputerowych. Dodatkowo API może korzystać z komponentów **graficznego interfejsu użytkownika**. Dobre API ułatwia budowę oprogramowania, sprowadzając ją do łączenia przez programistę bloków elementów w ustalonej konwencji^[10].

Synchronization itself almost does not affect performance since java 1.4. However applications that are designed with a lot of synchronization blocks can run slowly because different threads are waiting each other.

Jeśli konstruktory w klasie będą prywatne to nie jest możliwe dziedziczenie tej klasy

Java Runtime Environment (JRE) – środowisko uruchomieniowe (środowisko wykonawcze) dla programów napisanych w języku Java jak również innych języków JVM. JRE składa się z wirtualnej maszyny Javy, klas podstawowych oraz plików pomocniczych.

Java Development Kit (JDK) – darmowe oprogramowanie firmy Sun Microsystems (będący własnością Oracle Corporation) udostępniające środowisko niezbędne do programowania w języku Java. W skład JDK wchodzi środowisko Java Runtime Environment (JRE), kompilator języka Java i interfejsy API.

Środowisko wykonawcze Java, lub inaczej JRE, to warstwa oprogramowania, która działa w systemie operacyjnym komputera i udostępnia biblioteki klas oraz inne zasoby potrzebne do uruchomienia określonego programu Java.

Środowisko JRE jest jednym z trzech wzajemnie powiązanych komponentów niezbędnych do tworzenia i uruchamiania programów Java. Pozostałe dwa komponenty to:

Pakiet Java Development Kit (JDK) to zestaw narzędzi służących do tworzenia aplikacji Java. Pakiety JDK są dostępne w różnych wersjach, pakietach i edycjach, takich jak Java Enterprise Edition (Java EE), Java Special Edition (Java SE) lub Java Mobile Edition (Java ME). Każdy pakiet JDK zawiera kompatybilne środowisko JRE, ponieważ uruchamianie programów Java jest częścią procesu ich tworzenia.

Wirtualna maszyna języka Java (JVM) wykonuje aktywne aplikacje Java. Każde środowisko JRE zawiera domyślną maszynę JVM, ale programiści mogą wybrać inną, która spełnia określone zapotrzebowania aplikacji na zasoby.

Środowisko JRE łączy kod Java utworzony przy użyciu pakietu JDK z bibliotekami niezbędnymi do uruchomienia go na maszynie JVM, a następnie tworzy instancję maszyny JVM, która wykonuje program wynikowy. Dostępne są maszyny JVM dla różnych systemów operacyjnych, a programy utworzone przy użyciu środowiska JRE będą działać na każdej z nich. Dzięki temu środowisko wykonawcze Java umożliwia uruchamianie programów Java w dowolnym systemie operacyjnym, bez żadnych modyfikacji.

1. Omów elementy twierdzenia `public static void main(String args[])`.

To jedno z najpopularniejszych pytań na rozmowie kwalifikacyjnej dla programisty dotyczących języka Java, otwierających dyskusję i dodatkowo jest to pytanie bardzo proste.

- `public` jest modyfikatorem dostępu. Używamy go do określenia dostępu do tej metody. Modyfikator tutaj jest "publiczny", więc każda klasa może mieć dostęp do tej metody.
- `static`. To słowo kluczowe Java oznacza, że używamy tej metody bez tworzenia obiektu klasy.
- `void` to typ zwracany metody. Oznacza to, że metoda ta nie zwraca żadnej wartości.
- `main` to nazwa metody. JVM traktuje tę nazwę jako punkt wejścia do aplikacji (musi mieć określony podpis). `main` jest metodą, w której następuje główne wykonanie.

- `String args[]`. To jest parametr przekazywany do metody głównej. Tutaj mamy argumenty typu `String`, które nasza aplikacja Java akceptuje podczas uruchamiania. Możesz wprowadzić je w konsoli.

2. Jaka jest różnica pomiędzy `equals()` i `==` ?

Po pierwsze, `==` jest operatorem, a `equals()` to metoda. Używamy operatora `==` do porównania referencji (lub porównania adresów), a metody `equals()` do porównania zawartości. Oznacza to, że `==` sprawdza, czy oba obiekty wskazują na tę samą lokalizację w pamięci, podczas gdy `equals()` porównuje wartości w obiektach.

3. Czy możemy wykonać program bez metody `main()`?

Wiele podstawowych pytań w rozmowach kwalifikacyjnych dotyczących języka Java jest naprawdę łatwych. Jak na przykład to. Więc, krótka odpowiedź brzmi: tak, możemy. Możemy to m.in. zrobić za pomocą statycznego bloku.

Można to zrobić również z klasami innymi niż `main`, wtedy blok `static` będzie wykonywany przed utworzeniem instancji obiektu.

Możesz użyć bloku statycznego, aby zainicjować statyczny element z danymi. Jest on wykonywany przed metodą `main`, podczas ładowania klas.

```
class Example {  
    static {  
        System.out.println("Wywoływany jest blok statyczny");  
    }  
    public static void main(String args[]) {  
        System.out.println("W głównej metodzie, teraz");  
    }  
}
```

Drukowane będzie:

```
Wywoływany jest blok statyczny  
W głównej metodzie, teraz
```

Yes, sequence is as follows:

- jvm loads class
- executes static blocks

- looks for main method and invokes it

So, if there's code in a static block, it will be executed. But there's no point in doing that.

4. Czym jest obiekt immutable? Czy można stworzyć obiekt immutable?

Nie można modyfikować obiektów klasy immutable po ich utworzeniu. Innymi słowy, kiedy już je stworzysz, nie możesz ich zmienić. Jeśli spróbujesz zmodyfikować obiekt immutable, otrzymasz nowy obiekt (klon) i podmienisz ten klon podczas tworzenia.

Dobrym przykładem jest String. W Javie jest immutable (niezmienny). Oznacza to, że nie możesz zmienić samego obiektu String, ale możesz zmienić odniesienie do obiektu.

You make a class immutable like this:

```
public final class Immutable
{
    Private final String name;

    public Immutable(String name)
    {
        this.name = name;
    }

    public String getName() { return this.name; }

    // No setter;
}
```

Following are the requirements to make a Java class immutable:

- **Class** must be declared as final (So that child classes can't be created)
- **Members** in the class must be declared as final (So that we can't change the value of it after object creation)
- **Write Getter methods** for all the variables in it to get **Members** values
- **No Setters method**

5. Ile obiektów jest tworzonych w poniższym kodzie?

Jest to pytanie techniczne dotyczące języka Java, które może zastąpić punkt 4, powyżej.

```
String s1="Witaj";
```

```
String s2="Witaj";
```

```
String s3="Witaj";
```

Odpowiedź to "tylko jeden", ponieważ Java posiada String Pool. Kiedy tworzymy obiekt String za pomocą operatora new (), Java tworzy nowy obiekt w pamięci sterty. Jeśli użyjemy

składni literału String, tak jak w naszym przykładzie powyżej, może ona zwrócić obiekt z puli String, jeżeli taki już istnieje.

String Pool jest miejscem na stercie (ang. *heap*), dzięki któremu JVM może optymalizować użycie pamięci. Każdy String jest przechowywany w String Pool, ale gdy wartość Stringa jest już dostępna w puli, to zamiast tworzyć nowy obiekt, po prostu tworzona jest referencja do już istniejącego łańcucha. Chyba, że String został stworzony przy użyciu słowa kluczowego `new`, wówczas i tak tworzony jest nowy obiekt. Stąd inny `hashCode` dla pola `testC`.

Spójrz teraz na trzy ostatnie wyniki. Mimo, że ich wartości będą takie same i w każdym przypadku będzie to `"TEST"`, to każde pole zwraca inny `hashCode`. Dzieje się tak, ponieważ implementacja metody `toUpperCase()` zwraca Stringa ze słowem `new`.

Dlaczego jednak wyszliśmy od niemutowalności łańcuchów? Gdyby Stringi można było modyfikować String Pool nie mógłby zaistnieć. Co gdybyśmy zmieni wartość pola `testA`, czy wtedy zmieniałaby się wartość pola `testB`, skoro w obu przechowywana jest ta sama referencja?

Porównania Stringów `==` czy `equals()`?

Wiesz już może, jakie są różnice między porównaniem `==` a metodą `equals()` w Javie? Pokrótkę w przypadku obiektów porównanie `==` sprawdza czy referencje są takie same. Natomiast metoda `equals()` sprawdza czy dane obiekty są takie same według określonych założeń, i tak implementacja metody `equals()` dla Stringa sprawdza czy wszystkie znaki w porównywanych Stringach są takie same.

Jakie więc otrzymamy efekty porównania Stringów `testA`, `testB` i `testC`?

```
System.out.println(testA == testB); // true
System.out.println(testA == testC); // false
System.out.println(testA.equals(testB)); // true
System.out.println(testA.equals(testC)); // true
```

Istnieje jednak sposób by za pomocą `==` uzyskać zgodność dla pól `testA` i `testC`. Służy do tego metoda `intern()` wywoływana na Stringu. Dokonuje ona tzw. internowania, czyli przechowywania obiektu w puli unikatowych wartości. Czyli takiej puli, w której nie ma możliwości utworzenia dwóch łańcuchów o takiej samej treści. Dwa Stringi utworzone przez słowo kluczowe `new`, dzięki internowaniu będą zwracać tę samą referencję i można je porównać za pomocą `==`.

```
System.out.println(testA == testC.intern()); // true
System.out.println(new String("test").intern() == testC.intern()); // true
```

Mam nadzieję, że przybliżyłem temat i chociaż może on wydawać się mało istotny, sam spotkałem się kiedyś z pytaniem rekrutacyjnym o internowanie Stringów.

6. Ile obiektów jest tworzonych w poniższym kodzie?

```
String s = new String("Witaj");
```

Tu stworzono 2 obiekty. Jeden znajduje się w puli stałych typu String, a drugi w stercie.

7. Jaka jest różnica pomiędzy klasami String, StringBuilder i StringBuffer w Javie?

To jedno z najczęściej zadawanych pytań podczas rozmów rekrutacyjnych dotyczących języka Java.

Po pierwsze, String to klasa niezmienna. Oznacza to, że nie możesz modyfikować jej zawartości po utworzeniu. Ale StringBuffer i StringBuilder to klasy zmienne, więc masz możliwość zmodyfikować je później. Jeśli spróbujemy zmienić zawartość obiektu String, JVM utworzy nowy ciąg pozostawiając oryginalny bez zmian. Dlatego wydajność jest lepsza w przypadku StringBuffer niż w String.

Główna różnica między StringBuffer i StringBuilder polega na tym, że metody StringBuffer są synchronizowane, a metody StringBuilder nie.

No.	StringBuffer	StringBuilder
1)	StringBuffer is <i>synchronized</i> i.e. thread safe. It means two threads can't call the methods of StringBuffer simultaneously.	StringBuilder is <i>non-synchronized</i> i.e. not thread safe. It means two threads can call the methods of StringBuilder simultaneously.
2)	StringBuffer is <i>less efficient</i> than StringBuilder.	StringBuilder is <i>more efficient</i> than StringBuffer.
3)	StringBuffer was introduced in Java 1.0	StringBuilder was introduced in Java 1.5

8. Czy jest jakaś różnica między obiektami typu String, gdy pierwszy utworzony został jako literał, a drugi za pomocą operatora new()?

When we create a String object using the new() operator, it always creates a new object in heap memory. On the other hand, if we create an object using String literal syntax e.g. "Baeldung", it may return an existing object from the String pool, if it already exists. String pool is nothing but a storage area in Java heap where string literals stores.

9. Czy można nadpisać metodę private lub static w Javie?

To dla początkujących programistów Java dość podstępne pytanie rekrutacyjne w Java. Tak na prawdę nie można nadpisać metody private lub static w Javie.

Nie można nadpisywać metod private, ponieważ zakres specyfikatora dostępu prywatnego ograniczony jest do klasy. Aby coś nadpisać, musimy mieć klasę podstawową i potomną. Jeśli metoda nadklasy jest private, klasa potomna nie może jej używać, a metody z klasy podrzędnej będą traktowane jako nowe metody (a nie nadpisanie).

Metody static również nie mogą być nadpisane, ponieważ metody static są częścią samej klasy. Nie są częścią żadnego obiektu klasy. Oczywiście, możesz zadeklarować tę samą metodę static z tym samym podpisem w klasach potomnych, lecz znowu będą one traktowane jako nowe metody.

10. Jaka jest różnica pomiędzy klasą abstrakcyjną - abstract class, a interfejsem - interface

To popularne pytanie na rozmowę kwalifikacyjną programisty Java, które opiera się na zasadach OOP. Przede wszystkim w Javie interface definiuje zachowanie, podczas gdy abstract class tworzy hierarchię.

Abstract Class	Interfaces
Klasa abstrakcyjna może dostarczyć kompletny, domyślny kod i/lub tylko szczegóły, które należy nadpisać	Interfejs nie może w ogóle dostarczyć żadnego kodu, tylko szablon (obecnie mogą występować metody prywatne i domyślne)
W przypadku klasy abstrakcyjnej, klasa może rozszerzać tylko jedną klasę abstrakcyjną	Klasa może implementować kilka interfejsów
Klasa abstrakcyjna może mieć metody nieabstrakcyjne	Wszystkie metody interfejsu są abstrakcyjne (oprócz domyślnych i prywatnych)
Klasa abstrakcyjna może mieć zmienne instancji	Interfejs nie może mieć zmiennych instancji (Wszystkie statyczne i finalne)
Klasa abstrakcyjna może mieć dowolną widoczność: publiczną, prywatną,	Widoczność interfejsu musi być publiczna

chronioną, package private	(lub) package private
Jeśli dodamy nową metodę do klasy abstrakcyjnej to mamy możliwość dostarczenia domyślnej implementacji i dlatego cały istniejący kod może działać poprawnie	Jeśli dodamy nową metodę do interfejsu, musimy prześledzić wszystkie implementacje interfejsu i zdefiniować implementację dla nowej metody
Klasa abstrakcyjna może zawierać konstruktory	Interfejs nie może zawierać konstruktorów
Klasy abstrakcyjne są szybkie	Interfejsy są powolne, ponieważ wymaga dodatkowego pośrednictwa, aby znaleźć odpowiednią metodę w rzeczywistej klasie

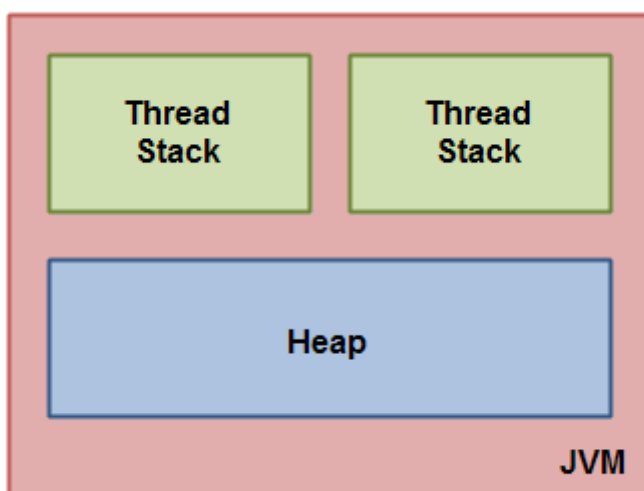
W interfejsie ponadto można definiować metody prywatne (muszą mieć ciało) i metody domyślne. Metody domyślne tylko w interfejsach. Metody domyślne można nadpisywać.

11. Czy możemy zadeklarować zmienne static i metody w klasie abstract?

Tak, można zadeklarować zmienne static i metody w klasie abstract. Nie ma wymogu, aby obiekt miał dostęp do kontekstu statycznego. Na przykład, możemy uzyskać dostęp do statycznego kontekstu zadeklarowanego wewnątrz klasy abstract poprzez użycie nazwy klasy abstract.

12. Jakie typy obszarów pamięci są przydzielane przez JVM?

The Java memory model used internally in the JVM divides memory between thread stacks and the heap. This diagram illustrates the Java memory model from a logic perspective:



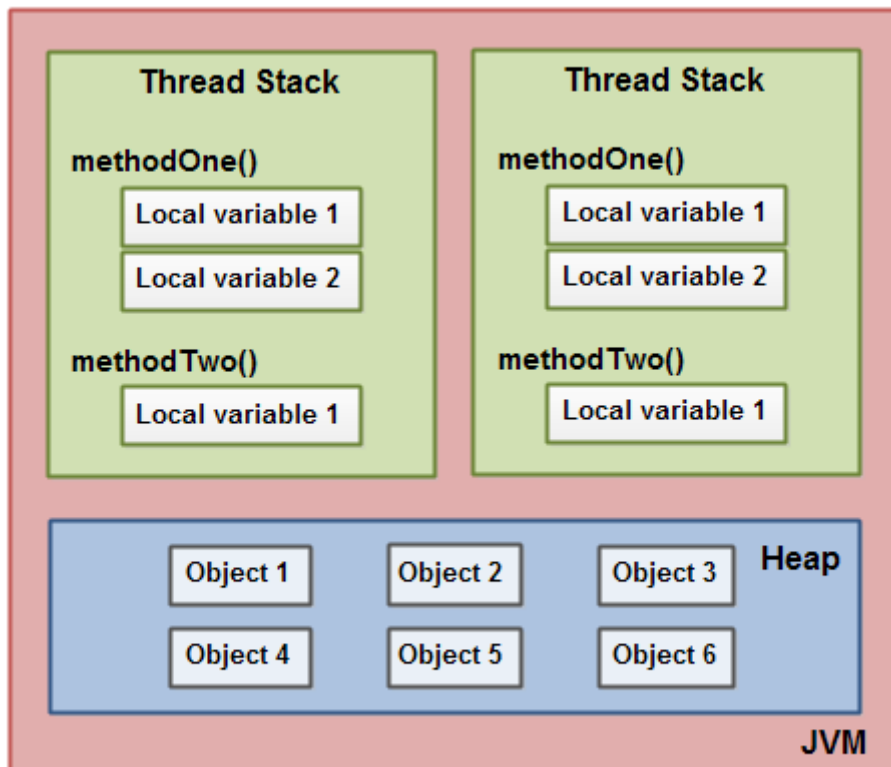
Each thread running in the Java virtual machine has its own thread stack. The thread stack contains information about what methods the thread has called to reach the current point of execution. I will refer to this as the "call stack". As the thread executes its code, the call stack changes.

The thread stack also contains all local variables for each method being executed (all methods on the call stack). A thread can only access its own thread stack. Local variables created by a thread are invisible to all other threads than the thread who created it. Even if two threads are executing the exact same code, the two threads will still create the local variables of that code in each their own thread stack. Thus, each thread has its own version of each local variable.

All local variables of primitive types (boolean, byte, short, char, int, long, float, double) are fully stored on the thread stack and are thus not visible to other threads. One thread may pass a copy of a primitive variable to another thread, but it cannot share the primitive local variable itself.

The heap contains all objects created in your Java application, regardless of what thread created the object. This includes the object versions of the primitive types (e.g. Byte, Integer, Long etc.). It does not matter if an object was created and assigned to a local variable, or created as a member variable of another object, the object is still stored on the heap.

Here is a diagram illustrating the call stack and local variables stored on the thread stacks, and objects stored on the heap:



A local variable may be of a primitive type, in which case it is totally kept on the thread stack. A local variable may also be a reference to an object. In that case the reference (the local variable) is stored on the thread stack, but the object itself is stored on the heap.

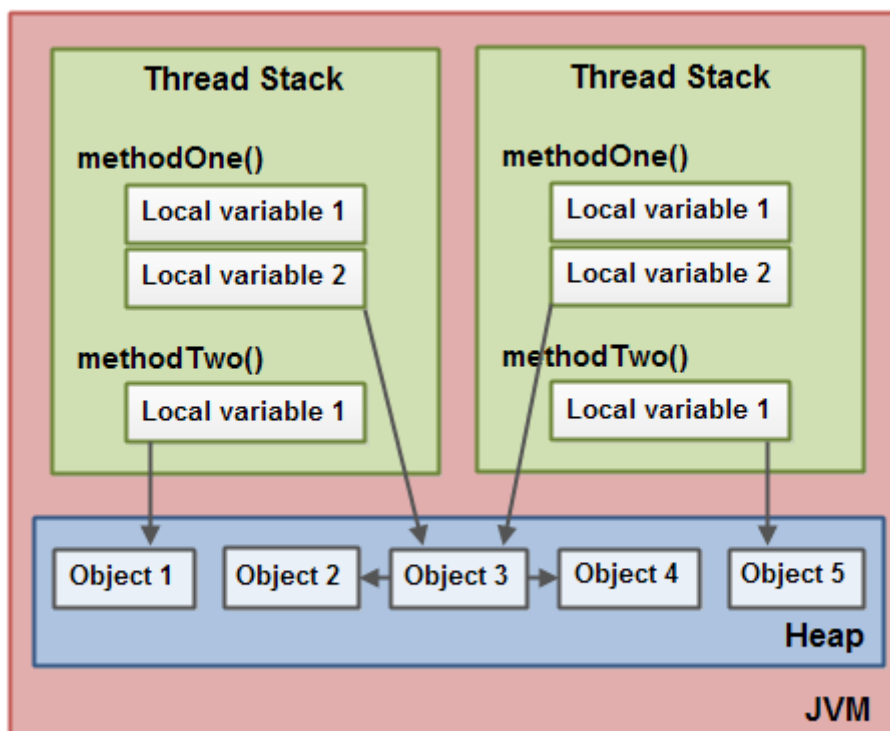
An object may contain methods and these methods may contain local variables. These local variables are also stored on the thread stack, even if the object the method belongs to is stored on the heap.

An object's member variables are stored on the heap along with the object itself. That is true both when the member variable is of a primitive type, and if it is a reference to an object.

Static class variables are also stored on the heap along with the class definition.

Objects on the heap can be accessed by all threads that have a reference to the object. When a thread has access to an object, it can also get access to that object's member variables. If two threads call a method on the same object at the same time, they will both have access to the object's member variables, but each thread will have its own copy of the local variables.

Here is a diagram illustrating the points above:



Two threads have a set of local variables. One of the local variables (Local Variable 2) point to a shared object on the heap (Object 3). The two threads each have a different reference to the same object. Their references are local variables and are thus stored in each thread's thread stack (on each). The two different references point to the same object on the heap, though.

Notice how the shared object (Object 3) has a reference to Object 2 and Object 4 as member variables (illustrated by the arrows from Object 3 to Object 2 and Object 4). Via these member variable references in Object 3 the two threads can access Object 2 and Object 4.

The diagram also shows a local variable which point to two different objects on the heap. In this case the references point to two different objects (Object 1 and Object 5), not the same object. In theory both threads could access both Object 1 and Object 5, if both threads had references to both objects. But in the diagram above each thread only has a reference to one of the two objects.

So, what kind of Java code could lead to the above memory graph? Well, code as simple as the code below:

```
public class MyRunnable implements Runnable() {
```

```

public void run() {
    methodOne();
}

public void methodOne() {
    int localVariable1 = 45;

    MySharedObject localVariable2 =
        MySharedObject.sharedInstance;

    //... do more with local variables.

    methodTwo();
}

public void methodTwo() {
    Integer localVariable1 = new Integer(99);

    //... do more with local variable.
}
}

public class MySharedObject {

    //static variable pointing to instance of
    MySharedObject

    public static final MySharedObject sharedInstance =
        new MySharedObject();

    //member variables pointing to two objects on the
    heap

    public Integer object2 = new Integer(22);
    public Integer object4 = new Integer(44);

    public long member1 = 12345;
    public long member2 = 67890;
}

```

If two threads were executing the run() method then the diagram shown earlier would be the outcome. The run() method calls methodOne() and methodOne() calls methodTwo().

`methodOne()` declares a primitive local variable (`localVariable1` of type `int`) and an local variable which is an object reference (`localVariable2`).

Each thread executing `methodOne()` will create its own copy of `localVariable1` and `localVariable2` on their respective thread stacks.

The `localVariable1` variables will be completely separated from each other, only living on each thread's thread stack. One thread cannot see what changes another thread makes to its copy of `localVariable1`.

Each thread executing `methodOne()` will also create their own copy of `localVariable2`. However, the two different copies of `localVariable2` both end up pointing to the same object on the heap. The code sets `localVariable2` to point to an object referenced by a static variable. There is only one copy of a static variable and this copy is stored on the heap. Thus, both of the two copies of `localVariable2` end up pointing to the same instance of `MySharedObject` which the static variable points to. The `MySharedObject` instance is also stored on the heap. It corresponds to Object 3 in the diagram above.

Notice how the `MySharedObject` class contains two member variables too. The member variables themselves are stored on the heap along with the object. The two member variables point to two other `Integer` objects. These `Integer` objects correspond to Object 2 and Object 4 in the diagram above.

Notice also how `methodTwo()` creates a local variable named `localVariable1`. This local variable is an object reference to an `Integer` object. The method sets the `localVariable1` reference to point to a new `Integer` instance.

The `localVariable1` reference will be stored in one copy per thread executing `methodTwo()`. The two `Integer` objects instantiated will be stored on the heap, but since the method creates a new `Integer` object every time the method is executed, two threads executing this method will create separate `Integer` instances. The `Integer` objects created inside `methodTwo()` correspond to Object 1 and Object 5 in the diagram above.

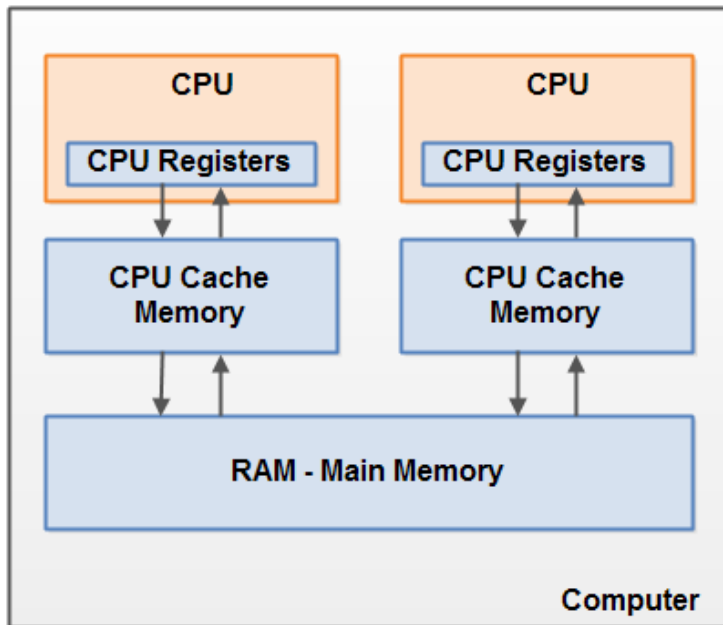
Notice also the two member variables in the class `MySharedObject` of type `long` which is a primitive type. Since these variables are member variables, they are still stored on the heap along with the object. Only local variables are stored on the thread stack.

Hardware Memory Architecture

Modern hardware memory architecture is somewhat different from the internal Java memory model. It is important to understand the hardware memory architecture too, to understand how the Java memory model works with it. This section describes the common

hardware memory architecture, and a later section will describe how the Java memory model works with it.

Here is a simplified diagram of modern computer hardware architecture:



A modern computer often has 2 or more CPUs in it. Some of these CPUs may have multiple cores too. The point is, that on a modern computer with 2 or more CPUs it is possible to have more than one thread running simultaneously. Each CPU is capable of running one thread at any given time. That means that if your Java application is multithreaded, one thread per CPU may be running simultaneously (concurrently) inside your Java application.

Each CPU contains a set of registers which are essentially in-CPU memory. The CPU can perform operations much faster on these registers than it can perform on variables in main memory. That is because the CPU can access these registers much faster than it can access main memory.

Each CPU may also have a CPU cache memory layer. In fact, most modern CPUs have a cache memory layer of some size. The CPU can access its cache memory much faster than main memory, but typically not as fast as it can access its internal registers. So, the CPU cache memory is somewhere in between the speed of the internal registers and main memory. Some CPUs may have multiple cache layers (Level 1 and Level 2), but this is not so important to know to understand how the Java memory model interacts with memory. What matters is to know that CPUs can have a cache memory layer of some sort.

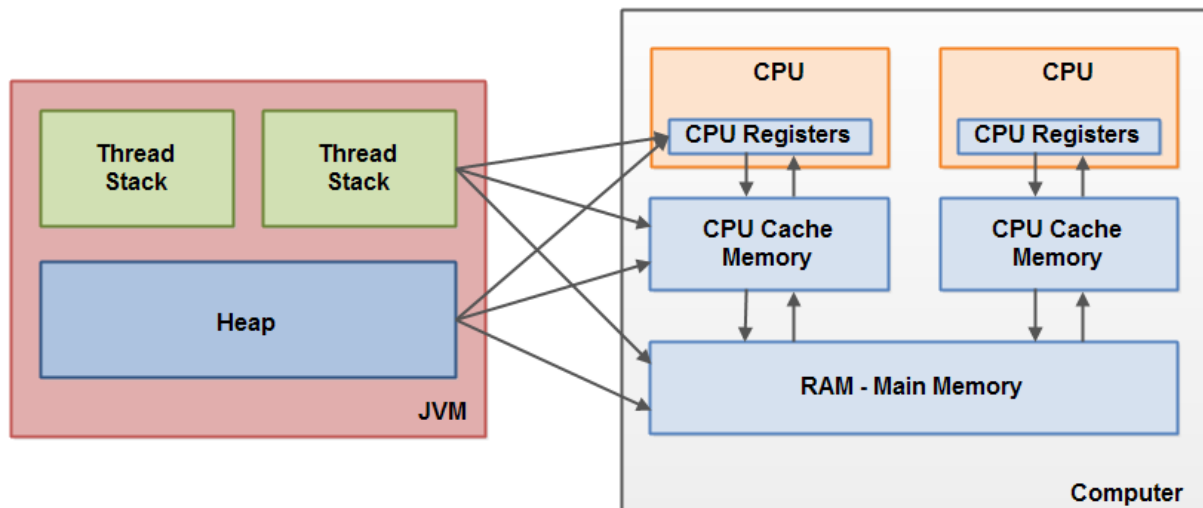
A computer also contains a main memory area (RAM). All CPUs can access the main memory. The main memory area is typically much bigger than the cache memories of the CPUs.

Typically, when a CPU needs to access main memory it will read part of main memory into its CPU cache. It may even read part of the cache into its internal registers and then perform operations on it. When the CPU needs to write the result back to main memory it will flush the value from its internal register to the cache memory, and at some point flush the value back to main memory.

The values stored in the cache memory is typically flushed back to main memory when the CPU needs to store something else in the cache memory. The CPU cache can have data written to part of its memory at a time, and flush part of its memory at a time. It does not have to read / write the full cache each time it is updated. Typically the cache is updated in smaller memory blocks called "cache lines". One or more cache lines may be read into the cache memory, and one or more cache lines may be flushed back to main memory again.

Bridging The Gap Between The Java Memory Model And The Hardware Memory Architecture

As already mentioned, the Java memory model and the hardware memory architecture are different. The hardware memory architecture does not distinguish between thread stacks and heap. On the hardware, both the thread stack and the heap are located in main memory. Parts of the thread stacks and heap may sometimes be present in CPU caches and in internal CPU registers. **This is illustrated in this diagram:**



When objects and variables can be stored in various different memory areas in the computer, certain problems may occur. The two main problems are:

- Visibility of thread updates (writes) to shared variables.
- Race conditions when reading, checking and writing shared variables.

Both of these problems will be explained in the following sections.

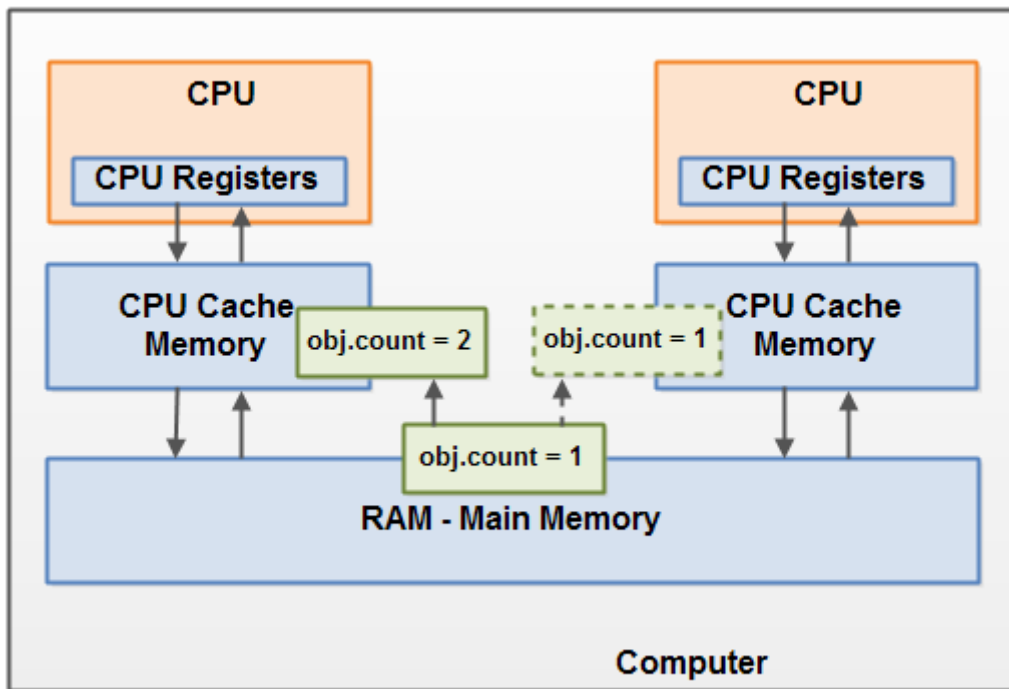
Visibility of Shared Objects

If two or more threads are sharing an object, without the proper use of either volatile declarations or synchronization, updates to the shared object made by one thread may not be visible to other threads.

Imagine that the shared object is initially stored in main memory. A thread running on CPU one then reads the shared object into its CPU cache. There it makes a change to the shared object. As long as the CPU cache has not been flushed back to main memory, the changed version of the shared object is not visible to threads running on other CPUs. This way each thread may end up with its own copy of the shared object, each copy sitting in a different CPU cache.

The following diagram illustrates the sketched situation. One thread running on the left CPU copies the shared object into its CPU cache, and changes its count variable to 2. This

change is not visible to other threads running on the right CPU, because the update to count has not been flushed back to main memory yet.



To solve this problem you can use [Java's volatile keyword](#). The volatile keyword can make sure that a given variable is read directly from main memory, and always written back to main memory when updated.

Race Conditions

If two or more threads share an object, and more than one thread updates variables in that shared object, [race conditions](#) may occur.

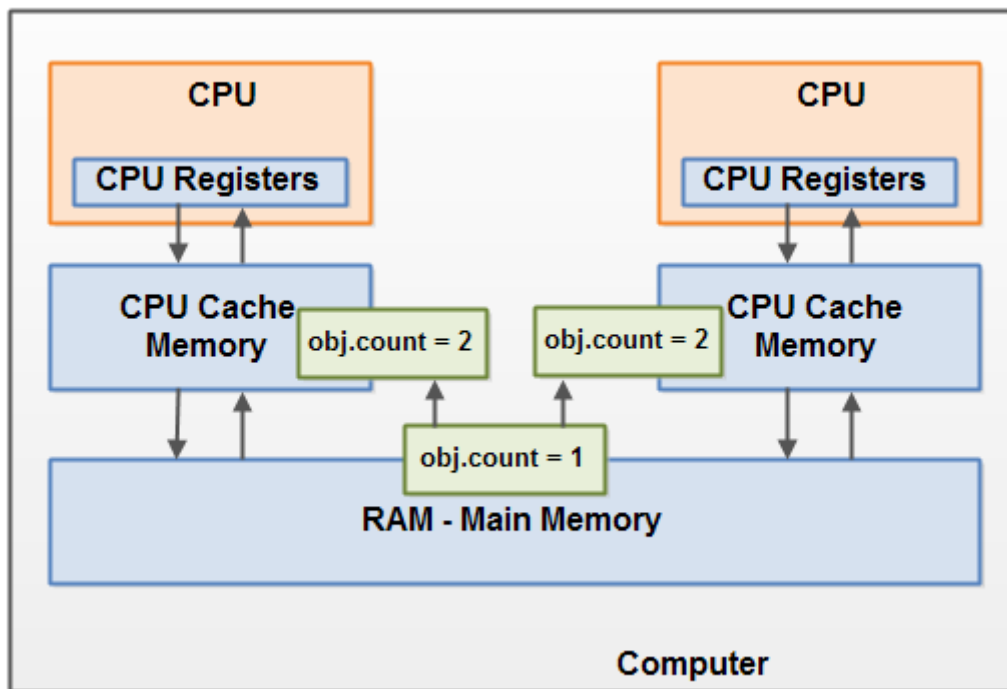
Imagine if thread A reads the variable count of a shared object into its CPU cache. Imagine too, that thread B does the same, but into a different CPU cache. Now thread A adds one to count, and thread B does the same. Now var1 has been incremented two times, once in each CPU cache.

If these increments had been carried out sequentially, the variable count would be incremented twice and had the original value + 2 written back to main memory.

However, the two increments have been carried out concurrently without proper synchronization. Regardless of which of thread A and B that writes its updated version

of count back to main memory, the updated value will only be 1 higher than the original value, despite the two increments.

This diagram illustrates an occurrence of the problem with race conditions as described above:



To solve this problem you can use a [Java synchronized block](#). A synchronized block guarantees that only one thread can enter a given critical section of the code at any given time. Synchronized blocks also guarantee that all variables accessed inside the synchronized block will be read in from main memory, and when the thread exits the synchronized block, all updated variables will be flushed back to main memory again, regardless of whether the variable is declared volatile or not.

13. Dlaczego dziedziczenie wielokrotne jest niedozwolone w Javie?

To byłoby naprawdę skomplikowane. Wyobraź sobie trzy klasy A, B, i C, gdzie C dziedziczy A i B. Teraz założmy, że klasy A i B mają tę samą metodę, a wywołujesz ją z obiektu klasy potomnej... Która z nich zostanie wywołana? A czy B? Mamy tu dwuznaczność.

Jeśli spróbujesz dziedziczyć dwie klasy, Java wygeneruje błąd kompilacji.

W ostatecznym rozrachunku okazuje się, że dziedziczenie wielokrotne **może być ryzykowne** i wcale nie mam na myśli powiązania słowa "ryzykowny" [z wyjątkami](#). Istnieje zagrożenie zetknięcia się z pewnym dylematem rozróżniania metod. Problem "śmiertelnego rombu" (ang. "diamond problem" lub "deadly diamond of death") polega na wystąpieniu sytuacji w

której wywołujemy jedną z metod znajdujących się w obu dziedziczonych klasach w klasie potomnej która, co ważne, w obu miejscach posiada tę samą nazwę i listę parametrów formalnych! W wyniku niemożności rozróżnienia przez kompilator o jaką metodę konkretnie może nam chodzić, dochodzi do problemu uznanego za niemożliwy do rozwiązania.

Pomyślcie sami zanim zadacie sobie pytanie "a może jest jakaś możliwość, tylko jej nie wymyślono?". Dziedziczenie wielokrotne nie patrzy na "sąsiada" czy ma tę samą metodę, parametry i czy nawet ich kolejność jest taka sama! Gdy dochodzi do instrukcji wywołania metody, nie da się zdecydować z której klasy chcemy ją wywołać. Program nie może się "zdecydować" i może się brutalnie zakończyć przerywając proces o kodzie wyjściowym różnym od zera (informacja, że proces nie został zakończony w sposób pożądanym).

Język [Java](#) został tak skonstruowany, aby "łatał" wszystkie niedoskonałości języka C++ oraz konstrukcje uznane za "zbyt trudne". Ale koniec końców, żeby przybierał postać C++ oferując w prezencie "gotowce". To jest cała historia dlaczego dziedziczenie wielokrotne zostało zablokowane. Nie tylko w Javie, ale w większości innych języków programowania.

14. Czy można przeciążyć metodę main()?

Jasne, możemy mieć wiele metod main w programie Java, używając przeciążania metod. Spróbuj!

Yes, you can.

1. The normal main method acts as an entry point for the JVM to start the execution of program.
2. We can overload the main method in Java. But the program doesn't. execute the overloaded main method when we run your program, we need to call the overloaded main method from the actual main method only.

15. Czy możemy zadeklarować konstruktor jako final?

Nie. Konstruktor nie może być zadeklarowany jako final, ponieważ nie może być dziedziczony. Nie ma więc sensu deklarować konstruktorów jako final. Jeśli jednak spróbujesz, kompilator Java zgłosi błąd.

In inheritance whenever you extend a class. The child class inherits all the members of the superclass except the constructors.

In other words, constructors cannot be inherited in Java, therefore, you cannot **override** constructors.

So, writing final before constructors make no sense. Therefore, java does not allow final keyword before a constructor.

If you try, make a constructor final a compile-time error will be generated saying “modifier final not allowed here”.

16. Czy możemy zadeklarować interfejs jako final?

Nie, nie możemy tego zrobić. Interfejs nie może być final, ponieważ interfejs musi być zaimplementowany przez jakąś klasę zgodnie z jego definicją. Dlatego nie ma sensu tworzyć interfejsu final. Jeśli jednak spróbujesz to zrobić, kompilator wygeneruje błąd.

17. Jaka jest różnica pomiędzy static binding i dynamic binding?

Binding is a **mechanism creating link between method call and method actual implementation**. As per the polymorphism concept in Java , object can have many different forms. Object forms can be resolved at compile time and run time

Wiązanie - Binding, które może zostać rozwiązane w czasie kompilacji przez kompilator, nazywane jest statycznym - static lub wczesnym wiązaniem. Wiązanie (Binding) wszystkich metod static, private i final wykonywane jest w trakcie kompilacji.

W dynamic binding, kompilator nie może określić, która metoda ma zostać wywołana. Nadpisywanie metody jest doskonałym przykładem dynamic binding. W przypadku nadpisywania metody obie klasy nadrzędne i podrzędne mają tę samą metodę.

Static Binding

```
class Cat{
    private void talk() {
        System.out.println("Kot miauczy...");
    }

    public static void main(String args[]) {
        Cat cat=new Cat();
        cat.talk();
    }
}
```

Wiazanie dynamiczne

```
class Animal {
    void talk() {
        System.out.println("Zwierzę mówi...");
    }
}
```

```

    }
}

class Cat extends Animal {
    void talk() {
        System.out.println("Kot mówi...");
    }
    public static void main(String args[]) {
        Animal animal=new Cat();
        animal.talk();
    }
}

```

18. Jak utworzyć klasę tylko do odczytu w Javie?

Możesz to zrobić, ustawiając wszystkie pola klasy jako prywatne. Klasa tylko do odczytu ma tylko gettery, które zwracają prywatne właściwości klasy do metody main. Nie możesz modyfikować tych właściwości z powodu braku setterów.

```

public class HockeyPlayer {
    private String team ="Liść klonu";
    public String getTeam() {
        return team;
    }
}

```

19. Jak utworzyć klasę tylko do zapisu w Javie?

Ponownie, należałoby ustawić wszystkie pola klasy jako private. Tym razem twoja klasa tylko do zapisu musi posiadać jedynie metody ustawiające, a nie pobierające. W rezultacie nie można odczytać właściwości klasy.

```

public class HockeyPlayer {
    private String team;
    public void setTeam(String college) {
        this.team = team;
    }
}

```

20. Po każdym bloku try musi następować blok catch, prawda?

Nie. Nie jest to konieczne. Blok try nie musi mieć bloku catch. Mógłby po nim następować albo blok catch, albo blok finally.

```
public class Main {  
    public static void main(String[] args) {  
        try {  
            int variable = 1;  
            System.out.println(variable/0);  
        }  
        finally  
        {  
            System.out.println("Reszta programu...");  
        }  
    }  
}
```

Wydruk:

Exception in thread main java.lang.ArithmeticException:/ by zero Reszta programu...

W przypadku checked exception blok catch jest jednak również wymagany (wymusza to kompilator).

Po bloku try musi być albo catch albo finally nawet w unchecked exception.

21. Jaka jest różnica między słowami kluczowymi throw i throws?

Throws służy do deklarowania wyjątku, więc działa podobnie do bloku try-catch. Słowo kluczowe throw służy do bezpośredniego zgłaszania wyjątku z metody lub dowolnego innego bloku kodu.

Po Throw następuje instancja klasy Exception, a po «throws» następują nazwy klas wyjątków.

Throw jest używany w treści metody do wyrzucania wyjątku. Throws używane jest w sygnaturze metody do deklarowania wyjątków, które mogą wystąpić w twierdzeniach obecnych w metodzie.

Możesz zgłosić jeden wyjątek na raz, ale możesz obsłużyć wiele wyjątków, deklarując je za pomocą słowa kluczowego throws. Możesz zadeklarować wiele wyjątków, np. public void method() throws IOException, SQLException.

22. Is Java 100% Object-oriented?

Java is not 100% Object-oriented because it makes use of eight primitive data types such as boolean, byte, char, int, float, double, long, short which are not objects.

23. Java Wrapper Classes

Wrapper classes provide a way to use primitive data types (int, boolean, etc..) as objects. Sometimes you must use wrapper classes, for example when working with Collection objects, such as ArrayList, where primitive types cannot be used (the list can only store objects). Since you're now working with objects, you can use certain methods to get information about the specific object. For example, the following methods are used to get the value associated with the corresponding wrapper object: intValue(), byteValue(), shortValue(), longValue(), floatValue(), doubleValue(), charValue(), booleanValue().

Another useful method is the `toString()` method, which is used to convert wrapper objects to strings.

In the following example, we convert an `Integer` to a `String`, and use the `length()` method of the `String` class to output the length of the "string":

Example

```
public class Main {  
    public static void main(String[] args) {  
        Integer myInt = 100;  
        String myString = myInt.toString();  
        System.out.println(myString.length());  
    }  
}
```

24. What are constructors in Java?

Konstruktor – specjalna metoda danej klasy, wywoływana podczas tworzenia jej instancji.

In Java, constructor refers to a block of code which is used to initialize an object. It must have the same name as that of the class. Also, it has no return type and it is automatically called when an object is created.

Konstruktor inicjalizuje obiekt podczas jego tworzenia

25. Vector vs. ArrayList in Java

S. No. ArrayList Vector

1. ArrayList is not synchronized. Vector is synchronized.
2. ArrayList increments 50% of the current array size if the number of elements exceeds its capacity.

Vector increments 100% means doubles the array size if the total number of elements exceeds its capacity.

3. ArrayList is not a legacy class. It is introduced in JDK 1.2.
4. Vector is a legacy class.
4. ArrayList is fast because it is non-synchronized.

Vector is slow because it is synchronized, i.e., in a multithreading environment, it holds the other threads in a runnable or non-runnable state until the current thread releases the lock of the object.

5. ArrayList uses the Iterator interface to traverse the elements.

A Vector can use the Iterator interface or Enumeration interface to traverse the elements.

26. When can you use the super keyword?

In Java, the super keyword is a reference variable that refers to an immediate parent class object. When you create a subclass instance, you're also creating an instance of the parent class, which is referenced to by the super reference variable.

The uses of the Java super Keyword are-

1. To refer to an immediate parent class instance variable, use super.
2. The keyword super can be used to call the method of an immediate parent class.
3. Super() can be used to call the constructor of the immediate parent class.

27. What makes a HashSet different from a TreeSet?

HashSet	TreeSet
It is implemented through a hash table.	TreeSet implements SortedSet Interface that uses trees for storing data.
It permits the null object.	It does not allow the null object.
It is faster than TreeSet especially for search, insert, and delete operations.	It is slower than HashSet for these operations.
It does not maintain elements in an ordered way.	The elements are maintained in a sorted order.
It uses equals() method to compare two objects.	It uses compareTo() method for comparing two objects.
It does not permit a heterogenous object.	It permits a heterogenous object.

28. What are the differences between HashMap and Hashtable in Java?

HashMap	Hashtable
It is non synchronized. It cannot be shared between many threads without proper synchronization code.	It is synchronized. It is thread-safe and can be shared with many threads.
It permits one null key and multiple null values.	It does not permit any null key or value.
is a new class introduced in JDK 1.2.	It was present in earlier versions of java as well.
It is faster.	It is slower.
It is traversed through the iterator.	It is traversed through Enumeration and Iterator.
It uses fail fast iterator.	It uses an enumerator which is not fail fast.
It inherits AbstractMap class.	It inherits Dictionary class.

29. Co to znaczy, że metoda jest synchronizowa.

Oznacza to, że w tym samym momencie nie może z niej korzystać kilka wątków na raz co mogłoby prowadzić do błędów.

30. Protected vs Package Private

Metody Package Private mogą być dziedziczone pod warunkiem, że sub-class jest w tym samym pakiecie co super-class.

31. Primitive types vs reference types

Reference types hold references to objects (instances of classes). Unlike primitive types that hold their values in the memory where the variable is allocated, references don't hold the value of the object they refer to.

Note that Java doesn't allow us to discover what the physical memory address is. Rather, we can only use the reference to refer to the object.

32. Constructor of supertype

Every constructor that doesn't call another constructor in the same class has a call to its parent constructor whether it was written explicitly or inserted by the compiler through `super()`.

33. Create constants variables

- The *final* keyword applied to a field means that the field's value can no longer be changed after initialization. In this way, we can define constants in Java.
- Let's add a constant to our *User* class:
- `private static final int YEAR = 2000;`
- Constants must be initialized either when they're declared or in a constructor.

34. Initializers in Java

In Java, an **initializer is a block of code that has no associated name or data type** and is placed outside of any method, constructor, or another block of code.

Java offers two types of initializers, static and instance initializers. Let's see how we can use each of them.

8.1. Instance Initializers

We can use these to initialize instance variables.

To demonstrate, let's provide a value for a user *id* using an instance initializer in our *User* class:

```
{  
    id = 0;  
}
```

8.2. Static Initialization Block

A static initializer or static block – is a block of code which is used to initialize *static* fields. In other words, it's a simple initializer marked with the keyword *static*:

```
private static String forum;  
static {  
    forum = "Java";  
}
```

Statyczny inicjalizator wykona się zarówno tworząc nowy obiekt klasy jak i wywołując jej kontekst statyczny. Zwykły inicjalizator zaś wykona się tylko podczas tworzenia obiektu.

35. Object Life Cycle

Now that we've learned how to declare and initialize objects, let's discover what happens to objects when they're not in use.

Unlike other languages where we have to worry about object destruction, Java takes care of obsolete objects through its garbage collector.

All objects in Java are stored in our program's heap memory. In fact, the heap represents a large pool of unused memory, allocated for our Java application.

On the other hand, the **garbage collector is a Java program that takes care of automatic memory management** by deleting objects that are no longer reachable.

For a Java object to become unreachable, it has to encounter one of the following situations:

- The object no longer has any references pointing to it
- All reference pointing to the object are out of scope

In conclusion, an object is first created from a class, usually using the keyword *new*. Then the object lives its life and provides us with access to its methods and fields.

Finally, when it's no longer needed, the garbage collector destroys it.

36. What is a package in Java? List down various advantages of packages.

Packages in Java, are the collection of related classes and interfaces which are bundled together. By using packages, developers can easily modularize the code and optimize its reuse. Also, the code within the packages can be imported by other classes and reused. Below I have listed down a few of its advantages:

- Packages help in avoiding name clashes
- They provide easier access control on the code
- Packages can also contain hidden classes which are not visible to the outer classes and only used within the package
- Creates a proper hierarchical structure which makes it easier to locate the related classes

37. Differentiate between static and non-static methods in Java.

Static Method	Non-Static Method
1. The <i>static</i> keyword must be used before the method name	1. No need to use the <i>static</i> keyword before the method name
2. It is called using the class (className.methodName)	2. It can be called like any general method
3. They can't access any non-static instance	3. It can access any static method and any

variables or methods

static variable without creating an instance of the class

38. What is a classloader in Java?

The **Java ClassLoader** is a subset of JVM (Java Virtual Machine) that is responsible for loading the class files. Whenever a Java program is executed it is first loaded by the classloader.

39. What is the difference between an Array and an ArrayList?

Array	ArrayList
Cannot contain values of different data types	Can contain values of different data types.
Size must be defined at the time of declaration	Size can be dynamically changed
Need to specify the index in order to add data	No need to specify the index
Arrays are not type parameterized	ArrayLists are type
Arrays can contain primitive data types as well as objects	ArrayLists can contain only objects, no primitive data types are allowed

add() or get() operation : adding an element or retrieving an element from the array or arraylist object has almost same performance , as for ArrayList object these operations run in constant time. Array can be multi dimensional , while ArrayList is always single dimensional.

40. What is Object Oriented Programming?

Object-oriented programming or popularly known as OOPs is a programming model or approach where the programs are organized around objects rather than logic and functions. In other words, OOP mainly focuses on the objects that are required to be manipulated instead of logic. This approach is ideal for the programs large and complex codes and needs to be actively updated or maintained.

Object-Oriented Programming or OOPs is a programming style that is associated with concepts like:

1. *Inheritance*: Inheritance is a process where one class acquires the properties of another.

2. *Encapsulation*: Encapsulation in Java is a mechanism of wrapping up the data and code together as a single unit.
3. *Abstraction*: Abstraction is the methodology of hiding the implementation details from the user and only providing the functionality to the users.
4. *Polymorphism*: Polymorphism is the ability of a variable, function or object to take multiple forms.

41. Czym jest polimorfizm

Polimorfizm jest krótko opisywany jako „jeden interfejs, wiele implementacji”. Polimorfizm jest cechą zdolności do przypisania innego znaczenia lub użycia do czegoś w różnych kontekstach – w szczególności, aby umożliwić jednostce, takiej jak zmienna, funkcja lub obiekt, mieć więcej niż jedną formę. Istnieją dwa rodzaje polimorfizmu:

1. Compile time polymorphism
2. Run time polymorphism

Compile time polymorphism is method overloading whereas Runtime time polymorphism is done using inheritance and interface.

Polymorphic Variables

A variable is called polymorphic if it refers to different values under different conditions.

Object variables (instance variables) represent the behavior of polymorphic variables in Java. It is because object variables of a class can refer to objects of its class as well as objects of its subclasses.

Example: Polymorphic Variables

```
class ProgrammingLanguage {  
    public void display() {  
        System.out.println("I am Programming Language.");  
    }  
}
```

```

    }
}

class Java extends ProgrammingLanguage {
    @Override
    public void display() {
        System.out.println("I am Object-Oriented Programming Language.");
    }
}

class Main {
    public static void main(String[] args) {

        // declare an object variable
        ProgrammingLanguage pl;

        // create object of ProgrammingLanguage
        pl = new ProgrammingLanguage();
        pl.display();

        // create object of Java class
        pl = new Java();
        pl.display();
    }
}

```

Output:

I am Programming Language.

I am Object-Oriented Programming Language.

In the above example, we have created an object variable pl of the ProgrammingLanguage class. Here, pl is a polymorphic variable. This is because,

- In statement `pl = new ProgrammingLanguage()`, pl refer to the object of the ProgrammingLanguage class.
- And, in statement `pl = new Java()`, pl refer to the object of the Java class.

42. What is runtime polymorphism or dynamic method dispatch?

In Java, runtime polymorphism or dynamic method dispatch is a process in which a call to an overridden method is resolved at runtime rather than at compile-time. In this process, an overridden method is called through the reference variable of a superclass. Let's take a look at the example below to understand it better.

```
1    class Car {
2    void run()
3    {
4    System.out.println("car is running");
5    }
6    }
7    class Audi extends Car {
8    void run()
9    {
10   System.out.println("Audi is running safely with 100km");
11   }
12   public static void main(String args[])
13   {
14   Car b= new Audi(); //upcasting
15   b.run();
16   }
17   }
```

43. Czym jest abstrakcja w Javie

Abstrakcja odnosi się do jakości radzenia sobie z ideami, a nie zdarzeniami. Zasadniczo zajmuje się ukrywaniem szczegółów i pokazywaniem podstawowych rzeczy użytkownikowi. Można więc powiedzieć, że abstrakcja w Javie to proces ukrywania przed użytkownikiem szczegółów implementacji i ujawniania mu tylko funkcjonalności. Abstrakcję można osiągnąć na dwa sposoby:

1. Klasy abstrakcyjne (można osiągnąć 0-100% abstrakcji)

2. Interfejsy (można osiągnąć 100% abstrakcji)

44. Czym jest interfejs w Javie

Interfejs w Javie to projekt klasy lub można powiedzieć, że jest to zbiór abstrakcyjnych metod i stałych statycznych. W interfejsie każda metoda jest publiczna i abstrakcyjna, ale nie zawiera żadnego konstruktora. Tak więc interfejs jest w zasadzie grupą powiązanych metod z pustymi ciałami. Przykład:

```
public interface Animal {  
  
    public void eat();  
  
    public void sleep();  
  
    public void run();  
  
}
```

45. Czym jest dziedziczenie w Javie

Inheritance in Java is the concept where the properties of one class can be inherited by the other. It helps to reuse the code and establish a relationship between different classes. Inheritance is performed between two types of classes:

1. Parent class (Super or Base class)
2. Child class (Subclass or Derived class)

A class which inherits the properties is known as Child Class whereas a class whose properties are inherited is known as Parent class.

Dziedziczenie jest jedną z podstaw programowania obiektowego (nie tylko w języku Java). Dzięki dziedziczeniu możemy ograniczyć ilość powielonego kodu poprzez definiowanie atrybutów, konstruktorów, metod w klasach bazowych.

46. Czym jest Overloading i Overriding

Method Overloading :

- In Method Overloading, Methods of the same class shares the same name but each method must have a different number of parameters or parameters having different types and order.
- Method Overloading is to “add” or “extend” more to the method’s behavior.
- It is a compile-time polymorphism.
- The methods must have a different signature.
- It may or may not need inheritance in Method Overloading.

Let’s take a look at the example below to understand it better.

```
1      class Adder {  
2          Static int add(int a, int b)  
3      {  
4          return a+b;  
5      }  
6      Static double add( double a, double b)  
7      {  
8          return a+b;  
9      }  
10     public static void main(String args[])  
11     {  
12         System.out.println(Adder.add(11,11));  
13         System.out.println(Adder.add(12.3,12.6));  
14     }}
```

```
public class Owoc {

    void pom(String a, Integer b) {

    }

    void pom(Integer b, String a) {

    }

}
```

Takie coś jest jak najbardziej dozwolone (choć niezalecane)

Method Overriding:

- In Method Overriding, the subclass has the same method with the same name and exactly the same number and type of parameters and same return type as a superclass.
- Method Overriding is to “Change” existing behavior of the method.
- It is a run time polymorphism.
- The methods must have the same signature.
- It always requires inheritance in Method Overriding.

Let’s take a look at the example below to understand it better.

```
1      class Car {
2          void run(){
3              System.out.println("car is running");
4          }
5      Class Audi extends Car{
6          void run()
7          {
8              System.out.println("Audi is running safely with 100km");
9          }
}
```

```
10    public static void main( String args[])
11    {
12        Car b=new Audi();
13        b.run();
14    }
15    }
```

47. Dziedziczenie wielokrotne

If a child class inherits the property from multiple classes is known as multiple inheritance. Java does not allow to extend multiple classes.

The problem with multiple inheritance is that if multiple parent classes have the same method name, then at runtime it becomes difficult for the compiler to decide which method to execute from the child class.

Therefore, Java doesn't support multiple inheritance. The problem is commonly referred to as Diamond Problem.

48. Czym jest enkapsulacja

Enkapsulacja to mechanizm, w którym łączysz dane (zmienne) i kod (metody) w jedną całość. Tutaj dane są ukryte przed światem zewnętrznym i można uzyskać do nich dostęp tylko za pomocą obecnych metod klas. **Pomaga to chronić dane przed niepotrzebną modyfikacją.** Enkapsulację w Javie możemy osiągnąć poprzez:

- Declaring the variables of a class as private.
- Providing public setter and getter methods to modify and view the values of the variables.

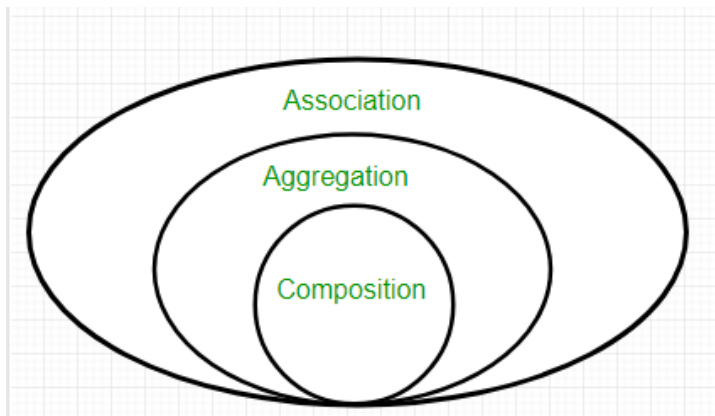
Enkapsulacja == Hermetyzacja

49. Constructor in an interface

- **An Interface in Java doesn't have a constructor because all data members in interfaces are public static final by default, they are constants (assign the values at the time of declaration).**
- There are no data members in an interface to initialize them through the constructor.
- In order to call a method, we need an object, since the methods in the interface don't have a body there is no need for calling the methods in an interface.
- **Since we cannot call the methods in the interface, there is no need of creating an object for an interface and there is no need of having a constructor in it.**

50. Association, Composition and Aggregation in Java

Asocjacja to relacja między dwiema oddzielnymi klasami, która ustanawia się poprzez ich Obiekty. Powiązanie może być typu jeden-do-jednego, jeden-do-wielu, wiele-do-jednego, wiele-do-wielu. W programowaniu obiektowym obiekt komunikuje się z innym obiektem, aby korzystać z funkcji i usług dostarczanych przez ten obiekt. Kompozycja i agregacja to dwie formy asocjacji.



Agregacja

Jest to szczególna forma asocjacji, w której:

- Reprezentuje związek **Has-A**.
- Jest to asocjacja jednokierunkowa, czyli relacja w jedną stronę. Na przykład wydział może mieć studentów, ale odwrotnie nie jest to możliwe, a zatem ma charakter jednokierunkowy.

- W agregacji oba wpisy mogą przetrwać pojedynczo, co oznacza, że zakończenie jednej jednostki nie wpłynie na drugą.

```
// Java program to illustrate Concept of Aggregation

// Importing required classes
import java.io.*;
import java.util.*;

// Class 1
// Student class
class Student {

    // Attributes of student
    String name;
    int id;
    String dept;

    // Constructor of student class
    Student(String name, int id, String dept)
    {

        // This keyword refers to current instance itself
        this.name = name;
        this.id = id;
        this.dept = dept;
    }
}

// Class 2
// Department class contains list of student objects
// It is associated with student class through its Objects
class Department {
    // Attributes of Department class
    String name;
    private List<Student> students;
    Department(String name, List<Student> students)
    {
        // this keyword refers to current instance itself
        this.name = name;
        this.students = students;
    }

    // Method of Department class
    public List<Student> getStudents()
    {
        // Returning list of user defined type
        // Student type
        return students;
    }
}

// Class 3
// Institute class contains list of Department
// Objects. It is associated with Department
// class through its Objects
```

```

class Institute {

    // Attributes of Institute
    String instituteName;
    private List<Department> departments;

    // Constructor of institute class
    Institute(String instituteName, List<Department> departments)
    {
        // This keyword refers to current instance itself
        this.instituteName = instituteName;
        this.departments = departments;
    }

    // Method of Institute class
    // Counting total students of all departments
    // in a given institute
    public int getTotalStudentsInInstitute()
    {
        int noOfStudents = 0;
        List<Student> students;

        for (Department dept : departments) {
            students = dept.getStudents();

            for (Student s : students) {
                noOfStudents++;
            }
        }

        return noOfStudents;
    }
}

// Class 4
// main class
class GFG {

    // main driver method
    public static void main(String[] args)
    {
        // Creating object of Student class inside main()
        Student s1 = new Student("Mia", 1, "CSE");
        Student s2 = new Student("Priya", 2, "CSE");
        Student s3 = new Student("John", 1, "EE");
        Student s4 = new Student("Rahul", 2, "EE");

        // Creating a List of CSE Students
        List<Student> cse_students = new ArrayList<Student>();

        // Adding CSE students
        cse_students.add(s1);
        cse_students.add(s2);

        // Creating a List of EE Students
        List<Student> ee_students
            = new ArrayList<Student>();
    }
}

```

```

// Adding EE students
ee_students.add(s3);
ee_students.add(s4);

// Creating objects of EE and CSE class inside
// main()
Department CSE = new Department("CSE", cse_students);
Department EE = new Department("EE", ee_students);

List<Department> departments = new ArrayList<Department>();
departments.add(CSE);
departments.add(EE);

// Lastly creating an instance of Institute
Institute institute = new Institute("BITS", departments);

// Display message for better readability
System.out.print("Total students in institute: ");

// Calling method to get total number of students
// in institute and printing on console
System.out.print(institute.getTotalStudentsInInstitute());
}
}

```

Output Explanation: In this example, there is an Institute which has no. of departments like CSE, EE. Every department has no. of students. So, we make an Institute class that has a reference to Object or no. of Objects (i.e. List of Objects) of the Department class. That means Institute class is associated with Department class through its Object(s). And Department class has also a reference to Object or Objects (i.e. List of Objects) of the Student class means it is associated with the Student class through its Object(s).

It represents a **Has-A** relationship. In the above example: Student **Has-A** name. Student **Has-A** ID. Student **Has-A** Dept. Department **Has-A** Students as depicted from the below media.

When do we use Aggregation ??

Code reuse is best achieved by aggregation.

Kompozycja

Kompozycja jest ograniczoną formą agregacji, w której dwie jednostki są od siebie wysoce zależne.

- Reprezentuje relację **part-of**.
- W składzie oba podmioty są od siebie zależne.

- Gdy istnieje kompozycja między dwiema jednostkami, złożony obiekt nie może istnieć bez drugiej jednostki.

```
// Java program to illustrate
// the concept of Composition

// Importing required classes
import java.io.*;
import java.util.*;

// Class 1
// Book
class Book {

    // Attributes of book
    public String title;
    public String author;

    // Constructor of Book class
    Book(String title, String author)
    {

        // This keyword refers to current instance itself
        this.title = title;
        this.author = author;
    }
}

// Class 2
class Library {

    // Reference to refer to list of books
    private final List<Book> books;

    // Library class contains list of books
    Library(List<Book> books)
    {

        // Referring to same book as
        // this keyword refers to same instance itself
        this.books = books;
    }

    // Method
    // To get total number of books in library
    public List<Book> getTotalBooksInLibrary()
    {

        return books;
    }
}

// Class 3
```

```

// Main class
class GFG {

    // Main driver method
    public static void main(String[] args)
    {

        // Creating objects of Book class inside main()
        // method Custom inputs
        Book b1
            = new Book("EffectiveJ Java", "Joshua Bloch");
        Book b2
            = new Book("Thinking in Java", "Bruce Eckel");
        Book b3 = new Book("Java: The Complete Reference",
                           "Herbert Schildt");

        // Creating the list which contains number of books
        List<Book> books = new ArrayList<Book>();

        // Adding books
        // using add() method
        books.add(b1);
        books.add(b2);
        books.add(b3);

        Library library = new Library(books);

        // Calling method to get total books in library
        // and storing it in list of user0defined type -
        // Books
        List<Book> bks = library.getTotalBooksInLibrary();

        // Iterating over books using for each loop
        for (Book bk : bks) {

            // Printing the title and author name of book on
            // console
            System.out.println("Title : " + bk.title
                               + " and "
                               + " Author : " + bk.author);
        }
    }
}

```

Output explanation: In the above example, a library can have no. of **books** on the same or different subjects. So, If Library gets destroyed then All books within that particular library will be destroyed. i.e. books can not exist without libraries. That's why it is composition. Book is **Part-of** Library.

Aggregation vs Composition

1. Dependency: Aggregation implies a relationship where the child **can exist independently** of the parent. For example, Bank and Employee, delete the Bank and the Employee still exist. whereas Composition implies a relationship where the child **cannot exist independent** of the parent. Example: Human and heart, heart don't exist separate to a Human

2. Type of Relationship: Aggregation relation is “**has-a**” and composition is “**part-of**” relation.

3. Type of association: Composition is a **strong** Association whereas Aggregation is a **weak** Association.

```
// Java Program to Illustrate Difference between
// Aggregation and Composition

// Importing I/O classes
import java.io.*;

// Class 1
// Engine class which will
// be used by car. so 'Car'
// class will have a field
// of Engine type.
class Engine {

    // Method to starting an engine
    public void work()
    {

        // Print statement whenever this method is called
        System.out.println(
            "Engine of car has been started ");
    }
}

// Class 2
// Engine class
final class Car {

    // For a car to move,
    // it needs to have an engine.

    // Composition
    private final Engine engine;

    // Note: Uncommented part refers to Aggregation
    // private Engine engine;

    // Constructor of this class
    Car(Engine engine)
    {

        // This keywords refers to same instance
```

```

        this.engine = engine;
    }

    // Method
    // Car start moving by starting engine
    public void move()
    {

        // if(engine != null)
        {
            // Calling method for working of engine
            engine.work();

            // Print statement
            System.out.println("Car is moving ");
        }
    }
}

// Class 3
// Main class
class GFG {

    // Main driver method
    public static void main(String[] args)
    {

        // Making an engine by creating
        // an instance of Engine class.
        Engine engine = new Engine();

        // Making a car with engine so we are
        // passing a engine instance as an argument
        // while creating instance of Car
        Car car = new Car(engine);

        // Making car to move by calling
        // move() method inside main()
        car.move();
    }
}

```

In case of aggregation, the Car also performs its functions through an Engine. but the Engine is not always an internal part of the Car. An engine can be swapped out or even can be removed from the car. That's why we make The Engine type field non-final.

51. What is a marker interface?

A Marker interface can be defined as the interface having no data member and member functions. In simpler terms, an empty interface is called the Marker interface. The most common examples of Marker interface in Java are Serializable, Cloneable etc. The marker interface can be declared as follows.

```
1 public interface Serializable{  
2 }
```

52. What is the difference between Error and Exception?

An error is an irrecoverable condition occurring at runtime. Such as OutOfMemory error. These JVM errors you cannot repair them at runtime. Though error can be caught in the catch block but the execution of application will come to a halt and is not recoverable.

While exceptions are conditions that occur because of bad input or human error etc. e.g. FileNotFoundException will be thrown if the specified file does not exist. Or a NullPointerException will take place if you try using a null reference. In most of the cases it is possible to recover from an exception (probably by giving the user feedback for entering proper values etc.

Error along with RuntimeException & their subclasses are unchecked exceptions. All other Exception classes are checked exceptions.

Checked exceptions are generally those from which a program can recover & it might be a good idea to recover from such exceptions programmatically. Examples include FileNotFoundException, ParseException, etc. A programmer is expected to check for these exceptions by using the try-catch block or throw it back to the caller

On the other hand we have unchecked exceptions. These are those exceptions that might not happen if everything is in order, but they do occur. Examples include ArrayIndexOutOfBoundsException, ClassCastException, etc. Many applications will use try-catch or throws clause for RuntimeExceptions & their subclasses but from the language perspective it is not required to do so. Do note that recovery from a RuntimeException is generally possible but the guys who designed the class/exception deemed it unnecessary for the end programmer to check for such exceptions.

Errors are also unchecked exception & the programmer is not required to do anything with these. In fact it is a bad idea to use a try-catch clause for Errors. Most often, recovery from an Error is not possible & the program should be allowed to terminate. Examples include OutOfMemoryError, StackOverflowError, etc.

Do note that although Errors are unchecked exceptions, we shouldn't try to deal with them, but it is ok to deal with RuntimeExceptions(also unchecked exceptions) in code. Checked exceptions should be handled by the code.

53. How can you handle Java exceptions?

There are five keywords used to handle exceptions in Java:

1. try
2. catch
3. finally
4. throw
5. throws

54. What are the differences between Checked Exception and Unchecked Exception?

Checked Exception

- The classes that extend Throwable class except RuntimeException and Error are known as checked exceptions.
- Checked exceptions are checked at compile-time.
- Example: IOException, SQLException etc.

Unchecked Exception

- The classes that extend RuntimeException are known as unchecked exceptions.
- Unchecked exceptions are not checked at compile-time.
- Example: ArithmeticException, NullPointerException etc.

55. What are the different ways of thread usage?

There are two ways to create a thread:

- Extending Thread class

This creates a thread by creating an instance of a new class that extends the Thread class. The extending class must override the run() function, which is the thread's entry point.

- Implementing Runnable interface

This is the easiest way to create a thread, by creating a class that implements the runnable interface. After implementing the runnable interface, the class must implement the public void run() method ()

The run() method creates a parallel thread in your programme. When run() returns, the thread will come to an end.

Within the run() method, you must specify the thread's code.

Like any other method, the run() method can call other methods, use other classes, and define variables.

56. Will the finally block get executed when the return statement is written at the end of try block and catch block as shown below?

The finally block always gets executed even when the return statement is written at the end of the try block and the catch block. It always executes, whether there is an exception or not. There are only a few situations in which the finally block does not execute, such as VM crash, power failure, software crash, etc. If you don't want to execute the finally block, you need to call the System.exit() method explicitly in the finally block.

57. Can you explain the Java thread lifecycle?

The java thread lifecycle has the following states-

New

When a thread is created, and before the program starts the thread, it is in the new state. It is also referred to as a born thread.

Runnable

Waiting for its turn to be picked for execution by the thread scheduler based on thread priorities.

Running

The processor is actively executing the thread code. It runs until it becomes blocked, or voluntarily gives up its turn.

Waiting

A thread is in a blocked state while it waits for some external processing such as file I/O to finish.

Sleeping

Java threads are forcibly put to sleep (suspended) with Thread.sleep. they can resume using Thread.resume method.

Blocked on I/O

Will move to runnable after I/O condition like reading bytes of data etc changes.

Blocked on synchronization

Will move to Runnable when a lock is acquired.

Dead

The thread is finished working.

58. Finalize keyword

Finalize is used to perform clean up processing just before the object is garbage collected.

The finalize method is called when an object is about to get garbage collected. That can be at any time after it has become eligible for garbage collection.

Note that it's entirely possible that an object never gets garbage collected (and thus finalize is never called). This can happen when the object never becomes eligible for gc (because it's reachable through the entire lifetime of the JVM) or when no garbage collection actually runs between the time the object become eligible and the time the JVM stops running (this often occurs with simple test programs).

There are ways to tell the JVM to run finalize on objects that it wasn't called on yet, but using them isn't a good idea either (the guarantees of that method aren't very strong either).

If you rely on finalize for the correct operation of your application, then you're doing something wrong. finalize should **only** be used for cleanup of (usually non-Java) resources. And that's *exactly* because the JVM doesn't guarantee that finalize is ever called on any object.

59. What are the differences between throw and throws?

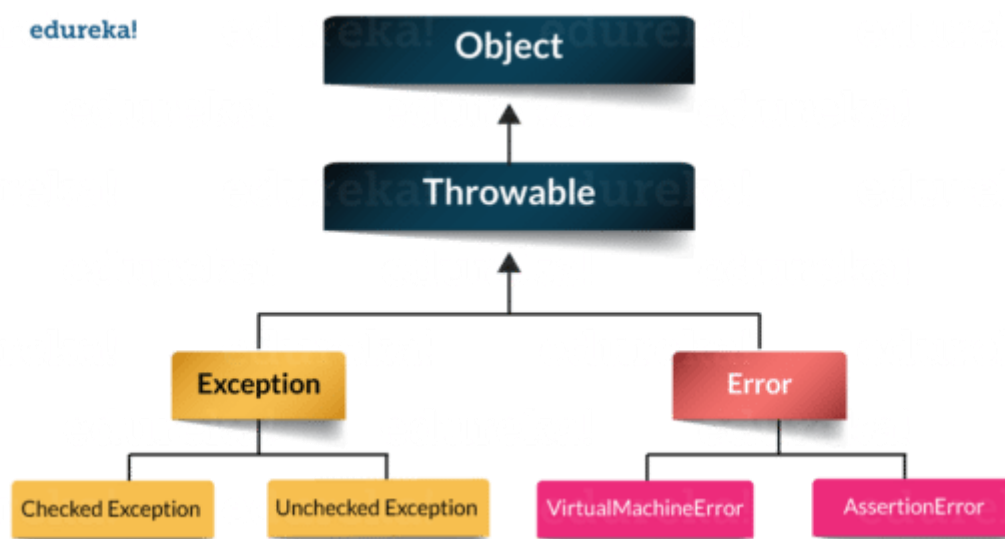
throw keyword	throws keyword
Throw is used to explicitly throw an exception.	Throws is used to declare an exception.
Checked exceptions can not be propagated with throw only.	Checked exception can be propagated with throws.
Throw is followed by an instance.	Throws is followed by class.
Throw is used within the method.	Throws is used with the method signature.

You cannot throw multiple exception	You can declare multiple exception e.g. <code>public void method()throws IOException,SQLException.</code>
-------------------------------------	---

60. What is exception hierarchy in java?

The hierarchy is as follows:

Throwable is a parent class of all Exception classes. There are two types of Exceptions: Checked exceptions and UncheckedExceptions or RunTimeExceptions. Both type of exceptions extends Exception class whereas errors are further classified into Virtual Machine error and Assertion error.



61. How to create a custom Exception?

To create you own exception extend the Exception class or any of its subclasses.

- `class New1Exception extends Exception { }` // this will create Checked Exception
- `class NewException extends IOException { }` // this will create Checked exception
- `class NewException extends NullPonterExcpetion { }` // this will create UnChecked exception

62. What are the important methods of Java Exception Class?

Exception and all of its subclasses doesn't provide any specific methods and all of the methods are defined in the base class Throwable.

1. **String getMessage()** – This method returns the message String of Throwable and the message can be provided while creating the exception through its constructor.
2. **String getLocalizedMessage()** – This method is provided so that subclasses can override it to provide locale specific message to the calling program. Throwable class implementation of this method simply use getMessage() method to return the exception message.
3. **Synchronized Throwable getCause()** – This method returns the cause of the exception or null id if the cause is unknown.
4. **String toString()** – This method returns the information about Throwable in String format, the returned String contains the name of Throwable class and localized message.
5. **void printStackTrace()** – This method prints the stack trace information to the standard error stream, this method is overloaded and we can pass PrintStream or PrintWriter as an argument to write the stack trace information to the file or stream.

63. What are the differences between processes and threads?

	Process	Thread
Definition	An executing instance of a program is called a process.	A thread is a subset of the process.
Communication	Processes must use inter-process communication to communicate with sibling processes.	Threads can directly communicate with other threads of its process.
Control	Processes can only exercise control over child processes.	Threads can exercise considerable control over threads of the same process.
Changes	Any change in the parent process does not affect child processes.	Any change in the main thread may affect the behavior of the other threads of the process.
Memory	Run in separate memory spaces.	Run in shared memory spaces.
Controlled by	Process is controlled by the operating system.	Threads are controlled by programmer in a program.

Dependence	Processes are independent.	Threads are dependent.
-------------------	----------------------------	------------------------

A process is an active program i.e. a program that is under execution. It is more than the program code as it includes the program counter, process stack, registers, program code etc. Compared to this, the program code is only the text section.

A thread is a lightweight process that can be managed independently by a scheduler. It improves the application performance using parallelism. A thread shares information like data segment, code segment, files etc. with its peer threads while it contains its own registers, stack, counter etc.

The major differences between a process and a thread are given as follows –

Comparison Basis	Process	Thread
Definition	A process is a program under execution i.e an active program.	A thread is a lightweight process that can be managed independently by a scheduler.
Context switching time	Processes require more time for context switching as they are more heavy.	Threads require less time for context switching as they are lighter than processes.
Memory Sharing	Processes are totally independent and don't share memory.	A thread may share some memory with its peer threads.
Communication	Communication between processes requires more time than between threads.	Communication between threads requires less time than between processes .
Blocked	If a process gets blocked, remaining processes can continue execution.	If a user level thread gets blocked, all of its peer threads also get blocked.
Resource Consumption	Processes require more resources than threads.	Threads generally need less resources than processes.
Dependency	Individual processes are independent of each other.	Threads are parts of a process and so are dependent.

Comparison Basis	Process	Thread
Data and Code sharing	Processes have independent data and code segments.	A thread shares the data segment, code segment, files etc. with its peer threads.
Treatment by OS	All the different processes are treated separately by the operating system.	All user level peer threads are treated as a single task by the operating system.
Time for creation	Processes require more time for creation.	Threads require less time for creation.
Time for termination	Processes require more time for termination.	Threads require less time for termination.

64. What is a finally block? Is there a case when finally will not execute?

Finally block is a block which always executes a set of statements. **It is always associated with a try block regardless of any exception that occurs or not.**

Yes, finally will not be executed if the program exits either by calling `System.exit()` or by causing a fatal error that causes the process to abort.

An infinite loop would also prevent a finally being called.

65. What is synchronization?

Synchronizacja odnosi się do wielowątkowości. Zsynchronizowany blok kodu może być jednocześnie wykonywany tylko przez jeden wątek. Ponieważ Java obsługuje wykonywanie wielu wątków, dwa lub więcej wątków może uzyskać dostęp do tych samych pól lub obiektów. Synchronizacja to proces, który utrzymuje synchronizację wszystkich współbieżnych wątków. Synchronizacja pozwala uniknąć błędów spójności pamięci spowodowanych niespójnym widokiem pamięci współdzielonej. Gdy metoda jest

zadeklarowana jako zsynchronizowana, wątek przechowuje monitor dla obiektu tej metody. Jeśli inny wątek wykonuje zsynchronizowaną metodę, wątek jest blokowany, dopóki ten wątek nie zwolni monitora.

66. Can we write multiple catch blocks under single try block?

Yes we can have multiple catch blocks under single try block but the approach should be from specific to general. Let's understand this with a programmatic example.

```
public class Example {
    public static void main(String args[]) {
        try {
            int a[] = new int[10];
            a[10] = 10/0;
        }
        catch(ArithmeticException e)
        {
            System.out.println("Arithmetic exception in first catch block");
        }
        catch(ArrayIndexOutOfBoundsException e)
        {
            System.out.println("Array index out of bounds in second catch block");
        }
        catch(Exception e)
        {
            System.out.println("Any exception in third catch block");
        }
    }
}
```

67. What is a Thread?

Wątek to najmniejszy fragment instrukcji programowania, które mogą być wykonywane niezależnie przez program. W Javie wszystkie programy będą miały co najmniej jeden wątek, który jest znany jako wątek główny. Ten wątek główny jest tworzony przez maszynę JVM, gdy program rozpoczyna wykonywanie. Główny wątek służy do wywołania funkcji main() programu.

68. What is concrete class?

A concrete class is any normal class in a Java program. This class will not have any abstract methods. All the methods in the concrete class are completely implemented. A concrete class can inherit from another class, even an abstract class or implement an interface.

69. Finalize()

finalize() to metoda klasy Object, która jest wywoływana dokładnie raz, zanim obiekt zostanie usunięty z pamięci przez garbage collector (może się jednak zdarzyć, że nie zostanie ona wywołana, ponieważ obiekt nie jest usuwany); ogólnie nie jest najlepszą praktyką polegać na tej metodzie, lepszym rozwiązaniem jest samodzielne zarządzanie cyklem życia obiektu.

Note that it's entirely possible that an object never gets garbage collected (and thus finalize is never called). This can happen when the object never becomes eligible for gc (because it's reachable through the entire lifetime of the JVM) or when no garbage collection actually runs between the time the object become eligible and the time the JVM stops running (this often occurs with simple test programs).

70. What is the Difference Between ArrayList and HashSet in Java?

The following ones are the main differences between ArrayList and HashSet:

Implementation

ArrayList: Implements List interface.

HashSet: Implements Set interface.

Duplicates

ArrayList: Allows duplicates values.

HashSet: Doesn't allow duplicates values.

Constructor

ArrayList: Have three constructor which are ArrayList (), ArrayList (int capacity), ArrayList (int Collection c).

HashSet: Have four constructor which are HashSet (), HashSet (int capacity), HashSet (Collection c), and HashSet (int capacity, float loadFactor).

Order

ArrayList: Maintains the order of the object in which they are inserted.

HashSet: Doesn't maintain any order.

Null Object

ArrayList: It's possible to add any number of null value.

HashSet: Allow one null value.

71. Java Type Casting

Type casting is when you assign a value of one primitive data type to another type.

In Java, there are two types of casting:

- **Widening Casting** (automatically) - converting a smaller type to a larger type size
`byte -> short -> char -> int -> long -> float -> double`
- **Narrowing Casting** (manually) - converting a larger type to a smaller size type
`double -> float -> long -> int -> char -> short -> byte`

Widening Casting

Widening casting is done automatically when passing a smaller size type to a larger size type:

Example

```
public class Main {  
    public static void main(String[] args) {  
        int myInt = 9;  
        double myDouble = myInt; // Automatic casting: int to double  
  
        System.out.println(myInt);    // Outputs 9  
        System.out.println(myDouble); // Outputs 9.0  
    }  
}
```

Narrowing Casting

Narrowing casting must be done manually by placing the type in parentheses in front of the value:

Example

```
public class Main {  
    public static void main(String[] args) {  
        double myDouble = 9.78d;  
        int myInt = (int) myDouble; // Manual casting: double to int  
  
        System.out.println(myDouble); // Outputs 9.78  
        System.out.println(myInt);    // Outputs 9  
    }  
}
```

72. Java reference type casting

Although primitive conversions and reference variable casting may look similar, they're quite **different concepts**.

In both cases, we're "turning" one type into another. But, in a simplified way, a primitive variable contains its value, and conversion of a primitive variable means irreversible changes in its value:

```
double myDouble = 1.1;  
int myInt = (int) myDouble;
```

```
assertNotEquals(myDouble, myInt);
```

After the conversion in the above example, *myInt* variable is 1, and we can't restore the previous value 1.1 from it.

Reference variables are different; the reference variable only refers to an object but doesn't contain the object itself.

And casting a reference variable doesn't touch the object it refers to but only labels this object in another way, expanding or narrowing opportunities to work with it. **Upcasting narrows the list of methods and properties available to this object, and downcasting can extend it.**

A reference is like a remote control to an object. The remote control has more or fewer buttons depending on its type, and the object itself is

stored in a heap. When we do casting, we change the type of the remote control but don't change the object itself.

Upcasting

Casting from a subclass to a superclass is called upcasting. Typically, the upcasting is implicitly performed by the compiler.

Upcasting is closely related to inheritance — another core concept in Java. It's common to use reference variables to refer to a more specific type. And every time we do this, implicit upcasting takes place.

To demonstrate upcasting, let's define an *Animal* class:

```
public class Animal {  
  
    public void eat() {  
        // ...  
    }  
}
```

Now let's extend *Animal*:

```
public class Cat extends Animal {  
  
    public void eat() {  
        // ...  
    }  
  
    public void meow() {  
        // ...  
    }  
}
```

Now we can create an object of *Cat* class and assign it to the reference variable of type *Cat*:

```
Cat cat = new Cat();
```

And we can also assign it to the reference variable of type *Animal*:

```
Animal animal = cat;
```

In the above assignment, implicit upcasting takes place.

We could do it explicitly:

```
animal = (Animal) cat;
```

But there is no need to do explicit cast up the inheritance tree. The compiler knows that *cat* is an *Animal* and doesn't display any errors.

Note that reference can refer to any subtype of the declared type.

Using upcasting, we've restricted the number of methods available to *Cat* instance but haven't changed the instance itself. Now we can't do anything that is specific to *Cat* — we can't invoke *meow()* on the *animal* variable.

Although *Cat* object remains *Cat* object, calling *meow()* would cause the compiler error:

```
// animal.meow(); The method meow() is undefined for the type Animal
```

To invoke *meow()* we need to downcast *animal*, and we'll do this later.

But now we'll describe what gives us the upcasting. Thanks to upcasting, we can take advantage of polymorphism.

Polymorphism

Let's define another subclass of *Animal*, a *Dog* class:

```
public class Dog extends Animal {  
  
    public void eat() {  
        // ...  
    }  
}
```

Now we can define the *feed()* method, which treats all cats and dogs like *animals*:

```
public class AnimalFeeder {  
  
    public void feed(List<Animal> animals) {  
        animals.forEach(animal -> {  
            animal.eat();  
        });  
    }  
}
```

We don't want *AnimalFeeder* to care about which *animal* is on the list — a *Cat* or a *Dog*. In the *feed()* method they are all *animals*.

Implicit upcasting occurs when we add objects of a specific type to the *animals* list:

```
List<Animal> animals = new ArrayList<>();  
animals.add(new Cat());  
animals.add(new Dog());  
new AnimalFeeder().feed(animals);
```

We add cats and dogs, and they are upcast to *Animal* type implicitly. Each *Cat* is an *Animal* and each *Dog* is an *Animal*. They're polymorphic.

By the way, all Java objects are polymorphic because each object is an *Object* at least. We can assign an instance of *Animal* to the reference variable of *Object* type and the compiler won't complain:

```
Object object = new Animal();
```

That's why all Java objects we create already have *Object*-specific methods, for example *toString()*.

Upcasting to an interface is also common.

We can create *Mew* interface and make *Cat* implement it:

```

public interface Mew {
    public void meow();
}

public class Cat extends Animal implements Mew {

    public void eat() {
        // ...
    }

    public void meow() {
        // ...
    }
}

```

Now any *Cat* object can also be upcast to *Mew*:

```
Mew mew = new Cat();
```

Cat is a *Mew*; upcasting is legal and done implicitly.

Therefore, *Cat* is a *Mew*, *Animal*, *Object* and *Cat*. It can be assigned to reference variables of all four types in our example.

Overriding

In the example above, the *eat()* method is overridden. This means that although *eat()* is called on the variable of the *Animal* type, the work is done by methods invoked on real objects — cats and dogs:

```

public void feed(List<Animal> animals) {
    animals.forEach(animal -> {
        animal.eat();
    });
}

```

If we add some logging to our classes, we'll see that *Cat* and *Dog* methods are called:

```

web - 2018-02-15 22:48:49,354 [main] INFO com.baeldung.casting.Cat - cat is eating
web - 2018-02-15 22:48:49,363 [main] INFO com.baeldung.casting.Dog - dog is eating

```

To sum up:

- A reference variable can refer to an object if the object is of the same type as a variable or if it is a subtype.
- Upcasting happens implicitly.
- All Java objects are polymorphic and can be treated as objects of supertype due to upcasting.

Downcasting

What if we want to use the variable of type *Animal* to invoke a method available only to *Cat* class? Here comes the downcasting. **It's the casting from a superclass to a subclass.**

Let's look at an example:

```
Animal animal = new Cat();
```

We know that *animal* variable refers to the instance of *Cat*. And we want to invoke *Cat*'s *meow()* method on the *animal*. But the compiler complains that *meow()* method doesn't exist for the type *Animal*.

To call *meow()* we should downcast *animal* to *Cat*:

```
((Cat) animal).meow();
```

The inner parentheses and the type they contain are sometimes called the cast operator. Note that external parentheses are also needed to compile the code.

Let's rewrite the previous *AnimalFeeder* example with *meow()* method:

```
public class AnimalFeeder {  
  
    public void feed(List<Animal> animals) {  
        animals.forEach(animal -> {  
            animal.eat();  
            if (animal instanceof Cat) {  
                ((Cat) animal).meow();  
            }  
        });  
    }  
}
```

Now we gain access to all methods available to *Cat* class. Look at the log to make sure that *meow()* is actually called:

```
web - 2018-02-16 18:13:45,445 [main] INFO com.baeldung.casting.Cat - cat is eating  
web - 2018-02-16 18:13:45,454 [main] INFO com.baeldung.casting.Cat - meow  
web - 2018-02-16 18:13:45,455 [main] INFO com.baeldung.casting.Dog - dog is eating
```

Note that in the above example we're trying to downcast only those objects that are really instances of *Cat*. To do this, we use the operator *instanceof*.

***instanceof* Operator**

We often use *instanceof* operator before downcasting to check if the object belongs to the specific type:

```
if (animal instanceof Cat) {  
    ((Cat) animal).meow();  
}
```

ClassCastException

If we hadn't checked the type with the *instanceof* operator, the compiler wouldn't have complained. But at runtime, there would be an exception.

To demonstrate this, let's remove the *instanceof* operator from the above code:

```
public void uncheckedFeed(List<Animal> animals) {  
    animals.forEach(animal -> {  
        animal.eat();  
        ((Cat) animal).meow();  
    });  
}
```

This code compiles without issues. But if we try to run it, we'll see an exception:

java.lang.ClassCastException: com.baeldung.casting.Dog cannot be cast to com.baeldung.casting.Cat

This means that we are trying to convert an object that is an instance of *Dog* into a *Cat* instance.

ClassCastException is always thrown at runtime if the type we downcast to doesn't match the type of the real object.

Note that if we try to downcast to an unrelated type, the compiler won't allow this:

```
Animal animal;  
String s = (String) animal;
```

The compiler says "Cannot cast from Animal to String."

For the code to compile, both types should be in the same inheritance tree.

Let's sum up:

- Downcasting is necessary to gain access to members specific to subclass.
- Downcasting is done using cast operator.
- To downcast an object safely, we need *instanceof* operator.
- If the real object doesn't match the type we downcast to, then *ClassCastException* will be thrown at runtime.

***cast()* Method**

There's another way to cast objects using the methods of *Class*:

```
public void whenDowncastToCatWithCastMethod_thenMeowIsCalled() {
```

```

Animal animal = new Cat();
if (Cat.class.isInstance(animal)) {
    Cat cat = Cat.class.cast(animal);
    cat.meow();
}
}

```

In the above example, *cast()* and *isInstance()* methods are used instead of *cast* and *instanceof* operators correspondingly.

It's common to use *cast()* and *isInstance()* methods with generic types.

Let's create *AnimalFeederGeneric<T>* class with *feed()* method that "feeds" only one type of animal, cats or dogs, depending on the value of the type parameter:

```

public class AnimalFeederGeneric<T> {
    private Class<T> type;

    public AnimalFeederGeneric(Class<T> type) {
        this.type = type;
    }

    public List<T> feed(List<Animal> animals) {
        List<T> list = new ArrayList<T>();
        animals.forEach(animal -> {
            if (type.isInstance(animal)) {
                T objAsType = type.cast(animal);
                list.add(objAsType);
            }
        });
        return list;
    }
}

```

The *feed()* method checks each animal and returns only those that are instances of *T*.

Note that the *Class* instance should also be passed to the generic class, as we can't get it from the type parameter *T*. In our example, we pass it in the constructor.

Let's make *T* equal to *Cat* and make sure that the method returns only cats:

@Test

```

public void whenParameterCat_thenOnlyCatsFed() {

```

```

    List<Animal> animals = new ArrayList<>();

```

```

    animals.add(new Cat());

```

```

    animals.add(new Dog());

```

```

    AnimalFeederGeneric<Cat> catFeeder

```

```

        = new AnimalFeederGeneric<Cat>(Cat.class);

List<Cat> fedAnimals = catFeeder.feed(animals);

assertTrue(fedAnimals.size() == 1);

assertTrue(fedAnimals.get(0) instanceof Cat);

}

```

73. Różnica między enkapsulacją, a abstrakcją

1. Abstrakcja rozwiązuje problem na poziomie projektu, podczas gdy hermetyzacja rozwiązuje problem na poziomie wdrożenia
2. Abstrakcja służy do ukrywania niechcianych danych i podawania odpowiednich danych. podczas gdy enkapsulacja oznacza ukrywanie kodu i danych w jednej jednostce w celu ochrony danych przed światem zewnętrznym.
3. Abstrakcja pozwala skupić się na tym, co robi obiekt, zamiast na tym, jak to robi, podczas gdy enkapsulacja oznacza ukrywanie wewnętrznych szczegółów lub mechaniki tego, jak obiekt coś robi.
4. Na przykład: Zewnętrzny wygląd telewizora, tak jak ma ekran wyświetlacza i przyciski kanałów do zmiany kanału, wyjaśnia Abstrakcja ale wewnętrzne szczegóły dotyczące implementacji telewizora, w jaki sposób CRT i ekran wyświetlacza łączą się ze sobą za pomocą różnych obwodów, wyjaśnia Enkapsulacja.

Abstraction is the method of hiding the unwanted information. Whereas encapsulation is a method to hide the data in a single entity or unit along with a method to protect information from outside.

We can implement abstraction using abstract class and interfaces. Whereas encapsulation can be implemented using by access modifier i.e. private, protected and public.

In abstraction, problems are solved at the design or interface level. While in encapsulation, problems are solved at the implementation level.

74. What is a final variable ?

Final variable is a constant variable. Variable value can't be changed after instantiation.

75. What is the difference between collections class vs collection interface ?

Collections class is a utility class having static methods for doing operations on objects of classes which implement the Collection interface. For example, Collections has methods for finding the max element in a Collection.

76. What is the advantage of using arrays over variables ?

Tablice zapewniają strukturę, w której można uzyskać dostęp do wielu wartości za pomocą pojedynczego odwołania i indeksu. Pomaga to w iterowaniu wartości za pomocą pętli.

77. What does it mean that Java arrays are homogeneous, but ArrayLists are not?

Arrays have a runtime check on the type of the added element. That is, if a new element that is not of the same type is added, an [ArrayStoreException](#) is thrown at runtime. That's why they are considered as "homogeneous".

This is not true for ArrayLists (Lists in general). Due to type erasure at runtime, it can practically hold any object.

The following throws an exception when running:

```
Object[] array = new String[3];
array[0] = "a";
array[1] = 1; // throws java.lang.ArrayStoreException
```

unlike the following which compiles and runs without problem (although with a compiler warning as it doesn't properly use generics):

```
ArrayList list = new ArrayList<String>();
list.add("a");
list.add(1); // OK
list.add(new Object()); // OK
```

With a correct use of generics, i.e. declaring the variable `list` above of type `ArrayList<String>` instead of `ArrayList`, the problem is avoided at compile-time:

```
ArrayList<String> list = new ArrayList<String>();
list.add("a");
list.add(1); // compilation error
list.add(new Object()); // compilation error
```

But even with a generically declared list, you can have something like this work without an exception at runtime:

```
ArrayList<String> list = new ArrayList<String>();
list.add("a");
Method[] methods = List.class.getMethods();
for(Method m : methods) {
    if(m.getName().equals("add")) {
        m.invoke(list, 1);
        break;
    }
}
```



```
}  
System.out.println(list.get(0));  
System.out.println((Object) list.get(1));
```

Output:

a

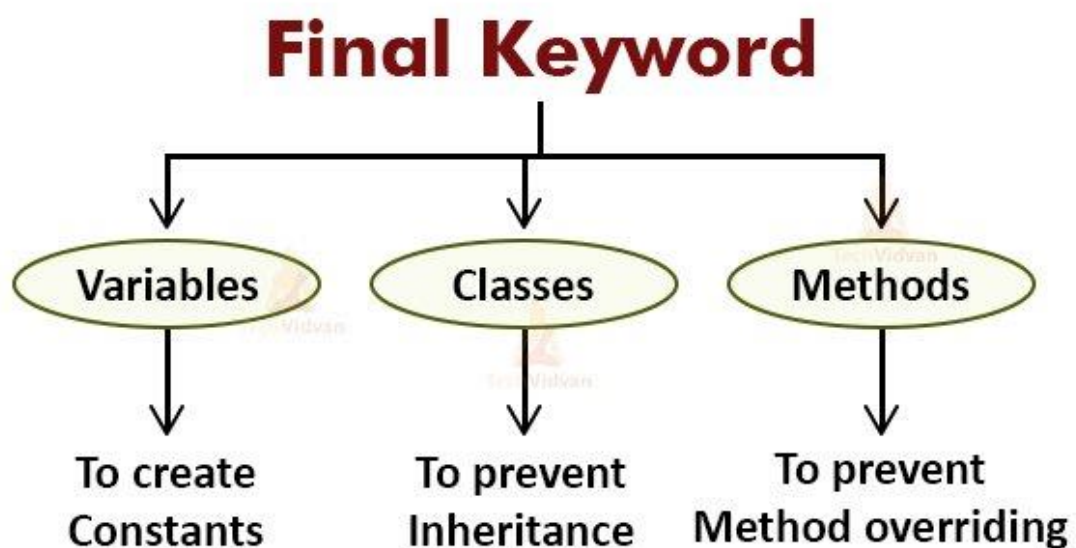
1

78. What is the difference between float and double?

Float can represent up to 7 digits accurately after decimal point, where as double can represent up to 15 digits accurately after decimal point.

79. Final Method

Metoda finalna nie może zostać nadpisana, ale oczywiście jest ona dziedziczona.



80. Interfejsy

Poza zwykłymi metodami w interfejsie mogą się znajdować

- metody domyślne,
- metody prywatne,
- metody statyczne,
- stałe.

Metody domyślne

Istnieje możliwość zdefiniowania tak zwanych metod domyślnych. Metody te mogą mieć właściwą implementację w ciele interfejsu. Metody takie poprzedzone są słowem kluczowym `default` jak w przykładzie poniżej

```
public interface MicrowaveOven {  
    void start();  
  
    void setDuration(int durationInSeconds);  
  
    boolean isFinished();  
  
    void setPower(int power);  
  
    default String getName() {  
        return "MicrowaveOven";  
    }  
}
```

Klasy, które implementują interfejs mogą nadpisać metodę domyślną.

Metody prywatne

Metody prywatne poprzedzone są słowem kluczowym `private`. Metody prywatne, w odróżnieniu od pozostałych, mogą być wywołane wyłącznie w definicji interfejsu.

Z racji tego ograniczenia, metody prywatne w interfejsach mają sens wyłącznie w połączeniu z metodami domyślnymi. Proszę spojrz na przykład poniżej, w którym modyfikuję interfejs `MicrowaveOven`:

```
public interface MicrowaveOven {  
    // removed for brevity  
    default Duration getRecommendedDefrostTime(double foodWeightInGrams) {  
        double frostRate = 0.8;  
        int power = 300;  
        return getRecommendedTime(power, frostRate, foodWeightInGrams);  
    }  
}
```

```

    default Duration getRecommendedWarmingUpTime(double foodWeightInGrams) {
        double frostRate = 0.2;
        int power = 700;
        return getRecommendedTime(power, frostRate, foodWeightInGrams);
    }

    private Duration getRecommendedTime(int power, double frostRate, double
foodWeightInGrams) {
        double durationInMinutes = foodWeightInGrams / ((1 - frostRate) * power);
        long durationInSeconds = (long) (durationInMinutes * 60);
        return Duration.ofSeconds(durationInSeconds);
    }
}

```

Metody prywatne w interfejsach pozwalają na usunięcie kodu, który powtarza się w wielu miejscach. Ten powtarzający się kod jest wówczas zawarty w ciele metody prywatnej.

W przykładzie powyżej dwie domyślne metody `getRecommendedDefrostTime` i `getRecommendedWarmingUpTime` używają metody prywatnej `getRecommendedTime`, która pozwala na użycie „magicznego” wzoru na obliczanie zalecanej długości czasu pracy mikrofalówki. Bez tej metody wzór musiałby znaleźć się w obu metodach co powodowałoby duplikację kodu.

Wartości niezmiennie i stałe

```
int counter = 123;
```

`counter` to zmienna. Do zmiennej `counter` możemy przypisać nową wartość:

```
counter = counter + 1;
```

Wartości niezmiennie w odróżnieniu od zmiennych poprzedzamy słowem kluczowym `final`. Poniżej możesz zobaczyć przykład klasy z atrybutem, którego wartości nie możemy przypisać na nowo. Atrybuty tego typu możemy inicjalizować jak w przykładzie poniżej: bezpośrednio bądź w ciele konstruktora.

```

public class Calculator {
    public final double PI = 3.14;
    public final double SQRT_2;

    public Calculator() {
        SQRT_2 = Math.sqrt(2);
    }
}

```

Wartości niezmiennie, podobnie jak metody, mogą być przypisane do instancji bądź klasy. Jeśli taka wartość przypisana jest do klasy mówimy wówczas o stałej. Jeśli chcemy aby stała była przypisana do klasy poprzedzamy ją słowem kluczowym `static`.

Do stałych wartość możemy przypisać wyłącznie raz – podczas inicjalizacji klasy. Zgodnie z konwencją nazewnictwa stałe piszemy wielkimi literami.

```
public interface Cat {  
    int NUMBER_OF_PAWS = 4;  
}
```

W interfejsie powyżej mamy stałą, która pokazuje ile łap ma kot. Domyślnie wszystkie atrybuty interfejsu są stałymi publicznymi przypisanymi do interfejsu więc słowa kluczowe `public static final` mogą zostać pominięte.

Interfejs dziedziczący drugi interfejs dziedziczy jego metody, może je nadpisać dodając słówko `default` do nazwy metody.

Interfejs znacznikowy

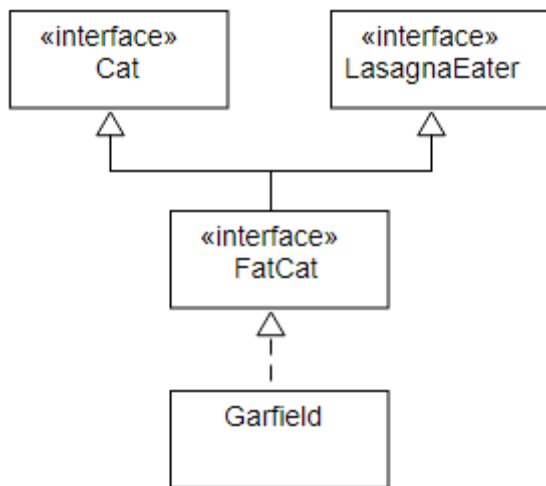
A czy możliwa jest sytuacja kiedy interfejs nie ma żadnej metody? Oczywiście, że tak. Mówimy wówczas o interfejsie znacznikowym. Jak sama nazwa wskazuje służy on do oznaczenia, danej klasy. Dzięki temu możesz przekazać zestaw dodatkowych informacji. Przykładem takiego interfejsu jest [java.io.Serializable](#), którego używamy aby dać znać kompilatorowi, że dana klasa jest serializowalna (o [serializacji](#) przeczytasz w innym artykule).

Interfejs a typ obiektu

Każdy obiekt w języku Java może być przypisany do zmiennej określonego typu. W najprostszym przypadku jest to jego klasa.

Interfejsy pozwalają na przypisanie obiektu do zmiennej typu interfejsu. Wydaje się to trochę skomplikowane jednak mam nadzieję, że przykład poniżej pomoże w zrozumieniu tego tematu.

```
public class Garfield implements FatCat {  
    // implementacja metod  
}
```



Przykład hierarchii dziedziczenia

```
Garfield garfield = new Garfield();  
FatCat fatCat = new Garfield();  
Cat cat = new Garfield();  
LasagnaEater lasagnaEater = new Garfield();
```

Instancję klasy `Garfield` możemy przypisać zarówno do zmiennej klasy `Garfield` jak i każdego z interfejsów, który ta klasa implementuje (bezpośrednio lub pośrednio). Chociaż w trakcie wykonania programu każdy z obiektów jest tego samego typu (instancja klasy `Garfield`), to w trakcie kompilacji sprawa wygląda trochę inaczej:

- na obiekcie `garfield` możemy wykonać wszystkie metody udostępnione w klasie `Garfield` i interfejsach, które ta klasa implementuje:

- `getWeight()`,

- `getName()`,
- `getLasagnaReceipe()`.

- na obiekcie `fatCat` możemy wykonać wszystkie metody udostępnione w interfejsie `FatCat` i interfejsach po których dziedziczy:

- `getWeight()`,
- `getName()`,
- `getLasagnaReceipe()`.

- na obiekcie `cat` możemy wykonać wyłącznie metody z interfejsu `Cat`:

- `getName()`.

- na obiekcie `lasagnaEater` możemy wykonać wyłącznie metody z interfejsu `LasagnaEater`:

- `getLasagnaReceipe()`.

Zastosowania interfejsów

Do czego właściwie potrzebne są nam interfejsy? Czy nie jest to po prostu zestaw dodatkowych linijek kodu, które trzeba napisać i nic one nie wnoszą? Otóż nie.

Interfejsy w bardzo prosty sposób ułatwiają różnego rodzaju integrację różnych fragmentów kodu. Wyobraź sobie sytuację, w której Piotrek pisze program obliczający średnią temperaturę w każdym z województw. Współpracuje on z Kasią, która pisze program udostępniający aktualną temperaturę w danej miejscowości.

Aby Piotrek mógł napisać swój program musi skorzystać z programu Kasi. Musi się z nim zintegrować. Taką integrację ułatwiają właśnie interfejsy.

Piotrek z Kasią uzgadniają, że będą używali następującego interfejsu

```
public interface Thermometer {  
    double getCurrentTemperatureFor(String city);  
}
```

Dzięki niemu Piotrek może pisać swój program równolegle z Kasią.

Co więcej może się okazać, że implementacja Kasi nie jest zbyt dokładna. Ania implementuje ten sam interfejs ale temperatury przez nią zwracane są dokładniejsze. Wówczas Piotrek w ogóle nie musi zmieniać swojego programu. Wystarczy, że użyje innej implementacji interfejsu Thermometer dostarczonej przez Anię.

To właśnie jest kolejna zaleta interfejsów. Dzięki nim możemy pisać programy, które możemy w łatwiejszy sposób modyfikować. Interfejsy jasno oddzielają komponenty programu. Dzięki takiemu podejściu komponenty można z łatwością wymieniać.

Klasa abstrakcyjna może mieć
normalne zmienne instancyjne. Tego

nie było w interfejsach. Może posiadać zmienne statyczne, finalne, publiczne, chronione, prywatne (wszystkie).

Metoda domyślna interfejsu może zostać nadpisana jako abstrakcyjna, a abstrakcyjna jako domyślna.

W klasie abstrakcyjnej metoda z ciałem może stać się abstrakcyjna w klasie abstrakcyjnej dziedziczącej i na odwrót.

81. Optional w Javie

Dwie kluczowe cechy Java Optional

Przede wszystkim Optional ma nas chronić przed otrzymaniem wartości `NULL`. Może zaistnieć to w przypadku kiedy próbujemy się odwołać do elementu (np. w kolekcji), który nie istnieje.

Podobnie ogromną korzyścią wynikającą ze stosowania Optional jest bogate API, dzięki któremu w elegancki sposób, możemy obsłużyć brak zasobów.

Dokumentacja Oracle

Jeśli chcesz korzystać z zasobów Oracle, do nauki tego zagadnienia, to koniecznie korzystaj z najnowszej dokumentacji dla Java 11. Dlatego tu podaje Ci właściwy link – [click!](#)

W zawiązku z tym, że dokumentacja mało tłumaczy toteż w tym artykule przedstawię najczęściej wykorzystane przykłady.

Kiedy Optional uratuje Ci skórę?

Założmy, że mamy w kodzie kolekcję 5-cio elementową książek. Następnie otrzymujemy żądanie, które próbuje się odwołać do elementu, który nie istnieje na liście:

```
1 books.get(10);
```

Wówczas, taka akcja kończy się błędem – a przecież można ładnie ją obsłużyć. Z pewnością to zapewnia nam właśnie Optional.

Przykłady stosowania Optional w Java

Po pierwsze kod startowy – poligon w ramach którego będziemy realizować późniejsze przypadki. Przede wszystkim mamy w nim listę 3 książek. Natomiast książka składa się z tytułu i nr ISBN.

```
1 public class BasicExample {
2
3     private List<Book> books;
4
5     public BasicExample() {
6         init();
7         Optional<Book> book = books.stream().filter(element ->
8 element.getTitle().equals("Książka Spring Boot 2")).findFirst();
9     }
10
11     private void init() {
12         books = new ArrayList<>();
13         books.add(new Book("Książka Spring Boot 2", "9782123456803"));
14         books.add(new Book("Aplikcje internetowe", "9782123456803"));
15         books.add(new Book("Java dla zaawansowanych", "9782123456803"));
16     }
17
18     public static void main(String[] args) {
19         new BasicExample();
20     }
```

W rezultacie linijki 7 otrzymujemy typ zwrotny jakim jest:

```
1 Optional<book> book;
```

A zatem możemy wykonać kolejne przypadki pochodzące z klasy Optional.

Przykład 1 – *ifPresent*

Który wykonuje akcję zdefiniowaną w ramach interfejsu funkcyjnego w przypadku kiedy dany element istnieje. Więc w tym przypadku wypiszę zawartość:

```
1 book.ifPresent(System.out::println)
```

Przykład 2 – *ifPresentOrElse*

Która dodatkowo w drugim parametrze przyjmuje akcję, która ma się wykonać w przypadku nieznaalezienia danego elementu:

```
1 book.ifPresentOrElse(
```

```
2         System.out::println,  
3         () -> System.out.println("book doesn't exist"));
```

Przykład 3 – *orElseThrow*

W przypadku nieznaalezienia danego nieznaalezienia można również rzucić komunikat o błędzie:

```
1 Book book1 = book.orElseThrow(  
2     () -> new RuntimeException("book doesn't exist"));
```

Przykład 4 – *orElseGet*

Jeśli dany element istnieje, to go zwróci. Natomiast w przypadku jego braku zwróci element, który został zdefiniowany w ramach interfejsu funkcyjnego:

```
1 book.orElseGet(  
2     () -> new Book("Spring w akcji", "9782123226803"));
```

Podsumowując powyższe cztery przykłady są najczęstszymi przykładami zastosowania *Optional* i warto je znać

82. Czym jest persystencja

Persystencja danych – zapisywanie danych na zewnątrz naszego programu. Przykładami takiego działania jest zapisywanie danych do bazy danych (najpopularniejsze) lub do pliku.

83. Access modifiers

Java - Access Modifiers

- Visible to the package, the default. No modifiers are needed.
- Visible to the class only (private).
- Visible to the world (public).
- Visible to the package and all subclasses (protected).

84. Kontekst statyczny

Metody i zmienne w klasie mogą odwoływać się do swoich metod i zmiennych statycznych i do zwykłych metod i zmiennych, ale metody i zmienne statyczne nie mogą odwoływać się do zwykłych metod i zmiennych.

85. Najpierw wywoływany jest blok inicjalizacyjny czy konstruktor?

Bloki inicjalizacji instancji są wykonywane za każdym razem, gdy klasa jest inicjowana i przed wywołaniem konstruktorów.

86. Integer.parseInt() vs .valueOf()

Integer.parseInt()

It can only take a String as a parameter.

It returns a primitive int value.

When an integer is passed as parameter, it produces an error due to incompatible types

This method produces an error(incompatible types) when a character is passed as parameter.

This lags behind in terms of performance since parsing a string takes a lot of time when compared to generating one.

If we need the primitive int datatype then Integer.parseInt() method is to be used.

Integer.valueOf()

It can take a String as well as an integer as parameter.

It returns an Integer object.

When an integer is passed as parameter, it returns an Integer object corresponding to the given parameter.

This method can take a character as parameter and will return the corresponding unicode.

This method is likely to yield significantly better space and time performance by caching frequently requested values.

If Wrapper Integer object is needed then valueOf() method is to be used.

87. Difference Between ArrayList and LinkedList

ArrayList and LinkedList both implement the List interface and maintain insertion order. Both are non-synchronized classes.

However, there are many differences between the ArrayList and LinkedList classes that are given below.

ArrayList

Implemented as a dynamic array using an array of objects to store elements sequentially

Implements the List interface

Faster to store and access data using index

Removal and addition are slow because all the other items have to be shifted left or right accordingly.

Removing an item while iterating through the list is slower because all the other elements have to be shuffled to fill the gaps in the list created by the removed elements

You can specify the size of the initial ArrayList using ArrayList(int capacity) constructor. The size can still grow later, however, an initial amount can be set. ArrayList works even faster if the capacity specified is not exceeded.

LinkedList

Implemented as a doubly-linked list (in Java), consists of pointers to previous and next nodes.

Implements List and Deque (double ended Queue) which gives it additional functionality

Retrieval and storage are a bit slow as the list has to traverse through each node to identify the correct one.

Faster and more efficient for data manipulation like adding and removing elements from any part of the list

Removing an element while iterating is simple and only the next and previous nodes have to be adjusted.

LinkedList supports only no-argument constructor – LinkedList()

No provision for getting the methods of Queue or Deque.

Implements Deque, that provides descendingIterator() that returns iterator having list value in reverse sequence. Same way methods like peek(), poll() and offer() (and its variants) are available so that the list function as a queue.

ArrayList cannot be used as a Stack.

As Deque also supports the methods push(), peek() and pop(), LinkedList can be used as a Stack.

ArrayList vs LinkedList

There are few **similarities between** these classes which are as follows:

1. Both ArrayList and LinkedList are implementation of List interface.
2. They both maintain the elements insertion order which means while displaying ArrayList and LinkedList elements the result set would be having the same order in which the elements got inserted into the List.
3. Both these classes are non-synchronized and can be made synchronized explicitly by using Collections.synchronizedList method.
4. The iterator and listIterator returned by these classes are fail-fast (if list is structurally modified at any time after the iterator is created, in any way except through the iterator's own remove or add methods, the iterator will throw a ConcurrentModificationException).

Add elements

ArrayList:

When we're creating an empty ArrayList, it initializes its backing array with a default capacity (currently 10):

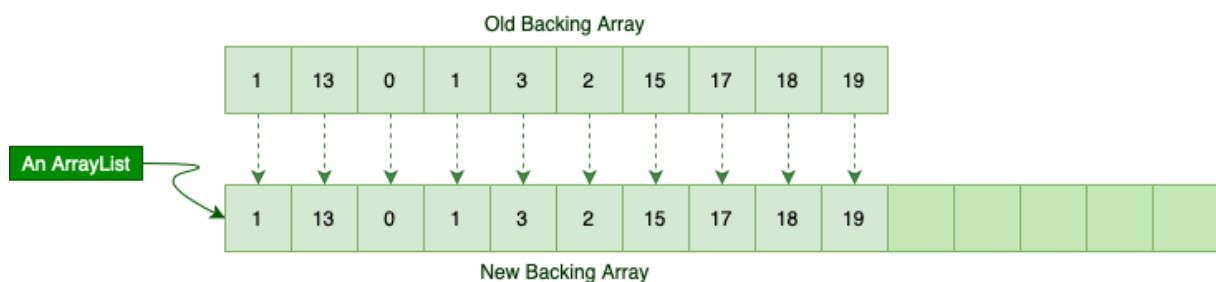


Adding a new item while that array is not yet full is as simple as assigning that item to a specific array index. This array index is determined by the current array size since we're practically appending to the list:

```
backingArray[size] = newItem;
```

```
size++;
```

So, in best and average cases, the time complexity for the add operation is $O(1)$, which is pretty fast. As the backing array becomes full, however, the add implementation becomes less efficient:

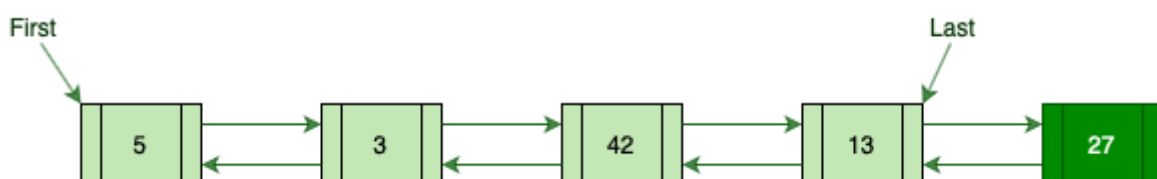


To add a new item, we should first initialize a brand new array with more capacity and copy all existing items to the new array. **Only after copying current elements can we add the new item. Hence, the time complexity is $O(n)$ in the worst case since we have to copy n elements first.**

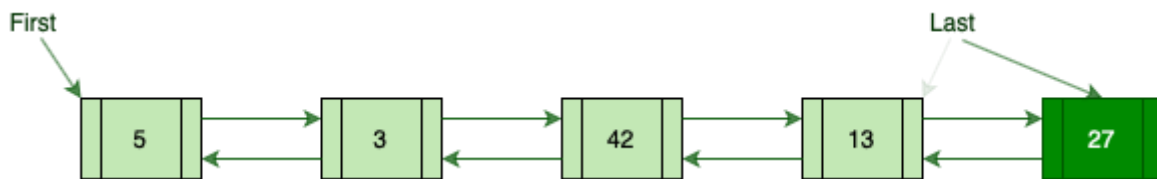
Theoretically speaking, adding a new element runs in amortized constant time. **That is, adding n elements requires $O(n)$ time.** However, some single additions may perform poorly because of the copy overhead.

LinkedList:

In order to add a new node, first, we should link the current last node to the new node:



And then update the last pointer:



As both of these operations are trivial, the time complexity for the add operation is always **$O(1)$** .

Access by Index

ArrayList:

Accessing items by their indices is where the *ArrayList* really shines. To retrieve an item at index i , we just have to return the item residing at the i^{th} index from the backing array. Consequently, **the time complexity for access by index operation is always $O(1)$** .

LinkedList:

LinkedList, as opposed to ArrayList, does not support fast random access. So, in order to find an element by index, we should traverse some portion of the list manually.

In the best case, when the requested item is near the start or end of the list, the time complexity would be as fast as **$O(1)$** . However, in the average and worst-case scenarios, we may end up with an **$O(n)$** access time since we have to examine many nodes one after another.

Remove by Index

LinkedList remove operation gives $O(1)$ performance while ArrayList gives variable performance: $O(n)$ in worst case (while removing first element) and $O(1)$ in best case (While removing last element).

Conclusion: LinkedList element deletion is faster compared to ArrayList.

Reason: LinkedList's each element maintains two pointers (addresses) which points to the both neighbor elements in the list. Hence removal only requires change in the pointer location in the two neighbor nodes (elements) of the node which is going to be removed. While In ArrayList all the elements need to be shifted to fill out the space created by removed element.

ArrayList:

Suppose we're going to remove the index 6 from our *ArrayList*, which corresponds to the element 15 in our backing array:



After marking the desired element as deleted, we should move all elements after it back by one index. **Obviously, the nearer the element to the start of the array, the more elements we should move.** So the time complexity is **$O(1)$** at the best-case and **$O(n)$** on average and worst-cases.

LinkedList:

In order to remove an item, we should first find the requested item and then un-link it from the list. Consequently, the access time determines the time complexity — that is, **$O(1)$** at best-case and **$O(n)$** on average and in worst-case scenarios.

Applications and Limitations

ArrayList:

Zazwyczaj ArrayList jest domyślnym wyborem dla wielu programistów, gdy potrzebują implementacji listy. W rzeczywistości jest to rozsądny wybór, gdy liczba odczytów jest znacznie większa niż liczba zapisów.

Czasami potrzebujemy równie częstych czytań i zapisów. Jeśli mamy oszacowaną maksymalną liczbę możliwych elementów, nadal sensowne jest użycie ArrayList. W takim przypadku możemy zainicjować ArrayList z początkową pojemnością:

```
int possibleUpperBound = 10_000;
```

```
List<String> items = new ArrayList<>(possibleUpperBound);
```

Takie oszacowanie może zapobiec wielu niepotrzebnemu kopiowaniu i alokacji tablicy.

Ponadto tablice są indeksowane według wartości int w Javie. W związku z tym nie można przechowywać więcej niż 2^{32} elementów w tablicy Java, a co za tym idzie, w ArrayList.

LinkedList:

LinkedLists są bardziej odpowiednie, gdy ilość dodanych elementów jest znacznie wyższa niż ilość odczytów.

Może być również używany w scenariuszach z dużą ilością odczytu, gdy przez większość czasu chcemy mieć pierwszy lub ostatni element. Warto wspomnieć, że LinkedList implementuje również interfejs **Deque** – wspierając wydajny dostęp do obu końców kolekcji.

Ogólnie rzecz biorąc, jeśli znamy różnice w ich implementacji, możemy łatwo wybrać jeden dla konkretnego przypadku użycia.

Założmy na przykład, że będziemy przechowywać wiele zdarzeń szeregów czasowych w strukturze danych podobnej do listy. Wiemy, że co sekundę otrzymywalibyśmy wybuchy wydarzeń.

Ponadto musimy okresowo sprawdzać wszystkie wydarzenia jeden po drugim i dostarczać pewne statystyki. W tym przypadku **LinkedList jest lepszym wyborem, ponieważ szybkość dodawania jest znacznie wyższa niż szybkość odczytu.**

Ponadto przeczytalibyśmy wszystkie pozycje, więc nie możemy pokonać górnej granicy $O(n)$.

88. Referencje do metod

W Javie poza standardowym użyciem lambda, możemy korzystać także z **Metod Reference**. Jest to uproszczony zapis niektórych lambda. Jeśli wyrażenie lambda nie robi nic poza wywołaniem jakiejś metody, i ta metoda jest wywoływana z taką samą listą parametrów, jak wyrażenie lambda, to możemy wtedy użyć referencji do metody. Co sprowadza się do uproszczonego zapisu wywołania metody (z operatorem ::) bez parametrów.

```
// lambda
Consumer<String> consumer = s -> System.out.println(s);
// method reference
Consumer<String> consumer = System.out::println;
```

89. Stream

Na tablicach nawet prymitywnych też można używać Stream, korzystając z `Arrays.stream(nazwa kolekcji)`

Strumień to sekwencje elementów, które mogą być przetwarzane sekwencyjnie, ale także równoległe (parallel), przez różnego rodzaju operacje agregujące. W Javie wiele elementów takich jak kolekcje, czy tablice, mogą być przetwarzane jako strumień (lub mówiąc bardziej precyzyjnie mogą być konwertowane na strumień).

Najprostszym przykładem przetwarzania strumienia jest filtrowanie elementów z listy:

```
List<String> list = Arrays.asList("a", "aa", "b", "c", "cc", "dd", "e");
```

załóżmy że chcemy odfiltrować z naszej listy tylko elementy, które mają tylko jeden znak `length() == 1`. Możemy skorzystać z metody `filter` (w klasie `Stream`), która jako parametr przyjmuje predykat:

```
List<String> collected = list.stream()
    .filter(s -> s.length() == 1)
    .collect(Collectors.toList());
System.out.println(collected);
```

Po przefiltrowaniu strumienia używamy metody `collect(...)` i odpowiedniego kolektora (w tym wypadku `Collectors.toList()`), żeby zebrać elementy w **nową listę**.

Po wydrukowaniu otrzymamy listę zawierającą elementy:

[a, b, c, e]

W przetwarzaniu strumieni w Javie mamy dostępnych wiele operacji, które możemy łączyć ze sobą, często w różnej kolejności. Operacje te dzielą się na **pośrednie** i **terminalne**. I to te drugie uruchamiają całe przetwarzanie streamu. W powyższym przykładzie, operacją pośrednią jest metoda filter, a terminalną collect. Łatwo możemy rozróżnić te metody, sprawdzając jaki typ zwracają. Operacje pośrednie zwracają obiekt Stream, a operacje terminalne zwracają jakiś **konkretny obiekt** (może to także być typ prymitywny), **kolekcję lub Optional**.

Poza filtrowaniem mamy całą gamę operacji pośrednich, które możemy łączyć ze sobą:

map – mapuje jeden obiekt na inny

flatMap – zwraca „spłaszczony” strumień elementów (np. strumień, który powstał z kolekcji)

distinct – zwraca strumień unikalnych elementów

sorted – zwraca strumień naturalnie posortowanych elementów

peek – zwraca element strumienia (można jej użyć jako pomoc przy debugowaniu)

limit – ogranicza elementy strumienia

skip – pomija określoną ilość elementów strumienia

Mamy także dostępnych kilka operacji terminalnych:

collect – grupuje elementy strumienia do odpowiedniego obiektu

forEach – wykonuje zadaną akcję dla każdego elementu strumienia

reduce – redukuje strumień do jakiejś wartości (np. obiektu)

min – wyciąga minimalną wartość ze strumienia

max – wyciąga maksymalną wartość ze strumienia

count – zlicza elementy strumienia

anyMatch – sprawdza, czy którykolwiek z elementów strumienia odpowiada zadanemu predykatowi

allMatch – sprawdza, czy wszystkie elementy strumienia odpowiadają zadanemu predykatowi

noneMatch – sprawdza, czy wszystkie elementy nie odpowiadają zadanemu predykatowi

findFirst – zwraca pierwszy element z przetworzonego strumienia

findAny – zwraca jakiś element strumienia (w przypadku strumieni przetwarzanych równoległe w wielu wątkach, poszukiwany element może być dowolnym elementem strumienia)

90. Typy generyczne

W uproszczeniu można powiedzieć, że typy generyczne są “szablonami”.

Szablonami, które parametryzują daną klasę/metodę.

Dzięki typom generycznym możemy uniknąć niepotrzebnego rzutowania. Ponadto przy ich pomocy kompilator jest w stanie sprawdzić poprawność typów na etapie kompilacji, oznacza to więcej błędów wykrytych w jej trakcie.

Poza tym dzięki typom generycznym możemy konstruować bardziej złożone klasy, które możemy używać w wielu kontekstach, łatwiej będzie Ci to zrozumieć na przykładzie.

Typy generyczne w Javie pozwalają realizować programowanie uogólnione, i znacznie redukują ilość powtarzającego się kodu. Zostały wprowadzone w **Javie 5** i stały się jednym z kluczowych elementów języka. Typowym przykładem ich wykorzystania są kolekcje wchodzące w skład Collections Framework, ale bez problemu można także spotkać ich zastosowanie we frameworkach oraz codziennym kodzie.

Po co nam typy generyczne?

Wyobraź sobie sytuację, w której chcesz przechować w swoim programie kilka obiektów tego samego typu, np. Book. Można do tego wykorzystać tablice i zapisać np.:

```
Book[] books = new Book[1000];
```

Tablice mają jednak swoje wady, nie są w stanie zwiększać swojego rozmiaru, gdy zabraknie w nich miejsca, usuwanie elementów jest kłopotliwe. Zamiast tego najczęściej korzystamy więc z kolekcji, np. list, albo zbiorów, które zostały wprowadzone do Javy w wersji 1.2 w postaci Collections Framework. Ze względu na to, że kolekcje powinny być uniwersalne, to pierwotnie operowały one na typie Object, który dawał możliwość wykorzystania polimorfizmu.

Tworząc więc np. obiekt ArrayList w taki sposób:

```
List books = new ArrayList();
```

do przechowywania obiektów w ramach kolekcji tworzona była tablica typu Object[]. Ma to oczywiście zaletę w postaci tego, że w tablicy takiej mogą być przechowywane obiekty dowolnego typu, ale ma równie duże wady. Przy pobieraniu obiektów z takiej kolekcji zwracane są referencje typu Object, co wymusza na nas późniejsze rzutowanie. Błąd jest więc zapis:

```
List books = new ArrayList();
books.add(new Book("Potop", "Henryk Sienkiewicz"));
Object book = books.get(0);
System.out.println(book.getTitle()); //błąd, w typie Object nie ma metody getTitle()
```

Konieczne jest rzutowanie:

```
List books = new ArrayList();
books.add(new Book("Potop", "Henryk Sienkiewicz"));
Book book = (Book) books.get(0);
System.out.println(book.getTitle()); //ok, referencja typu Book
```

To, co moglibyśmy zrobić w takiej sytuacji, to stworzenie własnych klas reprezentujących listy, dedykowane do odpowiednich typów, np. BookList, MagazineList, AuthorList itd. Zdecydowana większość kodu takich klas byłaby jednak powtarzalna, a różniłoby je tylko to, na jakich typach operują.

Typy generyczne

Znanym rozwiązaniem tego problemu jest programowanie uogólnione / generyczne. Problem polegał na tym, że wprowadzenie tego paradygmatu do Javy nie było takie proste, ponieważ od samego początku stawiano w niej na pierwszym miejscu kompatybilność wsteczną. Rozwiązaniem okazały się typy generyczne wprowadzone w Javie 1.5, które

reprezentowane są przez ostre nawiasy. Od teraz powyższy kod można było zapisać w sposób, który nie wymaga rzutowania:

```
List<Book> books = new ArrayList<Book>();  
books.add(new Book("Potop", "Henryk Sienkiewicz"));  
Book book = books.get(0); //metoda get() zwraca referencję Book, a  
nie Object  
System.out.println(book.getTitle()); //ok, referencja typu Book
```

W kolejnych wersjach Javy wprowadzano jeszcze pewne usprawnienia. W Javie 1.7 dodano automatyczne wnioskowanie typów (ang. type inference), dzięki czemu nie trzeba było już podawać typu generycznego dwukrotnie, tylko, zamiast tego można wykorzystać operator diamentu:

```
//przed Javą 7  
List<Book> books = new ArrayList<Book>();  
//od Javy 7  
List<Book> books = new ArrayList<>();
```

Oczywiście kolekcje są najprostszym przykładem wykorzystania typów generycznych, ale wykraczają one znacząco poza to. Możemy definiować zarówno własne klasy jak i pojedyncze metody generyczne.

Ograniczenia

Typy generyczne wprowadzone w Javie nie są rozwiązaniem idealnym i posiadają istotne ograniczenia. Najważniejsze z nich jest to, że jako typów generycznych możemy używać wyłącznie typów obiektowych. Nie możemy natomiast używać typów prostych. Z tego powodu możemy stworzyć listę, która będzie przechowywała obiekty String:

```
List<String> words = new ArrayList<>(); //ok
```

ale nie możemy stworzyć listy do zapisywania wartości int:

```
List<int> numbers = new ArrayList<>(); //błąd
```

Z tego powodu wraz z typami generycznymi w Javie 1.5 wprowadzono jednocześnie typy opakowujące i w celu stworzenia listy przechowującej liczby całkowite należy zapisać:

```
List<Integer> numbers = new ArrayList<>(); //ok
```

Przykład klasy generycznej

```
public class BoxOnSteroids<T> {  
    public T fruit;  
  
    public BoxOnSteroids(T fruit) {  
        this.fruit = fruit;  
    }  
  
    public T getFruit() {  
        return fruit;  
    }  
}
```

Metody z generycznymi argumentami

```
private static void method1(FancyBox<?> box) {  
    Object object = box.object;  
    System.out.println(object);  
}  
  
private static void plainWildcard() {  
    method1(new FancyBox<>(new Object()));  
    method1(new FancyBox<>(new Square()));  
    method1(new FancyBox<>(new Apple()));  
}
```

What is a finally block? Is there a case when finally will not execute?

Finally block is a block which always executes a set of statements. **It is always associated with a try block regardless of any exception that occurs or not.**

W interfejsie wszystkie metody są publiczne (oprócz private)

91. Reason of using **List = new ArrayList** instead of **ArrayList = new ArrayList**

Głównym powodem, dla którego to robisz, jest oddzielenie kodu od określonej implementacji interfejsu. Kiedy piszesz swój kod w ten sposób:

```
List list = new ArrayList();
```

reszta kodu wie tylko, że dane są typu List, co jest preferowane, ponieważ umożliwia łatwe przełączanie się między różnymi implementacjami interfejsu List. Załóżmy na przykład, że pisziesz dość dużą bibliotekę innej firmy i że zdecydowałeś się zaimplementować rdzeń swojej biblioteki za pomocą LinkedList. Jeśli twoja biblioteka w dużym stopniu opiera się na dostępie do elementów z tych list, w końcu okaże się, że podjąłeś złą decyzję projektową; zdasz sobie sprawę, że powinieneś użyć ArrayList (która daje czas dostępu $O(1)$) zamiast LinkedList (która daje czas dostępu $O(n)$). Zakładając, że programujesz interfejs, wprowadzenie takiej zmiany jest łatwe. Możesz po prostu zmienić instancję List z,

```
List list = new LinkedList();
```

to

```
List list = new ArrayList();
```

i wiesz, że to zadziała, ponieważ napisałeś swój kod zgodnie z umową dostarczoną przez interfejs List. Z drugiej strony, gdybyś zaimplementował rdzeń swojej biblioteki za pomocą LinkedList `list = new LinkedList()`, wprowadzenie takiej zmiany nie byłoby takie proste, ponieważ nie ma gwarancji, że reszta kodu nie korzysta z metod specyficznych dla klasy LinkedList. Podsumowując, wybór jest po prostu kwestią projektu... ale ten rodzaj projektowania jest bardzo ważny (szczególnie podczas pracy nad dużymi projektami), ponieważ pozwoli później wprowadzić zmiany specyficzne dla implementacji bez łamania istniejącego kodu.

92. Co zwróci poniższa metoda?

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


Odpowiedź: 5

Nie zostanie tu złapany żaden wyjątek, więc 4 nie może być zwrócone.

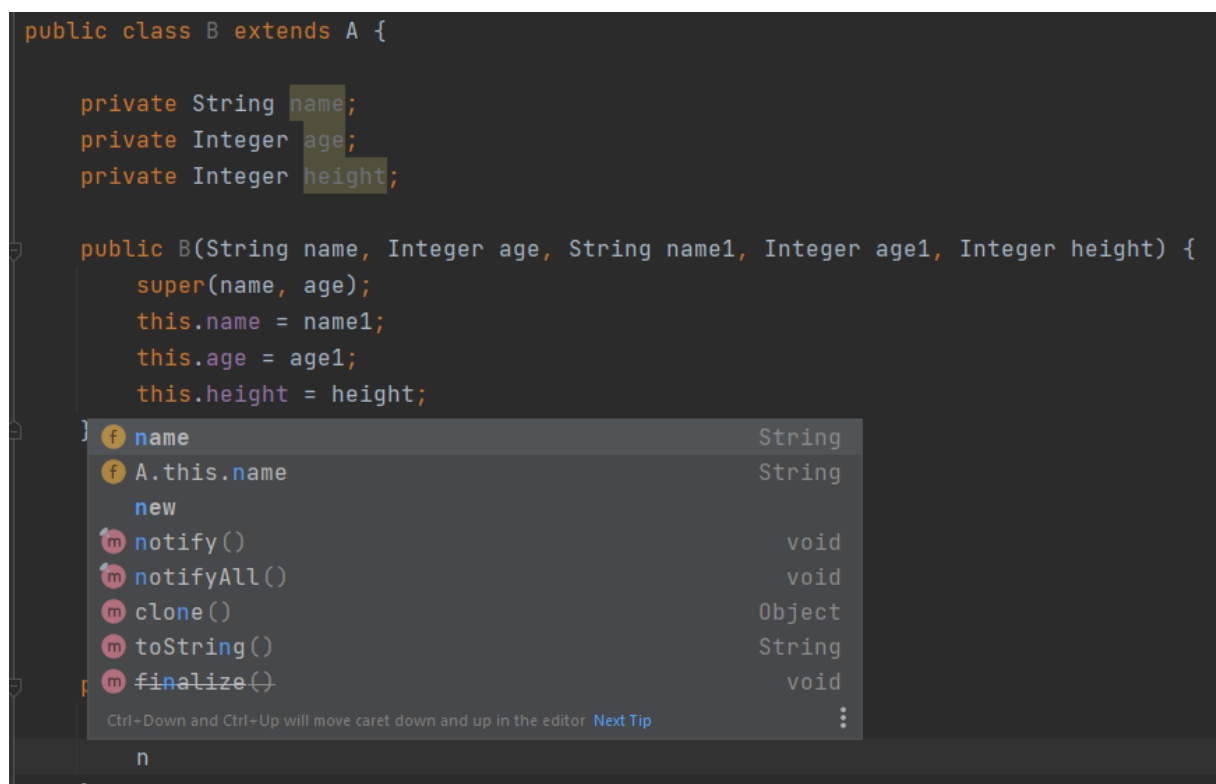
Kluczem tutaj jest fakt, że blok finally wykonuje się zawsze i w tym wypadku return 5 nadpisuje return 3.

Gdyby wyjątek był złapany również zwrócone byłoby 5.

93. Co się stanie jeśli w klasie dziedziczącej zadeklarujemy atrybuty o tych samych nazwach co w klasie, z której dziedziczymy.

Wtedy będą to dwa osobne atrybuty

```
public class B extends A {  
  
    private String name;  
    private Integer age;  
    private Integer height;  
  
    public B(String name, Integer age, String name1, Integer age1, Integer height) {  
        super(name, age);  
        this.name = name1;  
        this.age = age1;  
        this.height = height;  
    }  
}
```



94. Compilation and Execution of a Java Program

Java, being a platform-independent programming language, doesn't work on the one-step compilation. Instead, it involves a two-step execution, first through an OS-independent

compiler; and second, in a virtual machine (JVM) which is custom-built for every operating system.

The two principal stages are explained below:

Principle 1: Compilation

First, the source '.java' file is passed through the compiler, which then encodes the source code into a machine-independent encoding, known as Bytecode. The content of each class contained in the source file is stored in a separate '.class' file. While converting the source code into the bytecode, the compiler follows the following steps:

Step 1: Parse: Reads a set of *.java source files and maps the resulting token sequence into AST (Abstract Syntax Tree)-Nodes.

Step 2: Enter: Enters symbols for the definitions into the symbol table.

Step 3: Process annotations: If Requested, processes annotations found in the specified compilation units.

Step 4: Attribute: Attributes the Syntax trees. This step includes name resolution, type checking and constant folding.

Step 5: Flow: Performs dataflow analysis on the trees from the previous step. This includes checks for assignments and reachability.

Step 6: Desugar: Rewrites the AST and translates away some syntactic sugar.

Step 7: Generate: Generates '.Class' files.

Principle 2: Execution

The class files generated by the compiler are independent of the machine or the OS, which allows them to be run on any system. To run, the main class file (the class that contains the method main) is passed to the JVM and then goes through three main stages before the final machine code is executed. These stages are:

These states do include:

- 1. ClassLoader**
- 2. Bytecode Verifier**
- 3. Just-In-Time Compiler**

Let us discuss all 3 stages.

Stage 1: Class Loader

The main class is loaded into the memory bypassing its '.class' file to the JVM, through invoking the latter. All the other classes referenced in the program are loaded through the class loader.

A class loader, itself an object, creates a flat namespace of class bodies that are referenced by a string name. The method definition is provided below illustration as follows:

Illustration:

// loadClass function prototype

Class r = loadClass(String className, boolean resolveIt);

// className: name of the class to be loaded

// resolveIt: flag to decide whether any referenced class should be loaded or not.

There are two types of class loaders

- Primordial (pierwotny)
- non-primordial

The primordial class loader is embedded into all the JVMs and is the default class loader. A non-primordial class loader is a user-defined class loader, which can be coded in order to customize the class-loading process. Non-primordial class loader, if defined, is preferred over the default one, to load classes.

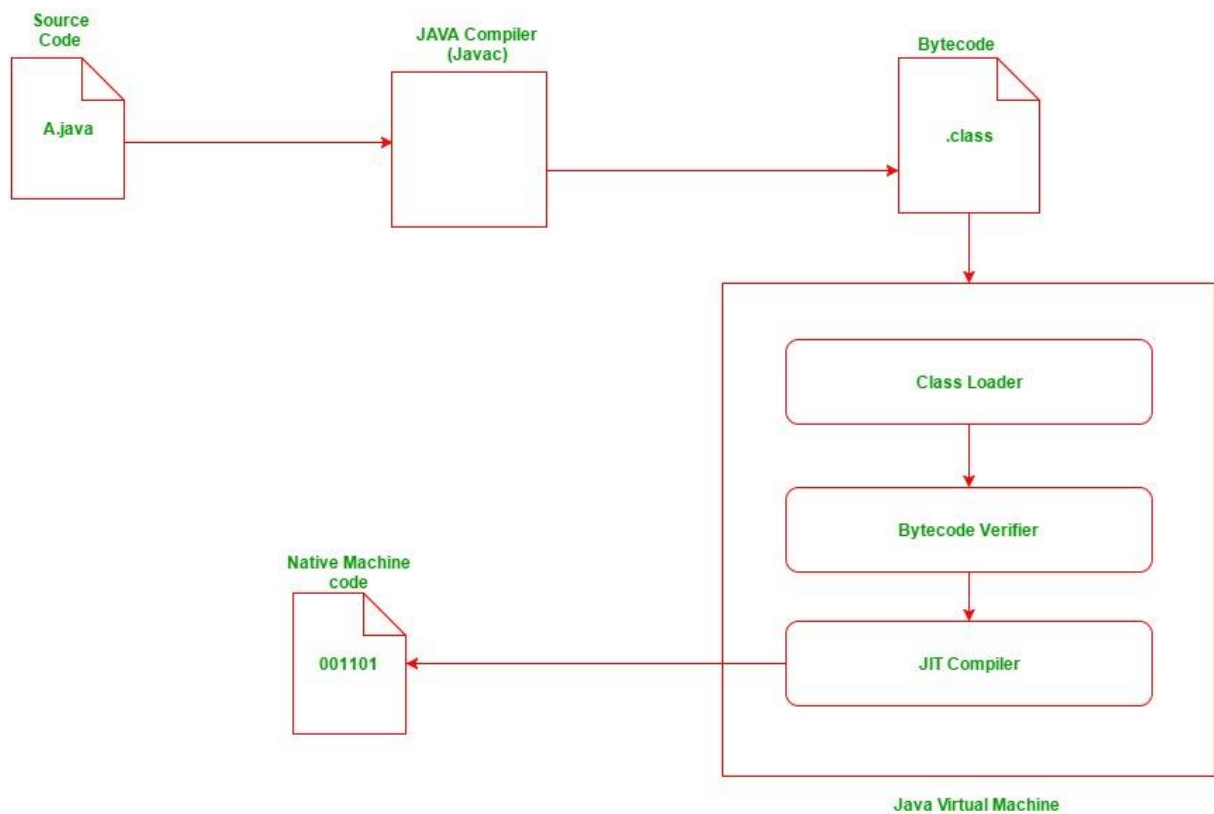
Stage 2: Bytecode Verifier

- Variables are initialized before they are used.
- Method calls match the types of object references.
- Rules for accessing private data and methods are not violated.

- Local variable accesses fall within the runtime stack.
- The run-time stack does not overflow.
- If any of the above checks fail, the verifier doesn't allow the class to be loaded.

Stage 3: Just-In-Time Compiler

This is the final stage encountered by the java program, and its job is to convert the loaded bytecode into machine code. When using a JIT compiler, the hardware can execute the native code, as opposed to having the JVM interpret the same sequence of bytecode repeatedly and incurring the penalty of a relatively lengthy translation process. This can lead to performance gains in the execution speed unless methods are executed less frequently.



The process can be well-illustrated by the following diagram as given above as follows from which we landed up to the conclusion.

Conclusion: *Due to the two-step execution process described above, a java program is independent of the target operating system. However, because of the same, the execution time is way more than a similar program written in a compiled platform-dependent program.*

95. Just In Time Compiler

Kompilator Just-In-Time (JIT) jest istotną częścią środowiska JRE, czyli Java Runtime Environment, która odpowiada za optymalizację wydajności aplikacji opartych na javie w czasie wykonywania. Kompilator jest jednym z kluczowych aspektów decydujących o wydajności aplikacji dla obu stron, tj. użytkownika końcowego i twórcy aplikacji.

Java JIT Compiler : General Overview

Kod bajtowy jest jedną z najważniejszych cech Javy, która pomaga w wykonywaniu na wielu platformach. Sposób konwersji kodu bajtowego na natywny język maszynowy w celu wykonania ma ogromny wpływ na jego szybkość. Te kody bajtowe muszą być interpretowane lub skompilowane do odpowiednich instrukcji maszynowych w zależności od architektury zestawu instrukcji. Co więcej, mogą one być wykonywane bezpośrednio, jeśli architektura instrukcji jest oparta na kodzie bajtowym. Interpretacja kodu bajtowego wpływa na szybkość wykonania. Aby poprawić wydajność, kompilatory JIT współdziałają z wirtualną maszyną Java (JVM) w czasie wykonywania i kompilują odpowiednie sekwencje kodu bajtowego w natywny kod maszynowy. Używając kompilatora JIT, sprzęt jest w stanie wykonać kod natywny, w przeciwieństwie do tego, że JVM wielokrotnie interpretuje tę samą sekwencję kodu bajtowego i ponosi koszty procesu tłumaczenia. Prowadzi to następnie do zwiększenia wydajności szybkości wykonywania, chyba że skompilowane metody są wykonywane rzadziej. Kompilator JIT jest w stanie wykonać pewne proste optymalizacje podczas kompilowania serii kodu bajtowego do natywnego języka maszynowego. Niektóre z tych optymalizacji wykonywanych przez kompilatory JIT to analiza danych, redukcja dostępu do pamięci poprzez alokację rejestru, translacja z operacji stosu na operacje na rejestrach, eliminacja wspólnych wyrażeń podrzędnych itp. Im większy stopień optymalizacji, tym więcej czasu kompilator JIT spędza na etapie wykonywania. Dlatego nie może sobie pozwolić na wszystkie optymalizacje, do których jest zdolny statyczny kompilator, ze względu na dodatkowy narzut dodany do czasu wykonania, a ponadto jego widok na program jest również ograniczony.

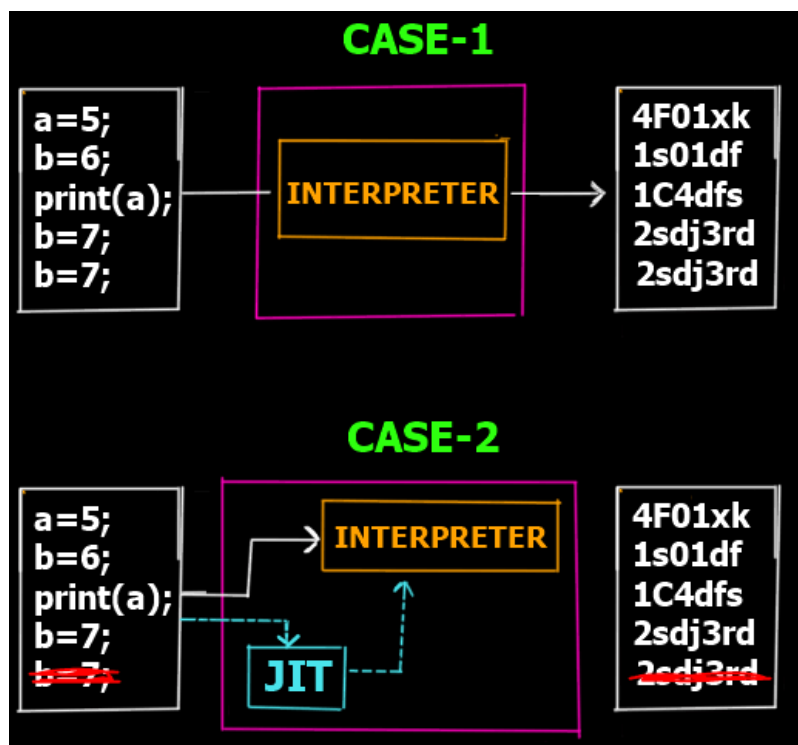
Working of JIT Compiler

Java podąża za podejściem obiektowym, w wyniku czego składa się z klas. Składają się one z kodu bajtowego, który jest neutralny dla platformy i jest wykonywany przez JVM w zróżnicowanych architekturach.

- W czasie wykonywania JVM ładuje pliki klas, określana jest semantyka każdego z nich i wykonywane są odpowiednie obliczenia. Dodatkowe użycie procesora i pamięci podczas interpretacji powoduje, że aplikacja Java działa wolniej w porównaniu z aplikacją natywną
- Kompilator JIT pomaga w poprawie wydajności programów Java poprzez kompilację kodu bajtowego w natywny kod maszynowy w czasie wykonywania.
- Kompilator JIT jest włączony przez cały czas, gdy jest aktywowany, gdy wywoływana jest metoda. W przypadku skompilowanej metody JVM bezpośrednio wywołuje skompilowany kod, zamiast go interpretować. Teoretycznie, gdyby kompilacja nie wymagała żadnego czasu procesora ani użycia pamięci, szybkość kompilatora natywnego i kompilatora Javy byłaby taka sama.
- Kompilacja JIT wymaga czasu procesora i użycia pamięci. Kiedy maszyna wirtualna Java uruchamia się po raz pierwszy, wywoływane są tysiące metod. Kompilacja wszystkich tych metod może znacząco wpłynąć na czas uruchamiania, nawet jeśli efektem końcowym jest bardzo dobra optymalizacja wydajności.

Working of JIT (Just-In-Time) Compiler

In the above steps we didn't mention the working of JIT compiler. Lets understand this by taking 2 case examples –



- **Case 1:**

- In case 1 you can see that we are at the interpretation phase (Step 5 of the overall program execution). Lets assume we have 5 lines which are supposed to be interpreted to their corresponding machine code lines. So as you can see in the Case 1 there is no JIT involved. thus the interpreter converts each line into its corresponding machine code line. However if you notice the last 2 lines are the same (consider it a redundant line inserted by mistake). Clearly that line is redundant and does not have any effect on the actual output but yet since the interpreter works line by line it still creates 5 lines of machine code for 5 lines of the bytecode.
- Now this is inefficient right? lets see case 2

- **Case 2:**

- In case 2 we have the JIT compiler. Now before the bytecode is passed onto the interpreter for conversion to machine code, the **JIT compiler scans** the full code to see if it can be **optimized**. As it finds the last line is redundant it removes it from the bytecode and passes only 4 lines to the interpreter thus making it more **efficient** and **faster** as the interpreter now has 1 line less to interpret.
- So this is how JIT compiler speeds up the overall execution process.

This was just one scenario where JIT compiler can help in making the execution process fast and efficient. There are other cases like *inclusion of only those packages needed in the code, code optimizations, redundant code removal* etc which overall makes the process very fast and efficient. Also different JITs developed by different companies work differently and JIT compilers are an **optional** step and not invoked everytime.

So this was the complete Execution Process of Java Program in Detail with the Working of JIT Compiler.

Moimi słowami

Just In Time Compiler sprawdza na bieżąco kod bajtowy interpretowany przez interpreter jvm. Jeżeli zauważy sytuację kiedy np. dana linia kodu powtarza się kilka razy kompiluje on ten fragment kodu i przekazuje do pamięci podręcznej. W wyniku tego przy następnym uruchamianiu skompilowanego programu interpreter zamiast interpretować redundantne linie zastępuje je kodem maszynowym z pamięci podręcznej dzięki czemu program wykonuje się szybciej.

96. Kolekcje w Javie

Tablica

Dla przekory pierwszą omawianą strukturą będzie tablica. Dla przekory ponieważ tablice nie są stricte kolekcjami (w tym sensie, że nie implementują interfejsu Collection), a elementem języka Java. Jako jedyne mogą przechowywać prymitywy (we wszystkich innych przypadkach ma miejsce autoboxing — automatyczna zamiana prymitywu na odpowiadającą mu reprezentację obiektową — jeśli spróbujemy dodać prymityw).

Tablica to uporządkowany (mający kolejność, ale niekoniecznie posortowany!) zbiór elementów, które fizycznie w pamięci komputera są umieszczone obok siebie. Umożliwiają dostęp losowy (tzw. Random Access), czyli dostęp do elementu na dowolnie wybranej pozycji. Elementy tablicy mogą mieć wartość null! Aby utworzyć tablicę stosujemy poniższą konstrukcję:

```
String[] tablica = new String[6];
```

Dozwolone jest także umieszczenie nawiasów kwadratowych za nazwą zmiennej:

```
String tablica[] = new String[6];
```

Jak pewnie zauważyłaś, nie używamy tutaj nawiasów () jak w przypadku konstruktorów obiektów.

Inna dopuszczalna składnia, która od razu inicjuje tablicę wartościami jest poniższa:


```
String[] tablica = {"jeden", "dwa", "trzy", "cztery", "pięć", "sześć"};
```

Istnieje jeszcze jedna możliwość zadeklarowania tablicy — jako **ostatni** argument metody, dzięki czemu może one przyjmować nieokreśloną liczbę argumentów. Weźmy na przykład poniższą metodę:

```
public void wypiszStringiWNowychLiniach(String... tablica) { //argument ten jest tablicą for (i = 0; i<tablica.length; i++) { System.out.println(tablica[i]); } } //przykładowe wywołania wypiszStringiWNowychLiniach("jeden", "dwa", "trzy", "cztery", "pięć"); String[] tablica = {"jeden", "dwa", "trzy", "cztery", "pięć"}; wypiszStringiWNowychLiniach(tablica);
```

W takim przypadku metoda może przyjmować zarówno tablicę Stringów jak i pojedynczego Stringa albo nawet można ją wywołać bez żadnych argumentów.

W ten sposób nie wypiszę się listy Stringów, chyba, że `String.valueOf()`, ale wtedy to będzie traktowane jako pojedynczy String z klamrami (jako tablica)

Lista (java.util.List)

Listy są jednymi z najczęściej używanych kolekcji — koncepcyjnie są podobne do tablic, można jednak (tak jak w przypadku wszystkich pozostałych kolekcji) dynamicznie zmieniać ich rozmiar (dodawać i usuwać elementy). Elementy w liście mogą się powtarzać (tzn. ten sam obiekt może być na liście wielokrotnie). Istnieje kilka standardowych implementacji, z których najważniejsze zostały opisane poniżej:

Klasa	Cechy	Zastosowanie
ArrayList	<ul style="list-style-type: none">w 'tł' przechowuje dane w tablicy, której samodzielnie zmienia rozmiar wg potrzebdostęp do dowolnego elementu w czasie $O(1)$dodanie elementu $O(1)$, pesymistyczny przypadek $O(n)$usunięcie elementu $O(n)$	Najczęstszy wybór z racji najbardziej uniwersalnego zastosowania. Inne implementacje mają przewagę tylko w bardzo specyficznych przypadkach. Jeśli nie wiesz, jakiej listy potrzebujesz, wybierz tą.
	<ul style="list-style-type: none">przechowuje elementy jako lista powiązanych ze sobą obiektów (tj. pierwszy element wie, gdzie jest drugi, drugi wie, gdzie trzeci itd)	LinkedList ma przewagę w przypadku dodawania elementów pojedynczo, w dużej ilości, w sposób trudny do przewidzenia wcześniej, kiedy przejmujemy się ilością zajmowanej pamięci. W

Klasa	Cechy	Zastosowanie
	<ul style="list-style-type: none"> dostęp do dowolnego elementu $O(n)$ (iteracyjnie $O(1)$) dodanie elementu $O(1)$ usunięcie elementu $O(1)$ Istnieje od początku Javy, z założenia miała to być obiektowa reprezentacja tablicy Funkcjonalność i cechy analogiczne do <code>ArrayList</code>, ale znacznie mniej wyrafinowane 	<p>praktyce nie spotkałem się z sytuacją, w której <code>LinkedList</code> byłoby wydajniejsze od <code>ArrayList</code></p> <p>API oficjalnie zaleca korzystanie z klasy <code>ArrayList</code> zamiast <code>Vector</code></p>
<code>Vector</code>		

Implementacją interfejsu `List` jest także klasa `Stack` — będziemy jednak omawiać stos jako osobną strukturę i tam omówimy tą klasę.

Set / zbiór (`java.util.Set`)

Sety to zbiory elementów, w których nie możemy uzyskać dostępu do n 'tego elementu, a jedynie możemy pobrać iterator (obiekt, który pozwoli nam pobrać kolejne elementy, w zależności od implementacji w porządku posortowanym lub losowym). Elementy w Setach nie mogą się powtarzać.

Klasa	Cechy	Zastosowanie
HashSet	<ul style="list-style-type: none"> zbiór nieposortowany kolejność iteracji nieokreślona, może się zmieniać dodanie elementu oraz sprawdzenie czy istnieje ma złożoność $O(1)$ pobranie kolejnego elementu ma złożoność $O(n/h)$, gdzie h to parametr wewnętrzny (pesymistyczny przypadek: $O(n)$) 	Sytuacje, kiedy nie potrzebujemy dostępu do konkretnego elementu, ale potrzebujemy często sprawdzać, czy dany element już istnieje w kolekcji (czyli chcemy zbudować zbiór unikalnych wartości). W praktyce często znajduje zastosowanie do raportowania/zliczeń oraz jako 'przechowalnia' obiektów do przetworzenia (jeśli ich kolejność nie ma znaczenia).
Linked-HashSet	<ul style="list-style-type: none"> analogiczne, jak HashSet (dziedziczy po nim) kolejność elementów używając iteratora jest deterministyczna i powtarzalna (zawsze będziemy przechodzić przez elementy w tej samej kolejności) pobranie kolejnego elementu ma złożoność $O(1)$ 	Jesto to mniej znana i rzadziej stosowana implementacja, często może ona zastąpić HashSet poza specyficznymi przypadkami. W praktyce do iterowania w określonej kolejności często używane sa inne kolekcje (Listy, kolejki)
TreeSet	<ul style="list-style-type: none"> Przechowuje elementy posortowane wg porządku naturalnego (jeśli implementują one interfejs Comparable, w przeciwnym wypadku porządek jest nieokreślony) Wszystkie operacje (dodanie, sprawdzenie czy istnieje oraz pobranie kolejnego elementu) mają złożoność $O(\log n)$ 	Jeśli potrzebujemy, aby nasz zbiór był posortowany bez dodatkowych operacji (sortowanie następuje już w momencie dodania elementu) oraz iterować po posortowanej kolekcji.

Aby utworzyć nowy set (do przykładów bedziemy korzystać z HashSet) wystarczy wywołać konstruktor:

```
Set<String> set = new HashSet<String>();
```

Do dodawania elementu służy metoda add:

```
set.add("jeden");
```

Pobieranie elementów odbywa się z użyciem iteratora, np w pętli for:

```
for (String string : set) {
    ...
}
```

Stan zbioru możemy sprawdzić korzystając z pomocniczych metod:

```
int iloscElementow = set.size();
boolean zbiorJestPusty = set.isEmpty();
```

Map (java.util.Map)

Mapy nie implementują wprawdzie interfejsu Collection, ale są także częścią Java Collections API. Przechowują one mapowania pomiędzy kluczami, a wartościami. Zarówno dodanie elementu, jak i jego pobranie wymaga podania klucza. Kluczem może być dowolny obiekt. Elementy w Mapach mogą się powtarzać, pod warunkiem, że są one wartościami. Klucze nie mogą się powtarzać. Mapy mają wiele wspólnego z omawianymi wyżej zbiorami (można o nich myśleć jak o zbiorach powiązań pomiędzy kluczem i wartością), przez co jak pewnie zauważysz dzielą wiele cech wspólnych.

Klasa	Cechy	Zastosowanie
HashMap	<ul style="list-style-type: none"> mapa nieposortowana kolejność iteracji nieokreślona, może się zmieniać dodanie elementu oraz sprawdzenie czy klucz istnieje ma złożoność $O(1)$ pobranie kolejnego elementu ma złożoność $O(h/n)$, gdzie h to parametr wewnętrzny 	Ogólny przypadek, tworzenie lokalnej pamięci podręcznej czy 'słownika' o ograniczonym rozmiarze, zliczanie wg klucza.
Linked-HashMap	<ul style="list-style-type: none"> analogiczne, jak HashMap (dziedziczy po niej) kolejność kluczy używając iteratora jest deterministyczna i powtarzalna (zawsze będziemy przechodzić przez klucze w tej samej kolejności) pobranie kolejnego elementu ma złożoność $O(1)$ 	Podobnie jak LinkedHashSet, rzadziej znana i stosowana. Przydatna, jeśli potrzebujemy iterować po kluczach w założonej kolejności.
TreeMap	<ul style="list-style-type: none"> Przechowuje elementy posortowane wg porządku naturalnego kluczy (jeśli implementują one interfejs Comparable, w przeciwnym wypadku porządek jest nieokreślony) Wszystkie operacje (dodanie, sprawdzenie czy klucz istnieje oraz pobranie kolejnego elementu) mają złożoność $O(\log n)$ 	Jeśli potrzebujemy, aby nasz zbiór był posortowany wg kluczy bez dodatkowych operacji (sortowanie następuje już w momencie dodania elementu) oraz iterować po posortowanej kolekcji.
Hashtable	<ul style="list-style-type: none"> Historyczna klasa, która w Javie 1.2 została włączona do Java Collection API 	Oficjalnie zaleca się korzystanie z HashSet w większości wypadków zamiast Hashtable

Aby utworzyć nową mapę (do przykładów będziemy korzystać z HashMap) wystarczy wywołać konstruktor:

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

Do dodawania elementu służy metoda put:

```
map.put(1, "jeden");
```

Pobieranie elementów odbywa się z użyciem metody get, podając klucz jako argument:

```
String string = map.get(1)
```

Stan mapy możemy sprawdzić korzystając z pomocniczych metod:

```
int iloscElementow = map.size();  
boolean mapaJestPusta = map.isEmpty();  
boolean zawieraKlucz = map.containsKey(1);  
boolean zawieraWartosc = map.containsValue("jeden");
```

Kolejki (java.util.Queue)

Kolejki dzielą się na dwa typy — LIFO (last-in, first-out) oraz FIFO (first-in, first-out).

Przykładem kolejki LIFO jest stos, mówiąc o kolejce często mamy na myśli kolejkę FIFO. Co ciekawe, w języku Java klasa Stack implementuje interfejs listy, a nie kolejki. Drugą ciekawostką jest fakt, że klasa LinkedList także implementuje interfejs Queue (ponieważ była ona już omówiona wcześniej, nie będziemy powtarzać).

W kolejkach wyróżniamy głowę oraz ogon — standardowo interfejs Queue pozwala na pracę tylko z głową, interfejs Dequeue (który dziedziczy po Queue) pozwala na pracę zarówno z głową jak i ogonem.

Ideą kolejek jest przechowywanie 'kolejki' obiektów do przetworzenia w określonej kolejności, stanowią pewnego rodzaju bufor obiektów.

Klasa	Cechy	Zastosowanie
ArrayDeque	<ul style="list-style-type: none"> Przechowuje elementy w tablicy Ma nieograniczoną pojemność (automatycznie rozszerza tablicę w pamięci) Dodanie i podejrzenie elementu mają złożoność $O(1)$ Pobranie elementu ma złożoność $O(n)$ 	Uniwersalna implementacja pasująca do większości przypadków, działa na zasadzie FIFO, możliwość pracy także z ogonem powoduje że sprawdzi się także jako kolejka LIFO
PriorityQueue	<ul style="list-style-type: none"> Nie umożliwia przechowywania wartości null Nie jest stricte kolejką FIFO, ponieważ kolejność elementów zależy od 'priorytetu' — w najprostszym przypadku elementy są po prostu sortowane 	Używana najczęściej jako kolejka zadań, które mają określony priorytet, tzn. niektóre są ważniejsze od innych.
Stack	<ul style="list-style-type: none"> Kolejka typu LIFO (stos) API niezgodne z interfejsem Queue! implementuje interfejs List, dziedziczy po Vector 	Implementacja ta była od początku Javy, przed powstaniem Collections API. Obecnie raczej zalecane jest korzystanie np. z ArrayDeque

Poniższy opis obejmuje jedynie interfejs Queue, którego klasa Stack nie implementuje. Aby z niej skorzystać, zapoznaj się z dokumentacją API.

Aby utworzyć nową kolejkę (do przykładów będziemy używać ArrayDeque) wystarczy wywołać konstruktor:

```
Queue<String> kolejka = new ArrayDeque<String>();
```

Do dodawania elementu służą metody add oraz offer:

```
kolejka.add("jeden"); //jeżeli kolejka jest pełna, rzuci wyjątek
kolejka.offer("jeden"); //nie rzuci wyjątku, a jedynie zwróci false
```

Pobieranie elementów odbywa się z użyciem metod remove i poll:

```
String string = kolejka.remove(); //jeżeli kolejka jest pusta, rzuci wyjątek
String string = kolejka.poll(); //jeżeli kolejka jest pusta, zwróci null
```

Kolejka pozwala nam także 'podejrzyć' kolejny element nie usuwając go z kolejki, służą do tego metody element oraz peek:

```
String string = kolejka.element(); //jeżeli kolejka jest pusta, rzuci wyjątek
String string = kolejka.peek(); //jeżeli kolejka jest pusta, zwróci null
```

Stan kolejki możemy sprawdzić korzystając z pomocniczych metod:

```
int iloscElementow = kolejka.size();
boolean kolejkaJestPusta = kolejka.isEmpty();
```

97. Serializacja

W jednym z poprzednich artykułów przeczytałeś o strumieniach danych, które pozwalały na zapisywanie oraz odczytywanie danych. Poznałeś wówczas między innymi klasy `DataInputStream` oraz `DataOutputStream`. Klasy te pomagają zapisywać typy proste i łańcuchy znaków.

Serializacja to wbudowany mechanizm zapisywania obiektów, który pozwala na binarny zapis całego drzewa obiektów. Oznacza to tyle, że jeśli mamy obiekt X, który posiada referencję do obiektu Y to serializując X również Y zostanie automatycznie zapisany w strumieniu wyjściowym.

Tak zapisany obiekt możesz później otworzyć przy kolejnym uruchomieniu programu. Jednak serializacja ma więcej zastosowań.

Dzięki temu mechanizmowi można na przykład przysyłać obiekty przez sieć. Obiekt, który stworzyliśmy na jednym komputerze (wewnątrz pamięci jednej wirtualnej maszyny Java) może być zserializowany, przesłany przez sieć i zdeserializowany na drugim komputerze tworząc nową instancję obiektu (wewnątrz pamięci drugiej wirtualnej maszyny Javy). Na obu tych komputerach wirtualna maszyna Javy musi mieć dostęp do skompilowanej wersji klasy.

Warunki wymagane do serializacji

Chociaż serializacja dostępna jest automatycznie dla większości obiektów z biblioteki standardowej to jeśli chcesz móc serializować instancje klas, które sam napiszesz musisz spełnić kilka warunków.

Interfejs `java.io.Serializable`

Jest to tak zwany interfejs znacznikowy, innymi słowy nie zawiera on żadnej metody. Służy on do pokazania wirtualnej maszynie, że instancje danej klasy implementującej ten interfejs mogą być serializowane. Musisz implementować ten interfejs jeśli chcesz aby twoje klasy były serializowalne. Jeśli będziesz próbował zserializować klasę, która nie implementuje tego interfejsu zostanie rzucony wyjątek typu `NotSerializableException`.

Konstruktor bezparametrowy

Tutaj reguła niestety nie jest trywialna. Pierwsza klasa w hierarchii dziedziczenia, która nie jest serializowalna musi mieć dostępny konstruktor bezparametrowy. Łatwiej to będzie zrozumieć na przykładzie:

```
public class Fruit {}  
public class Apple extends Fruit implements Serializable {}  
public class Tomato implements Serializable {}
```

W przykładzie powyżej klasa Fruit musi mieć konstruktor bezparametrowy abyśmy mogli poprawnie serializować instancje klasy Apple. Natomiast ani Apple, ani Tomato takiego konstruktora już nie wymagają (Tomato dziedziczy po Object, który taki konstruktor posiada).

Dodatkowo istnieje interfejs `java.io.Externalizable` (opiszę go dokładnie kilka akapitów niżej), który również zapewnia, że obiekty go implementujące są serializowalne. Jednak w tym przypadku obiekt taki musi także zapewnić konstruktor bezparametrowy, który jest wywoływany w trakcie deserializacji.

Określić pola, które nie są serializowalne

Ten krok jest opcjonalny, jednak w bardziej zaawansowanych przypadkach niezbędny. Wyobraź sobie, że napisałeś klasę Human, która jako jeden z atrybutów posiada wiek zapisany w minutach od urodzenia. Zapisanie tego pola mogłoby prowadzić do odczytania niepoprawnego stanu (zapisujemy obiekt dzisiaj, odczytujemy jutro, wiek w minutach jest zupełnie inny).

Tutaj dochodzimy do słowa kluczowego `transient`. Otóż słowo to może być stosowane przed atrybutami klasy. Oznacza ono, że dany atrybut nie jest serializowalny i zostanie pominięty przez mechanizm serializacji.

Przykład serializacji obiektu

Proszę zwróć uwagę na fragment kodu poniżej, który pokazuje jak mechanizm serializacji działa w praktyce.

```
try (ObjectOutputStream outputStream = new ObjectOutputStream(new  
FileOutputStream("objects.bin"))) {  
    outputStream.writeObject(Integer.valueOf(1));  
    outputStream.writeObject(Integer.valueOf(2));  
}  
  
try (ObjectInputStream inputStream = new ObjectInputStream(new  
FileInputStream("objects.bin"))) {  
    Integer number = (Integer) inputStream.readObject();  
    System.out.println(number);  
    number = (Integer) inputStream.readObject();  
    System.out.println(number);  
}
```


W pierwszym bloku try-with-resources otwieramy strumień typu `ObjectOutputStream`, na którym następnie wywołujemy metodę `writeObject` zapisując do strumienia dwie liczby.

W kolejnym bloku dzięki instancji `ObjectInputStream` odczytujemy wcześniej zapisane obiekty. Obiekty odczytywane są w takiej samej kolejności w jakiej zostały zapisane, w naszym przypadku na konsoli zostaną wyświetlone liczby 1 a później 2.

Serializacja drzewa obiektów

Wspomniałem już wcześniej, że mechanizm serializacji automatycznie obsługuje drzewa obiektów. W przykładzie poniżej pokazana jest właśnie taka sytuacja. Instancja klasy `Car` posiada atrybuty typów `Engine` oraz `Tyre[]`. Serializując a następnie deserializując instancję tej klasy wszystkie jej atrybuty zostały także zapisane.

```
Tyre[] tyres = new Tyre[] {new Tyre(16), new Tyre(16), new Tyre(16), new Tyre(16)};
Engine engine = new Engine("some model");
Car serializedCar = new Car(engine, tyres);
try (ObjectOutputStream outputStream = new ObjectOutputStream(new
FileOutputStream("object-graph.bin"))) {
    outputStream.writeObject(serializedCar);
}

Car deserializedCar = null;
try (ObjectInputStream inputStream = new ObjectInputStream(new
FileInputStream("object-graph.bin"))) {
    deserializedCar = (Car) inputStream.readObject();
    System.out.println(deserializedCar.getEngine().getModel());
    System.out.println(deserializedCar.getTyres().length);
}

System.out.println(serializedCar == deserializedCar);
```

Zwróć proszę uwagę na ostatnią linię. W linijce tej porównywane są dwa adresy instancji klasy `Car` (pamiętasz różnicę między `==` a `equals`?). Oczywiście linijka ta wyświetli `false` na konsoli co dowodzi, że w procesie deserializacji został stworzony zupełnie nowy obiekt klasy `Engine`.

Deserializacja atrybutów transient

Zaraz, jak to? Przecież kilka akapitów wyżej napisałem, że atrybuty poprzedzone słowem kluczowym `transient` nie są serializowane. Tak to prawda, jednak podczas deserializacji atrybuty tego typu należy zainicjalizować pewną wartością. Otóż dla każdego typu mamy taką domyślną wartość:

- boolean – false,
- liczby całkowite (int, long, itd.) – 0,
- liczby ułamkowe (float, double) – 0.0,
- obiekty (Integer, Float, String, CustomClass, itd.) – null

```
public class Human implements Serializable {
    private static final long serialVersionUID = 1L;

    private transient Integer age;
    private String name;

    public Human(String name, Integer age) {
        this.age = age;
        this.name = name;
    }

    public Integer getAge() {
        return age;
    }

    public String getName() {
        return name;
    }

    public static void main(String[] args) throws IOException,
    ClassNotFoundException {
        Human human = new Human("Krzysiek", 21);

        try (ObjectOutputStream output = new ObjectOutputStream(new
        FileOutputStream("human.bin"))) {
            output.writeObject(human);
        }

        try (ObjectInputStream input = new ObjectInputStream(new
        FileInputStream("human.bin"))) {
            Human readHuman = (Human) input.readObject();
            System.out.println(readHuman.getName());
            System.out.println(readHuman.getAge());
        }
    }
}
```

W przykładzie powyżej po deserializacji pole age będzie miało wartość null ponieważ jest to wartość domyślna dla atrybutów poprzedzonych słowem kluczowym transient, które są obiektami.

Pola statyczne a serializacja

Serializacja dotyczy instancji klasy, nie samej klasy. Zatem jeśli zmodyfikowałeś pole statyczne a następnie zdeserializowałeś taki obiekt wprowadzone zmiany zostaną pominięte. Proszę spojrz na przykład poniżej.

```
StaticSerialization object = new StaticSerialization();
object.someField = 200;
System.out.println(object.someField);

try (ObjectOutputStream output = new ObjectOutputStream(new
FileOutputStream("static.bin"))) {
    output.writeObject(object);
}
```

W przykładzie tym modyfikujemy wartość pola statycznego someField a następnie serializujemy instancję klasy do pliku.

```
try (ObjectInputStream input = new ObjectInputStream(new
FileInputStream("static.bin"))) {
    StaticSerialization otherObject = (StaticSerialization) input.readObject();
    System.out.println(otherObject.someField);
}
```

W drugim uruchomieniu programu (w którym nie zmodyfikowaliśmy wartości atrybutu statycznego someField) deserializujemy ten sam plik. W tym przypadku otrzymamy wartość 100 a nie 200, które miał obiekt zapisywany do pliku.

To co trzeba zapamiętać to to, że pola statyczne nie są serializowane a są pobierane z aktualnej definicji klasy (nie z klasy z momentu serializacji).

Możemy powiedzieć, że atrybuty static są też domyślnie transient. Jak zatem takie zmiany odzwierciedlić podczas deserializacji? Jest na to sposób :)

Specjalna obsługa serializacji/deserializacji

W specyficznych przypadkach masz możliwość zmodyfikowania domyślnego zachowania mechanizmu serializacji. Możesz to zrobić jeśli zaimplementujesz poniższe metody.

```
private void readObject(java.io.ObjectInputStream stream) throws IOException,
ClassNotFoundException
private void writeObject(java.io.ObjectOutputStream stream) throws IOException
```

Metoda `readObject(java.io.ObjectInputStream stream)`, którą zaimplementujesz jest automatycznie wywoływana w momencie odczytywania obiektu ze strumienia, czyli w trakcie wywołania metody `ObjectInputStream.readObject()`.

Mechanizm ten wygląda podobnie w przypadku zapisu obiektu.

Metoda `writeObject(java.io.ObjectOutputStream stream)`, którą zaimplementujesz jest automatycznie wywoływana w momencie zapisywania obiektu do strumienia, czyli w trakcie wywołania metody `ObjectOutputStream.writeObject()`.

Poniższy przykład powinien Ci pomóc w zrozumieniu tego mechanizmu:

```
public class CustomSerialization implements Serializable {
    private static final long serialVersionUID = 1L;

    private transient int someField;
    private String otherField;

    public CustomSerialization(int someField, String otherField) {
        this.someField = someField;
        this.otherField = otherField;
    }

    public static void main(String[] args) throws IOException,
    ClassNotFoundException {
        CustomSerialization writtenObject = new CustomSerialization(10,
        "something");

        try (ObjectOutputStream outputStream = new ObjectOutputStream(new
        FileOutputStream("custom-serialization.bin"))) {
            outputStream.writeObject(writtenObject);
        }

        try (ObjectInputStream inputStream = new ObjectInputStream(new
        FileInputStream("custom-serialization.bin" ))) {
            CustomSerialization readObject = (CustomSerialization)
        inputStream.readObject();
            System.out.println(readObject.someField);
            System.out.println(readObject.otherField);
        }
    }

    private void readObject(ObjectInputStream stream) throws IOException,
    ClassNotFoundException {
        stream.defaultReadObject();
        someField = stream.readInt();
    }

    private void writeObject(ObjectOutputStream stream) throws IOException {
        stream.defaultWriteObject();
        stream.writeInt(someField + 1000);
    }
}
```

Jak widzisz obie metody są tu zaimplementowane. `writeObject` jako argument dostaje strumień, do którego powinniśmy zapisać nasz obiekt. Metoda `readObject` jako jedyny argument przyjmuje strumień, z którego powinniśmy odczytać stan obiektu.

Warto tutaj zwrócić uwagę na to, że klasa `ObjectInputStream` posiada metodę `defaultReadObject`, która przeprowadza standardową deserializację, którą możesz rozszerzyć. Podobnie wygląda to w przypadku klasy `ObjectOutputStream` i metody `defaultWriteObject`. Metody te mogą być wywołane wyłącznie w trakcie (de)serializacji obiektu. Zajmują się one (de)serializacją atrybutów klasy, które nie są oznaczone jako `static` lub `transient`.

Serializacja a dziedziczenie

W poprzednich przykładach użyliśmy klasy `Engine`, która implementuje interfejs `Serializable`. Załóżmy, że utworzyliśmy klasę `DieselEngine`, która dziedziczy po `Engine`. Automatycznie instancje klasy `DieselEngine` będą implementowały interfejs `Serializable` (dziedzicząc go z `Engine`). Co powinniśmy zrobić jeśli nie chcielibyśmy aby nasz `DieselEngine` był serializowalny? Należy użyć wspomnianego już wyjątku `NotSerializableException` jak w przykładzie poniżej:

```
public class DieselEngine extends Engine {
    public DieselEngine() {
        super("diesel");
    }

    private void writeObject(ObjectOutputStream out) throws IOException {
        throw new NotSerializableException("DieselEngine isn't serializable!");
    }

    private void readObject(ObjectInputStream in) throws IOException {
        throw new NotSerializableException("DieselEngine isn't serializable!");
    }
}
```

Pełny wpływ na mechanizm serializacji

Istnieje jeszcze jeden, dużo mniej popularny sposób zapewnienia iż obiekt może być serializowany. Jest nim interfejs `Externalizable`. W tym przypadku interfejs ten zawiera dwie metody, które musimy zaimplementować. Dodatkowo takie klasy muszą mieć konstruktor bezparametrowy, reszta pozostaje bez zmian. W przypadku tego podejścia cały protokół serializacji, kolejność zapisanych pól, format etc. leży po naszej stronie. Poniżej prosty przykład, w którym używam właśnie takiego podejścia.

W tym przypadku do utworzenia obiektu mechanizm serializacji używa standardowego konstruktora bezparametrowego. Po czym wywołuje na tej instancji metodę `readExternal`.

```

public class CustomProtocolSerialization implements Externalizable {
    private String field;

    public CustomProtocolSerialization() {
    }

    public CustomProtocolSerialization(String field) {
        this.field = field;
    }

    public static void main(String[] args) throws IOException,
ClassNotFoundException {
        CustomProtocolSerialization object = new
CustomProtocolSerialization("field value");

        try (ObjectOutputStream output = new ObjectOutputStream(new
FileOutputStream("externalizable.bin"))) {
            output.writeObject(object);
        }

        try (ObjectInputStream input = new ObjectInputStream(new
FileInputStream("externalizable.bin"))) {
            CustomProtocolSerialization readObject = (CustomProtocolSerialization)
input.readObject();
            System.out.println(readObject.field);
        }
    }

    public void writeExternal(ObjectOutput out) throws IOException {
        out.writeUTF(field);
    }

    public void readExternal(ObjectInput in) throws IOException,
ClassNotFoundException {
        field = in.readUTF();
    }
}

```

Pole serialVersionUID

Dodatkowo musisz wiedzieć o statycznym polu w klasie o nazwie serialVersionUID. Jego pełna definicja wygląda następująco:

```
private static long serialVersionUID;
```

Pole to ma specyficzne zastosowanie. Mechanizm serializacji używa go do upewnienia się, że deserializowany obiekt „pasuje” do danych zapisanych w strumieniu. Wie o tym na podstawie wartości tego pola. Jeśli w zdeserializowanym obiekcie wartość tego pola jest taka sama jak aktualnej definicji klasy wówczas można bezpiecznie przeprowadzić deserializację.

Kiedy taka sytuacja może wystąpić? Załóżmy, że dzisiaj napiszesz klasę Human, zdeserializujesz jej instancję i zapiszesz w pliku na dysku. Po jakimś czasie wprowadzisz zmiany w klasie i będziesz chciał odczytać starą wersję z pliku. W niektórych przypadkach taka operacja nie będzie dozwolona. Właśnie wtedy pole serialVersionUID może pomóc w wykryciu takiej sytuacji.

Pole to możesz ustawić samodzielnie, jeśli tego nie zrobisz kompilator wygeneruje tę wartość za Ciebie na podstawie definicji klasy.

98. Object

Object class is present in **java.lang** package. Every class in Java is directly or indirectly derived from the **Object** class. If a class does not extend any other class then it is a direct child class of **Object** and if extends another class then it is indirectly derived. Therefore the Object class methods are available to all Java classes. Hence Object class acts as a root of inheritance hierarchy in any Java Program.

Using Object Class Methods

There are methods in the **Object** class:

1. toString(): The toString() provides a String representation of an object and is used to convert an object to String. The default toString() method for class Object returns a string consisting of the name of the class of which the object is an instance, the at-sign character '@', and the unsigned hexadecimal representation of the hash code of the object. In other words, it is defined as:

```
// Default behavior of toString() is to print class name, then
// @, then unsigned hexadecimal representation of the hash code
// of the object

public String toString()
{
    return getClass().getName() + "@" + Integer.toHexString(hashCode());
}
```

It is always recommended to override the `toString()` method to get our own String representation of Object. For more on override of `toString()` method refer – Overriding `toString()` in Java

Note: Whenever we try to print any Object reference, then internally `toString()` method is called.

```
Student s = new Student();

// Below two statements are equivalent
System.out.println(s);
System.out.println(s.toString());
```

2. hashCode(): For every object, JVM generates a unique number which is hashCode. It returns distinct integers for distinct objects. A common misconception about this method is that the `hashCode()` method returns the address of the object, which is not correct. It converts the internal address of the object to an integer by using an algorithm. The `hashCode()` method is **native** because in Java it is impossible to find the address of an object, so it uses native languages like C/C++ to find the address of the object.

Use of hashCode() method: It returns a hash value that is used to search objects in a collection. JVM(Java Virtual Machine) uses the hashCode method while saving objects into hashing-related data structures like HashSet, HashMap, Hashtable, etc. The main advantage of saving objects based on hash code is that searching becomes easy.

Note: Override of **hashCode()** method needs to be done such that for every object we generate a unique number. For example, for a Student class, we can return the roll no. of a student from the `hashCode()` method as it is unique.

3. equals(Object obj): It compares the given object to “this” object (the object on which the method is called). It gives a generic way to compare objects for equality. It is recommended to override the **equals(Object obj)** method to get our own equality condition on Objects. For more on override of `equals(Object obj)` method refer – Overriding equals method in Java

Note: It is generally necessary to override the **hashCode()** method whenever this method is overridden, so as to maintain the general contract for the hashCode method, which states that equal

ects must have equal hash codes.

4. getClass(): It returns the class object of “this” object and is used to get the actual runtime class of the object. It can also be used to get metadata of this class. The returned Class object is the object that is locked by static synchronized methods of the represented class. As it is final so we don’t override it.

```
// Java program to demonstrate working of getClass()

public class Test {
    public static void main(String[] args)
    {
        Object obj = new String("GeeksForGeeks");
        Class c = obj.getClass();
        System.out.println("Class of Object obj is : "
                           + c.getName());
    }
}
```

Output:

```
Class of Object obj is : java.lang.String
```

Note: After loading a .class file, JVM will create an object of the type java.lang.Class in the Heap area. We can use this class object to get Class level information. It is widely used in [Reflection](#)

5. finalize() method: This method is called just before an object is garbage collected. It is called the Garbage Collector on an object when the garbage collector determines that there are no more references to the object. We should override finalize() method to dispose of system resources, perform clean-up activities and minimize memory leaks. For example, before destroying Servlet objects web container, always called finalize method to perform clean-up activities of the session.

Note: The finalize method is called just **once** on an object even though that object is eligible for garbage collection multiple times.

```
// Java program to demonstrate working of finalize()

public class Test {
    public static void main(String[] args)
    {
        Test t = new Test();
        System.out.println(t.hashCode());

        t = null;

        // calling garbage collector
        System.gc();

        System.out.println("end");
    }

    @Override protected void finalize()
    {
        System.out.println("finalize method called");
    }
}
```

Output:

```
366712642
finalize method called
end
```

6. clone(): It returns a new object that is exactly the same as this object. For clone() method refer Clone().

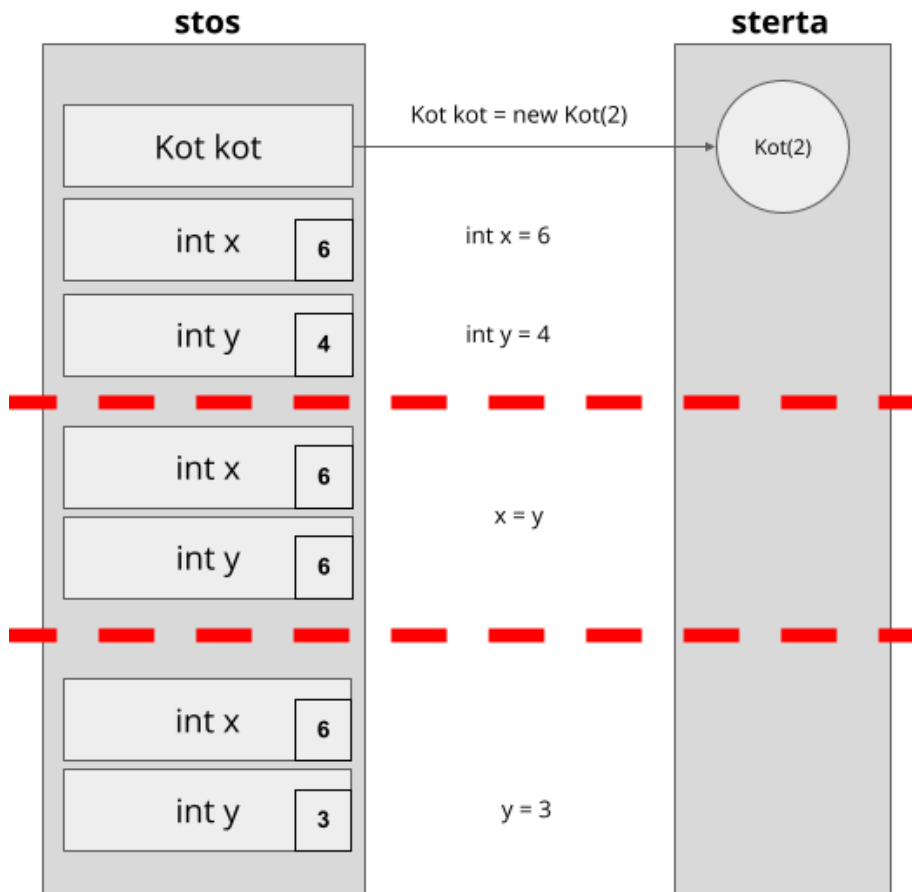
The remaining three methods **wait()**, **notify()** **notifyAll()** are related to Concurrency. Refer to Inter-thread Communication in Java for details.

99. Referencje typów prymitywnych

Zmienne typu obiektowego przechowują **referencje (adresy)** obiektów a zmienne typów prostych (int, double etc.) **wartości**. Przy "przypisywaniu obiektu" do zmiennej przypisywana jest referencja/adres obiektu, który kryje się za zmienną. W przypadku typów prymitywnych to adres jest już tą wartością. Głupio byłoby trzymać informację, że pod adresem 423413, znajduje się liczba 3, lepiej zamiast adresu trzymać po prostu liczbę 3.

```
int x = 6;
int y = 4;

// w przypadku typów prymitywnych zamiast adresu, pod którym znajdziemy dany
// obiekt, przechowujemy już tę konkretną wartość
x = y;
```



100. Referencje, a parametry metod

Częstym tematem, który wprowadza w zakłopotanie jest zachowanie referencji w obrębie parametrów metod. Omówimy sobie zachowanie Javy w tym obszarze.

W sytuacji, kiedy podajemy jakiś parametr do metody, **NIE** przekazujemy jej referencji, a jedynie wartość. Rozjaśnijmy to na przykładzie.

```
public class ReferencjeTest {
    public static void main(String[] args) {
        double n = 2;

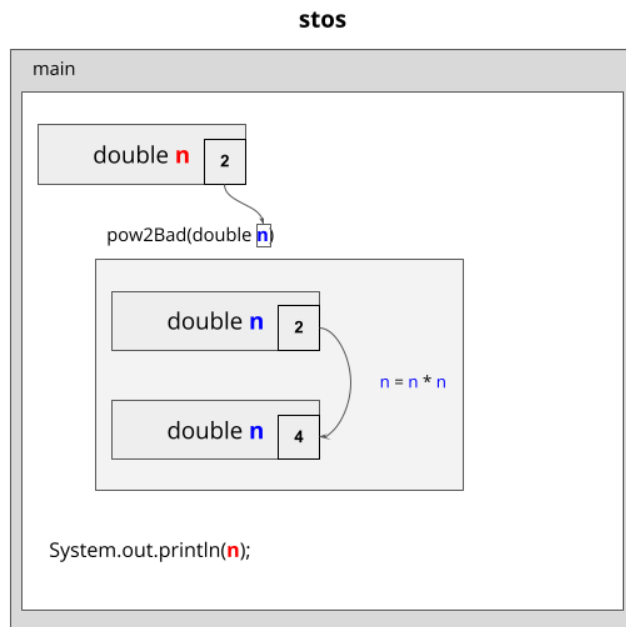
        System.out.println("--- źle ---");
        pow2Bad(n);
        System.out.println(n); // wydrukuje 2
    }

    public static void pow2Bad(double n){
        n = n * n;
    }
}
```

W powyższym programie w metodzie `main` tworzę zmienną `n` o wartości 2. Następnie uruchamiam metodę `pow2Bad`, która powinna zmienić wartość tej zmiennej podnosząc ją do

potęgi 2, czyli spodziewamy się rezultatu 4. Po wydrukowaniu tej zmiennej `System.out.println(n)`, na konsoli zobaczymy jednak 2.

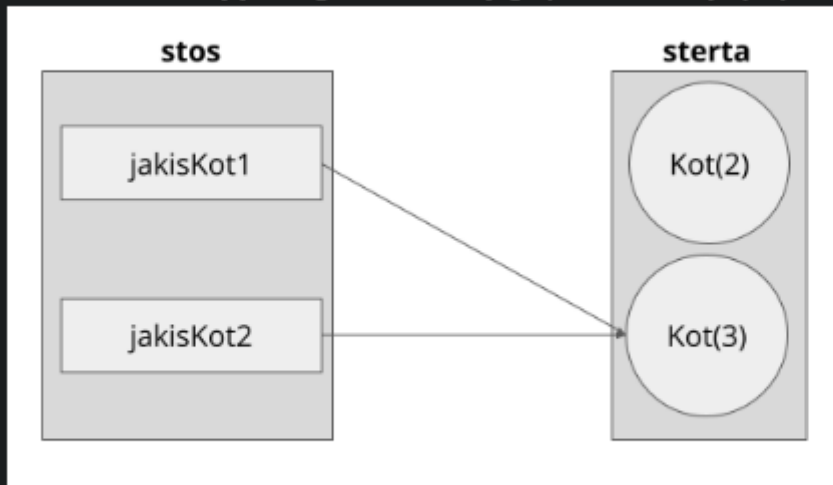
O co chodzi? Widzimy tutaj w praktyce mechanizm "pass by value". Okazuje się, że Java nie przekazuje do metody referencji zmiennej, za to wewnątrz metody tworzona jest nowa zmienna `n` o tej samej nazwie... To brzmi skomplikowanie więc zobaczmy to na obrazku.



101. Equals - jak porównać obiekty

Dowiedzieliśmy się, że *zmienne obiektowe przechowują referencje*. Operator `==` **zawsze porównuje wartość zmiennej**, czyli w przypadku zmiennych obiektowych referencje - czy to jest **dokładnie ten sam obiekt**. Porównując zmienne z użyciem tego operatora nie sprawdzamy, czy *obiekty są takie same (mają taką samą zawartość)* tylko czy te zmienne wskazują na *ten sam obiekt*.

Operator `==` zwróci zatem `true`, jeżeli dwie zmienne wskazują na dokładnie ten sam obiekt, nie zwracając uwagi na wartości jego pól. Musi to być po prostu ten sam obiekt!



W przypadku

zmiennych prymitywnych `int`, `boolean`, `double`, `char`, itp. operator ten porównuje już wartości. Ponieważ referencja jest już wartością.

102. Metoda equals()

Każdy obiekt w Javie posiada metodę `equals(Object obj)` służącą do porównywania go z innym obiektem. Jest to metoda odziedziczona z klasy `Object`.

Właściwości metody `equals`

- metoda przyjmuje parametr typu `Object` czyli **może zostać do niej przekazany obiekt dowolnej klasy**
- metoda zwraca wartość typu `boolean` - `true` jeśli porównywane obiekty są identyczne, `false` - jeśli są różne

Ważne! Domyślnie metoda `equals` działa tak samo jak operator `==` - sprawdza czy to są dokładnie te same obiekty. To od nas zależy, co oznacza, że dwa obiekty naszej klasy są sobie równe, dlatego aby ta metoda poprawnie działała, musimy w naszej klasie ją przesiłnić.

103. hashCode - czyli PESEL obiektu

Podobnie jak to było w przypadku metody `equals`, każdy obiekt w Javie posiada metodę `public int hashCode()` odziedziczoną z klasy `Object`. Służy ona do generowania "hashcode'u" czyli liczby, która powinna być *możliwie* różna dla różnych obiektów. Liczba taka pozwala na znaczne przyspieszenie różnego typu operacji, szczególnie takich jak wyszukiwanie obiektów. Na jej podstawie różne biblioteki (np. `HashSet`, `HashMap`) potrafią stworzyć coś na wzór spisu treści, tak żeby wydajnie wyszukiwać konkretne obiekty.

Metoda hashCode może zostać zatem porównana do peselu. Wyobraź sobie sytuację, że idziesz do urzędu i chcesz załatwić jakąś sprawę. Urzędnik w celu identyfikacji, musiałby zadać Ci szereg pytań:

- Kiedy się urodziłeś?
- Jak się nazywasz?
- Jak się nazywają Twoi rodzice?
- Jakie mieli nazwiska przed ślubem?
- Gdzie się urodziłeś?

Żeby uprościć nam załatwianie spraw urzędowych powstał numer pesel. Dzięki niemu cały taki wywiad można skrócić do pytania "Jaki jest Twój pesel?". W świecie rzeczywistym pesel jest "generowany/wyliczany" przez urząd. W Javie natomiast, każdy obiekt musi sam potrafić wyliczyć swój hashCode.

Jak więc sprawić, żeby każdy obiekt mógł przedstawić się za pomocą liczby? Weźmiemy pod uwagę obiekt pudełko, które ma dwie cechy numeryczne wymiarX i wymiarY.

- możemy mnożyć wymiary tj. $x * y$, szybko jednak okaże się, że pudełko 100 x 50, uzyska taki sam "pesel" jak pudełko 50 x 100
- możemy więc zminimalizować ryzyko takiego konfliktu mnożąc poszczególne elementy przez jakąś liczbę, np. 31, w taki sposób:
 - x zapisz do zmiennej rezultat
 - pomnóż rezultat razy 31 i dodaj do tego y

To jedna z możliwych propozycji i tak mniej więcej wygląda przyjęty sposób liczenia hashCode'ów, który minimalizuje ryzyko konfliktów.

Dla prostej klasy z 2 polami typu int może on wyglądać następująco:

```

public class TesterHashCode {
    public static void main(String[] args){
        Pudelko pudelko1 = new Pudelko(10, 20);
        Pudelko pudelko2 = new Pudelko(5, 20);
        Pudelko pudelko3 = new Pudelko(10, 20);

        System.out.println("HashCode pudelko1: " + pudelko1.hashCode());
        System.out.println("HashCode pudelko2: " + pudelko2.hashCode());
        System.out.println("HashCode pudelko3: " + pudelko3.hashCode());
    }
}

public class Pudelko {
    private int wymiarX;
    private int wymiarY;

    public Pudelko(int wymiarX, int wymiarY) {
        this.wymiarX = wymiarX;
        this.wymiarY = wymiarY;
    }

    @Override
    public int hashCode() {
        int hash = wymiarX;
        hash = 31 * hash + wymiarY;
        return hash;
    }
}

```

Zauważ, że dla dwóch obiektów o tej samej zawartości metoda hashCode zwróciła taką samą wartość.

W przypadku bardziej złożonych obiektów, takich jak String przemnaża i dodaje się reprezentacje liczbowe kolejnych liter. Z kolei w przypadku Twoich klas, np. User możesz przemnażać hashCody swoich składników, czyli:

- hashCode ze String email przemnoż przez 31 i dodaj hashCode ze String username, czyli tak return 31 * email.hashCode() + username.hashCode();

Ta magiczna liczba 31 wynika z jakichś obserwacji poczynionych przez mądrych ludzi, nie znam szczegółów matematycznych, ważne, że się sprawdza.

Objects.hash()

Napisanie optymalnej metody hashCode tak, by maksymalnie zmniejszyć ryzyko konfliktów (różne obiekty o tym samym "hash-code") jest dość skomplikowanym zadaniem wymagającym dokładnej wiedzy na temat użycia danego typu oraz systemu binarnego. Wprowadzono zatem metodę Objects.hash(...) liczącą *wystarczająco dobry* hash code dla podanych pól.

@Override

```

public int hashCode() {

    return Objects.hash(wymiarX, wymiarY);

}

```

104. Hashcode

`hashCode` to metoda, która powinna zawierać algorytm hashujący, który na podstawie danych obiektu wylicza liczbę całkowitą (hash), reprezentujący ten obiekt.

Hash jest wykorzystywany pod spodem przez niektóre kolekcje (np. `HashMap` i `HashSet`) do przechowywania referencji do obiektów, co w rezultacie daje bardzo wydajną kolekcję o stałym czasie dostępu do danych. Ten czas w dużej mierze zależy od wydajności naszego algorytmu hashującego, więc warto na to zwracać uwagę.

W poprzednim poście podkreśliłem, że **niezbędne jest nadpisywanie metody `hashCode` za każdym razem, gdy nadpisujesz `equals`**.

Jest to spowodowane tym, że między nimi zawarty jest nienaruszalny kontrakt. Jeśli zostanie naruszony, to obiekty nie będą działać prawidłowo w kolekcjach takich jak `HashMap` i `HashSet` czy w każdej innej klasie, która polega na *hash* kodach obiektów.

Kontrakt pomiędzy metodami `equals(Object object)` i `hashCode()`:

- Kiedykolwiek metoda `hashCode()` jest wywołana na tym samym obiekcie więcej niż raz, musi za każdym razem zwrócić tę samą wartość (int) `hashCode` niezależnie od metody `equals(Object)`.
- Jeśli dwa obiekty są równe zgodnie z metodą `equals(Object)`, wtedy każde wywołanie metody `hashCode()` dla tych obiektów musi zwrócić tę samą wartość (`hashCode'y` są równe).
- Jeśli dwa obiekty nie są równe zgodnie z metodą `equals(Object)`, nie muszą zwracać różnych `hashcodów`. Mówiąc inaczej obiekty mogą mieć zgodny `hascode` i być nie równe zgodnie z metodą `equals(Object)` (`equals` zwróci `false`). Jednak jest to pożądane, bo ma to znaczenie wydajnościowe dla `hash` tablic (jeśli będą dwa takie same `hashe`, to będzie musiała wywołać metody `equals`, aby rozróżnić obiekty).

Szczególnie ważny jest 2 punkt. - jeśli nie nadpiszemy również metody `hashCode` wraz z `equals` to według metody `equals` dwa obiekty są sobie równe, a dla metody `hashCode` te same obiekty nie mają nic ze sobą wspólnego. Zwracane są dwa różne numery, zamiast dwóch identycznych.

Teraz dla przykładu założymy, że będziemy chcieli użyć `PhoneNumber` z poprzedniego postu jako klucze w `HashMap` bez definiowania `hashCode`:

```
Map<PhoneNumber, String> map = new HashMap<>();
map.put(new PhoneNumber(707, 867, 5309), "Jenny");
```


Wydawałoby się, że wywołując:

```
map.get(new PhoneNumber(707, 867, 5309))
```

Dostaniemy "Jenny", ale tak się nie stanie, bo dostaniemy null. Zauważ, że są używane dwie instancje PhoneNumber. Są one identyczne, ale z punktu widzenia hashCode, na którym bazuje HashMap, nie są.

Napisanie poprawnej implementacji hashCode dla PhoneNumber rozwiązuje problem.

Jak poprawnie napisać metodę hashCode?

Podobnie jak pisałem we wcześniejszym poście, jeśli nie potrzebujesz specyficznej implementacji, to możesz zdać się na automatyczne wygenerowanie tej metody przez IDE czy odpowiednią bibliotekę jak np. Lombok.

Dobra metoda hashCode powinna dawać różne hash kody, dla różnych obiektów. Osiągnięcie tego może być trudne, ale możemy uzyskać przybliżony efekt.

Od Javy 7, mamy klasę Objects, która posiada statyczną metodę hash. Przyjmuje ona dowolną liczbę argumentów i zwraca dla nich *hash code*. Pozwala to napisać implementację metody hashującej w jednej linijce. Dla przykładowej klasy PhoneNumber będzie wyglądać tak:

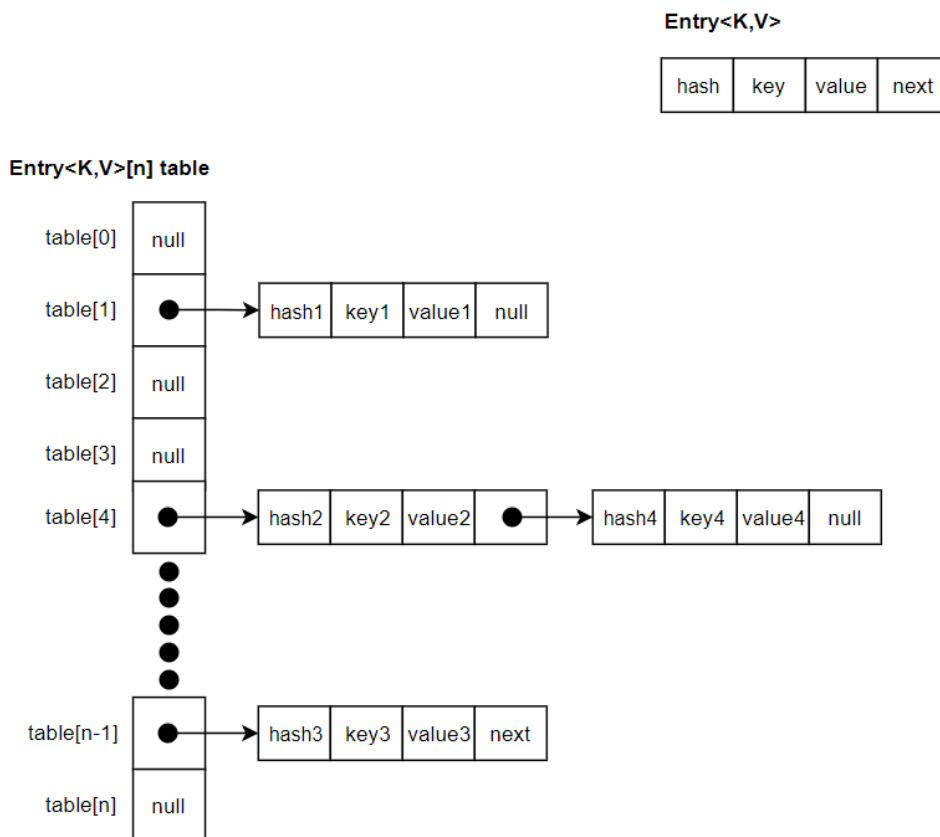
```
// One-line hashCode method - mediocre performance
@Override public int hashCode() {
    return Objects.hash(lineNum, prefix, areaCode);
}
```

Jej jakość jest podobna do tych napisanych ręcznie według przepisu, który zaraz przedstawię, jednak jest delikatnie mniej wydajna, jeśli częstotliwość jej wywołań jest wysoka i używane są wartości prymitywne. Spowodowane jest to tym, że pod spodem używana jest tablica, a prymitywy muszą być autoboxowane. Jednak w większości wypadków, gdy wydajność nie ma dla nas aż takiego znaczenia, możemy śmiało tego używać.

105. Jak działa HashMap

A więc jak działa Hash Mapa w Javie ?

HashMapa swoje działanie opiera na tablicy obiektów klasy statycznej `HashMap.Node`, która implementuje interfejs `Map.ENTRY` oraz mechanizmie haszowania klucza wykorzystywanego do odnajdywania indeksu tablicy. Dzięki temu rozwiązaniu zapewnione jest przechowywanie i wyciąganie obiektów (jeśli znamy klucz) ze złożonością $O(1)$. Wizualizacja HashMap wygląda następująco:



Hash – liczba całkowita, obliczona w prosty sposób w momencie dodawania elementu do mapy za pomocą metody `HashMap.hash(key)`, metoda ta oblicza hash(ang skrót) obiektu i przyporządkowuje mu, krótką deterministyczną wartość posiadającą zawsze stały rozmiar, tzw. skrót nieodwracalny. W tej konkretnej implementacji wykorzystywane jest do tego logiczne przesunięcie bitowe w prawo. Dla osób zainteresowanych szczegółami na koniec artykułu postaram się dokładnie wyjaśnić jak przebiega ten proces.

key – Obiekt (Implementacja pozwala na przechowywanie jednego klucza nullowego) dobrymi kandydatami na klucze na pewno będą obiekty niezmiennie, spełniające kontrakt

hashCode i equal . W skrócie jeśli dwa różne obiekty zwrócą ten sam hashCode to obiekty te mogą być równoznaczne, wtedy equals może zwrócić true lub false. Jeżeli jednak hashCode obydwu obiektów jest różny, wtedy equals zawsze zwróci false. Dobrymi kandydatami na klucze są String, Integer oraz inne Wrappery typów prymitywnych. String jest prawdopodobnie najczęściej używanym typem klucza w mapie, dlatego że jest niezmienny i ma poprawnie zaimplementowane metody equals i hashCode oraz jest czytelny. Niezmiennosc obiektów jest kluczowa aby zapobiec zmianie pól używanych do obliczania hashCode w trakcie trwania programu, ponieważ kody wyliczane są podczas wkładania i pobierania elementów z mapy. Jeśli pomiędzy tymi operacjami pola w kluczu się zmieniają nie będziemy w stanie znaleźć żadanego obiektu, gdyż hashCode wyliczone przed i po będą się różnić.

value – dowolny obiekt lub null

next – referencja na następny node (wyjaśnię to pole w dalszej części artykułu przy omawianiu kolizji)

Tworzenie HashMapy i optymalizacja w różnych wersjach Javy

Na początek warto wspomnieć o tym, iż wielkość wewnętrznej tablicy jest zawsze potęgą dwójki (jeśli w konstruktorze podamy np.. Liczbę 18 wewnętrzna tablica za alokuje pamięć na 32 elementy ponieważ jest to najbliższa potęga dwójki większa od podanej liczby 18), dzięki temu HashMapa jest w stanie zapewnić nam złożoność na poziomie **O(1)**.

Warto też zauważyć iż niektóre zmienne w klasie HashMap ustawiane są defaultowo np.:

DEFAULT_INITIAL_CAPACITY – jest to początkowa wielkość naszej wewnętrznej tablicy ustawione na 16 elementów, co ważne jest to potęga dwójki , jeśli przy tworzeniu mapy podamy wielkość w konstruktorze wartość ta jest pomijana.

DEFAULT_LOAD_FACTOR – współczynnik obciążenia ustawiony na poziomie **0.75** jest to dozwolony procent załadowania elementami naszej mapy.

Int threshold – zmienna wyliczana z dwóch powyższych wartości oznacza próg ilości elementów w mapie po przekroczeniu którego zostanie ona rozszerzona.

We wcześniejszych wersjach Javy przed wersja 1.7 przy tworzeniu pustej mapy bez podania jej wielkości automatycznie pod spodem alokowała się tablica o wielkości 16 elementów

```

/**
 * Constructs an empty <tt>HashMap</tt> with the default initial capacity
 * (16) and the default load factor (0.75).
 */
public HashMap() {
    this.loadFactor = DEFAULT_LOAD_FACTOR;
    threshold = (int) (DEFAULT_INITIAL_CAPACITY * DEFAULT_LOAD_FACTOR);
    table = new Entry[DEFAULT_INITIAL_CAPACITY];
    init();
}

```

W Javie w wersji 1.7 oraz 1.8 zostało to zoptymalizowane aby ją przyspieszyć i zredukować zajmowaną pamięć, wprowadzono tzw. lazy initialization, oznacza to, że przy tworzeniu mapy nie alokujemy tak jak wcześniej 16 elementowej tablicy, zamiast tego tablica tworzona jest dopiero przy wykonaniu metody Put . Widocznie zrozumiano że często mapy są tymczasowe i nie potrzebują tak wielkiej wewnętrznej struktury inicjalizowanej od razu po stworzeniu, która marnuje pamięć.

```

/**
 * Constructs an empty <tt>HashMap</tt> with the default initial capacity
 * (16) and the default load factor (0.75).
 */
public HashMap() {
    this.loadFactor = DEFAULT_LOAD_FACTOR; // all other fields defaulted
}

```

Powyższy kod dobrze obrazuje czemu komentarze w kodzie są słabe. Zmieniając implementację HashMapy zmieniono działanie konstruktora ale nie zaktualizowano komentarza.

Dodawanie elementów do mapy metoda put(key,value)

Dodawanie elementów do mapy odbywa się za pomocą metody put, przyjmuje ona dwa parametry klucz i wartość. Podczas wykonywania tej operacji, jeśli wersja używanej javy jest wyższa niż 1.6 sprawdzany jest aktualny rozmiar tablicy. Jeśli nie została ona stworzona wcześniej, wówczas wykorzystywany zostaje Lazy Loading, o którym wspominałem kilka akapitów wyżej. Następstwem tego jest stworzenie tablicy o defaultowym rozmiarze i obliczenie rozmiaru threshold. Po sprawdzeniu i ewentualnym stworzeniu tablicy, obliczony zostaje Hash podanego klucza wykorzystany do znalezienia miejsca w tablicy aby zapisać obiekt Entry zawierający hash, klucz i wartość.

Wyliczanie indexu w tablicy

Jak już wspominałem wcześniej HashMapa dba o to aby rozmiar wewnętrznej tablicy był potęgą dwójki. Dzieje się tak, ponieważ wielkość ta wykorzystywana jest wraz z wartością hash do wyznaczania indexu dla dodawanych obiektów Entry. Aby indeks nie wychodził poza zakres tablicy nie możemy użyć bezpośrednio wartości Hash, zamiast tego wykonywane jest dzielenie modulo, którego wynik zawsze będzie mieścić się w podanym zakresie **n(rozmiar tablicy)** wzór **tab[i=hash % n]**. Jednak taki zabieg jest zasobożerny, dlatego wymuszono rozmiar tablicy odpowiadający wielkości potęgi dwójki, aby można było zastosować szybsze dzielenie modulo binarne. Jest to bardziej optymalne rozwiązanie. Wzór wygląda następująco **tab[i=(n-1)&hash]**.

```
V get(Object key)
    hash(key)

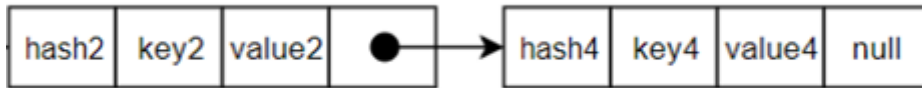
    index = hash & (n-1)
```

Dzięki takiemu zabiegowi index wyliczany jest szybko i nie wychodzi poza rozmiar tablicy a także wstawianie i wyciąganie elementów odbywa się ze złożonością **O(1)**. Nie jest to jednak idealne rozwiązanie i czasami powoduje kolizje.

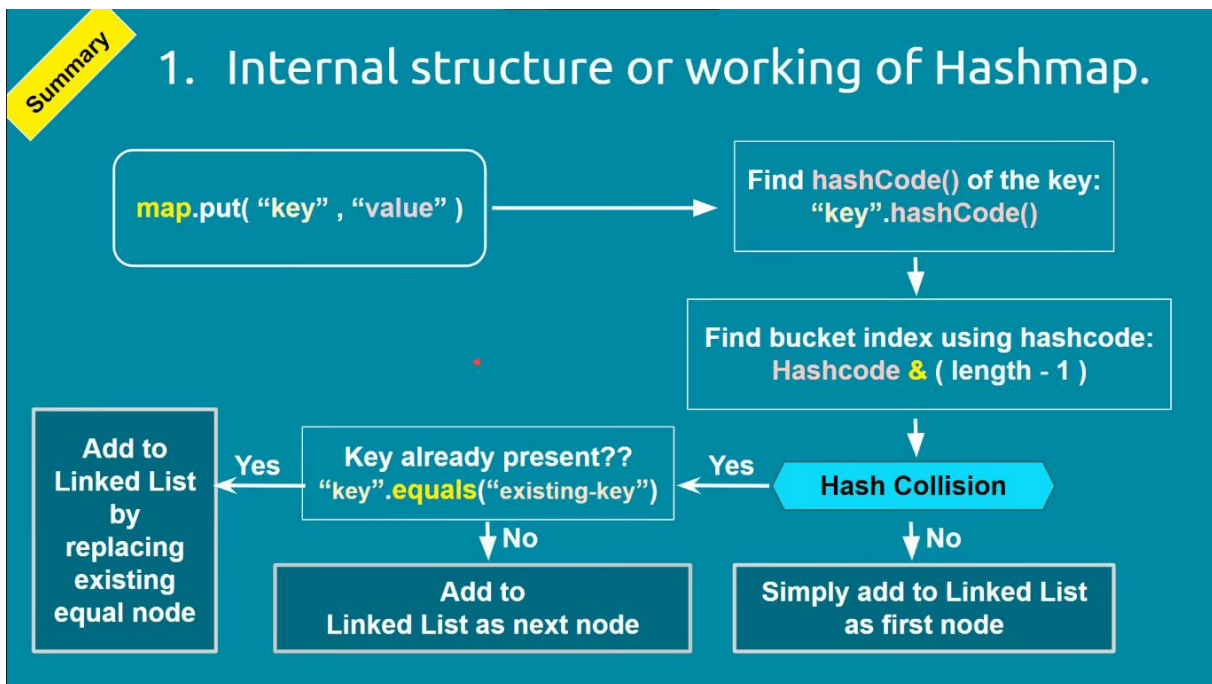
Kolizje obiektów o tej samej wartości Hash

Wyliczanie indeksów opierając się na metodzie hashCode danego klucza nie jest idealnym rozwiązaniem, czasami powoduje kolizje kluczy . Dzieje się tak gdy hash dodawanego obiektu jest taki sam jak hash już istniejącego obiektu w tablicy. Prowadzi to do kolizji, która w zależności od sytuacji może być obsłużona na 2 sposoby:

1. W tablicy przechowywany jest obiekt Entry zawierający hash, **klucz** i wartość. Jeśli **klucz** przekazany do metody put jest równy **kluczowi** przechowywanemu w obiekcie Entry, pod indexem wyliczonym za pomocą metody hash, wtedy wiemy, że klucz jest taki sam i wartość zostaje podmieniana na nową.
2. Jeżeli jednak klucze mają taki sam Hash ale metoda equals zwraca false, oznacza to iż te dwa klucze różnią się od siebie co za tym idzie wstawiany obiekt musi znaleźć się w podanym indeksie w tablicy, bez nadpisywania już istniejącego. Z pomocą przychodzi zmienna **next**, gdy wystąpi **kolizja kluczy**, wewnątrz tablicy o wyliczonym indeksie obiekt Entry traktowany jest jako lista, a nowy obiekt dodawany jest w miejsce **next** !



Jako że jest to lista obiektów, w najgorszym przypadku złożoność wyszukiwania wynosi $O(n)$. Zostało to zoptymalizowane w wersji javy **1.8**. Kiedy rozmiar listy przekroczy określony **threshold**, dokonywana jest zamiana listy na drzewo binarne, dzięki czemu wyszukiwanie w wyżej podanym przypadku ma złożoność na poziomie **$O(\log N)$** . **Threshold** – określa ilość elementów po której przekroczeniu lista zostaje przetransformowana w drzewo binarne, jest ustawiony w stałej o nazwie. **TREEIFY_THRESHOLD** aktualnie wartość ta wynosi **8**, co oznacza że po przekroczeniu tej ilości elementów zamiast listy do przechowywania elementów zostanie użyte zrównoważone drzewo binarne o złożoności **$O(\log N)$** .



Wyciąganie obiektów z Mapy metoda get(key)

W przeciwieństwie do metody put(klucz , wartość) metoda get jest dużo prostsza i łatwiejsza w wyjaśnieniu. W momencie wywołania tej metody na obiekcie kluczu wykonana zostaje metoda hashCode(), która jak sama nazwa wskazuje zwraca HashCode obiektu. Index tablicy obliczany jest za pomocą binarnego dzielenia modulo tak jak w przypadku metody put, dlatego obliczony hashCode klucza przekazywany jest do funkcji HashMap.hash. Ponieważ Hashcode dla różnych obiektów może zwracać tą samą wartość funkcja hash zapewnia większą unikalność poprzez zastosowanie przesunięcia bitowego w prawo na

wartości hashcode (funkcja hash dodatkowo ignoruje znak liczby ujemnej i traktuje ją jak dodatnią) , aczkolwiek nie eliminuje to kolizji w 100 procentach. Wynik przeprowadzonych działań wskazuje na indeks tablicy pod którym należy szukać wartości dla podanego klucza.

Teraz nie pozostaje nic innego jak sięgnąć pod podany indeks tablicy i spróbować wyciągnąć wartość powiązaną z przekazywanym kluczem. O ile w tablicy nie jest przechowywana lista/drzewo obiektów, metoda kończy swoje działanie i zwraca wartość dla podanego klucza.

Aktualizacja: przy wyciąganiu elementu sprawdzana jest referencja klucza w wyliczonym indeksie i referencja przekazywanego klucza gdy się nie zgadzają wykonywana jest metoda equals żeby sprawdzić czy to te same klucze.

Ale co w przypadku jeśli w tablicy znajduje się lista obiektów Entry ? Skąd wiemy która wartość pasuje do klucza skoro tylko hash wskazuje nam szukany obiekt a wszystkie wartości hash obiektów w liście są równe wyliczonej wartości? Tutaj warto pamiętać o tym, że Obiekty Entry przechowują zarówno hash, **klucz**, wartość oraz wskazanie na następny element o ile istnieje.

Dlatego ważne jest aby obiekt klucz w mapie był Immutable i nie zmieniał swoich pól pomiędzy wywołaniem metody put oraz get.

Dynamiczne powiększanie HashMapy oraz Race Condition

Co dzieje się z naszą mapą, jeśli przekroczymy próg elementów zdefiniowany przez zmienną **Int threshold**?

W takim przypadku tworzona jest nowa mapa o rozmiarze dwa razy większym niż poprzednia. Elementy ze starej mapy zostają przepisane do nowej z przeliczeniem indeksów wykorzystując do tego hash zapisany w obiekcie i nowy rozmiar tablicy **tab[i=(n-1)&hash]**.

Teoretycznie wydaje się to dobrym rozwiązaniem, ale jeśli odbywa się to w środowisku wielowątkowym operacja ta narażona jest na poważne konsekwencje potencjalnego race condition tzw. wyścigu wątków. Występuje on np. w sytuacji kiedy dwa wątki w tym samym czasie zorientują się iż mapa osiągnęła swój threshold i należy ją rozszerzyć. Każdy z wątków tworzy nową tablicę dwa razy większą od pierwotnej i zaczyna alokować zawartość starej tablicy w dopiero co utworzonej. Następstwem tych działań mogą być utracone dane. W starszych wersjach Javy przy rozszerzaniu tablicy przez 2 wątki jeden z nich mógł wpaść w pętlę nieskończoną.

Innym przykładem jest dodawanie przez 2 wątki po 1000 elementów każdy, do mapy. Wielkość mapy przy każdym odpaleniu programu może się różnić, elementy mogą zostać zgubione.

Generalnie korzystanie z HashMap nie jest dobrym pomysłem jeśli chodzi o środowisko wielowątkowe. Dużo lepszym rozwiązaniem jest użycie implementacji ConcurrentHashMap, która jest bezpieczna wielowątkowo. Różnice pomiędzy obiema implementacjami opisze w przyszłości w osobnym poście.

Mapy nie implementują interfejsu Collection<E>, ale są częścią Collections API

106. Autoboxing and Unboxing

Literały a autoboxing

Pisząc swój pierwszy kod na pewno zastanawiałeś/aś się dlaczego klasy takie jak **String**, **Integer**, czy **Double** inicjalizuje się poprzez znak '='. Przecież mówiłem w poprzedniej lekcji, że obiekty tworzy się poprzez słowo kluczowe 'new'. Tak jest w rzeczy samej. Także obiekty wyżej wymienionych klas można stworzyć w ten sam sposób. Dlaczego więc istnieje opcja przypisywania poprzez znak równości, tak jak w przypadku zmiennych prostych? Nie ma tu żadnej pokrętnej logiki, jest to już cecha samego języka. Prawdopodobnie twórcy Java stwierdzili, że tak będzie po prostu łatwiej dla wielu programistów. Obiekty, które możemy stworzyć zarówno poprzez *new* jak i poprzez operator przypisania nazywa się literałami.

Przykłady literałów w Javie:

- **String**
- Char
- Byte
- Short
- Integer
- Long
- Float
- Double
- Boolean

Tak jak się pewnie domyślasz, są to obiektowe odpowiedniki wszystkich typów prostych (poza String, który nie ma typu prostego)

```
public class Autoboxing {
    public static void main(String [] args) {
        Integer mySalary = new Integer(10);
        int doubleSalary = 2 * mySalary;
        Integer salaryPlusBonus = doubleSalary + 500;
        int intValue = salaryPlusBonus.intValue();
        Integer integerValue = new Integer(intValue);
    }
}
```

Czym jest *autoboxing*?

W przykładzie wyżej widać, że typy proste i ich odpowiedniki można łatwo przekształcić w siebie nawzajem. Poza łatwym zapisem takie rozwiązanie daje jeszcze jedną zaletę. Czasami potrzebujemy liczb całkowitą w formie obiektowej. Wtedy z pomocą przychodzą klasy osłonowe. Oba typy można używać zamiennie, ponieważ kompilator wykonuje automatycznie konwersje z jednego na drugi.

Dlaczego, więc w ogóle istnieją typy proste i ich obiektowe odpowiedniki?

Otóż dawno temu typy proste wystarczały w wielu sytuacjach do programowania (np. gdy potrzebowaliśmy liczb całkowitych lub ułamkowych). Od Javy w wersji 5 powstały specjalne typy generyczne, które umożliwiają na przykład wygodną pracę z kolekcjami (np. z tablicą dynamiczną *ArrayList*). Niestety ich wadą jest używanie jedynie obiektów wewnątrz siebie, a nie typów prostych. Dlatego też stworzono obiektową wersję każdego typu prostego.

Jeśli chcesz możesz śmiało zawsze używać klas osłonowych zamiast prymitywów, istnieje jednak jeden powód, dla którego warto rozważyć deklaracje zmiennej np. *int* czy *float*. Jest nim optymalizacja pamięci, każdy obiekt poza danymi 'waży' też swoją referencję. Poza tym klasy osłonowe takie jak *Integer* posiadają w sobie dodatkowe implementacje. W praktyce jednak poza skrajnymi przypadkami nie ma to dużego znaczenia czy użyjesz typu prostego czy jego wersji obiektowej.

In Java, primitive data types are treated differently so do there comes the introduction of **wrapper classes** where two components play a role namely Autoboxing and Unboxing. **Autoboxing** refers to the conversion of a primitive value into an object of the corresponding wrapper class is called autoboxing. For example, converting int to Integer class. The Java compiler applies autoboxing when a primitive value is:

- Passed as a parameter to a method that **expects an object** of the corresponding wrapper class.
- Assigned to a variable of the corresponding **wrapper class**.

Unboxing on the other hand refers to converting an object of a wrapper type to its corresponding primitive value. For example conversion of Integer to int. The Java compiler applies to unbox when an object of a wrapper class is:

- Passed as a parameter to a method that **expects a value** of the corresponding primitive type.
- Assigned to a variable of the corresponding **primitive type**.

Now let us discuss a few advantages of autoboxing and unboxing in order to get why we are using it.

- Autoboxing and unboxing lets developers write cleaner code, making it easier to read.
- The technique lets us use primitive types and Wrapper class objects interchangeably and we do not need to perform any typecasting explicitly.

Let's understand how the compiler did autoboxing and unboxing in the example of Collections in Java using generics.

```

// Java Program to Illustrate Autoboxing

// Importing required classes
import java.io.*;
import java.util.*;

// Main class
class GFG {

    // Main driver method
    public static void main(String[] args)
    {

        // Creating an empty ArrayList of integer type
        ArrayList<Integer> al = new ArrayList<Integer>();

        // Adding the int primitives type values
        // using add() method
        // Autoboxing
        al.add(1);
        al.add(2);
        al.add(24);

        // Printing the ArrayList elements
        System.out.println("ArrayList: " + al);
    }
}

```

In the above example, we have created a list of elements of the Integer type. We are adding int primitive type values instead of Integer Object and the code is successfully compiled. It does not generate a compile-time error as the Java compiler creates an Integer wrapper Object from primitive int i and adds it to the list.

Autoboxing and unboxing lets developers write cleaner code, making it easier to read.

107. Stream metoda Peek

Zacniemy od metody peek(...), która jest metodą bliźniaczą do wcześniej poznanej metody forEach(...). Podobnie jak ona umożliwia wykonanie dowolnej operacji na każdym elemencie strumienia (używa tego samego interfejsu - Consumer). Istnieje jednak między nimi dość istotna różnica - **metoda peek(...) nie kończy strumienia** (w przeciwieństwie do forEach(...)) co oznacza, że **może być użyta do wykonania operacji na elemencie**

"wewnątrz" strumienia (jako operacja pośrednia pomiędzy innymi). Jest to o tyle istotne, ponieważ dzięki niej możemy zaobserwować jak po kolei przebiegają operacje strumienia, np. wypisując element przed i po przefiltrowaniu.

```
Przed filtrowaniem: 5
Przed filtrowaniem: 2
Przed filtrowaniem: 6
Po filtrowaniu: 6
Przed filtrowaniem: 9
Po filtrowaniu: 9
Przed filtrowaniem: 4
Przed filtrowaniem: 8
Po filtrowaniu: 8
```

Zauważ, że strumień nie wykonuje każdej operacji dla wszystkich elementów tylko przepuszcza po kolei wszystkie elementy przez wszystkie operacje. Ludzkimi słowami - powyższy strumień bierze każdy element, wypisuje go (peek), wykonuje filtrowanie (filter) i jeśli "przeżył" filtrowanie, wypisuje go (forEach) a następnie przechodzi do następnego elementu. Znowu widzimy analogię do linii produkcyjnej - elementy poruszają się stale zamiast czekać na wykonanie jednej operacji na wszystkich.

108. Rodzaje Interfejsów funkcyjnych

Większość interfejsów funkcyjnych znajdziesz w pakiecie *java.util.function*.

Istnieją jednak wyjątki od tej reguły.

Biorąc pod uwagę pakiet *java.util.function* wyróżnia się kilka podstawowych kategorii interfejsów funkcyjnych:

Interfejs Supplier

Zadaniem tego typu interfejsów jest produkowanie (dostarczanie) danych. Możesz go rozumieć jako metodę, która **nie przyjmuje żadnego parametru ale zwraca jakieś wartości**.

```
@FunctionalInterface
public interface Supplier<T> {

    /**
     * Gets a result.
     *
     * @return a result
     */
    T get();
}
```

Przykładowe użycie:

```
int age = 20;
Supplier<String> ageInfo = () -> "Ania age is " + age;
List<String> namesWithAges = new ArrayList<>();
namesWithAges.add(ageInfo.get());
```

Przypisanie tutaj zostało zastosowane tylko dla pokazania, że faktycznie jest to interfejs *Supplier*. Wartość *age* nie jest parametrem lambdy, traktuj ją jak finalną zmienną klasową w przypadku standardowych metod.

Interfejs Consumer

W przeciwieństwie do interfejsu *Supplier*, ten rodzaj interfejsu skupia się na przetwarzaniu dostarczonej z zewnątrz wartości jednocześnie bez zwracania jakiegokolwiek rezultatu.

```
@FunctionalInterface
public interface Consumer<T> {

    /**
     * Performs this operation on the given argument.
     *
     * @param t the input argument
     */
    void accept(T t);
```

Świetnym przykładem wykorzystania interfejsu *Consumer* jest nowo dodana do JDK 8 metoda *forEach*, która umożliwia łatwą iterację po kolekcjach:

```
List<String> names = new ArrayList<>();
names.add("Ania");
names.add("Tomek");
names.add("Janusz");
Consumer<String> action = name -> System.out.println("Name is " + name);
names.forEach(action);
```

Teraz spytasz się, jak ten zapis nic nie zwraca, skoro ewidentnie widać przypisanie do *name*. Nie jest to przypisanie wartości zwracanej, ale uproszczenie zapisu lambdy. Ta funkcja przecież nic nie zwraca, bo *System.out.println* nie daje rezultatu, jedynie go wyświetla.

Normalnie wyglądałby mniej więcej tak: *Consumer action = (name) -> ...*). W tym przypadku można jednak opuścić nawiasy.

Interfejs Predicate

Predykat to kolejny ciekawy interfejs, który umożliwia sprawdzenie, czy dany warunek zachodzi. Funkcja ta waliduje jakiś obiekt, po czym zwraca prawdę albo fałsz.

```
@FunctionalInterface
public interface Predicate<T> {

    /**
     * Evaluates this predicate on the given argument.
     *
     * @param t the input argument
     * @return {@code true} if the input argument matches the predicate,
     * otherwise {@code false}
     */
    boolean test(T t);
}
```

Skorzystam teraz z poprzedniej listy imion oraz wcześniej omawianej metody *foreach*.

```
names.forEach(nameToCheck -> {
    Predicate<String> checkName = (name) -> name.equals("Ania");

    if (checkName.test(nameToCheck)) {
        System.out.println("I found " + nameToCheck);
    }
});
```

Zawartość *forEach* jest oczywiście lambdą (jak widzisz można je zagnieźdźać). Za pomocą metody *equals* mój predykat weryfikuje czy sprawdzane imię to „Ania”. Następnie korzystam z niego, walidując mój warunek poprzez metodę *test*.

Interfejs Function

Interfejsy funkcyjne typu *Function* są jednymi z najczęściej używanych. Powodem jest to, że łączy on strukturę zarówno interfejsów *Supplier*, jak i *Consumer*. Jest on odpowiedzią na potrzebę stworzenia funkcji, która będzie zarówno przyjmować jakiś parametr, jak i go przetwarzać i zwracać rezultat, będący innym obiektem.

```
@FunctionalInterface
public interface Function<T, R> {

    /**
     * Applies this function to the given argument.
     *
     * @param t the function argument
     * @return the function result
     */
    R apply(T t);
}
```

Świetnym przykładem wykorzystania tego interfejsu jest wyliczenie długości *stringa* za pomocą metody *length*.

```
names.forEach(nameToCheck -> {
    Function<String, Integer> func = name -> name.length();
    Integer nameLength = func.apply(nameToCheck);
    System.out.println("Length of name " + nameLength);
});
```

Udało się w ten prosty sposób napisać krótki kod, który pracuje na obiekcie typu *String*, ale zwraca wartość typu *Integer* (długość napisu). Wszystko dzięki prostej lambdzie, użytej w metodzie *apply*. Podobnym interfejsem funkcyjnym jest do *Function* jest dziedziczony po nim interfejs **Operator**.

Interfejs Operator

Interfejs operator nie występuje w ogólnej formie. Za to najpopularniejsze interfejsy tego typu to *UnaryOperator* i *BinaryOperator*.

UnaryOperator

UnaryOperator, czyli operator jednoargumentowy. Reprezentuje on operację na pojedynczym parametrze, która daje wynik tego samego typu. Przykładem operatora jednoargumentowego jest znana Ci operacja inkrementacji (np. *i++*). Masz tu tylko jeden operand (czyli zmienną *i*).

```
@FunctionalInterface
public interface UnaryOperator<T> extends Function<T, T> {

    /**
     * Returns a unary operator that always returns its input argument.
     *
     * @param <T> the type of the input and output of the operator
     * @return a unary operator that always returns its input argument
     */
    static <T> UnaryOperator<T> identity() {
        return t -> t;
    }
}
```

UnaryOperator znaleźć można wśród chociażby metod z klasy *String*, takich jak np. *toUpperCase*. Powodem, dlaczego ta metoda, odpowiada strukturze tego typu operatora, jest jego sposób użycia. Zauważ, że jest ona użyta na konkretnej zmiennej typu *String*. Nie przyjmuje żadnego parametru, nie potrzebuje by wywołana z innej klasy (podobnie jak inkrementacja dla typów prostych).

```
UnaryOperator<String> operator = String::toUpperCase; // alternative to name -> name.toUpperCase()
names.replaceAll(operator);
```

Teraz wszystkie imiona będą zapisane z pomocą wielkich liter. W pierwszej linijce można zauważyć zapis, korzystający z **podwójnego dwukropka**. Jest to alternatywna implementacja, zwana **referencjami do metod** (ang. *method reference*), w stosunku do kodu z wykorzystaniem strzałki (*->*). Możesz go zastosować, kiedy używasz prostej jednolinijkowej lambdy. Pamiętaj jednak, że ten zapis zataja argumenty funkcji użyte w lambdzie, także bezpieczniej jest używać zwykłej lambdy (szczególnie w przypadku, gdy korzystasz z większej liczby argumentów).

BinaryOperator

Przyjmuje dwa argumenty tego samego typu i zwraca wynik tego samego typu


```

@FunctionalInterface
interface BinaryOperator<T> extends BiFunction<T, T, T> {

    //Zwraca mniejszy element zgodnie z przekazany Comparator
    public static <T> BinaryOperator<T> minBy(Comparator<? super T> comparator) {
        Objects.requireNonNull(comparator);
        return (a, b) -> comparator.compare(a, b) <= 0 ? a : b;
    }

    //Zwraca większy element zgodnie z przekazany Comparator
    public static <T> BinaryOperator<T> maxBy(Comparator<? super T> comparator) {
        Objects.requireNonNull(comparator);
        return (a, b) -> comparator.compare(a, b) >= 0 ? a : b;
    }
}

```

BinaryOperator rozszerza interfejs **BiFunction**, który jest przedstawiony wyżej. Działanie tych interfejsów jest prawie takie same, z małą różnicą. **BinaryOperator** działa tylko na jednym i takim samym typie. **BinaryOperator** dodatkowo posiada dwie metody default **minBy** oraz **maxBy**.

Na początek metoda **apply**:

```

BinaryOperator<Integer> binaryOperator = (x, y) -> x + y;
System.out.println(binaryOperator.apply(4,5));

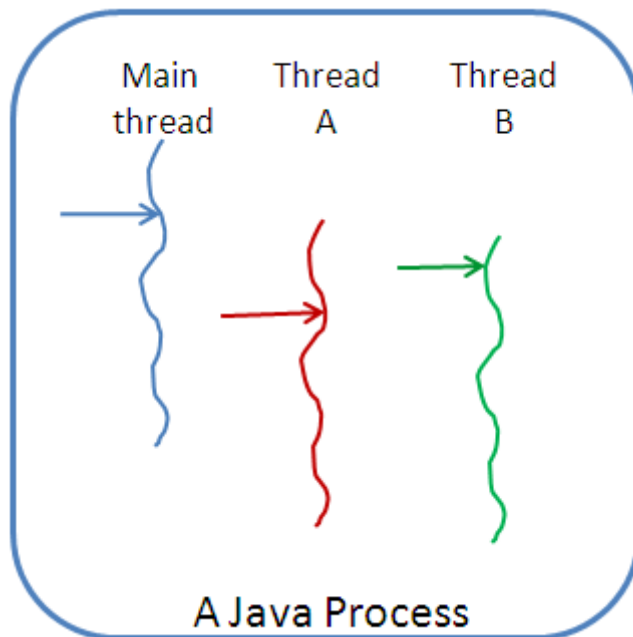
```

109. Wątki

Tworząc swoje aplikacje, szczególnie te wykorzystujące interfejs użytkownika z pewnością spotkałeś się z sytuacją, w której pewna czynność, jak na przykład obliczenie wyniku skomplikowanej funkcji, czy pobranie pewnych danych z bazy danych, zabierało dużo czasu, a przez to aplikacja sprawiała wrażenie jakby się zawiesiła. Nie jest to pożądana funkcjonalność i każdy programista chciałby jej uniknąć.

Istnieje jednak rozwiązanie, które w stosunkowo prosty sposób pozwala poradzić sobie z tym problemem – są to **wątki**.

Wątki pozwalają na symultaniczne wykonywanie pewnych operacji dzięki czemu czas wykonania pewnych operacji można znacząco skrócić. W przypadku przykładu z „zawieszeniem” się interfejsu użytkownika można pewne skomplikowane obliczenia wykonać **asynchronicznie** w tle, dzięki czemu użytkownik aplikacji będzie miał lepsze odczucia w związku z jej użytkowaniem. Na obrazku wygląda to tak:



Kiedy wykorzystywać wątki? W wielu sytuacjach:

- Wszelkie obliczenia, które mogą zablokować interfejs użytkownika powinny być wykonywane asynchronicznie
- Animacje, które powinny być przetwarzane niezależnie od interfejsu użytkownika
- Pobieranie danych z internetu (zamiast przetwarzać strony internetowe jedna po drugiej można połączyć się np. z 10 jednocześnie)
- W ogólności wszystkie operacje wejścia/wyjścia, zapis i odczyt plików, czy baz danych
- Złożone obliczenia, które mogą być podzielone na mniejsze podzadania
- I wiele innych

Wątki w Javie można tworzyć na kilka sposobów, poprzez:

- jawne rozszerzenie klasy `Thread`
- stworzenie klasy implementującej interfejs `Runnable`, który może być wykonany w osobnym wątku (`Thread`)
- stworzenie klasy implementującej interfejs `Callable`, który może być wykonany w osobnym wątku (`Thread`)

Preferowane jest stosowanie rozszerzeń interfejsów (czyli 2 i 3 punkt), ponieważ dają one dużo lepszą elastyczność, szczególnie jeśli dojdziemy do momentu szeregowania wątków, utrzymywania stałej puli wątków wykonujących się w tle. Interfejsy `Runnable` i `Callable` są do siebie bardzo podobne, jednak najważniejszą różnicą jest to, że `Callable` może zwrócić w wyniku pewną wartość, natomiast w przypadku `Runnable` nie ma takiej możliwości.

110. Runnable vs Callable

Both interfaces are designed to represent a task that can be executed by multiple threads. *Runnable* tasks can be run using the *Thread* class or *ExecutorService* whereas *Callables* can be run only using the latter.

The **Runnable** interface is a functional interface and has a single *run()* method which doesn't accept any parameters and does not return any values.

This is suitable for situations where we are not looking for a result of the thread execution, for example, incoming events logging:

```
public interface Runnable {  
    public void run();  
}
```

Let's understand this with an example:

```
public class EventLoggingTask implements Runnable{
    private Logger logger
        = LoggerFactory.getLogger(EventLoggingTask.class);

    @Override
    public void run() {
        logger.info("Message");
    }
}
```

In this example, the thread will just read a message from the queue and log it in a log file. There's no value returned from the task; the task can be launched using *ExecutorService*:

```
public void executeTask() {
    executorService = Executors.newSingleThreadExecutor();
    Future future = executorService.submit(new EventLoggingTask());
    executorService.shutdown();
}
```

In this case, the *Future* object will not hold any value.

The **Callable** interface is a generic interface containing a single *call()* method – which returns a generic value *V*:

```
public interface Callable<V> {
    V call() throws Exception;
}
```

Let's have a look at calculating the factorial of a number:

```
public class FactorialTask implements Callable<Integer> {
    int number;

    // standard constructors

    public Integer call() throws InvalidParamaterException {
        int fact = 1;
        // ...
        for(int count = number; count > 1; count--) {
            fact = fact * count;
        }

        return fact;
    }
}
```

The result of *call()* method is returned within a *Future* object:

```
@Test
public void whenTaskSubmitted_ThenFutureResultObtained(){
    FactorialTask task = new FactorialTask(5);
    Future<Integer> future = executorService.submit(task)

    assertEquals(120, future.get().intValue());
}
```

Exception Handling

With Runnable

Since the method signature does not have the “throws” clause specified, there is no way to propagate further checked exceptions.

With Callable

Callable's *call()* method contains “throws *Exception*” clause so we can easily propagate checked exceptions further:

```

public class FactorialTask implements Callable<Integer> {
    // ...
    public Integer call() throws InvalidParamaterException {

        if(number < 0) {
            throw new InvalidParamaterException("Number should be positive");
        }
        // ...
    }
}

```

In case of running a *Callable* using an *ExecutorService*, the exceptions are collected in the *Future* object, which can be checked by making a call to the *Future.get()* method. This will throw an *ExecutionException* – which wraps the original exception:

```

@Test(expected = ExecutionException.class)
public void whenException_ThenCallableThrowsIt() {

    FactorialCallableTask task = new FactorialCallableTask(-5);
    Future<Integer> future = executorService.submit(task);
    Integer result = future.get().intValue();
}

```

In the above test, the *ExecutionException* is being thrown as we are passing an invalid number. We can call the *getCause()* method on this exception object to get the original checked exception.

If we don't make the call to the *get()* method of *Future* class – then the exception thrown by *call()* method will not be reported back, and the task will still be marked as completed:

```

@Test
public void whenException_ThenCallableDoesntThrowsItIfGetIsNotCalled(){
    FactorialCallableTask task = new FactorialCallableTask(-5);
    Future<Integer> future = executorService.submit(task);

    assertEquals(false, future.isDone());
}

```

The above test will pass successfully even though we've thrown an exception for the negative values of the parameter to *FactorialCallableTask*.

111. Try-with-resources Statement

The try-with-resources statement is a try statement that declares one or more resources. A *resource* is an object that must be closed after the program is finished with it. The try-with-resources statement ensures that each resource is closed at the end of the statement. Any object that implements `java.lang.AutoCloseable`, which includes all objects which implement `java.io.Closeable`, can be used as a resource.

The following example reads the first line from a file. It uses an instance of `FileReader` and `BufferedReader` to read data from the file. `FileReader` and `BufferedReader` are resources that must be closed after the program is finished with it:

```
static String readFirstLineFromFile(String path) throws IOException {  
  
    try (FileReader fr = new FileReader(path);  
  
        BufferedReader br = new BufferedReader(fr)) {  
  
        return br.readLine();  
  
    }  
  
    }
```

In this example, the resources declared in the try-with-resources statement are a `FileReader` and a `BufferedReader`. The declaration statements of these resources appear within parentheses immediately after the try keyword. The classes `FileReader` and `BufferedReader`, in Java SE 7 and later, implement the interface `java.lang.AutoCloseable`. Because the `FileReader` and `BufferedReader` instances are declared in a try-with-resource statement, they will be closed regardless of whether the try statement completes normally or abruptly (as a result of the method `BufferedReader.readLine` throwing an `IOException`).

Prior to Java SE 7, you can use a finally block to ensure that a resource is closed regardless of whether the try statement completes normally or abruptly. The following example uses a finally block instead of a try-with-resources statement:

```
static String readFirstLineFromFileWithFinallyBlock(String path) throws IOException {  
  
    FileReader fr = new FileReader(path);  
  
    BufferedReader br = new BufferedReader(fr);
```

```
try {  
    return br.readLine();  
} finally {  
    br.close();  
    fr.close();  
}  
}
```

However, this example might have a resource leak. A program has to do more than rely on the garbage collector (GC) to reclaim a resource's memory when it's finished with it. The program must also release the resource back to the operating system, typically by calling the resource's close method. However, if a program fails to do this before the GC reclaims the resource, then the information needed to release the resource is lost. The resource, which is still considered by the operating system to be in use, has leaked.

In this example, if the readLine method throws an exception, and the statement br.close() in the finally block throws an exception, then the FileReader has leaked. Therefore, use a try-with-resources statement instead of a finally block to close your program's resources.

112. W celu przechowywania haseł użytkowników bezpieczniej będzie użyć tablicy znaków czy zmiennej typu String?

Tablicy znaków, stringi są niezmiennie i dłużej przechowywane w pamięci.

113. String...

Do metody można przekazać zarówno String... jak i Integer... jak i inne obiekty. Typy prymitywne też można.

W takiej formie też można:


```

    printOut( ...inty: 1, 3, 4);
}
static void printOut(int... inty) {
    for (int s : inty) {
        System.out.println(s);
    }
}
}

```


114. Dlaczego Comparator jest interfejsem funkcyjnym, a ma dwie metody abstrakcyjne?

If an interface declares an abstract method overriding one of the public methods of `java.lang.Object`, that also does not count toward the interface's abstract method count since any implementation of the interface will have an implementation from `java.lang.Object` or elsewhere.

And since `equals` is one of those methods, the "abstract method count" of the interface is still 1.

Rules:

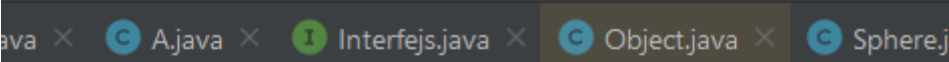
1. A functional interface has one and only one abstract method.
2. The default methods are not abstract methods because they are already implemented.
3. A method that overrides any public method in the superclass `Object` class is not considered an abstract method in the interface.

```
public interface Interfejs {  
     boolean equals(Object obj);  
}
```



- Insert '@Contract(value = "null -> false", pure = true)' >
- Make 'equals()' default >
- Add Javadoc >
- Generate overloaded method with default parameter values >
- Go to super method** Ctrl+U


Press Ctrl+Q to open preview



Implementation The equals method for class Object implements the following requirements: discriminating possible equivalence relation on the objects of the class. For non-null reference values x and y, this method returns true if and only if x and y refer to the same object (x == y). In other words, under the reference equality equivalence relation, the equals method of the Object class only has a single element.

See Also: [hashCode\(\)](#), [java.util.HashMap](#)

```
public boolean equals(Object obj) {  
    return (this == obj);  
}
```



Creates and returns a copy of this object. The precise meaning of the copy is dependent on the object's class.

115. Comparable vs Comparator

Comparable in Java is used to sort the objects with natural ordering while Comparator is used to sort attributes of different objects. Let's understand these interfaces through the medium of this article.

What is Comparable in Java?

As the name itself suggests, **Comparable** is an interface which defines a way to compare an object with other objects of the same type. It helps to sort the objects that have self-tendency to sort themselves, i.e., the objects must know how to order themselves. **Eg:** Roll number, age, salary. This interface is found in *java.lang package* and it contains only one method, i.e., *compareTo()*. Comparable is not capable of sorting the objects on its own, but the interface defines a method *int compareTo()* which is responsible for sorting.

Further, you must be thinking what is the compareTo method? Well, let me explain that to you!

What is the compareTo method and how it is used?

This method is used to compare the given object with the current object.

The **compareTo()** method returns an int value. The value can be either positive, negative, or zero. So now we are well acquainted with the theoretical knowledge of Comparable interface in Java and **compareTo** method.

Let's hop into understanding the implementation process. First, let's see how to implement Comparable.

Java Comparable example

```
public class Student implements Comparable {
    private String name;
    private int age;
    public Student(String name, int age) {
        this.name = name;
        this.age = age;
    }
    public int getAge() {
        return this.age;
    }
    public String getName() {
        return this.name;
    }
    @Override
    public String toString() {
        return "";
    }
    @Override
    public int compareTo(Student per) {
        if(this.age == per.age)
            return 0;
        else
            return this.age > per.age ? 1 : -1;
    }
}
```

```

public static void main(String[] args) {
    Person e1 = new Person("Adam", 45);
    Person e2 = new Person("Steve", 60);
    int retval = e1.compareTo(e2);

    switch(retval) {
        case -1: {
            System.out.println("The " + e2.getName() + " is older!");
            break;
        }

        case 1: {
            System.out.println("The " + e1.getName() + " is older!");
            break;
        }

        default:
            System.out.println("The two persons are of the same age!");
    }
}

```

In the above example, I have created a class Student with two fields, name and age. Class Student is implementing the Comparable interface and overrides the compareTo method. This method sorts the instances of the Student class, based on their age.

Now that I have covered Comparable in Java, moving on I will talk about another interface i.e. Comparator in Java. Let's move to understanding Comparator in Java!

What is Comparator in Java?

A Comparator interface is used to order the objects of a specific class. This interface is found in java.util package. It contains two methods;

- compare(Object obj1, Object obj2)
- equals(Object element).

The first method, compare(Object obj1, Object obj2) compares its two input arguments and showcase the output. It returns a negative integer, zero, or a positive integer to state whether the first argument is less than, equal to, or greater than the second.

The second method, equals(Object element), requires an Object as a parameter and shows if the input object is equal to the comparator. The method will return true, only if the mentioned object is also a Comparator. The order remains the same as that of the Comparator.

After attaining brief learning about Comparator in Java, it's time to move a step ahead. Let me show you an example depicting Comparator in Java.

How to implement Comparator in Java

```
import java.util.Comparator;

public class School {
    private int num_of_students;
    private String name;
    public School(String name, int num_of_students) {
        this.name = name;
        this.num_of_students = num_of_students;
    }
    public int getNumOfStudents() {
        return this.num_of_students;
    }
    public String getName() {
        return this.name;
    }
}

public class SortSchools implements Comparator {
    @Override
    public int compare(School sch1, School sch2) {
        if(sch1.getNumOfStudents()== sch2.getNumOfStudents())
            return 0;
        else
            return sch1.getNumOfStudents() > sch2.getNumOfStudents() ? 1 : -1;
    }
    public static void main(String[] args) {
        School sch1 = new School("sch1", 20);
        School sch2 = new School("sch2", 15);
        SortSchools sortSch = new SortSchools();
        int retval = sortSch.compare(sch1, sch2);
        switch(retval) {
            case -1: {
                System.out.println("The " + sch2.getName() + " is bigger!");
                break;
            }
            case 1: {
                System.out.println("The " + sch1.getName() + " is bigger!");
                break;
            }
            default:
                System.out.println("The two schools are of the same size!");
        }
    }
}
```

Output:

The sch1 is bigger!

Well, no need to panic here. The above-written code is really easy to understand. Let's go!

First, I created a class School that consists of the name and age of the students. After that, I created another class, SortSchools, in order to implement the Comparator interface which accomplishes the goal of imposing an order between instances of the first class named, School, according to the number of students.

Comparable v/s Comparator in Java

Comparable in Java	Comparator in Java
Comparable interface is used to sort the objects with natural ordering.	Comparator in Java is used to sort object by its specific attribute
Comparable interface compares "this" reference with the object specified.	Comparator in Java compares two different class objects provided.
Comparable is present in java.lang package.	A Comparator is present in the java.util package.
Comparable affects the original class, i.e., the actual class is modified.	Comparator doesn't affect the original class
Comparable provides compareTo() method to sort elements.	Comparator provides compare() method, equals() method to sort elements.

116. Iterator vs ListIterator

Iterator

Iterators are used in Collection framework in Java to retrieve elements one by one. It can be applied to any Collection object. By using Iterator, we can perform both read and remove operations. Iterator must be used whenever we want to enumerate elements in all Collection framework implemented interfaces like Set, List, Queue, Deque and also in all implemented classes of Map interface. Iterator is the only cursor available for entire collection framework.

Iterator object can be created by calling iterator() method present in Collection interface.

// Here "c" is any Collection object. itr is of

// type Iterator interface and refers to "c"

Iterator itr = c.iterator();

ListIterator

It is only applicable for List collection implemented classes like arraylist, linkedlist etc. It provides bi-directional iteration. ListIterator must be used when we want to enumerate elements of List. This cursor has more functionality(methods) than iterator.

ListIterator object can be created by calling listIterator() method present in List interface.

```
// Here "l" is any List object, ltr is of type
```

```
// ListIterator interface and refers to "l"
```

```
ListIterator ltr = l.listIterator();
```

Differences between Iterator and ListIterator:

1. Iterator can traverse only in forward direction whereas ListIterator traverses both in forward and backward directions.

Example:

```
public static void main(String[] args)
{
    ArrayList<Integer> list
        = new ArrayList<Integer>();

    list.add(1);
    list.add(2);
    list.add(3);
    list.add(4);
    list.add(5);

    // Iterator
    Iterator itr = list.iterator();

    System.out.println("Iterator:");
    System.out.println("Forward traversal: ");

    while (itr.hasNext())
        System.out.print(itr.next() + " ");

    System.out.println();

    // ListIterator
    ListIterator i = list.listIterator();

    System.out.println("ListIterator:");
    System.out.println("Forward Traversal : ");

    while (i.hasNext())
        System.out.print(i.next() + " ");

    System.out.println();

    System.out.println("Backward Traversal : ");

    while (i.hasPrevious())
        System.out.print(i.previous() + " ");

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

2. ListIterator can help to replace an element whereas Iterator cannot.

Example:

```
public class ListIteratorDemo2 {
    public static void main(String[] args)
    {
        ArrayList<Integer> aList
            = new ArrayList<Integer>();
        aList.add(1);
        aList.add(2);
        aList.add(3);
        aList.add(4);
        aList.add(5);

        System.out.println("Elements of ArrayList: ");
        for (Integer i : aList) {
            System.out.println(i);
        }
        ListIterator<Integer> l
            = aList.listIterator();
        l.next();
        l.set(80000);

        System.out.println("\nNow the ArrayList"
            + " elements are: ");
        for (Integer i : aList) {
            System.out.println(i);
        }
    }
}
```

Table showing Difference between Iterator and ListIterator

Iterator	ListIterator
Can traverse elements present in Collection only in the forward direction.	Can traverse elements present in Collection both in forward and backward directions.
Helps to traverse Map, List and Set.	Can only traverse List and not the other two.
Indexes cannot be obtained by using Iterator.	It has methods like nextIndex() and previousIndex() to obtain indexes of elements at any time while traversing List.

Iterator

Cannot modify or replace elements present in Collection

Cannot add elements and it throws `ConcurrentModificationException`.

Certain methods of `Iterator` are `next()`, `remove()` and `hasNext()`.

ListIterator

We can modify or replace elements with the help of `set(E e)`

Can easily add elements to a collection at any time.

Certain methods of `ListIterator` are `next()`, `previous()`, `hasNext()`, `hasPrevious()`, `add(E e)`.

117. Co się stanie w tej sytuacji

```
public static void main(String[] args) {  
  
    watchVideo( num: null);  
  
}  
  
static void watchVideo(int num) {  
    System.out.println("Video watched views: " + num);  
}
```

Odpowiedź:

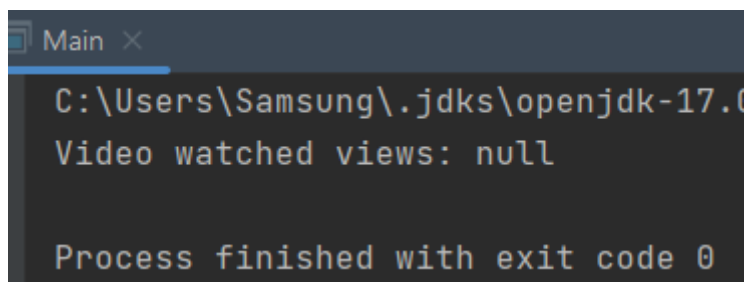
```
C:\Users\Samsung\IdeaProjects\BlazejProject\src>Main.java:28:20  
java: incompatible types: <nulltype> cannot be converted to int
```

Kompilator nie dopuści do skompilowania programu.

118. A w tej?

```
public static void main(String[] args) {  
  
    watchVideo(num: null);  
  
}  
  
static void watchVideo(Integer num) {  
    System.out.println("Video watched views: " + num);  
}
```

Odpowiedź:



```
Main ×  
C:\Users\Samsung\.jdk\openjdk-17.0  
Video watched views: null  
  
Process finished with exit code 0
```

119. A w tej?

```
public static void main(String[] args) {  
  
    watchVideo(num: null);  
  
}  
  
static void watchVideo(Integer num) {  
    System.out.println("Video watched views: " + num);  
}  
  
static void watchVideo(String name) {  
    System.out.println("Video watched by: " + name);  
}
```

Odpowiedź:

```
C:\Users\Samsung\IdeaProjects\BlazejProject\src\Main.java:28:9  
java: reference to watchVideo is ambiguous  
    both method watchVideo(java.lang.Integer) in Main and method watchVideo(java.l
```

Kompilator nie dopuści do skompilowania programu.

120. Czy modyfikator protected jest dozwolony w interfejsach?

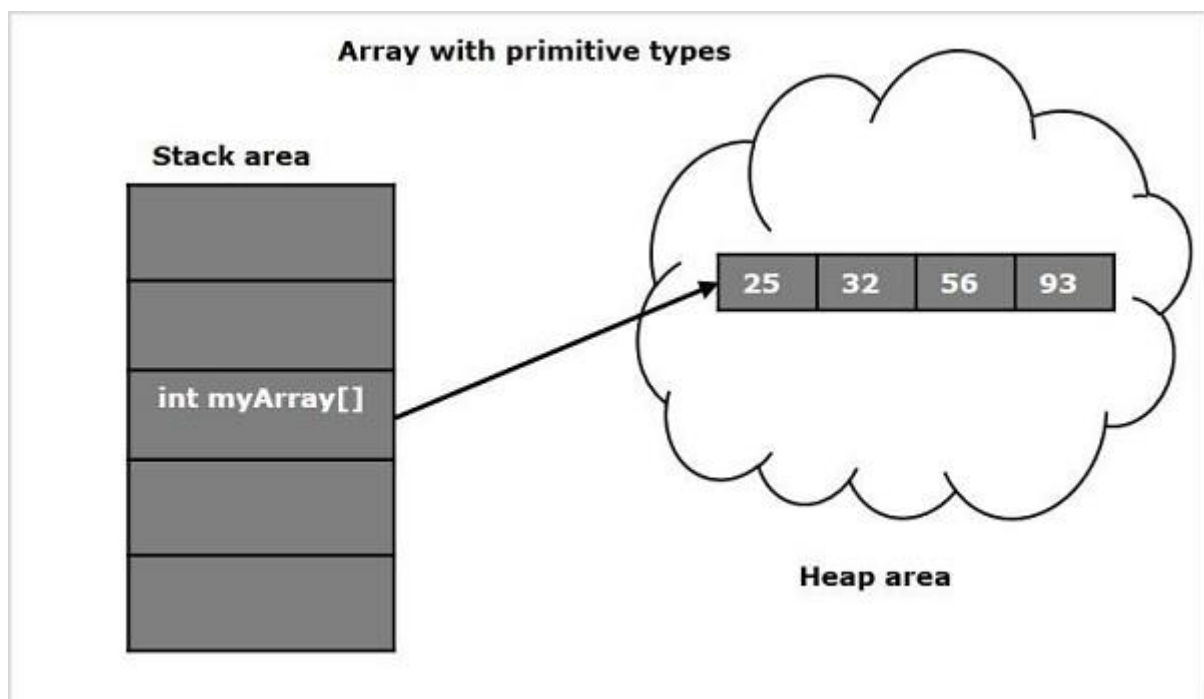
Modyfikator protected nie jest dozwolony ani w atrybutach, ani w metodach.

What are the new features introduced in Java 8?

- Default and static methods in Interfaces.
- Lambda Expressions
- @Functional Interface
- forEach() method
- Stream API
- LocalDate , LocalTime, LocalDateTime

121. Czy tablice są przechowywane na stosie czy na sterpie obiektów?

Jak wspomniano, typy referencyjne w Javie są przechowywane w obszarze sterty. Ponieważ tablice są typami referencyjnymi (możemy je utworzyć za pomocą słowa kluczowego new), są one również przechowywane w obszarze sterty.

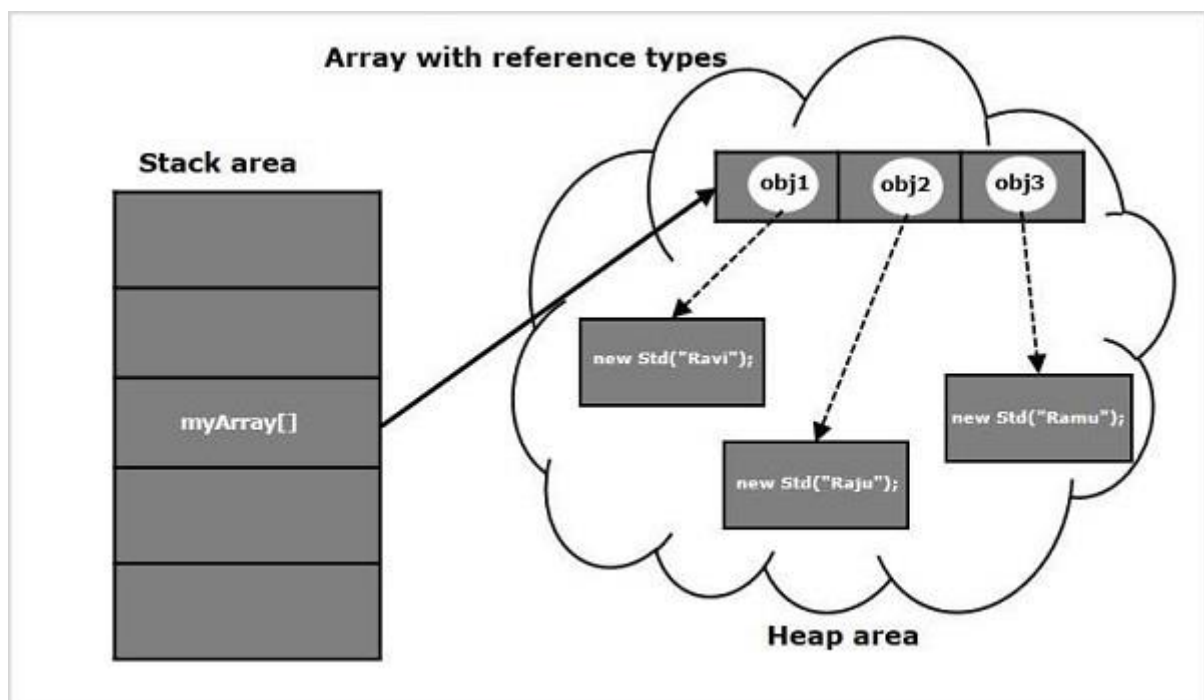


Oprócz prymitywnych tablic typów danych przechowują również typy referencyjne: Kolejne tablice (wielowymiarowe), Obiekty. W tym przypadku obiekt array/multiwymiarowy przechowuje referencje obiektów/tablic init, co wskazuje na położenie obiektów/tablic.

Założmy, że mamy klasę o nazwie Std z konstruktorem akceptującym nazwisko ucznia i zdefiniowaliśmy tablicę tej klasy i wypełniliśmy ją tak, jak pokazano poniżej.

```
class Std {  
    private String name;  
    public Std(String name){  
        this.name = name;  
    }  
}  
  
public class Sample {  
    public static void main(String args[]) throws Exception {  
        //Creating an array to store objects of type Std  
        Std myArray[] = new Std[3];  
        //Populating the array  
        myArray [0] = new Std("Ravi");  
        myArray [1] = new Std("Raju");  
        myArray [2] = new Std("Ramu");  
    }  
}
```

The memory of the array myArray might be like:



Tablice są obiektem i dla tablic są dostępne metody z klasy OBJECT.

122. Czy tablice mogą być oznaczone słowem „final”?

final in Java affects the variable, it has nothing to do with the object you are assigning to it.

```
final String[] myArray = { "hi", "there" };  
myArray = anotherArray; // Error, you can't do that. myArray is final  
myArray[0] = "over"; // perfectly fine, final has nothing to do with it
```

Edit to add from comments: Note that I said *object you are assigning to it*. In Java an array is an object. This same thing applies to any other object:

```
final List<String> myList = new ArrayList<String>();  
myList = anotherList; // error, you can't do that  
myList.add("Hi there!"); // perfectly fine.
```

123. Unitialized variables

Zmienne lokalne nie otrzymują wartości domyślnych. Ich początkowe wartości są niezdefiniowane bez przypisywania wartości w jakiś sposób. Zanim będzie można używać zmiennych lokalnych, muszą one zostać zainicjalizowane.

```
public class Main {  
  
    public static void main(String[] args) {  
  
        int x;  
        int y = (x - 1) * x;  
        System.out.println("y= " + y);  
  
    }  
}
```

```
C:\Users\Samsung\IdeaProjects\BlazejProject\src\Main.java:6:18  
java: variable x might not have been initialized
```

Istnieje duża różnica, kiedy deklarujesz zmienną na poziomie klasy (jako element, tj. jako pole) i na poziomie metody.

Jeśli zadeklarujesz pole na poziomie klasy, otrzymają wartości domyślne zgodnie z ich typem.

Jeśli zadeklarujesz zmienną na poziomie metody lub jako blok (czyli dowolny kod wewnątrz {}), pozostanie ona niezdefiniowana, dopóki w jakiś sposób nie otrzyma wartości początkowej, tj. niektórych wartości przypisanych do niej

Metoda:

```
static void doSomething() {  
    int x;  
    System.out.println(x);  
}
```

```
C:\Users\Samsung\IdeaProjects\BlazejProject\src\Main.java:6:18  
java: variable x might not have been initialized
```

```
public class Osoba {  
  
    private String imie;  
    private String nazwisko;  
    private int wiek;  
    private float wzrost;  
    private float waga;
```

```
public static void main(String[] args) {  
  
    Osoba osoba = new Osoba();  
    System.out.println("IMIE");  
    System.out.println(osoba.getImie());  
    System.out.println();  
    System.out.println("WIEK");  
    System.out.println(osoba.getWiek());  
    System.out.println();  
    System.out.println("WAGA");  
    System.out.println(osoba.getWaga());  
  
}
```

```
IMIE  
null
```

```
WIEK  
0
```

```
WAGA  
0.0
```

```
Process finished with exit code 0
```


Tablice to również obiekty, więc:

```
public static void main(String[] args) {  
  
    int[] tab = new int[10];  
    for (int element : tab) {  
        System.out.println(element);  
    }  
  
}
```

```
0  
0  
0  
0  
0  
0  
0  
0  
0  
0
```

```
Process finished with exit code 0
```

124. Can we create an object of the class in which main function is defined in Java?

Yes. The main method is just an entry point. The class is like any other, except it has an additional public static method. The main method is static and therefore not part of the instance of the object, but you shouldn't be using the main method anyway except for starting the program.

Yes, you can create object for the class which has main method. There is no difference in this class and a class which don't have main method with respect to creating objects and using.

```
public class Scratchpad {  
  
    public static void main(String[] args) {  
        Scratchpad scratchpad = new Scratchpad();  
        scratchpad.someMethod();  
    }  
  
    public Scratchpad() {  
    }  
  
    private void someMethod() {  
        System.out.println("Non-static method prints");  
    }  
}
```

```
public class Osoba {  
  
    private String imie;  
    private String nazwisko;  
    private int wiek;  
    private float wzrost;  
    private float waga;  
  
    public Main newMain() {  
        return new Main();  
    }  
}
```

125.

```
public static void main(String[] args) {  
  
    List list = new ArrayList();  
    List list = new LinkedList();  
}
```

Niedozwolone bo to tak jakbyśmy deklarowali nowy obiekt;