SCJP Notatki

---

1. Deklaracje i kontrola dostępu.

Klasa - szablon dzięki któremu tworzone są obiekty (instancje obiektów)
Obiekt - po słowie *new* tworzony jest obiekt klasy mający pewien stan i dostęp do zachowań zdefiniowanych w klasie.
Stan - wartości zmiennych klasy dla konkretnego obiektu (instancji)
Zachowanie - metody, a w nich logika i algorytmy

1.1. Konwencje nazewnicze klas i metod

Wymagane:

- must start with a letter, a currency character ($), or a connecting character such as the underscore (_). Identifiers cannot start with a number

- after the first character, identifiers can contain any combination of **letters**, **currency** chars, **connecting** chars, or **numbers**.

- nie ma limitu długości
- nie można używac keywords
- case sensitive.

Zalecane (Java Naming Conventions):

- CamelCase dla klas i interfejsów
- przymiotnik dla nazwy interfejsu np Runnable, Serializable
- camelCase dla metod i zmiennych
- verb-noun dla metod np getBalance
- UPPER_CASE dla stałych (variables *static* and *final*)

1.2. Java Beans Naming Standards

Java Bean to klasa posiadająca *properties* (właściwości) czyli **prywatne** pola i **settery/gettery** modyfikujące je.

Konwencje:

- przedrostek *get* dla *"niebooleanowskich"* zmiennych np *getSize*() (ale nie musi wcale być takiej zmiennej jak *size*)
- przedrostek *get* lub *is* dla *"booleanowskich"* (mogą być w jednym Beanie obie: *getStopped* i *isStopped* - ale to ma mało sensu)
- setter musi być *public*, *void* oraz argument zgodny z typem zmiennej ustawianej (property).
- getter musi być *public*, bezargumentowy oraz zwracac typ zgodny ze zmienną ustawianą (property).

Listener (events) naming convetions:

- metoda rejestrująca listener musi sie zaczynac od *add* , nastepnie typ listenera (na końcu słówko Listener), np *addActionListener()*
- metoda wy-rejestrowująca listener musi sie zaczynac od *remove* a nastepnie typ listenera i na końcu słówko Listener, np *removeActionListener*()
- argumentem rejestratora/derejestratora musi byc typ Listenera np *addMyListener(MyListener m)*

Powyższe reguły typu Naming Concentions dla JavaBeanów są na egzaminie!!

(przykłady patrz str 10)

1.3. Zasady plików źródłowych

- W pliku źródłowym może być zadeklarowana **jedna klasa publiczna**, ale klas nie publicznych może już być dowolnie wiele.
- Jeśli plik zawiera klasę publiczną to musi się on nazywać tak jak ta klasa + '.java'. Jeśli nie ma klasy *public*, plik może się nazywać dowolnie
- Jeśli klasa znajduje się w jakimś pakiecie to deklaracja pakietu (*package*) musi być pierwszą instrukcją w pliku.
- Jeśli klasa importuje inne klasy (*import*) to musi to zrobić po deklaracji pakietu a przed deklaracją klasy.
- Importy i deklaracja pakietu dotyczą wszystkich klas w pliku

1.4. Deklaracje i modyfikatory klas

Modyfikatory dla klas:
- Access modifiers: dla klas **public**, oraz ***default*** *(bez* modyfikatora) [dla pozostałych dodatkowo *protected* i *private*].
- Non-access modifiers**:** dla klas **strictfp**, **final**, and **abstract**.

Modyfikatory dostepu:
Dostęp do klasy oznacza - jeśli class A ma dostęp do B to może: 1) stworzyć instancje B, 2) rozszerzać B, 3) ma dostęp do metod i zmiennych B (odp do ich modyfikatorów)

- Default access (**bez modyfikatora**) - jeśli class A ma *default access* to jest widoczna tylko w tym samym pakiecie. Próba dobicia się do tej klasy (nawet import) z innego pakietu spowoduje błąd kompilacji: *Can't access class X. Class or interface must be public, in same package, or an accessible member class*.
- Public access - klasa publiczna jest dostępna (widoczna) z każdego pakietu (ale trzeba ja oczysiście zaimportować w piku)

Inne modyfikatory klasowe

Można je mieszać z modyf. dostepu, np. *public final class A{}*. **Nie można mieszać final i abstract** - błąd kompilatora.

- **strictfp class** - wszystkie operacje zmiennoprzecinkowe w danej klasie będą zgodne ze standardem IEEE 754. *Tylko dla klas i metod !*
- **final class -** taka klasa nie może być rozszerzana (subclassed). Próba spowoduje błąd kompilacji: *Can't subclass final classes*
  - po co? by mieć pewność że metody tej klasy nie zostaną *overriden*, gdy np jesteśmy zależni od ich implementacji
  - klasa final nie może mieć metod abstract - tylko klasy abstract moga miec metody abstract.

- **abstract class** - jest po to aby ją rozszerzać i nie można jej instancjonować
  - można taką klase skompilować i uruchomić ale próba instancji (new) wywoła błąd kompilacji: *Class X is an abstract class. It can't be instantiate*.
  - klasa taka może mieć metody z impementacją (z ciałem) lub z samymi deklaracjami (jak w interfejsach) ale wtedy te metody muszą być także **abstract**

- jeśli jakaś metoda w klasie jest *abstract* (więc nie ma ciała i kończy sie ;) to cała klasa musi byc **abstract** ale może mieć tez inne metody non-abstract
- **metoda** kończąca się ; musi byc poprzedzona słowem **abstract**
- metoda będąca *abstract* musi byc zaimplementowana w podklasie - podobnie jak interfejs.

1.5 Deklaracje interfejsów

Słowo kluczowe ***interface***
- Interfejs obliguje klasę, która go implementuje do implementacji **wszystkich** metod które on posiada
  - jeśli klasa tego nie robi kompilator daje 2 rady: "*zaimplementuj wszystkie metody z interfejsu*" lub "*zmień klasę na abstrakcyjną*".
- interfejs moze być implementowany przez każdy typ klasy z każdego miejsca w "drzewie dziedziczenia"

- metoda zadeklarowana w interfejsie **jest** zawsze metodą **abstrakcyjną** i **publiczną** (i można pominąć słówka *public* i *abstract*)
  - cały **interfejs** jako taki też **jest *abstract*** (można ale nie trzeba dawać *abstract* przed: *public abstract interface Rollable{}*)
  - nie może być prywatnych metod w interfejsie
- metody **nie mogą być statyczne**
- metody nie mogą byc *final*, *native* ani *strictfp* bo są abstrakcyjne
- interfejs może rozszeżać (**extends**) **tylko** interfejsy - **jeden lub wiele** (w przeciwieństwie do klasy która może rozszeżać tylko 1 klase)
  - **nie może** implementować innego interfejsu czy klasy (**implements**)
- interfejs może mieć modyfikator dostępu *public* i defaultowy (bez modyf.) - działa to tak jak w klasach

- **zmienne** w interfejsie **muszą być**:
  - *public* nie może byc oczywiście prywatnych zmiennych w interfejsie.
  - *final* zmienna musi być **zainicjalizowana** z wartością (nie może być np *int a*; Musi być *int a = 1*; i tej stałej nie można zmienić nigdzie indziej
  - *static* zmienna ta jest tez **statyczna** a więc jest widoczna w każdej klasie implementującej ten interfejs

1.6 Deklaracje metod i zmiennych

Modyfikatory dostępu

Modyfikatory dostępu dotyczą tylko zmiennych klasowych. Zmienne lokalne nie moga miec modyf. dostepu.

Modyfikatory dostepu do klas i zmiennych klasy rozpatrujemy dwojako:
i. Dostęp kodu klasy A do membera klasy B oznacza że member klasy B jest widoczny w A (trzeba też pamiętać o widoczności samych klas)
ii. Modyfikatory te definiują też możliwość odziedziczenia danego membera z nad-klasy.

public - taki member jest widoczny w każdej innej klasie niezaleznie w jakim jest pakiecie (*Clazz.a*)
-(pod warunkiem ze klasa sama w sobie jest widoczna w dla innej klasy - patrz

default access dla klasy)
- także taki member zostanie odziedziczony przez podklasę jako public (tak samo jakby ta klasa go miała w sobie zadeklarowanego).

private - taki member jest widoczny tylko w tej klasie w której jest zadeklarowany, próba dostania się z innej klasy (*Clazz.a*) spowoduje błąd kompilatora
- także taki member nie będzie odziedziczony a więc w podklasie też nie widać tego membera.
* overriding technicznie jest niemozliwy, ale w podklasie może być taka sama metoda - ale nie jest to overriding tylko po prostu nowa metoda.

default - member **jest** dziedziczony (wraz z modyf. *default*), ale **podklasa dziedzicząca musi być w tym samym pakiecie** co klasa posiadająca tego membera
- member jest widoczny (do użycia przez ref.) tylko w klasach z **tego samego pakietu** co klasa posiadająca tego membera

protected - member **jest** dziedziczony, a **podklasa dziedzicząca może być w innym pakiecie**
- member jest widoczny (do użycia przez ref.) tylko w klasach z **tego samego pakietu** co klasa posiadająca tego membera

Czyli: Dla default i protected **użycie** takich memberow jest mozliwe tylko **w tym samym pakiecie** (i nie ważne czy klasa dziedziczy czy nie). Druga sprawa to dziedziczenie. Membery protected **sa dziedziczone** w **tym samym pakiecie**, a **dodatkowo** w *protected* takze jesli klasa dziedzicząca jest w innym pakiecie.

Modyfikatory inne

final metody : - **zabezpiecza przed nadpisaniem** (override, przedefiniowaniem) tej metody w klasie dziedziczącej (pod-klasie)
final argumentu metody : - zabezpiecza przed zmianą wartości podczas działania całej metody (np. *metoda(int final a){..}* oznacza ze zadane "a" nie zmieni sie w metodzie)
final zmiennej : - zabezpiecza przed zmianą wartości tej zmiennej (czyli de-facto stałej) - deklarowanie stałych
- modyfikator *final* to jedyny modyfikator którym można poprzedzic zmienna lokalną

abstract metoda : - metoda taka nie ma implementacji (nie może mieć {} tylko na końcu ;) - zcedowanie **obowiązku** implementacji na pod-klasę.
- metoda abstrakcyjna może sie znaleźć tylko wklasie abstrakcyjnej (lub interfejsie ktory jest de-facto abstrakcyjny)
- klasa dziedzicząca **musi** zaimplementowac metode abstrakcyjną (chyba że sama również jest abstrakcyjna to może ale nie musi, wtedy obowiązek znowu spada niżej)
- UWAGA nie dać się złapać na przeładowanie metody (overload) czyli metody z innym argumentem czy "zwrotem" (patrz str 44 na dole)
- **nie można łączyć *abstract* i *private*** oraz ***final* i *abstract*** te modyfikatory wykluczają sie logicznie!
- **nie można łączyć *abstract* i *static***
- dotyczy **tylko metod i klas**

static metoda **-** metoda o zasięgu klasowym

- static method can't directly access an "instance variable" from the class it's in, because it doesn't have an explicit reference to any particular instance of the class
- błąd kompilatora: "*Cannot make a static refference to the non-static field"*

synchronized - dotyczy **tylko metod**
- taka metoda może być dostępna (wywoływania) tylko przez jeden **wątek** w **tym samym czasie**.
- można go mixować z wszystkimi czterema modyfikatorami dostępu, ale **nie może się łączyć z abstract**

native - dotyczy tylko metod
- mówi że metoda jest implementowana w **kodzie platform-dependent** (np w C) - **nie jest wymagana** na egzamin
- metoda *native* nie może być jednocześnie *abstract* ani *strictfp*
- musi kończyć się ";"

strictfp - dotyczy metod i klas.
- mówi że wszystkie **operacje zmiennoprzecinkowe** w danej metodzie lub klasie będą zgodne ze standardem IEEE 754 - nie jest wymagana na egzamin
- nie może być użyty razem z *abstract* ani *native*

Var-args methods
Argument - podczas wywołania metody: *metoda (2, "s");* --> To są **argumenty**
Parametr - podczas definicji metody: *void metoda (int a, String s){..}* --> to są **parametry**

- Począwszy od **Javy 5** możemy zadeklarować, że **metoda akceptuje dowolną liczbę argumentów** *(zero lub wiele)* pewnego ustalonego typu, przy czym dopuszczone są wszystkie typy, zarówno prymitywne jak i obiektowe.
- Deklaracja: po nazwie typu umieszczamy 3 kropeczki, po czym podajemy nazwę zmiennej, która de facto będzie tablicą.
  - np *void metoda (int... a){ }* lub *void metoda(int ... a){ }*
  - może byc tylko **jeden** parametr typu var-arg i musi on wystapić jako **ostatni** na liście parametrów.
  - *metoda(int... a){ }* jest ok ale *metoda(int... a, float b){ }* już nie

CODE:
```
public class AnyClass {
 void testFun() {
 fun("any", 1, 2, 3, 4);
 }

 void fun(String str, Integer... ints) {
 System.out.println(str + ints.length);

 for (Integer x : ints) {
 System.out.println(x);
 }
 }
}
```

1.7. Deklaracja konstruktora

- Konstruktor jest **zawsze** wykonywany nawet gdy nie jest jawnie zadeklarowany (gdy nie zdefiniowano żadnego konstruktora kompilator generuje sam bezargumentowy)
  - konstruktor wygenerowany przez kompilator wywołuje konstruktor bezparametrowy klasy nadrzędnej, a więc 'super()'
    - w takim przypadku klasa nadrzędna musi mieć zdefiniowany jawnie konstruktor bezargumentowy.
- Konstruktor **nie może nic zwracać**
- Konstruktor musi miec taką samą **nazwę jak klasa**
  - ale uwaga!!! metoda także może się nazywać dokładnie tak jak klasa, więc aby orzec czy to konstruktor trzeba spojrzeć na zwracany typ (jego brak)
- Może przyjmowac dowolna ilość argumentów, także var-args
- Może mieć **dowolny** (jeden z 4) **modyfikator dostępu**
- **Nie że być**: *abstract, final (*bo **nie można ich przedefiniować**) ani *static* bo dotyczą instancji
- **Nie są dziedziczone**

Ciekawostka:
Zdawałoby się, że skoro konstruktory nie podlegają dziedziczeniu, to nie ma różnicy między zakresem *protected* i zakresem *domyślnym*. Poniższy przykład pokazuje, że jednak taka różnica **jest**. Jeśli poniższe klasy są w różnych pakietach, to kod jest poprawny dla konstruktora protected, a dla zakresu domyślnego już nie.

```
package a;
public class NewClass {
 protected NewClass() {
 }
}

package b;
public class OtherClass extends NewClass {
 public OtherClass() {
 super();
 }
}
```

1.8. Deklaracje zmiennych

Są dwa rodzaje zmiennych w Javie:
- prymitywne: *char, boolean, byte, short, int, long, double* i *float*
- referencyjne: dostęp do obiektu danego typu lub podtypu.

Primitive variables

Wszystkie 6 typów liczbowych w Javie są *signed*: mogą być + i -
Najbardziej znaczący bit oznacza znak (+/-) liczby (0 to plus a 1 to minus)

Wielkości
byte **8 bitów**, od -2^7 do 2^7-1

short **16 bitów** od -2^{15} do $2^{15}-1$
int **32 bity** od -2^{31} do $2^{31}-1$
long **64 bity** od -2^{63} do $2^{63}-1$
float **32 bity** od n/a do n/a
double **64 bity** od n/a do n/a

boolean *VM-dependent* bitów, true/false
char 16 bitów

Można deklarować zmienne w jednej deklaracji:
```
int a = 0, b, c = 2;
Integer i = 5, j;
```

Reference variables
np Object o;

Instance variables (pola klasy)
Czyli zmienne obiektowe definiowane wewnatrz klasy ale na zewnatrz metod.
Są inicjowane w tylko gdy gdy jest inicjowana
Inicjowane są na domyślne wartości;

- mogą mieć wszystkie z czterech modyfikatorów dostępu
- **moga być** *final* i *transient* (transient: nie są serializowane i ich stan nie jest odtwarzany przy deserializacji)
- final reference variables **must be initialized** before the constructor completes
- **nie mogą być** abstract, synchronized, native ani strictfp bo są to modyf. dla metod
- dodając mod. static staje się to zmienna o zasięgu klasowym
- przyjmują defaultowo jakieś wartości (np null)

Local variables
- Czyli zmienne zadeklarowane wewnatrz metod.
- Jest niszczona gdy metoda sie kończy.
- Jest odkładana **na stosie** (stack) a nie stercie (heap)
  - ale jeśli jest to zmienna referecyjna to sama zmienna jest na stosie ale juz obiekt na ktorym wskazuje jest na stercie.
- Może być oznaczona tylko jako *final* - inen modyfikatory (także dostępu) **nie moga być użyte**
- **musi** być zainicjalizowana przed uzyciem - błąd kompilatora "*The local variable might not have been initialized*"
  - zmienne lokalne nie przyjmują defaultowo żadnych wartości.
- mają ograniczony zasięg - nie są globalne.
- *przesłanianie (shadowing)* czyli użycie zmiennej o tej samej nazwie co zmienna klasowa lub instancji jako zmiennej lokalnej.
  - ważniejsza jest zmienna lokalna
  - aby się dostać do zmiennej obiektu po przesłonięciu trzeba użyć *this*
  - przesłanianie jest tylko w przypadku zmiennych (pól) klasy. Nie można zadeklarować wewnątrz metody 2 zmiennych tak samo nazywajacych sie.

Zmienne tablicowe

- Tablice mogą trzymac obiekty tego samego typu (bądź podtypów) lub prymitywy
- Tablice są trzymane na stercie
  - nie ma "tablic prymitywnych" a jedynie mogą być "tablice z prymitywami"
- Deklarowanie za pomocą []:

int[] key; (recommended)
int key [];
Object[] o; (recommended)
Object o [];
String[][][] occupantName; (wielowymiarowa - czyli tablica tablic)
String[] ManagerName []; (tak też można zadeklarować wielowymiarową)

- **nie można** określać rozmiaru tablicy podczas deklaracji: int[5] tab; **ŹLE** (tak to tylko w C/C++ :))
  - robi się to podczas inicjalizacji (instancjonowania): int[] tab = new int[5];

Zmienne (STAŁE) final
Omówione w 1.6...
- zmienne prymitywne: raz zdefiniowana nie może zostać zmieniona.
- zmienne referencyjne: nie można zmienic żeby wskazywał na inny obiekt - sam obiekt wewnątrz (jego pola) może być zmieniony
  - **nie ma "obiektów final" a jedynie "referencje final**"

Zmienne volatile
Zmienna ta oznacza że wątek chcący mieć dostęp do takiej zmiennej musi uzgodnić swoja własną kopię z kopią główną... **to nie wymagane**

Zmienne volatile jak i transient moga byc użyte tylko w przypadku zmiennych instancyjnych

Zmienne i metody statyczne
Zmienne i metody statyczne istnieja niezależnie od instancji i są dostępne na poziomie klas a nie konkretnych instancji. Zmienne takie np są inicjowane zanim zostanie zainicjonwana jakakolwiek instancja danej klasy.

static mogą być:
- metody, zmienne
- klasa zagnieżdżona w innej klasie (ale nie w metodzie)
- initialization blocks

nie moga za to być: konstruktory, klasy (chyba ze zagniezdzone), interfejsy, lokalne zmienne, Method local inner classes, Inner class methods and *instance variables* (bo to juz nie instance variable tylko static variable)

1.9. Deklaracja typu wyliczeniowego: enum

- najprostrza deklaracja enuma: *enum CoffeeSize { BIG, HUGE, OVERWHELMING }* (duze litery to tylko "konwencja" nie obowiązek)
  - na koncu deklaracji **może byc ale nie musi** '**;**' czyli można też tak: *enum CoffeeSize { BIG, HUGE, OVERWHELMING }**;***
- enum **może być** zadeklarowany **na zewnatrz** jako osobna "klasa" , w pliku, w interfejsie albo wewnątrz innego typu wyliczeniowego
- enum **może być** zadeklarowany **wewnątrz** (jako member) innej klasy
  - Do takiego enuma dostajemy się jak do każdego innego membera o ile modyf. na to pozwala: Klasa.Enum.VARIABLE;

- enum **nie może być** zadeklarowany wewnatrz metody

- dostępne **modyfikatory dostepu** dla *enum* dla deklaracji bezpośrednio w pliku w pliku *(*to: *public* oraz *default* oraz strictfp(?))
  - dla deklaracji wewnątrz innych typów dodatkowo można użyć *private, protected i static*

- stałe wyliczeniowe **są takiego typu, w jakim je zadeklarowano** i są to jedyne instancje tego typu. W powyższym przykładzie stałe *Size.SMALL, Size.LARGE i Size.HUGE* są zatem typu *Size* i to **jedyne** instancje tego typu jakie kiedykolwiek będą występowały w przyrodzie
  - Oznacza to między innymi, że możemy używać operatora == zamiast operacji equals()

Zaawansowany enum:
Enum może posiadac prócz stałych jeszcze konstruktory, metody, pola klasy wewnętrzne, inne enumy oraz *constant specific class body*

- **nie można nigdy** wywołać konstruktora - jest on wywolywany przez JVM podczas wywolania do stałej
- można mieć konstruktowy wielo-argumentowe i można je przeciążać
- można definiować coś co przypomina "wewnętrzną klase anonimową" -czyli *constant specific class body*
  - pozwala np na przeciążenie metody wykonywane dla konkretnej "stałej enumeracji".

```
enum CoffeeSize {
  BIG(8), // tu zostaje wykonany konstruktor, (i zapisana zmienna private
ounces )
  HUGE(10), // tu tez
  OVERWHELMING(16) { // start a code block that defines
                     // the "body" for this constant
        public String getLidCode() { // override the method
                                     // defined in CoffeeSize (ponizej)
          return "A";
        }
   }; // <-- the semicolon is REQUIRED when you have a body - tak jak w anon.
class

   CoffeeSize(int ounces) {
      this.ounces = ounces;
   }
   private int ounces;  // ---> prywatna zmienna enuma

   public int getOunces() {
      return ounces;
   }
   public String getLidCode() { // this method is overridden
                                // by the OVERWHELMING constant

      return "B"; // the default value we want to return for
                  // CoffeeSize constants
   }
}
```

- konstruktor w enum **może być tylko** private

- defaultowo jest private i nie trzeba pisac tego explicite, ale można.
- **na początku enumeracji muszą być wartości enumeracji** - stałe wyliczeniowe (BIG, HUGE etc) i
- w przypadku **gdy dalej pojawia sie "coś jeszcze"** to ciąg enumeracji **musi** kończyć sie **';'**
- ZAWSZE!!!! nawet jak w enum nie ma zadnych stałych a jedynie "cos jeszcze": *enum moje { **;** public String getDesc(){return "foo";} }*
- w powyższym przykładzie można zaobserwować jak w 1 enumeracji można trzymac kilka wartości, oraz jak przypisac jakąś wartość do enumeracji (np *BIG(3)*)

- typy wyliczeniowe **mogą implementować interfejsy**,
- enumy **nie mogą dziedziczyć z innych typów** wyliczeniowych
- ale typy wyliczeniowe również należą do wspólnej hierarchii z klasą Object w korzeniu :)
- typ wyliczeniowy E dziedziczy bowiem implicite z typu Enum<E>, a znów ten z Object

- enum może mieć metodę abstrakcyjną ale wtedy wszytskie stałe wyliczeniowe muszą ja implementować (i nie może nie byc stałych wogole):

```
enum Size {
 SMALL {
 public String getDescription() {
 return "Taki calkiem maly";
 }
 },

 LARGE {
 public String getDescription() {
 return "Dosyc duzy";
 }
 };
         // 1) patrz: jest na koncu, 2) stale implemenują ją
 public abstract String getDescription();
}
```

Domyślne metody każdego typy wyliczeniowego (enuma):
Jako że enum E dziedziczy z Enum<E> to każdy enum posiada nastepujące metody:
- values() - zwraca **tablicę[]** stałych wyliczeniowych zdefiniowanych w typie E
- valueOf(String name) - służy do konwersji z typu String – zwraca instancję typu E (jedną ze stałych wyliczeniowych) o podanej nazwie

---
TEST 1:
moj wynik 7/9, pytania 3-7 OK
źle:
1. dałem E ma być C,D- nie zauwazylem ze w klasie implementujacej brakuje *public* a w interfejsie wszytskei domyslnie sa public wiec i w klasie impementujacej musi byc public przy metodzie. Ponadto niewiedzialem ze klasa abstrakcyjna moze implementowac interfejs (a przeciez moze!! moze miec implementacje metod roznych) - czyli odp C, D
2. dałem A ma być B - dałem sie złapać że metoda konczaca sie na ; musi byc abstract a klasa abstract nie musi miec zadnej metody abstract a tylko zwykłe... oczywiste!
8. dałem C ma być A - dalem sie zlapac ze zmienna w enumie nie musi być *private* explicite.
co ciekawe moze byc public ale i tak nie da sie do niej dostac - tylko do stałych

wyliczeniowych.
POPRAWIĆ
---

2. Obiektowość

Co wyróżnia dobry program obiektowy:
"hermetyzacja (ang. encapsulation), zależność (ang. coupling), która im prostsza tym lepsza (ang. loose coupling) i spójność (ang. cohesion) która z kolei powinna być możliwie duża (ang. high cohesion)."

-

2.1. Enkapsulacja (encasulation) , zależności (coupling), spójność (cohesion)

Encapculation
Jest wewnętrzną sprawą klasy jak jej mechanizmy zostały zaimplementowane i poprzez nie ujawnianie tych szczegółów chcemy zagwarantować, że sposób implementacji będzie można w przyszłości zmienić bez obawy o konieczność modyfikacji innych klas

Klasy powinny być **hermetyczne**. Oznacza to, że klasy powinny ukrywać swoje wewnętrzne struktury i operacje a udostępniać tylko te, dla których udostępnienia zostały powołane, tj. swoje dobrze określone interfejsy

A Więc:
- ukrywaj zmienne instancji (w klasie) - najczęściej jako private
- modyfikacja zmiennych przez accesory (metody set i get)
    - używaj konwencji nazweniczej dla Java Beans

Cohesion
- Wymóg spójności oznacza, że klasy powinny implementować dobrze określony, **wąski zakres funkcjonalności**.
    - Oznacza to, że tam gdzie tylko ma to sens należy używać delegacji odpowiedzialności

Coupling
Klasy **nie powinny być we wzajemnych skomplikowanych zależnościach** (loosly coupled). Kod jednej klasy nie powinien wykorzystywać cech innej klasy, które wynikają z jej konkretnej implementacji a nie są elementem zaprojektowanego interfejsu. Zagadnienie to jest ściśle powiązane z hermetyzacją w tym sensie, że skomplikowane zależności mogą wystąpić tylko wtedy, kiedy tej hermetyzacji brakuje. W kontekście egzaminu SCJP, ale nie tylko ważne jest, aby zrozumieć różnicę – różnicę między stanem faktycznym istnienia ścisłych zależności a brakiem hermetyzacji, który jedynie umożliwia istnienie tych zależności, ale niczego nie implikuje.

2.2. Dziedziczenie, relacje IS-A i HAS-A

Dziedziczenie
- Reusability kodu
- Polimorfizm - Jest to coś co wynika z dziedziczenia, obiekty klasy pochodnej sa takze obiektami nadklasy - wielość form.
- Można wywolywać metody np przyjmujące jako parametr klase Parent z argumentem jako obiekt Child który dziedziczy po klasie Parent.
- można wywoływac w tym przypadku tylko metody należące do Parent - obiekt Parent bowiem wie tylko o swoich metodach a nie dziecka.

Relacja IS-A

Polega na **dziedziczeniu klasy** lub **implementacji interfejsu**
- formalnie: jeśli wyrażenie '(new A()) instanceof B' ma wartość 'true'.
- np Subaru IS-A Car - *class Subaru extends Car{}* oraz np Subaru IS-A Vehicle gdy dodatkowo *class Car extends Vehicle { }*
- Vegetable IS-A Eatable gdy np *class Vegetable implements Eatable { }*

Relacja HAS-A

Relacja zachodzi gdy Klasa A posiada referencję do instancji klasy B
- A HAS-A B gdy *class A { private B b; }*
- z tą relacją łączy się tez minizagadnienie delegacja odpowiedzialności - gdy np w klasie A jest B oraz metoda test() a w niej wywolana metoda na obiekcie B z podobnymi parametrami. - Typowa sprawa z zagniezdzaniem metod.

2.3. Polimorfizm

Java object that can pass more than one IS-A test can be considered polymorphic.

- A reference variable can refer to any object of the same type as the declared reference, or it can refer to any **subtype** of the declared type!
- Można dziedziczyc **tylko od jednej klasy** bezpośrednio.
- Co gdy chcemy aby tylko pewne podklasy miały pewne metody a inne nie? Wtedy najlepiej stworzyć ***interface*** ktory bedzie implementowany tylko przez niektore podklasy. Można implementowac wiele interfejsów.
- W przypadku implementowaniu interface równiez mamy polimorfizm:
Klaska k = new Klaska();
Podklaska k1 = k;
Interfejsik i = k;

Powyższa zgodność wzgledem przypisania to ang. assignment compatible

- W przypadku gdy podklasa "overrides" metodę nadklasy:
oraz Klaska k = new Podklasa();
to wysołanie k.metodka(); wywoła metode podklasy a więc rzeczywistego obiektu który jest wskazywany pod tą referencja (k).
Są tu dwie kwestie:
- jakie metody kompilator widzi w momencie gdy chcemy jakas wywolac na nadklasie:
- np KlaskaA ma metodę a() a podklasa KlaskaB metodę b() (i odziedziczoną a) to w przypadku KlaskaA a = new KlaskaB(); mozna wywolac na obiekcie a tylko metodę a.
- można to obejść stosując rzutowanie: *((KlaskaB)a).b()* ale to juz inna bajka
- natomiast wywołując ta metodę wywołujemy już nie tę zaimplementowaną w

KlaskaA tylko te na która wskazuje referencja czyli (tutaj) KlaskaB
- jest to "***virtual method invocation***"

Polimorfizm dotyczy tylko metod i to tylko metod instancji, tj. nie statycznych
- gdy metoda nadklasy stała by się *static* to zaqwsze będzie wywołana metoda tego typu zmiennej - przestaje się bowiem liczyć typ obiektu, ważny jest typ zmiennej

Jeżeli natomiast podklasa nie bedzie miala implementacji naszej metody to, pomimo ze referencja wskazuje na podklase
KlaskaA a = new KlaskaB();
to wywolanie a.naszaMetoda(); spowoduje wywolanie metody zaimplementowanej w **nadklasie** - pomimo odziedziczenia

Inna ciekawostka:

```
class A {
 void test() {
 System.out.println(getId()); // tutaj getId() jest de facto this.getId() a
wiec wykonuje swoje getId
 }

 String getId() {
 return "A";
 }
}

class B extends A {
 String getId() {
 return "B";
 }
}
```

Nastepnie wywolania:

```
A var = new A();
 var.test();
var = new B();
 var.test();
```

Tutaj program wypisze A a potem B. Metoda getId() wykonuje się przecież dla obiektu this, a obiektem tym za drugim razem jest obiekt klasy B.

2.3. Nadpisanie (Przedefiniowanie) metod: overriding

Overriding - Możliwość przedefiniowania metody w pod-klasie, będącej zdefiniowaną w nad-klasie.
- możliwośc taka nie istnieje gdy metoda a nad-klasie jest oznaczona jako *final*

Reguły overidding'u:

- Metoda przedefiniowująca **nie może mieć bardziej restrykcyjnego modyfikatora dostępu od metody przedefiniowywanej**
  - nie można np przedefiniowac petody publicznej i oznaczyć ją np protected
  - ze wzgledow bezpieczenstwa (gdy nadklasa ma metode to nie może zaistniec sytuacja ze polimorfizm nie bedzie mogl wywolac metody na obiekcie pod-klasy bo ona jest

prywatna)
  - takie wywołanie wywoła błąd kompilatora
- Metoda przedefiniowująca **może mieć mniej restrykcyjny modyfikator dostepu** od metody przedefiniowywanej
- Liczba i typy parametrów metody muszą byc **takie jak** w metodzie przedefiniowywanej (nadklasy) - w przeciwnym razie będzie to *overloading* a nie *overriding*
- Zwracany typ metody musi być **taki sam lub być typem pochodnym**
- Przedefiniowanie dotyczy tylko w przypadku dziedziczenia - patrz: protected i default
  - można przedefiniowac metodę odziedziczoną
- The overriding method CAN throw any unchecked (runtime) exception, regardless of whether the overridden method declares the exception. (More in Chapter 5.)
- Metoda przedefiniowująca **nie może rzucać (*throws)* innym ani szerszym wyjątkiem** jak metoda przedefiniowywana *(bo to juz nie bedzie overrriding)*
  - np metoda przedefiniowywana rzuca FileNotFoundException to metoda przedefinowujaca nie może rzucic SQLExcetpion (inny) ani Exception (szerszy)
  - takie błędne wywołanie wywoła błąd kompilatora
- Metoda przedefinowująca **może rzucić (*throws)* węższym wyjątkiem** (pod-wyjatkiem tego z metody przedefiniowywanej) **lub mniejszą liczbą** wyjątków
  - dlatego że metoda przedefiniowywana przejmuje odpowiedzialność za wyjątek - przeciez kompilator widzi na początku metode nad-klasy (referencji a nie obiektu)
    - podczas gdy mamy referencje *Nadklasa a = new Podklasa()* i metoda w podklasie nie rzuca wyjątków to wywolujac ja i tak musimy ten wyjatek obsluzyc dlatego że kompilator widzi metodę nadklasy wiec wymaga obslugi wyjątku - pomimo ze de facto zostanie wywolana metoda Podklasy.

- **Nie można** przedefiniowac metody ***final* -** temu służy final, taka próba wywoła błąd kompilatora
- **Nie można** przedefiniowac metody ***static*** - takie metody nie podlagają overriding'owi - nie będzie błedu ale metody nie zostana przedefiniowane

Użycie metody z nad-klasy w metodzie przedefiniowującej:

```
public void metodka(){
    // może byc jakis kod
    super.metodka();
    // może być dalej jakiś kod
}
```

Po to aby kompilator powiedział nam wcześniej czy metoda ktorą piszemy jest poprawnie przedefiniowana można posłużyć się adnotacją ***@Override***
Jeśli umieścimy ją nad jakąś metodą to kompilator sprawdzi, czy rzeczywiście przedefiniowuje ona jakąś metodę z nadklasy i zgłosi błąd kompilacji, jeśli jest inaczej.
Ale adnotacji **nie ma na egzaminie**

2.4. Przeciążanie metod: overloading

Overloading - Możliwość uzycia metody o tej samej nazwie ale:
  - innych parametrach (liczba, typy)
  - ewentualnie **dodatkowo** innym typie zwracanym

Reguły przeciążania overloading'u:

- Metoda przeciążająca **musi zmienić liste argumentów** (liczbę lub typy parametrów)
- Metoda przeciążająca **może dodatkowo** zmienić zwracany typ (ale przy zmienionej juz liście argumentów)
- Metoda przeciążająca **może dodatkowo** zmienić modyfikator dostępu
- Metoda przeciążająca **może dodatkowo** zmienić rzucane wyjątki (także na nowe i szersze)
- Metoda przeciążająca może byc w tej samej klasie lub w podklasie (w podklasie nadal można przedefiniować metodę - to się nie wyklucza)

Wywoływanie metod przeciążonych:
- Wybór którą metodę wywołac **zależy od listy argumentów**
- Gdy wywołujemy metodę z parametrem referencyjnym wybór metody nastąpi **na podstawie typu referencji** a nie rzeczywistego obiektu na jaki wskazuje

Przykład

- np: **gdy** są 2 metody (1) *void metoda(Nadklasa a){ ... }* oraz (2) *void metoda(Podklasa b){ ... }*

to gdy mamy *Nadklasa a = new Podklasa();* i wywołamy *metoda(a);* wywołamy metodę numer (1) bo referencja a jest typu Nadklasa

- **gdyby jednak** było: *Podklasa a = new Podklasa();* to powyższe wywołanie uruchomiło by metodę numer (2) bo referencja jest tu typu Podklasa

- w tym przypadku na wybór uruchamianej metody polimorfizm nie ma wpływu ale mimo to obiektem jaki zostanie wpuszczony do metody jest obiekt rzeczywiscie wskazywany przez referencje a wiec Podklasy.

- Gdy natomiast argumenty przeciazonych metod nie koliduja ze sobą typami dziedziczonymi (jak w poprzednim przykładzie) to po wywołaniu *metoda(Nadklasa a){}* z argumentem typu Podklasa (*metoda(podklasa)*;) to oczywiście zostanie wywołana metoda z podanym obiektem podklasy.

- oczywiste jest też że odwrotna sytuacja nie zadziała

Ciekawy przykład:

```
class Test {
 public static void main(String[] args) {
 Test obj = new Test();

 System.out.println(obj.getString(1));
 }

 String getString(float f) {
 return "float value: " + f;
 }

 String getString(long l) { // to bedzie wywolane
 return "long value: " + l;
 }
}
```

WYNIK: "**long** value: 1"

DLACZEGO?
Kompilator jak i JVM najpierw stara się sprowadzić liczbę 1 do inta. Następnie szuka **pierwszej pasującej metody**. Nie ma dla int więc w kolejnym kroku szukamy najlepiej pasującej (long), a potem (ustalone eksperymentalnie) jest float, double. Short jako mniej dokładny wymaga rzutowania podobnie char.

Ciekawy przykład: str 110, **tabela** pokazuje jak działa łączenie overriding i overloading oraz gdy w gre wchodzi polimorfizm.
Ciekawa tabela porównawcza: str 111 - porównuje overriding i overloading

Podsumowanie 2.2. i 2.3.: trzeba uważać bo to że coś nie jest poprawnym overriding nie oznacza ze sie nie skompiluje bo moze to być poprawny overloading. Na egzaminie trzeba na to zwrocic uwage i bacznie rozpoznac co jest co. Czasem gdy nie jest to overriding ani overloading (np inny wyjatek ale lista arg ta sama) to wystapi błąd kompilatora.

2.5. Rzutowanie typów referencyjnych: casting

Rzutowanie "w dół" (downcasting):
Animal a = new Dog();
(**(Dog)a**).bark(); <--- rzutowanie na Dog w dół drzewa dziedziczenia

- kompilator jest zmuszony **zaufać** programiście że rzutowany obiekt jest faktycznie tym obiektem na który rzutujemy.
  - wyjątek: ClassCastException rzucony podczas wykonania... (może zostać wyłapany try ale to bez sensu)
    - podczas kompilacji może być OK bo ClassCastException wywala sie podczas wykonania
  - aby przetestować czy referencja wskazuje na obiekt jakiegoś typu warto użyć **instanceof**

```
if (a instanceof Dog) {  /* wejdzie tu jeśli 'a' będzie obiektem klasy Dog */  }
```

- w oczywistych przypadkach kompilator zaalarmuje błąd castowania w momencie kompilacji
  - gdy np chcemy zrzutować niekompatybilne obiekty np (String)animal
- można rzutować na interfejs: *Pet d = new Dog();* // gdy Pet to interfejs implementowany przez Dog
  - możliwe gdy dana klasa implementuje ten interfejs
  - dotyczy to też podklas które dziedziczą z klasy implementującej interfejs (bo klasa "dziedziczy" tez interfejs)

Rzutowanie "w górę" (upcasting)
- jest de facto ograniczeniem możliwości podklasy.
- Ponizszy przykład nie spowoduje wykonania metod nadklasy lecz **podklasy Dog**:

```
Dog d = new Dog();
((Animal)d).eat();    <--- wykona się metoda eat() dla Dog a nie Animal
```

Jest robione podczas polimorfizmu implicite.
- Upcasting implicite:

```
Dog d = new Dog();
 Animal a1 = d;
```

- Upcasting explicite:
```
Animal a2 = (Animal) d;
```

2.6. Implementacja interfejsów

Klasa implementująca interface (implements) :
- musi dostarczyć konkretne (nieabstrakcyjne) implementacje metod z deklarowanego interfejsu
  - chocby puste { }
- musi przestrzegać zasad poprawnego przedefiniowania (override)
- deklaracja implementowanej klasy musi mieć taką sama liste arg, typ zwracany
- **nie musi** rzucac wyjatkiem
  - ale jeśli rzuca - to tak jak przy overridingu - nie może rzucać **innym ani szerszym wyjątkiem**

- metoda implementująca **może** być abstrakcyjna - a wtedy nie musi explicite implementować tych metod
  - odpowiedzialność implementacji spoczywa wówczas na podklasie.
  - w tym przypadku także klasa abstrakcyjna może ale nie musi implementować wszystkich metod

- klasa **może implementować więcej jak jeden** interfejs
  - wtedy wszystkie metody ze wsyztskich interfejsów muszą mieć implementację w tej klasie

Interfejs **może dziedziczyc** po **więcej niż jednym** interfejsie ale **nie może go implementować**
  - interface Interfejs extends Interfejs2 { } ; interface Interfejs extends Interfejs2, Interfejs3 { }
  - wtedy klasa implementująca **musi** dostarczyc implementacje metod ze **wsyztskich rozszerzanych interfejsów**

2.7. Zwracanie typów

- można uzyć `return null;` ale w deklaracji metody musi byc jakis typ obiektowy
- można zwrócić tablice `public String[] metoda() { }`
- gdy w deklaracji jest zwracanie *prymitywa* można zwrócić wartość lub zmienną **którą można skonwertować implicite** na taki prymityw

```
public int foo() {
    char c = 'c';
    return c; // char is compatible with int
}
```

  - można też zwrócić wartość dającą się explicite skonwertować (castować) na taki prymityw
- **można** zwrócić wartość **dającą sie skonwertować (castować) implicite** na zwracany w deklaracji obiekt

```
public Animal getAnimal() {
    return new Horse(); // Assume Horse extends Animal
}
public Object getObject() {
    int[] nums = {1,2,3};
    return nums; // Return an int array, which is still an object

}
```

- **nie można** użyć słówka *return* gdy metoda jest *void*

2.8. Konstruktory i tworzenie obiektów (instantiating)

- podczas tworzenia obiektu **zawsze wywoływany** jest konstruktor
  - podczas tworzenia obiektu **wywoływane są też konstruktory superclass**
- konstruktor jest wykonywany po uzyciu słowa **`new`**

Podstawy

- **Każda** klasa (także abstrakcyjna) **posiada** konstruktor. Konstruktor posiadają tez **enum**, ale **nie** interfejsy (one nie są w drzewie dziedziczenia)
  - nie zawsze jest on napisany jawnie przez programiste
- Konstruktor **nie może zwracać ani deklarować** zwrotu (**nawet** void'a)

- **konstruktory nie są dziedziczone** ani **nie moga byc overriden**

- **MOŻE być w klasie metoda o tej samej nazwie ale nie będąca konstruktorem**
  - to czy jest to konstruktor można rozpoznać po zwracanym typie - Konstruktor nie ma zwrotu

- Nazwa konstruktora musi być **identyczna z nazwą klasu**
- **Gdy nie ma** jawnego konstruktora, JVM tworzy automatycznie bezargumentowy
  - **gdy jest stworzony** konstruktor bezargumentowy nie jest tworzony i jedynym jest ten utworzony przez programiste
- konstruktor domyślny jest bezargumentowy
- Konstruktory można przeciążać
- Konstruktor może mieć **którykolwiek modyfikator dostępu**
  - także **private** - oznacza że tylko kod wewnątrz tej klasy może instancjonować ten obiekt
- Jeżeli konstruktor **napisany przez programiste** nie bedzie posiadał wywołania konstruktora nadklasy *super()*, to kompilator **wstawi go sam** (zawsze !!!) na początku kodu konstruktora
  - Jeżeli naklasa nie będzie miała konstruktora bezargumentowego (bo programista zdefiniowal tylko argumentowy, a wtedy kompilator nie robi go sam) to będzie błąd kompilatora
    - wtedy **trzeba samemu wywołać konstruktor super(param) na początku** konstruktora - w przeciwnym razie błąd kompilatora
- wywołanie *super()* lub *this()* musi sie pojawić na poczatku konstruktora.
  - Nie można wywoływać konstruktora rekursywnie - tzn *this()* w konstruktorze bezargumentowym.

- można wywołać inny konstruktor np *this()* w *Konstruktor(String name)*
- klasa **abstrakcyjna** także posiada konstruktor i jest on wywołany w momencie instancji podklasy
- nie można wywołac konstruktora jawnie poza innym konstruktorem oraz tylko za pomocą **super** i **this**
- *super LUB this()* **muszą** być **pierwsze w konstruktorze** (nigdy oba naraz)
  - jak sie poda this() to super bedzie wywolane w innym konstruktorze - wywolanym de facto przez this()
- jako argument super() lub this() można podać **tylko zmienne lub metody statyczne tej samej klay** lub wartości "ustalone" (np "stringi", liczby: 1,2, 1.3 itp)
  - dlatego że nie można wywołać zadnej metody zanim nie będzie instancji obiektu
  - **można** natomiast użyć innego obiektu i jego metod normalnie, pamietajac ze super i this maja byc piwerwsze:
    Klasa Jablko: *super((new Pestka()) .getKolor());*

Ciekawe:

```
class Clothing {
    Clothing(String s) { }
}
class TShirt extends Clothing { }
```

Powyższy kod wywoła **błąd** - ponieważ default constructor TShirt posiada defaultowe wywołanie super() a nie istnieje w Clothing taki konstruktor bezarg.

Konstruktor domyślny:
- Jest tworzony **tylko wówczas** gdy programista nie napisał żadnego własnego konstruktora
- KOnstruktor domyślny posiada taki sam modyfikator dostępu jak klasa
- Nie posiada argumentów
- **Posiada** wywołanie do konstruktora nadklasy super()

Wywołanie konstruktorów

Hierarchia: *Object --> Animal --> Horse*
Tworzenie *Horse h = new Horse();*

1. Wywołany jest konstruktor **Horse**
2. Wywołany jest konstruktor **Animal**
3. Wywołany jest konstruktor **Object**
4. Zmienne instancji Object są instancjonowane z wartościami podanymi przy deklaracji (int x = 5;)
5. Konstruktor Object zakańcza się
6. Zmienne instancji (jeśli są takowe) Animal są instancjonowane z wartościami podanymi przy deklaracji
7. Konstruktor Animal zakańcza się
8. Zmienne instancji (jeśli są takowe) Horse są instancjonowane z wartościami podanymi przy deklaracji
9. Konstruktor Horse zakańcza się

"Slowami Mariusza L":
- wywoływany jest domyślny konstruktor klasy Child, jego pierwszą instrukcją jest wywołanie super()
- wywoływany jest domyślny konstruktor klasy Parent, jego pierwszą instrukcją jest wywołanie super()
- wywoływany jest konstruktor w klasie Object, ta nie ma już nadklas, więc po wykonaniu kodu konstruktora następuje powrót do konstruktora z klasy Parent
- wykonywana jest inicjalizacja zmiennych instancyjnych w klasie Parent, a więc przypisanie ‘str = "some value"’ po czym następuje powrót do konstruktora z klasy Child
- wykonywana jest inicjalizacja zmiennych instancyjnych w klasie Child, a więc przypisanie ‘childStr = "some other value"’ co jest ostatnim krokiem, obiekt został utworzony i zainicjalizowany

Przeładowanie (overload) konstruktorów

Konstruktory takie **różnią się lista argumentów**

- W przeładowanym konstruktorze także jest wywołanie super() chyba że programista sam wywoła inny (np this albo super(arg))

2.9 Zmiene i metody statyczne: static

- **statyczna metoda nie ma dostepu do niestatycznych** pól instancji (ponieważ nie ma takowej instancji) oraz niezstatycznych metod
  - *Przykład*
    Funkcja **main**: jest statyczna wiec nie można się odnieść do niestatycznych zmiennych instancji (klasy). Bo wtedy funkcja main (statyczna) nie wie do jakiego obiektu chcemy sie odnieść.
  - oczywiście sama może działać na obiekcie i na nim wykonywać metody
- wykonanie statycznej metody (czy dostep do statycznego pola) **nie powołuje obiektu do zycia** [np main()]
- w statycznej metodzie (w tym main) mozna używać normalnie metod statycznych

- metody statyczne **nie mogą być overriden** ale to nie to samo co przedefiniowanie metody statycznej w podklasie
  - różnica polega na tym iż nie będzie tu działać polimorfizm - **patrz przykład:**

```
class Animal {
  static void doStuff() {
    System.out.print("a ");
  }
}
class Dog extends Animal {
  static void dostuff() {         // it's a redefinition, not an override
    System.out.print("d ");
  }
  public static void main(String [] args) {
    Animal [] a = {new Animal(), new Dog(), new Animal()};
    for(int x = 0; x < a.length; x++)
     a[x].doStuff();               // invoke the static method
  }
```

```
}
```

Output: a a a

Dlaczego? Bo kompilator wie ze a to Animal więc wykonuje statyczna metode Animal.doStuff() - nie tak jak w polimorfizmie tutaj wazna jest nazwa klasy a nie obiekt na jaki wskazuje referencja. Referencja mówi tylko JVM na jakim obiekcie wywolujemy te statyczna metode.

Dostęp do metod i pól statycznych
- poprzez `NazwaKlasy.STATYCZNE_POLE` bez instancjonowania obiekty typu *NazwaKlasy*
- poprzez obiekt: `obiektKlasy.STATYCZNE_POLE` gdzie *obiektKlasy* jest obiektem powołanym do życia poprzez *new*

Nieważne którą z powyższych metod użyjemy, efekt jest ten sam - wykonanie statycznej metody na Klasie a nie Obiekcie.

Poniższy przykład wykona się poprawnie:
```
static void printMessage() { } // metoda klasy SomeClass
...
SomeClass obj = null;
obj.printMessage(); // nie będzie NullPointerException bo kompilator juz wie na jakiej klasie ją wykonać
```

2.10. Zależność (coupling) i spójność (cohesion)

Aplikacja powinna byc **łatwa w utworzeniu, utrzymaniu i rozszerzaniu**

Coupling
- miara w jakim stopniu jedna klasa wie o implementacji w drugiej klasie
  - jeśli klasa A wie o B tylko na podstawie interfejsu to jest to *loose coupling* (**dobrze**)
  - jeśli A wie o szczegolach implementacji B to jest to *tight coupling* (**źle**)
- silna zależność powoduje że zmiany szczegółów implementacji jednej klasy powodują błędy w działaniu drugiej klasy.
- **wymagana** głównie **enkapsulacja**

EGZAMIN: kilka pytań z oczywistym przykładem złej zależności

Cohesion

- miara jak wąskie zadanie ma dana klasa. Im węższe, bardziej sprecyzowane zadanie tym lepiej
- mocno spójne klasy łatwiej utrzymac i rzadziej zmieniać

Przykład
Klasa mało spójna (wiele różnych zadań w jednej klasie):

```
class BudgetReport {
```

```
  void connectToRDBMS(){ }
  void generateBudgetReport() { }
  void print() { }
}
```

Rozwiązanie - podzielenie funkcjonalności na wiele klas:

```
class BudgetReport {
  Options getReportingOptions() { }
  void generateBudgetReport(Options o) { }
}
class ConnectToRDBMS {
  DBconnection getRDBMS() { }
}
class PrintStuff {
  PrintOptions getPrintOptions() { }
}
```

Źródła inne: http://javaboutique.internet.com/tutorials/coupcoh/

TEST 2:
moj wynik 9/14, pytania OK: 2-4, 6, 8-10, 12, 14
źle:
1. dałem BE ma być B
5. dałem AF ma być EF
7. nie zauwazylem (); na koncu i nic nie dodalem w "gapy" w 2 klasie. 1 klasa ok
11. dałem E ma być D
13. dałem D ma być AB

POPRAWIĆ

Po poprawce 11/14
zle 3 (dalem ABE), 5 (dalem AF), 8 (dalem ACD)

---

3. Zadania

Stos (stack) i sterta (heap)

- Instance variables and objects live on the heap.
- Local variables live on the stack.

3.1. Literały

Reprezentacja prymitywnych danych w kodzie np 'b' - literał char, 42 - literał int, false - literał boolean, 2342.342 - literał double itd

Integer
Są 3 sposoby reprezentacji intów:
- dziesiętny: np *int l = 234;*
- ósemkówy: cyfry od 0 do 7; **zapis z zerem na początku**: *int six = 06; int nine = 011;* maximum 21 cyfr w systemie ósemkówym
- szesnastkowy (hex): znaki 0 - 9 i a - f (**małe lub duże** obojętnie); **zapis z 0x (lub 0X) na początku**: *int x = 0x0001; int y = 0X7fAf;*

Wsyztskie powższe literały w 3 systemach są domyślnie zapisane w *int* ale mogą zostac zapisane w ***long* dodając litere L** (lub l) na końcu
*long lo = 11102**L;***
*long so = 0xFFF**l;***

Floating-point - double i float

Literały zmiennoprzecinkowe domyślnie są zapisywane jako double (64 bity) z kropką:
double d = 123221.233212;

Dla double można dodać na końcu **D** (lub d) aby zaznaczyć że to double (ale nie jest to konieczne - D jest defaultowe)
double d = *123221.233212**D;***

Aby zapisać je jako **float** (32 bity) należy **dodać na końcu F (lub f)**
*float f = 23.34323**F;*** *float f2 = 33.232**f;***
- zapis bez **F** spowoduje błąd kompilacji: `float badFloat = 12.1212; //Compiler err:` Cannot convert double to float

Literały liczbowe **nie moga mieć przecinka -** tylko **kropkę!**

Boolean

Przyjmują **tylko i wyłącznie** literały **true** i **false**
Nie jest tak jak w C++ że false to 0 a true 1. W Javie jest tylko true i false:
```
boolean b = true;
```

Poniższe **nie zadziała:**
```
int x = 1; if (x) { } // Compiler error!
```

Character

Chary są tak naprawde literałami zapisanymi jako **dodatnie 16 bitowe inty**

- pojedynczy znak:
```
    char a = 'a';
    char b = '@';
```

- w Unicode poprzedzamy przedrostkiem **\u** w apostrofach 'poj':

```
    char letterN = '\u004E';
```

- poniższe **prawidłowe** przykłady:

```
char a = 0x892; // hexadecimal literal
char b = 982; // int literal
char c = (char)70000; // The cast is required; 70000 is out of char range
char d = (char) -98; // Ridiculous, but legal
```

- poniższe **nieprawidłowe** przykłady:

```
char e = -29; // Possible loss of precision; needs a cast
char f = 70000 // Possible loss of precision; needs a cast
```

- znaki można escapować ukośnikiem \

```
char c = '\"'; // A double quote
char d = '\n'; // A newline
```

Stringi

Stringi **nie są prymitywami** ale moga byc reprezentowane przez literały:
- W cudzysłowach: "to jest string"
- Łącznikiem jest +: "To jest" + " string"

3.2. Operatory przypisania

Operatorem przypisania zmiennej do wartości jest =

Przypisanie Button b = new Button(); zapisuje pod b **tylko** referencje do obszaru pamieci gdzie jest obiekt a **nie sam obiekt**. Format bitów takiej zmiennej referencyjnej jest **zależny od JVM**.

Literał liczbowy jest zawsze castowany do inta. np:
byte b = 27; jest tak naprawde zmieniane na *byte b = (byte) 27;* przez JVM
Tak samo z **char** oraz **short**

Mnożenie, dodawanie itp wartości int i innej lub np byte i short jest automatycznie castowany do inta.
ale np byte d = b + c; wywoła błąd kompilatora i wymagane jest ręczne castowanie byte c = (byte) (a + b);

Przejdźmy do typów wyrażeń arytmetycznych. Zerknijmy na poniższy przykład.

```
byte a = 1;
byte b = a + 1; // błąd!

byte c = 1 + 1;
byte d = 64 + 64; // błąd!

byte e = 1 + 1L; // błąd!
```

Zacznijmy od tego, że jeśli wyrażenie arytmetyczne używa tylko operandów całkowitych to typ wyrażenia jest również całkowity i jest to zawsze int albo long. Typem takiego wyrażenia

jest int jeśli żaden z operandów nie jest typu long. Jeśli choć jeden z operandów jest typu long to całe wyrażenie jest też typu long. Wyrażenie ‘1 + 1’ jest zatem typu int i kompilator godzi się wykonać konwersję automatyczną do typu byte, ale wyrażenie ‘1 + 1L’ jest już typu long, dla którego konwersja implicite nie jest przewidziana. Wartość wyrażenia, w którym nie występują zmienne jest wyliczana w czasie kompilacji. Wartością wyrażenia ’64 + 64’ jest 128 co jest poza zakresem wartości typu byte i stąd błąd w powyższym przykładzie. Ale czemu wartość wyrażenia ‘1 + 1’ można przypisać do zmiennej typu byte, a wartości wyrażenia ‘a + 1’ już nie? Oba te wyrażenia są typu int, ale w tym drugim występuje zmienna, a więc w czasie kompilacji nie wiadomo, jaka jest jego wartość. Skoro nie wiadomo jaka jest wartość to nie wiadomo czy jest ona z zakresu wartości typu byte i stąd błąd. Aby kod ten się skompilował należy zastosować explicite operację rzutowania. Z typami short oraz char – który można traktować także jak typ liczbowy – sytuacja jest analogiczna.

Jeśli wyrażenie arytmetyczne zawiera choćby jeden operand typu float a nie zawiera operandów typu double to typem takiego wyrażenia jest też float. Jeśli wyrażenie arytmetyczne zawiera choćby jeden operand typu double to typem wyrażenia jest double. Analogiczne przypisanie jak pokazane w ostatnim przykładzie dla typu byte dla typu float jest poprawne. Poniższy kod kompiluje się.

```
float a = 1.1f;
float b = a + 1.2f;
```

Naturalnie wszystkie te problemy związane z konwersją wartości i typów liczbowych spowodowane są troską o poprawność kodu. Co się dzieje, jeśli zmusimy kompilator do wykonania konwersji poprzez rzutowanie, a wartość jest zbyt duża by móc być przypisana do zmiennej danego typu? Nastąpi konwersja stratna, czyli w przypadku liczby, zostaną odrzucone jej początkowe bity.

Gdy rzutowanie jest określone explicite, więc kompilator nie protestuje, program się kompiluje. Ale jaki jest efekt jego uruchomienia? Otrzymujemy napis np "[byte]: x = -128 y = 0", a więc wartości naszych zmiennych byte są "nieco dziwne". Nie ma cudów i trzeba o tym pamiętać. Jeśli wartość jest zbyt duża by zmieścić się w danym typie, to się tam po prostu nie zmieści.

Rzutowanie prymitywów
Rzutowanie implicite (czyli bez pisania, domyslne) zdarza sie gdy castujemy do "szerszych" wartości:
Animal a = dog; czy *int i = jakisByte*

Błędy typu: *"possible loss of precision"* są spowodowane gdy chcemy zmieścić coś większego w mniejszym np *long l = jakisShort;*

Rzutowanie zawężające (w dół) wymaga napisania castowania explicite.

```
float a = 100.001f;
int b = (int)a; // Explicit cast, the float could lose info
```

Wartości integer (int, long, itp) mogą być castowane implicite do double: double d = 56L; nie można jednak castować implicite wartości np int do double: int i = 12.89; --> zposoduje to **błąd kompilacji**
Można natomiast castować explicite ale trzeba się liczyc ze stratą precyzji:
int x = (int) 12.32; utnie do 12

Podobnie jest z double i float. Double są domyślne. Double są szersze więc **castowanie**

implicite floata do doubla spowoduje błąd:
float f = 12.2; // **błąd**
float s = (float) 12.2; // **skompiluje sie ale** wywali **podczas runtime**
float z = 12.2F; // OK

Przypisanie prymitywów
int a = 6;
int b = a; // pod b kopiowany jest ciąg bitów odpowiadających 6 i dalej te zmienne nie mają ze sobą żadnych zależności (inaczej niz w obiektach)

Przypisanie typów obiektowych
Button b1 = new Button(); // tworzy zmienną ref, następnie tworzy obiekt na stercie, a nastepnie przypisuje referencje do adresu obiektu w pamięci
Button b2 = b1; // spowoduje przypisanie referencji a nie skopiowanie obiektu, b1 i b2 wskazują na ten sam fizycznie obiekt
b2 = null; // spowoduje że referencja nie będzie już wskazywala na obiekt ale b1 nadal na niego wskazuje więc obiekt zyje nadal

Obiekt **String** jest inny. **Nie można zmienić wartości Stringa ponieważ jest immutable**

```
    String x = "Java";
    String y = x;
    x = x + " Bean";  // to nie zmieni wartości y a tylko x
```

Za każdym razem jak zmieniamy wartość Stringa (jak przy x) JVM **zmienia po prostu jego referencje a nie sama wartość**!

```
1. String s = "Fred";
2. String t = s;     // Now t and s refer to the same
                     // String object
3. t.toUpperCase();  // Invoke a String method that changes
                     // the String
```

3.3. Zasięg widoczności zmiennych (scope)

```
class Layout { // class
    static int s = 343; // static variable

    int x; // instance variable

    { x = 7; int x2 = 5; } // initialization block

    Layout() { x += 8; int x3 = 6;} // constructor

    void doStuff() { // method
        int y = 0; // local variable
        for(int z = 0; z < 4; z++) { // 'for' code block
            y += z + x;
        }
```

```
    }
}
```

- zmienna statyczna ma największy zasięg
  - jest tworzona gdy klasa jest ładowana i zyje cały czas
- zmienne instancji są 2 w kolejności
  - sa tworzone gdy jest tworzona instancja
- zmienne lokalne sa nastepne
  - żyją póki wykonywana jest metoda (póki metoda jest na stosie)
- zmienne w bloku żyja gdy wykonywany jest blok.

Należy też **pamiętac o przesłanianiu** zmiennych lokalnych

Przykład typowego błędu:

```
static long power(long base, long exponent) {
 for(int x = 1, y = 0; y < exponent; y++) {
 x *= base;
 }

 return x; // błąd!
}
```

Zmienna x została zadeklarowana w pętli i do tejże pętli ograniczony jest zakres jej widoczności

Innym **typowym błędem jest odwołanie sie do zmiennych instancji w metodzie statyczne (najczęściej main)**

```
class ScopeErrors {
   int x = 5;
   public static void main(String[] args) {
      x++; // won't compile, x is an 'instance' variable
   }
}
```

3.4. Zmienne niezainicjalizowane i nieprzypisane

Zmienne instancji czyli zadeklarowane w klasie ale poza metodami i konstruktorem
Są inicjalizowane na wartości domyślne w momencie instancjonowania obiektu (new):
- referencja do obiektu (także String oraz tablica - jako ze jest obiektem): **null**
- byte, short, int, long: **0**
- float, double: **0.0**
- boolean: **false**
- char: '\u0000'

Tablice jako zmienne instancji będą domyślnie zainicjalizowane jako null.
Elementy tablicy zostaną zainicjalizowane domyślnie **na swoje wartości domyślne** (tak jak przy zmiennych instancji - powyżej)
- int [] year = new int[100]; wygeneruje tablicę intów z samymi zerami.

3.5. Zmienne lokalne (stack, automatic)

Zmienne lokalne to takie w metodach i w parametrach metod. Inna nazwa lokalnych (local) to automatic lub stack variables

Prymitywy

Zmienne lokalne muszą być **ZAWSZE** zainicjalizowane z jakąś wartością zanim zostana użyte.
Zmienne lokalne **nie są inicjalizowane automatycznie (domyślnie)**
Próba uzycia niezainicjalizowanej zmiennej lokalnej spowoduje **błąd kompilacji** (nie tak jak w zmiennych instancji gdzie są wartości domyślne i można ich uzyć)
Jeśli zmienne lokalna nie jest zainicjalizowana ale także nie jest użyta to nic sie nie stanie. Błąd jest tylko **w momencie użycia**

Obiektowe, referencje lokalne

Referencje lokalne nie są przypisywane do null. Nie są przypisane do niczego:

```
Date date; // nie null
if (date == null) {  } // nawet takie uzycie spowoduje błąd
```

Dobra rada = zawsze przy deklaracji przypisywac do nula

Tablice
Podobnie jak obiektowe, ale nie potrzeba inicjalizować elementów tablicy po jej skonstruowaniu
- elementy są inicjowane na wartości domyślne.

3.6. Parametry metod

"W języku Java wszystkie parametry są przekazywane przez wartość"

Parametry referencyjne

- Przekazując zmienną do metody przekazujemy jej **kopię "w sensie bitowym"**
  - czyli jeśli jest to **referencja** do obiektu to przekazujemy kopie tej referencji ale ta referencja wskazuje na ten sam obiekt
    - Przekazując parametr referencyjny do metody **bazujemy na obiekcie na który wskazuje referencja**, tj na tymże obiekcie (a nie na jego kopii)
  - można oczywiście zmienić wiązanie tej referencji na inny obiekt.
  - jeśli jest to **zmienna prymitywna bazujemy na jej kopii** (tutaj w dosłownym sensie)

Zmienne prymitywne

- przekazując zmienna prymitywną przekazujemy jej **kopię**

3.7. Tablice - deklaracje, konstruowanie, inicjalizacja

Deklaracje:
- ***int[] key;*** lub *int key[];* ***Thread[] threads;*** *Thread threads[];* (pogrubione - zalecane)
- podczas deklarowania **nie można specyfikować wielkości:** *int[5] key*; **błąd!**
  - JVM doesn't allocate space until you actually instantiate the array object
  - wielkość więc specyfikujemy podczas inicjalizacji: **int[] key = new int[5];**

- zadeklarowana referencja tablicowa na dany **typ prymitywny** (np int[]) może być przypisana do różnych obiektów tablicowych (o różnych długościach) ale tylko będących takim samym typem (int[]).
  - **Nie można przypisać** int[] a do tablicy np char[] ponieważ jest to obiekt tablicowy a nie zmienna prymitywna gdzie takie castowanie działa. - **tu nie działa!**.

- zadeklarowana referencja tablicowa na dany **typ obiektowy** może byc przypisana do obiektu tablicowego **typu podklasy lub klasy implementującej interfejs**
  Car[] cars = new Honda[5];
  - nie działa to oczywiście w drugą strone bo Car niekoniecznie może byc Honda

Constructing

Konstruowanie tablicy to stworzenie obiektu tablicowego na stercie.
Podczas tworzenia JVM **musi** wiedzieć jak duża będzie tablica i ile miejsca na nią przeznaczyć - brak ilości elem. spowoduje błąd kompilacji

```
int[] testScores;          // Declares the array of ints
testScores = new int[4]; // constructs an array and assigns it
                           // the testScores variable
```

  - testScores jest referencją do miejsca na stercie gdzie jest obiekt tablicowy.
  - Obiekt tablicowy posiadający 4 elementy jest inicjowany (zawsze) wartościami domyślnymi (tutaj zerami)

- **podczas tworzenia tablicy z typami obiektowymi NIE SĄ WYWOŁYWANE konstruktory tego typu:**
  - gdy: *Thread[] threads = new Thread[5];* **nie jest wywoływany konstruktor Therad()**
  - nie tworzymy de facto obiektu Thread ale obiekt tablicowy Thread'ów - żaden obiekt Thread nie istnieje jeszcze w tym momencie

Wielowymiarowe:
  - są tablicami trzymającymi referencje na kolejne tablice.
  - deklaracja 1: ***int[][] myArray = new int[3][];*** jest dozwolona ponieważ JVM zna wielkość 1 wymiaru.
    - następne wymiary trzeba zainicjalizować: ***myArray[0] = new int[4];***
  - deklaracja 2: ***int[][] myArray2 = new int[5][4];***

- referencja tablicy zadeklarowane jako wielowymiarowa może być przypisana do innej tablicy ale o tym samym wymiarze:
  - jednowymiarowa tylko do jednowymiarowych
  - dwuwymiarowa tylko do dwuwymiarowych ... itp

- jeśli tablica jest np dwuwymiarowa to do 1 wymiaru można przypisać tylko tablice (tablicew wielowymiarowe to tablice tablic):

```
int[][] books = new int[3][];
int[] numbers = new int[6];
int aNumber = 7;
books[0] = aNumber; // NO, expecting an int array not an int
books[0] = numbers; // OK, numbers is an int array
```

Poniżej **przykład** pokazujący że można zadeklarować tablice [][] a do jednej pozycji dopisać inną**:**

```
int[][] myTable = new int[4][2]; (tablica 4x2)
 myTable[1] = new int[4]; (nowa tablica 4 elem)
```

spowoduje to stworzenie tablicy
00
0000
00
00

Inicjalizacja

Czyli wkładaie elementów

Elementami tablicy są wartości prymitywne lub **referencje** na obiekty

pets[0] = new Animal(); // tworzy obiekt Animal w peirwszej komórce tablicy. pets[0] jest referencją na fizyczny obiekt Animal
liczby[0] = 5; // zapełnia 1 element wartścią prymitywną 5

ArrayIndexOutOfBoundsException - wyjątek w momencie czytania z elementu który jest poza tablicą (element 4 tablicy 3 elementowej)

- Array objects have a single public **variable**, **`length`** that gives you the number of elements in the array
  - myArray.length

Pętla for dla Javy5

```
for(Dog d : myDogs)
d = new Dog();
```

Deklaracja, inicjalizacja w jednej linijce:

int[] dots = {6,x,8}; // 1. tworzy referencje na tablice dots; 2. tworzy obiekt tablicy z 3 elem.; 3. zapełnia elementami; 4. przypisuje do dots obiekt tablicy.
int[][] scores = {{5,2,4,7}, {9,2}, {3,4}}; // wielowymiarowa tablica
  - The size of each of the three int arrays is derived from the number of items within the corresponding inner curly braces

anonymous array creation
- can be used to construct and initialize an array, and then assign the array to a previously

declared array reference variable:
int[] testScores;
testScores = new int[] {4,7,2};
- taką konstrukcję wykorzystuje się najczęściej jako tablice chwilowe z wartościami np jako argument metody:
f.takesAnArray(new int[] {7,7,8,2,5});

Inicjalizacja tablicy innymi wartościami

Elementami tablicy moga byc elementy mogące się implicite zcastowac na zadeklarowany typ.
Np:

```
int[] weightList = new int[5];
 byte b = 4;
 char c = 'c';
 short s = 7;
      weightList[0] = b; // OK, byte is smaller than int
      weightlist[1] = c; // OK, char is smaller than int
 weightList[2] = s; // OK, short is smaller than int
```

Elementami tablicy mogą byc tez **obiekty podklasy** zadeklarowanej w tablicy
Car [] myCars = {new Subaru(), new Car(), new Ferrari()};
albo klasami implementującymi dany interfejs
ISporty[] sporty = {new SubaruImplSporty(), new CarImplSporty(), new FerrariImplSporty()};

3.8. Bloki inicjalizacyjne

Bloki inicjalizacyjne są trzecim miejscem (po metodach i konstruktorach) gdzie można zapisać operacje.
Bloki te wpisuje sie w klasie poza metodami i konstruktorami.
Dzielimy je na **statyczne** i **instancyjne**

```
class SmallInit {
    static int x;
    int y;

    static { x = 7 ; }  // static init block
    { y = 8; }          // instance init block
}
```

Bloki **nie mają nazw**, **nie mają argumentów, nie zwaracają niczego**
Według konwencji Javy powinny się znaleźć na górze klasy ale nie jest to obowiązek.

Statyczne
- wykonywany **jeden raz** gdy po raz pierwszy ładowana jest klasa

Instancyjne
- wykonywane **jeden raz** gdy inicjalizowany jest każdy nowy obiekt
- wykonywany w konstruktorze **zaraz po** wywołaniu konstruktora nadklasy super() **a przed kolejymi instrukcjami** konstruktora

```
Konstruktor(){
     super(); // nie musi byc explicite
    // blok inicjalizacyjny
    // dalszy kod konstruktora
}
```

- jeśli jest więcej bloków inicjalizujących kolejność wykonywania pokrywa sie z kolejnością wystepowania

- bloki te służą do tego aby wykonać te instrukcje, które powinny posiadać (wykonać) wszystkie konstruktory klasy.
- **błąd** w bloku inicjalizacyjnym będzie sygnalizowany poprzez: **ExceptionInIninitalizationError** a w Javie 6 (?) po prostu błąd runtime.

3.9. Klasy wrappery

Wrappery służą:
- do obudowania wartości prymitywnych w miejscach gdzie potrzebne sa obiekty np w Kolekcjach
- dostarczyć metod niedostepnych w wartościach primitive

- moga być **null**

Wrappery są **final** a więc nie można po nich dziedziczyć

Wrapperami są np **Integer** (dla int), **Float** (float), **Character** (dla char) itd.

Primitive	Wrapper Class	Constructor Arguments
boolean	Boolean	boolean or String
byte	Byte	byte or String
char	Character	char
double	Double	double or String
float	Float	float, double, or String
int	Integer	int or String
long	Long	long or String
short	Short	short or String

Konstruowanie wrapperów

Większość wrapperów na wart. prymitywne (prócz Character) przyjmuje String jako argument
- może wyrzucic NumberFormatException gdy String nie da się sparsowac na ten format

numeryczny
- klasy wrapperów są **immutable** a więc nie można zmienić ich wartości

```
Integer i1 = new Integer(42);
Integer i2 = new Integer("42")  // string
Float f1 = new Float(3.14f);
Float f2 = new Float("3.14f");   // string
```

Character przyjmuje tylko char jako argument konstruktora

Boolean przyjmuje bool lub **nie-casesensitive** stringa wartością "true" lub "false"
- **konstruktor tej klasy nie zwraca wyjątków**. Obiekt klasy 'Boolean' reprezentujący wartość 'true' uzyskamy z napisu "true", przy czym **nie ważna jest wielkość liter**, zaś obiekt reprezentujący wartość '**false' z każdego innego napisu**, oraz gdy zamiast obiektu klasy 'String' **przekażemy wartość 'null'** (!)
- Boolean **może być uzyty jako warunek tylko od Javy 5** (if (b) zwróci błąd w 1.4) - z powodu autoboxingu

Innym sposobem na konstruowanie wrappera jest metoda **valueOf()**
- metoda ta zwraca obiekt "klasy wrappera" z wartościa podaną w argumencie jako String
- **czasem** drugi argument tej metody to `int radix` mówiacy o systemie (binary, oct, ten, hex itd) w jakim podany jest String

```
Integer i1 = Integer.valueOf("101011", 2);  // przypisze 42 pod i1
Integer i2 = Integer.valueOf("20");          // przypisze 20 pod i2
```

Pobieranie wartości z wrappera

Konwersja obiektu wrappera na typ prymitywny odbywa sie poprez metody z rodziny **xxxValue()**

Przykłady:

```
Integer i2 = new Integer(42);
    byte b = i2.byteValue();
    short s = i2.shortValue();
    double d = i2.doubleValue();
Float f2 = new Float(3.14f);
    short s = f2.shortValue();
```

... itd

Parsowanie

Metoda **parseXxx()** jest podobna do **valueOf()**.
- także przyjmuje stringa oraz może przyjmowac `int radix`
- **zwraca prymityw** w przeciwieństwie do valueOf która zwraca obiekt wrappera.
- **valueOf()**: w celu optymalizacji – używane są istniejące, ale już nie wykorzystywane instancje

toString()

- Odziedziczona z Object
- **bezargumentowa** lub **statyczna argumentowa** która **jako argument bierze**

prymityw który obsługuje dany wrapper
- d.toString()
- Double.toString(3.14); (bierze jako argument double bo obsługuje Double) -
statyczna
- Integer i Long oposiadaja jeszcze trzecią meodę toString() przyjmująca jako drugi arg. podstawe systemu liczbowego `int radix`
Long.toString(254,16); // wyswietli "hex = fe"

- zwraca wartość jaka jest opakowana przez wrapper

toxxxString() (Binary, Hexadecimal, Octal)

Integer i **Float** posiadają metody pozwalajace na konwersje z systemu 10nego na inny.

- Integer.toHexString(254); // convert 254 to hex result: "254 is fe"
- Long.toOctalString(254); //convert 254 to octal result: "254(oct) =376"
- Integer.toOctalString(254);
- itd....

Podsumowaniemetod we wrapperach

Method s = static n = NFE exception	Boolean	Byte	Character	Double	Float	Integer	Long	Short
byteValue		x		x	x	x	x	x
doubleValue		x		x	x	x	x	x
floatValue		x		x	x	x	x	x
intValue		x		x	x	x	x	x
longValue		x		x	x	x	x	x
shortValue		x		x	x	x	x	x
parseXxx s,n		x		x	x	x	x	x
parseXxx s,n (with radix)		x				x	x	x
valueOf s,n	x	x		x	x	x	x	x
valueOf s,n (with radix)		x				x	x	x
toString	x	x	x	x	x	x	x	x
toString s (primitive)	x	x	x	x	x	x	x	x
toString s (primitive, radix)						x	x	

In summary, the essential method signatures for Wrapper conversion methods are

primitive xxxValue() - to convert a Wrapper to a primitive

primitive parseXxx(String) - to convert a String to a primitive

Wrapper valueOf(String) - to convert a String to a Wrapper

3.10. Autoboxing

- Nowa funkcja w Java 5
- **Umożliwia użycie obiektów wrapperów**
  - automatycznie robiony jest *unwrap* przed uzyciem (np inkrementacji) i ponowny *wrap* po uzyciu

```
Integer y = new Integer(567);   // make it
y++;                              // unwrap it, increment it,
                                  // rewrap it
```

  - pozwala jak w powyższym przykładzie robic inkrementacje y++

- **ale** ponieważ wrappery sa **immutable** de facto podczas boxingu **tworzony jest nowy obiekt** pod ta referencją
  - patrz przykład na stronie 234/235

equals i operatory porównania

equals() - sprawdza czy 2 obiekty są "znaczeniowo" równe

- w equals obiekty-wrappery są równe gdy maja **tą sama wartość i typ**
- uzywając ==, != wyniki są **inne!!** wręcz inne dla różnych JVM
  - dlatego **zalecane jest uzywanie equals!!!!**

Oto wycinek dokumentacji:
"*If the value p being boxed is true, false, a byte, a char in the range \u0000 to \u007f, or an int or short number between -128 and 127, then let r1 and r2 be the results of any two boxing conversions of p. It is always the case that r1 == r2.*"

3.11. Overloading wraz z Boxing, widening i var-args - kolizje

Kolidujące metody overloading

Widening w overloadingu

Podczas przeładowania metod uzywana jest konwersja implicite typów argumentów - tzw "**widening**":

```
void go(int x)    { System.out.print("int "); }
void go(long x)   { System.out.print("long "); }
void go(double x) { System.out.print("double "); }

byte b = 5;   short s = 5;   long l = 5;   float f = 5.0f;

go(b); // arg: byte, podpasowuje do int
go(s); // arg: short, podpasowuje do int
go(l); // arg: long
go(f); // arg: float, podpasowuje do double
```

Da na wyjściu **int int long double**
- byte i short (jako typy całkowite "mniejsze" od inta są "widening'owane" do inta)
- long pasuje do metody long
- float podchodzi pod double.

Gdyby były wersje metod takze dla byte i short to oczywiście byte i short byly by przypisane do swoich metod.
Gdyby była wersja dla short (ale nie dla byte) to byte podpasował by do short.

- jeśli nie ma metody odpowiadającej idealnie typom parametrów wywołania, to wybierana jest ta metoda, której argumenty są jak **najmniej "większe"**, przy czym "najmniejszy" jest typ ‘byte’ i kolejno ‘short’, ‘int’, ‘long’, ‘float’, ‘double’. Parametry typu ‘char’ traktowane są jakby były typu ‘int’

- **konwersja do bardziej "pojemnego" typu prostego ma pierwszeństwo przed opakowywaniem** (ang. in-boxing).

Przykład:

```
    static void go(double x) { System.out.print("double "); }   // ta bedzie wykonana
    static void go(Integer x) { System.out.print("Integer "); }

    int x = 1;   go(x) ---> da wynik double a nie Integer!
```

Widening nie dotyczy klas wrapperów tzn Short nie jest Integerem!!!

- *void test(Long x) { }* **nie zadziała dla** *d.test(new Integer(5));*

Boxing i var-args w overloadingu

Podobnie jak prze widening **podczas przeładowania metod dokonuje się boxing typów** takich jak Integer, Double **do typów prymitywnych i odwrotnie**.

- Mając metodę *go(Integer i)* można wywołać ja z parametrem typu int [*go (2);*] i odwrotnie
- W przypadku gdy są dwie metody kolidujące **pierwszeństwo ma widening** przed boxingiem
  - (czyli metoda **wcześniejsza historycznie** - w Java 1.4. byl widening ale nie było boxingu)
  - mając: *void go(Integer x); void go(long x)*; wywołanie z int a: *go(a)* **wywoła metodę *go(long)*** bo widening ma priorytet
- Podobnie jak podczas gdy koliduje boxing i typ prosty sprawa się ma z metodami var-args
- priorytet ma to co jest **historycznie pierwsze** czyli argumenty proste

  void go(int x, int y) // 1
  void go(**byte**... x) // 2
- wywołanie go(i, y) gdzie i oraz y to **byte** spowoduje wywołanie **metody 1** czyli **go(int, int)**

Metody z konkretną liczba parametrów maja priorytet przed var-args nawet jeśli varargs maja typ prosty a metoda z ustaloną liczbą parametrów wymaga autobxingu.

Widening i Boxing

Połączenie WidenAndBox **nie powiedzie się** np gdy:

void go(Long x){} byte b = 5; **go(b);** --> błąd kompilatora z powodu tego że byte może byc boxowany do Byte... a Byte to nie Long

Ciekawostka:

```
static void go(Object o) {
      Byte b2 = (Byte) o; // ok - it's a Byte object
      System.out.println(b2);
}
public static void main(String [] args) {
      byte b = 5;
      go(b); // byte - Boxowany do Byte - podchodzi pod go(Object)
            // ale metoda jest dosyć niebezpieczna bo pasuje tam kazdy obiekt
            // i moze sie wywalic przy probie castowania na Byte.
```

```
}
```

Here's a review of the rules for overloading methods using widening, boxing, and var-args:
- Primitive widening uses the "smallest" method argument possible.
- Used individually, boxing and var-args are compatible with overloading.
- You CANNOT widen from one wrapper type to another. (IS-A fails.)
- You CANNOT widen and then box. (An int can't become a Long.)
- You can box and then widen. (An int can become an Object, via Integer.)
- You can combine var-args with either widening or boxing.

Podsumowując:
jeśli tylko się da, kompilator wybiera **1. konwersję typów prostych do innych, "pojemniejszych"** typów prostych.
W dalszej kolejności, kompilator próbuje znaleźć **2. metodę, która wymaga opakowania w obiekty** a dopiero,
gdy to zawiedzie, rozważa **3. metody używające argumentów wielo-arnych**.

3.12. Garbage collection

Heap (sterta) - jest tylko jedna.

- Zadaniem Garbage Collectora (**GC**) jest **odnalezienie i usunięcie** obiektów **do których nie można się dostać** (nie ma do nich referencji)
- jest zażądzany przez JVM
- można wysłac prośbę do JVM o uruchomienie GC ale **nie ma żadnej gwarancji** że zostanie uruchomiony.

- obiekt **może być usunięty poprzez GC** gdy **ani jeden z aktywnych wątków nie ma do niego dostępu.**
  - troszke inaczej zachowują sie Stringi spowodu swojej "dziwnej" natury
- jeśli obiekt jest możliwy do usunięcia przez GC **to i tak nie ma gwarancji** ze GC go usunie - są to tylko prośby a nie żądania
- pomimo działania GC aplikacja Java może zapełnić całą pamięć
- stack variables are not dealt with by the GC (- z testu)

Pisanie kodu bezpiecznego "pamięciowo"

ELIGIBLE = Oznaczone dla GC jako możliwe do usuwania.

- **nullowanie referencji**
  - np StringBuffer sb = null;
  - obiekty stworzone w metodzie są *eligible* po ukonczeniu metody

- przepisywanie referencji
  - ustawianie referencji aby wskazywała na inny obiekt
  - StringBuffer sb1 = new... ; StringBuffer sb2 = new... ; .... jakies operacje .... sb1 = sb2;
  - **metody które zwracają obiekt a nie są przypisane** do referencji nie są *eligible*

- **izolowanie referencji**
  -Występuje gdy Klasa ma referencje do samej siebie i stworzymy kilka takich obiektow referujących do siebie. a następnie znullujemy je. Takie "wyspy" także są *eligible* dla GC

- **prośby włączenie GC**
  - są to tylko sugestie dla JVM że powinna włączyć GC, nie ma jednak gwarancji kiedy to wykona i czy wsyztsko usunie co trzeba.

```
System.gc();
```

  - powyższa metoda wykonuje tak naprawde Runtime.getRuntime() która zwraca singletona na którym to jest wywolana metoda gc().
    - **`java.lang.Runtime.getRuntime().gc()`**
  - **teoretycznie** po wywołaniu gc() będziesz miał tyle wolnej pamięci ile jest możliwe
    - niektóre JVM'y różnie implementują ta metodę lub nie implementuja jej wcale

- **czyszczenie ręczne przed wywołaniem GC** - metoda: public void **finalize()**
  - jest odziedziczona przez każdy obiekt z Object
  - jest to metoda wywoływana tuż przed usunięciem obiektu przez GC
    - **nie należy na niej polegać** ponieważ nigdy niewiadomo kiedy obiekt będzie usunięty przez GC i czy wogóle.

  - finalize() może byc wywołane **tylko raz przez Garbage Collectora**
  - wywołanie finalize() ręcznie **może spowodować że obiekt nie będzie usunięty** przez GC
  - finalize() **może zostac nadpisana (override)**

Metoda sprawdzająca ilość pamięci:

```
Runtime rt = Runtime.getRuntime();  // singleton Runtime
rt.totalMemory();
rt.freeMemory();
```

TEST 7/14 - nedza okropna!!

4. Operatory

4.1. Przypisania

- podczas przypisywania wartości do prymitywów rozmiar wartości ma znaczenie
  - pamietać o kastowaniu imlicite i explicite, i że może wystapic obcięcie wartości
- referencja to nie obiekt ale wskazanie na obiekt
- podczas przypisania wartości do referencji typ ma znaczenie
  - pamietac o podtypach, nadtypach, i tablicach

4.1.1 Przypisania+działania

Jest ich 11 ale na ezaminie i w uzyciu są: **+=, -=, *=, /=**
- to co jest po **prawej stronie** operatora jest wykonane **pierwsze**:
x *= 2 + 5; odpowiada **x = x * (2 + 5);** a nie x = (x * 2) + 5;

4.2. Relacyjne

relacyjne **<, <=, >, >= (integers, floating points, chars)**
równości **==, !=** (liczby, chary, booleany, referencje - **nie można niekombaptybilnych typów**

- zawsze ich wynikiem jest **boolean** (true, false)
- można ich uzyć w **if** (najczęściej), przypisaniu: boolean b = 100 > 999;,

Równości

- można porównywać liczby (w tym chary), booleany, referencje
- **nie można niekombaptybilnych typów**
  - np boolean z charem albo Button ze String
- == porównuje wartośc pod zmienną jako ciąg bitów (pamietaj: prymitywy i referencje)

- **UWAGA: rezultatem operacji przypisania poprzez '=' jest wartość jaką przypisano**:

```
boolean b = false;
if (b = true)
```

---> przypisano wartość true więc wewnątrz ifa jest true.
  - ponieważ **w if może być tylko wartośc typu boolean** (nigdy int i inne) więc poniższe nie skompiluje się

```
int x = 1;
if (x = 0)
```

---> Przypisano 0 więc wartością ifa jest 0 - ale **INT więc się nie skompiluje**

Porównywanie prymitywów:

character 'a' == 'a'? true
character 'a' == 'b'? false
5 != 6? true
5.0 == 5L? true
true == false? false

Porównywanie referencji:

Przy porównaniu referencji pod uwage jest brane ciagi bitów - czyli **to na co wskazują** *referencja_a == referencja_b* jeśli te referencje **wskazują na ten sam** fizyczny obiekt

- dlatego też podczas porównania "znaczeniowego" obiektów (także String) należy używać *equals()*

Porównywanie enuma:

- Porównanie dwóch **referencji typu enum można dokonać za pomocą == oraz equals**

- jest to tożsame:

```
Color c1 = Color.RED; Color c2 = Color.RED;
c1 == c2        TRUE
c1.equals(c2)   TRUE
```

4.3. Operator instanceof

- uzywany względem **zmiennych referencyjnych** oraz
- sprawdza **czy obiekt jest pewnego typu**
  - obiekt (**typ referencji**) z lewej striny operatora przechodzi test **IS-A** na klasie po prawej stronie operatora

```
s instanceof String
```

  - referencja **musi wskazywać na obiekt** aby instanceof dało true
    - typ referencji (np Apple a = null) nie wystarczy
- jego wynikiem jest **boolean** (true/false)

- można testować referencje na swoją klase lub jej podklasy, *nadklasy* **oraz** interfejsy

Przykłady:

B b = new B();
if (b instanceof Object) **TRUE** (nadklasa)

A a= new B();
if (a instanceof B) **TRUE** (podklasa)

B a= new B(); //B extends A
if (a instanceof A) **TRUE** gdy B extends A (podklasa)

A a= new **A**(); // B extends A
f (a instanceof B) **FALSE** (podklasa ale typ jest A i obiekt jest A - mimo że B extends A)

A implements Foo
a instanceof Foo **TRUE**

A a = null;
if (a instanceof A) **FALSE** bo obiekt jest null

- **nie można testować instanceof z dwóch niezależnych drzew dziedziczenia**
  - operatora 'instanceof' nie możemy użyć, jeśli nie ma szans na powodzenie

Dog d = new Dog();
d instanceof Cat ---> gdy ani Dog ani Cat nie są w tym samym drzewie: **błąd kompilacji**

- w przypadku tablic: **tablice to Object** więc
int [] nums = new int[3]; if (nums instanceof Object) **TRUE**
ale Foo[] foos = new Foo[4]; if (foos instanceof Foo) **FALSE**

4.4. Operatory arytmetyczne

+ - * / - standardowe

Operator reszty **%**

% dzieli lewy operand przez prawy a rezultatem jest reszta z dzielenia

Konkatenacja Stringów

"String1" **+** "String2"

- Przy stringu numery (inty) **są brane jako znaki** a nie liczby **gdy conajmniej jeden operand jest stringiem**
  - **-** (1+ "string") da nam 1string
  - ("string" + 1 + 2) da nam string12
  - ("string" + (1+2)) da nam string3 - bo wtedy 1+2 jest liczone jako 1.
- jest możliwy operator konkatenacji+przypisania: **+=**

4.5. Operatory inkrementujący i dekrementujący

++ i --

- operatory te maja pierwszeństwo przed innymi *, /, % ,+ - (+ i - są najsłabsze)

Prefix: ++i, --i
  - inkrementacja następuje **przed uzyciem zmiennej w wyrażeniu** (zanim zostanie uzyta)

Postfix: i++, i--
  - inkrementacja następuje **po uzyciu tej zmiennej w wyrażeniu** (po tym jak zostanie uzyta)

```
int x = 2; int y = 3;
if ((y == x++) | (x < ++y)) {                    // 1. test: 3 == 2   (potem inkr x)
                                                 //  2. test 3 < 4 (inkr y przed podstawieniem)
      System.out.println("x = " + x + ", y = " + y);  // output: x = 3, y = 4
}
```

- zmiennych **final** niemożna inkrementowac ani dekrementować (oczywiste) - błąd kompilacji

4.6. Operator warunkowy

- Jest oparatorem trój-operandowym
- Działa jak **if** z tym wyjątkiem że bloki wykonawcze służą do przypisania ich do jakiejś zmiennej:

x = **(***boolean expression***) ?** *value to assign* if true **:** *value to assign* if false

String status = (numOfPets < 4) ? "Pet limit not exceeded" : "too many pets";

- **można operatory zagnieżdzać**

```
String status = (numOfPets<4)?"Pet count OK"
        :(sizeOfYard > 8)? "Pet limit on the edge"
           :"too many pets";
```

4.7. Operatory logiczne

Operatory bitowe

NIE MA NA EGZAMINIE SCJP5 (były w SCJP 1.4)
- Są to:
  - **&** AND
  - **|** OR
  - **^** XOR
- są operatorami dla operandów całkowitoliczbowych

*Logiczne typu "**short-circuit**" - leniwie*

Sprawdzany jest najpierw pierwszy warunek i w zależności od wyniku **drugi warunek nie zawsze jest sprawdzany**

&& AND
- sprawdzany jest pierwszy warunek i gdy jest false, drugi warunek nie jest nawet sprawdzany.

|| OR
- sprawdzany jest pierwszy warunek i gdy jest true to drugi juz nie jest sprawdzany - bo nie musi.

- powyższe operatory **działają tylko z typami boolean**
  - taki kod nie skompiluje sie: *if (5 && 6) { }*

*Logiczne typu "**non short-circuit**" - zachłannie*

- dla operandów logicznych

Używane podobnie jak te powyżej ale **sprawdzane są zawsze oba warunki**

& AND
- nawet gdy pierwszy warunek jest false, to drugi tez jest sprawdzany (wykonywany)

I OR
- nawet gdy pierwszy warunek jest true, to drugi tez jest sprawdzany (wykonywany)

Logiczne operatory ^ i !

Działa **tylko na wartościach boolean**

^ XOR (ALBO)
- gdy tylko jeden warunek jest prawdziwy
! NOT (Odwrócenie warunku)

TEST pytanie 3 = przyjrzec sie - dalem F F X T X, 8 pytanie nie chcialo mi sie liczyc
---

5. Flow Control, Wyjątki, Asercje

Przepływ danych - pętle,warunki

5.1. Instrukcje warunkowe

5.1.1 Instrukcja if-else

if (**only_boolean_expresion**) { jesli true } **else** { w p.p. }

- jako warunek może wystąpić **tylko wyrażenie typu boolean**
- blok else jest opcjonalny
- nawiasy {} są także opcjonalne - ale wtedy wywoływana jest tylko jedna (nast) linijka.
- gdy istnieje klauzula warunkowa else if można dać ostatni else ale wówczas może on wystąpić na końcu
- w przypadku gdy "else if" przejdzie (true) to kolejne else-ify ani elsy nie są wykonane

- **Uwaga** na zapis if (bl = true) gdzie bl to boolean. - zapisa sie skompiluje i test powiedzie:
- uzywając w teście przypisania "=" test sie wykona, przypisanie też sie wykona a test będzie równy przypisanej wartości (tutaj true)
- w przypadku gdy robimy przypisanie do wartości innej niz **boolean** kompilacja sie nie powiedzie - wartośc przypisania bedzie nie-bool

Uwaga na egzaminie na formatowanie ifów i brak { }

5.1.2. Instrukcja switch

- odpowiednik sekwencji ifów gdy jest w kazdym case instrukcja break;

```
switch (testowana_zmienna) { // tylko typy: int, short, byte, char oraz enum
        case wartosc1:
                polecenia ...    // blok poleceń   może być w nawiasach { }
                break;           // opcjonalny, ale ważny
        case wartoscx:
                polecenia ...   // blok poleceń
                break;          // opcjonalny, ale ważny
        default:
                polecenia ...   // blok poleceń
    }
```

- blok poleceń (wraz z break) może byc w nawiasach {} ale nie musi - gdy nie ma to rónież wykonają się wszystkie polecenia
  - **blok poleceń może byc także pusty** (ciało puste)

```
...
    case 1:
case 2:
...
```

- dwukropek w case jest obowiązkowy
- wartość testowana **może być tylko** typu ***int, short, byte, char*** oraz ***enum*** lub typem dającym się na nie castować implicite
  - **nie moga to być:** double, float, long **!!!!!!!!!!**
  - testowaną zmienną może byc także wartość: **switch (7)** --> OK

- Argumenty dla wariantów (**przy case**) muszą być wartościami znanymi w czasie kompilacji,
  - **literały,** lub **zmienne finalne**, z wartością przypisaną **w czasie deklaracji**

```
final int a = 1;
    final int b;
    b = 2;
    int x = 0;
    switch (x) {
       case 2: // ok - literał
            case a: // ok - stała zadeklarowana z wartością
            case b: // compiler error   = b jest nieprzypisana od razu
...
```

- case może sprawdzić tylko relację **równości** tj inne testy typu >, <, >=, <= **są niemożliwe**
- **można wywoływać metody** wewtątrz: np: *switch(str**.length()**)*
- należy uważać aby wartości w case mieściły się w swoim typie tzn np aby byte nie był za duzy np 128

```
byte g = 2;
switch(g) {
    case 128:    / błąd kompilacji - byte nie może przyjąć wartości 128
```

- wartości w case **nie mogą sie powtarzać** - błąd kompilacji
- można polegac na autoboxingu (uzywac wrapperów np Integer, Short)

- ale **nie można** Double, Float, Long podomnie jak prymitywów

Break w switch-case

Stałe w *case* są wykonywane od góry do dołu (po kolei) od momentu napotkania wartości pasujące:
- pierwsza napotkana wartość pasująca jest *entry pointem* i jest tykonany oraz wszystkie nastepne.

- Przykład bez breaków:

```
enum Color {red, green, blue}
class SwitchEnum {
      public static void main(String [] args) {
            Color c = Color.green;
            switch(c) {
               case red: System.out.print("red ");
                  case green: System.out.print("green ");
                  case blue: System.out.print("blue ");
                  default: System.out.println("done");
       }
    }
}
```

Output: green blue done

- takie przejscie "z góry na dół" bez breaków nosi nazwę **fall through** i wg rad Suna powinno byc zaznaczone komentarzem dla czytelności
- *fall through* działa do napotkania pierwszego breaka

default

- gdy żadna zmienna nie pasuje do casów sterowanie przechodzi do default.
  - gdyby nie było default switch zakończył by się nie wchodząc do żadnego case
- w przypadku "fall through" blok default także jest brany pod uwagę i w przypadku braku breaka także jest wykonany.
- default nie musi być na końcu - może się znaleźć na dowolnym miejscu.
  - uwaga na fall through:
    - gdy *entry point* będzie wczesniej default też się wykona i case za nim także
    - gdy *entry point*em będzie default (nic nie podpasuje) to wykona sie default oraz ewentualne casy za nim.

```
int x = 7;
switch (x) {
  case 2: System.out.println("2");
default: System.out.println("default");
  case 3: System.out.println("3");
  case 4: System.out.println("4");
}
```

Output: default, 3, 4 // nie pasowala zadna wartosc więc wziął default oraz kolejne (bo nie było break)
// tutaj default stał się entry pointem

5.2. Pętle i iteratory

5.2.1. Pętla while

```
while (boolean_expression) {
        // do stuff while expresion is true;
}
```

- pętla sie wykonuje **dopóki** warunek jest *true*
- zmienne w warunku muszą byc **zadeklarowane wcześniej**
  - **nie można deklarować** zmiennych w warunku**:** while(int i = 2)
- pętla while może nie wykonać się ani razu - jeśli warunek od razu jest spełniony
  - warunek jest sprawdzany przed wykonaniem pętli

5.2.2. Pętla do-while

```
do {
        // do stuff first;
} while (boolean_expresion) ;    // ze srednikiem na końcu
```

- w przeciwieństwie do while pętla jest wykonywana **przed sprawdzeniem warunku**
- na końcu warunku **musi byc średnik**
- ***zasady są takie jak w while***

5.2.3. Pętla for

Od Javy w wersji 5 są dwie petle for:
- podstawowa
- rozszerzona zwana pętla for-each lub for-in (z dokumentacji Sun'a) - do tablic i kolekcji

Podstawowy for

```
for ( /*Var_Initialization*/ ; /*Condition*/ ; /* Iteration */ ) {
        /* loop body */
}
```

- są trzy części oddzielone średnikiem **;** :
  1. deklaracja i inicjalizacja zmiennej
  2. warunek (wyrażenie boolean, test)
  3. wyrażenie iterujące

ad 1.
- deklaracja i inicjalizacja jest wykonywana na samym początku instrukcji for
- zmiennych może byc kilka oddzielonych przecinkami: for (**int i = 1, j = 2**; j < 10; j++)
  - zasięg zmiennych tu zadeklarowanych jest tylko wewnątrz tej pętli
- sekcja ta **może być pusta** jeżeli uzywamy zmiennej zadeklarowanej wcześniej
  int i =0; for (**;** i<5; i++){}

ad 2.

- wyrażenie testujące **musi być** typu **boolean**
- można dać **tylko jedno wyrażenie** testujące
  - ale wyrażenie to może byc skomplikowane:

for (int x = 0; ***((((x < 10) && (y-- > 2)) | x == 3))*** *; x++) { }*

- wyrażenie warunkowe **może być puste**
  - ale jeśli metoda posiada za pętlą jakiś kod i wewnatrz petli nie ma breaka to ta reszta kodu jest niedostępna - przez to błąd kompilacji
  - *tylko w javie 6???* - ksiazka nic o tym nie mowi.
- nie można wpisać do warunku słówka **false** (for (...;false;...)) - błąd kompilacji: ciało niedostepne
  - ale można podać zmienną wcześniej zadeklarowaną boolean która ma wartośc false :)

ad 3.

- iteracja następuje **po wykonaniu ciała pętli**
- **można iterować więcej niż jedną** zmienną
  - po iteracji następuje ponowne sprawdzenie warunku
- jeśli sprawdzenie warunku okaże się **false** to natychmiast wychodzi z pętli
- warunek nie jest sprawdzany gdy wewnątrz pętli są polecenia wyjścia: **break, return** lub **System.exit()**
- jeśli wewnątrz pętli nie ma poleceń wyjścia to na ostatnim miejscu wykonania pętli jest sprawdzanie warunku
- **ta sekcja nie musi tak naprawdę nic iterować -** można jej uzyc do wykonania **czegokolwiek** za każdym obrotem pętli
- sekcja ta **może być pusta**

- powyższe trzy sekcje są **od siebie niezależne** i nawet **nie muszą operować na tych samych zmiennych**

Polecenia wyjścia:

break - natychmiast opuszcza pętle i program zaczyna wykonywanie od pierwszej instrukcji po for'ze

return - wychodzi z pętli oraz z metody w której jest (+ **może zwracać** wynik **lub nie musi** gdy metoda jest void)

- gdy metoda jest **void** można wykonac tylko **return;**
- **-** gdy metoda zwraca jakiś typ **trzeba** w return podać typ: **return *zmienna_typu*;**

System.exit(); - wykonanie programu zatrzymuje się, maszyna wirtualna zamyka się

continue - przerywa **tylką bierzącą iteracje** i wykonuje kolejną (sekcja iteracji + kolejne sprawdzenie warunku)

- polecenie continue może wystąpić **tylko** wewnątrz pętli (for, while, do-while)

Przykłady pętli:

for(; ;) { } // **OK** ale gdy za petlą jest kod mój kompilator krzyczy że kod niedostępny - chyba ze jest break w petli
int i = 0; for (;i<10;) { i++; /*do stuff */ } // **OK** - działa podobnie jak while
for (int i = 0,j = 0; (i<10) && (j<10); i++, j++) { } // **OK** iterowanie i deklarowanie wiecej niz 1 zmiennej i rozbudowany test

```
int b = 3;
for (int a = 1; b != 1; System.out.println("iterate")) {  // OK  choć są inne zmienne w sekcjach i brak iteracji w sekcji 3
        b = b - a;   // zmiana warunku
}
// Output: iterate iterate
```

Rozszerzony for - dla tablic i kolekcji

- rozszeżony for posiada **dwie sekcje**

for (***declaration*** **:** ***expression***)

- deklaracja: jest to deklaracja nowej zmiennej **kompatybilnej** (nadklasy też) typem z elementami tablicy (kolekcji), zmienna ta będzie zawierala **element** tablicy.
- **wyrażenie:** tutaj musi się znaleźć **tablica (kolekcja)** lub **metoda zwracająca tablicę (kolekcję)**, tablica może mieć elementy prymitywne, obiektowe lub wielowymiarowe

```
int[] tablica = new int[4];
for(int element : tablica) { print(element); }
```

- można stosować autoboxing:
  for (long l : tablicaDuzychLongow)
- pętla po tablicy wielowymiarowej:

```
        int [][] twoDee = {{1,2,3}, {4,5,6}, {7,8,9}};
        for(int[] n : twoDee)  {   ....   }
```

- przy wielowymiarowych **uważać na**:
  for(***int x2*** : twoDee) ; --> nie można przypisać tablicy to inta.

- można kożystac z "poleceń wcześniejszego wyjścia": np break, continue

5.2.3. Etykiety w pętlach

- **etykiety** PRZED PĘTLĄ tworzymy z dwukropikiem:

```
        nazwa_etykiety:
                for (.....) {}
```

- break z etykietą przerywa działanie pętli z podaną etykietą
  break *nazwa_etykiety;*

- nazwa etykiet za break lub continue mówi którą pętlę przerwać lub kontunuować
  - w zagieżdżoych pętlach

```
boolean isTrue = true;
outer:                                  // etykieta zewnetrznej pętli
      for(int i=0; i<5; i++) {
```

```
            while (isTrue) {
                  System.out.print("Hello");
                  break outer;                    // przerywa pętlę outer a nie while
            } // end of inner while loop
            System.out.println("Outer loop."); // nie dojdzie tu
      } // end of outer for loop
System.out.print("Good-Bye");
```
output: *Hello Goodbye*

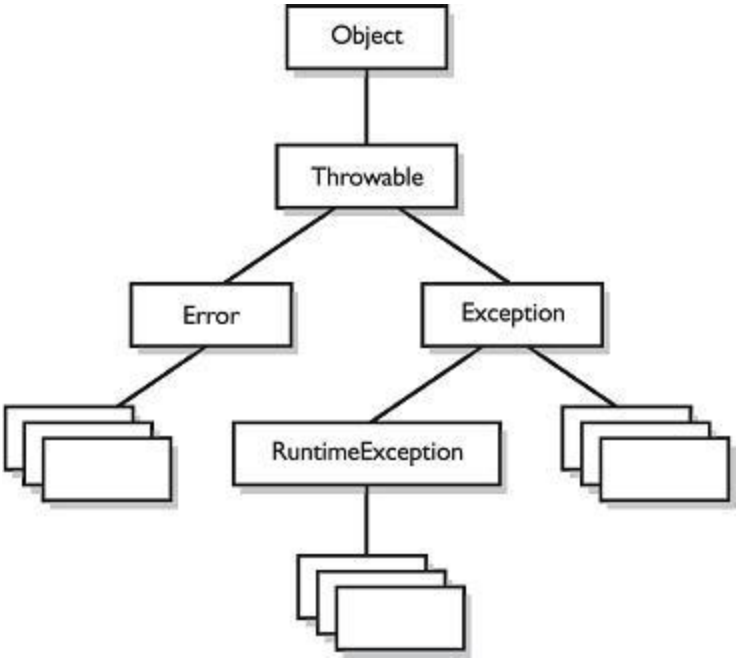
- W powyższym przykładzie gdyby breaka podmienić na continue for byłby przerwany i spowodowało by to wyzwolenie kolejnej iteracji for.
    output: *Hello Hello Hello Hello Hello Goodbye*

- **Uwaga:** etykietowane break lub continue **muszą się znaleźć wewnątrz pętli z nazwą tej etykiety** która jest przy break/continue
  - w przeciwnym razie --> błąd kompilatora

Wyjątki

5.3. Obsługa wyjątków

5.3.1. Hierarchia wyjatków



- **Error** - błędy powstałe w wyniku okoliczności niezwiązanych z błedami w aplikacji czy działania JVM ale nieoczekiwanych jak np awaria sprzetu
  - w większości nie ma możliwości obsłużenia takiego błędu
  - nie ma obowiązku wyłapywania takich błędów - program sie skompiluje
  - technicznie nie są wyjątkami - nie dziedzicza po Exception
- **RuntimeException** - są to błędy programu

- Wszystkie powyższe: **Exception, Error, RuntimeException** i **Throwable mogą zostać wyrzucone poprzez throw** oraz **wyłapane poprzez try/catch**
- metoda printStackTrace() znajduje sie w każdym z wyjątków - wydruk stosu wywołania

Wyjątki kontrolowane i niekontrolowane

- *kontrolowane* (ang. checked) i
- *niekontrolowane* (ang. unchecked).

- Wyjątki **kontrolowane** tym różnią się od niekontrolowanych, że muszą być jawnie obsłużone w metodach,
  -tj. **albo muszą być złapane, albo zadeklarowane** na liście wyjątków rzucanych przez metodę.
- Wyjątki **niekontrolowane** to instancje klas **'Error' i 'RuntimeException'** oraz ich dowolnych podklas.
- Wyjątki **kontrolowane** to instancje klas 'Throwable' i 'Exception' oraz ich podklas, oczywiście z wyłączeniem klas 'Error' i 'RuntimeException'
- Wyjątków **niekontrolowanych nie musimy deklarować czy obsługiwać, ale możemy**

5.3.2. Propagacja niewyłapanych wyjątków

- propagacja czyli "przenoszenie" wyjątku w dół stosu wywołań
  - stos wywołań: metoda wywołuje metodę któa wywołuje metodę - na górze jest ostatnio wywołana
  - gdy wyjatek dojdzie do samego końca stosu i nie zostanie wywołany - aplikacja sie wywala
- gdy wyjatek nie zostanie wyłapany aplikacja wypisuje "dumpa" od metody gdzie wyjatek wystapił do dołu

5.3.3. "Łapanie" wyjatków poprzez try-catch

```
try {
   // risky code

} catch (MyFirstException e1){  // nie wymagane gdy jest finally
   // handler 1

} catch (MySecondException ew){     // nie wymagane gdy jest finally
        // handler 2

} finally {              // opcjonalny
   // cleanup stuff
}
```

- **catch** musi wystąpić **zaraz za blokiem try**
- catch **nie jest wymagany** gdy jest finally
        - try **musi** mieć swój **catch** lub **finally** (lub oba)
- finally jest opcjonalne
- finally musi wystapić **zaraz za catch** (lub try gdy nie ma catch)
- **finally zawsze jest wykonane**
        - niezależnie czy wyjątek wystąpi czy nie a nawet gdy wystąpi i nie zostanie złapany
  - jedynym wyjątkiem od tej reguły jest zatrzymanie JVM nagle lub przez System.exit()

- pomiędzy try a catch lub catch (try) a finally nie może wystąpic żaden kod

- wyjątek jest obiektem **dziedziczącym** po ***java.lang.Exception***
- podczas wyłapania wyjątku poprzez **try** tworzony jest obiekt wyjątku i jest on przekazywany jako argument do **catch**

- catch **wyłapuje wyjatek podany w argumencie** oraz **podtypy tego wyjatku**
  - nie jest jednak zalecane aby wyłapywać *Exception* ponieważ stracimy informacje szczegółowe

- gdy jest więcej catchów, JVM (podczas wystapienia wyjatku) szuka od góry do dołu pasującego argumentu catcha i **wykonuje pierwszy pasujący** (biorąc pod uwagę również podtypy łapanego wyjątku)
  - gdy wyłapujemy wyjątki będące rodzicem i dzieckiem **należy w try/catch napisać pierwszy wyjątek dziecko a potem rodzica**
            - w przeciwnym razie wystąpi **błąd kompilacji** --> wtedy blok catch wyjatku-dziecka będzie zawsze nieosiągalny
- jeśli wystapi wyjatek nie będący wyszczególniony w catch, **nastąpi propagacja wyjątku**

w dół stosu wykonań

5.3.4. Deklaracja wyjątku w metodach - throws

- deklaracja metody ze słówkiem **throws** mówi że dana metoda może wyrzucać wyjątek
  - **wyjatkiem jest tu *RuntimeException*** i jej podklasy

```
void myFunction() throws MyException1, MyException2 { ... }
```

- to że metoda jest zadeklarowana z wyżucanym wyjątkiem nie oznacza ze zawsze taki wyrzuci
- słowo **throws** służy do **propagowania wyjątku** dalej
  - zamiast łapania go try/catch, przekazujemy to łapanie do innej metody - która bedzie ją wywoływac

- **każda metoda musi obsłużyć "checked" exception poprzez try/catch lub słówko throws**
  - zasada ***"handle or declare"***
  - jeżeli metoda wyrzuca wyjątek i ani nie zostanie on obsluzony **try** ani nie będzie zadeklarowana z **throws** - wystąpi **błąd kompilacji**

  - gdy wyjątek jest z serii **unchecked** (czyli RuntimeException i Error i podtypy) nie musza być obsługiwane ani deklarowane
    - np NullPointerException jest unchecked

- metoda main tez moze miec slowko throws (kompilacja - OK) ale to i tak nic nie daje - wyjatek zostanie wyrzucony

Tworzenie własnych wyjątków

- wyjatek musi byc klasą która dziedziczyc po Exception

```
class MyException extends Exception { }
```

Wyrzucenie wyjątku - throw

- **`throw new MyException()`**
- wyrzucony wyjatek **musi być zadeklarowany lub obsłużony**

- można wyżucic ten sam wyjątek wewnatrz catch

```
...
catch (IOException e){

    throw e;
}
```

  - w tym przypadku trzeba go zadeklarować w metodzie (jeśli wyrzucany jest **checked** exception)
  - wszystkie pozostałe catche w tym try sa ignorowane i wykonany jest ewentualnie finally

Błędy typu *Error*

- Error nie jest wyjatkiem
- Podtyp błędu *Error* oraz sam *Error* **mogą byc wyrzucone** za pomoca **throw** lub wyłapane poprzez **try/catch**
    - jest to jednak bez sensu ponieważ sa to błędy które praktycznie uniemożliwiają dalsze działanie programu
        - warto wiedzieć ze można i nie spowoduje to błędu kompilacji
    - **Gdy jest wyrzucony nie musi być obsłuzony ani "zadeklarowany w metodzie"** (tak jak unchecked)

5.3.5. Popularne typy wyjątków

Wyjatki biora się z:
- JVM - wyjątki lub błędy wyrzucane przez JVM
- Programistyczne - czyli wyrzucane przez metody z API lub naszych metod

Poniższe wyjatki są **niekontrolowane (unchecked)**

Wyjatki rzucane przez JVM

NullPointerException
- gdy wywołujemy metodę na obiekcie lub chcemy wejśc do obiektu ktory jest null
    - wyrzucany w trakcie działania aplikacji
- kompilator nie wykrywa go

ArrayIndexOutOfBoundsException
-rzucany jest, gdy nastąpi odwołanie do indeksu tablicy, który jest z poza dopuszczalnego zakresu – tj. jeśli indeks jest liczbą ujemną lub większą bądź równą (numerujemy od zera) od ilości elementów w tablicy.

StackOverflowError
- gdy skonczy sie miejsce w pamięci na stos wywołania
- częste w przypadku wykonywania metod w nieskończonej pętli (lub np w rekurencji)

NoClassDefFoundError
- jest rzucany, gdy Wirtualna Maszyna Javy – a mówiąc precyzyjniej, mechanizm ładowania klas – nie może znaleźć odpowiedniej klasy.
    - przeważnie błąd ten spowodowany jest **brakiem odpowiedniej biblioteki** na ścieżce przeszukiwania (ang. **classpath**).

ClassCastException
- jest efektem próby rzutowania obiektu na typ z nim nie kompatybilny.

ExceptionInInitializerError
- jest rzucany, gdy wystąpi błąd w trakcie statycznej inicjalizacji zmiennej, bądź w trakcie wykonania bloku inicjalizacyjnego.

Wyjatki rzucane przez API Javy

NumberFormatException (extends IllegalArgumentException)
- podczes parsowania Stringa i przekształcania go na liczbe (we wrapperach)

IllegalStateException
- oznacza, że stan urządzeń bądź zewnętrznych systemów, których nasz program używa jest nieprawidłowy i wykonywana operacja nie może być z tego powodu zakończona.

AssertionError
- informuje, że nie jest prawdziwa jedna z asercji. Asercje będą tematem następnego artykułu, tak więc do tego tematu jeszcze powrócimy.

IllegalArgumentException
- wyrzucany gdy metoda otrzymuje argument sformatowany inaczej niż oczekuje tego metoda

Asercje

5.4. Praca z mechanizmem *asercji*

5.4.1. Używanie asercji

- Asercje pozwalają na testowanie założeń podczas developmentu

Np. zakładamy ze x będzie zawsze większe/równe zero. Jest to założenie
Zamiast pisać if lub obejmować je w try/catch można wykonać:

```
private void methodA(int num) {
      assert (num>=0);  // throws an AssertionError
                                    // if this test isn't true
      useNum(num + x);
}
```

Forma z dodatkową informacją

- za asercją **po dwukropku** można dopisac **string doklejany do *stack trace***'a

```
private void doStuff() {
      assert (y > x)): "y is " + y + " x is " + x;
      // more code assuming y is greater than x
}
```

- asercje są **domyslnie wyłączone** a więc powyższa metoda zawiera w domyślnym

przypadku tylko wywołąnie useNum(num+x);
  - assert jest wówczas niewidoczny dla kompilatora
  - w praktyce asercje wykorzystywane są podczas developmentu i testowania a w wersji produkcyjnej są wyłączone
- w asercjach **zawsze zakładamy że coś jest true**
- w przypadku gdy asercja okaże sie nieprawdziwa wystepuje wyjątek czasu wykonania **AssertionError**
  - tego wyjatku **nie należy wyłapywać**
  - wyjątek ten zatrzymuje program

- w założeniu **musi** pojawić się wyrażenie typu **boolean**
  - true - OK, false - AssertionError
- w wyrażeniu po dwukropku może byc czymkolwiek **co da się przekształcic w String**
  - podobnie jak w metodzie: System.out.println()

Legalne zapisy
```
assert(x == 1);
assert true;
assert(x == 1) : aReturn();
assert(x == 1) : new ValidAssert();
```

Nielegalne zapisy
```
assert(x = 1);                  // da  w wyrażeniu 1 (czyli inta)
assert(x == 1) : ;              // puste wyr. za ":" nie da żadnej wartości
assert(x == 1) : noReturn();    // metoda void nie ma wartości
assert(x == 1) : ValidAssert va;  // brak wartości
```

- na egzaminie gdy w pytaniu nie jest wyspecyfikowane o które wyrazenie chodzi - chodzi o 1. boolowskie

5.4.2. Włączanie asercji

- wymaganie: Java 1.4 lub wyższa
  - w wersjach wcześniejszych nie było słówka **assert**
  - w wersji 1.4. assert mogło byc słowem kluczowym albo identyfikatorem dla zmiennych
    - wymagało to podania podczas uruchomienia
  - w nowej wersji (1.5 i nowszej) kompilator *javac* **zawsze** traktuje **assert** jako słowo kluczowe

Kompilowanie kodu starej Javy

- Java 5 domyślnie bierze słówo assert jako kluczowe chyba że poda się inaczej
  - gdy znajdzie zmienna o takiej nazwie zgłosi błąd kompilacji

`javac -source 1.3 OldCode.java` --> słowo assert nieznane - może byc uzyte jako zmienna
`javac -source 1.4 NotQuiteSoOldCode.java` --> assert jako słowo kluczowe
`javac -source 1.5 NewCode.java` (lub `-source 5`) --> assert jako keyword

Uruchamianie kodu z asercjami

- asercje wkompilowane w kod **są dostepne po włączeniu podcas uruchamiania**
  - domyslnie wyłączone

Uruchomienie klasy z **włączona** asercją:

```
java -ea com.geeksanonymous.TestClass
```
lub
```
java -enableassertions com.geeksanonymous.TestClass
```

Uruchomienie klasy z **wyłączona** asercją (domyslne):

```
java -da com.geeksanonymous.TestClass
```
lub
```
java -disableassertions com.geeksanonymous.TestClass
```

Selektywne wyłączanie/włączanie asercji

Switche: -ea lub -da
Argumenty po dwukropku -ea:argument
- switch bez argumentu: dotyczy wszystkich klas
- switch z nazwą pakietu: dotyczy tylko klas w tym pakiecie i podpakietach
- switch z nazwą klasy: dotyczy tylko podanej klasy

Przykłady:
`java -ea -da:com.geeksanonymous.Foo` - włącza asercje wszedzie prócz klasy Foo
`java -ea -da:com.geeksanonymous...` - włącza asercje wszędzie prócz pakietu

5.4.3. Porządane i nieporządane stosowanie asercji

Nie wszystkie <u>legalne</u> zastosowania asercji <u>sa porządane</u>
Słówko ***appropriate*** na egzaminie <u>nie oznacza</u> legalne
Poniższe zasady mówią co "nie powinno się" robic a nie czego "nie można"

<u>Nieporządane:</u>
- nie powinno sie obsługiwac AssertionError
- nie uzywac asercji do walidowania **argumentów w publicznych metodach**
```
public void doStuff(int x) {
        assert (x > 0);          // nieporządane
```
  - należy samemu zapewnić aby metoda radziła sobie z argumentami
    - lepiej wykorzystac do tego IllegalArgumentException
  - w prywatnych metodach - OK
- nie uzywac asercji do **walidacji argumentów wywołania programu**
- nie używać wyrażeń asercji mogących przynieść efekt uboczny (efekt dodatkowy)
  - asercja powinna pozostawic program w tym samym stanie

```
assert (modifyThings());  ...
boolean modifyThings() { y=x++; return true;}
```

<u>Porządane:</u>

- **w prywatnych** metodach uzywanie asercji do argumentów jest **porządane**
- asercje **mozna** używac w **switch/case**:

```
switch (x){
        case 1 :.....
        default: assert false;    // tu nie powinno sie dojsc
      }
```

TEST 9/16 (56%)
źle: 1 D-> powinno byc E (glupie niedopatrzenie)
3 AD->AD**EF (bo runtime exception moze byc rzucane w metodach overload mimo ze nie ma ich w nadklasie)**
8 ABC --> AC;
12 ACDF--> ADF bo nie można zadeklarowac zmiennej poza "nowym forem": int y=0; for (**y:**tab)
13 B-->F
15 G--> E bo null+ " " zmienia sie w stringa "null"
16 nic --> BCD - powtórzyc wyjatki API i JVM

---

6. Formatowanie i parsowanie Stringów i I/O

6.1. Klasa String

Definiowanie Stringa
- String s = new String(); s = "abcdef";
- String s = new String("abcdef");
- String s = "abcdef";

- Klasa String **jest final** więc nie można przedefiniowywać (override) jej metod

Immutable
- Stringi są obiektami "**immutable** czyli "niezmienialne"
  - podczas zmiany (np konkateacji) stringa, tworzony jest nowy a następnie referencja jest przepisywana na ten nowy

-1-

```
String s ="abcdf";
String s1 = s;

s = s.concat(" ghijk");    // pod zmienna s zapisany jest nowy obiekt String z nową wartością

System.out.println(s);  // wypisze "abcdef ghijk"
```

```
System.out.println(s1); // wypisze "abcdef"
```
-2-
```
String x = "Java";
x.concat(" Rules!");  // jest stworzony nowy obiekt Stringa ale nie jest przypisany do żadnej referencji więc ginie
System.out.println("x = " + x); // wypisze "x = Java"
```

- obiekt który jest stworzony podczas zmainy stringa i **nie jest** przypisany do zadnej referencji jest niszczony przez GC
- nowe obiekty stringowe sa tworzone nawet podczas podawania argumentów w formie "abc" i po uzyciu 'tracone'

Zarządznie pamięcią w String

- JVM posiada "String constant pool"
- Gdy jest tworzony nowy string JVM sprawdza czy juz taki sam istnieje w pamięci - jesli tak to referencja wskazuje na ten obiekt
  - wtedy obiekt ma kolejna referencje na siebie
  - nie tworzy wtedy nowego obiektu

- wywołanie:
  String s = new String("abc");
  stworzy 2 *(??)* obiekty stringowe - nowy w normalnej pamięci - "non-pool" i "abc" który będzie stworzony w "poolu"

*Ważne **metody klasy String***

- *char* **charAt**(int index)
  Returns the character located at the specified index - indexed from 0

- *String* **concat**(String s)
  Appends one String to the end of another ("+" also works)
  Podomnie działa + i +=

- *boolean* **equalsIgnoreCase**(String s)
  Determines the equality of two Strings, ignoring case
  Metoda **equals** jest case-sensitive

- *int* **length**()
  Returns the number of characters in a String

- *String* **replace**(char old, char new)
  Replaces all occurrences of an character in *old* string with a *new* character

- *String* **substring**(int begin) String **substring**(int begin, int end)
  Zwraca część stringa od *begin* (**liczone od 0**) do końca lub do podanego *end* (**liczonego od 1**) włącznie

- *String* **toLowerCase**()
  Returns a String with uppercase characters converted

- *String* **toUpperCase**()
  Returns a String with lowercase characters converted

- *String* **toString**()
  Returns the value of a String - jest bo zostala odziedziczona po Object

- *String* **trim**()
  Removes whitespace from the ends of a String " str " --> "str"

6.1. Klasy StringBuffer i StringBuilder

- StringBuffer i StringBuilder są mutable a więc nie ma strat pamięci podczas wielu operacjach na Stringach
- Zostały stworzone aby zaczytywac pliki tekstowe
- StringBuilder jest od Javy5 i ma takie samo API jak StringBuffer
- StringBuilder **nie jest** *thread safe* // różnice
- StringBuilder **jest szybszy** // różnice
- Sun **zaleca wybór StringBuildera** tam gdzie to jest mozliwe

- Operacje na StringBuilder są operacjami na jego wartości i zmieniają wartośc a nie tworza nowy obiekt jak w String

```
StringBuffer sb = new StringBuffer("abc");
sb.append("def");                    // operuje na wartości i dodaje do niej "def"
System.out.println("sb = " + sb);     // wypisze "sb = abcdef"
```

- StringBuffer i StringBuilder można "łańcuchować" (podobnie jak String):

```
sb.append("def").reverse().insert(3, "---");
```

Ważne ***metody klas StringBuffer i StringBuilder***

- synchronized StringBuffer **append**(String s)
  Jako argument może przyjmować tez boolean, char, double, float, int, long i inne
  Dopisuje do łańcucha znaków ten podany w argumencie

- StringBuilder **delete**(int start, int end)
  Wycina z bierzącego łańcucha znaki od *start* **(liczonego od 0)** do *end* **(liczonego od 1)**

- StringBuilder **deleteCharAt**(int index)
  Wyrzuca znak na podanej pozycji index

- StringBuilder **insert**(int offset, String s)
  Wkleja podany *String s* do bierzącego łańcucha pomiędzy *offset* (**liczonego od 0**) a nastepny znak
  "01234567" --> sb.insert(4, "---"); --> "0123---4567"

- synchronized StringBuffer **reverse**()
  Odwraca wartość łańcucha --> od tyłu

- public String **toString**()
  Podaje wartość łańcucha StringBuilder i StringBuffer

- public int **length**()
  Długość łańcucha

- *char* **charAt**(int index)

6.3. Operacje na plikach

6.3.1. Ważniejsze klasy - nawigacja I/O

File
- abstrakcyjna **reprezentacja pliku** lub ścieżki do **katalogu**
- nie jest uzywana do zapisu/odczytu plików
- do tworzenia pustych plików, szukania plików, usuwania, pracy na ścieżkach

FileReader
- klasa niskiego poziomu
  - używana do czytania znaków z pliku; służy temu metoda ***read()***
- często jest "opakowywana" innymi klasami (wrapperami) np BufferedReader

BufferedReader
- służy łatwiejszemu uzyciu klas niskiego poziomu jak FileReader
- w przeciwieństwie do FileReadera, potrafi **czytać duże porcje** danych z pliku na raz
- trzyma dane w buforze dzięki czemu redukuje częsty dostęp do pliku
- posiada wygodniejszą metodę ***readLine()*** czytającą cały wiersz i zwraca null gdy brak wiersza

FileWriter
- klasa niższego poziomu
  - używana do zapisywania znaków z pliku; służy temu metoda ***write()***
- często jest "opakowywana" innymi klasami (wrapperami) np BufferedWriter

BufferedWriter
- służy łatwiejszemu uzyciu klas niskiego poziomu jak FileWriter
- w przeciwieństwie do FileWritera, potrafi **zapisać duże porcje** danych do pliku na raz
- redukuje częsty dostęp do pliku
- posiada metodę ***newLine()*** wpisującą do pliku znak końca linii

PrintWriter
- Print formatted representations of objects to a text-output stream
- dzięki metodom format(), printf(), append() zapis jest bardziej elastyczny
- println()

- **Klasy strumieniowe (Stream)** zapisują/czytają **bajty,** natomiast klasy typu **Reader i Writer** czytaja/zapisują **znaki**

Tabela 6.3.1: Funkcji i klas:

java.io Class	Extends From	Key Constructor(s) Arguments	Key Methods
File	Object	File, String String String, String	createNewFile() delete() exists() isDirectory() isFile() list() mkdir() renameTo()
FileWriter	Writer	File String	close() flush() write()
BufferedWriter	Writer	Writer	close() flush() newLine() write()
PrintWriter	Writer	File (as of Java 5) String (as of Java 5) OutputStream Writer	close() flush() format()*, printf()* print(), println() write()
FileReader	Reader	File String	read()
BufferedReader	Reader	Reader	read() readLine()

6.3.2. Tworzenie plików

Klasa **File** słuzy do reprezentowania pliku (lub katalogu) ale **nie jego zawartości.**
Znajduje sie w pakiecie:

```
import java.io.*;  // The Java 5 exam focuses on
                   // classes from java.io
```

Tworzenie "handlera" pliku/katalogu:

```
File file = new File("fileWrite1.txt"); // There's no file yet!
```

- jeśli plik nie istnieje - **nie tworzy go**
- jeśli plik istnieje - **tworzy do niego odnośnik** (handler)

Metody tworzące plik:

- **większość operacji na plikach należy objąć w klauzulę try / catch(IOException)**

- tworząc obiekt File z nazwą pliku **nie tworzy się fizycznego pliku** a jedynie jego **nazwę**
- ***boolean*** **`file.exists()`**
  -sprawdza czy plik (na który wskazuje obiekt) fizycznie istnieje
- ***boolean*** **`file.createNewFile()`**
  - tworzy nowy plik **jeżeli nie istnieje**
    - zwraca: true - udało się stworzyć, false - nie udało się bo np plik już istniał

6.3.3. Użycie FileWriter i FileReader

Klas tych używa się żadko ponieważ istnieją "wrappery" które robią to lepiej

```
File file = new File("fileWrite2.txt");  // obiekt File, handler do pliku
FileWriter fw = new FileWriter(file);  // stworzenie fizycznie pustego pliku i stworzenie FileWritera

fw.write("howdy\nfolks\n");            // zapis 2 wierszy do pliku - 12 znaków
fw.flush();                                                    // flush
fw.close();                                                    // zamykanie

FileReader fr = new FileReader(file);   // FileReader otwiera plik na dysku

char[] in = new char[50];    // tablica ze znakami z pliku - każdy znak w osobnej komórce
int size = fr.read (in) ;    //  czyta cały plik i zwraca jego wielkość (ilość znaków = 12)
   // czytanie

for(char c : in)  System.out.print(c);    // czytanie znak po znaku

fr.close();
```

- metoda ***flush()*** przy przetwarzaniu strumieniowym zapewnia że wszystkie dane ze strumienia zostaną wpisane do pliku
- metoda ***close()*** zamyka plik - czyli zwalnia zasób komputera jakim jest ten plik

Wady powyższego kodu
- używając zapisu musimy nowe linie pisać ręcznie jako "\n"
- odczytując dane zapisujemy je do tablicy znak po znaku co może spowodowac przepełnienie tablicy oraz jest trudne w czytaniu

- **z tego powodu** lepiej korzystać z **Buffered Writer** i **Bufered Reader**

6.3.4. Wykorzystanie klas I/O

- Klasy w IO wykorzystują mechanizmy:
  - wrapping
  - chaining
- Oznacza to że można wykorzystywać klasy I/o (**tabela 6.3.1**) do łączenia i rozszerzania funkcjonalności
  - szukamy dogodnej metody, tworzymy obiekt a argumentem konstruktora jest klasa bardziej ogólna
    - czesto w konstruktorach podana jest nadklasa (np Writer) gdzie można podać

podklase (np FileWriter)

Kombinowanie klas do zapisu:

```
File file = new File("fileWrite2.txt"); // create a File object
FileWriter fw = new FileWriter(file);   // create a FileWriter
                                        // that will send its output to a File

PrintWriter pw = new PrintWriter(fw);   // create a PrintWriter
                                      // that will send its output to a Writer

pw.println("line1");                  // write the data line
pw.println("line2");
pw.flush();                             // trzeba wyczyscic bufor!!
pw.close();                           // trzeba zamknac writera
```

Kombinowanie klas do odczytu:

```
File file = new File("fileWrite2.txt"); // create a File object AND
                                      // open "fileWrite2.txt"
FileReader fr = new FileReader(file);   // create a FileReader to get
                                        // data from 'file'
BufferedReader br = new BufferedReader(fr);  // create a BufferReader to
                                             // get its data from a Reader
String data = br.readLine();          // read some data line or in while loop:
while( (data = br.readLine()) != null) { .... reading more  ... }
br.close();                         // trzeba zamknac Readera
```

- podczas odczytu nie mamy flusha
  - Reader **nie posiada metody flush()**
- readera **trzeba zamknąć**

6.3.5. Tworzenie plików i katalogów

Obiekt File:

```
File file = new File("fileWrite1.txt");
```

- jeśli plik nie istnieje - **nie tworzy go**
- jeśli plik istnieje - **tworzy do niego odnośnik** (handler)

- metoda **`file.isFile()`** zwraca true/false gdy plik isnieje/nie istnieje

Pliki

Aby otworzyć plik fizycznie można zrobic jedną z 2 operacji:

- wywołać metodę na obiekcie typu `File` file.***createNewFile();***
- utworzyć instancję jednej z klas pochodnych od **Reader**, **Writer** lub **Stream**
  - w szczególności: ***FileWriter***, ***PrintWriter***, ***FileInputStream*** lub

FileOutputStream

- instancje powyższych obiektów tworzą plik (o ile ten nie istnieje)

Katalogi

Aby utwożyć fizycznie katalog należy:

- wywołać metodę na obiekcie typu `File` file.***mkdir();***

```
File myDir = new File("mydir");
```

Zapis **pliku** w takim katalogu można wykonać za pomocą:

```
File myFile = new File(myDir, "myFile.txt");
myFile.createNewFile();
```

- przykładowo, ponieważ można wykorzystac myFile np we Writer

Nie stworzenie katalogu a nastepnie próba zapisu w nim pliku spowoduje wyjątek IOException

- **trzeba** wywoływać metodę **mkdir()** na obiekcie typu File zanim stworzy sie nim plik

```
File myDir = new File("mydir");
// myDir.mkdir();                    // Brak tworzenia katalogu
File myFile = new File(myDir, "myFile.txt");
```

// utworzenie handlera do pliku w katalogu OK ale:

```
myFile.createNewFile();
```

// próba utworzenia pliku spowoduje **błąd**

- metoda **`myDir.isDirectory()`** zwraca true/false gdy katalog istnieje

Inne operacje na plikach i katalogach

- kasowanie:
  myFile.delete(); // kasowanie pliku
  myDir.delete(); // kasowanie katalogu **pustego** - niepusty **nie może byc skasowany**

- zmiana nazwy pliku:
  - wymaga stworzenia nowego handlera (w tym samym miejscu) i podania go w argumencie
    - **nie potrzeba tworzyc pliku** fizycznie poprzez *createNewFile()*
  - gdy newName byłby null - wyskoczyłby NullPointerException

```
File newName = new File("newName.txt");   // lub  File newName = new File(myDir, "newName.txt");
myFile.renameTo(newName);
```

- zmiana nazwy katalogu:
  - także wymaga stworzenie handlera (bez fizycznego katalogu)
  - zawarte w katalogu pliki pozostaną w nowym katalogu - zmiana nazwy!
    - mozna zmieniac nazwę pustych i niepustych katalogów

```
File newDir = new File("newDirectory");
myDir.renameTo(newDir);
```

- listowanie (wyszukiwanie) plików i katalogów

```
String[] files = new String[100];
```

// potrzebna tablica do zapisania wyników listowania

```
File search = new File("searchDir");
```
// katalog który chcemy wylistować - tworzymy do niego handler

```
files = search.list();
```
// listowanie zapisane w tablicy Stringów

Więcej informacji o I/O znajduje sie w API ---> **zajrzec do API Javy**

6.4. Serializacja

6.4.1. Serializacja obiektów - ObjectOutputStream i ObjectInputStream

- podstawowa serializacja to uzycie dwóch metod z klas:

- serializacja i zapis
  ObjectOutputStream.**writeObject**() throws **IOException**

- odczyt i deserializacja
  ObjectInputStream.**readObject**() throws **IOException, ClassNotFoundException**

- są to klasy wysokiego poziomu co oznacza że opakowuja klasy takie jak **FileOutputStream** i **FileInputStream**
- klasa serialzowana powinna implementować interfejs **Serializable**
- **trzeba obsłużyć wyjątki** podane wyżej

Przykład serializacji obiektu:

```
class ObiektKlasa implements Serializable { }
```
// serializowana klasa implementuje interfejs Serializable

```
FileOutputStream fs = new FileOutputStream("testSer.ser");
ObjectOutputStream os = new ObjectOutputStream(fs);
os.writeObject(obiekt);   // następuje serializacja
os.close();
```

Przyklad deserializacji obiektu:

```
FileInputStream fis = new FileInputStream("testSer.ser");
ObjectInputStream ois = new ObjectInputStream(fis);
obiekt = (ObiektKlasa) ois.readObject(); // nastepuje deserializacja i zapis do obiektu
ois.close();
```

- mechanizm serializacji obsługuje cały graf obiektów tzn są serializowane wszystkie obiekty powiązane
  - gdy serializowany obiekt posiada referencję do innego obiektu to oba te obiekty sa

automatycznie serializowane
- wystarczy zserializować tylko 'obiekt glowny' a wszystkieobiekty w dół grafu zserializują sie same
- wymagane jest aby te **wszystkie obiekty (główny i powiązane) muszą implementowac *Serializable***
- brak implementacji **Serializable** w obiekcie spowoduje ***runtime exception***: java.io.NotSerializableException

- w przypadku gdy nie ma możliwości zaimplementować Serializable w obiekcie (np. nie ma dostepu do kodu) można:
1. odziedziczyć taką klasę - ale nie zawsze jest to możliwe
2. oznaczyć referencje do powiązanego obiektu jako **transient**
- spowoduje to ominięcie takiego obiektu w procesie serializacji
- podczas deserializowania **obiekt taki będzie null**
- **chyba że:**

- istnieje **zestaw prywatnych metod wywoływanych podczas serializacji i deserializacji** automatycznie
- w przypadku obiektów **transient** aby była możliwość ich instancjonowania po deserializacji/serializacji

Te metody nie należą do żądnego interfejsu (są tak ustalone w specyfikacji) - **muszą wyglądac dokładnie tak:**

```
private void writeObject(ObjectOutputStream os) {
        // kod wywoływany podczas serializacji
    // defaultWriteObject() – komenda serializacji obiektu
  }

  private void readObject(ObjectInputStream os) {
        // kod wykonywany podczas deserializacji
    // defaultReadObject() – komenda serializacji obiektu
  }
```

- wewnatrz metody **writeObject() trzeba wywołać**

```
os.defaultWriteObject();
```

aby dokonac normalnego procesu serializacji dla tego obiektu.
- za lub przed tym wywołaniem można zapisać w strumieniu inne dane
- inne dane zapisuje sie odpowiednimi metodami:
os.**writeInt**(int a), os.**writeDouble**, os.**writeObject** itd

- wewnątrz **readObject() trzeba wywołać**:

```
is.defaultReadObject();
```

- w takiej samej kolejności jak `defaultWriteObject();` powyżej
- w tej samej kolejnosci należy dane odczytywać:
- inne dane odczytuje sie odpowiednimi metodami zwracającymi poszczególne typy:
os.**readInt**(), os.**readDouble**, os.**readObject** itd

- metody te służyć mogą do zapisu (writeObject) stanu obiektów ***ulotnych*** (**transient**)

oraz inicjalizowania (readObject) ich z odczytanym stanem
- ważne jest aby wysoływać w nich metody **`defaultWriteObject()`** i **`defaultReadObject()`** umożliwiające normalny proces serializacji obiektu nieulotnego
- komendy serializacji i deserializacji jak również odczyt danych w *readObject* **musi byc w takiej samej kolejności** jak we *writeObject*.
- inna kolejność spowoduje uszkodzenie strumienia i **runtime exception**: *StreamCorruptedException*

- Obiekt deserializowany **zachowuje się inaczej** od obiektu nowo-instancjonowanemu (new)
- **nie jest** wywoływany konstruktor
- zmienne **nie są** inicjalizowane wartościami domyślnymi
- zmienne transient moga być zainicjalizowane z wartościami domyślnymi gdy nie zaimplementujemy *defaultReadObject()*

```
class Bar implements Serializable {
        transient int x = 42;
} // po deserializacji x będzie miało wartość 0
```

- zmienne referencyjne ulotne (transient) są ustawiane na null (gdy nie ma *defaultReadObject*)
- podczas serializacji kolejcji lub tablicy **każdy z elementów kolekcjimusi być Serializable**
- każdy pojedynczy element nie będący Serializable spowoduje błąd serializacji

- **zmienne statyczne NIE SĄ SERIALIZOWANE** bo nie są to zmienne obiektu (jego stanu) lecz klasy.

6.4.2. Serializacja i dziedziczenie

Podklasy:

- Zgodnie z zasadami Javy **wszystkie klasy dziedziczące** po klasie implementującej interfejs Serializable **również implementują ten interfejs** (IS-A Serializable).
- Nie można zaznaczyc klasy jako nie-serializable jeżeli któraś z nadklas była Serializable
- jeśli jakaś z poklas implementuje Serializable to każda z podklas tez go implementuje

Nadklasa:

Gdy nadklasa nie jest Serializable i serializujemy podklase:
- podczas deserializacji - nieserializowana część nadklasy jest **inicjalizowana na nowo**
- w tym przypadku zmienne instancyjne odziedziczone z nadklasy **zostaną zaininicjowane z wartościami domyślnymi lub przypisanymi**
- konstruktor nadklasy (nieserializowanej) **jest wywoływany** (razem z konstruktorami kolejnych nadklas: super())

6.5. Obiekty Daty, Numery, Waluty - Date, Number, Currency

6.5.1. Klasy

java.util.**Date**
- większość metod jest *deprecated*
- jest klasą niezmienialną - **mutable**
- repreentuje czas w milisekundach od 1.01.1970

reprezentuje czas java.util.**Calendar**

- klasa służąca do konwertowania i manipulowania datami

java.text.**DateFormat**
- służy do formatowania dat na różne sposoby
- przyjmuje **Locale** do formatowania dat w stylu danego kraju

java.text.**NumberFormat**
- formatowanie liczb

java.util.**Locale**
- łączy się z NumberFormat i DateFormat i Currency tworząc obiekty specyficzne dla danego kraju

Sposoby uzycia powyższych klas:

Use Case	Steps
Get the current date and time.	1. Create a Date: `Date d = new Date();` 2. Get its value: `String s = d.toString();`
Get an object that lets you perform date and time calculations in your locale.	1. Create a Calendar: `Calendar c = Calendar.getInstance();` 2. Use `c.add(...)` and `c.roll(...)` to perform date and time manipulations.

Get an object that lets you perform date and time calculations in a different locale.	1. Create a Locale:
	`Locale loc = new Locale(language);` or
	`Locale loc = new Locale(language, country);`
	2. Create a Calendar for that locale:
	`Calendar c = Calendar.getInstance(loc);`
	3. Use `c.add(...)` and `c.roll(...)` to perform date and time manipulations.

Get an object that lets you perform date and time calculations, and then format it for output in different locales with different date styles.	1. Create a Calendar:
	`Calendar c = Calendar.getInstance();`
	2. Create a Locale for each location:
	`Locale loc = new Locale(...);`
	3. Convert your Calendar to a Date:
	`Date d = c.getTime();`
	4. Create a DateFormat for each Locale:
	`DateFormat df = DateFormat.getDateInstance (style, loc);`
	5. Use the format() method to create formatted dates:
	`String s = df.format(d);`

Get an object that lets you format numbers or currencies across many different locales.	1. Create a Locale for each location:
	`Locale loc = new Locale(...);`
	2. Create a NumberFormat:
	`NumberFormat nf = NumberFormat.getInstance(loc);`
	`-or- NumberFormat nf = NumberFormat. getCurrencyInstance(loc);`
	3. Use the format() method to create formatted output:
	`String s = nf.format(someNumber);`

6.5.2. Date

- większość metod - ***deprecated***
- trzymana ze wzgledu na kompatybilność wsteczną
- może służyć jako **klasa-most** do innych klas takich jak Calendar czy java.**sql**.Date
- data jest reprezentowana jako ilość milisekund on 1.01.1970
- **Tworzenie nowej daty** - teraźniejszej:

```
Date now = new Date();
```
`now.getTime()` - wypisuje liczbe milisekund
`now.toString()` - wypisze date wg Locale aktualnej maszyny

- zalecane używanie klasy **Calendar**

6.5.3. Calendar

- jest klasą **abstrakcyjną**
- aby stworzyć **instancję** klasy Calendar należy użyć metody fabrykującej

```
Calendar cal = Calendar.getInstance();
```

- w takim przypadku nie wiadomo do końca instancję jakiej podklasy otrzymano
- najpewniej będzie to java.util.**GregorianCalendar**

Metody:

- **setTime**(Date date) **ustawienie czasu obiektu Calendar**
- **pola:** SUNDAY, MONDAY, DAY_OF_WEEK itd
- **getFirstDayOfWeek**() - sprawdza co jest pierwszym dniem tyg: zwraca stałą (**pole**)
- **add**() - dodawanie czasu

```
c.add(Calendar.HOUR, -4);           // subtract 4 hours from c's value
  c.add(Calendar.YEAR, 2);                  // add 2 years to c's value
  c.add(Calendar.DAY_OF_WEEK, -2);     // subtract two days from c's value
```

- roll() - dodawanie czasu bez zmiany "większych części daty"
- `roll(Calendar.MONTH, 9)` - doda 9 miesięcy do daty ale nie zmieni to pola YEAR
- w przeciwieńswie do add() która po dodaniu wartości zmienia całą datę, roll() dodaje tylko wartość do tego pola
  - roll: October 8, 2001 --> roll(Calendar.MONTH, 9) --> July, 08 2001
  - add: October 8, 2001 --> add (`Calendar.MONTH, 9` --> July, 08, **2002**
- roll() po dodaniu godzinnie zmienia dnia, miesiąca, roku (czyli pól 'większej czesci daty')

- zapamiętać pola klasy Calendar
- patrz API

6.5.4. Formatowanie dat - DateFormat

- klasa **abstrackcyjna** z dwiema metodami fabrykującymi

```
DateFormat.getInstance()
DateFormat.getDateInstance();  // jako argument przyjmuje kilka typów
```

*Metoda **format***()

String **format**(Date date);

- Pobiera w argumencie obiekt Date i zwraca sformatowanego Stringa

Obiekty typu **DateFormat**:

dfa0 = DateFormat.getInstance();
dfa1 = DateFormat.getDateInstance();
dfa2 = DateFormat.getDateInstance(DateFormat.SHORT);
dfa3 = DateFormat.getDateInstance(DateFormat.MEDIUM);
dfa4 = DateFormat.getDateInstance(DateFormat.LONG);
dfa5 = DateFormat.getDateInstance(DateFormat.FULL);

- wykonanie na powyższych obiekatch metody format z tą sama datą
wyświetli nastepujące **style dat**:

0: 9/8/01 7:46 PM
1: Sep 8, 2001
2: 9/8/01
3: Sep 8, 2001
4: September 8, 2001
5: Saturday, September 8, 2001

*Metoda **parse***()

Date **parse**(String formatedDate) **throws ParseException**

- jest metodą klasy **DateFormat** (na obiekcie tego typu wykonujemy format)
- metoda bierze w argumencie Stringa ze sformatowaną datą przez DateFormat i
konwertuje tego Stringa na obiekt Date
- w zależności od formatu daty **precyzja sparsowanej daty jest różna**:

np: DateFormat.SHORT : 9/8/01 nie posiada godziny więc po sparsowaniu godzina jest 00:00:

Sat Sep 08 **00:00:00** MDT 2001

- trzeba obsłużyć wyjątek ParseException

6.5.5. Locale

- Internationalization - i18n
- jest wykorzystywana w DateFormat czy NumberFormat
- API Javy mówi: *locale is a specific geographical, political, or cultural region.*

```
Locale(String language)
```

```
Locale(String language, String country)
```

- argument *language* jest podawany w ISO 639 Language Code i ISO Country
  - np. Polski: "pl"
  - ponad 500 kodów

np: język włoski używany w szwajcarii:

```
Locale locBR = new Locale("it", "CH");
```

co da na wyjściu datę: sabato, 1. ottobre 2005

np. język włoski:

```
Locale locPT = new Locale("it");
```

co da na wyjściu datę: sabato 1 ottobre 2005

Formatowanie dat zgodnie z Locale:

Włoski format pełny (FULL):

```
DateFormat dfIT = DateFormat.getDateInstance(
                                DateFormat.FULL, new Locale("it", "IT") );
      dfIT.format( new Date() );
```

Output: domenica 14 dicembre 2010

- Date Format i NumberFormat **moga otrzymać Locale tylko w momencie instancjonowania**
  - nie ma metod które moga zmienić potem to Locale

Niektóre metody w Locale

getDisplayCountry() - Wypisuje kraj w Locale
getDisplayCountry(Locale loc) - jak wyżej ale wypisuje to w języku podanego w argumencie locale
getDisplayLanguage() - analogicznie - wypisuje język
getDisplayLanguage(Locale loc) - analogicznie

6.5.6. Formatowanie numerów i walut - NumberFormat

- jest to klasa **abstrakcyjna** tworzona przez metody fabrykujące
- służy do formatowania **numerów i walut**

getInstance()
getInstance(*Locale* loc)
getCurrencyInstance()
getCurrencyInstance(*Locale* loc)

Inne metody NumberFormat

- Klasa ta posiada metodę **format** która formatuje liczby na standard podany w Locale

String **format**(*double* d)
String **format**(*long* l)

- standardowo metoda format **zaokrągla** liczby do 3 miejsc do przecinku, ale wartość tę możemy zmienić:

void **setMaximumFractionDigits**(int value)
int **getMaximumFractionDigits**()
- podajemy w value ilosc miejsc po przecinku do **zaokrgląglenia matematycznego**

- parsowanie stringa w celu zwrócenia Number

Number **parse**(*String formatedNumber*) **throws ParseException**

- dodatkowo metoda
setParseIntegerOnly(*boolean f*)
powoduje że parsowanie bierze tylko **całkowitą część parsowanego stringa** i zamienia to na numer

Podsumowanie:

Class	Key Instance Creation Options
util.Date	new Date(); new Date(long millisecondsSince010170);
util.Calendar	Calendar.getInstance(); Calendar.getInstance(Locale);

util.Locale	Locale.getDefault();
	new Locale(String language);
	new Locale(String language, String country);

text.DateFormat	DateFormat.getInstance(); DateFormat.getDateInstance(); DateFormat.getDateInstance(style); DateFormat.getDateInstance(style, Locale);
text.NumberFormat	NumberFormat.getInstance() NumberFormat.getInstance(Locale) NumberFormat.getNumberInstance() NumberFormat.getNumberInstance(Locale) NumberFormat.getCurrencyInstance() NumberFormat.getCurrencyInstance(Locale)

6.5. Parsowanie, Tokenizacja, Formatowanie łańcuchów znakowych

6.1. Wyrażenia regularne (regex)

Zasada **regex**owa jet taka że
- przeszukiwanie działa od lewej do prawej
- jeśli jakiś znak został juz użyty w porównaniu ponownie nie może zostac uzyty:
  String: abababa, regex: aba
  wynik: znaleziono w pozycji 0 i 4. a nie 2 bo ta czesc juz byla przeszukana

Metacharacters - znaki specjalne

\d - cyfry (digits)
\s - whitespace
\w - znak alfanumeryczny oraz "_"
[abc] - znaki a, b i c

[a-f] - znaki od **a** do **f**
[a-fA-F] - znaki od a do f włącznie z dużymi literami
^ - negacja
+ - **jeden** lub **wiele** (np \d+ - jedna lub wiele cyfr, [a-f]+ jedna lub wiele liter od a do f)
* - **zero** lub **wiele**
? - **zero** lub **jeden**
() - grupuje (np **0[xX]([0-9a-fA-F])+**)
. - (kropka) jakikolwiek znak
\ - "escapowanie" znaku specjalnego

*Kwantyfikatory **chciwe, niechętne** i **zaborcze***

? chciwy, ?? niechętny , dla 0 lub 1
* chciwy, *? niechetny, dla 0 lub wiecej
+ chciwy, +? niechetny, dla 1 lub więcej

Przyklad:
source: yyxxxyxx
pattern: .*xx

wynik: 0 yyxxxyxx **bo chciwy** - czyta de facto cały łańcuch

pattern: .*?xx

wynik:
0 yyxx
4 xyxx
bo niechętny - czyta po kawałku

Mieszanie regexpów i Stringów

Porównanie nie skompiluje się:
String pattern = "\d"; // **compiler error!**
- Trzeba escapować znaki specjalne:
String pattern = "****d"; // **ok**

Inne:
String p = "."; // regex sees this **as the "." metacharacter**

String p = "\."; // the compiler sees this as an **illegal**
// Java escape sequence

String p = "****."; // **OK**, and regex sees **a dot, not a metacharacter**

\n = nowa linia (na egzaminie)
\b = backspace
\t = tab

6.2. Wyszukiwanie tekstu poprzez wzorce

Pattern i Matcher

- klasa do definiowania wzorca (regexpa):
  java.util.regex.**Pattern**

- klasa do definiowania źródła
  java.util.regex.**Matcher**

```
Pattern p  = Pattern.compile("ab");      //  expression
Matcher m = p.matcher("abaaaba");     //  source
```

- Pattern tworzymy za pomoca statycznej metody **compile**(String regex) która zwraca instancję Pattern
  - **nie istnieje konstruktor** w API
- Matcher jest tworzony także poprzez metodę klasy Pattern czyli **matcher**(CharSequence cs)
  - argumentem jest interfejs CharSequence implementowany przez **String**, StringBuilder i CharBuilder
  - wywoływana na stworzonym obiekcie Pattern p

m.**find**() - uruchamia silnik wyrażeń regularnych i zwraca **true gdy znajdzie** oraz **pamieta index wystąpnienia**
m.**start**() - wypisuje index w łańcuchu znaków wystapnienia (index pierwszego znaku)
m.**group**() - wypisuje wycinek łańcucha źródłowego pasujący do wyrażenia reg.

```
while(m.find()) {
  System.out.println(m.start() + " " + m.group());
}
```
Pattern: "\d\w" **Matcher**: "ab4 56_7ab"
Output:
4 56
7 7a

- Klasy Matcher uzywa sie głównie do operacji wyszukaj/zamień do czego pomocne sa metody:
appendReplacement(), appendTail(), replaceAll() bedące w API klasy Matcher. -- nie ma na egzaminie
- Matcher działa na tzw *regionach* -- nie ma na egzaminie

Scanner

- java.util.**Scanner** służy **do *tokenizacji*** Stringów ale może tez słyżyc **do powiedzenia ile jest wystapnień** danego wyrażenia w łańcuchu

```
Scanner s = new Scanner(System.in);
String token;
do {
  token = s.findInLine(args[0]);
  System.out.println("found " + token);
```

} while (token != null);

java ScanIn "\d\d"
input: 1b2c335f456

Out:
found 33
found 45
found null

6.3. Tokenizacja

- dzielenie łańcucha na cząstki (tokens) na podstawie znaków przedzielających (delimiters)
  - delimiter nie wchodzi w skład tokenów

*Metoda **split**() klasy String*

String[] tokens = sourceString . **split** (regexDelimiterString);

regexDelimiterString - wyrażenie regularne będące delimiterem

sourceString = "ab5 ccc 45 @";
regexDelimiterString = "\\d"; (cyfry)

tokeny: >ab< , > ccc < , >< , > @<

Klasa Scanner w tokenizacji

- Scanner może zostac użyty do tokenizacji plików, Stringów i strumieni
- może przerwać proces tokenizacji w trakcie jej trwania
  - metoda split musi przeparsowac cały String - tutaj mozna pzerwac np po n-tym wystapnieniu
- tokeny moga zostac automatycznie zkonwertowane do ich prymitywnych wartości
- **defaultowym delimiterem jest spacja**

Scanner s1 = new **Scanner**(sourceString);
lub
Scanner s1 = new **Scanner**(sourceString).**useDelimiter**(regexDelimiterString);

- useDelimiter() pobiera regex i zwraca obiekt Scanner

Nastepnie możemy przechodzić pomiędzy tokenami:

while(s1.**hasNext**()) { // hasNext zwraca true jeśli są dalsze tokeny

s = s1.**next**() // metoda next zwraca bierzący token i ustawiawskaźnik Scannera na nastepnym tokenie

}

Są także inne metody:

- metody **hasNext<u>Xxx</u>()** gdzie xxx to typ sprawdzające czy jest kolejny token o danym typie
  boolean **hasNextInt()**
  boolean **hasNextBoolean()** itd
  - te metody nie przesuwaja "wskaznika" dalej - tylko sprawdzają czy jest
- metdy **next<u>Xxx</u>()** zwraca nastepny token typu xxx
  int **nextInt()**
  - te metody zwracają typ i przesuwaja wskaźnik dalej

6.4. Formatowanie printf() i format()

- Metody format i printf zostały dodane do java.io.**PrintStream** w Javie 5
- metoda format korzysta z klasy java.util.**Formatter**

<u>printf()</u>

printf("format string", *argumenty*); / argumentów może byc kilka:

System.out.printf("%2$d + %1$d", 123, 456); ---> zwróci 456 + 123

`%[arg_index$][flags][width][.precision]`**`conversion_char`**

[] - opcjonalne

*arg_index$* - który argument podstawić: 1$ pierwszy, 2$ drugi itd

Flagi:
"-" Left justify this argument
"+" Include a sign (+ or -) with this argument
"0" Pad this argument with zeroes
"," Use locale-specific grouping separators (i.e., the comma in 123,456)
"(" Enclose negative numbers in parentheses

width
This value indicates the minimum number of characters to print. (If you want nice even columns, you'll use this value extensively.)

precision
For the exam you'll only need this when formatting a floating-point number, and in the case of floating point numbers, precision indicates the number o digits to print after the decimal point.

conversion
The type of argument you'll be formatting. You'll need to know:
b boolean
c char
d integer
f floating point
s string

Przykład:
```
int i1 = -123;
int i2 = 12345;
System.out.printf( ">%1$(7d< \n", i1);
System.out.printf( ">%0,7d< \n", i2);
System.out.format( ">%+-7d< \n", i2);
System.out.printf( ">%2$b + %1$5d< \n", i1, false);
```

Output:
```
> (123)<
>012,345<
>+12345 <
>false + -123<
```

TEST
3/15 TRAGEDIA

---

7. Kolekcje i generyki

7.1. Przedefiniowanie toString(), hashCode() i equals()

Metody obiektu Object:

Method	Description
`boolean equals (Object obj)`	Decides whether two objects are meaningfully equivalent.
`void finalize()`	Called by garbage collector when the garbage collector sees tha object cannot be referenced.
`int hashCode()`	Returns a hashcode `int` value for an object, so that the object be used in Collection classes that use hashing, including Hasht HashMap, and HashSet.
`final void notify()`	Wakes up a thread that is waiting for this object's lock.
`final void notifyAll()`	Wakes up *all* threads that are waiting for this object's lock.
`final void wait()`	Causes the current thread to wait until another thread calls `notify()` or `notifyAll()` on this subject.
`String toString()`	Returns a "text representation" of the object.

toString()

- Przedefiniowanie (override) tej metody w klasie A spowoduje że każde użycie referencji do tego obiektu A np w ***sysout*** zaowocuje wywołaniem toString().
- zamiast standardowego tekstu w stylu A@78hb712 wypisze sie to co chcemy

equals()

Przypomnienie: == porówna dwie referencje czy wskazują na ten sam obiekt
equals() - porównuje czy obiekty są takie same

- przedefiniowanie equals() daje nam szanse na **inne porównywanie dwóch obiektów** niż tylko porównywanie ich pól (atrybutów)

- jeśli nie przedefiniujemy metody equals nie będzie można użyć naszego obiektu jako klucza hashtable
- ponieważ trzeba jakoś ustalić że dwa obiekty nie są takie same (klucz jest unique)

Overriding:

- **deklaracja:**
public **boolean** *equals*(**Object** o) { }

- warto skorzystać z **instanceof** podczas implementacji - przydaje się
- - **dobra metoda equals** używa instrukcji **instanceof**
- dobrym zwyczajem jest sprawdzanie jak najmniejszej liczby atrybutów - względy wydajnościowe
- uwaga omawiane w tym rozdziale metody są **public**

---- UWAGA na egzaminie czasem pomijany jest ten modyfikator
---- UWAGA2 - czasem na egzaminie jest rzykład overloadingu - inny argument

Zasady implementacji:

- zwrotność: x.equals(x) = zawsze true

Ciekawostka:
Chciałoby się powiedzieć, że 'x.equals(y)' musi dawać dokładnie ten sam wynik co 'y.equals(x)', ale tak nie jest.
Jeśli bowiem 'x' wskazuje na pewien obiekt a 'y' ma wartość '**null**', to 'x.equals(y)' ma wartość 'false'
(musi być 'false', co jest kolejnym punktem kontraktu) a 'y.equals(x)' wywołuje wyjątek **NullPointerException**.

- symetryczność: jeśli x.equals(y) to y.equals(x)
- przechodniość: gdy x.equals(y) i y.equals(z) to również x.equals(z)
- x.equals(null) zawsze false dla x not-null

- Przy założeniu, że porównywane obiekty się w międzyczasie nie zmieniają każdorazowe wywołanie funkcji 'x.equals(y)' musi dawać taki sam wynik

hashCode()

- **deklaracja:**
public **int** *hashCode***()** { }

- W wiekszosci przypadku gdy nadpisujemy metode equals() to również nadpisujemy hashCode()

- metoda ta to swoiste ID obiektu (niekoniecznie unikalne)
- pomaga w zapisie obiektu do kolekcji oraz potem wyszukaniu tego obiektu w kolekcjach takich jak HashMap czy HashSet
  - na podstawie hashcode obiekt ląduje w odpowiednim miejscu Mapy/Seta
  - mając hashcode wiemy gdziejest obiekt
  - teoretycznie w jednym miejscu kolekcji może byc więcej niz jeden obiekt - ale wtedy staje się to malo efektywne
- **jesli dwa obiekty są równe (**equals = true**) to ich hashcode tez musi byc taki sam**
  - nie oznacza to jednak że obiekty różne nie mogą mieć tego samego hashcode - różne też **mogą** równy hashcode
- najczęściej hashcode to wyliczenie XOR pól obiektu i inne operacje matematyczne na polach.

- **NA EGZAMIN** trzeba rozpoznac która metoda hashCode pozwala na odszukanie obiektu

- z punktu widzenia efektywnosci obiekt majacy ten sam *hashcode* powinien zwrocic true w *equals*

Zasady implementacji:

- przy założeniu że produkt sie nie zmienia to kilkukrotne wykonanie hashCode() na tym samym obiekcie w tym samej aplikacji powinno za każdym razem dac tę samą wartość
- dwa obiekty równe (wg *equals()*) muszą zwrócić ten sam hashCode
- **nie jest zalecane** aby dwa różne obiekty (wg *equals()*) zwracały ten sam hashCode
patrz tabela na str 534 (na dole)
- **nie jest zalecane** aby w metodzie hashCode() uzywać pól oznaczonych ***transient*** i na ich podst. liczyc hashcode
    - bo gdy zapiszemy obiekt w kolekcji a nastepnie dokonamy serializacji i deserializacji to **może sie zdażyć** że obiektu juz nie odnajdziemy
        - bo moze zmienic sie to pole i zmieni sie wowczas hashcode

UWAGA NA EGZAMINIE: mieć pewność co do sygnatur metod że jest to override a nie overload.

7.2. Kolekcje

Operacje podstawowe na kolekcjach:
- dodawanie elementu
- usuwanie elementu
- szukanie czy element jest w kolekcji
- pobieranie elementu
- iterowanie po kolekcji

Użycie słówka "collection"

- **collection** (lowercase c), which represents any of the data structures in which objects are stored and iterated over.
- **Collection** (capital C), which is actually the java.util.Collection interface from which Set, List, and Queue extend. (That's right, extend, not implement. There are no direct implementations of Collection.)
- **Collections** (capital C and ends with s) is the java.util.Collections class that holds a pile of static utility methods for use with collections.

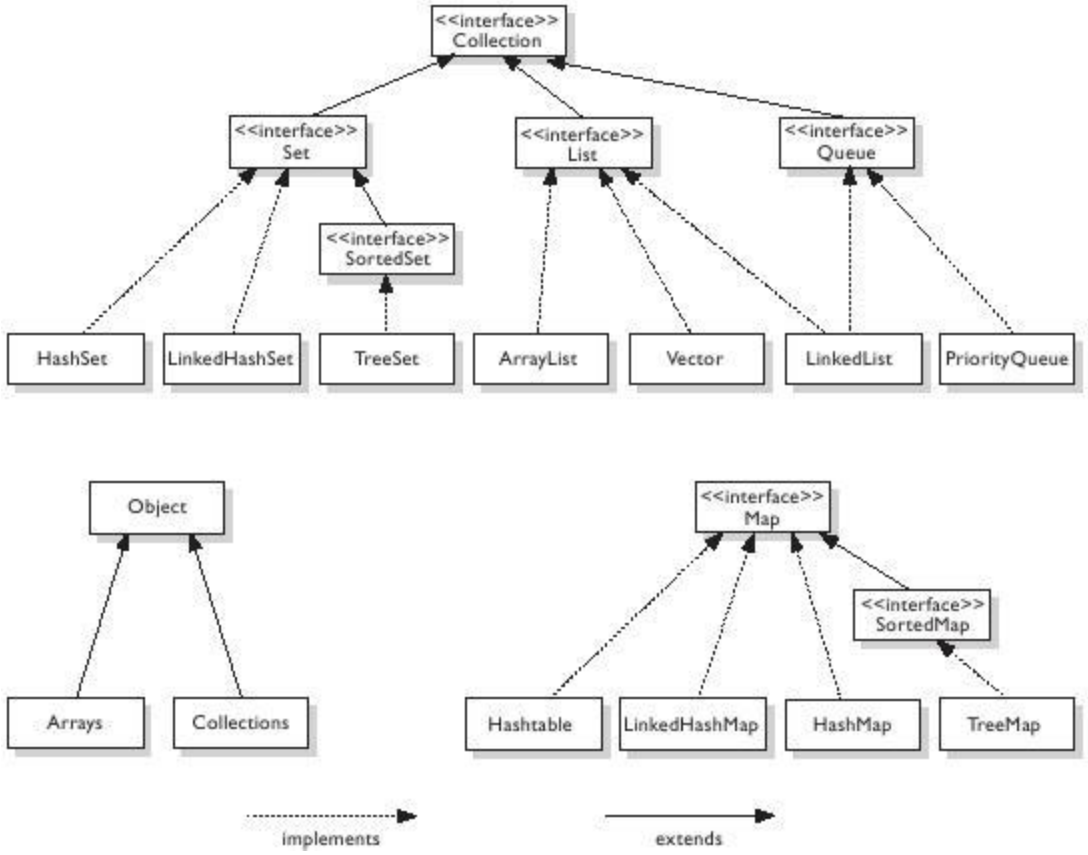
Podstawowe interfejsy kolekcji:
- Collection
- List
- Queue
- Set
- Map
- SortedSet
- SortedMap

Najważniejsze implementacje

Maps	Sets	Lists	Queues	Utilities

elem. z unikalnym ID (hashcode)	**unikalne elementy**	**listy elementów** (mogą się powtarzać)	**elementy w kolejności obrabiania**	statyczne metody utils
HashMap	HashSet	ArrayList	PriorityQueue	Collections
Hashtable	ArrayList	Vector		Arrays
TreeMap	LinkedHashSet	LinkedList		
LinkedHashMap	TreeSet			

Hierarchia kolekcji:



Dzielimy powyższe typy kolekcji na rodzaje:

- Ordered Unordered (uporządkowane)
- można iterować po kolekcji w określonej (specyficznej) kolejności
  ArrayList - porządek indexowy, LinkedHashSet - porzadek insertów
- posiada jakiś porządek (np indeksowy czy ostatniego włożenia)

- Sorted Unsorted (posortowane)
  - porzadek kolekcji jest zapewniony dzieki pewnym zasadom sortującym
  - reguła sortująca **zależy od właściwości elementów**
  - reguła sortująca nie może zależeć od czasu insertu czy ostanigo dostępu
  - z reguły sortowanie jest naturalne (alfabetyczne dla Stringa, czy liczbowe)
  - można dodawać własne reguły sortowania elementów
    - obiekty sortowane muszą **implemetować interfejs *Comparable*** lub ***Comparator***

- Implementacja **może być:** *unsorted i unordered*, *ordered i unsorted* lub *ordered i sorted*
  - **nie może** być sorted i unordered

7.2.1. Rodzaje (klasy) kolekcji

*Interfejs **List***

- oparte na indexach
  - zawiera metody typu get(index), indexOf(object) czy add(index, object)
- jest uporządkowana na podstawie indexu
- dodawanie możliwe na konkretne miejsce (index) lub na końcu (gdy nie podano indexu)

ArrayList
- jak mozliwa do rozszerzania tablica
- **szybka iteracja** i **szybki dostęp losowy** do danych
  - implementuje marker-interface RandomAccess mówiący że dostęp losowy ma stały czas wykonania
- **dobre do:** szybkich iteracji, **słabe gdy**: jest duzo operacji usuwania i wkładania

Vector
- to samo co ArrayList z wyjątkiem że **Vector jest Thread Safe**
  - raczej zalecane używanie ArrayList (Vector jest starą klasą)

LinkedList
- jest uporządkowana indexowo jak ArrayList
- elementy posiadają **podwójne linkowania jeden do drugiego**
  - są dzieki temu nowe metody: usuwanie i wkładanie na końcu i na początku
- implementuje (od Java5) interfejs Queue:
  peek(), poll(), offer()
- **dobre do**: szybkiego wkładania i usuwania elementów, **słabe do:** iterowania (wolnijsze)

*Interfejs **Set***

- zapewnia unikalność elementów w kolekcji (dwa takie same elementy nie mogą współistnieć w Set)
  - **oparte na metodzie equals()**

HashSet

- jest **nieposortowany** i **nieuszeregowanym** (unsorted i unordered)
- polega na metodzie **hashCode()** podczas wkładania i szukania
  - im wydajniejsza metoda tym wydajniejszy Set
- **dobre gdy**: potrzebujemy unikatowości **bez potrzeby porządku podczas iteracji**
  - kolejność iterowania jest nieprzewidywalna

LinkedHashSet

- uszeregowana wersja HashSet
- posiada **podwójne linkowania** do elementów
- **dobre gdy:** chemy iterować Set

TreeSet

- Set posortowany na podstawie algorytmu **drzewa Red-Black**
- gwarantuje **sortowanie naturalne, rosnące**
  - **można to zmienić** za pomocą konstruktora i uzywając *Comparable* lub *Comparator*

*Interfejs **Map***

- dba o **unikatowość kluczy**
- przypisuje klucz do wartości (z czego **oba sa obiektami**)
- korzsta z *equals()* do porównania kluczy

HashMap

- **nieposotowana i nieuporządkowana** mapa
- klucze bazują na **hashcode**
  - im lepsza metoda hashCode() tym wydajniejsza mapa
  - zezwala na **jeden klucz null** i wiele wartości null
- **dobre gdy** nie dbamy o porządek i sortowanie w mapie

Hashtable (małe "*t*")

- staroć i zaszłość (jak Vector)
- jest **synchronizowaną (thread safe)** odpowiedniczką HashMap
  - co oznacza ze metody mapy sa synchronized
  - **nie zezwala na klucze ani wartości null**

LinkedHashMap

- podobny do LinkedHashSet
- utrzymuje kolejność wkładania (lub opcjonalnie dostępu)
- **jest wolniejsza od HashMap** przy dodawaniu i usuwaniu elementów
- **jest szybsza od HashMap** przy iteracji

TreeMap

- sortowana mapa
  - sortowanie wg naturalnego porządku sortowania
  - sortowanie mozna zmienic poprzez Comparator lub Comparable (podczas konstruowania)

*Interfejs **Queue***

- kolejki
- zaprojektowana do trzymania elementów **do przetwarzania w określonej kolenności**
  - np FIFO
- posiada wsyztskie metody Kolekcji oraz dodatkowe

PriorityQueue

- nowa w Java5
- ponieważ LinkedList implementuje także Queue podstawowe kolejki można realizowac na LinkedList
- "priority-in Priority-out" w przeciwienstwie do FIFO
- posortowana w naturalnym porzadku lub poprzez Comparator

UWAGA na egzaminie pytania "wybierz interfejs" oczekuja wybrania Map, List itp a nie konkretnych klas. Tak samo odwrotnie - wybierz kalse czyli nie Map i nie List tylko np ArrayList.

Exam WATCH:
For the exam, you might be expected to choose a collection based on a particular requirement,
where that need is expressed as a scenario. For example, which collection would you use if you needed to maintain and search on a list of parts, identified by their unique alphanumeric serial number where the part would be of type Part? Would you change your answer at all if we modified the requirement such that you also need to be able to print out the parts in order, by their serial number? For the first question, you can see that since you have a Part class, but need to search for the objects based on a serial number, you need a Map. The key will be the serial number as a String, and the value will be the Part instance. The default choice should be HashMap, the quickest Map for access. But now when we amend the requirement to include getting the parts in order of their serial number, then we need a TreeMap—which maintains the natural order of the keys. Since the key is a String, the natural order for a String will be a standard alphabetical sort. If the requirement had been to keep track of which part was last accessed, then we'd probably need a LinkedHashMap. But since a LinkedHashMap loses the natural order (replacing it with last- accessed order), if we need to list the parts by serial number, we'll have to explicitly sort the collection, using a utility method.

Tabela porównawcza

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

7.2.2. Podstawy klasy ArrayList

java.util.ArrayList

- jest najczesciej uzywana
- jest jak dymanicznie rosnąca tablica
- ma lepsze od tablic algorytmy wyszukiwania i dokładania
- najlepiej inicjalizować ją polimorficznie:

```
    List arr = new ArrayList();
lub
    List<String> arr = new ArrayList<String>();
```

- **dodawanie**: metoda *void* **add**(Object o), *void* **add**(int INDEX, Object o)
- **rozmiar:** *int* **size**();
    - jest dynamiczny - nie tak jak w tablicy stały
- **szukanie**: *boolean* **contains**(Object o);
- **usuwanie**: *boolean* **remove**(Object o); *E* **remove**(int *INDEX) /zwraca usuniety elem/*;

7.2.3. Autoboxing w kolekcjach

- kolekcje **moga trzymać tylko OBIEKTY a nie prymitywy**
    - aby trzymac prymitywy trzeba je obudować wrapperem

- ale **można używac autoboxingu**:

add(42) działa ponieważ jest autoboxing i de facto jest to *add (new Integer(42))*

7.2.4. Sortowanie kolekcji i tablic

- **kolekcje nie posiadają metod sort**
- do sortowania kolekcji służy metoda **sort()** klasy ***java.util.*Collections**

- użycie:

Collections.**sort** (*mojaKolekcja);* // posortuje w naturalnym porządku kolekcję *mojaKolekcja*

- działa dla liczb i stringów bo String i klasy-wrappery implementują interfejs **Comparable**
- dla kolekcji obiektów własnych - **obiekty te muszą implementować** interfejs **Comparable**

*Interfejs **Comparable***

- jest używany przez metody **Collections.sort()** oraz java.utils.**Arrays.sort()**
- sygnatura:

```
interface Comparable<T> {
  public int compareTo ( T object );     // T - obiekt do ktorego bedzie porównywany "this"
}
```

bez generyków:
public *int* **compareTo**(***Object*** o)

- zwraca:

int x = thisObject.compareTo(anotherObject);

- liczba ujemna - gdy thisObject < anotherObject
- zero - gdy thisObject == anotherObject
- liczba dodatnia - gdy thisObject > anotherObject

- przykład:

```
class MyObject implements Comparable<MyObject> {
                        // MyObject bedzie porownywany do innego MyObject

      // existing code

public int compareTo(DVDInfo d) {          // Object d - bez generyków
      return title.compareTo(d.getTitle()); // przykładowa implementacja
} }
```

- powinno się jako argument **podawać obiekt do którego bedziemy porównywac** (stąd generyki)
  - dla porównania w ***equals()*** ZAWSZE będzie to Object!

- **wadą** *Comparable* jest to że możemu w klasie określić **tylko jeden rodzaj porównywania** (jedna metoda compareTo)

- jest implementowana w API w klasach takich jak: String, Wrapper classes, Date, Calendar itp

Interfejs ***Comparator***

- służy do wykorzystania w przeładowanej metodzie **Collections.sort**(List<T> list, **Comparator**<? super T> c) w której jako 2 argument można podać Comparator
- pozwala na **wiele sposobów sortowania** danego obiektu
- pozwala na **sortowanie klas do których nie ma możliwości modyfikacji**

- sygnatura:

```
interface Comparator <T> {

      public int compare ( T object_1,  T object_2 );     // T - obiekty porównywane

  public boolean equals ( Object o );

}
```

- metoda *compare*() **zwraca**:

  - liczba ujemna - gdy *object*_1< *object*_2
  - zero - gdy *object*_1== *object*_2
  - liczba dodatnia - gdy *object*_1> *object*_2

- **przykład:**

  - **wymaga stworzenia obiektu** który implementuje Comparator, **będącego obiektem porównującym**

```
import java.util.*;
class GenreSort implements Comparator<MyObject> {
    public int compare (MyObject one, MyObject two) {
        return one.getGenre().compareTo(two.getGenre());
    }
}
```

- **uzycie w sortowaniu**

```
GenreSort gs = new GenreSort();
    Collections.sort ( mojaLista, gs );
```

Sortowanie tablic z klasą Arrays

- sortowanie **tablic**

  - Arrays.**sort** (arrayToSort)
    - Arrays.**sort** (arrayToSort, *Comparator*)

- są to metody statyczne (tak jak w *Collections*)
- metoda *sort()* została przeciążona wielokrotnie aby była w stanie posortować wszytskie tablice z prymitywami
- metoda sort() **zawsze sortuje tablice prymitywów w porządku naturalnym**
  - **nie można sortować** tablic prymitywów **z Comparatorem**

- **nie można posortować tablic (ani kolekcji) z różnymi typami** (niepowiązanymi np wspólną nadklasą czy interfejsem)

7.2.5. Przeszukiwanie kolekcji i tablic

- klasy Arrays i Collections posiadają metodę wyszukującą ***binarySearch()***

- **sygnatura:**

  int **binarySearch (*List* *list*, *T* *key*)** *// dla* posortowanych w porządku naturalnym **rosn**.
  int **binarySearch (*List* *list*, *T* *key*, *Comparator* *c*)** *// dla* posortowanych Comparatorem
- **zwraca**:

  - wartość dodatnią lub 0 gdy znalazł - oznacza indeks w kolekcji gdzie został znaleziony element
  - wartość ujemną **-(insertion point) -1** gdzie insertion point oznacza gdzie **mógłby zostac element włożony**

- kolekcja lub tablica **musi byc posortowana** gdy chcemy w niej szukać
  - dla nieposortowanej tablicy **wynik nie będzie prawidłowy** (jest nieprzewidywalny)
- dla kolekcji posortowanych **w porządku naturalnym nie podajemy** Comparatora w argumencie
- dla kolekcji posortowanych **przez Comparator trzeba podac** ten sam Comparator
  - dla **prymitywów nie można podać Comparatora**

7.2.6. Konwersja tablic do list i odwrotnie

- interfejsy List i Set posiadają metodę **toArray()**

  T[] **toArray**(T[] a)

```
      Parameter:  a - the array into which the elements of this list are to
 be stored, if it is big enough; otherwise, a new array of the
 same runtime type is allocated for this purpose.

      - przykład: Object[] oa = iL.toArray();
```
- klasa Arrays posiada metodę statyczną **asList()**

public static List<T> **asList**(T**...** a)
Parametr - (**var-args)** lub the array by which the list will be backed

7.2.6. Użycie kolekcji

7.2.6.1. Użycie List

Iterator

- przed Javą5 używany był głównie Iterator do iterowania kolekcji
- **może byc generyczny Iterator <T>**
- obiekt Iterator posiada następujące metody:

boolean **hasNext()** - zwraca true gdy jest jeszcze jakis element w kolekcji, **nie przestawia** wskaźnika
Object **next()** **- przestawia wskaźnik**, oraz **zwraca element** kolekcji

```
Iterator<Dog> i3 = d.iterator(); // make an iterator with Generics (without is allowed but it require casting)
while (i3.hasNext()) {
        Dog d2 = i3.next(); // cast not required because there is Generics in Iterator
          // ...
}
```

7.2.6.2. Użycie Set

- Sety nie dopuszczają duplikatów
  - włożenie elementu duplikującego spowoduje że metoda **add()** zwróci **false**
- *HashSet* i *LinkedHashSet* **nie są sortowane** więc iteracja for **nie gwarantuje przewidywalnej kolejności**
- *TreeSet* (oraz inne sortowane kolekcje) **wymagają** aby **elementy były Comparable (**lub miały **Comparator)**
  - w przeciwnym przypadku dodanie takiego elementu do kolekcji spowoduje błąd wykonania

7.2.6.3. Użycie Map

- obiekty-klucze wykorzystywane w mapach **muszą przedefiniowywać** metody **hashCode()** i **equals()**
  - w przeciwnym razie **kod się skopiluje** ale nie będzie możliwe odnalezienie elementu
    - szukanie po kluczu nieposiadającym tych metod spowoduje **zwrot *null***
- jako klucze może być dowolny **obiekt** lub **enumeracja**

- **prymitywy nie** aczkolwiek z użyciem autoboxingu tak
- enumeracje **posiadają hashCode() i equals() domyślnie**

- nie powinno się używać w **hashCode()** pól które mogą się zmienić
- gdy włożymy do Map element z jakims hashCodem wygenerowanym na podstawie pola i potem zmienimy to pole hashcode se zmieni i elementu nie znajdziemy juz w mapie

- kontrakt wyszukiwania w mapie:
1. Use the hashcode() method to find the correct bucket
2. Use the equals() method to find the object in the bucket

7.2.6.4. Użycie PriorityQueue

- używa **priorytetów zdefiniowanych przez użytkownika - Comparator**
- może to oczywiście być natural order (domyślnie np dla Integerów)
- można podać Comparator w konstruktorze kolejki.
- **priorytety to** de facto **kolejność** (naturalna, określona w Comparable lub określona w Comparatorze)

- kolejka może być posortowana poprzez **Comparator** definiujący priorytety (porządek)
- dodatkowe metody
  peek(), - **pobiera** element o **najwyższym priorytecie** ale nie usuwa go
  poll(), - **pobiera** element o **najwyższym priorytecie** i **usuwa** go z kolejki
  offer() - **dodawanie** elementu

Key Methods in java.util.Arrays	Descriptions
`static List asList(T[])`	Convert an array to a List, (and bind them).
`static int binarySearch(Object[], key)` `static int binarySearch(primitive[], key)`	Search a sorted array for a given value, return an index or insertion point.
`static int binarySearch(T[], key, Comparator)`	Search a Comparator-sorted array for a value.
`static boolean equals(Object[], Object[])` `static boolean equals(primitive[], primitive[])`	Compare two arrays to determine if their contents are equal.
`public static void sort(Object[ ] )` `public static void sort(primitive[ ] )`	Sort the elements of an array by natural order.
`public static void sort(T[], Comparator)`	Sort the elements of an array using a Comparator.
`public static String toString(Object[])` `public static String toString(primitive[])`	Create a String containing the contents of an array.
Key Methods in java.util.Collections	**Descriptions**
`static int binarySearch(List, key)` `static int binarySearch(List, key, Comparator)`	Search a "sorted" List for a given value, return an index or insertion point.
`static void reverse(List)`	Reverse the order of elements in a List.
`static Comparator reverseOrder()` `static Comparator reverseOrder(Comparator)`	Return a Comparator that sorts the reverse of the collection's current sort sequence.
`static void sort(List)` `static void sort(List, Comparator)`	Sort a List either by natural order or by a Comparator.

Key Interface Methods	List	Set	Map	Descriptions
`boolean add(element)` `boolean add(index, element)`	X X	X		Add an element. For Lists, optionally add the element at an index point.

`boolean contains(object)`
`boolean containsKey(object key)`
`boolean containsValue(object value)`
	X	X	
			X
			X
				Search a collection for an object (or, optionally for Maps a key), return the result as a `boolean`.

`object get(index)` `object get(key)`	X		 X	Get an object from a collection, via an index or a key.
`int indexOf(object)`	X			Get the location of an object in a List.
`Iterator iterator()`	X	X		Get an Iterator for a List or a Set.
`Set keySet()`			X	Return a Set containing a Map's keys.
`put(key, value)`			X	Add a key/value pair to a Map.

`remove(index)`
`remove(object)`
`remove(key)`
	X
	X
		
		X
			
			X
				Remove an element via an index, or via the element's value, or via a key.

`int size()`	X	X	X	Return the number of elements in a collection.
`Object[] toArray()` `T[] toArray(T[])`	X	X		Return an array containing the elements of the collection.

7.2.7. Naturalny porządek Stringów:

Remember that spaces sort before characters
and that uppercase letters sort before lowercase characters

Natural order: **> f< >FF< >f < >ff<**

7.3. Generics Types

- w < > przechowywane są typy
- tworzenie "typowo bezpiecznych" kolekcji
  - (podobnie jak tablice są bezpieczne)
- generics weszły od Java5
  - **można mieszać** stary kod bez generyków **ale kompilator wygeneruje Warningi**
    - *może* to także spowodować późniejsze błędy w runtime
- bez generyków programista musiał być ostrozny podczas wkładania elementów do kolekcji

Bez generyków:

List myList = new ArrayList(); // może przechowywać każdy obiekt (**byle nie prymityw**)
myList.add("Fred"); // OK
myList.add(new Dog()); // OK

String s = (String)myList.get(0); // wydobywanie elementu wymaga castowania
// - **niebezpieczne** bo może tam sie znaleźć inny obiekt

Z generykami

List**<String>** myList = new ArrayList**<String>**();
myList.add(**"Fred"**); // OK, przechowuje Stringi
myList.add(new Dog()); // **NO - compiler error!!**

String s = myList.get(0); // **OK - nie wymaga castowania**

- można także wykorzystać *nowego fora* **bezpiecznie**
```
for (String s : myList)
   print(s);    // s zawsze będzie typu String
```

- jako parametr metody:
```
void takeListOfStrings(List<String> strings){
   strings.add(new Integer(42)); // NO - compiler error!! strings is type safe
   strings.add("Fred");  // OK
```

- jako zwrot metody:
```
public Set<String> getNamesList() { /* code */ }
```

Update starego kodu do kodu generycznego

- dodanie typów w < > przy referencji oraz konstruktorze (List**<T>** ref = new ArrayList**<T>**();)
- dodanie typów w < > przy kolekcjach w argumentach metod i zwrotach
- **dodatkowo można** usunąć castowania wspomniane wczesniej (nie trzeba ale można)

7.3.1. Mieszanie kolekcji z generykami i bez

- **można** podawać generyczna kolekcję do starej metody która nie posiada generyków.

```
deklaracja:  void myMethod(List list);                    // kod bez generyków
użycie:      myMethod (new ArrayList<Integer>());  // OK, podanie generyka działa
```

ale:
metoda myMethod() może być napisana "niebezpiecznie" i używać np castowania na inny typ. wówczas może byc wyjątek ClassCastException
metoda może także dodawać inne typy do kolekcji **(nawet gdy podamy w argumencie kolekcje generyczną):**
- tablice posiadają ochronę typów w kompilacji i runtime.. **kolekcje tylko w**

kompilacji

- jest tak z powodu kompatybilności wstecz Javy

--

Przykład:

```
void insert(List list) {              // STARA metoda bez generyków
  list.add(new String("42"));         // put a String in the list passed in
}

List<Integer> myList = new ArrayList<Integer>();
insert(myList);    // spowoduje to wykonanie metody, bez błedu kompilacji
```

wygeneruje tylko Warningi (nawet gdybyśmy wkładali Integery)

Dlaczego? Metoda zadziała bo podczas wykonania wsyztskei kolekcje (generyczne i nie) są przedstawione tak samo.

Generyczność jest tylko kwestią kompilatora (ochrona kompilacji) i jest rozpatrywana tylko podczas kompilacji.

Dlatego powyższy kod wykona się bez błedów runtime.

Warningi:

```
javac MojaKlasa.java
  Note: MojaKlasa.java uses unchecked or unsafe operations.
  Note: Recompile with -Xlint:unchecked for details.
```

Po skompilowaniu z opcją **`-Xlint:unchecked`** mamy więcej informacji:

```
javac -Xlint:unchecked TestBadLegacy.java
TestBadLegacy.java:17: warning: [unchecked] unchecked call to
add(E) as a member of the raw type java.util.List
       list.add(new String("42"));
                ^
1 warning
```

- błedy pojawić się mogą dopiero gdy potem będziemy wyjmować elementy z takiej "zepsutej" kolekcji i natrafimy na element który nie powinien sie tam znaleźć.

--

- **uwaga** W KOLEKCJACH NIEGENERYCZNYCH metoda get() zwraca **Object** a więc jeśli były tam Integer to **trzeba zcastować**
  - tutaj **nie zadziała autoboxing** bo z Object nie może...
- z Kolekcji Generycznych metoda get() zwróci obiekt typu określonego w generyku <T> a więc nie trzeba castować - autoboxing

1. List test = new ArrayList();
   int x = (Integer)test.get(0); // **you must cast !!**

2. List<**Integer**> test2 = new ArrayList<Integer>();
   int x2 = test2.get(0); // **cast not necessary**

- trzeba uważac na egzaminie na to czy jest castowanie czy nie ma.

7.3.2. Generyczność i polimorfizm

- polimorfizm działa tylko w odniesieniu do *"podstawy"* a **nie do typów w <generic>**

- typ w generyku w referencji **musi się zgadzać** z typem w obiekcie tj:

List<**Typ**> myList = new ArrayList<**Typ**>();

- **nie zadziała** tutaj poniższy zapis

List<**Parent**> myList = new ArrayList<**Child**>(); --> **błąd**

- **można** później wkładać **elementy podklas**:

List<Number> l = new ArrayList<Number>();

l.add(new Integer(3));

- ale **przy l.get() wymagać to może Castowania -** widoczny jest tylko typ z *generyka*

- inaczej sprawa się ma z tablicami bo można zrobic tak: Object[] myArray = new JButton[3];

7.3.3. Metody generyczne

- inaczej jak w przypadku polimorfizmu, metody z agrumentem kolekcji generycznej **mogą przyjąć tylko taki typ kolekcji jaki zadeklarowano:**

void *metoda*(**List<Animal>** *c*) --> metoda ta **nie przyjmie** w wywołaniu parametru **List<Dog> czyli podtypu Animal**

- metoda przyjmie natomiast **ArrayList**<**Animal**> bo kolekcja sama w sobie podlega juz prawu polimorfizmu - typ generyczny nie podlega.

Powód?

Mając metodę *przykładową*:

```
public void addAnimal(ArrayList<Animal> animals) {
    animals.add(new Dog()); //można dodać element podklasy Dog do kolekcji generykiem nadklasy Animal
}
```

... więc gdyby mozliwe było przekazanie ArrayList<***podtyp***> to można by bylo przekazać np ArrayList<**Cat**> a więc powyższa metoda **nie wykonała by się w tej sytuacji**

Hipotetycznie:
ArrayList<**Animal**> cats = new ArrayList<**Cat**>();
addAnimal(**cats**); // ***byłby*** **runtime error** w momencie wykonania

Ale w przypadku **tablic** jest to dozwolone i faktycznie (gdyby zmienić powyższy przykład na tablice) **będzie runtime exception 'ArrayStoreException'**
- przy generykach chciano tego uniknąć
- jak wspomniano (7.3.1. -> Przyklad) generyki są tylko w czasie kompilacji więc nie można było zastosować runtime exceptiona w tym przypadku.

Argumenty generyczne z podtypami

- można zmodyfikować parametr metody aby mogła przyjmować generyki z podtypami używając **<? extends Klasa>**
  - jest no niejako **obietnica** że **nie bedziemy wkładać "nieodpowiednich" elementów a tylko np wywoływać na nich metody** a więc polimorfizm
  - w tym przypadku można dodawac elementy

Deklaracja:

```
public void metoda(List<? extends Animal> animals)
```

- oznacza to że można w argumencie przekazać **Liste z podtypami** klasy Animal lub **przypadku interfejsu Animal także klasami implementującymi *(IS-A)***
  - czyli np List<**Animal**>, List<**Dog**>, List<**Cat**> itd...
  - w przypadku interfejsu jest także słówko extends: `metoda(List<? extends Serializable> arg)`

- **nie można** w tej metodzie **dodawać elementów**

```
public void addAnimal(List<? extends Animal> animals) {
  animals.add(new Dog()); // NO! Can't add if we use <? extends Animal> --> błąd kompilatora
  }
```

Argumenty generyczne z nadtypami

- można równiez definiowac parametr metody aby pasował do wsyztskich nad-typów danej klasy:

Deklaracja:

```
public void metoda(List<? super Dog> animals)
```

- powyższa definicja zaakceptuje **Listę typu Dog** oraz **wsyztskich nadtypów**
  - np List<Dog>, List<Animal>, List<Object>
- **nie będą akceptowane klasy pod-typów**

Argumenty generyczne z jakimkolwiek typem

- poniższa deklaracja **przyjmie Listę każdego podtypu**

```
public void metoda(List<?> animals)
```

- powyższa deklaracja **różni się** zasadniczo od *metoda* (List<**Object**> l) ponieważ powyższa przyjmie każdy **a ta tylko List<Object>**
- w tym przypadku **nie można dodawać** elementów do kolekcji w parametrze

- **BTW**: List<? extends Object> and List<?> **oznacza to samo**

UWAGA:

- **nie można używać "wildcart notation" (<?>) w konstruowaniu obiektów:**
  - np: *new ArrayList<? extends Animal>()* - **nie skompiluje się**

7.3.4. Deklaracje typów generycznych

- **API** mówi czy klasa posiada typ generyczny np:
  public interface List<**E**> - gdzie E określa typ który później jest używany w metodach tej klasy np"
  boolean add(**E** o) - E jest parametrem pod który jest podstawiana jakakolwiek klasa (typ)

<u>Tworzenie klasy generycznej:</u>

`public class Klasa<T> { }` - gdzie **T będzie parametrem** określającym **typ** (Litera **T może byc zamieniona** na inną */małą* lub *dużą* lub *słowo/*)

- następnie T jest używany jako parametr/zwrot w **metodach** np:

public **T** metodaKlasy() - jako zwrot
public void metodaKlasy(**T** param) - jako parametr

- **konstruktor** jak zwykla klasa:
`public Klasa(){ /*konstruktor*/ }`

- metody przyjmujące typ T będą mogły przy wywołaniu przyjąć tylko typ z generyka:

Klaska<**MojTyp**> k = new Klaska<**MojTyp**>();
k.metodaKlasy(**new MojTyp()**) // proba dodania obiektu innej klasy niz tej podanej w generyku spowoduje błąd

- można używac **więcej niż jeden parametr-typ**:
public class UseTwo<**T, X**>

- UWAGA - nie można w tym wypadku korzystac z **wildcardu <?>**
public class NumberHolder<?> { ? aNum; } // ***NIE*** **- błąd kompilatora!**
public class NumberHolder<T> { T aNum; } // **TAK**

<u>Przykład klasy</u>

```
public class TestGenerics<T> {  // as the class type
   T anInstance;                // as an instance variable type
   T [] anArrayOfTs;            // as an array type
   TestGenerics(T anInstance) {     // as an argument type in constructor or method
      this.anInstance = anInstance;
   }
   T getT() {                       // as a return type
      return anInstance;
   }
}
```

- **Uwaga** - obiekt klasy z typem generycznym **nie musi być** zadeklarowany z tym typem
  - jednak **kompilator wyświetli Warning** że **nie jest to zalecane:**

```
TestGenerics tg = new TestGenerics();     // Warning: TestGenerics is a raw type.
                                                                    // References to generic type TestGenerics<T> should be parameterized
```

- zalecane w Javie5 i 6 jest więc użycie takie: `TestGenerics<Typ> tg = new TestGenerics<Typ>();`

Tworzenie metod generycznych

- metoda generyczna może przyjąc jako argument obiekt dowolnej klasy

```
public <T> void makeArrayList(T t) { // take an object of an unknown type and use a "T" to represent the type
   List<T> list = new ArrayList<T>();   // przykładowa implementacja
        list.add(t);
}
```

- typ generyczny <T> musi byc zadeklarowany **przed type zwracanym** (*tu przed void)*
- **UWAGA:** public void makeList(T t) { } zadziała tylko gdy będzie klasa o nazwie T

- można zadeklarować żeby metoda przyjmowała np typy pochodne:

public <**T extends** *Number*> void makeArrayList(**T** t) // przyjmuje **tylko Number i pochodne Number** (np Integer)

- nie ma możliwości użycia tutaj **super** (*T super Number* --> **źle**) jak w argumentach generycznych: pkt 7.3.3.

- **konstruktor** również może byc generyczny - **także w klasie nie-generycznej:**

```
public class Radio {
   public <T> Radio(T t) { } // legal constructor
  }
```

- **bezsensowne ale legalne jest:**
  class X { public <X> X(X x) { } }

- X jako konstruktor nie ma nic wspólnego z <X> jako typem - **nie ma konfliktu nazw** pomiędzy nazwami typów a klasami czy parametrami...

str 606 - zrobic Drill i TEST.

8. Klasy wewnętrzne

- inner classes - definiowanie klasy wewnątrz innej klasy
  - tak jak klasa może mieć *member variables* może mieć także *member class*'ę
- wyróżniamy klasy: wewnetrzne (**inner**), lokalne (**method-local** inner), aninimowe (**anonymous**), statyczne zagnieżdzone (**static nested**)
- klasy wewnetrzne **mają dostep do wszystkich memberów** klasy zewnetrznej (outer)

8.1. Klasy wewnętrzne zwykłe - inner classes

- klasy takie posiadają swoją funkcjonalność ale jednocześnie muszą byc integralną częścią innej klasy
- przykładem (i pierwszym powodem wprowadzenia inner classes) są ***event handlers***
  - sa to (event handlery) jednak klasy ściśle związane z dana implementacją (klasy w której są)

- zwykła klasa wewnętrzna znajduje się wewnątzr klasy **ale poza metodami**
- klasy wewnetrzne **mają dostep do wszystkich memberów** klasy zewnetrznej (outer)

Definiowanie

```
class MyOuter {
   class MyInner { }
}
```

- kompilacja: *%javac MyOuter.java* stworzy 2 pliki klas: *MyOuter.class* i *MyOuter$MyInner.class*
  - nie jest jednak możliwy *zwykły* dostęp (uruchomienie) w stylu *%java MyOuter$MyInner* --> ***źle***
- uruchomienie jest **możliwe poprzez instancję klasy zewn.**

Instancjonowanie klasy wewn.

- **nie jest możliwe bez instancji klasy zewn**.
- instancjonowanie kalsy wewn. z wnętrza klasy zewn. nie rózni sie praktycznie od instancjonowania zwyklej klasy:

```
class MyOuter {
   public void makeInner() {
 MyInner in = new MyInner();  // make an inner instance
 in.seeOuter();
   }
 class MyInner {
   public void seeOuter() {/* impl */}
   }
}
```

- **nie można instancjonować klasy wewnetrznej wewnątrz metod statycznych bez referencji na obiekt klasy zewn.**
  - musi byc instancja!

- instancjonowanieklasy wewn. z poza klasy zewn. możliwe tylko gdy jest instancja zewn. klasy

```
MyOuter outer = new MyOuter();     // gotta get an instance!
MyOuter.MyInner inner = outer.new MyInner();
inner.seeOuter();
```

lub: `MyOuter.MyInner inner = new MyOuter().new MyInner();`

- **nie można instancjonować klasy wewnetrznej wewnątrz metod statycznych bez referencji na obiekt klasy zewn.**

Dostep do instancji klasy wewn lub zewn z wnetrza klasy wewnetrznej

- słowo **this** odnosi sie do klasy w której jest
  - wywołując ją wewnątrz klasy wewn. odwołuje się do jej memberów
- klasa wewn. ma dostęp domyślnie do memberów klasy zewn. wiec nie musimy jawnie określac że odwołujemy się do memberów zewnętrznych.
  - można to zrobić za pomocą **MyOuter.*this***
    - np **MyOuter**.***this***.*jakisMember*

Modyfikatory możliwe dla klasy wewnętrznej

- ponieważ kalsa wewnętrzna jest **memberem** klasy zewn. może miec modyfikatory takie jak każdy member:
  - final
  - abstract
  - public
  - private
  - protected
  - static ***— patrz 8.4***
  - strictfp

- klasy wewnetrzne **mogą po sobie dziedziczyć:**

```
class BigOuter {
       abstract class Nest {void go() { System.out.println("hi"); } }  // ale
nie musi być abstract
       class NestChild extends Nest{}
}
```

8.2. Method-local inner classes

- taka klasa wewnętrzna znajduje się wenątrz metod klasy
- instancjonowanie takiej klasy **musi się odbyć wewnątrz metody** oraz **za definicja tej klasy**
  - nie ma możliwości instancionowania takiej klasy poza tą metodą ani poza klasą zewn.

```
class MyOuter2 {
     private String x = "Outer2";
```

```
     void doStuff() {
       class MyInner {
           public void seeOuter() {
             System.out.println("Outer x is " + x);
           }                                          // close inner class method
         }                                            // close inner class
definition
         MyInner mi = new MyInner();  // This line must come after the class
         mi.seeOuter();
       } // close outer class method doStuff()
} // close outer class
```

- taka klasa ma dostęp do memberów klasy zewnetrznej (także prywatnych)
- taka klasa **nie ma dostępu do zmiennych lokalnych metody** w której sie znajduje (*)
  - **chyba że** ta zmienna lokalna jest **final**

(*) bo nie ma pewności czy zmienne lokalne będą żyć tak długo jak obiekt klasy wewn.

- klasa wewnętrzna zdefiniowana w metodzie **statycznej** ma dostęp **tylko do statycznych zmiennych klasy**

- **modyfikatory dla takiej klasy** są takie jak dla każdej zmiennej lokalnej
  - **tj nie może być** public, private, protected, static, transient
  - **może być: abstract** albo **filnal**

8.3. Wewnętrzne klasy anonimowe - anonymous inner classes

- używane jako **argument metody**
- może nie posiadać nazwy

Zwykła klasa anonimowa - podklasa

```
class Popcorn {                    // klasa zwykła
   public void pop() {
        System.out.println("popcorn");
    }
}
class Food {
   Popcorn p = new Popcorn() {          // p jest referencja typu Popcorn,
      public void pop() {                    // ale jest tu tworzona PODKLASA
typu Popcorn,
                                                          // overridująca
metode pop()
          System.out.println("anonymous popcorn");
      }
        public void bad() {}       // nowa metoda bad - nie bedzie mozna jej
nigdy wywołać
   };                 // średnik na końcu MUSI BYC
}
```

- powyżej **stworzona została nowa klasa** (podklasa Popcorn) i przypisana referencji p (**jak w przypadku polimorfizmu**)
- **UWAGA** musi być **średnik** na końcu takiej definicji!!!!
- wywołanie metody pop() na referencji p wywoła **jak w przypadku polmorfizmu** metode podklasy (**anonimowej**)

- wywołanie innej (NOWEJ) metody bad() **nie powiedzie się** ponieważ kompilator zna typ Popcorn w której nie ma takiej metody (patrz polimorfizm)

Zwykła klasa anonimowa - implementator interfejsu

- można także stworzyć (jak powyżej) anonimową klasę będącą implementacją interfejsu

```
interface Cookable {
   public void cook();
}
class Food {
   Cookable c = new Cookable() {   // klasa anonimowa jako implementacja interfejsu
      public void cook() {
         System.out.println("anonymous cookable implementer");
      }
   };                  // średnik na końcu MUSI BYC
}
```

- pomimo że jest słowo **new** przed interfejsem - to nie instancjonuje on interfejsu tylko klase (anonimową)
Runnable r = new Runnable(); **// can't instantiate interface**
Runnable r = new Runnable() { public void run() { } }**;** // **OK, impementacja z klasą anonimową**

- **takie klasy anonimowe moga implementować tylko jeden interfejs**

Klasa anonimowa w argumencie metody

```
class MyWonderfulClass {
  void go() {
    Bar b = new Bar();
    b.doStuff(  new Foo() {   // interfejs - instancja anonimowej klasy w argumencie.
       public void foof() {
         System.out.println("foofy");
       } // end foof method
     }  ); // koniec inner class oraz arggumentu b.doStuff wiec trzeba zamknąć );
   } // end go()
} // end class

interface Foo {
  void foof();
}
class Bar {
```

```
   void doStuff(Foo f) { }
 }
```

- może być oparte na **interfejsie** oraz **klasie**

8.4. Static nested classes

- nie są typowymi klasami wewnetrznymi bo maja ograniczone powiązanie z klasą zewnetrzną
  - związek bardziej "name-space'owy"

- klasa taka **jest jak statyczny member klasy zewnętrznej** (np metoda)
  - można sie więc do niej odwoływać **bez instancji klasy zewnętrznej**
    - **statyczna klasa** wewnętrzna **ma dostęp tylko do statycznych** zmiennych/metod klasy zewnetrznej
      - tak samo jak w przypadku statycznej metody - ona tez ma dostep tylko do statycznych zmiennych/metod swojej klasy

```
class BigOuter {
  static class Nested { void go() { System.out.println("hi"); }  }
}
```

- odwołanie z zewnątrz klasy:

```
BigOuter.Nest n = new BigOuter.Nest(); // both class names
n.go();
```

  - nie ma potrzeby instancjonowania *BigOuter*

- odwołanie z wewnątrz klasy nie różni się od odwołania do innej klasy wewnetrznej

```
Nest n2 = new Nest(); // access the enclosed class
n2.go();
```

- taka klasa **może mieć jeden z czterech modyfikatorów dostępu:** public, private, protected, *default*
- taka klasa może tez byc **abstract** lub **final**

- klasy wewnetrzne mogą po sobie dziedziczyć:

```
class BigOuter {
       static abstract class Nest {void go() { System.out.println("hi"); } }
       static class NestChild extends Nest{}
}
```

9. Wątki - Threads

- Wątek:
  - *java.lang.***Thread** - obiekt reprezentujący wątki

- thread (watek) - osobny *call stack* i osobny "proces" przetwarzania (1 callstack na wątek i 1 wątek na callstack)

- implementacja wątków w JVM opiera się na natywnych watkach systemu operacyjnego
  - mapowanie na procesy systemowe
- przetwarzanie wątków jest róznie zaimplementowane na różnych maszynach JVM
  - część zachowań **jest zapewniona a część nie** (działa inaczej)
    - na egzamin jest wymagane odrózniac co jest stałe a co nie.
  - nie można uzależniac swego kodu od konkretnej JVM

- **na egzaminie są tylko *"user threads"*** i nie ma *"daemon threads"*
  - *róznica polega na tym że JVM czeka tylko na wykonanie wszystkich wątków "user" zanim zakończy działanie programu.*
    - *daemon threads nie musza sie skonczyc*

9.1. Tworzenie i uruchamianie wątków

9.1.1. Tworzenie wątku

- jest tworzona **instancja klasy Thread**
  - podstawowe metody klasy *Thread*
    - `start()`
    - `yield()`
    - `sleep()`
    - `run()` - w tej metodzie wykonuje sie **kod danego wątku**

- tworzenie wątku następuje na dwa sposoby:

i. stworzenie **klasy dziedziczącej po Thread**
   - *łatwiejszy ale mniej polecany sposób - zła praktyka OO*
     - dlaczego?: bo dziedziczenie z założenia jest tworzeniem bardziej specjalizowanej klasy Thread

ii. **zaimplementowanie interfejsu *Runnable***
   - *bardziej zalecane*

- oba powyższe sposoby sprowadzają sie do utworzenia metody **run()**
- metoda **run()** będzie wywołana przez wątek

Dziedziczenie po Thread

```
class MyThread extends Thread {
  public void run() {
     System.out.println("Important job running in MyThread");
   }
 }
```

- wadą tego sposobu jest to **że nie można odziedziczyc niczego więcej**

- **można** przeładować metodę run() (np *run(String s)*) ale **taka metoda będzie ignorowana przez watek**
  - pod uwage brane jest tylko **run()** // (najwyżej mozna ja wywołac samemu)

Implementacja interfejsu Runnable

```
class MyRunnable implements Runnable {
   public void run() {
     System.out.println("Important job running in MyRunnable");
   }
}
```

- nadal można dziedziczyć poinnej klasie i implementowac inne interfejsy

9.1.2. Instancjonowanie wątku

- każde rozpoczęcie wykonania wątku musi być poprzedzone zainicjalizowaniem klasy Thread

- w przypadku klasy **dziedziczącej po Thread:**

  MyThread t = new MyThread()

  - w tym przypadku wątek **uruchomi metodę run() zaimplementowaną w MyThread**

- w przypadku klasy **implementującej Runnable**

  MyRunnable **r** = new **MyRunnable**(); // 1. Instancjonowanie klasy implementujacej Runnable
  Thread t = new **Thread**(**r**); // 2. Stworzenie instancji klasy Thread z referencją do *naszej klasy* jako parametrem

  - wątek **uruchomi metode run() w klasie podanej w argumencie konstruktora**
  - można utuchamiać naszą metodę run() **w wielu wątkach**:

```
Thread foo = new Thread(r);
   Thread bar = new Thread(r);
   Thread bat = new Thread(r);
```

// ta sama funkcja będzie uruchamiana w 3 różnych wątkach

- ponieważ Thread także implementuje Runnable więc do argumentu Thread można także podać **obiekt dziedziczący po Thread:**

  Thread t = new Thread(**new MyThread()**); // jest to legalne ale niezalecane - bo nadmiarowo tworzymy 2. obiekt

- powyższe metody **nie startuja wykonania wątku a jedynie powołują obiekt Thread**
  - wątek nie posiada jescze stanu ***alive*** *(patrz **metoda getState()** - nie wymagana*

na egzamin)
- run() nie jest jeszcze wykonana
- wystartowanie wątku: ***patrz 9.3***
- w tym miejscu wątek **ma status "new"**

- klasa **Thread posiada więcej konstruktorów**:

```
Thread()
Thread(Runnable target)                 // omówiono powyżej
Thread(Runnable target, String name)
Thread(String name)
```

9.1.3. Uruchamianie wątku

- startowanie wątku odbywa się poprzez metodę **start()** wykonana **na obiekcie Thread** lub **odziedziczonym po Thread**

i. *MyThread t = new MyThread()* lub ii. *Thread t = new Thread(new MyRunnable())*

t.**start();**

- po wywołaniu start() dokonuje się:
- nowy watek jest rozpoczynany **(nowy *call stack*) -- patrz Figure 9-1, str 684**
. np wywołanie metody w main spowoduje położenie main na call stacku a nad nią metody w tym samym callstacku.
wątek otwiera nowy call stack, obok.
- wątek zmienia status z "new" na "runnable"
- gdy wątek otrzymuje możliwość - uruchamia metodę **run()**

Przykład:

```
class FooRunnable implements Runnable {
   public void run() {
      for(int x =1; x < 6; x++) {
        System.out.println("Runnable running");
      }
   }
}
public class TestThreads {
   public static void main (String [] args) {
     FooRunnable r = new FooRunnable();
     Thread t = new Thread(r);
     t.start();
   }
}
```

- **Uwaga** - wykonanie metody **run() ręcznie jest legalne** ale **nie tworzy nowego wątku** a jedynie wykonuje te metodę

MyRunnable r = new MyRunnable();
r.run(); // **Legal, but does not start a separate thread**

- dopiero kroki z punktu 9.2 i 9.3 wystartują metodę run w nowym wątku

Uruchamianie wielu wątków

- można nadawać wątkom nazwy

```
Thread t = new Thread(nr);
  t.setName("Fred");     // nadanie nazwy wątkowi
  t.start();
```

- pobieranie nazwy aktualnego wątku wewnątrz tego wątku:
  `Thread.currentThread().getName()` `// metoda zwraca String z nazwą nadaną wcześniej`

- bez podania nazwy wątek posiada nazwę domyślną
  - metoda main() - nazwa: main
  - inny wątek: Thread-0, Thread-1 ...

- metoda statyczna `Thread.currentThread()` jest wywoływana wewnątrz metody **run()**
  - ponieważ run() nie posiada referencji do wątku używa się metody statycznej

```
Thread one = new Thread(nr);
Thread two = new Thread(nr);
Thread three = new Thread(nr);

one.setName("Fred");
two.setName("Lucy");
three.setName("Ricky");

one.start();
two.start();
three.start();
```

- start powyższych wątków spowoduje że każdy wątek się rozpocznie jeden po drugim ale ich działanie jest od siebie **niezależne** a zakończyć się mogą różnie (niekoniecznie w tej samej kolejności).
  - np wypisywanie coś na konsoli w każdym z wątków może dać różne wyniki dla różnych wywołań i JVM.
  - kolejność wykonania działań **w różnych wątkach nie gwarantuje kolejności działań**
  - oczywiście **wewnątrz pojedynczego wątku działania są przewidywalne**, zgodne z kodem
    - patrz przykład na str 686
  - jest to **kontrolowane przez SCHEDULER na który programista nie ma wpływu**

- metoda ***start()*** wątku **może byc wywołana tylko JEDEN RAZ**

- można wywoływac inne metody klasy Thread (lub dziedziczącej) jak na zwykłym obiekcie - ale start tylko raz
- wywołanie start ponownie **wyrzuci wyjątek (RuntimeException) IllegalThreadStateException**
- wątki o statusie *runnable* (gotowy do uruchomienia) i *dead* (zakończony) nie moga być restartowane

9.1.3.1. Thread Scheduler

- jest częścią JVM
- scheduler wybiera wątki w tanie *runnable* ale **kolejność wyboru nie jest gwarantowana**
- wątki są kolejkowane przez scheduler
    - wykonanie jest podzielone na tury, zakonczenie tury przez wątek spycha go na koniec kolejki (poola)
- kolejnośc w kolejce (a właścwie pool'a) nie jest gwarantowana

- **nie mamy kontroli nad schedulerem ale można na niego "lekko wpływać"**

<u>Metody wpływające na wątek</u>

java.lang.Thread

public static void **sleep**(long millis) throws InterruptedException
- posiada tez przeciązone wersje
public static void **yield**()
public final void **join**() throws InterruptedException
- posiada tez przeciązone wersje
public final void **setPriority**(int newPriority)

java.lang.Object

public final void **wait**() throws InterruptedException
- posiada tez przeciązone wersje
public final void **notify**()
public final void **notifyAll**()

9.2. Stany i przejścia wątków

9.2.1. Stany

- ***new*** od momentu powołania do życia instancji Thread ale <u>zanim wywołana zostanie metoda start</u>() (nie jest *alive*)
- ***runnable*** gdy wątek jest gotowy ro uruchomienia (jest *alive*)
    - zaraz po wywołaniu **start()** ale scheduler także może powrócić do stanu *runnable*
- ***running*** gdy jest uruchomiony, po tym jak **scheduler wybrał ten proces do bycia aktualnie wykonywanym**
- ***waiting/blocked/sleeping*** gdy proces czeka na zakończenei innego procecu lub czeka

na zasób albo jest uśpiony.
- metody t.**sleep()** lub t.**yield()** są metodami **statycznymi** i **zawsze odwołują sie do bierzącego procesu**
- pomimo że wygląda to na wywołanie na referencji t to jest to metody **static** więc można to zamienić na Thread.sleep()
- ***dead*** gdy metoda **run()** dobiegła końca, **nie można ponownie wywołąc na nim *start()***, (nie jest *alive*)

9.2.2. Sleeping

- metoda **statyczna** ***sleep()*** zmienia status wątku na *sleeping* - czyli spowalnia wątek

Thread.***sleep***(**long millis**) throws **InterruptedException**
Thread.***sleep***(**long millis**, ***int nanos***) throws **InterruptedException**

- **Uwaga na egzaminie** często jest zmyłka i wywoływana jest sleep() na referencji co oznacza to samo co na Thread.sleep()

- w stanie sleeping wątek nic nei robi zanim nie zostanie zbudzony - wówczas przechodzi w stan *runnable*
- rzuca **wyjatek (checked exception) InterruptedException** więc trzeba go obsłużyć (try lub throws)

- usypianie wątku może byc sposobem aby pozostałe wątki mogły wystartować
- **ale nadal nie jest to przewidywalne**

- wątek natrafiając na metodę sleep() **musi ją wykonac** na conajmniej czas podany w argumencie
- ponieważ wątek przechodzi w stan ***runnable*** a nie *running* oznacza to że nie jest pewne że po czasie podanym w arg wątek będzie działał
- po wybudzeniu scheduler może go wziąć w running od razu albo nie.

- gdy wątek zostanie przebudzony wczesniej niż czas podany w argumencie zostanie wyrzucony wyjątek **InterruptedException**

9.2.3. Priorytetowanie i yield()

Priorytety

- na podstawie priorytetów scheduler (JVM) wybiera jaki proces ma byc w tym momencie *running*
- jesli proces ma wyzszy priorytet od tych w pool'u oraz tego co działa, to on staje sie *running* a ten działający cofany jest na *runnable*
- a więc działający proces ma zawsze najwyższy priorytet od tych w poolu (gdy w pool pojawi sie wyzszy od razu staje sie *running*)
- lub równy najwyższemu

- pomimo że priorytety mogą poprawic efektywnośc aplikacji wielowątkowej to **nie można na tym polegać w 100%**

- **nie gwarantowane** jest gdy wątki w poolu maja taki sam priorytet jak ten działający
- **gwarantowane** jest to że najwyższy priorytet (gdy np jest jedyny) zadziała pierwszy

- JVM zaleznie od implementacji moze zachowywac sie roznie
  - dzielenie zcasu procesora na wątki jednoczesnie
  - lub wybiera wątek i czeka na jego koniec

<u>Nadawanie priorytetów</u>

- **domyślnie** wątek otrzymuje **priorytet taki sam jak wątek go wywołujący**
  - przeważnie jest to **'5'**

- można **ustawić priorytet ręcznie - setPriority(int)**

```
FooRunnable r = new FooRunnable();
Thread t = new Thread(r);
t.setPriority(8);
t.start();
```

- priorytety to liczby całkowite **(int) <u>zazwyczaj</u> od 1 do 10**
- numery te **nie są gwarantowane**

- można sterować wielkością domyślnych wartości:

static final:

```
Thread.MIN_PRIORITY (1)    minimalny
Thread.NORM_PRIORITY (5)   domyślny
Thread.MAX_PRIORITY (10)   maksymalny
```

***<u>Metoda yield</u>*()**

oddanie procesora innemu wątkowi o tym samym priorytecie (zwalnia cpu aby inny wątek mogl go uzyc)

<u>Metody Object.wait(), Object.notify() i Object notifyAll()</u>

metodę notify i notifyAll oraz wait można wywołać **TYLKO w bloku/metodzie SYNCHRONIZED** (inaczej bedzie RuntimeExc) i wołać je wolno tylko z obiektów **na których odbywa się synchronizacja.**

wait na obiekcie zaklada mu monitor. notify z innego wątku na tym samym obiekcie zdejmuje monitor co powiadamia wątki ktore wykonaly wait.
Wykorzystywane to moze byc w apliakcjach typu "producer" i "customer".

[***notify and notifyAll*** *are instance (non static) methods of the* ***Object*** *class. The notify() method wakes up a single thread that is waiting on this object's monitor. If any threads are waiting on this object, one of them is chosen to be awakened. It happens at the discretion of the thread scheduler and cannot be influenced.]*

Metoda join()

```
T1{
  T2.start();
  T2.join();
}
```

powoduje że wątek T1 który uruchomił inny wątek T2 będzie czekał na jego ukonczenie (w miejscu join) i pojdzie dalej po zakonczeniu T2

Dokończyć WĄTKI s 698-729

str 698

10. Development

10.1. Komendy *java* i *javac*

10.1.1. Kompilacja *javac*

javac to polecenie wywołujące **kompilowanie** kodu

Postać polecenie to:

javac [options] [source files]

opcje i pliki z kodem oddzielamy przecinkami.

javac -help wyświetla następującą strone pomocy (*pogrubione wymagane na egzaminie)*:

```
possible options include:
  -g                        Generate all debugging info
  -g:none                   Generate no debugging info
  -g:{lines,vars,source}    Generate only some debugging info
  -nowarn                   Generate no warnings
  -verbose                  Output messages about what the compiler is doing
  -deprecation              Output source locations where deprecated APIs are used
  -classpath <path>         Specify where to find user class files and annotation
processors
  -cp <path>                Specify where to find user class files and annotation
processors
  -sourcepath <path>        Specify where to find input source files
  -bootclasspath <path>     Override location of bootstrap class files
  -extdirs <dirs>           Override location of installed extensions
  -endorseddirs <dirs>      Override location of endorsed standards path
  -proc:{none,only}   Control whether annotation processing and/or compilation
is done.
  -processor <class1>[,<class2>,<class3>...] Names of the annotation processors
to run;
                                                                    bypasses
default discovery process
  -processorpath <path>     Specify where to find annotation processors
  -d <directory>            Specify where to place generated class files
  -s <directory>            Specify where to place generated source files
  -implicit:{none,class}    Specify whether or not to generate class files for
                                        implicitly referenced files
  -encoding <encoding>      Specify character encoding used by source files
  -source <release>         Provide source compatibility with specified release
  -target <release>         Generate class files for specific VM version
  -version                  Version information
  -help                     Print a synopsis of standard options
  -Akey[=value]             Options to pass to annotation processors
  -X                        Print a synopsis of nonstandard options
  -J<flag>                  Pass <flag> directly to the runtime system
```

10.1.2. Kompilacja z parametem -d

- Parametr **-d** określa **miejsce zapisu skompilowanych klas**

np: w katalogu *project* mamy dwa katalogi: src i classes. aby skompilować klasę z src do classes, z poziomu katalogu *project* podajemy:

*javac **-d classes src**/MyClass.java*

- parametr -d **tworzy strukture katalogów dla skonpilowanych klas** wg ich pakietów,tzn podajemy tylko katalog ./classes a kompilator stworzy dla klasy strukturę katalogów np *./classes***/com/example/scjp/MyClass.class**
dla wywołania: javac -d classes src**/com/example/scjp**/MyClass.java gdy MyClass jest w pakiecie com.example.scjp

- Jeżeli katalog podany w -d **nie istnieje to zostanie zgłoszony błąd kompilatora**:

java:5: error while writing MyClass: classes/MyClass.class (No such file or directory)

10.2. Uruchamianie aplikacji

10.2.1. Polecenie *java*

- Polecenie ***java*** uruchamia wirtualną maszynę i aplikację.
Postać polecenie:

java [options]* class *[args]

- options i args są opcjonalne
  - **options** to opcje uruchomieniowe **dla wirtualnej maszyny**
  - **args** to **argumenty dla aplikacji**
- podajemy **co najmniej jedną** skompilowaną klasę **bez rozszeżenia .class**

java *-DmyProp=myValue* **MyClass** *x 1*
-- spowoduje utworzenie propertiesa myProp i uruchomienie MyClass z argumentami x i 1

10.2.2. Użycie propertiesów

- mechanizm java.util.Properties umozliwia uzyskanie informacji takich jak wersja systemu, wersja jvm oraz przekazywania prpertiesów przy starcie
- „wstrzykniecie propertiesa” podczas uruchamiania odbywa się prze parametr -D

java -**D**myProp=myVal Program // gdzie myProp to nazwa propert a myVal to wartość.
//
wartość musi być podana bez spacji lub w cudzysłowach „”
//
między -D i nazwą **nie może być spacji**

- innym sposobem dodania propertiesa jest dodanie w kodzie:
  Properties p = System.getProperties();
  p.**setProperty("myProp", "myValue");**
- pobranie propertiesa w kodzie:
  p.**getProperty(String nazwa)** // p.getProperty(String nazwa, String defaulVal)

10.2.3. Użycie parametrów uruchomieniowych programu

- argumenty przekazujemy za pomocą:

java Program **param1 param2**

- są przypisywane do tablicy args w metodzie main:

```
public static void main(String[] args)   // args - nazwa oczywiście dowolna
public static void main(String... args)
```

- tablica args jest ma wielkosc równą ilości argumentów
- pierwszy argument jest pod args[0] itd...

10.3. "przeszukiwanie" klas - Classpath

- szukanie klas dotyczy zarówno **kompilacji javac** jak i **uruchomienia java**
- algorytm szukania dla java i javac jest taki sam:

- szukają w tym samym miejscu
- szukają w tej samej kolejności
- gdy znajdą, nie przeszukują dalej
- pierwszym miejscem szukania są klasy JSE (*pathToJava/jre/lib/ext*), następnie katalogi w classpath
- classpath można zadeklarować w **zmiennych srodowiskowych systemu** lub **w command-line podczas wywołania**

- zmienna JAVA_HOME jest uzywana aby skrócić długie classpath

10.3.1. Classpath - deklaracja i użycie

- classpath określa katalogi z klasami
- w sytemach unix separator katalogów to / a **katalogi oddzielone dwukropkiem :**
- w windowsie separator katalogów to \ a **katalogi oddzielone średnikiem** ;
- **egzamin: notacja unixowa**

- przykład jako parametr command-line'a
-classpath /com/foo/acct:/com/foo
-cp /com/foo/acct:/com/foo // skrót -cp **nie musi działać** na wszystkich JVM **!!!**

- deklarując katalog **nie jest uwzględniany jego katalog nadrzędny**
- kolejność katalogów ma znaczenie i jest przeszukiwana **od lewej do prawej**
- kropka oznacza katalog bierzący (gdy nie podano w **javac domyślnie jest dodawany katalog bierzacy** gdy *javac Klasa.java*)

10.3.2. Classpath przy PAKIETACH

- nazwa klasy wraz z pakietem jest nazwą ATOMOWĄ w znaczeniu classpatha:
  com.example.mypackage.Klasa

- klasa wraz z pakietem musi odpowiadać strukturze katalogów: com.example.package = com/example/package
- classpath musi zawierać ścieżkę **absolutna lub relatywną** względem katalogu biezacego
  - przy ścieżkach relatywnych trzeba uważać na to jaki jest bieżący katalog!

10.4. Pliki JAR

- pliki **JAR** to **archiwa ze skompresowanymi klasami** wraz z odpowiednią **strukturą katalogów**

- **tworzenie** JARa:
  z katalogu *nad* strukturą pakietów (katalog myApp zawiera strukturę z klasami):
  ```
  jar -cf MyJar.jar myApp
  ```

- **listowanie** JARa:
  ```
  jar -tf MyJar.jar
  ```

10.4.1.Przeszukiwanie klas w JARach

- podobne do przeszukiwania w classpath **ale należy dodać nazwę jara** wraz ze ścieżką:

```
javac -classpath myPath/myApp.jar Program.java
```

10.5. Importy statyczne

str 776

eof